Electronic Notes in
Theoretical Informatics
And Computer Science

ENTICS
https://entics.episciences.org

Volume 1
Proceedings of
MFPS 2022

Extended Addressing Machines for PCF, with Explicit Substitutions

Benedetto Intrigila[1]

Dipartimento di Ingegneria dell'Impresa,
University of Rome "Tor Vergata", Italy

Giulio Manzonetto[2,4] Nicolas Münnich[3,5]

Univ. USPN, Sorbonne Paris Cité,
LIPN, UMR 7030, CNRS, F-93430 Villetaneuse, France.

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, computational model.

Introduction

Turing machines (TM) and λ-calculus constitute two fundamental formalisms in theoretical computer science. Because of the difficulty in emulating higher-order calculations on a TM, their equivalence on partial numeric functions is not obtained directly, but rather composing different encodings. As a consequence, no model of λ-calculus (λ-*model*) based on TM's has arisen in the literature so far. Recently, Della Penna et al. have successfully built a λ-model based on so-called *addressing machines* (AM) [11]. The intent is to propose a model of computation, alternative to von Neumann architecture, where computation is based on communication between machines rather than performing local operations. In fact, 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 input-tape, store in a register the result of applying an address to another and, finally, pass the execution

[1] Email: benedetto.intrigila@uniroma2.it
[2] Partly supported by ANR Project PPS, ANR-19-CE48-0014.
[3] Partly supported by ANR JCJC Project CoGITARe, ANR-18-CE25-0001.
[4] Email: giulio.manzonetto@lipn.univ-paris13.fr
[5] Email: munnich@lipn.univ-paris13.fr

Published February 15, 2023
10.46298/entics.10533
Proceedings Available Online at
https://doi.org/10.46298/entics.proceedings.mfps38

to another machine by calling its address (possibly extending its input-tape). 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 is as awkward as using Church numerals in λ-calculus.

Contents. In this paper we extend the formalism of AM's with a set of 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, in order to model recursion, we rely on the existence of machines representing *fixed point combinators*. These machines can be programmed in the original formalism but we can avoid any dependency on *self-application* by manipulating the addressing mechanism so that they have access to *their own address*. This can be seen as a very basic version of the reflection principle which is present in some programming languages. We call the resulting formalism *extended addressing machines* (EAMs).

Considering these features, one might expect EAMs to be well-suited for simulating Plotkin's *Programming Computable Functions* (PCF) [23], 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 in the contractum $M[N/x]$ of the redex the substitution 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 [19]. 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 the equivalence between PCF and EPCF on terminating programs of type int, we are capable of drawing conclusions for the original language PCF.

In this paper we mainly focus on the properties of the translation, but our long-term goal is to construct a sequential model of higher-order computations. The problem of finding a fully abstract model of PCF was originally proposed by Robin Milner in [21] and is a difficult one. A model is called *fully abstract* (FA) whenever two programs sharing the type α get the same denotation in the model if and only if they are observationally indistinguishable when plugged in the same context $C[]$ of type $\alpha \to \mathsf{int}$. Therefore, a FA model provides a semantic characterization of the observational equivalence of PCF. 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 FA models of PCF can be obtained by defining suitable categories of games [4,3,13]. Preliminary investigations show that EAMs open the way to construct a more 'computational' FA model. E.g., in [21], the model construction starts with first-order definable functions and requires—to cope with fixed point operator—the addition of extra 'limit points' to ensure that the resulting partial order is direct complete. In the game semantics approach the fixed point operator is treated similarly, namely via its canonical interpretation in a cpo-enriched Cartesian closed category [4]. On the contrary, in our approach no limit construction is required to give the fixed point operator a meaning. The fact that EAMs possess a given recursor having its own address stored inside is easily obtained from a mathematical point of view and, as argued above, can be seen as an abstract view of the usual implementation of recursion. We believe this new point of view may increase our understanding of PCF observational equivalence.

Outline. 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. In Section 2 we define EAMs (no familiarity with [11] 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 4.6);

(ii) if an EPCF term reduces to a value, then their translations as machines are interconvertible (Thm. 4.7);

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

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

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

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

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

(iii) the operational semantics of PCF and EPCF coincide on terminating programs of type int (Thm. 4.13);

(iv) the translation of a PCF program computing a number is an EAM evaluating the corresponding numeral (Theorem 4.14).

Related works. A preliminary version of AMs was introduced in Della Penna's MSc thesis [10] in order to model computation as communication between distinguished processes by means of their addresses. They were subsequently refined in [11] with the theoretical purpose of constructing a model of λ-calculus. Similarly, our paper should be seen as a first step towards the construction of a denotational model of PCF. Thus, the natural comparison is [6] with other models rather than other machine-based formalisms that have been proposed in the literature (e.g., call-by-name: SECD [16], KAM [15], call-by-need: TIM [12], Lazy KAM [6,17]); call-by-value: ZINC [18]) from which they differ at an implementational level.

Compared with models of PCF based on Scott-continuous functions [21,5,7], EAMs provide a more operational interpretation of a program and naturally avoid parallel features that would lead to the failure of FA as in the continuous semantics. Compared with Curien's sequential algorithms [8] and categories of games [4,13] they share the intensionality of programs' denotations, while presenting an original way of modelling sequential computation. The model based on AMs also bares *some* similarities with the categories of assembly used to model PCF [20], mostly on a philosophical level, in the sense that these models are based on the 'codes' (rather than addresses) of recursive functions realizing a formula ($\cong$ type).

Concerning explicit substitutions we refer to the pioneering articles [1,2,9,19]. Explicit substitutions have been barely considered in the context of PCF—with the notable exception of [24].

1 Preliminaries

The paradigmatic programming language PCF [23] is a simply typed λ-calculus enriched with constants representing natural numbers, 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 [19]. We draw conclusions for the standard PCF by exploiting the fact that they are equivalent on programs (closed terms) of type int.

Definition 1.1 Consider fixed a countably infinite set Var of variables. EPCF *terms* and *explicit substitutions* are defined by (for $n \geq 0$ and $\vec{x} \in \text{Var}$):

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

As is customary, $M \cdot N$ stands for the *application* of a term M to its argument N, $\mathbf{0}$ represents the

[6] The reader interested in a comparison with other abstract machines or formalisms is invited to consult [14].

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 a list of assignments from variables to *closures* (terms with the associated substitutions), where each variable can only have one closure assigned to it.

Definition 1.2 (i) In an explicit substitution

$$\sigma = [x_1 \leftarrow (\sigma_1, M_1), \dots, x_n \leftarrow (\sigma_n, M_n)]$$

the x_i's are assumed to be fresh and distinguished.

(ii) By (i), we can define $\sigma(x_i) = (\sigma_i, M_i)$.

(iii) The *domain* of σ is given by $\mathrm{dom}(\sigma) = \{x_1, \dots, x_n\}$.

(iv) 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 $\mathrm{FV}(\lambda x.M\langle\sigma\rangle) = \mathrm{FV}(M) - (\{x\} \cup \mathrm{dom}(\sigma))$. 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 (i) *For every $n \in \mathbb{N}$, we let $\underline{n} = \mathbf{succ}^n(\mathbf{0})$. In particular, $\underline{0}$ is an alternative notation for $\mathbf{0}$.*

(ii) *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.*

(iii) *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, $\lambda x_1x_2x_3.L \cdot M \cdot N$ stands for $\lambda x_1.(\lambda x_2.(\lambda x_3.((L \cdot M) \cdot N)\langle\rangle)\langle\rangle)\langle\rangle$.*

(iv) *$M[N/x]$ denotes the capture-free substitution of N for all free occurrences of x in M.*

Example 1.3 We introduce some notations for the following (E)PCF programs, that will be used as running examples.

(i) $\mathbf{I} = \lambda x.x$, representing the identity.

(ii) $\mathbf{\Omega} = \mathbf{fix}\,(\mathbf{I})$ representing the paradigmatic looping program.

(iii) $\mathbf{succ1} = \lambda x.\mathbf{succ}\,(x)$, representing the successor function.

(iv) $\mathbf{succ2} = (\lambda sn.s \cdot (s \cdot n)) \cdot \mathbf{succ1}$, representing the function $f(x) = x + 2$.

(v) $\mathbf{add\_aux} = \lambda fxy.\mathbf{ifz}(y, x, (f \cdot (\mathbf{succ}\,x)) \cdot (\mathbf{pred}\,y))$, i.e. the functional

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

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

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

Definition 1.4 (i) We let $\mathrm{Val} = \{\underline{n} \mid n \in \mathbb{N}\} \cup \{\lambda x.M\langle\sigma\rangle \mid M \text{ is an EPCF term}\}$ be the set of EPCF *values.*

(ii) The *big-step weak reduction* is the least relation $\Downarrow_d$ from EPCF terms to Val, closed under the rules of Figure 1.

(iii) We say that an EPCF program M is *terminating* whenever $M \Downarrow_d V$ holds, for some $V \in \mathrm{Val}$. Otherwise, we say that M is a *non-terminating*, or *looping*, term.

Example 1.5 We show some of the terms from Example 1.3, at work.

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

$$\frac{\Gamma \vdash^{\mathsf{E}} M : \mathsf{int}}{\Gamma \vdash^{\mathsf{E}} \mathbf{succ}\, M : \mathsf{int}}\ (+) \qquad \frac{\Gamma \vdash^{\mathsf{E}} M : \mathsf{int}}{\Gamma \vdash^{\mathsf{E}} \mathbf{pred}\, M : \mathsf{int}}\ (-) \qquad \frac{\sigma \models \Delta \quad \Gamma, \Delta, x : \alpha \vdash^{\mathsf{E}} M : \beta}{\Gamma \vdash^{\mathsf{E}} \lambda x. M\langle \sigma \rangle : \alpha \to \beta}\ (\to_{\mathrm{I}})$$

$$\frac{\Gamma \vdash^{\mathsf{E}} M : \alpha \to \beta \quad \Gamma \vdash^{\mathsf{E}} N : \alpha}{\Gamma \vdash^{\mathsf{E}} M \cdot N : \beta}\ (\to_{\mathrm{E}}) \qquad \frac{\Gamma \vdash^{\mathsf{E}} L : \mathsf{int} \quad \Gamma \vdash^{\mathsf{E}} M : \alpha \quad \Gamma \vdash^{\mathsf{E}} N : \alpha}{\Gamma \vdash^{\mathsf{E}} \mathbf{ifz}(L, M, N) : \alpha}\ (\mathrm{ifz})$$

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

Fig. 2. EPCF type assignment system.

(i) 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:

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

(ii) 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 $[\,] \triangleright \mathbf{add} \cdot \underline{5} \cdot \underline{1} \Downarrow_d \underline{6}$.

(iii) 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.

Definition 1.6 (i) 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 $\alpha_1 \to \cdots \to \alpha_n \to \beta$ for $\alpha_1 \to (\cdots \to (\alpha_n \to \beta) \cdots)$ ($= \vec{\alpha} \to \beta$, for short).

(ii) 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 $x \notin \mathrm{dom}(\Gamma)$.

(iii) *Typing judgements* are triples, denoted $\Gamma \vdash^{\mathsf{E}} M : \alpha$, where Γ is a typing context, M is an EPCF term and $\alpha \in \mathbb{T}$.

(iv) *Typing derivations* are finite trees built bottom-up in such a way that the root has shape $\Gamma \vdash^{\mathsf{E}} M : \alpha$ and every node is an instance of a rule from Figure 2. In the rule $(\to_{\mathrm{I}})$ 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.

(v) When writing $\Gamma \vdash^{\mathsf{E}} M : \alpha$, we mean that this typing judgement is derivable.

(vi) We say that M *is typable* if $\Gamma \vdash^{\mathsf{E}} M : \alpha$ is derivable for some Γ, α.

Example 1.7 The following are examples of derivable typing judgments.

(i)

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

(ii) $\vdash^{\mathsf{E}} (\lambda x.\mathbf{succ}\,(x)) \cdot \mathbf{0} : \mathsf{int}$.

(iii) $\vdash^{\mathsf{E}} (\lambda sn.s \cdot (s \cdot n)) \cdot (\lambda x.\mathbf{succ}\,(x)) : \mathsf{int} \to \mathsf{int}$.

(iv) $\vdash^{\mathsf{E}} \mathbf{fix}\,(\lambda fxy.\mathbf{ifz}(y, x, f \cdot (\mathbf{succ}\,x) \cdot (\mathbf{pred}\,y))) : \mathsf{int} \to \mathsf{int} \to \mathsf{int}$.

(v) $\vdash^{\mathsf{E}} \mathbf{\Omega} : \alpha$, for all $\alpha \in \mathbb{T}$.

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

Lemma 1.8 *Let M be an* EPCF *term, $V \in \mathrm{Val}$, $\alpha, \beta \in \mathbb{T}$ and Γ be a context.*

(i) *(Syntax directedness) Every derivable judgement $\Gamma \vdash^{\mathsf{E}} M : \alpha$ admits a unique derivation.*

(ii) *(Strengthening) If $\Gamma, x : \beta \vdash^{\mathsf{E}} M : \alpha$ and $x \notin \mathrm{FV}(M)$ then $\Gamma \vdash^{\mathsf{E}} M : \alpha$.*

(iii) *(Subject reduction) For M closed, $\vdash^{\mathsf{E}} M : \alpha$ and $[\,] \triangleright M \Downarrow_d V$ entail $\vdash^{\mathsf{E}} V : \alpha$.*

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 [11] with instructions for performing arithmetic operations and conditional testing. Natural numbers are 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.

Let $\varnothing \notin \mathbb{A}$ be a “null” constant corresponding to an uninitialised register. Set $\mathbb{A}_\varnothing = \mathbb{A} \cup \{\varnothing\}$.

Definition 2.1 (i) An $\mathbb{A}$-*valued tape* T is a finite ordered list of addresses $T = [a_1, \dots, a_n]$ 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.

(ii) 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.

(iii) 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 ML, where ! represents an explicit dereferencing operator.)

(iv) Given $\mathbb{A}_\varnothing$-valued registers $R_0, \dots, R_n$ 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. $\vec{R}[R_i := a] = \vec{R}$.

Intuitively, the contents of the registers $R_0, \dots, R_n$ constitutes the *state* of a machine, while the tape correspond to the list of its inputs. The addressing machines from [11] 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. $k \leftarrow$ `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 2.4).

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 $!R_i \in \mathbb{N}$, the value of R_j becomes $!R_i \ominus 1 = \max(!R_i - 1, 0)$.
6. $j \leftarrow \texttt{Succ}(i)$: if $!R_i \in \mathbb{N}$, then the value of R_j becomes $!R_i + 1$.

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 3.5, below).

Definition 2.2 (i) 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:

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

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.

(ii) In a program, we write $\texttt{Load}\ (i_1, \dots, i_n)$ as an abbreviation for the instructions $\texttt{Load}\ i_1; \cdots; \texttt{Load}\ i_n$.

(iii) Let P be a program, $r \geq 0$, and $\mathcal{I} \subseteq \{0, \dots, r-1\}$ be a set of indices corresponding to the indices of initialized registers. Define the relation $\mathcal{I} \models^r P$, whose intent is to specify that P does not read uninitialized registers, as the least relation closed under the rules:

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

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

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

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

Example 2.3 For each of these programs, we specify its validity with respect to $R_0 = 7, R_1 = a, R_2 = \varnothing$ (i.e., $r = 3$).

$P_1 = 2 \leftarrow \texttt{Pred}(0);\ \texttt{Call}\ 2$ (valid)

$P_2 = \texttt{Load}\ (2, 8);\ 0 \leftarrow \texttt{Test}(0,\ 1,\ 2);\ \texttt{Call}\ 0$ (valid)

$P_3 = \texttt{Load}\ (0, 2, 8);\ \texttt{Call}\ 8$ (calling the uninitialized register R_8, thus not valid)

Lemma 2.4 *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. Decidability follows from the syntax directedness of Definition 2.2(iii), and the preservation of the invariant $\mathcal{I} \subseteq \{0, \dots, r-1\}$, since $\mathcal{I}$ is only extended with $k < r$. ☐

Definition 2.5 (i) An *extended addressing machine* (*EAM*) M with r registers over $\mathbb{A}$ is given by a tuple:

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

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.

(ii) 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 $!R_i = a$.

(iii) We say that an extended addressing machine M as above is *stuck*, written $\mathsf{stuck}(\mathsf{M})$, whenever its program has shape $\mathsf{M}.P = \mathtt{Load}\ i; P$ but its input-tape is empty $\mathsf{M}.T = []$. Otherwise M is *ready*, written $\neg\mathsf{stuck}(\mathsf{M})$.

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

(v) For $n \geq 0$, the *n-th numeral machine* is defined $\mathsf{n} = \langle R_0, \varepsilon, []\rangle$ with $!R_0 = n$.

(vi) For $n \geq 0$ and $a \in \mathbb{A}$, define

$$\mathsf{Y}_n^a = \langle (R_0 = a, R_1 = \varnothing, \dots, R_{n+1} = \varnothing, P, []\rangle$$

where $P = \mathtt{Load}\ (1, \dots, n+1); 0 \leftarrow \mathtt{App}(0,\ 1); \cdots ; 0 \leftarrow \mathtt{App}(0,\ n+1); 1 \leftarrow \mathtt{App}(1,\ 2); \cdots ;$
$1 \leftarrow \mathtt{App}(1,\ n+1); 1 \leftarrow \mathtt{App}(1,\ 0); \mathtt{Call}\ 1$

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

Definition 2.6 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 2.5(vi).

(i) 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. $\#(\mathsf{Y}_n) = a_{\mathsf{Y}_n}$.

(ii) For $a \in \mathbb{A}$, we write $\#^{-1}(a)$ for the unique machine having address a, i.e., $\#^{-1}(a) = \mathsf{M} \iff \#\mathsf{M} = a$.

(iii) Given $\mathsf{M} \in \mathcal{M}_\mathbb{A}$ and $T' \in \mathcal{T}_\mathbb{A}$, we write $\mathsf{M} @ T'$ for the machine $\langle \mathsf{M}.\vec{R}, \mathsf{M}.P, \mathsf{M}.T @ T'\rangle$.

(iv) Define the *application map* $(\cdot) : \mathbb{A} \times \mathbb{A} \to \mathbb{A}$ by setting $a \cdot b = \#(\#^{-1}(a) @ [b])$, 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 $\#^{-1}(a)$.

Example 2.7 The following are examples of EAMs (whose registers are assumed uninitialized, i.e. $\vec{R} = \vec{\varnothing}$).

(i) $\mathsf{Succ1} := \langle R_0, \mathtt{Load}\ 0; 0 \leftarrow \mathtt{Succ}(0); \mathtt{Call}\ 0, []\rangle$.

(ii) $\mathsf{Succ2} := \langle R_0, R_1, \mathtt{Load}\ 0; \mathtt{Load}\ 1; 1 \leftarrow \mathtt{App}(0,\ 1); 1 \leftarrow \mathtt{App}(0,\ 1); \mathtt{Call}\ 1, [a_\mathsf{S}]\rangle$, where $a_\mathsf{S} = \#\mathsf{Succ1}$.

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

Remark 2.8 In general, there are uncountably many possible address table maps of arbitrary computational complexity. A natural example of such maps is given by *Gödelization*, which can be performed effectively. The framework is however more general and allows to consider non-r.e. sets of addresses like the complement K^c of the halting set

$$K = \{(i, x) \mid \text{the program } i \text{ terminates when run on input } x\}$$

and a non-computable function $\# : \mathcal{M}_{K^c} \to K^c$ as a map.

In an implementation of EAMs the address table map should be computable—one can choose a fresh address from $\mathbb{A}$ whenever a new machine is constructed, save the correspondence in some table and retrieve it in constant time.

Remark 2.9 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., $\mathsf{M}_n = \langle R_0 = \#\mathsf{M}_{n+1}, \varepsilon, []\rangle$.

The results we present are independent from the choice of #.

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.

Definition 2.10 We introduce a fresh constant $\mathtt{err} \notin \mathcal{M}_{\mathbb{A}}$ to represent a machine raising an error.

(i) Define a reduction strategy $\to_{\mathsf{c}}$ on EAMs, representing one step of computation, as the least relation $\to_{\mathsf{c}} \subseteq \mathcal{M}_{\mathbb{A}} \times (\mathcal{M}_{\mathbb{A}} \cup \{\mathtt{err}\})$ closed under the rules in Figure 3.

(ii) The *multistep reduction* $\twoheadrightarrow_{\mathsf{c}}$ is defined as the transitive-reflexive closure of $\to_{\mathsf{c}}$.

(iii) 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 (unique) reduction path from M to N.

(iv) 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 $\mathsf{Z} \in \mathcal{M}_{\mathbb{A}} \cup \{\mathtt{err}\}$, i.e. $\mathsf{M} \twoheadrightarrow_{\mathsf{c}} \mathsf{Z} \;{}_{\mathsf{c}}\!\twoheadleftarrow \mathsf{N}$.

(v) 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 $\mathsf{M} \twoheadrightarrow_{\mathsf{c}} \mathtt{err}$; *does not terminate*, otherwise.

Notice that since the redexes in Figure 3 are not overlapping, the confluence of $\twoheadrightarrow_{\mathsf{c}}$ follows easily (cf. [11, Lemma 2.11(2)]).

Lemma 2.11 *If* $\mathsf{M} \twoheadrightarrow_{\mathsf{c}} \mathsf{M}'$*, then* $\mathsf{M} @ \#\mathsf{N} \twoheadrightarrow_{\mathsf{c}} \mathsf{M}' @ \#\mathsf{N}$.

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

Example 2.12 See Example 2.7 for the definition of Succ1, Succ2, Add_aux.

(i) We have $\mathsf{Succ1} @ [0] \twoheadrightarrow_{\mathsf{c}} 1$ and $\mathsf{Succ2} @ [1] \twoheadrightarrow_{\mathsf{c}} 3$.

(ii) Define $\mathsf{Add} = \mathsf{Y}_0 @ [\#\mathsf{Add\_aux}]$, an EAM performing the addition. We show:

$$
\begin{aligned}
&\mathsf{Add} @ [1,3] \to_{\mathsf{c}} \langle (R_0 = a_{\mathsf{Y}_0}, R_1 = \#\mathsf{Add\_aux}), 0 \leftarrow \mathtt{App}(0,\ 1); 1 \leftarrow \mathtt{App}(1,\ 0); \mathtt{Call}\ 1, [1,3]\rangle \\
&\twoheadrightarrow_{\mathsf{c}} \left\langle \begin{array}{r} \vec{R}, \mathtt{Load}\ (0,1,2); 3 \leftarrow \mathtt{Pred}(1); 4 \leftarrow \mathtt{Succ}(2); 0 \leftarrow \mathtt{App}(0,\ 3); 0 \leftarrow \mathtt{App}(0,\ 4); \\ 0 \leftarrow \mathtt{Test}(1,\ 2,\ 0); \mathtt{Call}\ 0, [\#\mathsf{Add}, 1, 3] \end{array} \right\rangle \\
&\twoheadrightarrow_{\mathsf{c}} \left\langle \begin{array}{r} R_0 = \#\mathsf{Add}, R_1 = 1, R_2 = 3, R_3, R_4, 3 \leftarrow \mathtt{Pred}(1); 4 \leftarrow \mathtt{Succ}(2); 0 \leftarrow \mathtt{App}(0,\ 3); \\ 0 \leftarrow \mathtt{App}(0,\ 4); 0 \leftarrow \mathtt{Test}(1,\ 2,\ 0); \mathtt{Call}\ 0, [] \end{array} \right\rangle \\
&\twoheadrightarrow_{\mathsf{c}} \langle R_0 = \#(\mathsf{Add} @ [0,4]\,), R_1 = 1, R_2 = 3, R_3 = 0, R_4 = 4, 0 \leftarrow \mathtt{Test}(1,\ 2,\ 0); \mathtt{Call}\ 0, []\rangle \\
&\twoheadrightarrow_{\mathsf{c}} \langle R_0 = \#(\mathsf{Add} @ [0,5]\,), R_1 = 0, R_2 = 4, R_3 = 0, R_4 = 5, 0 \leftarrow \mathtt{Test}(1,\ 2,\ 0); \mathtt{Call}\ 0, []\rangle \twoheadrightarrow_{\mathsf{c}} 4
\end{aligned}
$$

(iii) For $\mathsf{I} = \langle R_0 = \varnothing, \mathtt{Load}\ 0; \mathtt{Call}\ 0, []\rangle$, $\mathsf{Y}_0 @ [\#\mathsf{I}] \twoheadrightarrow_{\mathsf{c}} \mathsf{I} @ [\#(\mathsf{Y}_0 @ [\#\mathsf{I}]\,)]$.

(iv) $\mathsf{Y}_n @ [\#\mathsf{M}, d_1, \ldots, d_n] \twoheadrightarrow_{\mathsf{c}} \mathsf{M} @ [d_1, \ldots, d_n, \#(\mathsf{Y}_n @ [\#\mathsf{M}, d_1, \ldots, d_n]\,)]$, for all $n \geq 0$, $\mathsf{M} \in \mathcal{M}_{\mathbb{A}}$, $\vec{d} \in \mathbb{A}$.

Unconditional rewriting rules

$$\langle \vec{R}, \texttt{Call}\ i; P, T\rangle \to_{\mathsf{c}} \#^{-1}(!R_i)\,@\,T$$

$$\langle \vec{R}, \texttt{Load}\ i; P, a :: T\rangle \to_{\mathsf{c}} \langle \vec{R}[R_i := a], P, T\rangle$$

$$\langle \vec{R}, k \leftarrow \texttt{App}(i,\ j); P, T\rangle \to_{\mathsf{c}} \langle \vec{R}[R_k := !R_i \cdot !R_j], P, T\rangle$$

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

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

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

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

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

$$\langle \vec{R}, j \leftarrow \texttt{Pred}(i); P, T\rangle \to_{\mathsf{c}} \langle \vec{R}[R_i := \#\mathsf{A}], j \leftarrow \texttt{Pred}(i); P, T\rangle$$

$$\langle \vec{R}, j \leftarrow \texttt{Succ}(i); P, T\rangle \to_{\mathsf{c}} \langle \vec{R}[R_i := \#\mathsf{A}], j \leftarrow \texttt{Succ}(i); P, T\rangle$$

$$\langle \vec{R}, l \leftarrow \texttt{Test}(i,\ j,\ k); P, T\rangle \to_{\mathsf{c}} \langle \vec{R}[R_i := \#\mathsf{A}], l \leftarrow \texttt{Test}(i,\ j,\ k); P, T\rangle$$

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

3 Typing Algorithm

Recall that the set $\mathbb{T}$ of (simple) types has been introduced in Definition 1.6(i). We now show that certain EAMs can be typed, and that typable machines do not raise error during their execution.

Definition 3.1 (i) 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.

(ii) 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 $\Delta[R_i : \alpha] = \Delta, R_i : \alpha$.

(iii) 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 take precedence over $(R_\varnothing)$ and $(R_\mathbb{T})$.

(iv) For $R_{i_1}, \ldots, R_{i_n} \in \vec{R}$, write $R_{i_1} : \beta_{i_1}, \ldots, R_{i_n} : \beta_{i_n} \models \vec{R}$ if $\#^{-1}(!R_j) : \beta_j$, 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 are recognizable from their addresses and the rules (nat) and (fix) can thus be given 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

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

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

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

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

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

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

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

Fig. 4. Typing rules for extended addressing machines.

performed, one needs to derive a judgement of the form $R_{i_1} : \beta_{i_1}, \dots, R_{i_n} : \beta_{i_n} \Vdash (P, T) : \alpha$, 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 3.2 (i) 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.

(ii) If $\#\mathsf{M} \notin \mathbb{N} \cup \{a_{\mathsf{Y}_n} \mid n \geq 0\}$, then $\vdash \mathsf{M} : \alpha \iff \exists \Delta\,.\,[\Delta \models \mathsf{M}.\vec{R}\ \wedge\ \Delta \Vdash (\mathsf{M}.P, \mathsf{M}.T) : \alpha]$

(iii) The higher priority assigned to the rules (nat) and (fix) does not modify the set of typable machines, rather guarantees the syntax-directedness of the system.

Example 3.3 The following typing judgements are derivable.

(i) $\vdash \mathsf{Succ2} : \mathsf{int} \to \mathsf{int}$

(ii) $\vdash \mathsf{Add} : \mathsf{int} \to \mathsf{int} \to \mathsf{int}$, where $\mathsf{Add} = \mathsf{Y}_0 \,@\, [\#\mathsf{Add\_aux}]$

(iii) For a smaller example, like $\vdash \mathsf{Succ1} : \mathsf{int} \to \mathsf{int}$, we can provide the whole derivation tree:

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

Lemma 3.4 *Let* $\mathsf{M} \in \mathcal{M}_{\mathbb{A}}$, $\alpha \in \mathbb{T}$. *Assume that* $\# : \mathsf{M} \to \mathbb{A}$ *is effectively given.*

(i) *If* $\mathsf{M} = \langle \vec{R} = \varnothing, P, []\rangle$ *then the typing algorithm is capable of deciding whether* $\vdash \mathsf{M} : \alpha$ *holds.*

(ii) *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 ($\mathrm{load}_{\mathbb{T}}$), 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 2.9. □

The machine M_0 in Remark 2.9 cannot be typable because it would require an infinite derivation tree.

Proposition 3.5 *Let* $\mathsf{M}, \mathsf{M}', \mathsf{N}, \in \mathcal{M}_{\mathbb{A}}$ *and* $\alpha, \beta \in \mathbb{T}$*.*

(i) *If* $\vdash \mathsf{M} : \beta \to \alpha$ *and* $\vdash \mathsf{N} : \beta$ *then* $\vdash \mathsf{M} \,@\, [\#\mathsf{N}] \, : \alpha$.

(ii) *If* $\vdash \mathsf{M} : \alpha$ *and* $\mathsf{M} \to_{\mathsf{c}} \mathsf{N}$ *then* $\vdash \mathsf{N} : \alpha$.

(iii) *If* $\vdash \mathsf{M} : \mathsf{int}$ *then either* M *does not terminate or* $\mathsf{M} \twoheadrightarrow_{\mathsf{c}} \mathsf{n}$*, for some* $n \geq 0$*.*

(iv) *If* $\vdash \mathsf{M} : \alpha$ *then* M *does not raise an error.*

Proof. (i) Simultaneously, one proves that $\Delta \Vdash (P, T) : \beta \to \alpha$ and $\vdash \mathsf{N} : \beta$ imply $\Delta \Vdash (P, T \,@\, [\#\mathsf{N}]\,) : \alpha$. 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 ($\mathrm{fix_n}$). We show the case for $n = 0$, the others being similar. By definition of Y_0, we have:

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

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

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

Case $\mathrm{load}_{\varnothing}$. Then $P = \mathtt{Load}\ i; P'$, $T = []$ and $\Delta[R_i : \beta] \Vdash (P', []) : \alpha$. By assumption $\vdash \mathsf{N} : \beta$, so we conclude $\Delta \Vdash (\mathtt{Load}\ i; P', []) : \alpha$ by applying $\mathrm{load}_{\mathbb{T}}$. All other cases derive straightforwardly from the IH.

(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 3.2(ii), $\Delta \Vdash (\mathsf{M}.P, \mathsf{M}.T) : \alpha$ for some $\Delta \models \mathsf{M}.\vec{R}$. By cases on the shape of $\mathsf{M}.P$.

Case $P = \mathtt{Load}\ i; P'$. 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 ($\mathrm{Load}_{\mathbb{T}}$) 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 $\Delta[R_i : \beta] \models \vec{R}[R_i := a]$, so as $N = \left\langle \vec{R}[R_i := a], P', T' \right\rangle$, by Remark 3.2(ii), $\vdash N : \alpha$.

Case $P = \mathtt{Call}\ i$. 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, $\mathsf{N} = \#^{-1}(!(\mathsf{M}.R_i)) \,@\, T$ 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 IH.

(iii) 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.

(iv) The three cases from Figure 3 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. □

4 Translation and Simulation

We define a type-preserving translation from EPCF terms to extended addressing machines. More precisely, we show that if $\Gamma \vdash^{\mathsf{E}} M : \alpha$ is derivable then M is transformed into a machine M which is typable with the same α. By Proposition 3.5, 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 4.1 *Let $n \geq 0$. There are EAMs satisfying (for all $a, b, c, d_1, \dots, d_n \in \mathbb{A}$):*

(i) $\mathsf{Pr}^n_i \,@\, [d_1, \dots, d_n] \twoheadrightarrow_{\mathsf{c}} d_i$, *for* $1 \leq i \leq n$;

(ii) $\mathsf{Apply}_n \,@\, [a, b, d_1, \dots, d_n] \twoheadrightarrow_{\mathsf{c}} \#^{-1}(a) \,@\, [d_1, \dots, d_n, b \cdot d_1 \cdots d_n]$;

(iii) $\mathsf{Pred}_n \,@\, [a, d_1, \dots, d_n] \twoheadrightarrow_{\mathsf{c}} \langle R_0 = a \cdot d_1 \cdots d_n, \vec{R}, ; 0 \leftarrow \texttt{Pred}(0); \texttt{Call } 0, [] \rangle$;

(iv) $\mathsf{Succ}_n \,@\, [a, d_1, \dots, d_n] \twoheadrightarrow_{\mathsf{c}} \langle R_0 = a \cdot d_1 \cdots d_n, \vec{R}, ; 0 \leftarrow \texttt{Succ}(0); \texttt{Call } 0, [] \rangle$;

(v) $\mathsf{Ifz}_n \,@\, [a, b, c, d_1, \dots, d_n] \twoheadrightarrow_{\mathsf{c}} \langle R_0 = a \cdot \vec{d}, R_1 = b \cdot \vec{d}, R_2 = c \cdot \vec{d}, \vec{R}, 0 \leftarrow \texttt{Test}(0,\ 1,\ 2); \texttt{Call } 0, [] \rangle$.

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

$$\mathsf{Pred}_n = \langle R_0, \dots, R_n, \texttt{Load } (0, \dots, n); 0 \leftarrow \texttt{App}(0,\ 1); \cdots ; 0 \leftarrow \texttt{App}(0,\ n); 0 \leftarrow \texttt{Pred}(0); \texttt{Call } 0, [] \rangle$$

The others are similar. □

Lemma 4.2 *The EAMs in the previous lemma can be defined in order to ensure their typability (for all $n \geq 0$, $\alpha, \beta, \gamma, \delta_i \in \mathbb{T}$):*

(i) $\vdash \mathsf{Pr}^n_i : \vec{\delta} \to \delta_i$, *with* $\vec{\delta} = \delta_1 \to \cdots \to \delta_n$;

(ii) $\vdash \mathsf{Apply}_n : (\vec{\delta} \to \beta \to \alpha) \to (\vec{\delta} \to \beta) \to \vec{\delta} \to \alpha$;

(iii) $\vdash \mathsf{Pred}_n : (\vec{\delta} \to \mathsf{int}) \to \vec{\delta} \to \mathsf{int}$;

(iv) $\vdash \mathsf{Succ}_n : (\vec{\delta} \to \mathsf{int}) \to \vec{\delta} \to \mathsf{int}$;

(v) $\vdash \mathsf{Ifz}_n : (\vec{\delta} \to \mathsf{int}) \to (\vec{\delta} \to \alpha) \to (\vec{\delta} \to \alpha) \to \vec{\delta} \to \alpha$;

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

We will show that, using the auxiliary EAMs given in Lemma 4.2, we can translate any EPCF term into an EAM. In order to proceed by induction, we first need to define the size of an EPCF term.

Definition 4.3 Let M be an EPCF term and σ be an explicit substitution. The sizes $|-|$ of σ, M and (σ, M) are defined by mutual induction, e.g.

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

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

Intuitively, an EPCF term M having $x_1, \dots, x_n$ as free variables is translated as an EAM M loading n arguments as input.

Definition 4.4 (Translation) Let M be an EPCF term and σ be an explicit substitution such that $\mathrm{FV}(M) \subseteq \mathrm{dom}(\sigma) \cup \{\vec{x}\}$, where $\vec{x} = x_1, \dots, x_n$. The *translation* of the pair (σ, M) (w.r.t $\vec{x}$) is a machine denoted $(\!|\sigma, M|\!)^{\vec{x}} \in \mathcal{M}_{\mathbb{A}}$, or simply $[\![M]\!]^{\vec{x}}$ when σ is empty.[7] The machine $(\!|\sigma, M|\!)^{\vec{x}}$ is defined by

[7] In other words, we set $[\![M]\!]^{\vec{x}} = (\!|[], M|\!)^{\vec{x}}$.

induction on $|(\sigma, M)|$ as follows:

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

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

Example 4.5 1. $[\![(\lambda x.\mathbf{succ}\,(x)\langle\rangle) \cdot \underline{0}]\!] = \mathsf{Succ}_1 \,@\, [\#\mathsf{Pr}_1^1, 0]\,.$

2. $[\![(\lambda sn.s(sn))(\lambda x.\mathbf{succ}\,(x))]\!] = \mathsf{Apply}_2 \,@\, [\#\mathsf{Pr}_1^2, \#(\mathsf{Apply}_2 \,@\, [\#\mathsf{Pr}_1^2, \#\mathsf{Pr}_2^2]\,), \#(\mathsf{Succ}_1 \,@\, [\#\mathsf{Pr}_1^1]\,)]\,.$

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

where $[\![f \cdot (\mathbf{succ}\, x) \cdot (\mathbf{pred}\, y)]\!]^{f,x,y} = \mathsf{Apply}_3 \,@\, [\#(\mathsf{Apply}_3 \,@\, [\#\mathsf{Pr}_1^3, \#(\mathsf{Succ}_3 \,@\, [\#\mathsf{Pr}_2^3]\,)]\,), \#(\mathsf{Pred}_3 \,@\, [\#\mathsf{Pr}_3^3]\,)]\,.$

Theorem 4.6 *Let M be an* EPCF *term, $\alpha \in \mathbb{T}$, $\Gamma = x_1 : \beta_1, \dots, x_n : \beta_n$. Then*

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

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

Theorem 4.7 *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 4.8 *For an* EPCF *program M of type $\vdash^{\mathsf{E}} M : \mathsf{int}$ we have*

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

Proof. Assume that $[\,] \triangleright M \Downarrow_d \underline{n}$. By Theorem 4.7, 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

Let us show how to apply our machinery to the usual (call-by-name) PCF. Our presentation follows [22].

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

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

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

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

$$\frac{}{\Gamma, x : \alpha \vdash x : \alpha} \qquad \frac{\Gamma, x : \alpha \vdash P : \beta}{\Gamma \vdash \lambda x.P : \alpha \to \beta} \qquad \frac{}{\Gamma \vdash \underline{0} : \mathsf{int}} \qquad \frac{\Gamma \vdash P : \alpha \to \beta \quad \Gamma \vdash Q : \alpha}{\Gamma \vdash PQ : \beta}$$

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

Fig. 6. The type inference rules of PCF.

Definition 4.9 (i) PCF *terms* are defined by the grammar (for $n \geq 0$, $x \in \mathrm{Var}$):

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

(ii) A closed PCF term P is called a PCF *program*.

(iii) A PCF value U is a term of the form $\lambda x.P$ or $\underline{n}$, for some $n \geq 0$.

(iv) 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.

(v) The set $\mathbb{T}$ of *simple types* and *typing contexts* have already been defined in items (i) and (ii) of Definition 1.6, respectively.

(vi) 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.

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 $\vdash^{\mathsf{PCF}} P : \mathsf{int}$ and $P \Downarrow \underline{n}$ are *a priori* not sufficient for applying Corollary 4.8, since one needs to show that also the corresponding EPCF judgments $\vdash^{\mathsf{E}} P : \mathsf{int}$ and $P \Downarrow_d \underline{n}$ hold. The former is established by the following lemma.

Lemma 4.10 *Let M be a* PCF *term, $\alpha \in \mathbb{T}$ and Γ be a context. Then*

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

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 $P = P'[Q_1/x_1, \ldots, Q_n/x_n]$. Recall that the size $|(\sigma, M)|$ has been defined in Definition 4.3.

Definition 4.11 Let M be an EPCF term and σ be an explicit substitution. Define a PCF term $(\sigma, M)^*$

by induction on $|(\sigma, M)|$ as follows:

$$
\begin{aligned}
(\sigma, x)^* &= \begin{cases} \sigma(x)^*, & \text{if } x \in \mathrm{dom}(\sigma), \\ x, & \text{otherwise}, \end{cases} \\
(\sigma, \lambda x.M\langle\rho\rangle)^* &= \lambda x.(\sigma + \rho, M)^*, \\
(\sigma, M \cdot N)^* &= (\sigma, M)^* \cdot (\sigma, N)^*, \\
(\sigma, \mathbf{fix}\, M)^* &= \mathbf{fix}\,((\sigma, M)^*), \\
(\sigma, \underline{0})^* &= \underline{0}, \\
(\sigma, \mathbf{pred}\, M)^* &= \mathbf{pred}\,(\sigma, M)^*, \\
(\sigma, \mathbf{succ}\, M)^* &= \mathbf{succ}\,(\sigma, M)^*, \\
(\sigma, \mathbf{ifz}(L, M, N))^* &= \mathbf{ifz}((\sigma, L)^*, (\sigma, M)^*, (\sigma, N)^*).
\end{aligned}
$$

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

To show the equivalence between PCF and EPCF, we need yet another auxiliary lemma.

Lemma 4.12 (Substitution Lemma)

(i) *Let M, N be* EPCF *terms, σ, ρ be explicit substitutions and x be a variable.*

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

(ii) *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. (i) By structural induction on M.

Case $M = y$, with $y \neq x$: There are two subcases.

- If $y \in \mathrm{dom}(\sigma)$, then $(\sigma + [x \leftarrow (\rho, N)], y)^* = \sigma(y) = \sigma(y)[(\rho, N)^*/x] = (\sigma, y)^*[(\rho, N)^*/x]$, since $x \notin \mathrm{FV}(\sigma(y))$;
- if $y \notin \mathrm{dom}(\sigma)$, then $(\sigma + [x \leftarrow (\rho, N)], y)^* = y = y[(\rho, N)^*/x] = (\sigma, y)^*[(\rho, N)^*/x]$.

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

Case $M = \lambda y.M'$: Wlog, we may assume $y \neq x$. We have $(\sigma + [x \leftarrow (\rho, N)], \lambda y.M')^* = \lambda y.(\sigma + [x \leftarrow (\rho, N)], M')^* = \lambda y.((\sigma, M')^*[(\rho, N)^*/x]) = (\lambda y.(\sigma, M')^*)[(\rho, N)^*/x] = ((\sigma, \lambda y.M')^*)[(\rho, N)^*/x]$.

All other cases derive straightforwardly from the IH.

(ii) By an easy induction on P, using (i). □

Theorem 4.13 *The big-step weak reduction of* EPCF *is equivalent to the usual big-step operational semantics of* PCF*. Formally:*

(i) *Given an* EPCF *program M, a value V and an explicit substitution σ, we have:*

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

(ii) *Given a* PCF *program P and* PCF *value U. If $P \Downarrow U$ then*

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

Proof. *(Proof sketch) For the full proof, we refer to the technical Appendix A.*

(i) Proceed by induction on a derivation of $\sigma \triangleright M \Downarrow_d V$, using Lemma 4.12(i) in the (β_v)-case.

(ii) By induction on the lexicographically ordered pairs, whose first component is the length of a derivation of $P \Downarrow U$ and second component is $|(\sigma, M)|$, using Lemma 4.12(ii) in the (β_v)-case. ☐

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 4.14 *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 $\vdash^{\mathsf{E}} P : \mathsf{int}$ by Lemma 4.10. Thus $[\,] \triangleright P \Downarrow_d \underline{n}$ by Theorem 4.13(ii). Conclude by Corollary 4.8. ☐

References

[1] Abadi, M., L. Cardelli, P.-L. Curien and J.-J. Lévy, *Explicit substitutions*, in: F. E. Allen, editor, *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, pages 31–46, ACM Press (1990).
https://doi.org/10.1145/96709.96712

[2] Abadi, M., L. Cardelli, P.-L. Curien and J.-J. Lévy, *Explicit substitutions*, J. Funct. Program. **1**, pages 375–416 (1991).
https://doi.org/10.1017/S0956796800000186

[3] Abramsky, S., R. Jagadeesan and P. Malacaria, *Full abstraction for PCF*, Inf. Comput. **163**, pages 409–470 (2000).
https://doi.org/10.1006/inco.2000.2930

[4] Abramsky, S., P. Malacaria and R. Jagadeesan, *Full abstraction for PCF*, in: M. Hagiya and J. C. Mitchell, editors, *Theoretical Aspects of Computer Software, International Conference TACS '94, Sendai, Japan, April 19-22, 1994, Proceedings*, volume 789 of *Lecture Notes in Computer Science*, pages 1–15, Springer (1994).
https://doi.org/10.1007/3-540-57887-0_87

[5] Berry, G., P.-L. Curien and J.-J. Lévy, *Full abstraction for sequential languages: state of the art*, in: M. Nivat and J. Reynolds, editors, *Algebraic methods in semantics*, pages 89–132, Cambridge University Press (1985).
Available online at https://hal.inria.fr/inria-00076361/document.

[6] Cregut, P., *Machines a environnement pour la reduction symbolique et l'evaluation partielle*, Ph.D. thesis, Université Paris-Diderot (Paris VII) (1991). In French.
http://www.theses.fr/1991PA077152

[7] Curien, P., *Definability and full abstraction*, Electron. Notes Theor. Comput. Sci. **172**, pages 301–310 (2007).
https://doi.org/10.1016/j.entcs.2007.02.011

[8] 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*, pages 432–443, IEEE Computer Society (1992).
https://doi.org/10.1109/LICS.1992.185554

[9] Curien, P.-L., T. Hardin and J.-J. Lévy, *Confluence properties of weak and strong calculi of explicit substitutions*, J. ACM **43**, pages 362–397 (1996).
https://doi.org/10.1145/226643.226675

[10] 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.

[11] Della Penna, G., B. Intrigila and G. Manzonetto, *Addressing machines as models of lambda-calculus*, Log. Methods Comput. Sci. **18** (2022).
https://doi.org/10.48550/arXiv.2107.00319

[12] Fairbairn, J. and S. Wray, *Tim: A simple, lazy abstract machine to execute supercombinators*, in: G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 34–45, Springer Berlin Heidelberg, Berlin, Heidelberg (1987), ISBN 978-3-540-47879-9.
https://doi.org/10.1007/3-540-18317-5_3

[13] Hyland, J. M. E. and C. L. Ong, *On full abstraction for PCF: I, II, and III*, Inf. Comput. **163**, pages 285–408 (2000).
https://doi.org/10.1006/inco.2000.2917

[14] Intrigila, B., G. Manzonetto and N. Münnich, *Extended addressing machines a comparative view* (2022).
https://lipn.univ-paris13.fr/~gmanzonetto/papers/NotesIMM22

[15] Krivine, J.-L., *A call-by-name lambda-calculus machine*, Higher Order Symbol. Comput. **20**, page 199–207 (2007), ISSN 1388-3690.
https://doi.org/10.1007/s10990-007-9018-9

[16] Landin, P. J., *The Mechanical Evaluation of Expressions*, The Computer Journal **6**, pages 308–320 (1964), ISSN 0010-4620. https://academic.oup.com/comjnl/article-pdf/6/4/308/1067901/6-4-308.pdf.
https://doi.org/10.1093/comjnl/6.4.308

[17] Lang, F., *Explaining the lazy Krivine machine using explicit substitution and addresses*, Higher-Order and Symbolic Computation (2007).
https://hal.inria.fr/inria-00198756

[18] Leroy, X., *The ZINC experiment: an economical implementation of the ML language*, Technical report 117, INRIA (1990).

[19] Lévy, J.-J. and L. Maranget, *Explicit substitutions and programming languages*, in: C. P. Rangan, V. Raman and R. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science, 19th Conference, Chennai, India, December 13-15, 1999, Proceedings*, volume 1738 of *Lecture Notes in Computer Science*, pages 181–200, Springer (1999).
https://doi.org/10.1007/3-540-46691-6_14

[20] Longley, J., *Realizability toposes and language semantics*, Ph.D. thesis, University of Edinburgh (1995).
https://www.lfcs.inf.ed.ac.uk/reports/95/ECS-LFCS-95-332/

[21] Milner, R., *Fully abstract models of typed λ-calculi*, Theor. Comput. Sci. **4**, pages 1–22 (1977).
https://doi.org/10.1016/0304-3975(77)90053-6

[22] Ong, C.-H. L., *Correspondence between operational and denotational semantics of PCF*, in: S. Abramsky, D. Gabbay and T. S. E. Maibaum, editors, *Semantic Modelling*, volume 4 of *Handbook of Logic in Computer Science*, pages 269–356, Oxford University Press (1995).

[23] Plotkin, G. D., *LCF considered as a programming language*, Theor. Comput. Sci. **5**, pages 223–255 (1977).
https://doi.org/10.1016/0304-3975(77)90044-5

[24] Seaman, J. and S. P. Iyer, *An operational semantics of sharing in lazy evaluation*, Sci. Comput. Program. **27**, pages 289–322 (1996).
https://doi.org/10.1016/0167-6423(96)00012-3

A Technical Appendix

This technical appendix is devoted to provide the proofs that have been partially given, or completely omitted, in the body of the paper. As an abbreviation, we write IH for "induction hypothesis".

A.1 Proofs of Section 1

Proof. [Proof of Lemma 1.8] Items (i) and (ii) are straightforward. We prove (iii).

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 IH we conclude $\vdash V : \alpha$.

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

A.2 Proofs of Section 4

Proof. [Proof of Theorem 4.6] We prove the following statements by mutual induction and call the respective inductive hypotheses IH1 and IH2.

(i) Let M be an EPCF term, $\Gamma = x_1 : \beta_1, \dots, x_n : \beta_n$ and $\alpha \in \mathbb{T}$. Then $\Gamma \vdash M : \alpha \quad\Rightarrow\quad \vdash [\![M]\!]^{x_1,\dots,x_n} : \beta_1 \to \cdots \to \beta_n \to \alpha$.

(ii) For all $\Gamma = x_1 : \beta_1, \dots, x_n : \beta_n$ and σ, with $\mathrm{dom}(\sigma) = \{x_1, \dots, x_n\}$, we have $\sigma \models \Gamma \quad\Rightarrow\quad \vdash (\!|\sigma(x_1)|\!) : \beta_1, \dots, \vdash (\!|\sigma(x_n)|\!) : \beta_n$.

We start with the cases concerning (i).

Case (ax)**:** Then, $x_1 : \beta_1, \dots, x_n : \beta_n \vdash x_i : \beta_i$, and $[\![x_i]\!]^{x_1,\dots,x_n} = \mathsf{Pr}^n_i$. By Lemma 4.2(i) we conclude $\vdash \mathsf{Pr}^n_i : \beta_1 \to \cdots \to \beta_n \to \beta_i$.

Case (0)**:** In this case, we have $\Gamma \vdash \mathbf{0} : \mathsf{int}$ and $[\![\mathbf{0}]\!]^{x_1,\dots,x_n} = \mathsf{Pr}^{n+1}_1 @ [0]$. By Lemma 4.2(i) $\vdash \mathsf{Pr}^{n+1}_1 @ : \mathsf{int} \to \beta_1 \to \cdots \to \beta_n \to \mathsf{int}$ and, by Figure 4 $\vdash \#^{-1}(0) : \mathsf{int}$. By Proposition 3.5(i), $\vdash \mathsf{Pr}^{n+1}_1 @ [0] : \beta_1 \to \cdots \to \beta_n \to \mathsf{int}$.

Case (Y)**:** In this case $\Gamma \vdash \mathbf{fix}\, M' : \alpha$ and $[\![\mathbf{fix}\, M']\!]^{\vec{x}} = \mathsf{Y}_n @ [\#[\![M']\!]^{\vec{x}}]$. 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 $\vdash [\![M']\!]^{x_1,\dots,x_n} : \vec{\beta} \to \alpha \to \alpha$. By Proposition 3.5(i), we conclude that $\vdash \mathsf{Y}_n @ [\#[\![M']\!]^{x_1,\dots,x_n}] : \beta_1 \to \cdots \to \beta_n \to \alpha$.

Case (+)**:** In this case $\Gamma \vdash \mathbf{succ}\, M' : \mathsf{int}$ since $\Gamma \vdash M' : \mathsf{int}$. By definition, $[\![\mathbf{succ}\, M']\!]^{x_1,\dots,x_n} = \mathsf{Succ}_n @ [\#[\![M']\!]^{x_1,\dots,x_n}]$. By Lemma 4.2(iv), $\vdash \mathsf{Succ}_n : (\vec{\beta} \to \mathsf{int}) \to \vec{\beta} \to \mathsf{int}$. From IH1, we get $\vdash [\![M']\!]^{x_1,\dots,x_n} : \vec{\beta} \to \mathsf{int}$, and thus by Proposition 3.5(i), we conclude $\vdash \mathsf{Succ}_n @ [\#[\![M']\!]^{x_1,\dots,x_n}] : \vec{\beta} \to \mathsf{int}$.

Case (−)**:** Analogous, applying Lemma 4.2(iv).

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 $[\![\mathbf{ifz}(L, N_2, N_2)]\!]^{\vec{x}} = \mathsf{Ifz}_n @ [\#[\![L]\!]^{\vec{x}}, \#[\![N_1]\!]^{\vec{x}}, \#[\![N_2]\!]^{\vec{x}}]$. By applying Lemma 4.2(v), 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 3.5(i), we conclude $\vdash \mathsf{Ifz}_n @ [\#[\![L]\!]^{\vec{x}}, \#[\![N_1]\!]^{\vec{x}}, \#[\![N_2]\!]^{\vec{x}}] : \vec{\beta} \to \alpha$.

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 $\Delta = y_1 : \delta_1, \dots, y_m : \delta_m$ such that $\sigma \models \Delta$ and $\Gamma, \Delta, z : \alpha_1 \vdash M' : \alpha_2$. Then $\sigma \models \Delta$ entails $\sigma = [y_1 \leftarrow (\rho_1, N_1), \dots, y_m \leftarrow (\rho_m, N_m)]$ for appropriate $\vec{\rho}, \vec{N}$. By definition, we have $[\![\lambda z.M'\langle\sigma\rangle]\!]^{x_1,\dots,x_n} = (\!|\sigma, M'|\!)^{\vec{x},z} = [\![M']\!]^{y_1,\dots,y_m,\vec{x},z} @ [\#(\!|\rho_1, N_1|\!), \dots, \#(\!|\rho_m, N_m|\!)]$. 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 3.5(i), we derive $\vdash [\![M']\!]^{\vec{y},\vec{x},z} @ [\#(\!|\sigma_1, N_1|\!), \dots, \#(\!|\sigma_m, N_m|\!)] : \vec{\beta} \to \alpha_1 \to \alpha_2$.

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, $[\![M_1 \cdot M_2]\!]^{\vec{x}} = \mathsf{Apply}_n @ [\#[\![M_1]\!]^{\vec{x}}, \#[\![M_2]\!]^{\vec{x}}]$. By Lemma 4.2(ii), 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 3.5(i), that $\vdash \mathsf{Apply}_n @ [\#[\![M_1]\!]^{\vec{x}}, \#[\![M_2]\!]^{\vec{x}}] : \vec{\beta} \to \alpha$.

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 $\Delta = y_1 : \delta_1, \dots, y_m : \delta_m$. By IH2 on $\sigma' \models \Gamma'$, we get $(\!|\sigma(x_i)|\!) = (\!|\sigma'(x_i)|\!) : \beta_i$, for all $i \in \{1, \dots, n-1\}$. We show $(\!|(\rho, N)|\!) : \beta_n$. By IH1, we get $\vdash [\![N]\!]^{y_1,\dots,y_m} : \delta_1 \to \cdots \to \delta_m \to \beta_n$. By applying IH2 on $\rho \models \Delta$, we get $(\!|\rho(y_j)|\!) : \delta_j$, for all $j \in \{1, \dots, m\}$. Since $(\!|\rho, N|\!) = [\![N]\!]^{\vec{y}} @ [\#(\!|\rho(y_1)|\!), \dots, \#(\!|\rho(y_m)|\!)]$, we conclude $\vdash (\!|\rho, N|\!) : \beta_n$, by Proposition 3.5(i).

□

Proof. [Proof of Theorem 4.7] We prove $\sigma \triangleright M \Downarrow_d V \Rightarrow (\!|\sigma, M|\!) \leftrightarrow_{\mathsf{c}} [\![V]\!]$ by induction on a derivation of $\sigma \triangleright M \Downarrow_d V$. We let $\mathrm{dom}(\sigma) = \{x_1, \dots, x_n\}$ and sometimes use the convenient notation $\#(\!|\sigma(\vec{x})|\!) = \#(\!|\sigma(x_1)|\!), \dots, \#(\!|\sigma(x_n)|\!)$.

Case (nat)**:** Then $M = V = \underline{k}$, for some $k \geq 0$. Recall that we assume $k = \#\mathsf{k}$. Then using Lemma 4.1(i), $(\!|\sigma, \underline{k}|\!) = [\![\underline{k}]\!]^{x_1,\dots,x_n} @ [\#(\!|\sigma(\vec{x})|\!)] = \mathsf{Pr}_1^{n+1} @ [k, \#(\!|\sigma(\vec{x})|\!)] \twoheadrightarrow_{\mathsf{c}} \#^{-1}(k)\,{}_{\mathsf{c}}\!\twoheadleftarrow \mathsf{Pr}_1^1 @ [k] = [\![\underline{k}]\!]$

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, $\mathrm{dom}(\rho) = \{y_1, \dots, y_m\}$. Then

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

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, using Lemma 4.1(i), we have $(\!|\sigma, x_i|\!) = [\![x_i]\!]^{x_1,\dots,x_n} @ [\#(\!|\sigma(\vec{x})|\!)] = \mathsf{Pr}_i^n @ [\#(\!|\sigma(\vec{x})|\!)] \twoheadrightarrow_{\mathsf{c}} (\!|\sigma(x_i)|\!) = (\!|\rho, N|\!)$ We conclude since, by IH, 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:

$$\begin{aligned}&(\!|\sigma, M_1 \cdot M_2|\!) \\ =&[\![M_1 \cdot M_2]\!]^{x_1,\dots,x_n} @ [\#(\!|\sigma(x_1)|\!), \dots, \#(\!|\sigma(x_n)|\!)] \\ =&\mathsf{Apply}_n @ [[\![M_1]\!]^{\vec{x}}, \#[\![M_2]\!]^{\vec{x}}, \#(\!|\sigma(x_1)|\!), \dots, \#(\!|\sigma(x_n)|\!)] \\ \twoheadrightarrow_{\mathsf{c}}&[\![M_1]\!]^{\vec{x}} @ [\#(\!|\sigma(\vec{x})|\!), \#([\![M_2]\!]^{\vec{x}} @ [\#(\!|\sigma(\vec{x})|\!)]\,)]\,, && \text{by Lemma 4.1(ii)}, \\ =&(\!|\sigma, M_1|\!) @ [\#(\!|\sigma, M_2|\!)]\end{aligned}$$

By IH 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 $\mathrm{dom}(\rho) = \{y_1, \dots, y_m\}$, we get

$$\begin{aligned}(\!|\sigma, M_1|\!) @ [\#(\!|\sigma, M_2|\!)] &\leftrightarrow_{\mathsf{c}} (\!|\rho, M_1'|\!)^z @ [\#(\!|\sigma, M_2|\!)]\,, && \text{by def. of } \leftrightarrow_{\mathsf{c}} \text{ and Lemma 2.11}, \\ &= [\![M_1']\!]^{y_1,\dots,y_m,z} @ [\#(\!|\rho(\vec{y})|\!), \#(\!|\sigma, M_2|\!)] \\ &= (\!|\rho + [z \leftarrow (\sigma, M_2)], M_1'|\!)\end{aligned}$$

By IH 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 IH 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:

$$\begin{aligned}
(\!|\sigma, \mathbf{pred}\, M'|\!) &= [\![\mathbf{pred}\, M']\!]^{x_1,\dots,x_n} @ [\#(\!|\sigma(x_1)|\!),\dots,\#(\!|\sigma(x_n)|\!)] \\
&= \mathsf{Pred}_n @ [\#[\![M']\!]^{x_1,\dots,x_n}, \#(\!|\sigma(x_1)|\!),\dots,\#(\!|\sigma(x_n)|\!)] \\
&\twoheadrightarrow_{\mathsf{c}} \langle R_0 = \#([\![M']\!]^{\vec{x}} @ [\#(\!|\sigma(\vec{x})|\!)]\,), \vec{R}, 0 \leftarrow \texttt{Pred}(0); \texttt{Call}\ 0, []\rangle, \text{ by Lemma 4.1(iii)}, \\
&= \langle R_0 = \#(\!|\sigma, M'|\!), \vec{R}, ; 0 \leftarrow \texttt{Pred}(0); \texttt{Call}\ 0, []\rangle \\
&\twoheadrightarrow_{\mathsf{c}} \langle R_0 = n+1, \vec{R}, 0 \leftarrow \texttt{Pred}(0); \texttt{Call}\ 0, []\rangle \twoheadrightarrow_{\mathsf{c}} \#^{-1}(n)
\end{aligned}$$

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 IH 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:

$$\begin{aligned}
(\!|\sigma, \mathbf{succ}\, M'|\!) &= [\![\mathbf{succ}\, M']\!]^{x_1,\dots,x_n} @ [\#(\!|\sigma(x_1)|\!),\dots,\#(\!|\sigma(x_n)|\!)] \\
&= \mathsf{Succ}_n @ [\#[\![M']\!]^{x_1,\dots,x_n}, \#(\!|\sigma(x_1)|\!),\dots,\#(\!|\sigma(x_n)|\!)] \\
&\twoheadrightarrow_{\mathsf{c}} \langle R_0 = \#([\![M']\!]^{\vec{x}} @ [\#(\!|\sigma(\vec{x})|\!)]\,), \vec{R}, 0 \leftarrow \texttt{Succ}(0); \texttt{Call}\ 0, []\rangle, \text{ by Lemma 4.1(iii)}, \\
&= \langle R_0 = \#(\!|\sigma, M'|\!), \vec{R}, ; 0 \leftarrow \texttt{Succ}(0); \texttt{Call}\ 0, []\rangle \\
&\twoheadrightarrow_{\mathsf{c}} \langle R_0 = n, \vec{R}, 0 \leftarrow \texttt{Succ}(0); \texttt{Call}\ 0, []\rangle \twoheadrightarrow_{\mathsf{c}} \qquad \#^{-1}(n+1)
\end{aligned}$$

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

$$\begin{aligned}
(\!|\sigma, \mathbf{ifz}(L, N_1, N_2)|\!) &= [\![\mathbf{ifz}(L, N_1, N_2)]\!]^{x_1,\dots,x_n} @ [\#(\!|\sigma(x_1)|\!),\dots,\#(\!|\sigma(x_n)|\!)] \\
&= \mathsf{Ifz}_n @ [\#[\![L]\!]^{x_1,\dots,x_n}, \#[\![N_1]\!]^{x_1,\dots,x_n}, \#[\![N_2]\!]^{x_1,\dots,x_n}, \#(\!|\sigma(x_1)|\!),\dots,\#(\!|\sigma(x_n)|\!)] \\
&\twoheadrightarrow_{\mathsf{c}} \left\langle R_0 = \#(\!|\sigma, L|\!), R_1 = \#(\!|\sigma, N_1|\!), R_2 = \#(\!|\sigma, N_2|\!), \vec{R}, 0 \leftarrow \texttt{Test}(0,\ 1,\ 2); \texttt{Call}\ 0, []\right\rangle, \\
&\qquad \text{by Lemma 4.1(v)}, \\
&\twoheadrightarrow_{\mathsf{c}} \left\langle R_0 = n+1, R_1 = \#(\!|\sigma, N_1|\!), R_2 = \#(\!|\sigma, N_2|\!), \vec{R}, 0 \leftarrow \texttt{Test}(0,\ 1,\ 2); \texttt{Call}\ 0, []\right\rangle \\
&\twoheadrightarrow_{\mathsf{c}} (\!|\sigma, N_2|\!)
\end{aligned}$$

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

$$\begin{aligned}
(\!|\sigma, \mathbf{ifz}(L, N_1, N_2)|\!) &= [\![\mathbf{ifz}(L, N_1, N_2)]\!]^{x_1,\dots,x_n} @ [\#(\!|\sigma(x_1)|\!),\dots,\#(\!|\sigma(x_n)|\!)] \\
&= \mathsf{Ifz}_n @ [\#[\![L]\!]^{x_1,\dots,x_n}, \#[\![N_1]\!]^{x_1,\dots,x_n}, \#[\![N_2]\!]^{x_1,\dots,x_n}, \#(\!|\sigma(x_1)|\!),\dots,\#(\!|\sigma(x_n)|\!)] \\
&\twoheadrightarrow_{\mathsf{c}} \left\langle R_0 = \#(\!|\sigma, L|\!), R_1 = \#(\!|\sigma, N_1|\!), R_2 = \#(\!|\sigma, N_2|\!), \vec{R}, 0 \leftarrow \texttt{Test}(0,\ 1,\ 2); \texttt{Call}\ 0, []\right\rangle, \\
&\qquad \text{by Lemma 4.1(v)}, \\
&\twoheadrightarrow_{\mathsf{c}} \left\langle R_0 = 0, R_1 = \#(\!|\sigma, N_1|\!), R_2 = \#(\!|\sigma, N_2|\!), \vec{R}, 0 \leftarrow \texttt{Test}(0,\ 1,\ 2); \texttt{Call}\ 0, []\right\rangle \\
&\twoheadrightarrow_{\mathsf{c}} (\!|\sigma, N_1|\!)
\end{aligned}$$

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

Case (fix)**:** Then $M = \mathbf{fix}\, M'$ and $\sigma \rhd M' \cdot (\mathbf{fix}\, M')$. Then

$$\begin{array}{rcll} (\!|\sigma, \mathbf{fix}\, M'|\!) & = & [\![\mathbf{fix}\, M']\!]^{x_1,\dots,x_n} @ [\#(\!|\sigma(x_1)|\!), \dots, \#(\!|\sigma(x_n)|\!)] & \\ & = & \mathsf{Y}_n @ [\#[\![M']\!]^{x_1,\dots,x_n}, \#(\!|\sigma(x_1)|\!), \dots, \#(\!|\sigma(x_n)|\!)] & \\ & \twoheadrightarrow_{\mathsf{c}} & [\![M']\!]^{\vec{x}} @ [\#(\!|\sigma(\vec{x})|\!), \#(\mathsf{Y}_n @ [\#[\![M']\!]^{\vec{x}}, \#(\!|\sigma(\vec{x})|\!)]\,)] & \\ & = & [\![M']\!]^{\vec{x}} @ [\#(\!|\sigma(\vec{x})|\!), \#([\![\mathbf{fix}\, M']\!]^{\vec{x}} @ [\#(\!|\sigma(\vec{x})|\!)]\,)]\,, & \text{by Lemma 4.1(ii)}, \\ & {}_{\mathsf{c}}\!\twoheadleftarrow & \mathsf{Apply}_n @ [\#[\![M']\!]^{\vec{x}}, \#[\![\mathbf{fix}\, M']\!]^{\vec{x}}, \#(\!|\sigma(x_1)|\!), \dots, \#(\!|\sigma(x_n)|\!)] & \\ & = & [\![M' \cdot (\mathbf{fix}\, M')]\!]^{x_1,\dots,x_n} @ [\#(\!|\sigma(x_1)|\!), \dots, \#(\!|\sigma(x_n)|\!)] & \\ & = & (\!|\sigma, M' \cdot (\mathbf{fix}\, M'|\!). & \end{array}$$

This concludes the proof. ☐

Proof. [Proof of Theorem 4.13(i)] For an EPCF term M, an EPCF value V and an explicit substitution σ, we show $\sigma \rhd M \Downarrow_d V \Rightarrow (\sigma, M)^* \Downarrow ([\,], V)^*$ by induction on a derivation of $\sigma \rhd M \Downarrow_d V$.

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 $\rho \rhd N \Downarrow_d V$. By IH $(\rho, N)^* \Downarrow ([], V)^*$. We conclude since $(\sigma, M)^* = (\rho, N)^*$.

Case (β_v)**:** In this case $M = M_1 \cdot M_2$ with $\sigma \rhd M_1 \Downarrow_d \lambda x.M_1'\langle\rho\rangle$, wlog $x \notin \mathrm{dom}(\rho + \sigma)$, and $\rho + [x \leftarrow (\sigma, M_2)] \rhd M_1' \Downarrow_d V$. By IH 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 4.12(i) $(\rho + [x \leftarrow (\sigma, M_2)], M_1')^* = (\rho, M_1')^*[(\sigma, M_2)^*/x]$. 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 $\sigma \rhd N \cdot (\mathbf{fix}\, N) \Downarrow_d V$. From the IH 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 IH. ☐

Proof. [Proof of Theorem 4.13(ii)] For a PCF program P and PCF value U, we show that $P \Downarrow U$ entails:

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

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 IH 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 IH, 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 4.12(ii), 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 IH 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 IH. □