Extended Addressing Machines for PCF, with Explicit Substitutions

Benedetto Intrigila[1], Giulio Manzonetto[⋆2], and Nicolas Münnich[⋆⋆2]

[1] Dipartimento di Ingegneria dell'Impresa, University of Rome "Tor Vergata", Italy
benedetto.intrigila@uniroma2.it
[2] USPN, Sorbonne Paris Cité, LIPN, UMR 7030, CNRS, F-93430, France
{manzonetto,munnich}@lipn.univ-paris13.fr

Abstract. Addressing machines have been introduced as a formalism to construct models of the pure, untyped λ-calculus. We extend the syntax of their programs by adding instructions for executing arithmetic operations on natural numbers, and introduce a reflection principle allowing certain machines to access their own address and perform recursive calls. We prove that the resulting extended addressing machines naturally model a weak call-by-name PCF with explicit substitutions. Finally, we show that they are also well suited for representing regular PCF programs (closed terms) computing natural numbers.

Keywords: Addressing machines · PCF · explicit substitutions.

Introduction

Turing machines (TM) and λ-calculus constitute two fundamental formalisms in theoretical computer science. Despite their equivalence on partial numeric functions given by Church-Turing thesis, there are no models of λ-calculus based on TM's, since higher-order calculations are difficult to emulate in a TM. Recently, Della Penna et al. have successfully built a λ-model based on so-called addressing machines (AM) [9]. These machines are solely capable of manipulating the addresses of other machines — this opens the way for modelling higher-order computations since functions can be passed via their addresses. An AM can read an address from its tape, store the result of applying an address to another and pass the execution to another machine by calling its address. The set of instructions is deliberately small, to identify the minimal setting needed to represent λ-terms. The downside is that performing calculations on natural numbers in an AM is as awkward as using Church numerals in λ-calculus.

So we extend the formalism of AM's with instructions representing basic arithmetic operations and conditional tests on natural numbers. As we are entering a world of machines and addresses, we need specific machines to represent numerals and assign them recognizable addresses. Finally, to model recursion, we

⋆ Partly supported by ANR Project PPS, ANR-19-CE48-0014.
⋆⋆ Partly supported by ANR Project CoGITARe, ANR-18-CE25-0001.

rely on the existence of machines representing fixed-point combinators — these machines can be programmed in the original formalism but we need as an external assumption that they have access to their own address. This can be seen as a very basic version of the reflection principle in some programming languages. We call the resulting formalism *extended abstract machines* (EAMs).

Considering these features, one might expect EAMs to be well-suited for simulating Plotkin's PCF [16], a simply typed λ-calculus with constants, arithmetical operations, conditional testing and a fixed point combinator. A PCF term of the form $(\lambda x.M)N$ can indeed be translated into a machine M reading as input (x) from its tape the address of N. As M has control over the computation, it naturally models a weak leftmost call-by-name evaluation. However, while the contraction of the redex in $M[N/x]$ is instantaneous, M needs to pass the address of N to the machines representing the subterms of M, with the substitution only being performed if N gains control of the computation. As a result, rather than PCF, EAMs naturally emulate the behavior of EPCF — a weak call-by-name PCF with explicit substitutions that are only performed "on demand", as in [12]. We endow EAMs with a typing mechanism based on simple types and define a type-preserving translation from well-typed EPCF terms to EAMs. Subsequently, we prove that also the operational behavior of EPCF is faithfully represented by the translation. Finally, by showing an equivalence between PCF and EPCF on terminating programs of type int, we draw conclusions for PCF.

The paper is organized as follows. In Section 1 we introduce the language EPCF along with its syntax, simply typed assignment system and associated (call-by-name) big-step operational semantics. We assume familiarity with the standard (call-by-name) PCF, otherwise the reader may consult [15]. In Section 2 we define EAMs (no familiarity with [9] is assumed) and introduce their operational semantics. In Section 3 we describe a type–checking algorithm for determining whether an EAM is well-typed. In Section 4 we present our main results, namely: (*i*) the translation of a well-typed EPCF term is an EAM typable with the same type (Theorem 1); (*ii*) if an EPCF term big-step reduces to a value, then their translations as machines are interconvertible (Theorem 2); (*iii*) the operational semantics of PCF and EPCF coincide on terminating programs of type int (Theorem 3); (*iv*) the translation of a PCF program computing a number is an EAM evaluating the corresponding numeral (Theorem 4).

Related works. A preliminary version of addressing machines appeared in Della Penna's MSc thesis [8]. Presenting a comprehensive survey of all machine based formalisms introduced in the literature would be unrealistic. The most similar ones are likely by Fairbairn and Stuart [10], but they were studied with a specific focus on the implementation of supercombinators. The model based on AMs also bares *some* similarities with the categories of assembly used by Longley to model PCF [13], but more on a philosophical rather than implementational level. For other models of PCF based on game semantics, the reader may consult [4,11]. Concerning explicit substitutions we refer to the pioneering articles [1,2,7,12]. As far as we can tell, they have been barely considered in the context of PCF — with the notable exception of [17],

$$\frac{\underline{n} \in \mathbb{N}}{\sigma \triangleright \underline{n} \Downarrow_d \underline{n}}\ (\text{nat}) \qquad \frac{}{\sigma \triangleright \lambda x.M\langle\rho\rangle \Downarrow_d \lambda x.M\langle\sigma+\rho\rangle}\ (\text{fun})$$

$$\frac{\sigma(x) = (\rho, N) \quad \rho \triangleright N \Downarrow_d V}{\sigma \triangleright x \Downarrow_d V}\ (\text{var}) \qquad \frac{\sigma \triangleright M \cdot (\mathbf{fix}\, M) \Downarrow_d V}{\sigma \triangleright \mathbf{fix}\, M \Downarrow_d V}\ (\text{fix})$$

$$\frac{\sigma \triangleright M \Downarrow_d \underline{0} \quad \sigma \triangleright N_1 \Downarrow_d V_1}{\sigma \triangleright \mathbf{ifz}(M, N_1, N_2) \Downarrow_d V_1}\ (\text{ifz}_0) \qquad \frac{\sigma \triangleright M \Downarrow_d \underline{n+1} \quad \sigma \triangleright N_2 \Downarrow_d V_2}{\sigma \triangleright \mathbf{ifz}(M, N_1, N_2) \Downarrow_d V_2}\ (\text{ifz}_{>0})$$

$$\frac{\sigma \triangleright M \Downarrow_d \underline{n+1}}{\sigma \triangleright \mathbf{pred}\, M \Downarrow_d \underline{n}}\ (\text{pr}) \qquad \frac{\sigma \triangleright M \Downarrow_d \underline{0}}{\sigma \triangleright \mathbf{pred}\, M \Downarrow_d \underline{0}}\ (\text{pr}_0) \qquad \frac{\sigma \triangleright M \Downarrow_d \underline{n}}{\sigma \triangleright \mathbf{succ}\, M \Downarrow_d \underline{n+1}}\ (\text{sc})$$

$$\frac{\sigma \triangleright M \Downarrow_d \lambda x.M'\langle\rho\rangle \quad \rho + [x \leftarrow (\sigma, N)] \triangleright M' \Downarrow_d V}{\sigma \triangleright M \cdot N \Downarrow_d V}\ (\beta_v)$$

Fig. 1. The big-step operational semantics of EPCF.

1 Preliminaries

The paradigmatic programming language PCF [16] is a simply typed λ-calculus enriched with constants representing integers, the fundamental arithmetical operations, an if-then-else conditional instruction, and a fixed-point operator. We give PCF for granted and rather present EPCF, an extension of PCF with explicit substitutions [12]. We draw conclusions for the standard PCF by exploiting the fact that they are equivalent on programs (closed terms) of type int.

Definition 1. *Consider fixed a countably infinite set* Var *of variables.* EPCF terms *and* explicit substitutions *are defined by (for* $n \geq 0$ *and* $\vec{x} \in \mathrm{Var}$*):*

$$\begin{aligned} L, M, N ::= {} & x \mid M \cdot N \mid \lambda x.M\langle\sigma\rangle \\ & \mid \mathbf{0} \mid \mathbf{pred}\, M \mid \mathbf{succ}\, M \mid \mathbf{ifz}(L, M, N) \mid \mathbf{fix}\, M \\ \sigma, \rho \quad ::= {} & [x_1 \leftarrow (\sigma_1, M_1), \ldots, x_n \leftarrow (\sigma_n, M_n)] \end{aligned}$$

As is customary, $M \cdot N$ stands for the *application* of a term M to its argument N, $\mathbf{0}$ represents the natural number 0, **pred** and **succ** indicate the predecessor and successor respectively, **ifz** is the conditional test on zero, and finally, **fix** is a fixed-point operator. We assume that application – often denoted as juxtaposition – associates to the left and has higher precedence than abstraction. Concerning $\lambda x.M\langle\sigma\rangle$, it represents an *abstraction* where σ is an ordered list of assignments from variables to *closures* (terms with the associated substitutions), where each variable can only have one closure assigned to it.

In an explicit substitution $\sigma = [x_1 \leftarrow (\sigma_1, M_1), \ldots, x_n \leftarrow (\sigma_n, M_n)]$ the x_i's are assumed to be fresh and distinguished. So, we can define $\sigma(x_i) = (\sigma_i, M_i)$. The *domain* of σ is given by $\mathrm{dom}(\sigma) = \{x_1, \ldots, x_n\}$. We write $\sigma + \rho$ for the concatenation of σ and ρ, and in this case we assume $\mathrm{dom}(\sigma) \cap \mathrm{dom}(\rho) = \emptyset$.

The set $\mathrm{FV}(M)$ of *free variables* of an EPCF term M is defined as usual, except for the abstraction case $\mathrm{FV}(\lambda x.M\langle\sigma\rangle) = \mathrm{FV}(M) - (\{x\} + \mathrm{dom}(\sigma))$. The term M is *closed* if $\mathrm{FV}(M) = \emptyset$, and in that case it is called an EPCF *program.*

Hereafter terms are considered up to renaming of bound variables. Therefore the symbol $=$ will denote syntactic equality up to α*-conversion.*

Notation 1. *1. For every $n \in \mathbb{N}$, we let $\underline{n} = \mathbf{succ}^n(\mathbf{0})$. In particular, $\underline{0}$ is an alternative notation for $\mathbf{0}$.*
2. As a syntactic sugar, we write $\lambda x.M$ for $\lambda x.M\langle\rangle$. With this notation in place, PCF terms *are simply* EPCF *terms containing empty explicit substitutions.*
3. For $n \in \mathbb{N}$, we often write $\lambda x_1 \dots \lambda x_n.M$ as $\lambda x_1 \dots x_n.M$, or even $\lambda \vec{x}.M$ when n is clear from the context. Summing up, and recalling that $\cdot$ is left associative, $\lambda x_1 x_2 x_3.L \cdot M \cdot N$ stands for $\lambda x_1.(\lambda x_2.(\lambda x_3.((L \cdot M) \cdot N)\langle\rangle)\langle\rangle)\langle\rangle$.
4. As usual, $M[N/x]$ denotes the capture-free substitution of N for all free occurrences of x in M.

Example 1. We introduce some notations for the following (E)PCF programs, that will be used as running examples.

1. $\mathbf{I} = \lambda x.x$, representing the identity.
2. $\mathbf{\Omega} = \mathbf{fix}\,(\mathbf{I})$ representing the paradigmatic looping program.
3. $\mathbf{succ1} = \lambda x.\mathbf{succ}\,(x)$, representing the successor function.
4. $\mathbf{succ2} = (\lambda sn.s \cdot (s \cdot n)) \cdot \mathbf{succ1}$, representing the function $f(x) = x + 2$.
5. $\mathbf{add\_aux} = \lambda fxy.\mathbf{ifz}(y, x, (f \cdot (\mathbf{succ}\,x)) \cdot (\mathbf{pred}\,y))$, i.e. the functional

$$\Phi_f(x,y) = \begin{cases} x, & \text{if } y = 0, \\ f(x+1, y-1), & \text{if } y > 0. \end{cases}$$

6. $\mathbf{add} = \mathbf{fix}\,(\mathbf{add\_aux})$, i.e., the recursive definition of addition $f(x,y) = x+y$.

The operational semantics of EPCF is defined through a call-by-value big-step (leftmost) weak reduction.

Definition 2. *1. We let* $\mathrm{Val} = \{\underline{n} \mid n \in \mathbb{N}\} \cup \{\lambda x.M\langle\sigma\rangle \mid M$ *is an* EPCF *term*$\}$ *be the set of* EPCF values.
2. The big-step weak reduction *is the least relation $\Downarrow_d$ from* EPCF *terms to* Val*, closed under the rules of Figure 1.*
3. We say that an EPCF *term M is* terminating *whenever $M \Downarrow V$ for some $V \in$ Val. Otherwise, we say that M is a* non-terminating*, or* looping*, term.*

Example 2. We show some of the terms from Example 1, at work.

1. We have $[\,] \triangleright \mathbf{succ1} \cdot \mathbf{0} \Downarrow_d \underline{1}$. To get the reader familiar with the operational semantics, we give the details:

$$\dfrac{\dfrac{}{[\,] \triangleright \lambda x.\mathbf{succ}\,(x) \Downarrow_d \lambda x.\mathbf{succ}\,(x)}\ (\mathrm{fun}) \quad \dfrac{\dfrac{\dfrac{}{[\,] \triangleright \underline{0} \Downarrow_d \underline{0}}\ (\mathrm{nat})}{[x \leftarrow ([\,], \underline{0})] \triangleright x \Downarrow_d \underline{0}}\ (\mathrm{var})}{[x \leftarrow ([\,], \underline{0})] \triangleright \mathbf{succ}\,(x) \Downarrow_d \underline{1}}\ (\mathrm{sc})}{[\,] \triangleright (\lambda x.\mathbf{succ}\,(x)) \cdot \underline{0} \Downarrow_d \underline{1}}\ (\beta_v)$$

2. Similarly, $[\,] \triangleright \mathbf{I} \cdot \underline{4} \Downarrow_d \underline{4}$, $[\,] \triangleright \mathbf{I} \cdot \mathbf{I} \Downarrow_d \mathbf{I}$, $[\,] \triangleright \mathbf{succ2} \cdot \underline{1} \Downarrow_d \underline{3}$ and $[\,] \triangleright \mathbf{add} \cdot \underline{5} \cdot \underline{1} \Downarrow_d \underline{6}$.
3. Since $\mathbf{\Omega}$ is looping, there is no $V \in \mathrm{Val}$ such that $[\,] \triangleright \mathbf{\Omega} \Downarrow_d V$ is derivable.

We now endow EPCF terms with a type system based on simple types.

$$\frac{}{\Gamma, x : \alpha \vdash x : \alpha}\ (\text{ax}) \qquad \frac{}{\Gamma \vdash \mathbf{0} : \mathsf{int}}\ (0) \qquad \frac{\Gamma \vdash M : \alpha \to \alpha}{\Gamma \vdash \mathbf{fix}\, M : \alpha}\ (\text{Y})$$

$$\frac{\Gamma \vdash M : \mathsf{int}}{\Gamma \vdash \mathbf{succ}\, M : \mathsf{int}}\ (+) \qquad \frac{\Gamma \vdash M : \mathsf{int}}{\Gamma \vdash \mathbf{pred}\, M : \mathsf{int}}\ (-) \qquad \frac{\Gamma \vdash L : \mathsf{int} \quad \Gamma \vdash M : \alpha \quad \Gamma \vdash N : \alpha}{\Gamma \vdash \mathbf{ifz}(L, M, N) : \alpha}\ (\text{ifz})$$

$$\frac{\sigma \models \Delta \quad \Gamma, \Delta, x : \alpha \vdash M : \beta}{\Gamma \vdash \lambda x.M\langle\sigma\rangle : \alpha \to \beta}\ (\to_{\text{I}}) \qquad \frac{\Gamma \vdash M : \alpha \to \beta \quad \Gamma \vdash N : \alpha}{\Gamma \vdash M \cdot N : \beta}\ (\to_{\text{E}})$$

$$\frac{}{[] \models \emptyset}\ (\sigma_0) \qquad \frac{\sigma \models \Gamma \quad \rho \models \Delta \quad \Delta \vdash M : \alpha}{\sigma + [x \leftarrow (\rho, M)] \models \Gamma, x : \alpha}\ (\sigma)$$

Fig. 2. EPCF type assignment system.

Definition 3. *1. The set* $\mathbb{T}$ *of* (simple) types *over a* ground type int *is inductively defined by the grammar:*

$$\alpha, \beta \ ::= \ \mathsf{int} \mid \alpha \to \beta \qquad (\mathbb{T})$$

The arrow associates to the right, in other words we write $\alpha_1 \to \cdots \alpha_n \to \beta$ *for* $\alpha_1 \to (\cdots (\alpha_n \to \beta) \cdots)$ *(*$= \vec{\alpha} \to \beta$*, for short).*

2. *A* typing context Γ *is given by a set of associations between variables and types, written* $x_1 : \alpha_1, \ldots, x_n : \alpha_n$*. In this case, we let* $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\}$*. When writing* $\Gamma, x : \alpha$*, we silently assume that* $x \notin \mathrm{dom}(\Gamma)$*.*
3. Typing judgements *are triples, denoted* $\Gamma \vdash M : \alpha$*, where* Γ *is a typing context,* M *is an* EPCF *term and* $\alpha \in \mathbb{T}$*.*
4. Typing derivations *are finite trees built bottom-up in such a way that the root has shape* $\Gamma \vdash M : \alpha$ *and every node is an instance of a rule from Figure 2. In the rule* $(\to_{\text{I}})$ *we assume wlog that* $x \notin \Gamma$*, by* α*-conversion. We also use an auxiliary predicate* $\sigma \models \Gamma$ *whose intuitive meaning is that* Γ *is a typing context constructed from an explicit substitution* σ*.*
5. *When writing* $\Gamma \vdash M : \alpha$*, we mean that this typing judgement is derivable.*
6. *We say that* M is typable *if* $\Gamma \vdash M : \alpha$ *is derivable for some* Γ, α*.*

Example 3. The following are examples of derivable typing judgments.

1.
$$\dfrac{\dfrac{\dfrac{}{[] \models \emptyset}\ (\sigma_0) \quad \dfrac{}{[] \models \emptyset}\ (\sigma_0) \quad \dfrac{}{\vdash \mathbf{0} : \mathsf{int}}\ (0)}{[y \leftarrow ([], \mathbf{0})] \models y : \mathsf{int}}\ (\sigma) \quad \dfrac{}{y : \mathsf{int}, x : \alpha \vdash y : \mathsf{int}}\ (\text{ax})}{\vdash \lambda x.\mathbf{succ}\,(y)\langle [y \leftarrow ([], \mathbf{0})]\rangle : \alpha \to \mathsf{int}}\ (+)\ (\to_{\text{I}})$$

2. $\vdash (\lambda x.\mathbf{succ}\,(x)) \cdot \mathbf{0} : \mathsf{int}$.
3. $\vdash (\lambda sn.s \cdot (s \cdot n)) \cdot (\lambda x.\mathbf{succ}\,(x)) : \mathsf{int} \to \mathsf{int}$.
4. $\vdash \mathbf{fix}\,(\lambda fxy.\mathbf{ifz}(y, x, f \cdot (\mathbf{succ}\, x) \cdot (\mathbf{pred}\, y))) : \mathsf{int} \to \mathsf{int} \to \mathsf{int}$.
5. $\vdash \mathbf{\Omega} : \alpha$, for all $\alpha \in \mathbb{T}$.

The following lemma summarizes the main (rather standard) properties of the language EPCF.

Lemma 1. *Let M be an* EPCF *term, $V \in \mathrm{Val}$, $\alpha, \beta \in \mathbb{T}$ and Γ be a context.*

1. *(Syntax directedness) Every derivable judgement $\Gamma \vdash M : \alpha$ admits a unique derivation.*
2. *(Strengthening) If $\Gamma, x : \beta \vdash M : \alpha$ and $x \notin \mathrm{FV}(M)$ then $\Gamma \vdash M : \alpha$.*
3. *(Subject reduction) For M closed, $\vdash M : \alpha$ and $[\,] \triangleright M \Downarrow_d V$ entail $\vdash V : \alpha$.*

It follows that, if an EPCF program M is typable, then it is also typable in the empty context.

2 Extended addressing machines

We extend the addressing machines from [9] with instructions for performing arithmetic operations and conditional testing. Natural numbers will be represented by particular machines playing the role of numerals.

2.1 Main definitions

We consider fixed a countably infinite set $\mathbb{A}$ of *addresses* together with a distinguished countable subset $\mathbb{X} \subset \mathbb{A}$, such that $\mathbb{A} - \mathbb{X}$ remains infinite. Intuitively, $\mathbb{X}$ is the set of addresses that we reserve for the numerals, therefore hereafter we work under the hypothesis that $\mathbb{X} = \mathbb{N}$, an assumption that we can make without loss of generality.

Definition 4. *Let $\varnothing \notin \mathbb{A}$ be a "null" constant corresponding to an uninitialised register. Set $\mathbb{A}_\varnothing = \mathbb{A} \cup \{\varnothing\}$.*

1. *An* $\mathbb{A}$-valued tape *T is a finite ordered list of addresses $T = [a_1, \ldots, a_n]$ with $a_i \in \mathbb{A}$ for all $i\,(1 \leq i \leq n)$. When $\mathbb{A}$ is clear from the context, we simply call T a tape. We denote by $\mathcal{T}_\mathbb{A}$ the set of all $\mathbb{A}$-valued tapes.*
2. *Let $a \in \mathbb{A}$ and $T, T' \in \mathcal{T}_\mathbb{A}$. We denote by $a :: T$ the tape having a as first element and T as tail. We write $T \mathbin{@} T'$ for the concatenation of T and T', which is an $\mathbb{A}$-valued tape itself.*
3. *Given an index $i \geq 0$, an $\mathbb{A}_\varnothing$-valued* register *R_i is a memory-cell capable of storing either $\varnothing$ or an address $a \in \mathbb{A}$. We write $!R_i$ to represent the value stored in the register R_i. (The notation $!R_i$ is borrowed from OCaml, where ! represents an explicit dereferencing operator.)*
4. *Given $\mathbb{A}_\varnothing$-valued registers $R_0, \ldots, R_n$ for $n \geq 0$, an address $a \in \mathbb{A}$ and an index $i \geq 0$, we write $\vec{R}[R_i := a]$ for the list of registers $\vec{R}$ where the value of R_i has been updated by setting $!R_i = a$. Notice that, whenever $i > n$, we assume that the contents of $\vec{R}$ remains unchanged, i.e. $\vec{R}[R_i := a] = \vec{R}$.*

Addressing machines are endowed with only three instructions (i, j, k, l range over indices of registers):

1. `Load` i : reads an address a from the input tape, assuming it is non-empty, and stores a in the register R_i. If the tape is empty then the machine suspends its execution without raising an error.
2. $k \leftarrow \texttt{App}(i, j)$: reads the addresses a_1, a_2 from R_i and R_j respectively, and stores in R_k the address of the machine obtained by extending the tape of the machine of address a_1 with the address a_2. The resulting address is not calculated internally but rather obtained calling an external *application map*.
3. `Call` i : transfers the computation to the machine having as address the value stored in R_i, whose tape is extended with the remainder of the current machine's tape.

As a general principle, writing on a non-existing register does not cause issues as the value is simply discarded — this is in fact the way one can erase an argument. The attempt of reading an uninitialized register would raise an error — we however show that these kind of errors can be avoided statically (see Lemma 2).

We enrich the above set of instructions with arithmetic operations mimicking the ones present in PCF:

4. $l \leftarrow \texttt{Test}(i, j, k)$: implements the "*is zero?*" test on $!R_i$. Assuming that the value of R_i is an address $n \in \mathbb{N}$, the instruction stores in R_l the value of R_j or R_k, depending on whether $n = 0$.
5. $j \leftarrow \texttt{Pred}(i)$: if $!R_i \in \mathbb{N}$, then the value of R_j becomes $!R_i \oplus 1 = !R_i + 1$.
6. $j \leftarrow \texttt{Succ}(i)$: if $!R_i \in \mathbb{N}$, the value of R_j becomes $!R_i \ominus 1 = \max(!R_i - 1, 0)$.

Notice that the instructions above need R_i to contain a natural number to perform the corresponding operation. However, they are also supposed to work on addresses of machines that compute a numeral. For this reason, the machine whose address is stored in R_i must first be executed, and only if the computation terminates with a numeral is the arithmetic operation performed. Clearly, if the computation terminates in an address not representing a numeral, then an error should be raised at execution time. We will see that these kind of errors can be avoided using a type inference algorithm (see Proposition 1, below).

Definition 5. *1. A* program P *is a finite list of instructions generated by the following grammar, where* ε *represents the empty string and* i, j, k, l *are indices of registers:*

$$\begin{aligned}
\texttt{P} &::= \texttt{Load}\ i;\ \texttt{P} \mid \texttt{A} \\
\texttt{A} &::= k \leftarrow \texttt{App}(i, j);\ \texttt{A} \mid l \leftarrow \texttt{Test}(i, j, k);\ \texttt{A} \mid \\
&\qquad j \leftarrow \texttt{Pred}(i);\ \texttt{A} \mid j \leftarrow \texttt{Succ}(i);\ \texttt{A} \mid \texttt{C} \\
\texttt{C} &::= \texttt{Call}\ i \mid \varepsilon
\end{aligned}$$

Thus, a program starts with a list of `Load`*'s, continues with a list of* `App`*,* `Test`*,* `Pred`*,* `Succ`*, and possibly ends with a* `Call`*. Each of these lists may be empty, in particular the empty program* ε *can be generated.*

2. Let P *be a program,* $r \geq 0$*, and* $\mathcal{I} \subseteq \{0, \ldots, r-1\}$ *be a set of indices corresponding to the indices of initialized registers.*

Define the relation $\mathcal{I} \models^r P$ *as the least relation closed under the rules:*

$$\frac{}{\mathcal{I} \models^r \varepsilon} \qquad \frac{i \in \mathcal{I}}{\mathcal{I} \models^r \texttt{Call}\ i} \qquad \frac{\mathcal{I} \cup \{j\} \models^r \texttt{A} \quad i \in \mathcal{I} \quad j < r}{\mathcal{I} \models^r j \leftarrow \texttt{Pred}(i);\ \texttt{A}}$$

$$\frac{\mathcal{I} \cup \{i\} \models^r \texttt{P} \quad i < r}{\mathcal{I} \models^r \texttt{Load}\ i;\ \texttt{P}} \qquad \frac{\mathcal{I} \models^r \texttt{P} \quad i \geq r}{\mathcal{I} \models^r \texttt{Load}\ i;\ \texttt{P}} \qquad \frac{\mathcal{I} \cup \{j\} \models^r \texttt{A} \quad i \in \mathcal{I} \quad j < r}{\mathcal{I} \models^r j \leftarrow \texttt{Succ}(i);\ \texttt{A}}$$

$$\frac{\mathcal{I} \cup \{l\} \models^r \texttt{A} \quad i,j,k \in \mathcal{I} \quad l < r}{\mathcal{I} \models^r l \leftarrow \texttt{Test}(i,\ j,\ k);\ \texttt{A}} \qquad \frac{\mathcal{I} \cup \{k\} \models^r \texttt{A} \quad i,j \in \mathcal{I} \quad k < r}{\mathcal{I} \models^r k \leftarrow \texttt{App}(i,\ j);\ \texttt{A}}$$

3. *A program* P *is* valid *with respect to* $R_0, \ldots, R_{r-1}$ *if* $\mathcal{R} \models^r P$ *holds for* $\mathcal{R} = \{i \mid R_i \neq \varnothing \ \wedge\ 0 \leq i < r\}$.

Example 4. For each of these programs, we specify its validity with respect to $R_0 = 7, R_1 = a, R_2 = \varnothing$ (i.e., $r = 3$).

1. $P_1 = 2 \leftarrow \texttt{Pred}(0);\ \texttt{Call}\ 2$ (valid)
2. $P_2 = \texttt{Load}\ 2;\ \texttt{Load}\ 8;\ 1 \leftarrow \texttt{Pred}(0);\ 0 \leftarrow \texttt{Test}(0,\ 1,\ 2);\ \texttt{Call}\ 0$ (valid)
3. $P_3 = \texttt{Load}\ 0;\ \texttt{Load}\ 2;\ \texttt{Load}\ 8;\ \texttt{Call}\ 8$ (calling R_8, thus not valid)

Lemma 2. *Given* $\mathbb{A}_\varnothing$*-valued registers* $\vec{R}$ *and a program* P *it is decidable whether* P *is valid w.r.t.* $\vec{R}$.

Proof. The grammar in Definition 5(1) is right-linear, so it is decidable whether P is a production. Also, $r \in \mathbb{N}$ and therefore the set $\mathcal{R}$ in Definition 5(3) is finite. Since P is also finite, the set $\mathcal{R}$ remains finite during the execution of $\mathcal{R} \models^r P$. Decidability follows from these properties, together with the fact that the first instruction of P uniquely determines which rule from Definition 5(2) should be applied (and these rules are exhaustive). □

Definition 6. 1. *An* extended addressing machine *(*EAM*)* M *with* r *registers over* $\mathbb{A}$ *is given by a tuple:*

$$\mathsf{M} = \langle R_0, \ldots, R_{r-1}, P, T \rangle$$

where $\vec{R}$ *are* $\mathbb{A}_\varnothing$*-valued registers,* P *is a program valid w.r.t.* $\vec{R}$ *and* $T \in \mathcal{T}_\mathbb{A}$ *is an (input) tape.*

2. *We write* $\mathsf{M}.r$ *for the number of registers of* M, $\mathsf{M}.R_i$ *for its* i*-th register,* $\mathsf{M}.P$ *for the associated program and* $\mathsf{M}.T$ *for its input tape. When writing "*$R_i = a$*" in a tuple we indicate that* R_i *is present and* $!R_i = a$.
3. *We say that an extended addressing machine* M *as above is* stuck*, written* $\mathsf{stuck}(\mathsf{M})$*, whenever its program has shape* $\mathsf{M}.P = \texttt{Load}\ i; P$ *but its input-tape is empty* $\mathsf{M}.T = []$. *Otherwise* M *is* ready*, written* $\neg\mathsf{stuck}(\mathsf{M})$.
4. *The set of all extended addressing machines over* $\mathbb{A}$ *will be denoted by* $\mathcal{M}_\mathbb{A}$.
5. *For* $n \geq 0$*, the* n-th numeral machine *is defined* $\mathsf{n} = \langle R_0, \varepsilon, [] \rangle$ *with* $!R_0 = n$.
6. *For* $n \geq 0$ *and* $a \in \mathbb{A}$*, define* $\mathsf{Y}_n^a = \langle (R_0 = a, R_1 = \varnothing, \ldots, R_{n+1} = \varnothing, \mathtt{P}, [] \rangle$ *where*
$$\mathtt{P} = \texttt{Load}\ 1; \cdots; \texttt{Load}\ n+1; 0 \leftarrow \texttt{App}(0,\ 1); \cdots; 0 \leftarrow \texttt{App}(0,\ n+1);$$
$$1 \leftarrow \texttt{App}(1,\ 2); \cdots; 1 \leftarrow \texttt{App}(1,\ n+1); 1 \leftarrow \texttt{App}(1,\ 0); \texttt{Call}\ 1$$

Example 5. The following are examples of EAMs (whose registers are assumed uninitialized, i.e. $\vec{R} = \vec{\varnothing}$).

1. $\mathsf{Succ1} := \langle R_0, \mathtt{Load}\ 0; 0 \leftarrow \mathtt{Succ}(0); \mathtt{Call}\ 0, [] \rangle$. Let $a_{\mathsf{S}} = \#\mathsf{Succ1}$.
2. $\mathsf{Succ2} := \langle R_0, R_1, \mathtt{Load}\ 0; \mathtt{Load}\ 1; 1 \leftarrow \mathtt{App}(0,\ 1); 1 \leftarrow \mathtt{App}(0,\ 1); \mathtt{Call}\ 1, [a_{\mathsf{S}}] \rangle$.
3. $\mathsf{Add\_aux} := \langle \vec{R}, P, [] \rangle$ with $\mathsf{Add\_aux}.r = 5$ and $P = \mathtt{Load}\ 0; \mathtt{Load}\ 1; \mathtt{Load}\ 2; 3 \leftarrow \mathtt{Pred}(1); 4 \leftarrow \mathtt{Succ}(2); 0 \leftarrow \mathtt{App}(0,\ 3); 0 \leftarrow \mathtt{App}(0,\ 4); 0 \leftarrow \mathtt{Test}(1,\ 2,\ 0); \mathtt{Call}\ 0$.

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

Definition 7. *Recall that $\mathbb{N}$ stands for an infinite subset of $\mathbb{A}$, here identified with the set of natural numbers, and Y_n^a has been introduced in Definition 6(6).*

1. *Since $\mathcal{M}_{\mathbb{A}}$ is countable, we can fix a bijective function $\# : \mathcal{M}_{\mathbb{A}} \to \mathbb{A}$ satisfying the following conditions:*
   (a) *(Numerals) $\forall n \in \mathbb{N} \,.\, \#\mathsf{n} = n$, where n is the n-th numeral machine;*
   (b) *(Fixed point combinator) for all $n \geq 0$, there exists an address $a \in \mathbb{A} - \mathbb{N}$ such that*
   $$\#(\mathsf{Y}_n^a) = a.$$
   We say that the bijection $\#(\cdot)$ is an address table map *and call the element $\#\mathsf{M}$ the* address of the EAM M. *We simply write Y_n for the machine satisfying the equation above and a_{Y_n} for its address, i.e., $\#(\mathsf{Y}_n) = a_{\mathsf{Y}_n}$.*
2. *For $a \in \mathbb{A}$, we write $\#^{-1}(a)$ for the unique machine having address a, i.e.,*
   $$\#^{-1}(a) = \mathsf{M} \iff \#\mathsf{M} = a.$$
3. *Given $\mathsf{M} \in \mathcal{M}_{\mathbb{A}}$ and $T' \in \mathcal{T}_{\mathbb{A}}$, we write $\mathsf{M} \mathbin{@} T'$ for the machine*
   $$\left\langle \mathsf{M}.\vec{R}, \mathsf{M}.P, \mathsf{M}.T \mathbin{@} T' \right\rangle.$$
4. *Define the* application map $(\cdot) : \mathbb{A} \times \mathbb{A} \to \mathbb{A}$ *by setting $a \cdot b = \#(\#^{-1}(a) \mathbin{@} [b])$. I.e., the* application *of a to b is the unique address c of the EAM obtained by adding b at the end of the input tape of the EAM $\#^{-1}(a)$.*

Remark 1. It is easy to check that there are uncountably many possible address table maps, and some of them are even effective. Depending on the chosen address table map, it might be possible to construct infinite (static) chains of EAMs $(\mathsf{M})_{n \in \mathbb{N}}$, e.g.,
$$\mathsf{M}_n = \langle R_0 = \#\mathsf{M}_{n+1}, \varepsilon, [] \rangle.$$

2.2 Operational semantics

The operational semantics of extended addressing machines is given through a small-step rewriting system. The reduction strategy is deterministic, since the only applicable rule at every step is univocally determined by the first instruction of the internal program, the contents of the registers and the head of the tape.

Unconditional rewriting rules

$$\begin{aligned}
\left\langle \vec{R}, \texttt{Call}\ i; P, T \right\rangle &\to_{\mathsf{c}} \#^{-1}(!R_i) \mathbin{@} T \\
\left\langle \vec{R}, \texttt{Load}\ i; P, a :: T \right\rangle &\to_{\mathsf{c}} \left\langle \vec{R}[R_i := a], P, T \right\rangle \\
\left\langle \vec{R}, k \leftarrow \texttt{App}(i,\ j); P, T \right\rangle &\to_{\mathsf{c}} \left\langle \vec{R}[R_k := !R_i \cdot !R_j], P, T \right\rangle
\end{aligned}$$

Under the assumption that $\#^{-1}(!R_i) \not\to_{\mathsf{c}}$ (i.e., it is in final state).

$$\left\langle \vec{R}, j \leftarrow \texttt{Pred}(i); P, T \right\rangle \to_{\mathsf{c}} \begin{cases} \left\langle \vec{R}[R_j := !R_i \ominus 1, P, T \right\rangle, & \text{if } !R_i \in \mathbb{N}, \\ \texttt{err}, & \text{otherwise.} \end{cases}$$

$$\left\langle \vec{R}, j \leftarrow \texttt{Succ}(i); P, T \right\rangle \to_{\mathsf{c}} \begin{cases} \left\langle \vec{R}[R_j := !R_i \oplus 1, P, T \right\rangle, & \text{if } !R_i \in \mathbb{N}, \\ \texttt{err}, & \text{otherwise.} \end{cases}$$

$$\left\langle \vec{R}, l \leftarrow \texttt{Test}(i,\ j,\ k); P, T \right\rangle \to_{\mathsf{c}} \begin{cases} \left\langle \vec{R}[R_l := !R_j], P, T \right\rangle, & \text{if } !R_i = 0, \\ \left\langle \vec{R}[R_l := !R_k], P, T \right\rangle, & \text{if } !R_i \in \mathbb{N}^+, \\ \texttt{err}, & \text{otherwise.} \end{cases}$$

Under the assumption that $\#^{-1}(!R_i) \to_{\mathsf{c}} \mathsf{A}$ (i.e., it is not in final state).

$$\begin{aligned}
\left\langle \vec{R}, j \leftarrow \texttt{Pred}(i); P, T \right\rangle &\to_{\mathsf{c}} \left\langle \vec{R}[R_i := \#\mathsf{A}], j \leftarrow \texttt{Pred}(i); P, T \right\rangle \\
\left\langle \vec{R}, j \leftarrow \texttt{Succ}(i); P, T \right\rangle &\to_{\mathsf{c}} \left\langle \vec{R}[R_i := \#\mathsf{A}], j \leftarrow \texttt{Succ}(i); P, T \right\rangle \\
\left\langle \vec{R}, l \leftarrow \texttt{Test}(i,\ j,\ k); P, T \right\rangle &\to_{\mathsf{c}} \left\langle \vec{R}[R_i := \#\mathsf{A}], l \leftarrow \texttt{Test}(i,\ j,\ k); P, T \right\rangle
\end{aligned}$$

Fig. 3. Small-step operational semantics for extended addressing machines.

Definition 8. *We introduce a fresh constant* $\texttt{err} \notin \mathcal{M}_{\mathbb{A}}$ *to represent a machine raising an error.*

1. *Define a reduction strategy* $\to_{\mathsf{c}}$ *on EAMs, representing one step of computation, as the least relation* $\to_{\mathsf{c}} \subseteq \mathcal{M}_{\mathbb{A}} \times (\mathcal{M}_{\mathbb{A}} \cup \{\texttt{err}\})$ *closed under the rules in Figure 3.*
2. *The* multistep reduction $\twoheadrightarrow_{\mathsf{c}}$ *is defined as the transitive-reflexive closure of* $\to_{\mathsf{c}}$.
3. *Given* $\mathsf{M}, \mathsf{N}, \mathsf{M} \twoheadrightarrow_{\mathsf{c}} \mathsf{N}$*, we write* $|\mathsf{M} \twoheadrightarrow_{\mathsf{c}} \mathsf{N}| \in \mathbb{N}$ *for the length of the shortest reduction path from* M *to* N.
4. *For* $\mathsf{M}, \mathsf{N} \in \mathcal{M}_{\mathbb{A}}$*, we write* $\mathsf{M} \leftrightarrow_{\mathsf{c}} \mathsf{N}$ *if they have a common reduct* $\mathsf{Z} \in \mathcal{M}_{\mathbb{A}} \cup \{\texttt{err}\}$*, i.e.* $\mathsf{M} \twoheadrightarrow_{\mathsf{c}} \mathsf{Z} \ {}_{\mathsf{c}}\!\!\twoheadleftarrow \mathsf{N}$.
5. *An extended address machine* M*:* is in final state *if it cannot reduce, written* $\mathsf{M} \not\to_{\mathsf{c}}$*;* reaches a final state *if* $\mathsf{M} \twoheadrightarrow_{\mathsf{c}} \mathsf{M}'$ *for some* $\mathsf{M}' \in \mathcal{M}_{\mathbb{A}}$ *in final state;* raises an error *if* $\mathsf{M} \twoheadrightarrow_{\mathsf{c}} \texttt{err}$*;* does not terminate*, otherwise.*

As the redexes in Figure 3 are not overlapping, the confluence of $\twoheadrightarrow_{\mathsf{c}}$ follows easily (cf. [9, Lemma 2.11(2)]).

Lemma 3. *If* $\mathsf{M} \twoheadrightarrow_{\mathsf{c}} \mathsf{M}'$*, then* $\mathsf{M} \mathbin{@} \#\mathsf{N} \twoheadrightarrow_{\mathsf{c}} \mathsf{M}' \mathbin{@} \#\mathsf{N}$.

Proof. By induction on the length of $\mathsf{M} \twoheadrightarrow_{\mathsf{c}} \mathsf{M}'$. □

Example 6. See Example 5 for the definition of $\mathsf{Succ1}, \mathsf{Succ2}, \mathsf{Add\_aux}$.

1. We have $\mathsf{Succ1} \mathbin{@} [0] \twoheadrightarrow_{\mathsf{c}} \mathsf{1}$ and $\mathsf{Succ2} \mathbin{@} [1] \twoheadrightarrow_{\mathsf{c}} \mathsf{3}$.
2. Define $\mathsf{Add} = \mathsf{Y}_0 \mathbin{@} [\#\mathsf{Add\_aux}]$, an EAM performing the addition. We show:
$$\begin{aligned}
&\mathsf{Add} \mathbin{@} [1,3] \\
&\to_{\mathsf{c}} \left\langle \begin{array}{r}(R_0 = a_{\mathsf{Y}_0}, R_1 = \#\mathsf{Add_aux}), 0 \leftarrow \mathtt{App}(0,\,1); \\ 1 \leftarrow \mathtt{App}(1,\,0); \mathtt{Call}\ 1, [1,3]\end{array} \right\rangle \\
&\twoheadrightarrow_{\mathsf{c}} \left\langle \begin{array}{r}\vec{R}, \mathtt{Load}\ 0; \mathtt{Load}\ 1; \mathtt{Load}\ 2; 3 \leftarrow \mathtt{Pred}(1), 4 \leftarrow \mathtt{Succ}(2), 0 \leftarrow \mathtt{App}(0,\,3), \\ 0 \leftarrow \mathtt{App}(0,\,4), 0 \leftarrow \mathtt{Test}(1,\,2,\,0), \mathtt{Call}\ 0, [\#\mathsf{Add}, 1, 3]\end{array} \right\rangle \\
&\twoheadrightarrow_{\mathsf{c}} \left\langle \begin{array}{r}R_0 = \#\mathsf{Add}, R_1 = 1, R_2 = 3, R_3, R_4, 3 \leftarrow \mathtt{Pred}(1), 4 \leftarrow \mathtt{Succ}(2), \\ 0 \leftarrow \mathtt{App}(0,\,3), 0 \leftarrow \mathtt{App}(0,\,4), 0 \leftarrow \mathtt{Test}(1,\,2,\,0), \mathtt{Call}\ 0, []\end{array} \right\rangle \\
&\twoheadrightarrow_{\mathsf{c}} \left\langle \begin{array}{r}R_0 = \#(\mathsf{Add} \mathbin{@} [0,4]\,), R_1 = 1, R_2 = 3, R_3 = 0, R_4 = 4, \\ 0 \leftarrow \mathtt{Test}(1,\,2,\,0), \mathtt{Call}\ 0, []\end{array} \right\rangle \\
&\twoheadrightarrow_{\mathsf{c}} \left\langle \begin{array}{r}R_0 = \#(\mathsf{Add} \mathbin{@} [0,5]\,), R_1 = 0, R_2 = 4, R_3 = 0, R_4 = 5, \\ 0 \leftarrow \mathtt{Test}(1,\,2,\,0), \mathtt{Call}\ 0, []\end{array} \right\rangle \twoheadrightarrow_{\mathsf{c}} \mathsf{4}
\end{aligned}$$
3. For $\mathsf{I} = \langle R_0 = \varnothing, \mathtt{Load}\ 0; \mathtt{Call}\ 0, [] \rangle$, $\mathsf{Y}_0 \mathbin{@} [\#\mathsf{I}] \twoheadrightarrow_{\mathsf{c}} \mathsf{I} \mathbin{@} [\#(\mathsf{Y}_0 \mathbin{@} [\#\mathsf{I}]\,)]$.
4. $\mathsf{Y}_n \mathbin{@} [\#\mathsf{M}, d_1, \ldots, d_n] \twoheadrightarrow_{\mathsf{c}} \mathsf{M} \mathbin{@} [d_1, \ldots, d_n, \#(\mathsf{Y}_n \mathbin{@} [\#\mathsf{M}, d_1, \ldots, d_n]\,)]$, for all $n \geq 0$, $\mathsf{M} \in \mathcal{M}_{\mathbb{A}}$, $\vec{d} \in \mathbb{A}$

3 Typing algorithm

Recall that the set $\mathbb{T}$ of (simple) types has been introduced in Definition 3(1). We now show that certain extended addressing machines can typed, and that typable machines do not raise error during their execution.

Definition 9. *1. A* typing context *Δ is a finite set of associations between registers and types, represented as a list $R_{i_1} : \alpha_1, \ldots, R_{i_n} : \alpha_n$. The indices $i_1, \ldots, i_n$ are not necessarily consecutive.*

2. We denote by $\Delta[R_i : \alpha]$ the typing context Δ where the type associated with R_i becomes α. If R_i is not present in Δ, then $\Delta[R_i : \alpha] = \Delta, R_i : \alpha$.

3. Let Δ be a typing context, $\mathsf{M} \in \mathcal{M}_{\mathbb{A}}$, P be a program, $T \in \mathcal{T}_{\mathbb{A}}$ and $\alpha \in \mathbb{T}$. We define the typing judgements

$$\Delta \vdash \mathsf{M} : \alpha \qquad\qquad \Delta \Vdash (P, T) : \alpha$$

by mutual induction as the least relations closed under the rules of Figure 4. The rules (nat) *and* (fix) *are the base cases and they take precedence over* $(R_\varnothing)$ *and* $(R_{\mathbb{T}})$.

4. For $R_{i_1}, \ldots, R_{i_n} \in \vec{R}$, write $R_{i_1} : \beta_{i_1}, \ldots, R_{i_n} : \beta_{i_n} \models \vec{R}$ whenever $\#^{-1}(!R_j) : \beta_j$, for all $j \in \{i_1, \ldots, i_n\}$.

The algorithm in Figure 4 deserves some discussion. As it is presented as a set of inference rules, one should reason bottom-up. To give a machine M a type α, one needs to derive the judgement $\vdash \mathsf{M} : \alpha$. The machines n and Y_n, recognizable

$$\frac{\#\mathsf{M} \in \mathbb{N}}{\vdash \mathsf{M} : \mathsf{int}}\ \text{nat} \qquad \frac{\#\mathsf{M} = a_{\mathsf{Y}_n} \quad \vec{\delta} = \delta_1 \to \cdots \to \delta_n}{\vdash \mathsf{M} : (\vec{\delta} \to \alpha \to \alpha) \to \vec{\delta} \to \alpha}\ \text{fix}_n \qquad \frac{\Delta \Vdash (P,T) : \alpha}{\Delta \vdash \langle (), P, T\rangle : \alpha}\ R_{()}$$

$$\frac{\Delta \vdash \langle R_0, \ldots, R_{r-1}, P, T\rangle : \alpha \quad !R_r = \varnothing}{\Delta \vdash \langle (R_0, \ldots, R_r), P, T\rangle : \alpha}\ R_{\varnothing}$$

$$\frac{R_r : \beta, \Delta \vdash \langle R_0, \ldots, R_{r-1}, P, T\rangle : \alpha \quad \vdash \#^{-1}(!R_r) : \beta}{\Delta \vdash \langle (R_0, \ldots, R_r), P, T\rangle : \alpha}\ R_{\mathbb{T}}$$

$$\frac{\Delta[R_i : \beta] \Vdash (P, []) : \alpha}{\Delta \Vdash (\texttt{Load}\ i; P, []) : \beta \to \alpha}\ \text{load}_{\varnothing}$$

$$\frac{\Delta[R_i : \beta] \Vdash (P,T) : \alpha \quad \#^{-1}(a) : \beta}{\Delta \Vdash (\texttt{Load}\ i; P, a :: T) : \alpha}\ \text{load}_{\mathbb{T}}$$

$$\frac{\Delta[R_j : \mathsf{int}], R_i : \mathsf{int} \Vdash (P,T) : \alpha}{\Delta, R_i : \mathsf{int} \Vdash (j \leftarrow \texttt{Pred}(i); P, T) : \alpha}\ \text{pred}$$

$$\frac{\Delta[R_j : \mathsf{int}], R_i : \mathsf{int} \Vdash (P,T) : \alpha}{\Delta, R_i : \mathsf{int} \Vdash (j \leftarrow \texttt{Succ}(i); P, T) : \alpha}\ \text{succ}$$

$$\frac{(\Delta, R_i : \mathsf{int}, R_j : \beta, R_k : \beta)[R_l : \beta] \Vdash (P,T) : \alpha}{\Delta, R_i : \mathsf{int}, R_j : \beta, R_k : \beta \Vdash (l \leftarrow \texttt{Test}(i,\ j,\ k); P, T) : \alpha}\ \text{test}$$

$$\frac{\Delta[R_k : \beta], R_i : \alpha \to \beta, R_j : \alpha \Vdash (P,T) : \Delta}{\Delta, R_i : \alpha \to \beta, R_j : \alpha \Vdash (k \leftarrow \texttt{App}(i,\ j); P, T) : \Delta}\ \text{app}$$

$$\frac{\vdash \mathsf{M}_1 : \alpha_1 \quad \cdots \quad \vdash \mathsf{M}_n : \alpha_n}{\Delta, R_i : \alpha_1 \to \cdots \to \alpha_n \to \alpha \Vdash (\texttt{Call}\ i, [\#\mathsf{M}_1, \ldots, \#\mathsf{M}_n]) : \alpha}\ \text{call}$$

Fig. 4. Typing rules for extended addressing machines.

from their addresses, are special cases as they represent numerals and a recursion operator, respectively. A type can be associated with these machines without inspecting their structure, which explains why the rules (nat) and (fix) have higher precedence. Otherwise, the rule ($R_{\mathbb{T}}$) allows to check whether the value in a register is typable and only retain its type, the rule ($R_{\varnothing}$) allows to get rid of uninitialized registers. Once this initial step is performed, one needs to derive a judgement of the form $R_{i_1} : \beta_{i_1}, \ldots, R_{i_n} : \beta_{i_n} \Vdash (P,T) : \alpha$, where P and T are the program and the input tape of the original machine respectively. This is done by verifying the coherence of the instructions in the program with the types of the registers and of the values in the input tape.

As a final consideration, notice that the rules in Figure 4 can only be considered as an algorithm when the address table map is effectively given. Otherwise, the algorithm would depend on an oracle deciding $a = \#\mathsf{M}$.

Remark 2. 1. For all $\mathsf{M} \in \mathcal{M}_{\mathbb{A}}$ and $\alpha \in \mathbb{T}$, we have $\vdash \mathsf{M} : \alpha$ if and only if there exists $a \in \mathbb{A}$ such that both $\#^{-1}(a) : \alpha$ and $\#\mathsf{M} = a$ hold.

2. If $\#\mathsf{M} \notin \mathbb{N} \cup \{a_{\mathsf{Y}_n} \mid n \geq 0\}$, then we have $\vdash \mathsf{M} : \alpha$ if and only if there exists Δ such that $\Delta \models \mathsf{M}.\vec{R}$ and $\Delta \Vdash (\mathsf{M}.P, \mathsf{M}.T) : \alpha$.

Example 7. The following typing judgements are derivable.

1. $\vdash \mathsf{Succ2} : \mathsf{int} \to \mathsf{int}$
2. $\vdash \mathsf{Add} : \mathsf{int} \to \mathsf{int} \to \mathsf{int}$, where $\mathsf{Add} = \mathsf{Y}_0 \mathbin{@} [\#\mathsf{Add\_aux}]$
3. For a smaller example, like $\vdash \mathsf{Succ1} : \mathsf{int} \to \mathsf{int}$, we can provide the whole derivation tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{R_0 : \mathsf{int} \Vdash \langle \mathtt{Call}\ 0, [] \rangle : \mathsf{int}}\ \text{call}}{R_0 : \mathsf{int} \Vdash \langle 0 \leftarrow \mathtt{Succ}(0); \mathtt{Call}\ 0, [] \rangle : \mathsf{int}}\ \text{succ}}{\Vdash \langle \mathtt{Load}\ 0; 0 \leftarrow \mathtt{Succ}(0); \mathtt{Call}\ 0, [] \rangle : \mathsf{int} \to \mathsf{int}}\ \text{load}_\varnothing}{\vdash \langle (), \mathtt{Load}\ 0; 0 \leftarrow \mathtt{Succ}(0); \mathtt{Call}\ 0, [] \rangle : \mathsf{int} \to \mathsf{int}}\ R_() \qquad !R_0 = \varnothing}{\vdash \langle (R_0 = \varnothing), \mathtt{Load}\ 0; 0 \leftarrow \mathtt{Succ}(0); \mathtt{Call}\ 0, [] \rangle : \mathsf{int} \to \mathsf{int}}\ R_\varnothing$$

Lemma 4. *Let* $\mathsf{M} \in \mathcal{M}_{\mathbb{A}}$, $\alpha \in \mathbb{T}$. *Assume that* $\# : \mathsf{M} \to \mathbb{A}$ *is effectively given.*

1. *If* $\mathsf{M} = \langle \vec{R} = \varnothing, P, [] \rangle$ *then the typing algorithm is capable of deciding whether* $\vdash \mathsf{M} : \alpha$ *holds.*
2. *In general, the typing algorithm semi-decides whether* $\vdash \mathsf{M} : \alpha$ *holds.*

Proof. (Sketch) (i) In this case, $\vdash \mathsf{M} : \alpha$ holds if and only if $\Vdash (\mathsf{M}.P, [])$ does. By induction on the length of $\mathsf{M}.P$, one verifies if it is possible to construct a derivation. Otherwise, conclude that $\vdash \mathsf{M} : \alpha$ is not derivable.

(ii) In the rules $(R_{\mathbb{T}})$ and (Load_T), one needs to show that a type for the premises exists. As the set of types is countable, and effectively given, one can easily design an algorithm constructing a derivation tree (by dovetailing). However, the algorithm cannot terminate when executed on M_0 from Remark 1. □

The machine M_0 from Remark 1 cannot be typable because it would require an infinite derivation tree.

Proposition 1. *Let* $\mathsf{M}, \mathsf{M}', \mathsf{N}, \in \mathcal{M}_{\mathbb{A}}$ *and* $\alpha, \beta \in \mathbb{T}$.

1. *If* $\vdash \mathsf{M} : \beta \to \alpha$ *and* $\vdash \mathsf{N} : \beta$ *then* $\vdash \mathsf{M} \mathbin{@} [\#\mathsf{N}] : \alpha$.
2. *If* $\vdash \mathsf{M} : \alpha$ *and* $\mathsf{M} \to_{\mathsf{c}} \mathsf{N}$ *then* $\vdash \mathsf{N} : \alpha$.
3. *If* $\vdash \mathsf{M} : \alpha$ *then* M *does not raise an error.*
4. *If* $\vdash \mathsf{M} : \mathsf{int}$ *then either* M *does not terminate or* $\mathsf{M} \twoheadrightarrow_{\mathsf{c}} \mathsf{n}$, *for some* $n \geq 0$.

Proof. (i) Simultaneously, one proves that $\Delta \Vdash (P, T) : \beta \to \alpha$ and $\vdash \mathsf{N} : \beta$ imply $\Delta \Vdash (P, T \mathbin{@} [\#\mathsf{N}]) : \alpha$. Proceed by induction on a derivation of $\vdash \mathsf{M} : \beta \to \alpha$ (resp. $\Delta \Vdash (P, T) : \beta \to \alpha$).

Case (nat) is vacuous.

Case (fix_n). We show the case for $n = 0$, the others being similar. By definition of Y_0, we have:

$$\mathsf{Y}_0 \mathbin{@} [\#\mathsf{N}] = \langle (\varnothing, a_{\mathsf{Y}_0}), \mathtt{Load}\ 0; 1 \leftarrow \mathtt{App}(1,\ 0); 0 \leftarrow \mathtt{App}(0,\ 1); \mathtt{Call}\ 0, [\#\mathsf{N}] \rangle .$$

Notice that, in this case, $\beta = \alpha \to \alpha$. Using $\#^{-1}(a_{\mathsf{Y}_0}) = \mathsf{Y}_0$, we derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{R_0 : \alpha, R_1 : \alpha \Vdash (\texttt{Call}\ 0, []) : \alpha}{R_0 : \alpha \to \alpha, R_1 : \alpha \Vdash (0 \leftarrow \texttt{App}(0,\ 1); \texttt{Call}\ 0, []) : \alpha}\ \text{app}}{R_0 : \alpha \to \alpha, R_1 : (\alpha \to \alpha) \to \alpha \Vdash (1 \leftarrow \texttt{App}(1,\ 0); \cdots, []) : \alpha}\ \text{app} \qquad \vdash \mathsf{N} : \alpha \to \alpha}{R_1 : (\alpha \to \alpha) \to \alpha \Vdash (\texttt{Load}\ 0; \cdots, [\#\mathsf{N}]) : \alpha \qquad \vdash \mathsf{Y}_0 : (\alpha \to \alpha) \to \alpha}\ \text{load}_{\mathbb{T}}}{\vdash \langle (R_0 = \varnothing, R_1 = a_{\mathsf{Y}_0}), \texttt{Load}\ 0; 1 \leftarrow \texttt{App}(1,\ 0); 0 \leftarrow \texttt{App}(0,\ 1); \texttt{Call}\ 0, [\#\mathsf{N}] \rangle : \alpha}\ R_*$$

Case $\text{load}_\varnothing$. Then $P = \texttt{Load}\ i; P'$, $T = []$ and $\Delta[R_i : \beta] \Vdash (P', []) : \alpha$ holds. By assumption $\vdash \mathsf{N} : \beta$, so we conclude $\Delta \Vdash (\texttt{Load}\ i; P', []) : \alpha$ by applying $\text{load}_{\mathbb{T}}$.

All other cases derive straightforwardly from the induction hypothesis.

(ii) The cases $\mathsf{M} = \mathsf{Y}_n$ or $\mathsf{M} = \mathsf{n}$ for some $n \in \mathbb{N}$ are vacuous, as these machines are in final state. Otherwise, by Remark 2(2), we get $\Delta \Vdash (\mathsf{M}.P, \mathsf{M}.T) : \alpha$ for some $\Delta \models \mathsf{M}.\vec{R}$. By cases on the shape of $\mathsf{M}.P$.

Case $P = \texttt{Load}\ i; P'$. Then $\mathsf{M}.T = a :: T'$ otherwise M would be in final state, and $\mathsf{N} = \langle \vec{R}[R_i := a], P', T' \rangle$. From $(\text{Load}_{\mathbb{T}})$ we get $\Delta[R_i : \beta] \Vdash (P', T') : \alpha$ for some $\beta \in \mathbb{T}$ satisfying $\#^{-1}(a) : \beta$. As $\Delta \models \vec{R}$ we derive $\Delta[R_i : \beta] \models \vec{R}[R_i := a]$, therefore this case follows from the induction hypothesis.

Case $P = \texttt{Call}\ i; P'$. Then $R_i : \alpha_1 \to \cdots \to \alpha_n \to \alpha$, $T = [\#\mathsf{M}_1, \ldots, \#\mathsf{M}_n]$ and $\vdash \mathsf{M}_j : \alpha_j$, for all $j \leq n$. In this case, we have $\mathsf{N} = \#^{-1}(!(\mathsf{M}.R_i)) \mathbin{@} T$ with $\vdash \#^{-1}(!(\mathsf{M}.R_i)) : \alpha_1 \to \cdots \to \alpha_n \to \alpha$, so we conclude by (i).

All other cases follows easily from the induction hypothesis.

(iii) The three cases from Figure 4 where a machine can raise an error are ruled out by the typing rules (pred), (succ) and (test), respectively. Therefore, no error can be raised during the execution.

(iv) Assume that $\vdash \mathsf{M} : \mathsf{int}$ and $\mathsf{M} \twoheadrightarrow_{\mathsf{c}} \mathsf{N}$ for some N in final state. By (ii), we obtain that $\vdash \mathsf{N} : \mathsf{int}$ holds, therefore $\mathsf{N} = \mathsf{n}$ since numerals are the only machines in final state typable with int. □

4 Translation and simulation

We define a type-preserving translation from EPCF terms to extended addressing machines. More precisely, we show that if $\Gamma \vdash M : \alpha$ is derivable then M is transformed into a machine M which is typable with the same α. By Proposition 1, M never raises a runtime error and well–typedness is preserved during its execution. We then show that if a well-typed EPCF program M computes a value $\underline{n}$, then its translation M reduces to the corresponding EAM n. Finally, this result is transported to PCF using their equivalence on programs of type int.

We start by showing that EAMs implementing the main PCF instructions are definable. We do not need any machinery for representing explicit substitutions because they are naturally modelled by the evaluation strategy of EAMs.

Lemma 5. *Let $n \geq 0$. There are EAMs satisfying (for all $a, b, c, d_1, \ldots, d_n \in \mathbb{A}$):*

1. $\mathsf{Proj}_i^n \mathbin{@} [d_1, \ldots, d_n] \twoheadrightarrow_{\mathsf{c}} d_i$, *for* $1 \leq i \leq n$;
2. $\mathsf{Apply}_n \mathbin{@} [a, b, d_1, \ldots, d_n] \twoheadrightarrow_{\mathsf{c}} \#^{-1}(a) \mathbin{@} [d_1, \ldots, d_n, b \cdot d_1 \cdots d_n]$;

3. $\mathsf{Pred}_n \mathbin{@} [a, d_1, \ldots, d_n] \twoheadrightarrow_{\mathsf{c}} \langle R_0 = a \cdot d_1 \cdots d_n, \vec{R}, ; 0 \leftarrow \mathtt{Pred}(0); \mathtt{Call}\ 0, [] \rangle$;
4. $\mathsf{Succ}_n \mathbin{@} [a, d_1, \ldots, d_n] \twoheadrightarrow_{\mathsf{c}} \langle R_0 = a \cdot d_1 \cdots d_n, \vec{R}, ; 0 \leftarrow \mathtt{Succ}(0); \mathtt{Call}\ 0, [] \rangle$;
5. $\mathsf{Ifz}_n \mathbin{@} [a, b, c, d_1, \ldots, d_n] \twoheadrightarrow_{\mathsf{c}} \left\langle \begin{array}{r} R_0 = a \cdot \vec{d}, R_1 = b \cdot \vec{d}, R_2 = c \cdot \vec{d}, \vec{R}, \\ 0 \leftarrow \mathtt{Test}(0,\ 1,\ 2); \mathtt{Call}\ 0, [] \end{array} \right\rangle$;

Proof. Easy. As an example, we give a possible definition of the predecessor:

$$\mathsf{Pred}_n = \left\langle \begin{array}{r} R_0, \ldots, R_n, \mathtt{Load}\ 0; \cdots; \mathtt{Load}\ n; 0 \leftarrow \mathtt{App}(0,\ 1); \cdots; \\ 0 \leftarrow \mathtt{App}(0,\ n); 0 \leftarrow \mathtt{Pred}(0); \mathtt{Call}\ 0, [] \end{array} \right\rangle$$

The others are similar. □

Lemma 6. *The EAMs in the previous lemma can be defined in order to ensure their typability (for all $n \geq 0$, $\alpha, \beta, \gamma, \delta_i \in \mathbb{T}$):*

1. $\vdash \mathsf{Proj}_i^n : \vec{\delta} \to \delta_i$, *with* $\vec{\delta} = \delta_1 \to \cdots \to \delta_n$;
2. $\vdash \mathsf{Apply}_n : (\vec{\delta} \to \beta \to \alpha) \to (\vec{\delta} \to \beta) \to \vec{\delta} \to \alpha$;
3. $\vdash \mathsf{Pred}_n : (\vec{\delta} \to \mathsf{int}) \to \vec{\delta} \to \mathsf{int}$;
4. $\vdash \mathsf{Succ}_n : (\vec{\delta} \to \mathsf{int}) \to \vec{\delta} \to \mathsf{int}$;
5. $\vdash \mathsf{Ifz}_n : (\vec{\delta} \to \mathsf{int}) \to (\vec{\delta} \to \alpha) \to (\vec{\delta} \to \alpha) \to \vec{\delta} \to \alpha$;

Proof. Easy. The naive implementations are, in fact, typable. □

Using the auxiliary EAMs given above, we can translate any EPCF term proceeding as follows. Intuitively, an EPCF term M having $x_1, \ldots, x_n$ as free variables is translated as an EAM M loading n arguments as input.

Definition 10. *(Translation) Let M be an* EPCF *term, whose free variables are among $\vec{x} = x_1, \ldots, x_n$. The* translation *of M (w.r.t. $\vec{x}$) is a machine $[\![M]\!]^{\vec{x}} \in \mathcal{M}_{\mathbb{A}}$ defined by structural induction on M as follows:*

$$\begin{array}{ll}
(\!|\sigma + [y \leftarrow (\tau, N)], M|\!)^{\vec{x}} & = (\!|\sigma, M|\!)^{y,\vec{x}} \mathbin{@} [\#(\!|\tau, N|\!)]\,; \\
(\!|[\,], M|\!)^{\vec{x}} & = [\![M]\!]^{\vec{x}}; \\
[\![x_i]\!]^{\vec{x}} & = \mathsf{Proj}_i^n, \\
[\![\lambda y.M\langle\sigma\rangle]\!]^{\vec{x}} & = (\!|\sigma, M|\!)^{\vec{x},y}, \ \textit{where wlog } y \notin \vec{x}; \\
[\![M \cdot N]\!]^{\vec{x}} & = \mathsf{Apply}_n \mathbin{@} [\#[\![M]\!]^{\vec{x}}, \#[\![N]\!]^{\vec{x}}]\,; \\
[\![\underline{k}]\!]^{\vec{x}} & = \mathsf{Proj}_1^{n+1} \mathbin{@} [k]\,, \ \textit{where } k \in \mathbb{N}; \\
[\![\mathbf{pred}\, M]\!]^{\vec{x}} & = \mathsf{Pred}_n \mathbin{@} [\#[\![M]\!]^{\vec{x}}]\,; \\
[\![\mathbf{succ}\, M]\!]^{\vec{x}} & = \mathsf{Succ}_n \mathbin{@} [\#[\![M]\!]^{\vec{x}}]\,; \\
[\![\mathbf{ifz}(L, M, N)]\!]^{\vec{x}} & = \mathsf{Ifz}_n \mathbin{@} [\#[\![L]\!]^{\vec{x}}, \#[\![M]\!]^{\vec{x}}, \#[\![N]\!]^{\vec{x}}]\,; \\
[\![\mathbf{fix}\, M]\!]^{\vec{x}} & = \mathsf{Y}_n \mathbin{@} [\#[\![M]\!]^{\vec{x}}]\,.
\end{array}$$

We show the extended abstract machines associated by this translation to some of our running examples.

Example 8. 1. $[\![(\lambda x.\mathbf{succ}\,(x)\langle\rangle)\cdot\underline{0}]\!] = \mathsf{Succ}_1 @ [\#\mathsf{Proj}^1_1, 0]\,.$
2. $[\![(\lambda sn.s(sn))(\lambda x.\mathbf{succ}\,(x))]\!] =$

$$\mathsf{Apply}_2 @ [\#\mathsf{Proj}^2_1, \#(\mathsf{Apply}_2 @ [\#\mathsf{Proj}^2_1, \#\mathsf{Proj}^2_2]\,), \#(\mathsf{Succ}_1 @ [\#\mathsf{Proj}^1_1]\,)]\,.$$

3. $$\begin{aligned}[\![\mathbf{add}]\!] &= \mathsf{Y}_0 @ \#[\![\lambda fxy.\mathbf{ifz}(y, x, (f\cdot(\mathbf{succ}\,x)\cdot(\mathbf{pred}\,y)))]\!]\\ &= \mathsf{Y}_0 @ \#[\![\mathbf{ifz}(y, x, (f\cdot(\mathbf{succ}\,x)\cdot(\mathbf{pred}\,y)))]\!]^{f,x,y}\,,\\ &= \mathsf{Y}_0 @ \#(\mathsf{Ifz}_3 @ [\#\mathsf{Proj}^3_3, \#\mathsf{Proj}^3_2, \#[\![f\cdot(\mathbf{succ}\,x)\cdot(\mathbf{pred}\,y)]\!]^{f,x,y}\,]\,),\end{aligned}$$

where $[\![f\cdot(\mathbf{succ}\,x)\cdot(\mathbf{pred}\,y)]\!]^{f,x,y} =$

$$\mathsf{Apply}_3 @ [\#(\mathsf{Apply}_3 @ [\#\mathsf{Proj}^3_1, \#(\mathsf{Succ}_3 @ [\#\mathsf{Proj}^3_2]\,)]\,), \#(\mathsf{Pred}_3 @ [\#\mathsf{Proj}^3_3]\,)]$$

Theorem 1. *Let M be an* EPCF *term, $\alpha \in \mathbb{T}$, $\Gamma = x_1 : \beta_1, \ldots, x_n : \beta_n$. Then*

$$\Gamma \vdash M : \alpha \quad \Rightarrow \quad \vdash [\![M]\!]^{x_1,\ldots,x_n} : \beta_1 \to \cdots \to \beta_n \to \alpha.$$

Proof. By induction on a derivation of $\Gamma \vdash M : \alpha$. As an induction loading, one needs to prove simultaneously that for all explicit substitutions σ with $\mathrm{dom}(\sigma) = \{x_1, \ldots, x_n\}$, if $\sigma \models x_1 : \beta_1, \ldots, x_n : \beta_n$ then $\vdash (\!|\sigma(x_i)|\!) : \beta_i$, for all $i \leq n$. □

Theorem 2. *Let M be an* EPCF *term and $V \in \mathrm{Val}$. Then*

$$\sigma \triangleright M \Downarrow_d V \Rightarrow (\!|\sigma, M|\!) \leftrightarrow_{\mathsf{c}} [\![V]\!]$$

Proof. By induction on a derivation of $\sigma \triangleright M \Downarrow_d V$. □

Corollary 1. *For an* EPCF *program M of type $\vdash M : \mathsf{int}$ we have*

$$[\,] \triangleright M \Downarrow_d \underline{n} \Rightarrow [\![M]\!] \twoheadrightarrow_{\mathsf{c}} \mathsf{n}$$

Proof. Assume $[\,] \triangleright M \Downarrow_d \underline{n}$. By Theorem 2, we have $(\!|[\,], M|\!) \leftrightarrow_{\mathsf{c}} [\![\underline{n}]\!]$. Since $[\![\underline{n}]\!] \twoheadrightarrow_{\mathsf{c}} \mathsf{n}$ and the numeral machine n is in final state we conclude $[\![M]\!] \twoheadrightarrow_{\mathsf{c}} \mathsf{n}$. □

4.1 Applying the translation to regular PCF

We now show how to apply our machinery to the usual PCF: we write P, Q for its terms, U for its values, $\vdash^{\mathsf{PCF}}$ for its type system, and $P \Downarrow U$ for its big-step (call-by-name) operational semantics. For the precise rules, we refer to [15] (or the technical appendix). Recall that any PCF program P can be seen as an EPCF term, thanks to the notation $\lambda x.N := \lambda x.N\langle\rangle$. However, the hypotheses $P \vdash^{\mathsf{PCF}} P : \mathsf{int}$ and $P \Downarrow \underline{n}$ are *a priori* not sufficient for applying Corollary 1, since one needs to show that also the corresponding EPCF judgments $\vdash P : \mathsf{int}$ and $P \Downarrow_d \underline{n}$ hold. The former is established by the following lemma.

Lemma 7. *Let M be a* PCF *term, $\alpha \in \mathbb{T}$ and Γ be a context. Then*

$$\Gamma \vdash^{\mathsf{PCF}} M : \alpha \qquad \Rightarrow \qquad \Gamma \vdash M : \alpha$$

Proof. By a straightforward induction on a derivation of $\Gamma \vdash^{\mathsf{PCF}} M$. □

An EPCF term is easily translated into PCF by performing all its explicit substitutions. The converse is trickier as the representation is not unique: for every PCF term P there are several decompositions $P = P'[Q_1/x_1, \ldots, Q_n/x_n]$.

Definition 11. *1. Let M be an* EPCF *term and σ be an explicit substitution.*

(a) The sizes $|-|$ of σ, M and (σ, M) are defined by mutual induction, e.g.

$$|[]| = 0, \qquad |\rho[x \leftarrow (\rho', N)]| = |\rho| + |(\rho', N)|,$$
$$|(\sigma, M)| = |\sigma| + |M|, \qquad |\lambda x.M\langle\sigma\rangle| = |\sigma, M| + 1,$$

and the other cases of $|M|$ are standard whence they are omitted.

(b) Define a PCF *term $(\sigma, M)^*$ by induction on $|(\sigma, M)|$ as follows:*

$$(\sigma, x)^* = \begin{cases} \sigma(x)^*, & \text{if } x \in \mathrm{dom}(\sigma), \\ x, & \text{otherwise}, \end{cases}$$

$$(\sigma, \lambda x.M\langle\rho\rangle)^* = \lambda x.(\sigma + \rho, M)^*, \qquad (\sigma, M \cdot N)^* = (\sigma, M)^* \cdot (\sigma, N)^*,$$
$$(\sigma, \underline{0})^* = \underline{0}, \qquad (\sigma, \mathbf{pred}\, M)^* = \mathbf{pred}\, (\sigma, M)^*,$$
$$(\sigma, \mathbf{fix}\, M)^* = \mathbf{fix}\, ((\sigma, M)^*), \qquad (\sigma, \mathbf{succ}\, M)^* = \mathbf{succ}\, (\sigma, M)^*,$$
$$(\sigma, \mathbf{ifz}(L, M, N))^* = \mathbf{ifz}((\sigma, L)^*, (\sigma, M)^*, (\sigma, N)^*).$$

2. Given a PCF *term P, define $P^\dagger = \{(\sigma, M) \mid (\sigma, M)^* = P\}$.*

Theorem 3. *The big-step weak reduction of* EPCF *is equivalent to the usual big-step operational semantics of* PCF*. (See the technical appendix.)*

1. Given an EPCF *program M, value V and an explicit substitution σ:*

$$\sigma \triangleright M \Downarrow_d V \Rightarrow (\sigma, M)^* \Downarrow ([\,], V)^*$$

2. Given a PCF *program P and* PCF *value U:*

$$P \Downarrow U \Rightarrow \forall (\sigma, M) \in P^\dagger,\ \exists V \in \mathrm{Val}\,.\,(\ \sigma \triangleright M \Downarrow_d V \text{ and } ([\,], V) \in U^\dagger\ )$$

Proof. 1. First show $(\sigma + [x \leftarrow (\rho, N)], M)^* = (\sigma, M)^*[(\rho, N)^*/x]$ by structural induction on M. Then proceed by induction on a derivation of $\sigma \triangleright M \Downarrow_d V$.

2. By induction on the lexicographically ordered pairs, whose first component is the length of a derivation of $P \Downarrow U$ and second component is $|(\sigma, M)|$. In fact, if $M = y$ and $\sigma(y) = (\rho, N)$, with $(\rho, N) \in P^\dagger$, the length of the derivation $(\rho, N)^* \Downarrow U$ remains unchanged, while $|(\rho, N)| < |(\sigma, M)|$. Then the case follows from the induction hypothesis by applying (var).

Case $P \Downarrow U$ since $P = U = \lambda x.P_0$. Since $(\sigma, M) \in P^\dagger$, we have $M = \lambda x.M_0\langle\rho\rangle$ with $P_0 = (\sigma + \rho, M_0)^*$. Setting $V = \lambda x.M_0\langle\sigma + \rho\rangle$ we obtain $M \Downarrow_d V$ by (fun) with $([], V)^* = \lambda x.(\sigma + \rho, M_0) = \lambda x.P_0 = U$.

Case $P \Downarrow U$ since $P = U = \underline{n}$. Analogous, but simpler.

All other cases follow easily from the induction hypothesis on $P \Downarrow U$. □

As promised, we now draw conclusions for the regular PCF. As customary in PCF, we are interested on the properties of closed terms having ground type.

Theorem 4. *For a* PCF *program P of type* int*, $P \Downarrow \underline{n}$ entails $[\![P]\!] \twoheadrightarrow_{\mathsf{c}} \mathsf{n}$.*

Proof. Note that P is also an EPCF term such that $([], P) \in P^\dagger$, and that $\vdash P : \mathsf{int}$ by Lemma 7. Thus $[\,] \triangleright P \Downarrow_d \underline{n}$ by Theorem 3(2). Conclude by Corollary 1. □

5 Conclusions and future works

The Full Abstraction (FA) Problem for PCF, originally proposed by Robin Milner in [14] (see also [16,5,6]), can be informally formulated as the problem of finding a "natural" model of PCF which is fully abstract w.r.t. the operational semantics. More precisely, two programs sharing the type α get the same denotation in a FA model if and only if they are observationally indistinguishable when plugged in the same context $C[]$ of type $\alpha \to \mathsf{int}$. In the same paper, Milner proves that a fully abstract model can be obtained by a suitable completion of the "term model" construction, and that this model is unique up to isomorphism (under some natural conditions). The question is therefore whether it is possible to obtain this model using a direct "semantic" construction.

However, quoting from [3],

> "the problem is to understand what should be meant by a semantic characterization [...] Our view is that the essential content of the problem, what makes it important, is that it calls for a semantic characterization of sequential, functional computation at higher-types".

A celebrated result is that fully abstract models of PCF can be obtained by constructing suitable categories of games [4,3,11].

We believe that the formalism of extended addressing machines introduced in this paper may contribute to the understanding of what can be considered as a sequential higher-order computation. Indeed, EAMs are certainly sequential computational devices and higher-order computations are simply achieved by passing the address of a machine having a functional behaviour. An objection that could be raised is that EAMs essentially incorporate the contents of the rewriting rules defining the operational semantics, whence they can be seen as a representation of PCF operational semantics in disguise. We believe that this kind of criticism is due to an oversimplistic view. For instance, the fundamental difference in the way one handles variables and registers assignments in typing contexts shows that some machines can be hardly seen as representing subterms of the original PCF program. Another essential difference lies in the semantics of the recursion operator, which is achieved in the operational semantics through a (hidden) self-application, and in Scott-continuous models by computing the least fixed point of a continuous function. In the EAMs formalism the application mechanism is managed via a suitable assignment of addresses which by no means can be considered implicit in the operational semantics. Indeed the address table map is an arbitrary bijective function, constrained to some abstract conditions. As it is easy to show, there exist adequate address table maps having arbitrary computational complexity. With this mechanism in place, the semantics of the fixed point operator is obtained by specifying a suitable EAM having its own address stored inside. In future works we plan to examine whether EAMs can be turned into a FA model of PCF by quotienting the set $\mathcal{M}_{\mathbb{A}}$ w.r.t. a suitable observational equivalence.

References

1. Abadi, M., Cardelli, L., Curien, P.L., Lévy, J.J.: Explicit substitutions. In: Allen, F.E. (ed.) Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990. pp. 31–46. ACM Press (1990). https://doi.org/10.1145/96709.96712, https://doi.org/10.1145/96709.96712
2. Abadi, M., Cardelli, L., Curien, P.L., Lévy, J.J.: Explicit substitutions. J. Funct. Program. **1**(4), 375–416 (1991). https://doi.org/10.1017/S0956796800000186, https://doi.org/10.1017/S0956796800000186
3. Abramsky, S., Jagadeesan, R., Malacaria, P.: Full abstraction for PCF. Inf. Comput. **163**(2), 409–470 (2000). https://doi.org/10.1006/inco.2000.2930, https://doi.org/10.1006/inco.2000.2930
4. Abramsky, S., Malacaria, P., Jagadeesan, R.: Full abstraction for PCF. In: Hagiya, M., Mitchell, J.C. (eds.) Theoretical Aspects of Computer Software, International Conference TACS '94, Sendai, Japan, April 19-22, 1994, Proceedings. Lecture Notes in Computer Science, vol. 789, pp. 1–15. Springer (1994). https://doi.org/10.1007/3-540-57887-0_87, https://doi.org/10.1007/3-540-57887-0_87
5. Berry, G., Curien, P.L., Lévy, J.J.: Full abstraction for sequential languages: state of the art. In: Nivat, M., Reynolds, J. (eds.) Algebraic methods in semantics, pp. 89–132. Cambridge University Press (1985)
6. Curien, P.L.: Observable algorithms on concrete data structures. In: Proceedings of the Seventh Annual Symposium on Logic in Computer Science (LICS '92), Santa Cruz, California, USA, June 22-25, 1992. pp. 432–443. IEEE Computer Society (1992). https://doi.org/10.1109/LICS.1992.185554, https://doi.org/10.1109/LICS.1992.185554
7. Curien, P.L., Hardin, T., Lévy, J.J.: Confluence properties of weak and strong calculi of explicit substitutions. J. ACM **43**(2), 362–397 (1996). https://doi.org/10.1145/226643.226675, https://doi.org/10.1145/226643.226675
8. Della Penna, G.: Una semantica operazionale per il network computing: le macchine di Turing virtuali. Master's thesis, Università degli Studi di L'Aquila (1996-97), in Italian
9. Della Penna, G., Intrigila, B., Manzonetto, G.: Addressing machines as models of λ-calculus. CoRR **abs/2107.00319** (2021), https://arxiv.org/abs/2107.00319
10. Fairbairn, J., Wray, S.: Tim: A simple, lazy abstract machine to execute supercombinators. In: Kahn, G. (ed.) Functional Programming Languages and Computer Architecture. pp. 34–45. Springer Berlin Heidelberg, Berlin, Heidelberg (1987)
11. Hyland, J.M.E., Ong, C.L.: On full abstraction for PCF: I, II, and III. Inf. Comput. **163**(2), 285–408 (2000). https://doi.org/10.1006/inco.2000.2917, https://doi.org/10.1006/inco.2000.2917
12. Lévy, J.J., Maranget, L.: Explicit substitutions and programming languages. In: Rangan, C.P., Raman, V., Ramanujam, R. (eds.) Foundations of Software Technology and Theoretical Computer Science, 19th Conference, Chennai, India, December 13-15, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1738, pp. 181–200. Springer (1999). https://doi.org/10.1007/3-540-46691-6_14, https://doi.org/10.1007/3-540-46691-6_14
13. Longley, J.: Realizability toposes and language semantics. Ph.D. thesis, University of Edinburgh (1995)

14. Milner, R.: Fully abstract models of typed *lambda*-calculi. Theor. Comput. Sci. **4**(1), 1–22 (1977). https://doi.org/10.1016/0304-3975(77)90053-6, https://doi.org/10.1016/0304-3975(77)90053-6
15. Ong, C.H.L.: Correspondence between operational and denotational semantics of PCF. In: Abramsky, S., Gabbay, D., Maibaum, T.S.E. (eds.) Handbook of Logic in Computer Science, vol. 4, pp. 269–356. Oxford University Press (1995)
16. Plotkin, G.D.: LCF considered as a programming language. Theor. Comput. Sci. **5**(3), 223–255 (1977). https://doi.org/10.1016/0304-3975(77)90044-5, https://doi.org/10.1016/0304-3975(77)90044-5
17. Seaman, J., Iyer, S.P.: An operational semantics of sharing in lazy evaluation. Sci. Comput. Program. **27**(3), 289–322 (1996). https://doi.org/10.1016/0167-6423(96)00012-3, https://doi.org/10.1016/0167-6423(96)00012-3

$$\frac{U \in \mathrm{Val}}{U \Downarrow U}\ (\mathrm{val}) \qquad \frac{P \Downarrow \underline{n+1}}{\mathbf{pred}\, P \Downarrow \underline{n}}\ (\mathrm{pr}) \qquad \frac{P \Downarrow \mathbf{0}}{\mathbf{pred}\, P \Downarrow \mathbf{0}}\ (\mathrm{pr}_0)$$

$$\frac{P \Downarrow \underline{0} \quad Q \Downarrow U_1}{\mathbf{ifz}(P,Q,Q') \Downarrow U_1}\ (\mathrm{ifz}_0) \qquad \frac{P \Downarrow \underline{n+1} \quad Q' \Downarrow U_2}{\mathbf{ifz}(P,Q,Q') \Downarrow U_2}\ (\mathrm{ifz}_{>0}) \qquad \frac{P \Downarrow \underline{n}}{\mathbf{succ}\, P \Downarrow \underline{n+1}}\ (\mathrm{sc})$$

$$\frac{P \cdot (\mathbf{fix}\, P) \Downarrow U}{\mathbf{fix}\, P \Downarrow U}\ (\mathrm{fix}) \qquad \frac{P \Downarrow \lambda x.P' \quad P'[Q/x] \Downarrow U}{P \cdot Q \Downarrow U}\ (\beta_v)$$

Fig. 5. The big-step operational semantics of PCF.

$$\frac{}{\Gamma, x:\alpha \vdash x:\alpha} \qquad \frac{\Gamma, x:\alpha \vdash P:\beta}{\Gamma \vdash \lambda x.P : \alpha \to \beta} \qquad \frac{\Gamma \vdash P : \alpha \to \beta \quad \Gamma \vdash Q : \alpha}{\Gamma \vdash PQ : \beta}$$

$$\frac{}{\Gamma \vdash \underline{0} : \mathsf{int}} \qquad \frac{\Gamma \vdash P : \mathsf{int}}{\Gamma \vdash \mathbf{pred}\ P : \mathsf{int}} \qquad \frac{\Gamma \vdash P : \mathsf{int}}{\Gamma \vdash \mathbf{succ}\ P : \mathsf{int}}$$

$$\frac{\Gamma \vdash P : \mathsf{int} \quad \Gamma \vdash Q : \alpha \quad \Gamma \vdash Q' : \alpha}{\Gamma \vdash \mathbf{ifz}(P,Q,Q') : \alpha} \qquad \frac{\Gamma \vdash P : \alpha \to \alpha}{\Gamma \vdash \mathbf{fix}\, P : \alpha}$$

Fig. 6. The type inference rules of PCF.

A Technical Appendix

This appendix is devoted to recall the language PCF, which is given for granted in the paper, as well as providing some missing proofs.

A.1 The standard PCF

We describe the syntax of PCF terms, their typing system based on simple types and provide the associated (call-by-name) big-step operational semantics.

Our presentation of PCF mainly follows [15].

Definition 12. *1.* PCF terms *are defined by the grammar (for $n \geq 0$, $x \in \mathrm{Var}$):*

$$\begin{aligned} P, Q, Q' ::= \ & x \mid P \cdot Q \mid \lambda x.P \\ & \mid \mathbf{0} \mid \mathbf{pred}\, P \mid \mathbf{succ}\, P \mid \mathbf{ifz}(P,Q,Q') \mid \mathbf{fix}\, P \end{aligned}$$

2. A closed PCF *term P is called a* PCF program.
3. A PCF *value U is a term of the form $\lambda x.P$ or $\underline{n}$, for some $n \geq 0$.*
4. Given a PCF *term P and a value U, we write $P \Downarrow U$ if this judgement can be obtained by applying the rules from Figure 5.*
5. The set $\mathbb{T}$ of simple types *and* typing contexts *have already been defined in items* (1) *and* (2) *of Definition 3, respectively.*
6. Given a PCF *term P, a typing context Γ and $\alpha \in \mathbb{T}$, we write $\Gamma \vdash^{\mathsf{PCF}} P : \alpha$ if this typing judgement is derivable from the rules of Figure 6.*

A.2 Proofs of Section 1

Proof of Lemma 1. Items (1) and (2) are straightforward. We prove (3).

Given an EPCF term M, an EPCF value V, an explicit substitution σ, a type α, and a context Γ such that $\sigma \triangleright M \Downarrow_d V, \sigma \models \Gamma, \Gamma \vdash M : \alpha$, we prove by induction on a derivation of $\sigma \triangleright M \Downarrow_d V$ that $\vdash V : \alpha$.

Case (nat)**:** In this case $M = V = \underline{n}$, for $\underline{n} \in \mathbb{N}$, and $\alpha = \mathsf{int}$. By the typing rules of EPCF, $\vdash \underline{n} : \mathsf{int}$.

Case (fun)**:** In this case $M = \lambda x.M'\langle\rho\rangle, V = \lambda x.M'\langle\sigma + \rho\rangle, \alpha = \beta \to \gamma$. As $\Gamma \vdash \lambda x.M'\langle\rho\rangle : \beta \to \gamma$, $\exists!\Delta$ such that $\rho \models \Delta, \Gamma, \Delta, x : \beta \vdash M' : \gamma$. As $\sigma \models \Gamma$ and $\rho \models \Delta$, $\sigma + \rho \models \Gamma, \Delta$. Thus by the typing rules of EPCF, $\vdash \lambda x.M'\langle\sigma + \rho\rangle : \beta \to \gamma$.

Case (var)**:** In this case $M = x, \sigma(x) = (\rho, N), \Gamma = \Gamma', x : \alpha$. By the operational semantics of EPCF, $\rho \triangleright N \Downarrow_d V$, and by the type system of EPCF as $\sigma \models \Gamma', x : \alpha$, $[x \leftarrow (\rho, N)] \models x : \alpha$ and thus $\rho \models \Delta, \Delta \vdash N : \alpha$. By IH, $\vdash V : \alpha$.

Case (β_v)**:** In this case $M = N \cdot L$. By the operational semantics of EPCF, $\sigma \triangleright N \Downarrow_d \lambda x.N'\langle\rho\rangle$ and $\rho + [x \leftarrow (\sigma, L)] \triangleright N' \Downarrow_d V$. By the type system of EPCF, $\Gamma \vdash N : \beta \to \alpha$, $\Gamma \vdash L : \beta$. By IH, $\vdash \lambda x.N'\langle\rho\rangle : \beta \to \alpha$, and then by the type system of EPCF $\rho \models \Delta, \Delta, x : \beta \vdash N' : \alpha$. By the type system we also have $[x \leftarrow (\sigma, L)] \models x : \beta$, so $\rho + [x \leftarrow (\sigma, L)] \models \Delta, x : \beta$, and thus by induction hypothesis we conclude $\vdash V : \alpha$.

All other cases derive straightforwardly from applying the rules of the type system and the induction hypothesis. □

A.3 Proofs of Section 4

Proof of Theorem 1. We prove the following statements by mutual induction and call the respective inductive hypotheses IH1 and IH2.

(i) Let M be an EPCF term, $\Gamma = x_1 : \beta_1, \ldots, x_n : \beta_n$ and $\alpha \in \mathbb{T}$. Then

$$\Gamma \vdash M : \alpha \quad \Rightarrow \quad \vdash [\![M]\!]^{x_1,\ldots,x_n} : \beta_1 \to \cdots \to \beta_n \to \alpha.$$

(ii) For all $\Gamma = x_1 : \beta_1, \ldots, x_n : \beta_n$ and σ, with $\mathrm{dom}(\sigma) = \{x_1, \ldots, x_n\}$, we have

$$\sigma \models \Gamma \quad \Rightarrow \quad \vdash (\!|\sigma(x_1)|\!) : \beta_1, \ldots, \vdash (\!|\sigma(x_n)|\!) : \beta_n.$$

We start with the cases concerning (i).

Case (ax)**:** Then, $x_1 : \beta_1, \ldots, x_n : \beta_n \vdash x_i : \beta_i$, and $[\![x_i]\!]^{x_1,\ldots,x_n} = \mathsf{Proj}_i^n$. By Lemma 6(1) we conclude $\vdash \mathsf{Proj}_i^n : \beta_1 \to \cdots \to \beta_n \to \beta_i$.

Case (0)**:** In this case, we have $\Gamma \vdash \mathbf{0} : \mathsf{int}$ and $[\![\mathbf{0}]\!]^{x_1,\ldots,x_n} = \mathsf{Proj}_1^{n+1} @ [0]$. By Lemma 6(1) $\vdash \mathsf{Proj}_1^{n+1} @ : \mathsf{int} \to \beta_1 \to \cdots \to \beta_n \to \mathsf{int}$ and, by Figure 4 $\vdash \#^{-1}(0) : \mathsf{int}$. By Proposition 1(1), $\vdash \mathsf{Proj}_1^{n+1} @ [0] : \beta_1 \to \cdots \to \beta_n \to \mathsf{int}$.

Case (Y)**:** In this case $\Gamma \vdash \mathbf{fix}\, M' : \alpha$ and $[\![\mathbf{fix}\, M']\!]^{\vec{x}} = \mathsf{Y}_n \mathbin{@} [\#[\![M']\!]^{\vec{x}}]$. By (fix$_n$) in Figure 4, we have $\vdash \mathsf{Y}_n : (\vec{\beta} \to \alpha \to \alpha) \to \vec{\beta} \to \alpha$. From the hypothesis IH1, we obtain $\vdash [\![M']\!]^{x_1,\ldots,x_n} : \vec{\beta} \to \alpha \to \alpha$. By Proposition 1(1), we conclude that $\vdash \mathsf{Y}_n \mathbin{@} [\#[\![M']\!]^{x_1,\ldots,x_n}] : \beta_1 \to \cdots \to \beta_n \to \alpha$.

Case (+)**:** In this case $\Gamma \vdash \mathbf{succ}\, M' : \mathsf{int}$ since $\Gamma \vdash M' : \mathsf{int}$. By definition, $[\![\mathbf{succ}\, M']\!]^{x_1,\ldots,x_n} = \mathsf{Succ}_n \mathbin{@} [\#[\![M']\!]^{x_1,\ldots,x_n}]$. By Lemma 6(4), we get $\vdash \mathsf{Succ}_n : (\vec{\beta} \to \mathsf{int}) \to \vec{\beta} \to \mathsf{int}$. By IH1, $\vdash [\![M']\!]^{x_1,\ldots,x_n} : \vec{\beta} \to \mathsf{int}$, and thus by Proposition 1(1), we conclude $\vdash \mathsf{Succ}_n \mathbin{@} [\#[\![M']\!]^{x_1,\ldots,x_n}] : \vec{\beta} \to \mathsf{int}$.

Case (−)**:** Analogous, applying Lemma 6(4).

Case (ifz)**:** Assume $\Gamma \vdash \mathbf{ifz}(L, N_1, N_2) : \alpha$ since $\Gamma \vdash L : \mathsf{int}$ and $\Gamma \vdash N_i : \alpha$, for $i \in \{1, 2\}$. By definition of the translation, we have $[\![\mathbf{ifz}(L, N_2, N_2)]\!]^{\vec{x}} = \mathsf{Ifz}_n \mathbin{@} [\#[\![L]\!]^{\vec{x}}, \#[\![N_1]\!]^{\vec{x}}, \#[\![N_2]\!]^{\vec{x}}]$. By applying Lemma 6(5), we obtain $\vdash \mathsf{Ifz}_n : (\vec{\beta} \to \mathsf{int}) \to (\vec{\beta} \to \alpha) \to (\vec{\beta} \to \alpha) \to \vec{\beta} \to \alpha$. By the hypothesis IH1, we get $\vdash [\![L]\!]^{\vec{x}} : \vec{\beta} \to \mathsf{int}$ and $\vdash [\![N_i]\!]^{\vec{x}} : \vec{\beta} \to \alpha$ for $i \in \{1, 2\}$, and thus by Proposition 1(1), we conclude $\vdash \mathsf{Ifz}_n \mathbin{@} [\#[\![L]\!]^{\vec{x}}, \#[\![N_1]\!]^{\vec{x}}, \#[\![N_2]\!]^{\vec{x}}] : \vec{\beta} \to \alpha$.

Case ($\to_{\mathrm{I}}$)**:** Assume that $\Gamma \vdash \lambda z.M'\langle\sigma\rangle : \alpha_1 \to \alpha_2$, for $\alpha = \alpha_1 \to \alpha_2$, because there is $\Delta = y_1 : \delta_1, \ldots, y_m : \delta_m$ such that $\sigma \models \Delta$ and $\Gamma, \Delta, z : \alpha_1 \vdash M' : \alpha_2$. Then $\sigma \models \Delta$ entails $\sigma = [y_1 \leftarrow (\rho_1, N_1), \ldots, y_m \leftarrow (\rho_m, N_m)]$ for appropriate $\vec{\rho}, \vec{N}$. By definition, we have

$$\begin{aligned}[\![\lambda z.M'\langle\sigma\rangle]\!]^{x_1,\ldots,x_n} &= (\!|\sigma, M'|\!)^{\vec{x},z} \\ &= [\![M']\!]^{y_1,\ldots,y_m,\vec{x},z} \mathbin{@} [\#(\!|\rho_1, N_1|\!), \ldots, \#(\!|\rho_m, N_m|\!)]\,.\end{aligned}$$

By applying IH1, we obtain $\vdash [\![M']\!]^{\vec{y},\vec{x},z} : \vec{\delta} \to \vec{\beta} \to \alpha_1 \to \alpha_2$. From IH2, we get $\vdash (\!|\sigma_1, N_1|\!) : \delta_1 \cdots \vdash (\!|\sigma_m, N_m|\!) : \delta_m$. Finally, by Proposition 1(1), we derive $\vdash [\![M']\!]^{\vec{y},\vec{x},z} \mathbin{@} [\#(\!|\sigma_1, N_1|\!), \ldots, \#(\!|\sigma_m, N_m|\!)] : \vec{\beta} \to \alpha_1 \to \alpha_2$.

Case ($\to_{\mathrm{E}}$)**:** In this case $\Gamma \vdash M_1 \cdot M_2 : \alpha$ since, for some $\delta \in \mathbb{T}$, $\Gamma \vdash M_1 : \delta \to \alpha$ and $\Gamma \vdash M_2 : \delta$. By definition, $[\![M_1 \cdot M_2]\!]^{\vec{x}} = \mathsf{Apply}_n \mathbin{@} [\#[\![M_1]\!]^{\vec{x}}, \#[\![M_2]\!]^{\vec{x}}]$. By Lemma 6(2), we get $\vdash \mathsf{Apply}_n : (\vec{\beta} \to \delta \to \alpha) \to (\vec{\beta} \to \delta) \to \vec{\beta} \to \alpha$. By IH1, we obtain $\vdash [\![M_1]\!]^{\vec{x}} : \vec{\beta} \to \delta \to \alpha$ and $\vdash [\![M_2]\!]^{\vec{x}} : \vec{\beta} \to \delta$. Conclude, by Proposition 1(1), that $\vdash \mathsf{Apply}_n \mathbin{@} [\#[\![M_1]\!]^{\vec{x}}, \#[\![M_2]\!]^{\vec{x}}] : \vec{\beta} \to \alpha$.

We now consider the cases concerning (ii).

Case (σ_0) : In this case $[\,] \models \emptyset$, so we have nothing to prove.

Case (σ)**:** In this case $\Gamma = \Gamma', x_n : \beta_n$ and $\sigma = \sigma' + [x_n \leftarrow (\rho, N)] \models \Gamma', x_n : \beta_n$, because $\sigma' \models \Gamma'$, $\rho \models \Delta$ and $\Delta \vdash N : \beta_n$, for some $\Delta = y_1 : \delta_1, \ldots, y_m : \delta_m$. By IH2 on $\sigma' \models \Gamma'$, we get $(\!|\sigma(x_i)|\!) = (\!|\sigma'(x_i)|\!) : \beta_i$, for all $i \in \{1, \ldots, n-1\}$. We show $(\!|(\rho, N)|\!) : \beta_n$. By IH1, we get $\vdash [\![N]\!]^{y_1,\ldots,y_m} : \delta_1 \to \cdots \to \delta_m \to \beta_n$. By applying IH2 on $\rho \models \Delta$, we get $(\!|\rho(y_j)|\!) : \delta_j$, for all $j \in \{1, \ldots, m\}$. Since $(\!|\rho, N|\!) = [\![N]\!]^{\vec{y}} \mathbin{@} [\#(\!|\rho(y_1)|\!), \ldots, \#(\!|\rho(y_m)|\!)]$, we conclude $\vdash (\!|\rho, N|\!) : \beta_n$, by Proposition 1(1). □

Proof of Theorem 2. We prove

$$\sigma \triangleright M \Downarrow_d V \qquad \Rightarrow \qquad (\!|\sigma, M|\!) \leftrightarrow_{\mathsf{c}} [\![V]\!]$$

by induction on a derivation of $\sigma \triangleright M \Downarrow_d V$. We let $\mathrm{dom}(\sigma) = \{x_1, \ldots, x_n\}$ and sometimes use the convenient notation $\#(\!|\sigma(\vec{x})|\!) = \#(\!|\sigma(x_1)|\!), \ldots, \#(\!|\sigma(x_n)|\!)$.

Case (nat)**:** Then $M = V = \underline{k}$, for some $k \geq 0$. Recall that we assume $k = \#\mathsf{k}$. On the one side, we have:

$$\begin{aligned}(\!|\sigma, \underline{k}|\!) &= [\![\underline{k}]\!]^{x_1,\ldots,x_n} @ [\#(\!|\sigma(x_1)|\!), \ldots, \#(\!|\sigma(x_n)|\!)] \\ &= \mathsf{Proj}_1^{n+1} @ [k, \#(\!|\sigma(x_1)|\!), \ldots, \#(\!|\sigma(x_n)|\!)]\,, \text{ by Lemma 5(1),} \\ &\twoheadrightarrow_{\mathsf{c}} \#^{-1}(k)\end{aligned}$$

On the other side, we obtain $[\![\underline{k}]\!] = \mathsf{Proj}_1^1 @ [k] \twoheadrightarrow_{\mathsf{c}} \#^{-1}(k)$.

Case 2: (fun) We have $M = \lambda z.M'\langle\rho\rangle$ and $V = \lambda x.M'\langle\sigma + \rho\rangle$, with $\mathrm{dom}(\sigma) \cap \mathrm{dom}(\rho) = \emptyset$. Say, $\mathrm{dom}(\rho) = \{y_1, \ldots, y_m\}$. Then

$$\begin{aligned}(\!|\sigma, \lambda z.M'\langle\rho\rangle|\!) &= [\![\lambda z.M'\langle\rho\rangle]\!]^{x_1,\ldots,x_n} @ [\#(\!|\sigma(x_1)|\!), \ldots, \#(\!|\sigma(x_n)|\!)] \\ &= (\!|\rho, M'|\!)^{\vec{x},z} @ [\#(\!|\sigma(x_1)|\!), \ldots, \#(\!|\sigma(x_n)|\!)] \\ &= [\![M']\!]^{\vec{y},\vec{x},z} @ [\#(\!|\rho(\vec{y})|\!), \#(\!|\sigma(\vec{x})|\!)] \\ &= (\!|\sigma + \rho, M'|\!)^{z} \\ &= [\![\lambda z.M'\langle\sigma + \rho\rangle]\!]\end{aligned}$$

The case follows by reflexivity of $\leftrightarrow_{\mathsf{c}}$.

Case (var)**:** We have $M = x_i$, $\sigma(x_i) = (\rho, N)$ and $\rho \triangleright N \Downarrow_d V$. Then, we have

$$\begin{aligned}(\!|\sigma, x_i|\!) &= [\![x_i]\!]^{x_1,\ldots,x_n} @ [\#(\!|\sigma(x_1)|\!), \ldots, \#(\!|\sigma(x_n)|\!)] \\ &= \mathsf{Proj}_i^n @ [\#(\!|\sigma(x_1)|\!), \ldots, \#(\!|\sigma(x_n)|\!)]\,, \qquad \text{by Lemma 5(1),} \\ &\twoheadrightarrow_{\mathsf{c}} (\!|\sigma(x_i)|\!) = (\!|\rho, N|\!)\end{aligned}$$

We conclude since, by induction hypothesis, we have $(\!|\rho, N|\!) \leftrightarrow_{\mathsf{c}} [\![V]\!]$.

Case (β_v)**:** $M = M_1 \cdot M_2$, $\sigma \triangleright M_1 \Downarrow_d \lambda z.M_1'\langle\rho\rangle$ and $\rho + [z \leftarrow (\sigma, M_2)] \triangleright M_1' \Downarrow_d V$. Easy calculations give:

$$\begin{aligned}(\!|\sigma, M_1 \cdot M_2|\!) &= [\![M_1 \cdot M_2]\!]^{x_1,\ldots,x_n} @ [\#(\!|\sigma(x_1)|\!), \ldots, \#(\!|\sigma(x_n)|\!)] \\ &= \mathsf{Apply}_n @ [[\![M_1]\!]^{\vec{x}}, \#[\![M_2]\!]^{\vec{x}}, \#(\!|\sigma(x_1)|\!), \ldots, \#(\!|\sigma(x_n)|\!)] \\ &\twoheadrightarrow_{\mathsf{c}} [\![M_1]\!]^{\vec{x}} @ [\#(\!|\sigma(\vec{x})|\!), \#([\![M_2]\!]^{\vec{x}} @ [\#(\!|\sigma(\vec{x})|\!)])]\,, \\ &\text{by Lemma 5(2),} \\ &= (\!|\sigma, M_1|\!) @ [\#(\!|\sigma, M_2|\!)]\end{aligned}$$

By induction hypothesis on $\sigma \triangleright M_1 \Downarrow_d \lambda z.M_1'\langle\rho\rangle$, we have $(\!|\sigma, M_1|\!) \leftrightarrow_{\mathsf{c}} [\![\lambda z.M_1'\langle\rho\rangle]\!] = (\!|\rho, M_1'|\!)^z$. Calling $\mathrm{dom}(\rho) = \{y_1, \ldots, y_m\}$, we get

$$\begin{aligned}(\!|\sigma, M_1|\!) @ [\#(\!|\sigma, M_2|\!)] &\leftrightarrow_{\mathsf{c}} (\!|\rho, M_1'|\!)^z @ [\#(\!|\sigma, M_2|\!)]\,, \\ &\text{by def. of } \leftrightarrow_{\mathsf{c}} \text{ and Lemma 3,} \\ &= [\![M_1']\!]^{y_1,\ldots,y_m,z} @ [\#(\!|\rho(\vec{y})|\!), \#(\!|\sigma, M_2|\!)] \\ &= (\!|\rho + [z \leftarrow (\sigma, M_2)], M_1'|\!)\end{aligned}$$

By induction hypothesis conclude $(\!|\rho + [z \leftarrow (\sigma, M_2)], M_1'|\!) \leftrightarrow_{\mathsf{c}} [\![V]\!]$.

Case (pr)**:** $M = \mathbf{pred}\, M'$, $V = \underline{n}$ and $\sigma \triangleright M' \Downarrow_d \underline{n+1}$, for some $n \geq 0$. By induction hypothesis we get $(\!|\sigma, M'|\!) \leftrightarrow_{\mathsf{c}} [\![\underline{n+1}]\!] = \#^{-1}(n+1)$ and, since the $(n+1)$-th numeral machine is in final state, we derive $(\!|\sigma, M'|\!) \twoheadrightarrow_{\mathsf{c}} \#^{-1}(n+1)$. Conclude as follows:

$$\begin{aligned}
(\!|\sigma, \mathbf{pred}\, M'|\!) &= [\![\mathbf{pred}\, M']\!]^{x_1,\ldots,x_n} @ [\#(\!|\sigma(x_1)|\!), \ldots, \#(\!|\sigma(x_n)|\!)] \\
&= \mathsf{Pred}_n @ [\#[\![M']\!]^{x_1,\ldots,x_n}, \#(\!|\sigma(x_1)|\!), \ldots, \#(\!|\sigma(x_n)|\!)] \\
&\twoheadrightarrow_{\mathsf{c}} \left\langle R_0 = \#([\![M']\!]^{\vec{x}} @ [\#(\!|\sigma(\vec{x})|\!)]\,), \vec{R}, 0 \leftarrow \texttt{Pred}(0); \texttt{Call}\ 0, [] \right\rangle, \\
&\quad \text{by Lemma 5(3)}, \\
&= \left\langle R_0 = \#(\!|\sigma, M'|\!), \vec{R}, ; 0 \leftarrow \texttt{Pred}(0); \texttt{Call}\ 0, [] \right\rangle \\
&\twoheadrightarrow_{\mathsf{c}} \left\langle R_0 = n+1, \vec{R}, 0 \leftarrow \texttt{Pred}(0); \texttt{Call}\ 0, [] \right\rangle \\
&\twoheadrightarrow_{\mathsf{c}} \#^{-1}(n)
\end{aligned}$$

Case (pr_0)**:** Analogous to the previous case, using the fact that $\texttt{Pred}(0)$ is 0.

Case 7: (sc) $M = \mathbf{succ}\, M'$, $V = \underline{n+1}$ and $\sigma \triangleright M' \Downarrow_d \underline{n}$, for some $n \geq 0$. By induction hypothesis we get $(\!|\sigma, M'|\!) \leftrightarrow_{\mathsf{c}} [\![\underline{n}]\!] = \#^{-1}(n)$ and, since the n-th numeral machine is in final state, we derive $(\!|\sigma, M'|\!) \twoheadrightarrow_{\mathsf{c}} \#^{-1}(n)$. Conclude as follows:

$$\begin{aligned}
(\!|\sigma, \mathbf{succ}\, M'|\!) &= [\![\mathbf{succ}\, M']\!]^{x_1,\ldots,x_n} @ [\#(\!|\sigma(x_1)|\!), \ldots, \#(\!|\sigma(x_n)|\!)] \\
&= \mathsf{Succ}_n @ [\#[\![M']\!]^{x_1,\ldots,x_n}, \#(\!|\sigma(x_1)|\!), \ldots, \#(\!|\sigma(x_n)|\!)] \\
&\twoheadrightarrow_{\mathsf{c}} \left\langle R_0 = \#([\![M']\!]^{\vec{x}} @ [\#(\!|\sigma(\vec{x})|\!)]\,), \vec{R}, 0 \leftarrow \texttt{Succ}(0); \texttt{Call}\ 0, [] \right\rangle, \\
&\quad \text{by Lemma 5(3)}, \\
&= \left\langle R_0 = \#(\!|\sigma, M'|\!), \vec{R}, ; 0 \leftarrow \texttt{Succ}(0); \texttt{Call}\ 0, [] \right\rangle \\
&\twoheadrightarrow_{\mathsf{c}} \left\langle R_0 = n, \vec{R}, 0 \leftarrow \texttt{Succ}(0); \texttt{Call}\ 0, [] \right\rangle \\
&\twoheadrightarrow_{\mathsf{c}} \#^{-1}(n+1)
\end{aligned}$$

Case 8: ($\mathrm{ifz}_{>0}$) $M = \mathbf{ifz}(L, N_1, N_2)$, $\sigma \triangleright L \Downarrow_d \underline{n+1}$, $\sigma \triangleright N_2 \Downarrow_d V$ for some $n \geq 0$. By IH we get $(\!|\sigma, L|\!) \leftrightarrow_{\mathsf{c}} [\![\underline{n+1}]\!]$, and since the $(n+1)$-th numeral machine is in final state, we derive $(\!|\sigma, L|\!) \twoheadrightarrow_{\mathsf{c}} \#^{-1}(n+1)$. Then

$$\begin{aligned}
(\!|\sigma, \mathbf{ifz}(L, N_1, N_2)|\!) &= [\![\mathbf{ifz}(L, N_1, N_2)]\!]^{x_1,\ldots,x_n} @ [\#(\!|\sigma(x_1)|\!), \ldots, \#(\!|\sigma(x_n)|\!)] \\
&= \mathsf{Ifz}_n @ \begin{bmatrix} \#[\![L]\!]^{x_1,\ldots,x_n}, \#[\![N_1]\!]^{x_1,\ldots,x_n}, \#[\![N_2]\!]^{x_1,\ldots,x_n}, \\ \#(\!|\sigma(x_1)|\!), \ldots, \#(\!|\sigma(x_n)|\!) \end{bmatrix} \\
&\twoheadrightarrow_{\mathsf{c}} \left\langle \begin{array}{r} R_0 = \#(\!|\sigma, L|\!), R_1 = \#(\!|\sigma, N_1|\!), R_2 = \#(\!|\sigma, N_2|\!), \\ \vec{R}, 0 \leftarrow \texttt{Test}(0,\ 1,\ 2); \texttt{Call}\ 0, [] \end{array} \right\rangle, \\
&\quad \text{by Lemma 5(5)}, \\
&\twoheadrightarrow_{\mathsf{c}} \left\langle \begin{array}{r} R_0 = n+1, R_1 = \#(\!|\sigma, N_1|\!), R_2 = \#(\!|\sigma, N_2|\!), \\ \vec{R}, 0 \leftarrow \texttt{Test}(0,\ 1,\ 2); \texttt{Call}\ 0, [] \end{array} \right\rangle \\
&\twoheadrightarrow_{\mathsf{c}} (\!|\sigma, N_2|\!)
\end{aligned}$$

We conclude since, by IH, $(\!|\sigma, N_2|\!) \leftrightarrow_{\mathsf{c}} [\![V]\!]$.

Case 9: (ifz$_0$) $M = \mathbf{ifz}(L, N_1, N_2)$, $\sigma \triangleright L \Downarrow_d \underline{0}$, $\sigma \triangleright N_1 \Downarrow_d V$. By IH we get $(\!|\sigma, L|\!) \leftrightarrow_{\mathsf{c}} [\![\underline{0}]\!]$, and since the $(n+1)$-th numeral machine is in final state, we derive $(\!|\sigma, L|\!) \twoheadrightarrow_{\mathsf{c}} \#^{-1}(0)$. Then

$$
\begin{aligned}
(\!|\sigma, \mathbf{ifz}(L, N_1, N_2)|\!) &= [\![\mathbf{ifz}(L, N_1, N_2)]\!]^{x_1,\ldots,x_n} @ [\#(\!|\sigma(x_1)|\!), \ldots, \#(\!|\sigma(x_n)|\!)] \\
&= \mathsf{Ifz}_n @ \begin{bmatrix} \#[\![L]\!]^{x_1,\ldots,x_n}, \#[\![N_1]\!]^{x_1,\ldots,x_n}, \#[\![N_2]\!]^{x_1,\ldots,x_n}, \\ \#(\!|\sigma(x_1)|\!), \ldots, \#(\!|\sigma(x_n)|\!) \end{bmatrix} \\
&\twoheadrightarrow_{\mathsf{c}} \left\langle \begin{array}{r} R_0 = \#(\!|\sigma, L|\!), R_1 = \#(\!|\sigma, N_1|\!), R_2 = \#(\!|\sigma, N_2|\!), \\ \vec{R}, 0 \leftarrow \texttt{Test}(0,\ 1,\ 2); \texttt{Call}\ 0, [] \end{array} \right\rangle, \\
&\quad \text{by Lemma 5(5)}, \\
&\twoheadrightarrow_{\mathsf{c}} \left\langle \begin{array}{r} R_0 = 0, R_1 = \#(\!|\sigma, N_1|\!), R_2 = \#(\!|\sigma, N_2|\!), \\ \vec{R}, 0 \leftarrow \texttt{Test}(0,\ 1,\ 2); \texttt{Call}\ 0, [] \end{array} \right\rangle \\
&\twoheadrightarrow_{\mathsf{c}} (\!|\sigma, N_1|\!)
\end{aligned}
$$

We conclude since, by IH, $(\!|\sigma, N_1|\!) \leftrightarrow_{\mathsf{c}} [\![V]\!]$.

Case (fix)**:** Then $M = \mathbf{fix}\, M'$ and $\sigma \triangleright M' \cdot (\mathbf{fix}\, M')$. On the one side we have:

$$
\begin{aligned}
(\!|\sigma, \mathbf{fix}\, M'|\!) &= [\![\mathbf{fix}\, M']\!]^{x_1,\ldots,x_n} @ [\#(\!|\sigma(x_1)|\!), \ldots, \#(\!|\sigma(x_n)|\!)] \\
&= \mathsf{Y}_n @ [\#[\![M']\!]^{x_1,\ldots,x_n}, \#(\!|\sigma(x_1)|\!), \ldots, \#(\!|\sigma(x_n)|\!)] \\
&\twoheadrightarrow_{\mathsf{c}} [\![M']\!]^{\vec{x}} @ [\#(\!|\sigma(\vec{x})|\!), \#(\mathsf{Y}_n @ [\#[\![M']\!]^{\vec{x}}, \#(\!|\sigma(\vec{x})|\!)]\,)]
\end{aligned}
$$

On the other side:

$$
\begin{aligned}
&(\!|\sigma, M' \cdot (\mathbf{fix}\, M')|\!) \\
&= [\![M' \cdot (\mathbf{fix}\, M')]\!]^{x_1,\ldots,x_n} @ [\#(\!|\sigma(x_1)|\!), \ldots, \#(\!|\sigma(x_n)|\!)] \\
&= \mathsf{Apply}_n @ [\#[\![M']\!]^{\vec{x}}, \#[\![\mathbf{fix}\, M']\!]^{\vec{x}}, \#(\!|\sigma(x_1)|\!), \ldots, \#(\!|\sigma(x_n)|\!)] \\
&= [\![M']\!]^{\vec{x}} @ [\#(\!|\sigma(\vec{x})|\!), \#([\![\mathbf{fix}\, M']\!]^{\vec{x}} @ [\#(\!|\sigma(\vec{x})|\!)]\,)]\,, \text{ by Lemma 5(2)}, \\
&= [\![M']\!]^{\vec{x}} @ [\#(\!|\sigma(\vec{x})|\!), \#(\mathsf{Y}_n @ [\#[\![M']\!]^{\vec{x}}, \#(\!|\sigma(\vec{x})|\!)]\,)]
\end{aligned}
$$

This concludes the proof. □

Lemma 8. *1. Let M, N be* EPCF *terms, σ, ρ be explicit substitutions and x be a variable.*

$$(\sigma + [x \leftarrow (\rho, N)], M)^* = (\sigma, M)^*[(\rho, N)^*/x]$$

2. Let P, Q be PCF *terms with* $\mathrm{FV}(P) \subseteq \{x\}$ *and Q closed. For all* EPCF *terms M, N and explicit substitutions σ, ρ, we have:*

$$(\sigma, M) \in P^\dagger \wedge (\rho, N) \in Q^\dagger \Rightarrow (\sigma + [x \leftarrow (\rho, N)], M) \in (P[Q/x])^\dagger$$

We rely on the freshness hypothesis on the variables in $\mathrm{dom}(\sigma)$.

Proof. 1. By structural induction on M.

Case $M = y$**, with** $y \neq x$**:** There are two subcases. If $y \in \mathrm{dom}(\sigma)$, then

$$\begin{aligned}(\sigma + [x \leftarrow (\rho, N)], y)^* &= \sigma(y) \\ &= \sigma(y)[(\rho, N)^*/x], \text{ since } x \notin \mathrm{FV}(\sigma(y)), \\ &= (\sigma, y)^*[(\rho, N)^*/x]\end{aligned}$$

If $y \notin \mathrm{dom}(\sigma)$, then $\begin{aligned}(\sigma + [x \leftarrow (\rho, N)], y)^* &= y \\ &= y[(\rho, N)^*/x] \\ &= (\sigma, y)^*[(\rho, N)^*/x]\end{aligned}$

Case $M = x$**:** Then $\begin{aligned}(\sigma + [x \leftarrow (\rho, N)], x)^* &= (\rho, N)^* \\ &= x[(\rho, N)^*/x] \\ &= (\sigma, x)^*[(\rho, N)^*/x]\end{aligned}$

Case $M = \lambda y.M'$**:** Wlog, we may assume $y \neq x$. We have

$$\begin{aligned}(\sigma + [x \leftarrow (\rho, N)], \lambda y.M')^* &= \lambda y.(\sigma + [x \leftarrow (\rho, N)], M')^* \\ &= \lambda y.((\sigma, M')^*[(\rho, N)^*/x]) \\ &= (\lambda y.(\sigma, M')^*)[(\rho, N)^*/x] \\ &= ((\sigma, \lambda y.M')^*)[(\rho, N)^*/x]\end{aligned}$$

All other cases derive straightforwardly from the induction hypothesis.

2. By an easy induction on P, using (1). □

Proof of Theorem 3(1). For an EPCF term M, an EPCF value V and an explicit substitution σ, we show

$$\sigma \triangleright M \Downarrow_d V \Rightarrow (\sigma, M)^* \Downarrow ([\,], V)^*$$

by induction on a derivation of $\sigma \triangleright M \Downarrow_d V$.

Case (nat)**:** In this case $M = V = \underline{n}$ for some $n \geq 0$. By definition, $(\sigma, \underline{n})^* = \underline{n}$, so we apply PCF's rule (val) and get $\underline{n} \Downarrow \underline{n}$.

Case (fun)**:** We have $M = \lambda x.M'\langle\sigma'\rangle$ and $V = \lambda x.M'\langle\sigma + \sigma'\rangle$.
As $(\sigma, \lambda x.M'\langle\sigma'\rangle)^* = ([\,], \lambda x.M'\langle\sigma + \sigma'\rangle)^* = \lambda x.(\sigma + \sigma', M')^*$ and the latter is a PCF value, we can apply PCF's rule (val) to conclude $(\sigma, \lambda x.M'\langle\sigma'\rangle)^* \Downarrow ([\,], \lambda x.M'\langle\sigma + \sigma'\rangle)^*$.

Case (var)**:** In this case $M = x$, $\sigma(x) = (\rho, N)$ and $\rho \triangleright N \Downarrow_d V$. By induction hypothesis $(\rho, N)^* \Downarrow ([], V)^*$. We conclude since $(\sigma, M)^* = (\rho, N)^*$.

Case (β_v)**:** In this case $M = M_1 \cdot M_2$ with $\sigma \triangleright M_1 \Downarrow_d \lambda x.M_1'\langle\rho\rangle$, wlog $x \notin \mathrm{dom}(\rho + \sigma)$, and $\rho + [x \leftarrow (\sigma, M_2)] \triangleright M_1' \Downarrow_d V$. By induction hypothesis we obtain $(\sigma, M_1)^* \Downarrow ([\,], \lambda x.M_1'\langle\rho\rangle)^*$ and $(\rho + [x \leftarrow (\sigma, M_2)], M_1')^* \Downarrow ([\,], V)^*$. By Lemma 8(1) $(\rho + [x \leftarrow (\sigma, M_2)], M_1')^* = (\rho, M_1')^*[(\sigma, M_2)^*/x]$. By definition, $([\,], \lambda x.M_1'\langle\rho\rangle)^* = \lambda x.(\rho, M_1')^*$ and $(\sigma, M_1)^* \cdot (\sigma, M_2)^* = (\sigma, M_1 \cdot M_2)^*$. Thus

$$\frac{(\sigma, M_1)^* \Downarrow \lambda x.(\sigma', M_1')^* \quad (\sigma', M_1')^*[(\sigma, M_2)^*/x] \Downarrow ([\,], V)^*}{(\sigma, M_1 \cdot M_2)^* \Downarrow ([\,], V)^*}\ (\beta_v)$$

Case (fix)**:** In this case $M = \mathbf{fix}\, N$ and $\sigma \triangleright N \cdot (\mathbf{fix}\, N) \Downarrow_d V$. From the induction hypothesis we get $(\sigma, N \cdot (\mathbf{fix}\, N))^* \Downarrow ([\,], V)^*$. By definition, we have

$$(\sigma, N \cdot (\mathbf{fix}\, N))^* = (\sigma, N)^* \cdot (\mathbf{fix}\, (\sigma, N)^*)$$

and $\mathbf{fix}\, (\sigma, N)^* = (\sigma, \mathbf{fix}\, N)^*$. By applying PCF's rule (fix), we obtain

$$\frac{(\sigma, N \cdot (\mathbf{fix}\, N))^* \Downarrow ([\,], V)^*}{(\sigma, \mathbf{fix}\, N)^* \Downarrow ([\,], V)^*}\ (\text{fix})$$

All other cases derive straightforwardly from the induction hypothesis. □

Proof of Theorem 3(2). For a PCF program P and PCF value U, we show:

$$P \Downarrow U \Rightarrow \forall (\sigma, M) \in P^\dagger,\ \exists V \in \mathrm{Val}\,.\,(\ \sigma \triangleright M \Downarrow_d V \text{ and } ([\,], V) \in U^\dagger\ )$$

By induction on the lexicographically ordered pairs, whose first component is the length of a derivation of $P \Downarrow U$ and second component is $|(\sigma, M)|$.

First, consider the case $M = y$ and $\sigma(y) = (\rho, N)$, with $(\rho, N) \in P^\dagger$. In this case, the length of the derivation $(\rho, N)^* \Downarrow U$ remained unchanged, while $|(\rho, N)| < |(\sigma, M)|$. Thus, we may use the induction hypothesis and conclude by applying (var). Therefore, in the following we assume that M is not a variable.

Case (val)**:** In this case $P = U$. Given $(\sigma, M) \in P^\dagger$, we distinguish several cases:

- Case $P = U = \lambda x.P_0$. Then, $M = \lambda x.M_0\langle\rho\rangle$ with $P_0 = (\sigma + \rho, M_0)^*$. Setting $V = \lambda x.M_0\langle\sigma + \rho\rangle$ we obtain $M \Downarrow_d V$ by (fun) with

$$([\,], V)^* = \lambda x.(\sigma + \rho, M_0) = \lambda x.P_0 = U.$$

- $P = U = \mathbf{0}$. It follows $M = V = \mathbf{0}$ and $\sigma \triangleright \mathbf{0} \Downarrow_d \mathbf{0}$ by (nat).
- $P = U = \mathbf{succ}\,(\underline{n})$, for some $n \in \mathbb{N}$, and M is not a variable. There are two possibilities:
  - $M = \mathbf{succ}\,(\underline{n})$ in which case we are done, since $\sigma \triangleright \underline{n+1} \Downarrow_d \underline{n+1}$.
  - $M = \mathbf{succ}\,(\underline{y})$ with $\sigma(y) = (\rho, N)$ and $(\rho, N) \in \underline{n}^\dagger$. Again, the length of the derivation $(\rho, N)^* \Downarrow \underline{n}$ is unchanged, while $|(\rho, N)| < |(\sigma, M)|$. Once applied the induction hypothesis, we conclude by (var) + (sc).

Case (β_v)**:** $P = P_1 \cdot P_2$ with $P_1 \Downarrow \lambda x.Q_1$ and $Q_1[P_2/x] \Downarrow U$ for some Q_1. Notice that, since P is closed so are P_1, P_2 and hence $\mathrm{FV}(Q_1) \subseteq \{x\}$. Now, $(\sigma, M)^* = P$ entails $M = M_1 \cdot M_2$ with $(\sigma, M_1) \in P_1^\dagger$ and $(\sigma, M_2) \in P_2^\dagger$. By ind. hyp., there is $V' \in \mathrm{Val}$ such that $\sigma \triangleright M_1 \Downarrow_d V_1$ with $([\,], V_1) \in (\lambda x.Q_1)^\dagger$. This implies $V_1 = \lambda x.N_1\langle\rho\rangle$ for some $(\rho, N_1) \in Q_1^\dagger$. By Lemma 8(2), we get $(\rho + [x \leftarrow (\sigma, M_2)], N_1) \in (Q_1[P_2/x])^\dagger$. By ind. hyp., there is $V \in \mathrm{Val}$ such that $\rho + [x \leftarrow (\sigma, M_2)] \triangleright N_1 \Downarrow_d V$ and $V \in U^\dagger$. Conclude by EPCF's (β_v).

Case (fix)**:** $P = \mathbf{fix}\, Q$ and $Q \cdot (\mathbf{fix}\, Q) \Downarrow V$. Then $(\sigma, M) \in P^\dagger$ entails $M = \mathbf{fix}\, N$ with $(\sigma, N) \in Q^\dagger$. It follows that $(\sigma, N \cdot (\mathbf{fix}\, N)) \in (Q \cdot (\mathbf{fix}\, Q))^\dagger$, therefore by induction hypothesis we get $\sigma \triangleright N \cdot (\mathbf{fix}\, N) \Downarrow_d V$ for some $V \in U^\dagger$. We conclude by applying EPCF's rule (fix).

All other cases derive straightforwardly from the induction hypothesis. □