Addressing Machines — Modelling the Core of Functional Programming Languages

Giuseppe Della Penna\textsuperscript{a}, Benedetto Intrigila\textsuperscript{b}, Giulio Manzonetto\textsuperscript{c}

\textsuperscript{a}Department of Information Engineering, Computer Science and Mathematics, University of L’Aquila, Italy
\textsuperscript{b}Dipartimento di Ingegneria dell’Impresa, University of Rome “Tor Vergata”, Italy
\textsuperscript{c}Univ. Paris 13, Sorbonne Paris Cité, LIPN, UMR 7030, CNRS, F-93430 Villetaneuse, France.

Abstract

Turing machines and register machines have been used for decades in theoretical computer science as abstract models of computation. Also the $\lambda$-calculus has played a central role in this domain as it allows to focus on the notion of computation itself, while abstracting away from implementation details. The present article starts from the observation that the equivalence between these formalisms is based on the Church-Turing Thesis rather than an actual encoding of $\lambda$-terms into Turing (or register) machines. The reason is that these machines are not well-suited for modelling $\lambda$-calculus programs.

We introduce a new class of abstract machines that we call addressing machine because they are only able to manipulate memory addresses of other machines. The operations performed by such machines are very elementary: load an address in a register, apply a machine to another one via their addresses and, finally, call the address of another machine. We endow addressing machines with an operational semantics based on head-reduction and study their behaviour. The set of addresses of these machines can be easily turned into a combinatory algebra. In order to obtain a model of the full untyped $\lambda$-calculus, we need to introduce a rule that bares similarities with the $\omega$-rule and the rule $\zeta_\beta$ from combinatory logic.

Keywords: Addressing machines, $\lambda$-calculus, combinatory algebras, $\lambda$-models.

2020 MSC: 03 Mathematical logic and foundations
Introduction

A computation process executed by a processing unit (PU) is often seen as a sequence of instructions “driving” the PU. The code is evaluated, and a result is obtained at the end of the process, which then releases the PU. This is actually a simplification, since the execution is mediated by several layers of virtualization, whose aim is to make it more efficient, manageable and safe. As an example, any programming language contains so-called shortcut boolean operators, whose evaluation is truncated when the result is already decided (e.g., when the first argument of an OR expression evaluates to true), even if it contains further arguments that would lead to an error. This can be explained by considering the operator arguments as addresses pointing to sub-expressions, rather than actual sub-expressions. To evaluate a sub-expression, the PU must explicitly follow the address and execute the sub-expression code. Otherwise, it may safely skip (delete) it without producing any side effect. Similarly, at the operating system level, processes can be freely started but they actually get executed only when the Service Order (SO) monitor allows it, or they can be deleted to protect the system from unsafe operations. Again, the processes are not free to take over the machine: rather they are manipulated by the SO through their addresses (handles), and executed only if the SO explicitly passes the control to such addresses.

Addresses are also essential to implement higher-order features that are nowadays available in a variety of programming languages, including imperative ones like C#, F#, JavaScript, Lua and Python. Technically a higher-order function is simply a function taking functions as arguments, or returning a function as its result. This can only be achieved at low-level through functions pointers, that is, passing the addresses of those functions. The same mechanism is also essential to achieve another omnipresent construct of programming languages: recursion. Indeed, a recursive function can be seen as a higher-order function taking as input itself. Since a code that contains an arbitrary, variable number of nested copies of itself cannot be actually written, the self-call effect is achieved by telling the PU to continue the execution from the same starting address of the current code block. Finally, addresses are also handled by the PU as part of any data manipulation process, since data structures are not “hardcoded” in the code, but rather allocated on demand. Therefore, the code does not manipulate data structures directly, but through their addresses. This also concerns object oriented languages like Java, where the pointer arithmetic is mostly hidden.
We may conclude that such notion of address, or resource reference, is at the heart of the semantics of computation processes. Moreover, addresses model a primitive form of communication, since they can be used to reach, and then manipulate (execute, modify, read...), resources that are not “local” as they live in another part of the memory (e.g., memory addresses), of the machine (e.g., resource handles) or of a network (e.g., IP addresses). From this perspective, it is perhaps surprising that the classes of abstract machines commonly used in recursion theory, like Turing Machines (TMs) and Register Machines (RMs) \[1\], do not explicitly allow communication or introduce the concept of address. In fact, since TMs have a fixed program and only manipulate data, the effect of communication can be only achieved by encoding processes as data in a machine and then manipulate and execute such a code indirectly. This makes the simple, intuitive notions of addresses and communication described so far extremely difficult to realize.

Another well-established model of computation is certainly the untyped \(\lambda\)-calculus \[2\]. This formalism is interesting for our discussion because the concept of program execution is represented in a very different way. For instance, the notion of data is missing, to the point that processes are used also to represent natural numbers (through Church-numerals, or numerals from other numerical systems \[3\]). Communication is simpler to achieve, since a process can take another process as argument, whence the information can be passed to processes as parameters. This change of perspective makes it difficult to compare the expressive power of \(\lambda\)-calculus and that of TMs. It is well-known that \(\lambda\)-calculus is Turing–complete, but the result is achieved by proving that all computable numeric functions are representable as \(\lambda\)-terms, rather than defining faithful translations between the two formalisms. On the one side, researchers have shown that TMs are representable within \(\lambda\)-calculus \[4\]. On the other side, techniques for encoding \(\lambda\)-terms as TMs exist in the folklore, but they all rely on the choice of a specific evaluation strategy. To this day, no model of the full untyped \(\lambda\)-calculus (\(\lambda\)-model \[5, 6\]) is actually based on TMs or RMs. Similarly, all attempts at constructing a \(\lambda\)-model from Kleene’s partial combinatory algebra based on encodings of partial recursive functions were unsuccessful. Barendregt undertook this quest during his PhD thesis and, as a byproduct, arrived to the notion of \(\omega\)-rule — indeed, if such a model exists, it needs to satisfy the \(\omega\)-rule. Moreover, as Turing–completeness only concerns the representation of numeric functions, it does not give information on the actual expressiveness of the language. In particular its higher-order features are not taken into account (see also \[7\]).
In this article we define a new class of abstract machines, where the notions of address and communication (through addresses) are not only crucial to model computation, but they become the unique ingredients available. These machines are called \textit{addressing machines} and possess a finite tape from which they can read the input, some internal registers where they can store values read from the tape, and an internal program which is composed by a list of instructions that are executed sequentially. The input-tape and the internal registers are reminiscent of those in TMs and RMs, respectively. Every machine is uniquely identified by its address, which is a value taken from a fixed countable set $\mathbb{A}$. In this formalism, addresses are the only available data-type — this means that both the input-tape and the internal registers of a machine (once initialized) contain addresses from $\mathbb{A}$. Programs are written in an assembly language possessing only three instructions\footnote{This choice is made on purpose, in the attempt of determining the minimum amount of operations giving rise to a Turing-complete formalism.}. Besides reading its inputs, an addressing machine can apply two addresses $a, b$ with each other and store the resulting address $a \cdot b$ in an internal register. Intuitively, $a \cdot b$ is obtained by first taking the machine $M$ having address $a$, then appending $b$ to its input-tape, and finally calculating the address of this new machine. As a last step of its execution, an addressing machine can transfer the computation to another machine, possibly extending its input-tape, by retrieving its address from a register. Although not crucial in the abstract definition of an addressing machine, it should be clear at this point that any implementation of this formalism requires the association between the machines and their addresses to be effective (see Section \ref{sec:implementation} for more details).

Addressing machines share with $\lambda$-calculus the fact that there is no fundamental distinction between processes and data-types: in order to perform calculations on natural numbers a machine needs to manipulate the addresses of the corresponding numerals. Another similarity is the fact that in both settings communication is achieved by transferring the computation from one entity to another one. In the case of addressing machines, the machine currently “in execution” transfers the control by calling the address of another machine. In $\lambda$-calculus, the subterm “in charge” is the one occupying the so-called “head position” and the control of the computation is transferred when the head variable is substituted by another term. It is worth noting that process calculi such as the $\pi$-calculus also address communication using...
the concept of channel, where messages are exchanged \[8, 9\]. This is not the kind of communication that we are going to model here: our form of communication is encoded in the notion of address, so that a machine receiving a message results in a new machine with a different address. In other words, we do not model the dynamics of the communication, but the evolution of the machine addresses actually encodes the effects of communication. Another difference is the fact that \(\pi\)-calculus naturally models parallel computations as well as concurrency, while addressing machines are designed for representing sequential computations (one machine at a time is executed).

**Contents.** The aim of the paper is twofold. On the one side we want to present the class of addressing machines and analyze their fundamental properties. This is done in Section 2 where we describe their operational semantics in two different styles: as a term rewriting system (small-step semantics) and as a set of inference rules (big-step semantics). The two approaches are shown to be equivalent in case of addressing machines executing a terminating program (Proposition 2.15). On the other side, we wish to construct a model of the untyped \(\lambda\)-calculus based on addressing machines, and study the interpretations of \(\lambda\)-terms. For this reason, we recall in the preliminary Section 1 the main facts about \(\lambda\)-calculus, its equational theories and denotational models. It turns out that the set \(A\) of addresses, together with the operation of application previously described, is not a combinatory algebra (nor, *a fortiori*, a \(\lambda\)-model). In Section 3 we show that it can be turned into a combinatory algebra by quotienting under an equivalence relation arising naturally from our small-step operational semantics. Two addresses are equivalent if the corresponding machines are interconvertible using a more liberal rewriting relation. From the confluence property enjoyed by this relation, we infer the consistency of the algebra (Proposition 3.11). Unfortunately, the combinatory algebra so-obtained is not yet a model of \(\lambda\)-calculus – there are \(\beta\)-convertible \(\lambda\)-terms having different interpretations. Section 5 is devoted to showing that a \(\lambda\)-model actually arises when adding to the system a mild form of extensionality sharing similarities both with the \(\omega\)-rule in \(\lambda\)-calculus \[10\] and with the rule \(\zeta_\beta\) from combinatory logic \[11\]. The consistency of the model follows from an analysis of the underlying ordinal. Interestingly, the model itself is not extensional (Theorem 4.10).

**Disclaimer.** A preliminary version of addressing machines first appeared in Della Penna’s MSc thesis \[12\]. Although the main ingredients remain unchanged, there are some technical differences in the formalization. The definition of the \(\lambda\)-model in Section 5 and its consistency proof, are original.
1. Preliminaries

We present some notions that will be useful in the rest of the article.

1.1. The Lambda Calculus — Its Syntax

For the λ-calculus we mainly follow Barendregt’s first book [2]. We consider fixed a countable set Var of variables denoted by \( x, y, z, \ldots \).

**Definition 1.1.** The set \( \Lambda \) of \( \lambda \)-terms over Var is generated by the following simplified grammar (for \( x \in \text{Var} \)):

\[
M, N, P, Q ::= x \mid \lambda x.M \mid MN
\]

We assume that application is left-associative and has a higher precedence than \( \lambda \)-abstraction. Therefore \( \lambda x.\lambda y.\lambda z.xyz \) stands for \( (\lambda x.(\lambda y.(\lambda z.(xy))z)) \). We often write \( \lambda x_1 \ldots x_n.M \) for \( \lambda x_1 \ldots \lambda x_n.M \).

**Definition 1.2.** Let \( M \in \Lambda \).

(i) The set \( \text{FV}(M) \) of free variables of \( M \) is defined by induction:

\[
\begin{align*}
\text{FV}(x) &= \{x\}, \\
\text{FV}(\lambda x.P) &= \text{FV}(P) - \{x\}, \\
\text{FV}(PQ) &= \text{FV}(P) \cup \text{FV}(Q).
\end{align*}
\]

(ii) We say that \( M \) is closed, or a combinator, whenever \( \text{FV}(M) = \emptyset \).

(iii) We let \( \Lambda^o = \{ M \in \Lambda \mid \text{FV}(M) = \emptyset \} \) be the set of all combinators.

The variables occurring in \( M \) that are not free are called “bound”. From now on, \( \lambda \)-terms are considered modulo \( \alpha \)-conversion, namely, up to the renaming of bound variables (see [2, §2.1]).

**Notation 1.3.** Concerning specific combinators we let:

\[
\begin{align*}
\text{I} &= \lambda x.x, & \text{identity}, \\
\text{I} &= \lambda x.yxy, & \text{an } \eta\text{-expansion of the identity}, \\
\text{K} &= \lambda xy.x, & \text{first projection}, \\
\text{F} &= \lambda xy.y, & \text{second projection}, \\
\text{S} &= \lambda xyz.xz(yz), & \text{S-combinator from Combinatory Logic}, \\
\Delta &= \lambda x.xx, & \text{self-application}, \\
\Omega &= \Delta \Delta, & \text{paradigmatic looping combinator}, \\
\text{Y} &= \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)), & \text{Curry’s fixed point combinator}.
\end{align*}
\]

\[2\] This basically means that parentheses are left implicit.
The λ-calculus is given by the set Λ endowed with reduction relations that turn it into a higher-order term rewriting system.

We say that a relation \( R \subseteq \Lambda^2 \) is compatible if it is compatible w.r.t. application and λ-abstraction. This means that, for \( M, N, P \in \Lambda \), if \( M R N \) holds then also \( MP R NP \), \( PM R PN \) and \( \lambda x.M R \lambda x.N \) hold.

**Definition 1.4.** Define the following reduction relations.

(i) The \( \beta \)-reduction \( \rightarrow_{\beta} \) is the least compatible relation closed under the rule
\[
(\lambda x.M)N \rightarrow M[N/x]
\]
where \( M[N/x] \) denotes the λ-term obtained by substituting \( N \) for all free occurrences of \( x \) in \( M \), subject to the usual proviso about renaming bound variables in \( M \) to avoid capture of free variables in \( N \).

(ii) Similarly, the \( \eta \)-reduction \( \rightarrow_{\eta} \) is the least compatible relation closed under the rule
\[
\lambda x.Mx \rightarrow M, \text{ if } x \notin \text{FV}(M).
\]

(iii) Moreover, we define \( \rightarrow_{\beta\eta} = \rightarrow_{\beta} \cup \rightarrow_{\eta} \).

(iv) The relations \( \rightarrow_{\beta}, \rightarrow_{\eta} \) and \( \rightarrow_{\beta\eta} \) respectively generate the notions of multi-step reduction \( \rightarrow_{\beta}, \rightarrow_{\eta}, \rightarrow_{\beta\eta} \) (resp. conversion \( =_{\beta}, =_{\eta}, =_{\beta\eta} \)) by taking the reflexive and transitive (and symmetric) closure.

**Theorem 1.5 (Church-Rosser).** The reduction relation \( \rightarrow_{\beta(\eta)} \) is confluent:

\[
M \rightarrow_{\beta(\eta)} M_1 \land M \rightarrow_{\beta(\eta)} M_2 \Rightarrow \exists N \in \Lambda . M_1 \rightarrow_{\beta(\eta)} N \beta(\eta) =_{\beta(\eta)} M_2
\]

The λ-terms are classified into solvable and unsolvable, depending on their capability of interaction with the environment.

**Definition 1.6.** A λ-term \( M \) is called solvable if \( (\lambda \vec{x}.M)\vec{P} =_{\beta} \mathbf{I} \) for some \( \vec{x} \) and \( \vec{P} \in \Lambda \). Otherwise \( M \) is called unsolvable.

We say that a λ-term \( M \) has a head normal form (hnf) if it reduces to a λ-term of shape \( \lambda x_1 \ldots x_n . y M_1 \ldots M_k \) for some \( n, k \geq 0 \). As shown by Wadsworth in [13], a λ-term \( M \) is solvable if and only if \( M \) has a head normal form. The typical examples of unsolvable λ-terms are \( \Omega, \lambda x.\Omega \) and \( \text{YI} \).
1.2. Lambda theories and lambda models

Conservative extensions of $\beta$-conversion are known as “$\lambda$-theories” and have been extensively studied in the literature, see e.g. [2, 14, 15, 16, 17].

Definition 1.7.

(i) A $\lambda$-theory $\mathcal{T}$ is any congruence on $\Lambda^2$ including $\beta$-conversion $=_{\beta}$.

(ii) A $\lambda$-theory $\mathcal{T}$ is called:

- consistent, if $\mathcal{T}$ does not equate all $\lambda$-terms;
- inconsistent, if $\mathcal{T}$ is not consistent;
- extensional, if $\mathcal{T}$ contains the $\eta$-conversion $=_{\eta}$ as well;
- sensible, if $\mathcal{T}$ is consistent and equates all unsolvable $\lambda$-terms;
- semi-sensible, if $\mathcal{T}$ does not equate a solvable and an unsolvable.

We write $\mathcal{T} \vdash M = N$, or simply $M =_{\mathcal{T}} N$, whenever $(M, N) \in \mathcal{T}$.

The set of all $\lambda$-theories, ordered by inclusion, forms a quite rich complete lattice. We denote by $\lambda$ (resp. $\lambda\eta$) the smallest (resp. extensional) $\lambda$-theory. Both $\lambda$ and $\lambda\eta$ are consistent, semi-sensible but not sensible. A $\lambda$-theory can be introduced syntactically, or semantically as the theory of a model. The model theory of $\lambda$-calculus is largely based on the notion of combinatory algebras, and its variations (see, e.g., [5, 15, 16, 19] and [2, Ch. 5]).

Definition 1.8.

(i) An applicative structure is given by $\mathcal{A} = (A, \cdot)$ where $A$ is a set and $(\cdot)$ is a binary operation on $A$ called application. We represent application as juxtaposition and we assume it is left-associative, e.g., $abc = (a \cdot b) \cdot c$. An equivalence $\simeq$ on $\mathcal{A}$ is a congruence if it is compatible w.r.t. application:

$$a \simeq a' \land b \simeq b' \Rightarrow ab \simeq a'b'$$

(ii) A combinatory algebra $\mathcal{C} = (C, \cdot, k, s)$ is an applicative structure for a signature with two constants $k, s$, such that $k \neq s$ and $(\forall x, y, z \in \text{Var})$:

$$kxy = x, \text{ and } sxyz = xz(yz).$$

We say that $\mathcal{C}$ is extensional if the following holds:

$$\forall x.\forall y.(\forall z.(xz = yz) \Rightarrow x = y)$$
(iii) Given a combinatory algebra $C$ and a congruence $\simeq$ on $(C, \cdot)$, define:

$$C_{\simeq} = (C / \simeq, \cdot_{\simeq}, k_{\simeq}, s_{\simeq})$$

where $[a]_{\simeq} \cdot [b]_{\simeq} = [a \cdot b]_{\simeq}$, $k_{\simeq} = [k]_{\simeq}$ and $s_{\simeq} = [s]_{\simeq}$. It is easy to check that if $k \not\simeq s$ then $C_{\simeq}$ is a combinatory algebra.

We call $k$ and $s$ the basic combinator; the derived combinators $i$ and $\varepsilon$ are defined by $i = skk$ and $\varepsilon = s(ki)$. It is not difficult to verify that every combinatory algebra satisfies the identities $ix = x$ and $\varepsilon xy = xy$.

It is well-known that combinatory algebras are models of combinatory logic. A $\lambda$-term $M$ can be interpreted in any combinatory algebra $C$ by first translating $M$ into a term $X$ of combinatory logic, written $(M)_{CL} = X$, and then interpreting the latter in $C$. However, there might be $\beta$-convertible $\lambda$-terms $M, N$ that are interpreted as distinguished elements of $C$. For this reason, not all combinatory algebras are actually models of $\lambda$-calculus.

The axioms of an elementary subclass of combinatory algebras, called $\lambda$-models, were expressly chosen to make coherent the definition of interpretation of $\lambda$-terms (see [2, Def. 5.2.1]). The Meyer-Scott axiom is the most important axiom in the definition of a $\lambda$-model. In the first-order language of combinatory algebras it becomes:

$$\forall x. \forall y. (\forall z. (xz = yz) \Rightarrow \varepsilon x = \varepsilon y).$$

The combinator $\varepsilon$ becomes an inner choice operator, that makes coherent the interpretation of an abstraction $\lambda$-term.

1.3. Syntactic $\lambda$-models

The definition of a $\lambda$-model is difficult to handle in practice because the five Curry’s axioms [2, Thm. 5.2.5] are complicated to verify by hand. To prove that a certain combinatory algebra is actually a $\lambda$-model, it is preferable to exploit Hindley’s (equivalent) notion of a syntactic $\lambda$-model. See, e.g., [5].

The definition of syntactic $\lambda$-model in [5] is general enough to interpret $\lambda$-terms possibly containing constants $\hat{a}$ representing elements $a$ of a set $A$. We follow that tradition and denote by $\Lambda(A)$ the set of all $\lambda$-terms possibly containing constants $\hat{a}$ representing elements $a$ of a set $A$. For instance, given $a \in A$, we have $M = I(\lambda x.x\hat{a})\hat{b} \in \Lambda(A)$. All notions, notations and results from Subsection 1.1 extend to $\lambda A$-terms without any problem. In particular, substitution is extended by setting $\hat{a}[N/x] = \hat{a}$, for all $a \in A$ and
\( N \in \Lambda(A) \). As an example, the \( \lambda A \)-term \( M \) above reduces as follows: \( M \rightarrow_{\beta} (\lambda x. x \hat{a}) \hat{b} \rightarrow_{\beta} \hat{b} \hat{a} \in \Lambda(A) \). Notice that substitutions of variables by constants always permute, namely \( M[\hat{a}/x][\hat{b}/y] = M[\hat{b}/y][\hat{a}/x] \), for all \( a, b \in A \).

Given a set \( A \), a valuation in \( A \) is any map \( \rho : \text{Var} \rightarrow A \). We write \( \text{Val}_A \) for the set of all valuations in \( A \). Given \( \rho \in \text{Val}_A \) and \( a \in A \), define:

\[
(\rho[x := a])(y) = \begin{cases} 
  a, & \text{if } x = y, \\
  \rho(y), & \text{otherwise}.
\end{cases}
\]

**Definition 1.9.** A syntactic \( \lambda \)-model is a tuple \( S = (A, \cdot, [\cdot]) \) such that \((A, \cdot)\) is an applicative structure and the interpretation function

\[
[\cdot] : \Lambda(A) \times \text{Val}_A \rightarrow A
\]

satisfies

(i) \( [x]_\rho = \rho(x) \), for all \( x \in \text{Var} \);

(ii) \( [\hat{a}]_\rho = a \), for all \( a \in A \);

(iii) \( [PQ]_\rho = [P]_\rho \cdot [Q]_\rho \);

(iv) \( [\lambda x. P]_\rho \cdot a = [P]_\rho[x := a] \), for all \( a \in A \);

(v) \( \forall x \in \text{FV}(M) . \rho(x) = \rho'(x) \Rightarrow [M]_\rho = [M]_{\rho'} \);

(vi) \( \forall a \in A . [M]_{\rho[x := a]} = [N]_{\rho[x := a]} \Rightarrow [\lambda x. M]_\rho = [\lambda x. N]_\rho \).

If \( M \in \Lambda^o \), then \( [M]_\rho \) is independent from the valuation \( \rho \) and we simply write \( [M] \). We write \( S \models M = N \) if and only if \( \forall \rho \in \text{Val}_A . [M]_\rho = [N]_\rho \) holds. It is easy to check that \( \lambda \vdash M = N \) entails \( S \models M = N \).

The \( \lambda \)-theory induced by \( S \) is defined as follows:

\[
\text{Th}(S) = \{ M = N \mid S \models M = N \}.
\]

The precise correspondence between \( \lambda \)-models and syntactic \( \lambda \)-models is described in [2], Theorem 5.3.6. For our purposes, it is enough to know that if \( S \) is a syntactic \( \lambda \)-model then \( C_S = (A, \cdot, [K], [S]) \) is a \( \lambda \)-model. We say that \( S \) is extensional whenever \( C_S \) is extensional as a combinatory algebra. This holds if and only if \( \text{Th}(S) \) is extensional if and only if \( S \models I = 1 \).
2. Addressing Machines

In this section we introduce the notion of an *Addressing Machine*. We first provide some intuitions, then we proceed with the formal description of such machines. The general structure of an addressing machine is composed by two substructures:

- the *internal components*, organized as follows:
  - a finite number of *internal registers*;
  - an *internal program*.

- the *input-tape*.

As the name suggest, the addressing mechanism is central in this formalism. Each addressing machine is associated with an address, receives a list of addresses in its input-tape and is able to transfer the computation to another machine by calling its address, possibly extending its input-tape.

2.1. Tapes, Registers and Programs

We consider fixed a countable set $A$ of *addresses*, together with a constant $\emptyset \notin A$ that we call “null” and corresponds to an uninitialized register.

**Definition 2.1.** We let $A_\emptyset = A \cup \{\emptyset\}$.

(i) An $A_\emptyset$-valued tape $T$ is a finite ordered list of addresses $T = [a_1, \ldots, a_n]$ with $a_i \in A$ for all $i \leq n$. We write $T_A$ for the set of all $A_\emptyset$-valued tapes.

(ii) Let $a \in A$ and $T, T' \in T_A$. We denote by $a :: T$ the tape having $a$ as first element and $T$ as tail. We write $T @ T'$ for the concatenation of $T$ and $T'$, which is an $A_\emptyset$-valued tape itself.

(iii) Given an index $i \in \mathbb{N}$, an $A_\emptyset$-valued register $R_i$ is a memory-cell capable of storing either $\emptyset$ or an address $a \in A$.

(iv) Given $A_\emptyset$-valued registers $R_0, \ldots, R_n$ for $n \geq 0$, an address $a \in A$ and an index $i \in \mathbb{N}$, we write $\vec{R}[R_i := a]$ for the registers $\vec{R}$ where the value of $R_i$ has been updated:

$$R_0, \ldots, R_{i-1}, a, R_{i+1}, \ldots, R_n$$

Notice that, whenever $i > n$, we assume that $\vec{R}[R_i := a] = \vec{R}$. 

Addressing machines can be seen as having a RISC architecture, since their internal program is composed by only three instructions. We describe the effects of these basic operations on a machine having \( r_0 \) internal registers \( R_0, \ldots, R_{r_0-1} \). Therefore, when we say “if an internal register \( R_i \) exists” we mean that the condition \( 0 \leq i < r_0 \) is satisfied. In the following, \( i, j, k \in \mathbb{N} \) correspond to indices of internal registers:

- **Load** \( i \): corresponds to the action of reading the first element \( a \) from the input-tape \( T \), and writing \( a \) on the internal register \( R_i \).

  The *precondition* to execute the operation is that the input-tape is non-empty, namely \( T = a :: T' \); the *postconditions* are that \( R_i \), if it exists, contains the address \( a \) and the input-tape of the machine becomes \( T' \). (If \( R_i \) does not exist, i.e. when \( i \geq r_0 \), the content of \( \vec{R} \) is unchanged.)

- **\( k \leftarrow \text{App}(i, j) \)**: corresponds to the action of reading the contents of \( R_i \) and \( R_j \), calling an external *application map* on the corresponding addresses \( a_1, a_2 \), and writing the result in the internal register \( R_k \), if it exists.

  The *precondition* is that \( R_i, R_j \) exist and are initialized, i.e. \( R_i, R_j \neq \emptyset \). The *postcondition* is that \( R_j \), if it exists, contains the address of the machine of address \( a_1 \) whose input-tape has been extended with \( a_2 \).

- **Call** \( i \): transfers the computation to the machine having as address the value stored in \( R_i \).

  The *precondition* is that \( R_i \) exists and is initialized. The *postcondition* is that the machine having the address stored in \( R_i \) is executed.

In the following we define what is a syntactically valid program of this language, and introduce a decision procedure for verifying that the preconditions of each instruction are satisfied when it is executed. As we will see in Lemma 2.5, these properties are decidable and statically verifiable. As a consequence, addressing machines will never give rise to an error at run-time.

**Definition 2.2.**

(i) A *program* \( P \) is a finite list of instructions generated by the following grammar (\( \varepsilon \) represents the empty string, \( i, j, k \in \mathbb{N} \)):

\[
\begin{align*}
P & ::= \text{Load } i; P \mid A \\
A & ::= k \leftarrow \text{App}(i, j); A \mid C \\
C & ::= \text{Call } i \mid \varepsilon
\end{align*}
\]
In other words a program starts with a list of Load’s, continues with a list of App’s and possibly ends with a Call. Each of these lists may be empty, in particular the empty-program ε can be generated.

(ii) Given a program \( P \), an \( r_0 \in \mathbb{N} \), and a set \( I \subseteq \{0, \ldots, r_0 - 1\} \) of indices, define the relation \( I \vdash_{r_0} P \) as the least relation closed under the rules:

\[
\begin{align*}
I \cup \{ i \} \vdash_{r_0} P & \quad i < r_0 \\
I \vdash_{r_0} \text{Load } i; P & \\
I \vdash_{r_0} \text{Call } i & \\
I \cup \{ k \} \vdash_{r_0} A & \quad i, j \in I \\
I \vdash_{r_0} k \leftarrow \text{App}(i, j); A & \\
I \vdash_{r_0} \varepsilon &
\end{align*}
\]

(iii) Let \( r_0 \in \mathbb{N} \) and \( \vec{R} = R_0, \ldots, R_{r_0-1} \) be \( A_\omega \)-valued registers. We say that a program \( P \) is valid with respect to \( \vec{R} \) whenever \( R \vdash_{r_0} P \) holds for

\[
R = \{ i \mid R_i \neq \emptyset \land 0 \leq i < r_0 \}
\]

Examples 2.3. Consider addresses \( a_1 = 0x00A375, a_2 = 0x010FC2 \in A \), as well as \( A_\omega \)-valued registers \( R_0 = \emptyset, R_1 = a_1, R_2 = a_2, R_3 = \emptyset \) (so \( r_0 = 4 \)). In this example, the set \( R \) of initialized registers as defined in (1) is \( R = \{1, 2\} \).

<table>
<thead>
<tr>
<th>( P_n )</th>
<th>Program</th>
<th>( R \vdash_{r_0} P_n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_0 = )</td>
<td>Load 0; 2 ← App(0, 1); Call 2</td>
<td>✓</td>
</tr>
<tr>
<td>( P_1 = )</td>
<td>0 ← App(1, 2); 3 ← App(0, 2); Call 3</td>
<td>✓</td>
</tr>
<tr>
<td>( P_2 = )</td>
<td>Load 23; Load 0; Call 0</td>
<td>✓</td>
</tr>
<tr>
<td>( P_3 = )</td>
<td>Load 23; 23 ← App(1, 2); Call 2</td>
<td>✓</td>
</tr>
<tr>
<td>( P_4 = )</td>
<td>2 ← App(0, 1); Call 2</td>
<td>✗</td>
</tr>
<tr>
<td>( P_5 = )</td>
<td>Load 0; Call 3</td>
<td>✗</td>
</tr>
<tr>
<td>( P_6 = )</td>
<td>3 ← App(1, 2); Call 23</td>
<td>✗</td>
</tr>
</tbody>
</table>

Above we use “23” as an index of an unexisting register. Notice that a program trying to update an unexisting register remains valid (see \( P_2, P_3 \)). On the contrary, an attempt at reading the content of an uninitialized \( P_4, P_5 \) or unexisting \( P_6 \) register invalidates the whole program.

Notation 2.4. We use “−” to indicate an arbitrary index of an unexisting register. E.g., the program \( P_6 \) will be written 3 ← App(1, 2); Call −. We also write Load \( (i_1, \ldots, i_k) \) as an abbreviation for Load \( i_1; \cdots; \text{Load } i_k; \). By employing all these notations, \( P_2 \) can be written as \( P_2 = \text{Load } (−, 0); \text{Call } 0. \)
Lemma 2.5. For all $A_\sigma$-valued registers $\bar{R}$ and program $P$ it is decidable whether $P$ is valid with respect to $\bar{R}$.

Proof. First, notice that the grammar in Definition 2.2(i) is right-linear, therefore it is decidable whether $P$ is a production. Also, $r_0 \in \mathbb{N}$ therefore $\mathcal{R}$ is finite and, since $P$ is also finite, the set $\mathcal{R}$ remains finite during the execution of $\mathcal{R} \vdash r_0 \ P$. Decidability follows from these properties, together with the fact that the first instruction of $P$ uniquely determines which rule from Definition 2.2(ii) should be applied (and these rules are exhaustive). \qed

2.2. Addressing machines and their operational semantics

Everything is in place to introduce the definition of an addressing machine. Thanks to Lemma 2.5 it is reasonable to require that an addressing machine has a valid internal program.

Definition 2.6.

(i) An addressing machine $M$ (with $r_0$ registers) over $A$ is given by a tuple:

$$M = \langle \bar{R}, P, T \rangle$$

where:

- $\bar{R} = R_0, \ldots, R_{r_0 - 1}$ are $A_\sigma$-valued registers;
- $P$ is a program valid w.r.t. $\bar{R}$;
- $T$ is an $A$-valued (input) tape.

(ii) We write $M.r_0$ for the number of registers of $M$, $M.R_i$ for its $i$-th register, $M.P$ for the associated program and finally $M.T$ for its input tape.

(iii) We say that an addressing machine $M$ as above is stuck, in symbols $\text{stuck}(M)$, whenever its program has shape $M.P = \text{Load } i; P$ but its input-tape is empty $M.T = []$. Otherwise, $M$ is not stuck: $\neg \text{stuck}(M)$.

(iv) The set of all addressing machines over $A$ will be denoted by $\mathcal{M}_A$.

The machines below will be used as running examples in the next sections. Intuitively, the addressing machines $K, S, I, D, O$ mimic the behavior of the $\lambda$-terms $K, S, I, \Delta$ and $\Omega$, respectively. For writing their programs, we adopt the conventions introduced in Notation 2.4.
Examples 2.7. The following are addressing machines.

(i) For every $n \in \mathbb{N}$, define an addressing machine with $n + 1$ registers as:

$$x_n = \langle R_0, \ldots, R_n, \varepsilon, [] \rangle$$

where $\vec{R} = \emptyset$

We call $x_0, x_1, x_2, \ldots$ indeterminate machines because they share some analogies with variables (they can be used as place holders).

(ii) The addressing machine $K$ with 1 register $R_0$ is defined by:

$$K = \langle \emptyset, \text{Load}(0, -); \text{Call} 0, [] \rangle$$

(iii) The addressing machine $S$ with 3 registers is defined by:

$$S = \langle \emptyset, \emptyset, \emptyset, P, [] \rangle$$

where:

$$S.P = \text{Load}(0, 1, 2); 0 \leftarrow \text{App}(0, 2); 1 \leftarrow \text{App}(1, 2); 2 \leftarrow \text{App}(0, 1); \text{Call} 2$$

(iv) Assume that $k \in A$ represents the address associated with the addressing machine $K$. Define the addressing machine $I$ as $I = \langle \emptyset^3, S.P, [k, k] \rangle$.

(v) The addressing machine $D$ with 1 register is given by:

$$D = \langle \emptyset, \text{Load} 0; 0 \leftarrow \text{App}(0, 0); \text{Call} 0, [] \rangle$$

(vi) Assume that $d \in A$ represents the address of the addressing machine $D$. Define the addressing machine $O$ by setting $O = \langle D.\vec{R}, D.P, [d] \rangle$.

We now enter into the details of the addressing mechanism which constitutes the core of this formalism.

Definition 2.8. Let us fix a bijective map $\# : \mathcal{M}_A \to A$ from the set of all addressing machines over $A$ to the set $A$ of addresses. We call the map $\#(\cdot)$ an Address Table Function (ATF).

(i) Given $M \in \mathcal{M}_A$, we say that $\#M$ is the address of $M$.

(ii) Given an address $a \in A$, we write $\#^{-1}(a)$ for the unique machine having address $a$. In other words, we have $\#^{-1}(a) = M \iff \#M = a$. 

15
(iii) Given $M \in \mathcal{M}_A$ and $T' \in T_A$, we write $M @ T'$ for the machine

$$\langle M, \bar{R}, M . P, M . T @ T' \rangle$$

(iv) Define the application map $(\cdot) : A \times A \to A$ as follows

$$a \cdot b = \#(\#^{-1}(a) @ [b])$$

That is, the application of $a$ to $b$ is the unique address $c$ of the addressing machine obtained by adding $b$ at the end of the input tape of the addressing machine $\#^{-1}(a)$.

Observe that, in a certain sense, an ATF share analogies with the Domain Name Service (DNS) protocol implemented in the TCP/IP stack. Indeed:

1. The DNS takes as input a logical description (a string) of, say, a server and returns, via a suitable table, the associated IP address;
2. an ATF takes as input a structure (a tuple) representing an addressing machine and returns its (unique) address.

**Definition 2.9** (Small step operational semantics). Define a reduction strategy on addressing machines representing one head-step of computation

$$\rightarrow_h \subseteq \mathcal{M}_A \to \mathcal{M}_A$$

as the least relation closed under the following rules:

$$\langle \bar{R}, \text{Load } i; P, a :: T \rangle \rightarrow_h \langle \bar{R}[R_i := a], P, T \rangle,$n$$

$$\langle \bar{R}, k \leftarrow \text{App}(i, j); P, T \rangle \rightarrow_h \langle \bar{R}[R_k := R_i \cdot R_j], P, T \rangle,$n$$

$$\langle \bar{R}, \text{Call } i; P, T \rangle \rightarrow_h \#^{-1}(R_i) @ T.$n$$

As usual, we write $\rightarrow \rightarrow_h$ for the transitive-reflexive closure of $\rightarrow_h$. We say that an addressing machine $M$ is in a final state if there is no $N$ such that $M \rightarrow_h N$. We write $M \rightarrow_h \text{stuck}(N)$ whenever $M \rightarrow_h N$ and stuck($N$) hold. When $N$ is not important, we simply write $M \rightarrow_h \text{stuck}()$. Similarly, $M \not\rightarrow_h \text{stuck}()$ means that $M$ never reduces to a stuck addressing machine.

**Remark 2.10.**

(i) It is easy to check that the operational semantics defined above are independent from the choice of the chosen ATF $\#(-)$.

(ii) Addressing machines in a final state are either of the form $\langle \bar{R}, \varepsilon, T \rangle$ or $\langle \bar{R}, \text{Load } i; P, [] \rangle$, and in the latter case they are stuck.
Lemma 2.11. The reduction strategy \( \rightarrow_h \) enjoys the following properties:

(i) Determinism: \( M \rightarrow_h N_1 \land M \rightarrow_h N_2 \Rightarrow N_1 = N_2 \).

(ii) Closure under application: \( \forall a \in A. M \rightarrow_h N \Rightarrow M \circ [a] \rightarrow_h N \circ [a] \).

Proof. (i) Since the applicable rule from Definition 2.9 if any, is uniquely determined by the first instruction on \( M.P \) and its input-tape \( M.T \).

(ii) By cases on the rule applied for deriving \( M \rightarrow_h N \). \( \square \)

Examples 2.12. For brevity, we sometimes display only the first instruction of the internal program. Take \( a, b, c \in A \).

(i) We show that \( K \) behaves as the first projection:

\[
K \circ [a,b] = \langle \emptyset, \text{Load } (0, -) \circ \text{Call } 0, [a,b] \rangle \\
\rightarrow_h \langle a, \text{Load } - \circ \text{Call } 0, [b] \circ \text{Call } 0, [] \rangle \rightarrow_h \#^{-1}(a).
\]

(ii) We verify that \( S \) behaves as the combinator \( S \) from combinatory logic:

\[
S \circ [a,b,c] = \langle \emptyset^3, \text{Load } (0,1,2) ; \cdots ; [a,b,c] \rangle \\
\rightarrow_h \langle a, b, c, 0 \leftarrow \text{App}(0,2) ; \cdots ; [] \rangle \\
\rightarrow_h \langle a \cdot c, b, c, 1 \leftarrow \text{App}(1,2) ; \cdots ; [] \rangle \\
\rightarrow_h \langle a \cdot c, b \cdot c, c, 2 \leftarrow \text{App}(0,1) ; \cdots ; [] \rangle \\
\rightarrow_h \langle a \cdot c, b \cdot c, (a \cdot c) \cdot (b \cdot c) \rangle \circ \text{Call } 2 ; \cdots ; [] \rangle \\
\rightarrow_h \#^{-1}((a \cdot c) \cdot (b \cdot c))
\]

(iii) As expected, \( I = S \circ [\#K, \#K] \) behaves as the identity:

\[
I \circ [a] = \langle \emptyset^3, \text{Load } (0,1,2) ; \cdots ; [\#K, \#K, a] \rangle \\
\rightarrow_h \langle \#K, \#K, a, 0 \leftarrow \text{App}(0,2) ; \cdots ; [] \rangle \\
\rightarrow_h \langle \#K \cdot a, \#K, a, 1 \leftarrow \text{App}(1,2) ; \cdots ; [] \rangle \\
\rightarrow_h \langle \#K \cdot a, \#K \cdot a, 2 \leftarrow \text{App}(0,1) ; \cdots ; [] \rangle \\
\rightarrow_h \langle \#K \cdot a, \#K \cdot a, \#K \cdot a \cdot (\#K \cdot a), \text{Call } 2 ; [] \rangle \\
\rightarrow_h K \circ [a, \#K \cdot a] \\
= \langle \emptyset, \text{Load } (0, -) ; \cdots ; [a, \#K \cdot a] \rangle \\
\rightarrow_h \langle a, \#K \cdot a, \text{Call } 0, [] \rangle \rightarrow_h \#^{-1}(a)
\]

(iv) Finally, we check that \( O \) gives rise to an infinite reduction sequence:

\[
O = \langle \emptyset, \text{Load } 0; 0 \leftarrow \text{App}(0,0) ; \text{Call } 0, [\#D] \rangle \\
\rightarrow_h \langle \#D, 0 \leftarrow \text{App}(0,0) ; \text{Call } 0, [] \rangle \\
\rightarrow_h \langle \#(D \circ [\#D], \text{Call } 0, []) \circ \text{Call } 0, [] \rangle \rightarrow_h D \circ [\#D] = O \rightarrow_h \cdots
\]

17
Similarly, we can define a big-step operational semantics relating an addressing machine $M$ with its final result (if any).

**Definition 2.13** (Big-step semantics). Define $M \Downarrow V$, where $M, V \in \mathcal{M}_A$ and $V$ is in a final state, as the least relation closed under the following rules:

- $\frac{M.P = \text{Load } i; P' \quad M.T = \emptyset}{M \Downarrow M}$ \hspace{1cm} (Stuck)
- $\frac{M.P = \varepsilon}{M \Downarrow M}$ \hspace{1cm} (End)
- $\frac{M.P = \text{Load } i; P' \quad M.T = a :: T' \quad \langle M.R_i := a, P', T' \rangle \Downarrow V}{M \Downarrow V}$ \hspace{1cm} (Load)
- $\frac{M.P = k \leftarrow \text{App}(i, j); P' \quad a = M.R_i \cdot M.R_j \quad \langle M.R_k := a, M.P', M.T \rangle \Downarrow V}{M \Downarrow V}$ \hspace{1cm} (App)
- $\frac{M.P = \text{Call } i \quad M' = \#^{-1}(M.R_i) \quad M' @ [M.T] \Downarrow V}{M \Downarrow V}$ \hspace{1cm} (Call)

**Example 2.14.** Recall that $K.P = \text{Load } (0, -); \text{Call } 0$. Notice that we cannot prove $K @ [a, b] \Downarrow \#^{-1}(a)$ for an arbitrary $a \in A$, as we need to ensure that the resulting machine is in a final state. For this reason, we will use indeterminate machines $x_1, x_2$ from Example [2.7](#).

We now show that the two operational semantics are equivalent on terminating computations. (For a detailed proof, see the technical appendix.)

**Proposition 2.15.** For $M, N \in \mathcal{M}_A$, the following are equivalent:

1. $M \rightarrow_h N \not\rightarrow_h$
2. $M \Downarrow N$

**Proof.** (1 $\Rightarrow$ 2) By induction on the length of $M \rightarrow_h N$.
(2 $\Rightarrow$ 1) By induction on a derivation of $M \Downarrow N$. \qed
3. Combinatory Algebras via Evaluation Equivalence

In this section we show how to construct a combinatory algebra based on the addressing machines formalism. Recall that the addressing machines \( K \) and \( S \) have been defined in Example 2.7. Consider the algebraic structure

\[
\mathcal{A} = (\mathbb{A}, \#K, \#S, \cdot)
\]

Since the application \((\cdot)\) is total, \(\mathcal{A}\) is an applicative structure. However, it is not a combinatory algebra. For instance, the \(\lambda\mathbb{A}\)-term \(\hat{K} \hat{a} \hat{b}\) is interpreted as the address of the machine \(K@[a,b]\), which is \textit{a priori} different from the address "\(a\)" because no computation is involved. Therefore, we need to quotient the algebra \(\mathcal{A}\) by an equivalence relation equating at least all addresses corresponding to the same machine at different stages of the execution.

**Definition 3.1.** Every binary relation \(\equiv R \subseteq \mathcal{M}^2\mathbb{A}\) on addressing machines induces a relation \(\simeq R \subseteq \mathbb{A}^2\) defined by

\[
a \simeq R b \iff \#^{-1}(a) \equiv R \#^{-1}(b)
\]

which is then extended to:

(i) \(\mathbb{A}_{\#}\)-valued registers:

\[
R \simeq R' \iff (R = \emptyset = R') \lor (R = a \simeq R b = R')
\]

(ii) Tuples:

\[
a_1, \ldots, a_n \simeq R b_1, \ldots, b_m \iff (n = m) \land (\forall i \in \{1, \ldots, n\}.a_i \simeq R b_i)
\]

(This also applies to tuples of \(\mathbb{A}_{\#}\)-valued registers \(\vec{R} \simeq R \vec{R}\).)

(iii) \(\mathbb{A}\)-valued tapes:

\[
[a_1, \ldots, a_n] \simeq R [b_1, \ldots, b_m] \iff \vec{a} \simeq R \vec{b} \text{ (seen as tuples)}.
\]

Moreover, given two machines \(M, N \in \mathcal{M}_{\mathbb{A}}\), we define \(= R \subseteq \mathcal{M}^2\mathbb{A}\) by

\[
M = R N \iff (M.\vec{R} \simeq R N.\vec{R}) \land (M.\bar{P} = N.\bar{P}) \land (M.\bar{T} \simeq R N.\bar{T})
\]

In particular, \(M = R N\) entails that \(M\) and \(N\) share the same internal program, the number of internal registers, and the length of their input tape.

**Lemma 3.2.** If the relation \(\equiv R\) is an equivalence then so are \(\simeq R\) and \(= R\).

Proof. Easy.
Definition 3.3. Define \( \equiv_{\Lambda} \subseteq \mathcal{M}_{\Lambda}^2 \) as the least equivalence closed under:

\[
\frac{M \twoheadrightarrow h Z =_{\Lambda} N}{M \equiv_{\Lambda} N} \quad (\rightarrow_{\Lambda})
\]

We say that \( M, N \) are evaluation equivalent whenever \( M \equiv_{\Lambda} N \). Reflexivity can be treated as a special case of the rule \((\rightarrow_{\Lambda})\) since \( M \xrightarrow{\Lambda} Z =_{\Lambda} M \). Moreover, it follows from the definition that \( =_{\Lambda} \subseteq \equiv_{\Lambda} \) and that \( M \xrightarrow{\Lambda} Z =_{\Lambda} N \) entails \( M \equiv_{\Lambda} N \).

Examples 3.4. From the calculations in Examples 2.12, it follows that

\[
\begin{align*}
K @ [#x_1, #x_2] & \equiv_{\Lambda} x_1, \\
S @ [#x_1, #x_2, #x_3] & \equiv_{\Lambda} (x_1 @ [#x_3]) @ [#(x_2 @ [#x_3])].
\end{align*}
\]

Lemma 3.5. The relation \( \simeq_{\Lambda} \) is a congruence on \( A = (\Lambda, \#K, \#S, \cdot) \).

Proof. By definition \( \equiv_{\Lambda} \) is an equivalence, whence so is \( \simeq_{\Lambda} \) by Lemma 3.2. Let us check that \( \simeq_{\Lambda} \) is compatible w.r.t. \( \cdot \). Consider \( a \simeq_{\Lambda} a' \) and \( b \simeq_{\Lambda} b' \). Call \( M = \#^{-1}(a) \) and \( N = \#^{-1}(b) \) and proceed by induction on a derivation of \( M \equiv_{\Lambda} N \), splitting into cases depending on the last applied rule.

\((\rightarrow_{\Lambda})\) By definition, there exists \( Z \in \mathcal{M}_{\Lambda} \) such that \( M \xrightarrow{\Lambda} Z =_{\Lambda} N \). By Lemma 2.11(ii), \( M @ [b] \xrightarrow{\Lambda} Z @ [b] =_{\Lambda} N @ [b] \) whence \( a \cdot b \simeq_{\Lambda} a' \cdot b' \).

(Transitivity) and (Symmetry) follow from the induction hypothesis. \( \square \)

In order to prove that the congruence \( \simeq_{\Lambda} \) is non-trivial, we are going to characterize the equivalence \( M \equiv_{\Lambda} N \) it in terms of confluent reductions. For this purpose, we extend \( \xrightarrow{\Lambda} \) in such a way that reductions are also possible within registers and elements of the input-tape of an addressing machine.

Definition 3.6. Define the reduction relation \( \rightarrow_{c} \subseteq \mathcal{M}_{\Lambda}^2 \) as the least relation containing \( \rightarrow_{h} \) and closed under the following rules:

\[
\begin{align*}
\frac{R_i = a \in \Lambda \quad 0 \leq i < r_0 \quad \#^{-1}(a) \rightarrow_{c} M}{\langle R_0, \ldots, R_{r_0-1}, P, T \rangle \rightarrow_{c} \langle \hat{R}[R_i := \#M], P, T \rangle} \quad (\rightarrow_{c}^{R}) \\
\frac{0 \leq i \leq n \quad \#^{-1}(a_i) \rightarrow_{c} M}{\langle \hat{R}, P, [a_0, \ldots, a_n] \rangle \rightarrow_{c} \langle \hat{R}, P, [a_0, \ldots, a_{i-1}, \#M, a_{i+1}, \ldots, a_n] \rangle} \quad (\rightarrow_{c}^{T})
\end{align*}
\]

We write \( M \rightarrow_{i} N \) if \( N \) is obtained from \( M \) by directly applying one of the above rules — this is called an inner step of computation. The transitive and reflexive closure of \( \rightarrow_{c} \) and \( \rightarrow_{i} \) are denoted by \( \rightarrow_{c}^{*} \) and \( \rightarrow_{i}^{*} \), respectively.
Lemma 3.7 (Postponement of inner steps).
For $M, N, N' \in M_A$, if $M \rightarrow_i N \rightarrow_h N'$ then there exists $M' \in M_A$ such that $M \rightarrow_h M' \rightarrow_i N'$. In diagrammatic form:

$\begin{array}{c}
\xymatrix{M \ar[r]^i \ar[d]_h & N \\
M' \ar[r]_i & N'}
\end{array}$

Proof. By cases analysis over $M \rightarrow_i N$. The only interesting case is when the contracted redex is duplicated in $N \rightarrow_h N'$, namely:

Case $M = (\tilde{R}[R_i := a], P, T)$, $N = (\tilde{R}[R_i := b], P, T)$ with $M.P = N.P = k \leftarrow \text{App}(i, j); P'$ and $\#^{-1}(a) \rightarrow_c \#^{-1}(b)$. Assume $i \neq k < M.r_0$ and $i = j$, the other cases being easier. In this case $M' = (\tilde{R}[R_i := a][R_k := a \cdot a], P, T)$, therefore we need 3 inner steps to close the diagram:

$M' \rightarrow_i (\tilde{R}[R_i := b][R_k := a \cdot a], P, T)$
$\rightarrow_i (\tilde{R}[R_i := b][R_k := b \cdot a], P, T)$
$\rightarrow_i (\tilde{R}[R_i := b][R_k := b \cdot b], P, T) = N'$. \(\square\)

Morally, the term rewriting system $(M_A, \rightarrow_c)$ is orthogonal because $(i)$ the reduction rules defining $\rightarrow_c$ are non-overlapping as $\rightarrow_h$ is deterministic, $(\rightarrow_i^R)$ reduces a register and $(\rightarrow_i^T)$ reduces one element of the tape; $(ii)$ the terms on the left-hand side of the arrow are linear, as no equality among subterms is required. Now, it is well-known that orthogonal TRS are confluent, but one cannot apply [20, Thm.4.3.4] directly since we are not exactly dealing with first-order terms (because of the presence of the encoding).

Proposition 3.8. The reduction $\rightarrow_c$ is confluent.

Proof sketch. The Parallel Moves Lemma, which is the key property for proving [20, Thm. 4.3.4] generalizes easily. The rest of the proof follows. \(\square\)

Lemma 3.9. Let $M, N \in M_A$. Then $M \rightarrow_c N$ entails $M \equiv_A N$.

Proof. By induction on the length $n$ of the reduction $M \rightarrow_c N$, the case $n = 0$ being trivial (by reflexivity). Assume $n > 0$ and split into cases depending on the rule applied in the first step $M \rightarrow_c M'$. If it is a head-step, then we have $M \rightarrow_h M' =_A M'$ by reflexivity of $=_A$. If it is an inner-step, it follows by induction hypothesis that $M =_A M'$, so we conclude because $=_A \subseteq \equiv_A$. \(\square\)
Theorem 3.10. For $M, N \in \mathcal{M}_A$, we have:

$$M \equiv_A N \iff \exists Z \in \mathcal{M}_A . M \rightarrow_c Z \leftarrow_c N$$

Proof. ($\Rightarrow$) By induction on a derivation of $M \equiv_A N$.

($\rightarrow_A$) Assume that $M \rightarrow_A N$. From $Z \equiv_A N$ we get that $Z.r_0 = N.r_0$, $Z.R \simeq_A N.R$, $Z.P = N.P$ and $Z.T \simeq_A N.T$. Note that $Z.R_i = \emptyset$ iff $N.R_i = \emptyset$. Let us call $\mathcal{R}$ the set of indices $i$ of, say, $Z$ such that $Z.R_i \neq \emptyset$. By assumption, for every $i \in \mathcal{R}$, we have $Z.R_i = a_i, N.R_i = a'_i$ for $a_i \simeq_A a'_i$. Equivalently, $\#^{-1}(a_i) \equiv_A \#^{-1}(a'_i)$ holds and its derivation is smaller than $M \equiv_A N$. By induction hypothesis, they have a common reduct $\#^{-1}(a_i) \rightarrow_c X_i \leftarrow_c \#^{-1}(a'_i)$. Similarly, calling $Z.T = [b_1, \ldots, b_n]$ and $N.T = [b'_1, \ldots, b'_m]$ we must have $m = n$ and $b_j \simeq_A b'_j$ whence the induction hypothesis gives a common reduct $\#^{-1}(b_j) \rightarrow_c Y_j \leftarrow_c \#^{-1}(b'_j)$. Putting all reductions together, we conclude:

$$M \rightarrow_A Z \rightarrow_c \langle Z.R_i[\#X_i]_i \in \mathcal{R}, Z.P_i[\#Y_1, \ldots, \#Y_n] \rangle \leftarrow_c N$$

(Transitivity) By induction hypothesis and confluence (Proposition 3.8).

(Symmetry) Straightforward from the induction hypothesis.

($\Leftarrow$) By Lemma 3.9 we get $M \equiv_A Z$ and $N \equiv_A Z$, so we conclude by symmetry and transitivity. \qed

Proposition 3.11. $\mathcal{A}_{\equiv_A}$ is a non-extensional combinatory algebra.

Proof. From the calculations in Example 3.4 it follows that $\#K \cdot a \cdot b \simeq_A a$ and $\#S \cdot a \cdot b \cdot c \simeq_A (a \cdot c) \cdot (b \cdot c)$ hold, for all $a, b, c \in A$. Notice that both addressing machine $K$ and $S$ are stuck, and $K \neq_A S$ since, e.g., $K.r_0 \neq S.r_0$. By Theorem 3.10 we get $\#K \neq_A \#S$, whence $\mathcal{A}_{\equiv_A}$ is a combinatory algebra.

To check that it is not extensional, consider a different implementation of the combinator $K$, namely $K' = \langle \emptyset, \emptyset; \text{Load} \ (0,1); \text{Call} \ 0, [] \rangle$. For all $a, b \in A$, easy calculations give $\#K' \cdot a \cdot b \simeq_A a$. Thus, for all $a \in A$, we have $\#K \cdot a \cdot b \simeq_A a \simeq_A \#K' \cdot a \cdot b$, but $\#K \cdot a \neq_A \#K' \cdot a$. Also in this case, the two addressing machines are both stuck and $\#K \cdot a \neq_A \#K' \cdot a$, because $1 = (K \oplus [a]).r_0 \neq (K' \oplus [a]).r_0 = 2$. We conclude by Theorem 3.10. \qed

Lemma 3.12. The combinatory algebra $\mathcal{A}_{\equiv_A}$ is not a $\lambda$-model.

Proof. We need to find $M, N \in \Lambda$ satisfying $M =_{\beta} N$, while $\mathcal{A}_{\equiv_A} \not\models M = N$. Take $M = \lambda z.(\lambda x.x)z =_{\text{CL}} S(KI)I$ and $\lambda x.x =_{\text{CL}} I$ where $I = \text{SKK}$.
Recall that $I = S @ [#K, #K]$. Easy calculations give:

$$S @ [#K \cdot #I, #I] \rightarrow_h (\emptyset, \emptyset, \emptyset, \text{Load } 0; \cdots, [#K \cdot #I, #I])$$

$$\rightarrow_h \text{stuck}((#K \cdot #I, #I, \emptyset; \cdots, [[]]))$$

Similarly, $I = S @ [#K, #K] \rightarrow_h \text{stuck}((#K, #K, \emptyset; \cdots, [[]]))$. These two machines are both stuck and different modulo $\equiv_A$ since, e.g., the contents of their register $R_1$ are $#I$ and $#K$ respectively, and it is easy to check that $#I \not\equiv_A #K$. By Theorem 3.10, we conclude that $#I \not\equiv_A #S \cdot #K \cdot #K$.  

\section{4. Lambda Models via Applicative Equivalences}

In the previous section we have seen that the equivalence $\simeq_A$, thus $\equiv_A$, is too weak to give rise to a model of $\lambda$-calculus (Lemma 3.12). The main problem is that a $\lambda$-term $\lambda x.M$ is represented as an addressing machine performing a “Load” (to read $x$ from the tape) before evaluating the addressing machine corresponding to $M$. Since nothing is applied, the tape is empty and the machine gets stuck thus preventing the evaluation of the subterm $M$. In order to construct a $\lambda$-model we introduce the equivalence $\simeq^a_A$ below.

**Definition 4.1.** Define the relation $\equiv^a_A$ as the least equivalence satisfying:

$$\frac{M \rightarrow_h Z \equiv^a_A N}{M \equiv^a_A N} \quad (\Rightarrow^a_A)$$

$$\frac{M \rightarrow_h \text{stuck}(M')} \quad N \rightarrow_h \text{stuck}(N') \quad \forall a \in A. \frac{M @ [a] \equiv^a_A N @ [a]}{M \equiv^a_A N} \quad (a)$$

We say that $M$ and $N$ are *applicatively equivalent* whenever $M \equiv^a_A N$. Recall that $\simeq^a_A$ and $=^a_A$ are defined in terms of $\equiv^a_A$ as described in Definition 3.1. Also in this case, it is easy to check that $=^a_A \subseteq \equiv^a_A$ holds.

**Remark 4.2.** The rule $(a)$ shares similarities with the $(\omega)$-rule in $\lambda$-calculus [2, Def. 4.1.10], although being more restricted as only applicable to addressing machine that eventually become stuck. In particular, both rules have countably many premises, therefore a derivation of $M \equiv^a_A N$ is a well-founded $\omega$-branching tree (in particular, the tree is countable and there are no infinite paths). Techniques for performing induction “on the length of a derivation” in this kind of systems are well-established, see e.g. [10, 21]. More details about the underlying ordinals will be given in Section 5
Examples 4.3. Convince yourself of the following facts.

(i) As seen in the proof of Lemma 3.12, 1 and S @ [#K · #1, #K] both reduce to stuck machines. For all a ∈ A, we have that 1[a] →h #⁻¹(a) h ← S @ [#K · #1, #K, a]. By (æ), they are applicatively equivalent.

(ii) Since indeterminate machines x_k are not stuck, x_m ≡ æ A x_n entails m = n.

(iii) Let 1 = (∅², Load (0, 1); 0 ← App(0, 1); Call 0, []). It is easy to check that, for all a, b ∈ A, we have 1[a, b] →h #⁻¹(a) @ [b] h ← 1[a, b]. However, since 1[a, x_n] →h x_n and ¬stuck(x_n), one cannot apply (æ), whence (intuitively) they are not applicatively equivalent: 1 ̸≡ æ A 1.

Actually the inequalities claimed in examples (ii)-(iii) above, i.e. x_m ̸≡ æ A x_n for m ̸= n and 1 ̸≡ æ A 1, are difficult to prove formally (see Lemma 4.5(ii)).

Lemma 4.4. Let M, N ∈ M_Ä and a, b ∈ A.

(i) If M ≡ æ A N then M @ [a] ≡ æ A N @ [a].

(ii) The following rule is derivable:

\[
\frac{M \equiv æ A N \quad a \simeq æ A b} {M @ [a] \equiv æ A N @ [b]} \quad \text{(cong)}
\]

(iii) Therefore, ≃ æ A is a congruence on A = (A, ·, #K, #S).

Proof. (i) By induction on a proof of M ≡ æ A N. Possible cases are:

Case (↠ æ A). If M ↠ h Z ≡ æ A N then M @ [a] ↠ h Z @ [a] ≡ æ A N @ [a], by Lemma 2.11(ii) and the definition of ≡ æ A.

Case (æ). Trivial, as the thesis is a premise of this rule.

(Symmetry) and (Transitivity) follow from the induction hypothesis.

(ii) Assume that M ≡ æ A N and a ≃ æ A b. Then, we have:

\[
M @ [a] \quad M @ [b], \quad \text{by reflexivity and } a \simeq æ A b,
\]

\[
\equiv æ A N @ [b], \quad \text{by (i)}.
\]

So we conclude by transitivity.

(iii) By Lemma 3.2 ≃ æ A is an equivalence, by (ii) a congruence.

We need to show that the congruence ≃ æ A is non-trivial, and that the addresses of #K, #S remain distinguished modulo ≃ æ A.
Lemma 4.5. Let $M, N \in \mathcal{M}_\Lambda$.

(i) If $M \equiv \Lambda N$ then $M \equiv \Lambda N$.

(ii) If $M \equiv \Lambda N$ and $M \rightarrow^h x_n$ then $N \rightarrow^h x_n$.

(iii) Hence, the equivalence relation $\simeq \Lambda$ is non-trivial.

(iv) In particular, $\#K \not\equiv \Lambda \#S$.

Proof. (i) Easy.

(ii) This proof is the topic of Section 5.

(iii) By (i), the relation is non-empty. By (ii), $x_i \equiv \Lambda x_j$ if and only if $i = j$, whence there are infinitely many distinguished equivalence classes.

(iv) From Example 2.12, we get:

\[
K \times \#K, \#K, \#x_1 \rightarrow^h \langle \#x_1, \text{Load} - \text{Call} 0, \emptyset \rangle;
\]

\[
S \times \#K, \#K, \#x_1 \rightarrow^h x_1.
\]

For these machines to be $\equiv \Lambda$-equivalent, the former machine should reduce to $x_1$, by (ii), which is impossible since $\langle \#x_1, \text{Load} - \text{Call} 0, \emptyset \rangle$ is stuck. □

4.1. Constructing a $\lambda$-model

We define an interpretation transforming a $\lambda$-term with free variables $x_1, \ldots, x_n$ into an addressing machine reading the values of $\vec{x}$ from its tape. The definition is inspired from the well-known categorical interpretation of $\lambda$-calculus into a reflexive object of a cartesian closed category. In particular, variables are interpreted as projections. See, e.g., [5] or [18] for more details.

Definition 4.6 (Auxiliary interpretation). Let $M \in \Lambda(\Lambda)$ and $x_1, \ldots, x_n$ be such that $\text{FV}(M) \subseteq \vec{x}$. Define $| - |_{\vec{x}} : \Lambda(\Lambda) \rightarrow \mathcal{M}_\Lambda$ by induction as follows:

- $|x_i|_{\vec{x}} = \text{Pr}^n_i$;
- $|\hat{a}|_{\vec{x}} = \text{Cons}^n_a$, for $a \in \Lambda$;
- $|MN|_{\vec{x}} = \langle \emptyset^n, \# |M|_{\vec{x}}, \# |N|_{\vec{x}}, \emptyset, \text{Apply}_n, \emptyset \rangle$;
- $|\lambda y.M|_{\vec{x}} = |M|_{\vec{x}y}$, assuming wlog that $y \not\in \vec{x}$.
Remark 4.7. Let the following are basic properties of the interpretation map defined above.

See the Technical Appendix.

\[
\begin{align*}
\Pr^n_t &= \langle \emptyset, (\text{Load } -)^{i-1}; \text{Load } 0; (\text{Load } -)^{n-i-1}; \text{Call } 0, [] \rangle, \\
\Cons^n_a &= \langle a, (\text{Load } -)^n; \text{Call } 0, [] \rangle, \\
\Apply_n &= \text{Load } (0, \ldots, n-1); n \leftarrow \text{App}(n, 0); \ldots; n \leftarrow \text{App}(n, n-1); \\
&\quad n+1 \leftarrow \text{App}(n+1, 0); \ldots; n+1 \leftarrow \text{App}(n+1, n-1); \\
&\quad n+2 \leftarrow \text{App}(n+2, n+1); \text{Call } n+2.
\end{align*}
\]

Remark 4.7. Let \( n \in \mathbb{N} \), and \( T = [a_1, \ldots, a_n] \in T_A \). We have:

(i) \( \Pr^n_t \circ T \rightarrow_h \#^{-1}(a_i) \), for all \( i (1 \leq i \leq n) \);

(ii) \( \Cons^n_a \circ T \rightarrow_h \#^{-1}(b) \), for all \( b \in A \);

(iii) \( \langle \emptyset^n, \#^n, \emptyset, \Apply_n, T \rangle \rightarrow_h (M \circ T) \circ [\#(N \circ T)] \).

From now on, whenever writing \( |M|_\bar{x} \), we assume that \( \text{FV}(M) \subseteq \bar{x} \). The following are basic properties of the interpretation map defined above.

Lemma 4.8. Let \( M \in \Lambda(A) \), \( n \in \mathbb{N} \), \( \bar{x} = x_1, \ldots, x_n \) and \( \bar{a} = a_1, \ldots, a_n \in A \).

(i) \( |M|_\bar{x} = \langle \bar{R}, \text{Load } (i_1, \ldots, i_n); P, [] \rangle \) for some \( A_{\bar{x}} \)-valued registers \( \bar{R} \), program \( P \) and indices \( i_j \in \mathbb{N} \).

(ii) If \( m < n \) then \( |M|_{\bar{x}} @ [a_1, \ldots, a_m] \rightarrow_h \text{stuck}(N) \) for some \( N \in \mathcal{M}_A \).

(iii) For all \( b \in A \), we have \( |M|_{y, \bar{x}} @ [b] \equiv_A |M[b/y]|_\bar{x} \).

(iv) In particular, if \( y \notin \text{FV}(M) \) then \( |M|_{y, \bar{x}} @ [b] \equiv_A |M|_\bar{x} \).

(v) \( |M|_{\bar{x}} @ [\bar{a}] \equiv_A |M|_{x_{\sigma(1)}, \ldots, x_{\sigma(n)}} @ [a_{\sigma(1)}, \ldots, a_{\sigma(n)}] \) for all permutations \( \sigma \).

Proof. See the Technical Appendix. \( \square \)

Definition 4.9. Let \( \mathcal{S} = (A/\sim^e_A, \bullet, [-]_-) \), where

\[
\begin{align*}
[a]_{\sim^e_A} \bullet [b]_{\sim^e_A} &= [a \cdot b]_{\sim^e_A}, \\
[M]_\rho &\equiv_A \#(|M|_{\bar{x}} @ [\rho(x_1), \ldots, \rho(x_n)])
\end{align*}
\]

By Lemma 4.8, the definition of \( [M]_\rho \) is independent from the choice of \( \bar{x} \), as long as \( \text{FV}(M) \subseteq \bar{x} \). This is reminiscent of the standard way for defining a syntactic interpretation from a categorical one. (Again, see Koymans’s [5].)
Theorem 4.10. \( S \) is a non-extensional syntactic \( \lambda \)-model.

Proof. We need to check that the conditions (i)–(vi) from Definition 1.9 are satisfied by the interpretation function given in Definition 4.9.

Take \( \vec{x} = x_1, \ldots, x_n \), and write \( \rho(\vec{x}) \) for \( \rho(x_1), \ldots, \rho(x_n) \).

(i) \( [x_i]_\rho \simeq_A \#(\Pr^n_i @ [\rho(\vec{x})]) \simeq_A \rho(x_i) \), by Remark 4.7(i).

(ii) \( [\hat{a}]_\rho \simeq_A \#(\Cons^n_0 @ [\rho(\vec{x})]) \simeq_A a \), by Remark 4.7(ii).

(iii) In the application case, we have:

\[
[PQ]_\rho \simeq_A \#(\mid PQ \mid_{\vec{x}} @ [\rho(\vec{x})])
= \langle \emptyset^2, \mid P \mid_{\vec{x}}, \mid Q \mid_{\vec{x}}, \emptyset, \text{Apply}_n, [\rho(\vec{x})] \rangle,
\text{by Def. 4.6},
\simeq_A \#(M @ [\rho(\vec{x})]) \cdot \#(N @ [\rho(\vec{x})]),
\text{by Rem. 4.7(iii)},
= [P]_\rho \cdot [Q]_\rho.
\]

(iv) In the \( \lambda \)-abstraction case we have, for all \( a \in A \):

\[
[\lambda y.P]_\rho \cdot a \simeq_A \mid \lambda y.P \mid_{\vec{x}} @ [\rho(\vec{x})], a \simeq_A \mid P \mid_{\vec{x}, y}@ [\rho(\vec{x}), a] \simeq_A [P]_{\rho[y:=a]}.
\]

(v) This follows from Lemma 4.8(iv).

(vi) By definition \( [\lambda y.M]_\rho \simeq_A \#(\mid M \mid_{\vec{x}, y}@ [\rho(\vec{x})]) \) and by Lemma 4.8(ii) \( \mid M \mid_{\vec{x}, y}@ [\rho(\vec{x})] \) reduces to a stuck addressing machine. Similarly, for \( [\lambda x.N]_\rho \).

We conclude by applying (ae).

Remark 4.11. (i) For closed \( \lambda \)-terms \( M \in \Lambda^o \), we have \( \mid M \mid = [M] \).

(ii) It is easy to check that \( [K] \simeq_A \#K \) and \( [S] \simeq_A \#S \).

(iii) More generally, all addressing machines behaving as the combinator \( K \) (resp. \( S \)) are equated in the model.

Lemma 4.12. The syntactic \( \lambda \)-model \( S \) is not extensional.

Proof. It is enough to check that \( S \not\models 1 = I \). Now, we have:

\[
[1] = \langle \emptyset^2, \#Pr^2_1, \#Pr^2_2, \emptyset, \text{Apply}_2, [] \rangle;
[I] = \langle \emptyset, \text{Load 0; Call 0}, [] \rangle.
\]

By applying an indeterminate machine \( x_n \), the former reduces to a stuck machine, while the latter reduces to \( x_n \). By Lemma 4.5(ii), they must be different modulo \( \equiv_A \).

\]
A difficult problem that arises naturally is the characterization of the \( \lambda \)-theory induced by the \( \lambda \)-model \( S \) defined above.

**Proposition 4.13.** The \( \lambda \)-theory \( \text{Th}(S) \) is neither extensional nor sensible.

**Proof.** \( \text{Th}(S) \) is not extensional by Lemma 4.12. To show that it is not sensible, it is enough to check that \( S \not\models \lambda x.\Omega = \Omega \). Notice that

\[
|\Omega| = (\#|\Delta|, \#|\Delta|, \emptyset, 2 \leftarrow \text{App}(0, 1); \text{Call } 2, []),
\]

\[
\rightarrow_h |\Delta| \@ [\#|\Delta|],
\]

where:

\[
|\Delta| = (\emptyset, \#\text{Pr}_1, \#\text{Pr}_1, \emptyset, \text{Apply}_1, []).
\]

By induction on a derivation of \( M \equiv^\alpha_\lambda N \), one checks that \( M \equiv^\alpha_\lambda N \) and \( M \rightarrow_h D_1 \@ [D_2] \) with \( D_1 \simeq^\alpha_\lambda D_2 \simeq^\alpha_\lambda |\Delta| \) entails \( N \rightarrow_h D'_1 \@ [\#D'_2] \) for some \( D'_1 \simeq^\alpha_\lambda D'_2 \simeq^\alpha_\lambda |\Delta| \). We conclude because the machine \( |\lambda x.\Omega| \) is stuck. \( \square \)

We believe that \( \text{Th}(S) \) is semi-sensible, we conjecture that \( \text{Th}(S) = \lambda \).

5. Consistency Proof via Ordinal Analysis

In this section we adapt Barendregt’s proof of consistency of \( \lambda \omega \) (the least \( \lambda \)-theory closed under the (\( \omega \))-rule) to prove Lemma 4.5(ii), which entails the consistency of our system. First, we need to introduce in our setting the notion of context and underlined reduction, that are omnipresent techniques in the area of term rewriting systems.

5.1. Contexts and Underlined Head Reductions

In \( \lambda \)-calculus a context is a \( \lambda \)-term possibly containing occurrences of an algebraic variable, called hole, that can be substituted by any \( \lambda \)-term possibly with capture of free variables. We will define a context-machine similarly, namely as an addressing machine possibly having a “hole” denoted by \( \xi \). Formally, we introduce a new machine having no registers or program, only an empty tape (therefore distinguished from all machines populating \( M_\lambda \)):

\[
\xi = ([])
\]

We then extend our formalism to include machines working either directly or indirectly with one, or more, occurrences of \( \xi \). We wish to ensure the invariant that a machine \( M \) with no occurrences of \( \xi \) maintain as address \( \#M \) — for this reason we need to extend the range of addresses in a conservative way.

Consider a countable set \( \mathbb{B} \) of addresses such that \( A \cap \mathbb{B} = \emptyset \), and write \( X = A \cup \mathbb{B} \) for the set of extended addresses. As usual, we set \( X_\emptyset = X \cup \{\emptyset\} \).
Definition 5.1.  (i) An extended machine $X$ is either of the form

- $\xi @ T$
- $\langle \vec{R}, P, T \rangle$

where $\vec{R}$ are $X_{\mathcal{A}}$-valued registers, $P$ is a valid program, $T \in T_X$ is an $X$-valued tape. We write $\mathcal{M}^\xi_X$ for the set of all extended machines.

(ii) Fix a bijective map $\# : \mathcal{M}^\xi_X \rightarrow X$ satisfying $\#(M) = M$ for all addressing machine $M \in \mathcal{M}_X$. Write $\#^{-1}(\cdot) : X \rightarrow \mathcal{M}^\xi_X$ for its inverse.

(iii) The number of occurrences of $\xi$ in $X \in \mathcal{M}^\xi_X$ (resp. $R_i$, resp. $T$), written $\text{occ}_\xi(X) \in \mathbb{N} \cup \{\infty\}$ ($\text{occ}_\xi(R_i), \text{occ}_\xi(T) \in \mathbb{N} \cup \{\infty\}$), is defined as follows:

$$\text{occ}_\xi(\xi @ T) = 1 + \text{occ}_\xi(T);$$
$$\text{occ}_\xi(\langle \vec{R}, P, T \rangle) = \text{occ}_\xi(T) + \sum_{i=0}^{\infty} \text{occ}_\xi(R_i);$$
$$\text{occ}_\xi([a_1, \ldots, a_n]) = \text{occ}_\xi(\#^{-1}(a_1)) + \cdots + \text{occ}_\xi(\#^{-1}(a_n));$$
$$\text{occ}_\xi(R_i) = \begin{cases} 0, & \text{if } R_i = \emptyset, \\ \text{occ}_\xi(\#^{-1}(a)), & \text{if } R_i = a \in X. \end{cases}$$

Notice that $\text{occ}_\xi(M) \in \mathbb{N}$ entails that $\text{occ}_\xi(M.R_i), \text{occ}_\xi(M.T) \in \mathbb{N}$.

Examples 5.2. The following are examples of extended machines:

(i) $\xi$, with $\text{occ}_\xi(\xi) = 1$;

(ii) $K @ \#(\#(\xi @ \#(\xi)))$, with $\text{occ}_\xi(K @ \#(\#(\xi @ \#(\xi)))) = 3$;

(iii) for all $n \in \mathbb{N}$, $X_n = (\#(\xi, \varepsilon, [\#X_{n+1}])$. In this case, $\text{occ}_\xi(X_0) = \infty$.

As previously mentioned, a key property of contexts in $\lambda$-calculus is that one can plug a $\lambda$-term into the hole and obtain a regular $\lambda$-term. Similarly, given $M \in \mathcal{M}^\xi_X$ and $X \in \mathcal{M}^\xi_X$, we can define the addressing machine $X(M)$ obtained from $X$ by recursively substituting (even in the registers/tapes) each occurrence of $\xi$ by $M$. However, this operation is well-defined only when $\text{occ}_\xi(X)$ is finite, so we focus on extended machines enjoying this property.

Definition 5.3.

(i) A context-machine is any $C \in \mathcal{M}^\xi_X$ satisfying $\text{occ}_\xi(C) \in \mathbb{N}$.

29
(ii) Given a context-machine $C$ and $M \in \mathcal{M}_A$, define the addressing machine $C(M)$ as follows:

$$C(M) = \begin{cases} 
M @ T(M), & \text{if } C = \xi @ T, \\
\langle \check{R}(M), P, T(M) \rangle, & \text{if } C = \langle \check{R}, P, T \rangle;
\end{cases}$$

where (assuming $a \in X$, $T = [a_1, \ldots, a_n] \in T_X$ with $\text{occ}_\xi(a :: T) \in \mathbb{N}$):

$$a(M) = \#(\#^{-1}(a)(M));$$

$$R_i(M) = \begin{cases} 
\emptyset & \text{if } R_i = \emptyset, \\
a(M) & \text{if } R_i = a;
\end{cases}$$

$$T(M) = [a_1(M), \ldots, a_n(M)].$$

In the following, when writing $C(M)$ (resp. $a(M)$, $R_i(M)$, $T(M)$) we silently assume that the number of occurrences of $\xi$ in $C$ (resp. $a$, $R_i$, $T$) is finite. Let us introduce a notion of reduction for context-machines that allows to mimic the underlined reduction from [10]. The idea is to decompose a machine $N$ as $N = C(M)$ where $C$ is a context-machine and $M$ the underlined sub-machine. It is now possible to reduce $C$ independently from $M$ until either the machine reaches a final-state or $\xi$ reaches the head-position. In the latter case, we substitute the head occurrence of $\xi$ by $M$, and continue the computation.

**Definition 5.4.**

(i) The head reduction $\rightarrow_h$ is generalized to extended machines in the obvious way, using $\#(\cdot)$ rather than $\#(\cdot)$ to compute the addresses. In particular, the machine $\xi @ T \not\rightarrow_h$ is in final state, but it is not stuck.

(ii) Given $M \in \mathcal{M}_A$ and $C \in \mathcal{M}_A^\xi$, the $M$-underlined (head-)reduction $\rightarrow_h^M$ is defined by adding to (i) the rule

$$\xi @ T \rightarrow_h^M M @ T.$$

**Examples 5.5.** Let $C = S @ [\#\xi, \#\xi, \#x_n]$. Then $C(K) = S @ [\#K, \#K, \#x_n]$.

(i) $C \rightarrow_h^K \xi @ [\#x_n, \#(\xi @ [\#x_n])]$.

(ii) $C \rightarrow_h^K \xi @ [\#x_n, \#(\xi @ [\#x_n])] \rightarrow_h^K K @ [\#x_n, \#(\xi @ [\#x_n])] \rightarrow_h^K x_n.$
Lemma 5.6. For $C, C' \in \mathcal{M}_X^\xi$ and $M, N \in \mathcal{M}_A$, the following are equivalent:

1. $C \langle M \rangle \twoheadrightarrow h N$;
2. $C \xrightarrow{M} C'$ and $C' \langle M \rangle = N$.

Proof. ($1 \Rightarrow 2$) By induction on the length $n$ of the reduction $C \langle M \rangle \twoheadrightarrow h N$.

Case $n = 0$. Trivial, take $C' = C$.

Case $n > 0$. Let $C \langle M \rangle \twoheadrightarrow h N' \twoheadrightarrow h N$. Split into cases depending on $C$.

Subcase $C = \xi @ T$, therefore $C \langle M \rangle = M @ T \langle M \rangle \twoheadrightarrow h N'$. There are two possibilities:

- $M$ is stuck and $T \neq []$, say, $T = [a_0, \ldots, a_n]$. In this case $C \langle M \rangle = \langle \tilde{R}, \text{Load } i; P, [] \rangle$ and $N' = \langle \tilde{R}[R_i := a_0(\langle M \rangle)], \text{Load } i; P, [a_1(\langle M \rangle), \ldots, a_n(\langle M \rangle)] \rangle$.

Case $n > 0$, i.e. $C \xrightarrow{M} M @ T \xrightarrow{M} C''$ for

$$C'' = \langle \tilde{R}[R_i := a_0], \text{Load } i; P, [a_1, \ldots, a_n] \rangle$$

satisfying $C'' \langle M \rangle = N' \twoheadrightarrow h N$. We conclude by induction hypothesis.

- $M \twoheadrightarrow h M'$. In this case $N' = M' @ T \langle M \rangle$ and $C \xrightarrow{M} M @ T \xrightarrow{M} C''$ for $C'' = M' @ T$ satisfying $C'' \langle M \rangle = N' \twoheadrightarrow h N$. We conclude by IH.

Subcase $C = \langle \tilde{R}, P, T \rangle$. By case analysis on $P$. All cases follow easily from the induction hypothesis.

($2 \Rightarrow 1$) By induction on the length $n$ of the reduction $C \xrightarrow{M} C'$.

Case $n = 0$. Trivial, take $N = C \langle M \rangle$.

Case $n > 0$, i.e. $C \xrightarrow{M} C'' \xrightarrow{M} C'$, where the latter reduction is shorter.

Proceed by case analysis on the shape of $C$.

Subcase $C = \xi @ T$ and $C'' = M @ T$. Then $N = C'' \langle M \rangle = M @ T \langle M \rangle = C \langle M \rangle$. Conclude by induction hypothesis.

Subcase $C = \langle \tilde{R}, P, T \rangle$. By case analysis on $P$. All cases follow easily from the induction hypothesis.

5.2. Ordinal analysis

As mentioned in Remark 4.2, a derivation of $M \equiv^\omega_A N$ has the structure of a well-founded $\omega$-branching tree. Unfortunately, this makes it difficult to prove even simple properties like Lemma 4.5(ii). We need a more refined system exposing the underlying ordinal and handling the applications of the (Transitivity) rule separately.
\[ M \equiv_{\alpha} N \quad M \equiv_{\alpha} N (\approx_0) \quad M \sim_{\alpha} N (\subseteq_\alpha) \quad M \sim_{\alpha} N (\subseteq_\alpha) \]

\[ M, N \rightarrow_h \text{stuck}() \quad \forall a \in A, \exists \gamma < \alpha, M \equiv_a [a] \equiv_{\gamma} N \equiv_a [a] (\approx_\alpha) \]

\[ M \approx_{\alpha} N \]

\[ \#^{-1}(a) \sim_{\alpha} \#^{-1}(b) \] (R\(~_\alpha\)) \quad \[ \#^{-1}(a) \sim_{\alpha} \#^{-1}(b) \] (\@\(~_\alpha\))

\[ M \sim_{\alpha} N \quad T \in T^\alpha \quad (T_{\sim}) \quad M \equiv_{\alpha} N \quad T \in T^\alpha \quad (T_{\equiv}) \]

\[ M \equiv_{\alpha} a \quad M \equiv_{\alpha} b \] (R\(_\alpha\)) \quad \[ M \equiv_{\alpha} a \quad M \equiv_{\alpha} b \] (\@\(_\alpha\))

\[ M \equiv_{\alpha} N \quad \gamma \leq \alpha \quad (\leq_{\alpha}) \quad M \sim_{\alpha} N \quad \gamma \leq \alpha \quad (\subseteq_{\alpha}) \quad M \equiv_{\alpha} N \quad \gamma \leq \alpha \quad (\subseteq_{\alpha}) \]

\[ M \equiv_{\alpha} Z \quad Z \equiv_{\alpha} N \quad (\text{Tr}_{\alpha}) \]

Figure 1: Rules satisfied by \( \approx_{\alpha}, \sim_{\alpha} \) and \( \equiv_{\alpha} \), beyond reflexivity and symmetry.

**Definition 5.7.**

(i) Let \( \omega_1 \) be the set of all countable ordinals.

(ii) If \( \pi \) is a derivation of \( M \equiv_{\alpha} N \), we define its length \( \ell(\pi) \in \omega_1 \) in the usual inductive way for the rules \( (\rightarrow_{\alpha}), \text{ (Refl.)}, \text{ (Symm.)}, \text{ (Trans.)} \). Concerning the rule \( (\alpha) \) having countably many premises, we set:

\[ \ell \left( M, N \rightarrow_h \text{stuck}() \quad \forall a \in A, \exists \gamma < \alpha, M \equiv_a [a] \equiv_{\gamma} N \equiv_a [a] \right) = \sup_{a \in A} (\ell(\pi_a) + 1) \]

It is easy to check that, if a derivation \( \pi \) has premises \( (\pi_i)_{i \in I} \) for some countable set \( I \) then \( \ell(\pi) > \ell(\pi_i) \) for every \( i \in I \).

(iii) For all \( \alpha \in \omega_1 \), define \( \equiv_{\alpha}, \sim_{\alpha}, \approx_{\alpha} \subseteq M^2_{\alpha} \) as the least reflexive and symmetric relations closed under the rules of Figure 1.

The intuitive meanings of the relations \( \equiv_{\alpha}, \sim_{\alpha}, \approx_{\alpha} \) are the following:

- \( M \equiv_{\alpha} N \leftrightarrow M \equiv_{\alpha} N \) is derivable using the rule \( (\alpha) \) at most \( \alpha \) times;
- \( M \sim_{\alpha} N \leftrightarrow M \equiv_{\alpha} N \) is derivable without using transitivity;
• $M \approx_\alpha N \iff M \equiv^{\alpha}_\delta N$ in case $\alpha = 0$. Otherwise, if $\alpha > 0$ then

• $M \approx_\alpha N \iff M \sim_\alpha N$ follows directly from the rule $(\alpha e)$.

The proof of the next lemma is given in the technical appendix.

**Lemma 5.8.** Let $M, N \in \mathcal{M}_A$

(i) $M \equiv^{\alpha}_\delta N \iff \exists \alpha \in \omega. M \equiv_\alpha N$.

(ii) $M \equiv_0 N \iff M \equiv_\delta N$.

(iii) $M \equiv_\alpha N \iff \exists n \geq 0, Z_1, \ldots, Z_n \in \mathcal{M}_A. M \sim_\alpha Z_1 \sim_\alpha \cdots \sim_\alpha Z_n = N$.

(iv) $M \sim_\alpha N \iff \exists C \in \mathcal{M}_A^\xi, M', N' \in \mathcal{M}_A. M = C(M') \land N = C(N') \land M' \approx_\alpha N'$.

(v) $M \approx_\alpha N \land \alpha \neq 0 \iff M, N \downarrow \text{stuck}(\alpha) \land \forall a \in A, \exists \gamma < \alpha. M @ [a] \equiv_\gamma N @ [a]$.

**Lemma 5.9.** Let $\alpha > 0$, $C \in \mathcal{M}_A^\xi$, $M, N \in \mathcal{M}_A$ such that $M \approx_\alpha N$. If $C \rightarrow^M_h C'$ and $C'(M) \not\rightarrow^h \text{stuck}(\alpha)$, then there exists $\gamma < \alpha$ such that $C(N) \equiv_\gamma C'(N)$.

**Proof.** By cases on the shape of $C$.

Case $C = \xi @ T$ for some $T \in T_X$ and $C' = M @ T$. From $M \approx_\alpha N$ and Lemma 5.8[v], we get that $M \rightarrow^h \text{stuck}(M')$ for some $M' \in \mathcal{M}_A$. Since $C'(M) = M @ (T(M))$ cannot reduce to a stuck addressing machine, we must have $T(M) \neq \emptyset$. In other words, $T = [a_0, \ldots, a_n]$ for some $n \geq 0$. Notice that, for all $a_i \in T_X$, we have $a_i(N) \in A$ (by construction). By Lemma 5.8[v], there exists $\gamma < \alpha$ such that $N @ [a_0(N)] \equiv_\gamma M @ [a_0(N)]$. By definition:

$$C(N) = N @ T(N), \quad \text{and} \quad C'(N) = M @ T(N).$$

So we construct the proof:

$$\frac{N @ [a_0(N)], \ldots, a_n(N) \equiv_\gamma M @ [a_0(N)], \ldots, a_n(N)}{N @ [a_0(N)] \equiv_\gamma M @ [a_0(N)]} \quad (T_\gamma)$$

In all the other cases, $C(N) \rightarrow_h C'(N)$, therefore $C(N) =_0 C(N)$.

**Corollary 5.10.** Let $n \in \mathbb{N}, \alpha > 0$, $C \in \mathcal{M}_A^\xi$, $M, N \in \mathcal{M}_A$. If $C(M) \rightarrow^h x_n$ and $M \approx_\alpha N$ then there exists $\gamma < \alpha$ such that $C(N) \equiv_\gamma x_n$. 33
Proof. Assume $C(M) \rightarrow_h x_n$. Equivalently, by Lemma 5.6, we have $C \rightarrow^M h x_n$. By definition, there exists $C_1, \ldots, C_k \in \mathcal{M}_x^\xi$ such that

$$C = C_1 \rightarrow^M h \cdots \rightarrow^M h C_k = x_n$$

Notice that $C_i(M) \rightarrow_h x_n$ and, since $\neg\text{stuck}(x_n)$, we have $C_i(N) \not\rightarrow_h \text{stuck}()$. By Lemma 5.9, there exists $\gamma_1, \ldots, \gamma_k < \alpha$ such that $C_i(N) \equiv_{\gamma_i} C_{i+1}(N)$. By transitivity ($\text{Tr}_\alpha$) and ($\leq_\alpha$) we obtain $M \equiv_{\alpha} x_n$ for $\alpha = \sup_{i} \gamma_i$. □

**Proposition 5.11.** Let $M, N \in \mathcal{M}_x^\omega$, $\alpha \in \omega_1$ and $n \in \mathbb{N}$. If $M \equiv_{\alpha} N$ and $N \rightarrow_h x_n$ then $M \rightarrow_h x_n$.

**Proof.** We proceed by induction on $\alpha$. Since we perform a double induction, the induction hypothesis w.r.t. this induction is called the $\alpha$-IH ($\alpha$-ind. hyp.).

Case $\alpha = 0$. By Lemma 5.8(ii), we get $M \equiv^\omega_{\alpha} N \rightarrow_h x_n$, so we conclude $M \rightarrow_h x_n$ by confluence (Theorem 3.10) and $\to_i$-postponement (Lemma 3.7).

Case $\alpha > 0$. By Lemma 5.8(iii), there exist $Z_1, \ldots, Z_k$ such that

$$M \sim_{\alpha} Z_1 \sim_{\alpha} \cdots \sim_{\alpha} Z_k = N \rightarrow_h x_n \quad (2)$$

By induction on $k$, we prove that (2) implies $M \rightarrow_h x_n$. We call this $k$-IH.

Subcase $k = 0$. Then $M = N \rightarrow_h x_n$, and we are done.

Subcase $k > 0$. From the $k$-IH we derive $Z_1 \rightarrow_h x_n$. From $M \sim_{\alpha} Z_1$ and Lemma 5.8(iv), there is a context-machine $C$ such that $M = C[M']$ and $Z_1 = C[N']$ with $M' \sim_{\alpha} N'$ and $C[N'] \rightarrow_h x_n$. By applying Lemma 5.9 we obtain $C[M'] \equiv_{\gamma} x_n$ for some $\gamma < \alpha$. We conclude by applying the $\alpha$-IH. □

From this proposition, Lemma 4.5(iii) follows by applying Lemma 5.8(i).

6. Conclusions and Further Works

In this paper, we have shown that it is possible to obtain a model of the untyped $\lambda$-calculus based on a kind of computational machines that operate exclusively on “addresses”, without any reference to some basic data type. The result only depends on the assumption that every machine has a unique address (and vice versa every address identifies a machine) and is completely independent from the specific nature of the addresses themselves. The following appear to be natural natural developments for further works:
1. explore whether the theory of the $\lambda$-model $S$ defined in Section 5 depends on the specific nature of the bijection $\#(\cdot): A \to M_A$. Indeed, given an injection $f: \mathbb{N} \to A$, we might have machines $M'_n$ satisfying $M'_n = \langle f(n), \varepsilon, \#M'_{n+1} \rangle$ for all $n \in \mathbb{N}$ (cf. Example 5.2(iii)). The existence of these machines strictly depends on the chosen bijection $\#(\cdot)$ and, for cardinality reasons, not all machines of this kind exist (one would need a set of addresses having the size of the continuum). We conjecture that $\text{Th}(S)$ is actually independent from the choice of the lookup function $\#(\cdot)$, because none of these machines are $\lambda$-definable.

2. expand the computational capabilities of addressing machines by adding simple data-types and the associated basic operations. In fact, although data-types are unnecessary to achieve Turing-completeness, they are desirable to perform arithmetical operations and conditionals. E.g., one could add booleans (with logical connectives and conditional branching) and natural numbers (with successor, predecessor, and the like). A step in a different direction would be to extend addressing machine with an “internal state”, taking only a finite number of values and depending on the states of the machines in its “neighborhood”. The idea is to develop a kind of “cellular automata” model and study the interaction between the machines and their asymptotic behavior.

To perform some tests on addressing machines, we have implemented the formalism both in functional and imperative style. Even if the sources remain for internal use only, some technical choices deserve a discussion. Although not explicitly required by the definition, any implementation must rely on a computable association between addressing machines and the corresponding addresses. The problem is how to define a bijection between the set of all possible machines and the corresponding address. One could try to use as addresses the actual references pointing to the structures representing the machines, but the referenced data might change without affecting the address. We decided to use an association list $\ell$ of type $A \times M_A$ and an incremental approach. The list $\ell$ is initialized as the empty-list. When a new machine $M$ is created, one checks whether $M$ belongs to $\pi_2(\ell)$: in the positive case there is nothing to do as the machine is already known; otherwise, a new address $a$ is generated and the pair $(a, M)$ is added to the list $\ell$. This allows to guarantee that an address uniquely identifies a machine and that, when an address is used, the corresponding machine has already been introduced.
Acknowledgements. This work is partly supported by ANR Project PPS, ANR-19-CE48-0014. We would like to thank Henk Barendregt for interesting discussions concerning the problem of finding a model of λ-calculus based on recursive functions, as well as the role of the (ω)-rule. We are also thankful to Nicolas M{"u}nnik for his comments on the paper.

Appendix A. Technical Appendix

This technical appendix contains some lengthy proofs that were omitted in the body of the article.

Proof of Proposition 2.15. 
(1 ⇒ 2) By induction on the length n of the reduction \( M = M_1 \hookrightarrow_h M_2 \hookrightarrow_h \cdots \hookrightarrow_h M_n = N \not\hookrightarrow_h \).

Case n = 0. By assumption \( N \) is in a final state. By Remark 2.10(ii), it is either of the form \( N = \langle \vec{R}, \varepsilon, T \rangle \) or it is stuck \( N = \langle \vec{R}, \text{Load } i; P, [] \rangle \). In the former case we apply (End), in the latter (Stuck).

Case n > 1. Since \( M_1 \hookrightarrow_h M_2 \), we have \( M_1.P \not\varepsilon = \varepsilon \). As the length of \( M_2 \hookrightarrow_h N \) is \( n - 1 \), by induction hypothesis we have a derivation of \( M_2 \Downarrow N \).

Depending on the first instruction in \( M_1.P \), we use this derivation to apply the homonymous rule (Load), (App) or (Call) and derive \( M \Downarrow N \).

(2 ⇒ 1) By induction on a derivation of \( M \Downarrow N \).

Cases (Stuck) or (End). Then, \( M \hookrightarrow_h M = N \) by reflexivity of \( \hookrightarrow_h \).

Case (Load), i.e. \( M.P = \text{Load } i; P' \). In this case, we have that \( M \hookrightarrow_h \langle M.\vec{R}[R_i := a], P', M.T \rangle \hookrightarrow_h N \), by induction hypothesis.

Case (App), i.e. \( M.P = k \hookrightarrow \text{App}(i, j); P' \). Let us call \( a = M.R_j \cdot M.R_k \).

Then we have \( M \hookrightarrow_h \langle M.\vec{R}[R_k := a], P', M.T \rangle \hookrightarrow_h N \), by induction hypothesis.

Case (Call), i.e. \( M.P = \text{Call } i \). In this case \( M \hookrightarrow_h M' @ [M.T] \) for \( M' = \#^{-1}(M.R_i) \). By induction hypothesis \( M' @ [M.T] \hookrightarrow_h N \), whence \( M \hookrightarrow_h N \). \( \square \)

Proof of Lemma 4.8. 
(i) By a straightforward induction on \( M \).

(ii) It follows from (i).

(iii) We proceed by structural induction on \( M \). By (ii), if \( \vec{a} \neq \emptyset \) then both addressing machines reduce to stuck ones, so we can test the applicative equivalence by applying an arbitrary \( \vec{a} \) and conclude using (ae) \( n \)-times.

Case \( M = \hat{c} \). Then \( c[b/y] = c, \) and we have:

\[
|\hat{c}|_{y,\vec{a}} @ [b, \vec{a}] = \text{Cons}^{n+1}_c @ [b, \vec{a}] \hookrightarrow_h \#^{-1}(c) @ \text{Cons}^n_c @ [\vec{a}].
\]
Case $M = x_i$ for some $i (1 \leq i \leq n)$. Then $x_i[b/y] = x_i$ and

$|x_i|_{y, \bar{x}} @ [b, \bar{a}] = P_{r_{i+1}} \equiv M \leadsto_h \#^{-1}(a_i) \hastype h \#^{-1}(a_i) \hastype h \Rightarrow |x_i|_{\bar{x}} @ [\bar{a}]$.

Case $M = y$. Then $y[b/y] = \hat{b}$ and we have:

$|y|_{y, \bar{x}} @ [b, \bar{a}] = P_{r_1} \equiv M \leadsto_h \#^{-1}(b) \hastype h \#^{-1}(b) \hastype h \Rightarrow |y|_{\bar{x}} @ [\bar{a}]$.

Case $M = PQ$. Then $(PQ)[b/y] = (P[b/y])(Q[b/y])$ and we have:

$|PQ|_{y, \bar{x}} @ [b, \bar{a}] = \langle \emptyset^{n+1}, \#P|_{y, \bar{x}}, \#Q|_{y, \bar{x}}, \emptyset, \text{Apply}_{n+1}, [b, \bar{a}] \rangle$

$\leadsto_h |P|_{y, \bar{x}} @ [b, \bar{a}], \langle \emptyset(QQ_{y, \bar{x}} @ [b, \bar{a}] \rangle$. By IH,

$\cong A \langle \emptyset, \#(PQ)|_{y, \bar{x}}, \#PQ|_{y, \bar{x}} @ [b, \bar{a}] \rangle$,

so $(P[b/y])(Q[b/y])|_{\bar{x}} = (PQ)[b/y]|_{\bar{x}}$.

Case $M = \lambda z.P$, wlog $z \notin y, \bar{x}$, so $(\lambda z.P)[b/y] = \lambda z.P[b/y]$. By (iv) both machines reduce to stuck ones. So we have to apply an extra $a_{n+1} \in A$.

$|\lambda z.P|_{y, \bar{x}} @ [b, \bar{a}, a_{n+1}] = |P|_{y, \bar{x}, z} @ [b, \bar{a}, a_{n+1}]$

$\equiv A |P|_{b/x} @ [\bar{a}, a_{n+1}]$, by IH,

$= |\lambda z.P[b/y]|_{\bar{x}} @ [\bar{a}, a_{n+1}]$.

(iv) By (iii).

(v) By (iv), permuting the substitutions. \hfill \Box

Proof of Lemma 5.8 (i) (⇐) Easy.

(⇒) By induction on the length of a derivation of $M \equiv_A N$.

Case $(-^A \gamma)$. i.e., there exists $Z \in M_A$ such that $M \hastype h Z \equiv_A N$. By Theorem 3.10 we have $M \equiv_A Z$ whence $M \equiv_0 Z$ by $(\approx_0)$, which implies $M \equiv_\alpha Z$ for all $\alpha \in \omega_1$ using the rule $(\leq_\alpha)$. Now, consider the set

$\mathcal{R} = \{ i \mid Z.R_i \neq \emptyset \} = \{ i \mid N.R_i \neq \emptyset \}$

Note that $\mathcal{R} = \{ i_1, \ldots, i_k \}$ for some $k < Z.r_0(= N.r_0)$. For every $i \in \mathcal{R}$, let $Z.R_i = a_i$ and $N.R_i = a_i'$. Also, let $Z.T = [b_1, \ldots, b_m]$ and $N.T = [b_1', \ldots, b_m']$. By assumption, $a_i \equiv_A a_i'$ and $b_j \equiv_A b_j'$ for every $i \in \mathcal{R}$ and $j (1 \leq j \leq m)$.

By induction hypothesis, $\#^{-1}(a_i) \equiv_{\gamma_i} \#^{-1}(a_i')$ and $\#^{-1}(b_j) \equiv_{\delta_j} \#^{-1}(b_j')$. 

37
Using the rule \((<_\alpha)\), the same holds for \(\equiv_\alpha\) setting \(\alpha = \sup_{i \in R, 1 \leq j \leq m} \{\gamma_i, \delta_j\}\).

Putting everything together, we obtain:

\[
M \equiv_\alpha Z = \langle Z, \vec{R}, P, [b_1, \ldots, b_m] \rangle
\]

\[
\equiv_\alpha \langle Z, \vec{R}[R_i := a'_i], P, [b_1, \ldots, b_m] \rangle, \quad \text{by } (R_\alpha),
\]

\[
\equiv_\alpha \ldots
\]

\[
\equiv_\alpha \langle N, \vec{R}, P, [b'_1, b_2, \ldots, b_m] \rangle, \quad \text{by } (T_\alpha),
\]

\[
\equiv_\alpha \ldots
\]

\[
\equiv_\alpha \langle N, \vec{R}, P, [b'_1, \ldots, b'_m] \rangle, \quad \text{by } (T_\alpha),
\]

\[
= \mathcal{N}, \quad \text{by definition.}
\]

We conclude by applying the transitivity rule \((\text{Tr}_\alpha)\) that \(M \equiv_\alpha N\).

Case (\(\{\text{ae}\}\)). By induction hypothesis, for every \(a \in A\), there exists \(\gamma_a \in \omega_1\) such that \(M@[a] \equiv_\gamma N@[a]\). For \(\gamma = \sup_{a \in A} \gamma_a\), we get \(M@[a] \equiv_\gamma N@[a]\) by \((\leq_\alpha)\). By \((\sim_\alpha)\) we get \(M \sim_\alpha N\) for \(\alpha = \gamma + 1 \in \omega_1\), conclude by \((\subseteq_\alpha), (\subseteq_\sim)\).

(Reflexivity), (Symmetry) and (Transitivity) follow from the respective property of \(\equiv_\alpha\).

Concerning items (\[\text{ii}\])--(\[\text{v}\]) the implication \((\Leftarrow)\) is trivial. We analyze \((\Rightarrow)\).

\[\text{(ii)}\] By induction on a derivation of \(M \equiv_0 N\), using Theorem 3.10.

\[\text{(iii)}\] By induction on a derivation of \(M \equiv_\alpha N\).

Case \((\subseteq_\alpha)\). Trivial.

Case \((R_\alpha)\). I.e., \(M = Z[R_i := a], N = Z[R_i := b]\) and \(#^{-1}(a) \equiv_\alpha #^{-1}(b)\).

By induction hypothesis, there exist \(c_1, \ldots, c_k \in A\) such that

\[
#^{-1}(a) \sim_\alpha #^{-1}(c_1) \sim_\alpha \cdots \sim_\alpha #^{-1}(c_k) = #^{-1}(b).
\]

The case follows by applying the rule \((R_\sim)\).

Case \((@_\alpha)\). Analogous, by applying \((@_\sim)\).

Case \((T_\alpha)\). Analogous, by applying \((T_\sim)\).

Case \((\text{Tr}_\alpha)\). Straightforward from the IH.

Case \((\leq_\alpha)\). By IH and \((\leq_\sim)\).

Cases (Reflexivity), (Symmetry). Straightforward from the IH.

\[\text{(iv)}\] By induction on a derivation of \(M \sim_\alpha N\).

Case \((\subseteq_\alpha)\). Take \(C = \xi\).

Case \((R_\sim)\). I.e., \(M = Z[R_i := a], N = Z[R_i := b]\) and \(#^{-1}(a) \sim_\alpha #^{-1}(b)\).

By induction hypothesis, there exist \(C' \in \mathcal{M}_X^\xi\) having address \(c = \#C' \in X\),

38
\( M', N' \in \mathcal{M}_\alpha \) such that \( C'[M'] = \#^{-1}(a) \), \( C'[N'] = \#^{-1}(b) \) and \( M' \approx_\alpha N' \). We conclude by taking \( C = Z[R_i := c] \).

Case \( (\@_\alpha^\sim) \). Analogous.

Case \( (T^\alpha_\sim) \). Take \( C = C' \@ T \), where \( C' \) is obtained from the IH.

Case \( (\leq_\alpha^\approx) \). It follows from the IH, by applying \( (\leq_\alpha^\sim) \) and \( (\leq_\alpha^\approx) \).

Cases (Reflexivity), (Symmetry). Straightforward from the IH. \( \square \)

References


