Automata Theoretic Account of Proof Search

Aleksy Schubert*[1], Wil Dekkers[2], and Henk P. Barendregt[2]

1 **University of Warsaw, ul. Banacha 2, 02-097 Warsaw, Poland**
`alx@mimuw.edu.pl`
2 **Radboud University, Nijmegen, Faculty of Science, Postbus 9010, 6500 GL Nijmegen, The Netherlands**

Abstract

Techniques from automata theory are developed that handle search for inhabitants in the simply typed lambda calculus. The resulting method for inhabitant search, which can be viewed as proof search by the Curry-Howard isomorphism, is proven to be adequate by a reduction of the inhabitant existence problem to the emptiness problem for appropriately defined automata. To strengthen the claim, it is demonstrated that the latter has the same complexity as the former. We also discuss the basic closure properties of the automata.

1998 ACM Subject Classification E.1 Data structures, F.1.1 Models of computation, F.4.1 Mathematical logic, F.4.3 Formal languages

Keywords and phrases simple types, automata, trees, languages of proofs

Digital Object Identifier 10.4230/LIPIcs.CSL.2015.128

1 Introduction

A recent book on typed lambda calculi [1] contains a considerable number of graphical representations for constructing inhabitants of given types (or, by the Curry-Howard isomorphism, proofs for propositional intuitionistic logic). These follow the mechanics of the Ben-Yelles-Wajsberg algorithm [3, 9, 21], explain intuitively its operations, and materialise its mechanics for particular inputs. In this way, the runs of the algorithm become a particular compact data structure that can in itself, when defined formally, be subject to further computations, as finite automata are in automata theory.

However, it is not immediate how to turn the intuitive pictures into a formal notion. First of all, finite automata work on a finite alphabet, while λ-terms can contain bound variables from an infinite set. One approach available here is to restrict the language to a fixed set of first-order constants so that there is no need to introduce binders. This method was used in the work of Düdder et al. [6], where automata in this fashion were proposed for synthesis of programs in a simple functional language. Another approach would be to restrict the inhabitant search to a limited subset of the normal terms, that is sufficiently well distributed so that existence of a type inhabitant implies it is possible to find one of this form. Typically, *total discharge terms* are used for this purpose. In terms of this form, only one bound variable is needed for each subtype of the original type. Based upon this idea Takahashi et al [18] developed a context-free grammar approach to inhabitant search (which can be viewed as inhabitant search using tree automata due to known correspondence between grammars and tree automata).

* This work was partly supported by NCN grant DEC-2012/07/B/ST6/01532.

24th EACSL Annual Conference on Computer Science Logic (CSL 2015).
Editor: Stephan Kreutzer; pp. 128–143

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

These approaches have as natural limitation that they do not make it possible to recognise the collection of all inhabitants. For this a method is needed to deal with the infinite alphabet in the language. Automata that work with infinite alphabets were proposed by Kaminski and Francez [11] for strings and by Kaminski and Tan [12] for trees. These automata have, in addition to the standard control arranged through states, a fixed set of registers where data from an infinite set may be stored. Data elements stored in registers can be checked for equality with data elements from the input. This restricted check operation on an infinite domain makes it possible to work with such automata similarly to the usage of standard finite automata. Still, these automata do not fit well with the type inhabitation problem, as storing a fixed finite number of bound variable names is not sufficient to represent all potential normal inhabitants. To overcome this limitation we propose here a different notion of register, such that *a set* of data elements may be kept there and the check operation verifies whether a data element from the input *belongs* to the set. It turns out that this method recognises all trees that can be reasonably regarded as inhabitants of a particular type. Moreover, we show how it relates to earlier approaches, in which the *total discharge forms* are recognised.

Related work. Various kinds of finite automata have been proposed for dealing with semantics of the simply typed λ-calculus. The work of Salvati and Walukiewicz expresses the semantics through Krivine machines [15]. Another approach expresses semantics by description of β-reduction in the context of the higher-order matching problem (Ong and Tzevelekos [13], Stirling [17]). Another interesting related work goes in a different direction. Broda and Damas [4] proposed the formula-tree proof method, which partially realises the program of the current work, and concretisises the proof search procedure as a data structure. We believe that the automata theoretic view proposed here has the additional benefit of bridging proof theory with the rich theory of automata, enabling mutual influence.

Organisation of the paper. We fix the notation in Section 2. Next, we define our inhabitation machines in Section 3. This is continued by demonstration of the PSPACE-completeness of the emptiness problem for the machines in Section 4. We summarise the account in Section 5 by giving conclusions and showing the potential for further work.

2 Preliminaries

To make this paper self-contained we introduce some basic notation. In the automata theoretic setting it is convenient to use the notion of a *signature*, usually denoted by Σ, that is an indexed family of sets that contain elements called *symbols*. We sometimes abuse the notation by identifying Σ with $\bigcup \Sigma$ and write e.g. $a \in \Sigma$ for some symbol a in one of the members of Σ. The indices of the family are natural numbers and are called *arities*. The arity of a symbol f is written $\mathsf{arity}(f)$. For a natural number k we write $\overline{k}$ for the set $\{0, \ldots, k-1\}$. The set of all subsets of a given set A is written $P(A)$, and for the set of all finite subsets of A we write $P_{\mathsf{fin}}(A)$. Concatenation of two sequences π, π' of elements from some set A is written $\pi \cdot \pi'$. A special case here is when π' is a single symbol $i \in A$, then the concatenation is $\pi \cdot i$. The prefix order on sequences of natural numbers is written $\preceq$. A set C of finite sequences over $\mathbb{N}$ that is closed on the prefixes can be used as a domain of a tree. A *labelled tree* over L is a function $t : C \to L$ where L is called the *set of labels*. We write $\mathsf{dom}(t)$ for C. Elements of $\mathsf{dom}(t)$ are called *nodes in the tree* t. We write $t|_\pi$ for the subtree rooted at the node $\pi \in \mathsf{dom}(C)$, i.e. the tree with the domain $C' = \{\pi' \mid \pi \cdot \pi' \in \mathsf{dom}(t)\}$

and labelling t' defined as $t'(\pi') = t(\pi \cdot \pi')$ being t restricted to C'. Usually, the set of labels is a signature (flattened to $\bigcup \Sigma$) and then we assume that the tree respects the arity, i.e. if $\mathsf{arity}(t(\pi)) = n$ then $\pi \cdot i \in \mathsf{dom}(t)$ for $i \in \overline{n}$. For a function $f : A \to B$ we define its update $f[a \mapsto b]$ for $a \in A$ and $b \in B$ as $f[a \mapsto b](x) = f(x)$ for $x \neq a$ and $f[a \mapsto b](a) = b$.

3 Automata account of the inhabitation problem

In what follows we use a slightly modified exposition from [1]. The simply typed λ-calculus $\lambda_{\to}$ in the Church style is a language of expressions that have the following syntax expressed in *simplified syntax BNF*:

$$\begin{array}{lll} \mathbb{T} \ni & \tau & ::= \alpha \mid \tau_0 \to \tau_1 \\ \Lambda_{\to} \ni & M & ::= x^{\tau} \mid M_0 M_1 \mid \lambda x^{\tau}.M_0 \end{array}$$

This means that the parentheses are left implicit in the grammar above. The elements of $\mathbb{T}$ and $\Lambda_{\to}$ are called *types* and *terms*, respectively. We assume here that α are *type atoms* that are from an infinite, countable set A. We use metavariables σ, τ etc. for types. Term variables, noted x, y, F etc. are from an infinite countable set V. Compound expressions of the form x^{τ} are called *typed term variables* and the set that contains all of them is $\mathsf{V}^{\Lambda}_{\to}$. As usual we distinguish the set of free variables $\mathsf{FV}(M)$ and define it structurally over terms so that the binding operation is λ, and x^{τ} is bound in $\lambda x^{\tau}.M_0$. A term that has no occurrences of free variables is called *closed*. The λ-terms are identified up to α-conversion that makes it possible to rename bound variables. A *context*, usually written as Γ with possible ornaments, is a finite set of typed term variables.

We follow here a slightly non-standard take on contexts since we make it possible for a context to contain both x^{τ} and $x^{\tau'}$ for $\tau \neq \tau'$. Observe that this solution is not essential since the type makes the variables to be sufficiently distinct. One must only ensure that when a type erasure operation is performed, such two variables are made distinct, which can be done in different ways, e.g. by making the type a part of the variable name.

Terms of type τ in the context Γ, written $\Lambda^{\Gamma}_{\to}(\tau)$, are a family of sets defined as the smallest family that satisfies the conditions:

- $x^{\tau} \in \Lambda^{\Gamma}_{\to}(\tau)$ when $x^{\tau} \in \Gamma$,
- if $M_0 \in \Lambda^{\Gamma}_{\to}(\sigma \to \tau)$ and $M_1 \in \Lambda^{\Gamma}_{\to}(\sigma)$ then $M_0 M_1 \in \Lambda^{\Gamma}_{\to}(\tau)$,
- if $M_0 \in \Lambda^{\Gamma \cup \{x^{\sigma}\}}_{\to}(\sigma')$ then $\lambda x^{\sigma}.M_0 \in \Lambda^{\Gamma}_{\to}(\sigma \to \sigma')$ where $\sigma \to \sigma' = \tau$.

We often abbreviate $\Lambda^{\emptyset}_{\to}(\tau)$ as $\Lambda_{\to}(\tau)$.

Proof search procedures usually look for proof terms in normal form, i.e. ones that do not use a form of the *cut* rule. In the context of λ-calculi, the *cut* operation is represented as a beta redex. In case of $\lambda_{\to}$ in the Church style, this redex is

$$(\lambda x^{\tau}.M_0)M_1 \to_{\beta} M_0[x^{\tau} := M_1]$$

where $M_0[x^{\tau} := M_1]$ is understood as the term that results from M_0 by replacing all occurrences of the typed variable x^{τ} with M_1. This substitution, as usual, renames bound variables in M_0 so that no free variable in M_1 is captured by binding λ operators in M_0. The relation $\to_{\beta}$ is defined by syntax closure of the above mentioned redexes. The reflexive-transitive closure of $\to_{\beta}$ is written in $\to^*_{\beta}$. It is known that the relation $\to_{\beta}$ is strongly normalising, i.e. each sequence of terms $M_0, M_1, \ldots$, such that $M_i \to_{\beta} M_{i+1}$, has a finite number of elements ([1, Theorem 2.2.1]).

It is easy to see that *normal terms of type τ in the context* Γ, written $\Lambda^{\Gamma}_{n,\to}(\tau)$, are a family of sets defined as the smallest family that satisfies the conditions stated below. This definition uses a supplementary set $\Lambda^{\Gamma}_{s,\to}(\tau)$ (the letter 's' stands for 'spine' here).

- if $x^\tau \in \Gamma$ then $x^\tau \in \Lambda^\Gamma_{n,\to}(\tau)$ and $x^\tau \in \Lambda^\Gamma_{s,\to}(\tau)$,
- if $M_0 \in \Lambda^\Gamma_{s,\to}(\sigma \to \tau)$ and $M_1 \in \Lambda^\Gamma_{n,\to}(\sigma)$ then $M_0M_1 \in \Lambda^\Gamma_{s,\to}(\tau)$ and $M_0M_1 \in \Lambda^\Gamma_{n,\to}(\tau)$,
- if $M_0 \in \Lambda^{\Gamma\cup\{x^\sigma\}}_{n,\to}(\sigma')$ then $\lambda x^\sigma.M_0 \in \Lambda^\Gamma_{n,\to}(\sigma \to \sigma')$ where $\sigma \to \sigma' = \tau$.

A standard inductive argument shows the following proposition.

▶ **Proposition 1.** *If N is a subterm of $M \in \Lambda^\Gamma_{n,\to}(\tau)$ then $N \in \Lambda^{\Gamma'}_{n,\to}(\sigma)$ where $\Gamma \subseteq \Gamma'$ and all types in Γ' and σ are either subexpressions of τ or subexpressions of types in Γ.*

Note that the context Γ' may contain variables that do not occur in N.

The proof search when considered in the field of λ-calculi turns out to be, due to the Curry-Howard isomorphism, the search of inhabitants for types. Here is the precise formulation of the inhabitation problem.

▶ **Definition 2** (inhabitation problem)**.** The inhabitation problem for $\lambda_\to$ (or the decision problem for implicational fragment of propositional intuitionistic logic) is defined as follows
Input: A type τ.
Question: Is there a closed Church-style term M such that M has type τ?

▶ **Example 3.** Consider types $\mathbf{1} = 0 \to 0$, $\mathbf{2} = \mathbf{1} \to 0$ and $\mathbf{3} = \mathbf{2} \to 0$. A normal inhabitant of the type must have the form $\lambda F^{\mathbf{2}}.M_a$ where M_a is of type 0 (we use here the variable F instead of x to underline that it represents a functional). Next the only option we have is to use the typed variable $F^{\mathbf{2}}$, so $M_a = F^{\mathbf{2}}M_b$ where M_b is of type $\mathbf{1}$. Subsequently, we cannot use $F^{\mathbf{2}}$ so M_b must start with λ. Thus $M_b = \lambda y^0.M_c$ where M_c must be of type 0. We can now complete the process and let $M_c = y^0$, but we can continue the process by steps similar to the ones we used for M_a and obtain a sequence of terms

$$\lambda F^{\mathbf{2}}.F^{\mathbf{2}}(\lambda y^0.y^0), \quad \lambda F^{\mathbf{2}}.F^{\mathbf{2}}(\lambda y^0.F^{\mathbf{2}}(\lambda y^0_1.y^0_1)), \quad \lambda F^{\mathbf{2}}.F^{\mathbf{2}}(\lambda y^0.F^{\mathbf{2}}(\lambda y^0_1.F^{\mathbf{2}}(\lambda y^0_2.y^0_2))), \ldots \quad (1)$$

Note that this sequence does not exhaust the whole set of inhabitants of $\mathbf{3}$ since only the last variable of form y^0_i is used here while we have the liberty to use any of them.

In the following a special kind of terms called *total discharge terms* (or terms in Prawitz natural deduction style) [14, 19] is used as a technical device that helps in effective search for witnesses for non-emptiness. Suppose that we have an injection $\phi : \mathbb{T} \to \mathsf{V}$. Let us represent $\phi(\tau)$ as x_τ. We can now define a set of terms $\Lambda^{\mathsf{cnst}}_\to$ as the smallest subset of $\Lambda_\to$ such that

- all x^τ_τ belong to $\Lambda^{\mathsf{cnst}}_\to$,
- if M_0, M_1 belong to $\Lambda^{\mathsf{cnst}}_\to$ then M_0M_1 does,
- if M_0 belongs to $\Lambda^{\mathsf{cnst}}_\to$ and x^τ_τ is a typed variable then $\lambda x^\tau_\tau.M_0$ does.

The following proposition holds for $\lambda_\to$.

▶ **Proposition 4.** *For each context Γ if $\Lambda^\Gamma_\to(\tau) \neq \emptyset$ then $\Lambda^{\lfloor\Gamma\rfloor}_\to(\tau) \cap \Lambda^{\mathsf{cnst}}_\to \neq \emptyset$, where $\lfloor\Gamma\rfloor = \{x^\tau_\tau \mid y^\tau \in \Gamma\}$.*

Proof. Assume that some $M \in \Lambda^\Gamma_\to(\tau)$. We can now show by induction on the structure of M that there is a term $M' \in \Lambda^{\mathsf{cnst}}_\to$ that belongs to $\Lambda^{\lfloor\Gamma\rfloor}_\to(\tau)$. Details are left to the reader. ◀

▶ **Example 5.** The sequence of terms in (1) corresponds to the following sequence of terms in total discharge form.

$$\lambda F^{\mathbf{2}}.F^{\mathbf{2}}(\lambda y^0.y^0), \quad \lambda F^{\mathbf{2}}.F^{\mathbf{2}}(\lambda y^0.F^{\mathbf{2}}(\lambda y^0.y^0)), \quad \lambda F^{\mathbf{2}}.F^{\mathbf{2}}(\lambda y^0.F^{\mathbf{2}}(\lambda y^0.F^{\mathbf{2}}(\lambda y^0.y^0))), \ldots$$

Note that we use here y^0 only instead multiple $y^0_1, y^0_2 \ldots$ since in total discharge form only one variable for a type is allowed.

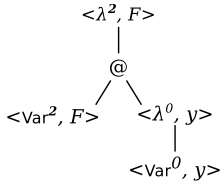

Figure 1 The tree representing the term $\lambda F^{\mathbf{2}}.F^{\mathbf{2}}(\lambda y^0.y^0)$.

3.1 Terms as trees

We identify terms in the Church style with trees in the following way. Let $\mathbb{T}^\sigma$ be the set of all subexpressions of σ. We define Σ_T^σ as the family with symbols $\{\mathsf{Var}^\tau \mid \tau \in \mathbb{T}^\sigma\}$ of arity 0, symbols $\{\lambda^\tau \mid \tau \in \mathbb{T}^\sigma\}$ of arity 1 and the symbol @ of arity 2. For a term $M \in \Lambda^\Gamma_{n,\to}(\tau)$ the tree t^M it corresponds to is defined inductively as follows:

- for $M = x^\tau$ it is a tree with a single node labelled with $\langle \mathsf{Var}^\tau, x\rangle$ and we write the tree as $\langle \mathsf{Var}^\tau, x\rangle$,
- for $M = M_0 M_1$ it is a tree t such that $t^M|_i = t^{M_i}$, for $i = 0, 1$ and $t(\varepsilon) = @$, we write the tree as $@(t^{M_0}, t^{M_1})$,
- for $M = \lambda x^\tau.M_0$ it is a tree t such that $t^M|_0 = t^{M_0}$ and $t(\varepsilon) = \langle \lambda^\tau, x\rangle$, we write the tree as $\langle \lambda^\tau, x\rangle(t^{M_0})$.

A tree that represents the first term in the sequence (1) in Example 3 is presented in Figure 1.

We can identify terms of λ-calculus with such trees since the sets are clearly in bijection one with the other. We introduce now the notion of α-conversion for trees and identify α-equivalent trees. First, let us define variable renaming.

▶ **Definition 6** (variable renaming)**.** We define inductively $t[y := x]^\tau$ in the following way

- $\langle \mathsf{Var}^{\tau'}, z\rangle[y := x]^\tau = \langle \mathsf{Var}^{\tau'}, z\rangle$ for $z \neq y$ or $\tau' \neq \tau$,
- $\langle \mathsf{Var}^\tau, y\rangle[y := x]^\tau = \langle \mathsf{Var}^\tau, x\rangle$,
- $@(t_0, t_1)[y := x]^\tau = @(t_0[y := x]^\tau, t_1[y := x]^\tau)$,
- $\langle \lambda^{\tau'}, z\rangle(t^{M_0})[y := x]^\tau = \langle \lambda^{\tau'}, z\rangle(t^{M_0}[y := x]^\tau)$ when $z \neq y, z \neq x$ or $\tau' \neq \tau$,
- $\langle \lambda^\tau, x\rangle(t^{M_0})[y := x]^\tau = \langle \lambda^\tau, z\rangle(t^{M_0}[x := z]^\tau[y := x]^\tau)$ where $z \neq y$ and $z \neq x$,
- $\langle \lambda^\tau, y\rangle(t^{M_0})[y := x]^\tau = \langle \lambda^\tau, y\rangle(t^{M_0})$.

The α-equivalence itself is defined as follows.

▶ **Definition 7** (α-equivalence)**.** For each variable x such that t does not have a free occurrence of the node $\langle \mathsf{Var}^\tau, x\rangle$ and each variable y we let $\langle \lambda^\tau, y\rangle(t) \equiv^0_\alpha \langle \lambda^\tau, x\rangle(t[y := x]^\tau)$. The closure of $\equiv^0_\alpha$ over the structure of trees is defined as $\equiv^s_\alpha$. The α conversion $\equiv_\alpha$ is defined as the reflexive-transitive closure of $\equiv^s_\alpha$.

As we can see, this definition is slightly non-standard since it makes it possible to use the same variable name in two different types as if they were two different variables. Indeed the variables are made different by their types. We admit that the term $\lambda x^{\alpha\to\alpha}.\lambda x^\alpha.x^{\alpha\to\alpha}x^\alpha$ is probably not legible for humans, but for machines it is as good as $\lambda x^{\alpha\to\alpha}.\lambda y^\alpha.x^{\alpha\to\alpha}y^\alpha$. Note that this does not work well for Curry-style terms where there is no way to distinguish different variables through their types. The advantage of this style is that it requires fewer variable names to represent λ-terms.

3.2 Inhabitation machines

The proof search associated with the inhabitation problem can be done in two fashions. In the generative fashion, we start with axioms and step-wise apply rules associated with

connectives bottom-up until the desired goal is reached. Another approach, called analytic fashion, consists in step-wise decomposition of the formula top-down until axioms are reached. Our definition of automaton follows the latter approach so it is a version of top-down tree automata.

▶ **Definition 8** (inhabitation machines)**.** A *(multiple assignment) inhabitation machine* (IM) $\mathcal{A}$ is a tuple $\langle \Sigma, N, Q, q_I, \mathcal{R}, \delta \rangle$ where Σ is a finite signature, N is an infinite set of data elements, Q is a finite set of states, $q_I \in Q$ is the initial state, $\mathcal{R}$ is a finite set (of register names), and $\delta \subseteq \Sigma \times Q \times P_{\mathsf{fin}}(\mathcal{R}) \times P_{\mathsf{fin}}(Q) \times P_{\mathsf{fin}}(\mathcal{R})$ is a set of rules written as $a, q, R \rightsquigarrow q_0, q_1, \ldots, q_{n-1}, W$ where $a \in \Sigma$, $q, q_0, \ldots, q_{n-1} \in Q$, and $R, W \in P_{\mathsf{fin}}(\mathcal{R})$.

The machine traverses labelled trees where the set of labels is $\Sigma \times N \cup \Sigma$. The arity of a pair $\langle a, x \rangle \in \Sigma \times N$ is the arity of a. We assume that all the rules respect the arity so that $\mathsf{arity}(a) = n$ in the rule above. In case all the rules are such that R, W are either empty or singleton sets, the machines are called *single assignment inhabitation machines.*

Observe that the transition rules of the machine do not include elements of the set N of data elements.

The operational semantics for such a machine is as follows. Configurations of $\mathcal{A}$ in a tree t are elements of $\mathsf{Config} = \mathsf{dom}(t) \times Q \times \mathsf{Reg}$ where $\mathsf{Reg} = \mathcal{R} \to P_{\mathsf{fin}}(N)$. Note that an element of Reg models a situation when a finite set of elements is held in a register of a given name from $\mathcal{R}$. Suppose we are in a configuration $\langle \pi, q, f \rangle$. Consider a rule

$$a, q, R \rightsquigarrow q_0, q_1, \ldots, q_{n-1}, W.$$

This rule is *applicable* in the configuration when

- $R = W = \emptyset$ and $t(\pi) = a$, or
- $t(\pi) = \langle a, x \rangle$ and $x \in f(r)$ for all $r \in R$.

As a result of such a rule the machine splits its control and moves to all n sons of the node π (recall that the arity must be respected both by the tree and by the rule) and for $i \in \overline{n}$ the *i-th resulting configuration* is $\langle \pi \cdot i, q_i, f^W_\downarrow \rangle$ where $f^W_\downarrow : \mathcal{R} \to P_{\mathsf{fin}}(N)$ is defined as

$$f^W_\downarrow(l) = \begin{cases} f(l) & \text{for } l \notin W, \\ f(l) \cup \{x\} & \text{for } l \in W. \end{cases} \tag{2}$$

Note that in case $W = \emptyset$ the condition in the second case of the definition is not possible so this pattern defines $f^W_\downarrow$ equal to f. We drop the superscript W whenever the set is clear from the context.

Whenever it does not lead to confusion we flatten the rules and instead of

$$a, q, \{i_0, \ldots, i_k\} \rightsquigarrow q_0, q_1, \ldots, q_{n-1}, \{j_0, \ldots, j_l\} \tag{3}$$

we write simply $a, q, i_0, \ldots, i_k \rightsquigarrow q_0, q_1, \ldots, q_{n-1}, j_0, \ldots, j_l$.

A *run* of a machine $\mathcal{A}$ on a tree t is a function $\mathfrak{r} : \mathsf{dom}(t) \to \mathsf{Config}$ that respects the rules of δ, i.e. for each node $\pi \in \mathsf{dom}(t)$ there is a rule $a, q, R \rightsquigarrow q_0, q_1, \ldots q_{n-1}, W \in \delta$ that is applicable in the configuration $\mathfrak{r}(\pi)$ and for each son i of the node π the configuration $\mathfrak{r}(\pi \cdot i)$ is the i-th resulting configuration of the rule.

We say that a machine $\mathcal{A}$ *accepts a tree t from a configuration* $\langle \pi, q, f \rangle$ when there is a correct run on $t|_\pi$ of $\mathcal{A}$ that starts with the configuration $\langle \varepsilon, q, f \rangle$. Let us define $f_I : \mathcal{R} \to P_{\mathsf{fin}}(N)$ so that $f_I(r) = \emptyset$ for all $r \in \mathcal{R}$. We say that the machine $\mathcal{A}$ *accepts a tree* t when there is a correct run of the machine that starts in $\langle \varepsilon, q_I, f_I \rangle$. The set of all trees t such that $\mathcal{A}$ accepts t is written $L(\mathcal{A})$.

▶ Remark (a version for the Curry style). A slightly different notion of machine is necessary to deal with terms in the Curry style. The definition of the resulting configuration must be modified. The state of the registers should change in a different way and the definition of the functions $f^W_\downarrow$ from (2) should be replaced with the following one

$$f^W_\downarrow(l) = \begin{cases} f(l)\backslash\{x\} & \text{for } l \notin W, \\ f(l) \cup \{x\} & \text{for } l \in W. \end{cases} \tag{4}$$

In this way, we maintain the interpretation that a particular variable name is active in a given scope for only one λ binder. The whole development of this paper could be redone for machines that use this version of register update. Its full examination is left for the full version of the paper.

Hereafter, a theorem is presented that relates inhabitation in $\lambda_\to$ and our machines. Before we formulate it, we define a crucial machine that is used there. The machine $\mathcal{A}_\tau$ is $\langle \Sigma^\tau_T, \mathsf{V}, Q, q_I, \mathcal{R}, \delta\rangle$ where

- $\Sigma^\tau_T, \mathsf{V}$ are defined as in Section 3.1,
- $Q = \{q_\sigma, q^s_\sigma \mid \sigma \text{ is a subexpression of } \tau\}$,
- $q_I = q_\tau$,
- $\mathcal{R}$ is the set of subexpressions of τ.

The rules of δ are as follows:

1. $\mathsf{Var}^\sigma, q_\sigma, \sigma \rightsquigarrow \emptyset$,
2. $\mathsf{Var}^\sigma, q^s_\sigma, \sigma \rightsquigarrow \emptyset$,
3. $@, q_\sigma, \emptyset \rightsquigarrow q^s_{\sigma'\to\sigma}, q_{\sigma'}, \emptyset$,
4. $@, q^s_\sigma, \emptyset \rightsquigarrow q^s_{\sigma'\to\sigma}, q_{\sigma'}, \emptyset$,
5. $\lambda^\sigma, q_{\sigma\to\sigma'}, \emptyset \rightsquigarrow q_{\sigma'}, \sigma$.

Note that these rules are such that the resulting machine is a single assignment IM.

▶ **Example 9.** Let us see how this construction works for the type $\mathbf{3}$. First note that subexpressions of $\mathbf{3}$ form the set $\mathcal{R}^{\mathbf{3}} = \{0, \mathbf{1}, \mathbf{2}, \mathbf{3}\}$. The automaton $\mathcal{A}_{\mathbf{3}} = \langle \Sigma^{\mathbf{3}}_T, \mathsf{V}, Q^{\mathbf{3}}, q_{\mathbf{3}}, \mathcal{R}^{\mathbf{3}}, \delta^{\mathbf{3}}\rangle$ where $\Sigma^{\mathbf{3}} = \{\mathsf{Var}^0, \mathsf{Var}^{\mathbf{1}}, \mathsf{Var}^{\mathbf{2}}, \mathsf{Var}^{\mathbf{3}}, \lambda^0, \lambda^{\mathbf{1}}, \lambda^{\mathbf{2}}, \lambda^{\mathbf{3}}, @\}$, $Q^{\mathbf{3}} = \{q_0, q_{\mathbf{1}}, q_{\mathbf{2}}, q_{\mathbf{3}}, q^s_0, q^s_{\mathbf{1}}, q^s_{\mathbf{2}}, q^s_{\mathbf{3}}\}$. The rules of $\delta^{\mathbf{3}}$ are

$\mathsf{Var}^0, q_0, 0 \rightsquigarrow \emptyset$,	$\mathsf{Var}^{\mathbf{1}}, q_{\mathbf{1}}, \mathbf{1} \rightsquigarrow \emptyset$,	$\mathsf{Var}^{\mathbf{2}}, q_{\mathbf{2}}, \mathbf{2} \rightsquigarrow \emptyset$,	$\mathsf{Var}^{\mathbf{3}}, q_{\mathbf{3}}, \mathbf{3} \rightsquigarrow \emptyset$,
$\mathsf{Var}^0, q^s_0, 0 \rightsquigarrow \emptyset$,	$\mathsf{Var}^{\mathbf{1}}, q^s_{\mathbf{1}}, \mathbf{1} \rightsquigarrow \emptyset$,	$\mathsf{Var}^{\mathbf{2}}, q^s_{\mathbf{2}}, \mathbf{2} \rightsquigarrow \emptyset$,	$\mathsf{Var}^{\mathbf{3}}, q^s_{\mathbf{3}}, \mathbf{3} \rightsquigarrow \emptyset$,
$@, q_0, \emptyset \rightsquigarrow q^s_{0\to 0}, q_0, \emptyset$,	$@, q_0, \emptyset \rightsquigarrow q^s_{\mathbf{1}\to 0}, q_{\mathbf{1}}, \emptyset$,	$@, q_0, \emptyset \rightsquigarrow q^s_{\mathbf{2}\to 0}, q_{\mathbf{2}}, \emptyset$,	
$@, q^s_0, \emptyset \rightsquigarrow q^s_{0\to 0}, q_0, \emptyset$,	$@, q^s_0, \emptyset \rightsquigarrow q^s_{\mathbf{1}\to 0}, q_{\mathbf{1}}, \emptyset$,	$@, q^s_0, \emptyset \rightsquigarrow q^s_{\mathbf{2}\to 0}, q_{\mathbf{2}}, \emptyset$,	
$\lambda^0, q_{0\to 0}, \emptyset \rightsquigarrow q_0, 0$,	$\lambda^{\mathbf{1}}, q_{\mathbf{1}\to 0}, \emptyset \rightsquigarrow q_0, \mathbf{1}$,	$\lambda^{\mathbf{2}}, q_{\mathbf{2}\to 0}, \emptyset \rightsquigarrow q_0, \mathbf{2}$.	

The construction presented here is a little bit not optimal as not all states and rules are reachable from the initial configuration. Yet, it is still simple in formulation and therefore convenient to handle in proofs.

Figure 2 presents the states of $\mathcal{A}_{\mathbf{3}}$ reachable from the initial configuration. An annotation next to an edge there indicates the alphabet symbol that is used to traverse it. For comparison with the machine in the book by Barendregt, Dekkers, and Statman [1, p. 36], the thick edges in the picture correspond directly to the edges there, while the thin edges should be collapsed to one edge labelled with F.

To demonstrate the operation of the automaton, we present here its run that witnesses that $\mathcal{A}_{\mathbf{3}}$ accepts the tree presented in Figure 1.

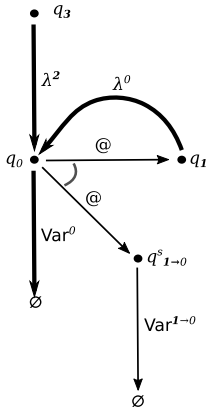

Figure 2 The automaton $\mathcal{A}_{\mathbf{3}}$ after removing non-reachable states.

1. $\langle \varepsilon, q_{\mathbf{3}}$, | 0 | **1** | **2** | **3** | $\rangle$ — We start at the root position in the initial state and with empty registers. The only possible rule to use is $\lambda^{\mathbf{2}}, q_{\mathbf{2}\to 0}, \emptyset \rightsquigarrow q_0, \mathbf{2}$.
2. $\langle 0, q_0$, | 0 | **1** | **2** | **3** | $\rangle$ (**2**: F) — The register **2** was filled with a variable (F). We cannot apply the rule $\mathsf{Var}^0, q_0, 0 \rightsquigarrow \emptyset$ since the register 0 is empty. The only rules that remain are $@, q_0, \emptyset \rightsquigarrow q^s_\sigma, q_0, \emptyset$ where $\sigma \in \{0 \to 0, \mathbf{1} \to 0, \mathbf{2} \to 0\}$. After a while of analysis we can see that options where $\sigma \neq \mathbf{1} \to 0$ cannot lead to a successful computation. Thus, we follow the rule with $\sigma = \mathbf{1} \to 0$ and the computation forks to points 3. and 4. below.
3. $\langle 0, q^s_{\mathbf{1}\to 0},$ | 0 | **1** | **2** | **3** | $\rangle$ (**2**: F) — Since $\mathbf{1} \to 0 = \mathbf{2}$ and the register **2** is not empty, we can apply the rule $\mathsf{Var}^{\mathbf{2}}, q^s_{\mathbf{2}}, \mathbf{2} \rightsquigarrow \emptyset$ and successfully terminate this branch of computation.
4. $\langle 0, q_{\mathbf{1}}$, | 0 | **1** | **2** | **3** | $\rangle$ (**2**: F) — The register **1** is empty so we cannot apply the rule $\mathsf{Var}^{\mathbf{1}}, q_{\mathbf{1}}, \mathbf{1} \rightsquigarrow \emptyset$. Therefore, the only option is to use $\lambda^0, q_{0\to 0}, \emptyset \rightsquigarrow q_0, 0$ here and come back to q_0.
5. $\langle 0, q_0$, | 0 | **1** | **2** | **3** | $\rangle$ (0: y, **2**: F) — The register 0 is no longer empty so we can apply this time the rule $\mathsf{Var}^0, q_0, 0 \rightsquigarrow \emptyset$, which concludes the run of the automaton.

In step 5. we could use the rule with @ as in the step 2. and get into another turn of the loop visible in Figure 2. Looping there makes it possible to obtain trees representing other terms from (1) on page 131.

▶ **Theorem 10.** *For each type τ the language $L(\mathcal{A}_\tau)$ is the set of normal forms that are closed inhabitants of τ.*

Proof. Given a state of registers $f \in \mathsf{Reg}$ we can define a context Γ^f as $\Gamma^f = \{x^\sigma \mid x \in f(\sigma)\}$. Similarly, given a context Γ we can define a state of registers $f^\Gamma : \mathcal{R} \to P_{\mathsf{fin}}(\mathsf{V})$ determined as $f^\Gamma(\sigma) = \{x^\sigma \mid x^\sigma \in \Gamma\}$ where σ is a subexpression of τ. We have now the following fact:

1. Let σ be a subexpression of τ. If Γ is a context that contains only elements of the form $x^{\sigma'}$ where σ' is a subexpression of τ and $M \in \Lambda^\Gamma_{n,\to}(\sigma)$ ($M \in \Lambda^\Gamma_{s,\to}(\sigma)$) then there is a tree t^M such that $\mathcal{A}_\tau$ accepts t^M from a configuration $\langle \varepsilon, q_\sigma, f^\Gamma \rangle$ ($\langle \varepsilon, q^s_\sigma, f^\Gamma \rangle$, respectively).
2. If $\mathcal{A}_\tau$ accepts a tree t from a configuration $\langle \varepsilon, q_\sigma, f \rangle$ ($\langle \varepsilon, q^s_\sigma, f \rangle$) then there is a term $M \in \Lambda^{\Gamma^f}_{n,\to}(\sigma)$ ($M \in \Lambda^{\Gamma^f}_{s,\to}(\sigma)$, respectively), where M is such that $t = t^M$.

The proof of (1) is by induction over the structure of M

In case $M = x^\sigma$, we observe that $x^\sigma \in \Lambda^\Gamma_{n,\to}(\sigma)$ is possible only when $x^\sigma \in \Gamma$. This implies, as σ is a subexpression of τ, that $x^\sigma \in f^\Gamma(\sigma)$. As a result $\mathcal{A}_\tau$ accepts the tree $\langle \mathsf{Var}^\sigma, x\rangle$ from the configuration $\langle \varepsilon, q_\sigma, f^\Gamma\rangle$ through the rule $\mathsf{Var}^\sigma, q_\sigma, \sigma \rightsquigarrow \emptyset$. Similar argument applies for $x^\sigma \in \Lambda^\Gamma_{s,\to}(\sigma)$, but we have to use the rule $\mathsf{Var}^\sigma, q^s_\sigma, \sigma \rightsquigarrow \emptyset$.

In case $M = M_0M_1$, we observe that $M_0M_1 \in \Lambda^\Gamma_{n,\to}(\sigma)$ is possible only when $M_0 \in \Lambda^\Gamma_{s,\to}(\sigma' \to \sigma)$ and $M_1 \in \Lambda^\Gamma_{n,\to}(\sigma')$. By Proposition 1, the types $\sigma' \to \sigma$ and σ' are subexpressions of τ. Note that by the induction hypothesis we obtain a tree t^{M_0} such that $\mathcal{A}_\tau$ accepts it from the configuration $\langle \varepsilon, q^s_{\sigma'\to\sigma}, f^\Gamma\rangle$ through a run $\mathfrak{r}_0$ and a tree t^{M_1} such that $\mathcal{A}_\tau$ accepts it from the configuration $\langle \varepsilon, q_{\sigma'}, f^\Gamma\rangle$ through a run $\mathfrak{r}_1$. Let us construct a tree $t^M = @(t^{M_0}, t^{M_1})$ and a run $\mathfrak{r}$ over t^M such that $\mathfrak{r}(\varepsilon) = \langle \varepsilon, q_\sigma, f^\Gamma\rangle$, $\mathfrak{r}(0\cdot\pi) = \langle 0\cdot\pi, q', f'\rangle$ where $\mathfrak{r}_0(\pi) = \langle \pi, q', f'\rangle$, and $\mathfrak{r}(1\cdot\pi) = \langle 1\cdot\pi, q', f'\rangle$ where $\mathfrak{r}_1(\pi) = \langle \pi, q', f'\rangle$. It is easy to see that the rule $@, q_\sigma, \emptyset \rightsquigarrow q^s_{\sigma'\to\sigma}, q_{\sigma'}, \emptyset$ is applicable in the root node of t and for other nodes the function $\mathfrak{r}$ respects δ as $\mathfrak{r}_0$ and $\mathfrak{r}_1$ did.

Similar argument applies for $M_0M_1 \in \Lambda^\Gamma_{s,\to}(\sigma)$, but we have to use the rule $@, q^s_\sigma, \emptyset \rightsquigarrow q^s_{\sigma'\to\sigma}, q_{\sigma'}, \emptyset$.

In case $M = \lambda x^{\sigma'}.M_0$ we observe that $\lambda x^{\sigma'}.M_0 \in \Lambda^\Gamma_{n,\to}(\sigma' \to \tau')$ is possible only when $M_0 \in \Lambda^\Gamma_{n,\to}(\tau')$. By the induction hypothesis we obtain a tree t^{M_0} such that $\mathcal{A}_\tau$ accepts it from the configuration $\langle \varepsilon, q_{\tau'}, f^{\Gamma, x^{\sigma'}}\rangle$ through a run $\mathfrak{r}_0$. Let us consider the tree $t^M = \langle \lambda^{\sigma'}, x\rangle(t^{M_0})$ and a run $\mathfrak{r}$ over t such that $\mathfrak{r}(\varepsilon) = \langle \varepsilon, q_{\sigma'\to\tau'}, f^\Gamma\rangle$, and $\mathfrak{r}(0\cdot\pi) = \langle 0\cdot\pi, q', f'\rangle$ where $\mathfrak{r}_0(\pi) = \langle \pi, q', f'\rangle$. It is easy to see that the rule $\lambda^{\sigma'}, q_{\sigma'\to\tau'}, \emptyset \rightsquigarrow q_{\tau'}, \sigma'$ is applicable in the root node of t^M, and for other nodes the function $\mathfrak{r}$ respects δ as $\mathfrak{r}_0$ did.

The proof of (2) is by induction over the structure of t

In case $t = \langle \mathsf{Var}^{\tau'}, x\rangle$ and $\mathcal{A}_\tau$ accepts it from a configuration $\langle \varepsilon, q_\sigma, f\rangle$ ($\langle \varepsilon, q^s_\sigma, f\rangle$) then it can happen only because a rule of the form

$$\mathsf{Var}^\sigma, q_\sigma, \sigma \rightsquigarrow \emptyset \qquad (\text{or } \mathsf{Var}^\sigma, q^s_\sigma, \sigma \rightsquigarrow \emptyset)$$

was used. Such a rule is applicable only when $\tau' = \sigma$ and the register σ contains x. As a result x^σ is in Γ^f. This means $x^\sigma \in \Lambda^{\Gamma^f}_{n,\to}(\sigma)$ ($x^\sigma \in \Lambda^{\Gamma^f}_{s,\to}(\sigma)$, respectively) and thus the set is not empty.

In case $t = @(t^0, t^1)$ and $\mathcal{A}_\tau$ accepts t from a configuration $\langle \varepsilon, q_\sigma, f\rangle$ ($\langle \varepsilon, q^s_\sigma, f\rangle$) then it can happen only because a rule of the form

$$@, q_\sigma, \emptyset \rightsquigarrow q^s_{\sigma'\to\sigma}, q_{\sigma'}, \emptyset \qquad (\text{or } @, q^s_\sigma, \emptyset \rightsquigarrow q^s_{\sigma'\to\sigma}, q_{\sigma'}, \emptyset)$$

was used. The machine $\mathcal{A}_\tau$ accepts the tree t^0 from the configuration $\langle \varepsilon, q^s_{\sigma'\to\sigma}, f\rangle$ and $\mathcal{A}_\tau$ accepts t^1 from the configuration $\langle \varepsilon, q_{\sigma'}, f\rangle$. By the induction hypothesis we obtain that some $M_0 \in \Lambda^{\Gamma^f}_{s,\to}(\sigma' \to \sigma)$ and $M_1 \in \Lambda^{\Gamma^f}_{n,\to}(\sigma')$. As a result $M_0M_1 \in \Lambda^{\Gamma^f}_{n,\to}(\sigma)$ (or $M_0M_1 \in \Lambda^{\Gamma^f}_{n,\to}(\sigma)$, respectively).

In case $t = \langle \lambda^{\sigma'}, x\rangle(t^0)$ and $\mathcal{A}_\tau$ accepts t from a configuration $\langle \varepsilon, q_\sigma, f\rangle$ ($\langle \varepsilon, q^s_\sigma, f\rangle$) then it can happen only because a rule of the form

$$\lambda^{\sigma'}, q_{\sigma'\to\tau'}, \emptyset \rightsquigarrow q_{\tau'}, \sigma'$$

was used, where $\sigma' \to \tau' = \sigma$. The machine $\mathcal{A}_\tau$ accepts t^0 from the configuration $\langle \varepsilon, f', q_{\tau'}\rangle$ where $f' = f[\sigma' \mapsto f(\sigma') \cup \{x\}]$. By the induction hypothesis we obtain that some $M_0 \in \Lambda^{\Gamma^{f'}}_{n,\to}(\tau')$, which gives us that $\lambda x^{\sigma'}.M_0 \in \Lambda^{\Gamma^f}_{n,\to}(\sigma' \to \tau') = \Lambda^{\Gamma^f}_{n,\to}(\sigma)$.

A direct check verifies that the terms M constructed above have the property that $t = t^M$.

We can apply the proven above fact to the subexpression $\tau' = \tau$ and obtain the desired conclusion of Theorem 10. ◀

The proof of the theorem above easily generalises to the formulation that involves open terms as follows – given a fixed set Γ of free variables, type τ the language $L(A^{\Gamma}_{\tau})$ is the set of normal forms that are closed inhabitants of τ with free variables in Γ. It is simply enough to start the automaton with registers appropriately filled with variables from Γ.

Although a precise account of the remark below goes beyond the scope of this paper, it is worth observing that we could omit from the construction the (spine) states of the form q^s_σ and we would still obtain representations of typable terms. These terms would not need to be in normal form, though. Still, we would not be able to obtain all typable terms as we are limited by the finite number of registers that hold variables of types being subexpressions of the original type. Notably, this kind of restriction is natural in certain scenarios, in particular non-normal accessible terms considered in the decidability proofs for various versions of the higher-order matching problem could be accepted by our machines.

3.3 Invariance of α-conversion

The machines accept trees that are constructed from a particular set of variables. Still, λ-terms are understood up to renaming of bound variables, i.e. α-conversion. To establish the connection with terms rather than their α-representants we need to establish that the languages of trees accepted by the IM's defined before the proof of Theorem 10 cannot separate two different α-equivalent trees. Let us start with a definition which are the machines of interest here.

▶ **Definition 11** (variable consistent IM's)**.** Let $\mathcal{A} = \langle \Sigma^{\tau}_{T}, \mathsf{V}, Q, q_I, \mathcal{R}, \delta\rangle$ where $\mathcal{R}$ is the set of subexpressions of τ. We say that $\mathcal{A}$ is *a variable consistent IM* when all its rules with symbols Var^{σ} have the form $\mathsf{Var}^{\sigma}, q, \sigma \rightsquigarrow W$ for some $\sigma \in \mathcal{R}$.

▶ **Proposition 12.** *If t_1, t_2 are trees representing λ-terms such that $t_1 \equiv^0_\alpha t_2$ and $\mathcal{A}$ is a variable consistent IM that accepts t_1 from a configuration $\langle \varepsilon, q, f\rangle$ then it accepts t_2 from the same configuration.*

Proof. Observe that $t_1 \equiv^0_\alpha t_2$ means that $t_1 = \langle \lambda^{\tau}, y\rangle(t)$ and $t_2 = \langle \lambda^{\tau}, x\rangle(t[y := x]^{\tau})$. Let $\mathcal{A} = \langle \Sigma^{\sigma}_{T}, \mathsf{V}, Q, q_I, \mathcal{R}, \delta\rangle$. The proof is by induction over the size of the tree t.

In case $t = \langle \mathsf{Var}^{\tau'}, z\rangle$ we observe that $\mathcal{A}$ accepts t_1 from the configuration $\langle \varepsilon, q, f\rangle$, it must be the case that two rules are used to accomplish this

$$\lambda^{\tau}, q_1, R \rightsquigarrow q_2, W, \qquad \mathsf{Var}^{\tau'}, q_2, R' \rightsquigarrow W' \tag{5}$$

where $R, W, R, W' \in P_{\mathsf{fin}}(\mathcal{R})$.

We have now two subcases, (a) $\tau = \tau'$ with $z = y$, (b) $\tau = \tau'$ with $z \neq y$ or $\tau \neq \tau'$. In case (a), we have $t_1 = \langle \lambda^{\tau}, y\rangle(\langle \mathsf{Var}^{\tau}, y\rangle)$, $t_2 = \langle \lambda^{\tau}, x\rangle(\langle \mathsf{Var}^{\tau}, x\rangle)$. By a direct check we can verify that the same two rules can be used to accept t_2. The only non-trivial case is when $R' \neq \emptyset$, and then the check for presence of element in a register $i \in R'$ in both cases is positive since either $i \notin W$ and then in both cases $f(i)$ is the same or $i \in W$ and then in both cases the variable being checked is exactly the one that was added.

In case (b), we observe first that the case $\tau \neq \tau'$ is impossible in variable consistent IM's. Thus, we obtain that $t_1 = \langle \lambda^{\tau}, y\rangle(\langle \mathsf{Var}^{\tau}, z\rangle)$, $t_2 = \langle \lambda^{\tau}, x\rangle(\langle \mathsf{Var}^{\tau}, z\rangle)$. In this case, $W \cap R = \emptyset$ as otherwise the presence of z in any register of the intersection would mean $z = y$. This implies that only registers that were not modified by the rule with λ^{τ} can be checked and this makes the rules in (5) trivially accept t_2.

In case $t = @(t^0, t^1)$, we observe that $\mathcal{A}$ accepts t_1 with a run that starts with the rules

$$\lambda^{\tau}, q_1, R \rightsquigarrow q_2, W, \qquad @, q_2, \emptyset \rightsquigarrow q'_0, q'_1, \emptyset.$$

Note that once the machine is started in the initial configuration $\langle \varepsilon, q, f\rangle$ where $q = q_1$, it moves to two configurations $\langle i, q_i, f_\downarrow\rangle$ for $i = 0, 1$ where $f_\downarrow = f$ in case $W = \emptyset$ or $f_\downarrow = f[r_1 \mapsto f(r_1) \cup \{y\}] \cdots [r_l \mapsto f(r_l) \cup \{y\}]$ when $W = \{r_1, \ldots, r_l\}$. We accept t^i from configurations $\langle \varepsilon, q'_i, f_\downarrow\rangle$ for $i = 0, 1$. We take now two mild modifications $\mathcal{A}_i$ for $i = 0, 1$ of $\mathcal{A}$ obtained by adding the transition

$$\lambda^\tau, q_\bullet, \emptyset \rightsquigarrow q'_i, W$$

respectively, where $q_\bullet$ is a fresh state. We can directly verify that for $\mathcal{A}_i$ with the configuration $\langle \varepsilon, q_\bullet, f\rangle$ at the root of $t^i_1 = \langle \lambda^\tau, y\rangle(t^i)$, the resulting configuration is $\langle 0, q'_i, f_\downarrow\rangle$, and this configuration accepts t^i_1 provided that the machine accepts t^i from $\langle \varepsilon, q'_i, f\rangle$ for $i = 0, 1$. This is guaranteed by the fact that this holds for $\mathcal{A}$ and each its run is also a run of $\mathcal{A}_i$. We can now use the induction hypothesis to verify that $\mathcal{A}_i$ accept $t^i_2 = \langle \lambda^\tau, x\rangle(t^i[y := x]^\tau)$ from respective configurations for $i = 0, 1$. Note that the runs arrive at the configurations $\langle 0, q'_i, f_\downarrow\rangle$ respectively. They can be then turned to runs that accept $t^i[y := x]^\tau$ from $\langle \varepsilon, q'_i, f\rangle$. As the initial state does not occur on the right-hand sides of the rules they are actually runs of $\mathcal{A}$ too. Since the rule $\lambda^\tau, q_1, R \rightsquigarrow q_2, W$ transforms the initial configuration to a tuple $\langle i, q'_i, f_\downarrow\rangle$ and these are accepting when $\langle \varepsilon, q'_i, f_\downarrow\rangle$ are accepting for t_i, we obtain our conclusion that $\mathcal{A}$ accepts $\langle \lambda^\tau, x\rangle(t[y := x]^\tau)$.

In case $t = \langle \lambda^\tau, x\rangle(t)$ the proof is similar as in the previous case. The details are left to the reader. ◀

In case the machines are not variable consistent, we can find two trees, namely $t_1 = \langle \lambda^\tau, y\rangle(\langle \mathsf{Var}^\sigma, z\rangle)$ and $t_2 = \langle \lambda^\tau, x\rangle(\langle \mathsf{Var}^\sigma, z\rangle)$, that are in the relation $\equiv^0_\alpha$ but t_1 is accepted by a machine that rejects t_2.

▶ **Proposition 13.** *For any variable consistent IM $\mathcal{A}$, if $t_1 \equiv^s_\alpha t_2$ and $t_1 \in L(\mathcal{A})$ then $t_2 \in L(\mathcal{A})$.*

Proof. The proof is by a straightforward induction over the structure of t_1. Details are left to the reader. ◀

▶ **Theorem 14** (invariance of α-conversion)**.** *For any variable consistent IM $\mathcal{A}$, if $t_1 \equiv_\alpha t_2$ and $t_1 \in L(\mathcal{A})$ then $t_2 \in L(\mathcal{A})$.*

Proof. The proof is by induction on the number n of $\equiv^s_\alpha$ steps to obtain $t_1 \equiv_\alpha t_2$. The case of 0 steps is trivial since then $t_1 = t_2$. In case $n > 0$ we have t'_2 such that $t_1 \equiv_\alpha t'_2 \equiv^s_\alpha t_2$ and $t_1 \equiv_\alpha t'_2$ requires less than n steps of $\equiv^s_\alpha$. We obtain that $t'_2 \in L(\mathcal{A})$ by Proposition 13 and then $t_1 \in L(\mathcal{A})$ by the induction hypothesis. ◀

3.4 Closure properties

The advantage of automata is that they make it possible to easily give constructs for the sum or intersection of languages. This is done through closure constructions. We present these for the (multiple assignment) IM's proposed here.

▶ **Theorem 15.** *For all tree languages L_1, L_2 over a signature Σ if there are IM's $\mathcal{A}_i$ such that $L_i = L(\mathcal{A}_i)$ for $i = 1, 2$ then*

1. *there is a machine $\mathcal{A}$ such that $L(\mathcal{A}) = L_1 \cup L_2$,*
2. *there is a machine $\mathcal{A}$ such that $L(\mathcal{A}) = L_1 \cap L_2$.*

Proof. Let us first assume that $\mathcal{A}_i = \langle \Sigma, N, Q_i, q_{i,I}, \mathcal{R}_i, \delta_i \rangle$ for $i = 1, 2$ where $Q_1 \cap Q_2 = \emptyset$ and $\mathcal{R}_1 \cap \mathcal{R}_2 = \emptyset$. This assumption does not weaken our proof.

For the proof of (1) we define the machine whose states are the sum of states from $\mathcal{A}_1$ and $\mathcal{A}_2$ with registers $\mathcal{R}_1 \cup \mathcal{R}_2$. In addition the machine has a fresh initial state from which one can move non-deterministically either to states of $\mathcal{A}_1$ or to states of $\mathcal{A}_2$ and then continue the run according to the set of rules from the chosen this way machine. More precisely, $\mathcal{A} = \langle \Sigma, N, Q', q'_I, \mathcal{R}', \delta' \rangle$ where $Q' = Q_1 \cup Q_2 \cup \{q'_I\}$, q'_I is a fresh state, $\mathcal{R}' = \mathcal{R}_1 \cup \mathcal{R}_2$, and at last

$$\begin{aligned}\delta' = \delta_1 \cup \delta_2 \cup\ & \{a, q'_I, R \rightsquigarrow q_0, q_1, \ldots, q_{n-1}, W \mid a, q_{1,I}, R \rightsquigarrow q_0, q_1, \ldots, q_{n-1}, W \in \delta_1\} \cup \\ & \{a, q'_I, R \rightsquigarrow q_0, q_1, \ldots, q_{n-1}, W \mid a, q_{2,I}, R \rightsquigarrow q_0, q_1, \ldots, q_{n-1}, W \in \delta_2\}.\end{aligned}$$

The details of demonstration that this machine indeed recognises $L_1 \cup L_2$ are left to the reader.

For the proof of (2) we define the machine whose states are the product of the states from $\mathcal{A}_1$ and $\mathcal{A}_2$ with registers $\mathcal{R}_1 \cup \mathcal{R}_2$. The resulting machine simulates a run of $\mathcal{A}_1$ on the first coordinate and a run of $\mathcal{A}_2$ on the second one. When a set of registers R_1 is read in a rule of $\mathcal{A}_1$ and R_2 is read in a rule of $\mathcal{A}_2$ we combine them in a rule of the resulting machine by taking $R_1 \cup R_2$. Similarly for registers to write to. In this way, each time a set of registers is checked for presence of the current element of data all the registers in the rule from $\mathcal{A}_1$ are checked as well as ones for the rule from $\mathcal{A}_2$. Similarly for writes. In more detail the resulting machine is $\mathcal{A} = \langle \Sigma, N, Q', q'_I, \mathcal{R}', \delta' \rangle$ where $Q' = Q_1 \times Q_2$, $q_I = \langle q_{1,I}, q_{2,I} \rangle$, $\mathcal{R}' = \mathcal{R}_1 \cup \mathcal{R}_2$, and at last

$$\begin{aligned}\delta' = \{a, \langle q_{1,0}, q_{2,0} \rangle, R_1 \cup R_2 \rightsquigarrow \langle q_{1,1}, q_{2,1} \rangle, \langle q_{1,2}, q_{2,2} \rangle, \ldots \langle q_{1,n}, q_{2,n} \rangle, W_1 \cup W_2 \mid \\ a, q_{1,0}, R_1 \rightsquigarrow q_{1,1}, q_{1,2}, \ldots q_{1,n}, W_1 \in \delta_1, \text{ and} \\ a, q_{2,0}, R_2 \rightsquigarrow q_{2,1}, q_{2,2}, \ldots q_{2,n}, W_2 \in \delta_2\}.\end{aligned}$$

The details of demonstration that this machine recognises $L_1 \cap L_2$ are left to the reader. ◀

The constructions above use multiple assignments. An observant reader may have spotted that the proof of Theorem 10 requires only singleton sets in rules of machines. As a result, the machines there are single assignment IM's. It is an open question if the automata are closed on intersection. Therefore, we decided to introduce to the general model multiple register manipulations in the fashion of *multiple assignment* automata considered by Kaminski and Francez [11] where the closure can be obtained as above.

An immediate application of the above closure properties is the extension of the language of closed terms typed in the simply typed λ-calculus to the calculus of intersection types of rank 1 [20] or sum types of rank 1 [8].

4 The emptiness problem

To see how the design of the machines fits the inhabitation problem we show that the emptiness problem for the machines has the same complexity as the one for $\lambda_{\rightarrow}$. For this, we need to introduce another kind of automata for which the decidability of the emptiness problem can be dealt with more straightforwardly. Although we do not explore this connection in detail, the automata can be viewed as a reformulation of the term recognition by grammars proposed by Takahashi et al. [18].

We give here a construction that works for single assignment IM's with yet another restriction. This restriction covers the machines defined for the proof of Theorem 10. The general case requires more work and its demonstration would depart too much from the topic of inhabitation for the simply typed λ-calculus.

▶ **Definition 16** (one operation machines). A single assignment IM $\mathcal{A}$ is a *one operation machine* when its rules have form $a, q, R \rightsquigarrow q_0, \ldots, q_{n-1}, W$ where at most one of the sets R, W is non-empty.

▶ **Definition 17** (binary automata). A *binary automaton* (BA) $\mathcal{A}$ is a tuple $\langle \Sigma, Q, q_I, \mathcal{R}, \delta \rangle$ where Σ is a finite signature, Q is a finite set of states, $q_I \in Q$ is the initial state, $\mathcal{R}$ is a finite set (of available register names), and δ is a set of rules of the form $a, q, R \rightsquigarrow q_0, q_1, \ldots, q_{n-1}, W$ where $a \in \Sigma$, $q, q_0, \ldots, q_{n-1} \in Q$, and $R, W \in \{\{r\} \mid r \in \mathcal{R}\} \cup \{\emptyset\}$.

The automaton traverses labelled trees where the set of labels is Σ. As before, we assume that all the rules respect the arity so that $\mathsf{arity}(a) = n$ in the definition above.

The operational semantics for such automaton is as follows. Configurations of $\mathcal{A}$ in a tree t are elements of $\mathsf{Config} = \mathsf{dom}(t) \times Q \times \mathsf{Reg}_\mathsf{b}$ where $\mathsf{Reg}_\mathsf{b} = \mathcal{R} \to \{0, 1\}$. Suppose we are in a configuration $\langle \pi, q, f \rangle$. Consider a rule

$$a, q, R \rightsquigarrow q_0, q_1, \ldots, q_{n-1}, W$$

the rule is *applicable* when

- $R = \emptyset$ and $t(\pi) = a$, or
- $R = \{r\}$ for some $r \in \mathcal{R}$, $t(\pi) = a$, and $f(r) = 1$.

When the rule is applied, the automaton forks the computation to all n sons of the tree (note that the arity must be respected both by the tree and by the rule) and for a node $i \in \overline{n}$ the *i-th resulting configuration* is $\langle \pi \cdot i, q_i, f^W_\Downarrow \rangle$ where $f^W_\Downarrow : \mathcal{R} \to \{0, 1\}$ is defined as

$$f^W_\Downarrow(l) = \begin{cases} f(l) & \text{for } l \notin W, \\ 1 & \text{for } l \in W. \end{cases} \tag{6}$$

Note that in case $W = \emptyset$ the condition in the second case of the definition is not possible so this pattern defines $f^W_\Downarrow$ equal to f. Again, we drop W whenever it is clear from the context.

A *run* of an automaton $\mathcal{A}$ on a tree t is a function $\mathfrak{r} : \mathsf{dom}(t) \to \mathsf{Config}$ that respects the rules of δ, i.e. for each node $\pi \in \mathsf{dom}(t)$ there is a rule

$$a, q, R \rightsquigarrow q_0, q_1, \ldots, q_{n-1}, W \in \delta$$

that is applicable in the configuration $\mathfrak{r}(\pi)$ and for each son i of the node π the configuration $\mathfrak{r}(\pi \cdot i)$ is the i-th resulting configuration of the rule.

We say that an automaton $\mathcal{A}$ *accepts a tree t from a configuration* $\langle \pi, q, f \rangle$ when there is a correct run of $\mathcal{A}$ on $t|_\pi$ that starts with the configuration $\langle \varepsilon, q, f \rangle$. Let us define the function $f_I : \mathcal{R} \to \{0, 1\}$ so that $f_I(r) = 0$ for $r \in \mathcal{R}$. We say that the automaton $\mathcal{A}$ *accepts a tree t* when there is a correct run of the automaton that starts in $\langle \varepsilon, q_I, f_I \rangle$. The set of all trees that $\mathcal{A}$ accepts is written $L(\mathcal{A})$.

We define a translation of one operation IM's to binary automata. Given a one operation inhabitation machine $\mathcal{A} = \langle \Sigma, N, Q, q_I, \mathcal{R}, \delta \rangle$ we fix a set $N_0 \subseteq N$ of size equal to $|\mathcal{R}|$ together with a bijection from $\mathcal{R}$ to N_0. We write n_r for the result of the bijection on an element $r \in \mathcal{R}$. We define a binary automaton $\mathcal{B} = \langle \Sigma_\mathcal{B}, Q_\mathcal{B}, q_{\mathcal{B},I}, \mathcal{R}_\mathcal{B}, \delta_\mathcal{B} \rangle$ where $\Sigma_\mathcal{B} = \Sigma \cup \Sigma \times N_0$, $Q_\mathcal{B} = Q$, $q_{\mathcal{B},I} = q_I$, $\mathcal{R}_\mathcal{B} = \mathcal{R}$, $\delta_\mathcal{B}$ contains for each rule

$$a, q, R \rightsquigarrow q_0, q_1, \ldots, q_{n-1}, W \in \delta \qquad \text{the rule} \qquad a', q, R \rightsquigarrow q_0, q_1, \ldots, q_{n-1}, W$$

where

- $a' = a$ when $R = W = \emptyset$,
- $a' = \langle a, n_r \rangle$ when $R = \emptyset$ and $W = \{r\}$,
- $a' = \langle a, n_r \rangle$ when $R = \{r\}$ and $W = \emptyset$.

We can now define the transformation operations for registers.

▶ **Definition 18** (register transformations). For $f : \mathcal{R} \to P_{\mathsf{fin}}(N)$ we define $f^\bullet : \mathcal{R} \to \{0, 1\}$ as $f^\bullet(r) = 0$ when $f(r) = \emptyset$ and $f^\bullet(r) = 1$ otherwise.

For $f : \mathcal{R} \to \{0, 1\}$ we define $f_\bullet : \mathcal{R} \to P_{\mathsf{fin}}(N)$ as $f_\bullet(r) = \emptyset$ when $f(r) = 0$ and $f_\bullet(r) = \{n_r\}$ where $n_r \in N_0$ otherwise.

▶ **Proposition 19.**

1. *If $\mathcal{A}$ accepts a tree t from a configuration $\langle \varepsilon, q, f \rangle$ then there is some t' such that $\mathcal{B}$ accepts t' from the configuration $\langle \varepsilon, q, f^\bullet \rangle$.*
2. *If $\mathcal{B}$ accepts a tree t from a configuration $\langle \varepsilon, f, q \rangle$ then $\mathcal{A}$ accepts t from the configuration $\langle \varepsilon, q, f_\bullet \rangle$.*

Proof. Both the proof of (1) and the proof of (2) are by induction over t. We only sketch the proof due to the lack of space. For illustration we present here a fragment of the proof for (1). The subcase concerns an internal node of the tree t. We know that a rule of the form

$$a, q, R \rightsquigarrow q_0, q_1, \ldots, q_{n-1}, W \in \delta$$

was applied in the configuration $\langle \varepsilon, q, f \rangle$ to accept t. We consider the subcase when $R = \emptyset$, $W = \{r\}$ and $t(\varepsilon) = \langle a, x \rangle$. We can consider the trees $t|_i$ and configurations $\langle \varepsilon, q_i, f_\downarrow \rangle$ for $i \in \overline{n}$ where $f_\downarrow$ is defined as in the pattern (2) on page 133. The automaton $\mathcal{A}$ accepts these trees from respective configurations. By the induction hypothesis there are trees t'_i for $i \in \overline{n}$ that $\mathcal{B}$ accepts from the configurations $\langle \varepsilon, q_i, (f_\downarrow)^\bullet \rangle$ for $i \in \overline{n}$ respectively. Note now that $(f_\downarrow)^\bullet = (f^\bullet)_\Downarrow$ so actually $\mathcal{B}$ accepts the trees from the configurations $\langle \varepsilon, q_i, (f^\bullet)_\Downarrow \rangle$. We can now define the tree $t' = \langle a, n_r \rangle(t'_0, \ldots, t'_{n-1})$ and by the definition of $\mathcal{B}$ the automaton contains the rule $a, q, \emptyset \rightsquigarrow q_0, \ldots, q_{n-1}, r \in \delta_{\mathcal{B}}$. This rule is applicable in the configuration $\langle \varepsilon, q, f^\bullet \rangle$ and leads to the mentioned above acceptable configurations $\langle \varepsilon, q_i, (f^\bullet)_\Downarrow \rangle$, which guarantees that $\mathcal{B}$ accepts t'. ◀

▶ **Proposition 20.** *The emptiness problem for one operation IM's is in PSPACE.*

Proof. To certify that the emptiness problem is in PSPACE we give a polynomial time alternating algorithm that given a machine $\mathcal{A} = \langle \Sigma, N, Q, q_I, \mathcal{R}, \delta \rangle$ checks for emptiness of $L(\mathcal{A})$. We first transform $\mathcal{A}$ to its corresponding binary automaton $\mathcal{B}$. Next, the algorithm keeps in its memory the configuration of $\mathcal{B}$ and a counter i of the number of steps. The initial configuration is $\langle \varepsilon, q_I, f \rangle$ and the initial counter $i = 0$. The algorithm proceeds by executing in loop the following three steps

1. it non-deterministically chooses a transition $a, q, R \rightsquigarrow q_0, \ldots, q_{n-1}, W \in \delta_{\mathcal{B}}$ and then
2. it universally moves to n configurations that result from applying the rule and that have $q_1, \ldots, q_n$ in their coordinates,
3. it increments i and in case $i > i_{\mathsf{max}} = k \cdot |Q|$ it leaves the loop with failure.

Observe that the loop is finished not only in step (3), but also in step (1) when a rule is chosen so that there are no states on the right-hand side of the chosen rule. In case the algorithm leaves the loop in this way it accepts. The bound $k \cdot |Q|$ is chosen so that in case there are more steps the state with register content must repeat (note that once a register is set to 1 it cannot be turned back to 0).

In this way the algorithm creates a potential tree t along a correct run on it. A closer examination of the procedure shows that this tree is in total discharge form. ◀

▶ **Proposition 21.** *The emptiness problem for IM's is hard for PSPACE.*

Proof. In Theorem 10, we defined machines that express inhabitation in $\lambda_{\rightarrow}$, which is PSPACE-hard [16]. ◀

As an immediate corollary of Proposition 20 and 21 we obtain the following theorem.

▶ **Theorem 22.** *The emptiness problem for one operation IM's is PSPACE-complete.*

5 Conclusions and further work

We have presented a model of automata and discussed how it corresponds to the inhabitation problem for the simply typed λ-calculus. As this was done for syntax with named binders, it is interesting to see how this would look like with de Bruijn indices. The binary automata presented in Section 4 recognise the language of terms in total discharge form. It would be interesting to see their version for depth bounded calculus of Dyckhoff-Hudelmeier [5, 10].

In addition to the presented closure properties for sum and intersection of languages one traditionally considers other operations such as substitution of languages, or cilindrification. The question concerning the closure for the operations remains open. Another interesting direction of study would be to give automata that deal with full propositional intuitionistic logic (i.e. one that includes logical alternative, conjunction, and negation). Automata for infinite tree languages with Büchi acceptance conditions can give a similar account to the one obtained in our Theorem 10 for Böhm trees.

We believe that Theorem 14 concerning the invariance of α-conversion can be generalised to a wider class of binding operators and to α-conversion that is expressed as a permutation of variables. In this way we would effectively obtain automata adequate for the Gabbay and Pitts [7] approach to binder syntax.

One more interesting direction would be to augment our automata with additional primitives that make it possible to recognise expressions in the relation of β-reduction. We believe that the automata with global equality constraints [2] can give here promising results.

References

1 Hendrik Pieter Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in logic. Cambridge University Press, 2013.

2 Luis Barguñó, Carles Creus, Guillem Godoy, Florent Jacquemard, and Camille Vacher. The emptiness problem for tree automata with global constraints. In Jean-Pierre Jouannaud, editor, *Proceedings of the LICS 2010*, pages 263–272. IEEE Computer Society, 2010.

3 Choukri-Bey Ben-Yelles. *Type-assignment in the lambda-calculus; syntax and semantics*. PhD thesis, Mathematics Department, University of Wales, Swansea, UK, 1979.

4 Sabine Broda and Luís Damas. On long normal inhabitants of a type. *Journal of Logic and Computation*, 15(3):353–390, 2005.

5 Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 57:795–807, 9 1992.

6 Boris Düdder, Moritz Martens, and Jakob Rehof. Staged composition synthesis. In Zhong Shao, editor, *Proceedings of ESOP 2014*, volume 8410 of *LNCS*, pages 67–86. Springer, 2014.

7 Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13(3-5):341–363, 2002.

8 Silvia Ghilezan. Inhabitation in intersection and union type assignment systems. *Journal of Logic and Computation*, 3(6):671–685, 1993.

9 J. Roger Hindley. *Basic simple type theory*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, New York, 1996.

10 Jörg Hudelmaier. An o(n log n)-space decision procedure for intuitionistic propositional logic. *Journal of Logic and Computation*, 3(1):63–75, 1993.

11 Michael Kaminski and Nissim Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.

12 Michael Kaminski and Tony Tan. Tree automata over infinite alphabets. In A. Avron, N. Dershowitz, and A. Rabinovich, editors, *Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*, volume 4800 of *LNCS*, pages 386–423. Springer, 2008.

13 C.-H. Luke Ong and Nikos Tzevelekos. Functional reachability. In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA*. IEEE Computer Society, 2009.

14 Dag Prawitz. *Natural Deduction*. Almqvist and Wiksell, Sweden, 1965.

15 Sylvain Salvati and Igor Walukiewicz. Krivine machines and higher-order schemes. *Information and Computation*, 239:340–355, 2014.

16 Richard Statman. Intuitionistic propositional logic is PSPACE-complete. *Theoretical Computer Science*, 9(1):67–72, 1979.

17 Colin Stirling. Dependency tree automata. In Luca de Alfaro, editor, *Proceedings of FOSSACS 2009*, volume 5504 of *LNCS*, pages 92–106. Springer, 2009.

18 Masako Takahashi, Yohji Akama, and Sachio Hirokawa. Normal proofs and their grammar. *Information and Computation*, 125(2):144–153, 1996.

19 Anne Troelstra and Helmut Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, 1996, 2000.

20 Paweł Urzyczyn. Inhabitation of low-rank intersection types. In Pierre-Louis Curien, editor, *Proceedings of TLCA 2009*, volume 5608 of *LNCS*, pages 356–370. Springer, 2009.

21 Mordchaj Wajsberg. Untersuchungen über den Aussagenkalkül von A. Heyting. *Wiadomości Matematyczne*, 46, 1938. English translation: On A. Heyting's propositional calculus, in *Mordchaj Wajsberg, Logical Works* (S. J. Surma, editor), Ossolineum, Wrocław, 1977, pages 132–171.