

Program Equivalence in Linear Contexts

Yuxin Deng^{1,*}

Department of Computer Science and Engineering, Shanghai Jiao Tong University, China

Yu Zhang²

*State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences, Beijing, China*

Abstract

Program equivalence in linear contexts, where programs are used or executed *exactly once*, is an important issue in programming languages. However, existing techniques like those based on bisimulations and logical relations only target at contextual equivalence in the usual (non-linear) functional languages, and fail in capturing non-trivial equivalent programs in linear contexts, particularly when non-determinism is present.

We propose the notion of *linear contextual equivalence* to formally characterize such program equivalence, as well as a novel and general approach to studying it in higher-order languages, based on labeled transition systems specifically designed for functional languages. We show that linear contextual equivalence indeed coincides with trace equivalence. We illustrate our technique in both deterministic (a linear version of PCF) and non-deterministic (linear PCF in Moggi's framework) functional languages.

Keywords: linear PCF, contextual equivalence, trace equivalence, non-determinism

1. Introduction

Contextual equivalence is an important concept in programming languages and can be used to formalize and reason about many interesting properties of computing systems. For functional languages, there are many techniques that can help to prove contextual equivalence. Among others, applicative bisimulations [1, 19] and logical relations [31, 34] are particularly successful.

On the other side, linear logic (and its term correspondence often known as linear λ -calculus) has seen significant applications in computer science ever since its birth,

*Corresponding author

Email addresses: deng-yx@cs.sjtu.edu.cn (Yuxin Deng), yzhang@ios.ac.cn (Yu Zhang)

¹Partially supported by the National Natural Science Foundation of China (61173033, 61033002, 61261130589) and ANR 12IS02001 "PACE".

²Partially supported by the National Natural Science Foundation of China (61170073, 61161130530).

due to its native mechanism of describing restricted use of resources. For example, the linear λ -calculus provides the core of a functional programming language with an expressive type system, in which statements like “this resource will be used exactly once” can be formally expressed and checked. Such properties become useful when introducing imperative concepts into functional programming [17], structural complexity theory [18], or analyzing memory allocation [36]. Moreover, the linear λ -calculus, when equipped with dependent types, can serve as a representation language within a logical framework, a general meta-language for the formalization of deductive systems [8]. The study of linearity in concurrent languages such as the π -calculus allows for a fine-grained analysis of process behavior [22, 38, 5].

Introducing linearity also leads to novel observation over program equivalences. In particular, if we consider a special sort of contexts where candidate programs must be used linearly (we call these contexts *linear contexts*), program equivalence with respect to these contexts should be a coarser relation than the usual notion of contextual equivalence, especially when non-determinism is present. For instance, take Moggi’s language for non-determinism [25], where we have a primitive \sqcap for non-deterministic choice (same as the internal choice in CSP [15]), and consider the following two functions:

$$\begin{aligned} f_1 &\stackrel{\text{def}}{=} \text{val}(\lambda x . \text{val}(0) \sqcap \text{val}(1)) \\ f_2 &\stackrel{\text{def}}{=} \text{val}(\lambda x . \text{val}(0)) \sqcap \text{val}(\lambda x . \text{val}(1)). \end{aligned} \tag{1}$$

Existing techniques such as bisimulation or logical relations distinguish these two functions. In fact, it is easy to show that they are not equivalent in arbitrary contexts, by considering, e.g., the context

$$\text{bind } f = [_] \text{ in bind } x = f(0) \text{ in bind } y = f(0) \text{ in val}(x = y).$$

The context makes a double evaluation of the function by applying it to concrete arguments (noticing that Moggi’s language enforces a call-by-value evaluation of non-deterministic computations): with the first function f_1 , the two evaluations of $f(0)$ can return different values since the non-deterministic choice is inside the function body; with the second function f_2 , the non-deterministic choice is made before both evaluations of $f(0)$ and computation inside the function is deterministic, so the two evaluations always return the same value. But if we consider only linear contexts, where programs will be evaluated *exactly once*, then the two functions must be equivalent. However, no existing technique, at least to the best of our knowledge, can characterize such an equivalence relation with respect to linear contexts.

1.1. Potential application in computational cryptography

The motivation of the work first comes from the second author’s work on building a logic (namely CSLR) for reasoning about *computational indistinguishability*, which is an essential concept in complexity-theoretic cryptography and helps to define many important security criteria [39, 11]. The CSLR logic is based on a functional language which characterizes probabilistic polynomial-time computations by typing, where linearity plays an important role.

Let us consider an example. The *semantic security* is a fundamental property of encryption schemes and in CSLR we can define it as

$$\lambda\eta . \lambda m_0 . \lambda m_1 . \mathbf{Enc}(\eta, m_0, pk) \simeq_C \lambda\eta . \lambda m_0 . \lambda m_1 . \mathbf{Enc}(\eta, m_1, pk) \quad (2)$$

where \mathbf{Enc} is the (probabilistic) encryption function, pk is a public key known to adversaries, and η is the security parameter. The equivalence relation \simeq_C is what we call *computationally indistinguishability* in cryptography and the detailed definition can be found in [39, Definition 1]. Intuitively, it says that no feasible adversary can distinguish the related programs with non-negligible probabilities (w.r.t. the programs' security parameter). Formula (2) formally states the indistinguishability between two encryption oracles: an adversary can only do encryption (with the key pk) by calling the oracles, who will ask the adversary to supply a pair of messages at his/her choice and return the encryption of one of them — the left oracle always returns the cipher-text of the first message and the right one returns the cipher-text of the second. Computational indistinguishability between the two oracles says that no feasible adversary, when calling one of the two oracle, can answer which message is encrypted with a significant winning probability.

Computational indistinguishability is indeed a notion of observational equivalence, where adversaries must be computable in polynomial-time on probabilistic Turing machines. Complexity is well manipulated in the CSLR type system, where linearity particularly plays the essential role of controlling complexity in higher-order functions. The original definition indicates that the two oracles as defined in (2) can be called only for a polynomial number of times, however, Theorem 5.2.11 in [12] shows that the definition where an adversary only submits a single pair of messages to the oracle, is equivalent to the multi-messages version. It suggests that we can effectively replace the computational indistinguishability by the notion of linear contextual equivalence in (2) and consider an adversary who calls the oracle *only once*.

In fact, we believe that linear contextual equivalence can be used to define security criteria with *non-adaptive* adversaries. Non-adaptive adversaries send multiple messages to oracles but choose messages independently of the oracles' responses, so we can actually define the oracles as functions receiving a list of messages. Adversaries will also send their messages to the oracle *all at once* and in that case they call the oracle *only once*.

Although the language of the CSLR logic is probabilistic, a general proof technique of linear contextual equivalence is missing from the literature, particularly in a non-deterministic setting where there exist programs that are equivalent in linear contexts but not in general, as we described previously.

1.2. Related work

Program equivalence with respect to *non-linear* contexts has been widely investigated. Logical relations are one of the powerful tools for proving contextual equivalence in typed lambda-calculi, in both operational [28, 29, 7] and denotational settings [31, 24, 14]. They are defined by induction on types, hence are relatively easy to use. But it is known that completeness of (strict) logical relations are often hard to

achieve, especially for higher-order types. It is even worse for monadic types, particularly when non-determinism is present [23].

Characterizations in terms of simulation relations have been studied in functional languages [19, 13, 27, 21], as well as languages with linear type systems [6, 9]. Due to the higher-order features of the languages, it is difficult to directly prove the precongruence property of similarity. A common feature crucial to this line of research is then to follow Howe’s approach [19], which requires to first define a precongruence candidate, a precongruence relation by construction, and then to show the coincidence of that relation with simulation. An alternative approach, such as environmental bisimulation proposed in [33], has a built-in congruence property, but then the definition itself has sophisticated conditions.

1.3. Contribution

In this paper we consider contextual equivalence with respect to *linear* contexts only. Our approach is developed in a linear version of PCF and we propose a formal definition of the so-called *linear contextual equivalence*, which characterizes the notion of equivalence when programs are used only once. Clearly linear contextual equivalence is coarser than the usual contextual equivalence that is insensitive to the number of times that programs are executed. This new notion of equivalence might give rise to a refined analysis of language-based security as we discussed above. We give a sound and complete characterization of the linear contextual equivalence in terms of *trace equivalence*³, based on appropriate labeled transition semantics for terms. In order to show the congruence property of trace equivalence, we exploit the internal structure of linear contexts, instead of relying on Howe’s approach.

While term transitions are a relatively standard concept, the notion of *context transitions* that we have introduced in the development is novel. It models the interactions between programs and contexts and may have potential use in game semantics [2, 20]. We also notice that such context transitions (along with program transitions) conforms to the idea of rely-guarantee reasoning, which has been successfully applied in the verification of concurrent programs [37, 35, 10], and may suggest an alternative approach.

Although the entire development is based on an operational treatment, the technique is general enough to be adapted in other languages with linear type systems. Indeed, we show that our approach can be applied in a non-deterministic extension of the linear PCF based on Moggi’s framework with monadic types, where trace equivalence also serves as a sound and complete characterization of linear contextual equivalence. The result particularly helps us to prove the equivalence of the two functions in the previous example, as we can show that they are trace equivalent.

One can probably employ Howe’s approach when proving linear contextual equivalence in a deterministic language. While Howe’s approach applies to a wider variety of occasions, it is more involved; our approach is much simpler because we take advantage of linearity in resource usage. Furthermore, in non-deterministic languages,

³Note that bisimulation equivalence is not used because in the presence of non-determinism in our language (Section 4) bisimulation equivalence is strictly finer than trace equivalence. If non-determinism is disallowed then the two equivalences coincide and thus the technique based on traces can be turned into a bisimulation technique.

simulation based program equivalences are usually finer than linear contextual equivalence, thus they fail to give a complete characterization of the latter.

1.4. Outline

The rest of the paper is organized as follows: Section 2 defines briefly a linear version of call-by-name PCF with a dual type system, as well as its operational semantics. In particular, a labeled transition system for the language is presented and the notion of trace equivalence is defined. In Section 3 we introduce the notion of linear contextual equivalence and show that trace equivalence in linear PCF coincides with linear contextual equivalence. Section 4 extends our approach in a non-deterministic setting with monadic types, where technical development follows the previous two sections, and we establish the coincidence between trace equivalence and linear contextual equivalence. With this result, we show that the two functions in (1) are indeed equivalent in linear contexts. Section 5 concludes the paper.

2. The call-by-name linear PCF

We start with a linear version of PCF (LPCF for short) with a call-by-name evaluation strategy. Types are given by the following grammar:

$$\tau, \tau', \dots ::= \text{Nat} \mid \text{Bool} \mid \tau \& \tau' \mid \tau \otimes \tau' \mid \tau \multimap \tau' \mid \tau \rightarrow \tau'$$

Here $\tau \& \tau'$ and $\tau \otimes \tau'$ are usual product and tensor product respectively. Linear functions will be given types in the form $\tau \multimap \tau'$. Following [32], we choose to make intuitionistic function types $\tau \rightarrow \tau'$ primitive rather than introducing exponential types. The choice makes our technical development simpler but does not affect the heart of the approach — one can certainly express non-linear function types in terms of !-types, using Girard's decomposition: $\tau \rightarrow \tau' = !\tau \multimap \tau'$, and adapt our technique accordingly.

Terms are built up from constants (boolean and integer values plus integer operations and fix-point recursion) and variables, using the following constructs.

$e, e', \dots ::=$	x	Variables
	$0 \mid 1 \mid 2 \mid \dots$	Integers
	$\text{succ} \mid \text{pred} \mid \text{iszero}$	Integer operations
	$\lambda x. e \mid e e'$	Abstractions and applications
	$\text{true} \mid \text{false}$	Booleans
	$\text{if } e_1 \text{ then } e_2 \text{ else } e_3$	Conditionals
	$\langle e_1, e_2 \rangle \mid \text{proj}_i(e)$	Products and projections
	fix_τ	Fix-point recursions
	$e_1 \otimes e_2 \mid \text{let } x \otimes y = e \text{ in } e'$	Tensor products and projections

Most of the language constructs are standard: the λ -abstraction $\lambda x. e$ defines a function, whose linearity will be judged by the type system, and the application $e e'$ applies the function e to the argument e' ; the conditional $\text{if } e_1 \text{ then } e_2 \text{ else } e_3$ evaluates like e_2 or e_3 , according to whether the boolean term e_1 evaluates to `true` or `false`;

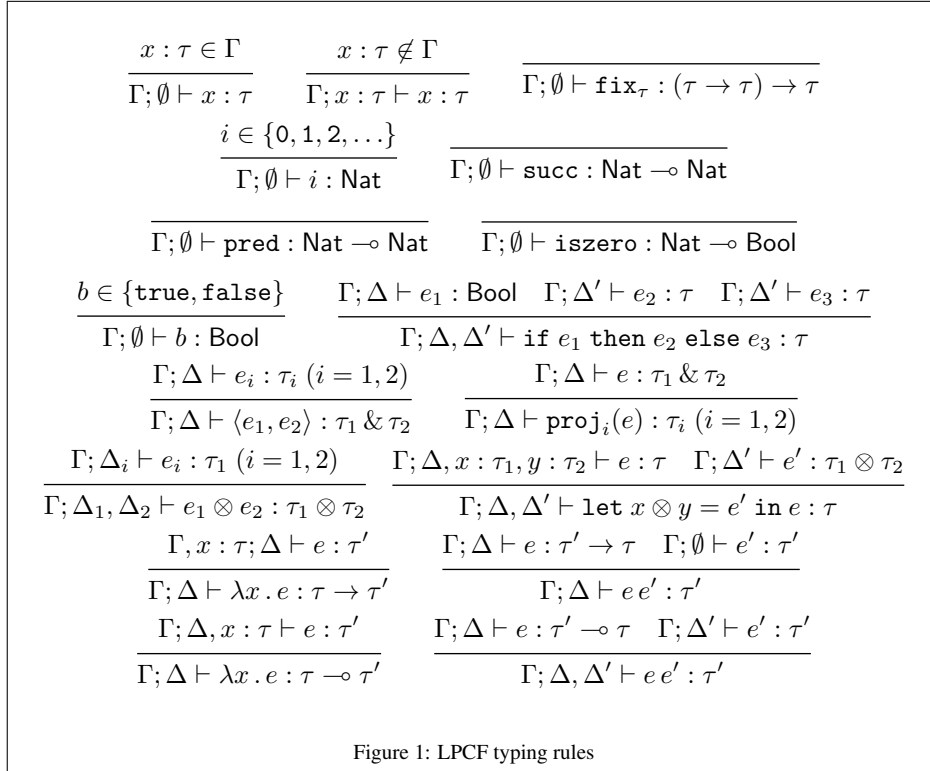


Figure 1: LPCF typing rules

$\langle e_1, e_2 \rangle$, $\text{proj}_1 e$ and $\text{proj}_2 e$ are normal products and corresponding projections; the term $\text{fix}_\tau e$ represents the least fix-point of the function e .

The tensor product and projection constructs are related to linearity — they differ from the normal products and projections in the fact that a tensor projection allows to consume both components of a tensor product without violating its linearity. Consider a program which applies a function f to both components of a product p . With normal products, we can only define it as $f(\text{proj}_1 p)(\text{proj}_2 p)$, where p has to be used at least *twice*; with a tensor product, the program can be defined as $\text{let } x \otimes y = p \text{ in } f x y$, where p is used *linearly*.

Variables appearing in the λ -binder and the let -binder (in tensor projections) are bound variables of LPCF programs. We write $FV(e)$, $FLV(e)$, $FNV(e)$ for the sets of, respectively, free variables, free linear variables, and free non-linear variables in term e . We will not distinguish α -equivalent terms, which are terms syntactically identical up to renaming of bound variables. If e and e' are terms and x is a variable, then $e[e'/x]$ denotes the term resulting from substituting e' for all free occurrences of x in e . More generally, given a list e_1, \dots, e_n of terms and a list x_1, \dots, x_n of distinct variables, we write $e[e_1/x_1, \dots, e_n/x_n]$ for the result of simultaneously substituting each term e_i for free occurrences in e of the corresponding variable x_i .

A *typing assertion* takes the form $\Gamma; \Delta \vdash e : \tau$, where Γ and Δ are finite partial functions from variables to types, e is a term, and τ is a type. We adopt the notation

from dual intuitionistic linear logic [3] by using Γ and Δ to represent typing environments for, respectively, non-linear variables and linear variables. It is assumed that the domains of the non-linear and linear typing environments are disjoint. The *type assignment relation* for the linear PCF consists of all typing assertions that can be derived from the axioms and rules in Figure 1, which are very standard. The notation $\Delta, x : \tau$ denotes the partial function which properly extends Δ by mapping x to τ , so it is implicitly assumed that x is not in the domain of Δ . We extend the convention for the extension of environments to the notation Δ, Δ' . We write $\mathcal{P}rog(\tau) = \{e \mid \emptyset; \emptyset \vdash e : \tau\}$ for the set of all closed programs of type τ .

The type system also distinguishes between the two forms of products: in the previous example, the typing assertion for the program with normal products is

$$p : \tau_1 \& \tau_2; f : \tau_1 \multimap \tau_2 \multimap \tau \vdash f(\text{proj}_1 p)(\text{proj}_2 p) : \tau,$$

and the typing assertion for the program with a tensor product is

$$\emptyset; p : \tau_1 \otimes \tau_2, f : \tau_1 \multimap \tau_2 \multimap \tau \vdash \text{let } x \otimes y = p \text{ in } f x y : \tau.$$

In the former case, if the type of f is $\tau_1 \rightarrow \tau_2 \rightarrow \tau$, the program still type-checks, but p must be a non-linear variable; in the latter case, p can be either linear or non-linear, but f must be a linear function. Tensor products are useful for linear function currying/uncurrying: if a linear function has type $\tau_1 \multimap \tau_2 \multimap \tau$, its uncurried function will have type $\tau_1 \otimes \tau_2 \multimap \tau$.

2.1. The operational semantics

We first define the notion of *values* of LPCF.

$$\begin{aligned} v, v', \dots ::= & \text{succ} \mid \text{pred} \mid \text{iszero} \mid \text{true} \mid \text{false} \mid 0 \mid 1 \mid 2 \mid \dots \\ & \mid \text{fix}_\tau \mid \langle e, e' \rangle \mid e \otimes e' \mid \lambda x. e \end{aligned}$$

These are also *canonical forms* of LPCF terms.

The one-step reduction \rightsquigarrow between terms is the smallest relation given by the basic reductions axioms

$$\begin{aligned} (\lambda x. e)e' &\rightsquigarrow e[e'/x] \\ \text{fix}_\tau e &\rightsquigarrow e(\text{fix}_\tau e) \\ \text{succ } n &\rightsquigarrow n + 1, \text{ where } n \in \{0, 1, 2, \dots\} \\ \text{pred } 0 &\rightsquigarrow 0 \\ \text{pred } n &\rightsquigarrow n - 1, \text{ where } n \in \{1, 2, \dots\} \\ \text{iszero } 0 &\rightsquigarrow \text{true} \\ \text{iszero } n &\rightsquigarrow \text{false}, \text{ where } n \in \{1, 2, \dots\} \\ \text{if true then } e_1 \text{ else } e_2 &\rightsquigarrow e_1 \\ \text{if false then } e_1 \text{ else } e_2 &\rightsquigarrow e_2 \\ \text{proj}_i \langle e_1, e_2 \rangle &\rightsquigarrow e_i, (i = 1, 2) \\ \text{let } x \otimes y = e_1 \otimes e_2 \text{ in } e &\rightsquigarrow e[e_1/x, e_2/y] \end{aligned}$$

together with the structural rule

$$\frac{e_1 \rightsquigarrow e_2}{\mathcal{E}[e_1] \rightsquigarrow \mathcal{E}[e_2]}$$

where \mathcal{E} is the evaluation context generated by the grammar

$$\begin{aligned} \mathcal{E} ::= & [] \mid \text{succ}(\mathcal{E}) \mid \text{pred}(\mathcal{E}) \mid \text{iszero}(\mathcal{E}) \mid \mathcal{E} e \mid \text{if } \mathcal{E} \text{ then } e_1 \text{ else } e_2 \\ & \mid \text{proj}_i(\mathcal{E}) \mid \text{let } x \otimes y = \mathcal{E} \text{ in } e \end{aligned}$$

We often call a term $\mathcal{E}[x]$ an evaluation context, if x is the *only* free variable of the term.

The operational semantics that we define for LPCF is essentially a *call-by-name* evaluation. Although our later development depends on the operational semantics, it does not really matter whether the evaluation strategy is call-by-name or call-by-value — one can easily adapt our approach to a call-by-value semantics. The only crucial point is that we should not allow the following forms of evaluation contexts:

$$\langle \mathcal{E}, e \rangle, \langle e, \mathcal{E} \rangle, \text{if } e \text{ then } \mathcal{E} \text{ else } e', \text{if } e \text{ then } e' \text{ else } \mathcal{E}.$$

This is because these contexts adopt syntactically duplicated linear variables without breaking the linearity restriction; if we substitute a reducible term for such a variable, multiple copies of the term can be generated in the context, then one of them may be reduced while all other copies remain unchanged. For instance, $\langle x, x \rangle$ is a well typed program with x being a linear variable, i.e., $\emptyset; x : \tau \vdash \langle x, x \rangle : \tau \& \tau$. Substituting x with an arbitrary closed term, e.g. $\text{pred } 1$, still produces a well-typed program: $\langle x, x \rangle[\text{pred } 1/x] \equiv \langle \text{pred } 1, \text{pred } 1 \rangle$. However, adopting $\langle \mathcal{E}, e \rangle$ and $\langle e, \mathcal{E} \rangle$ as evaluation contexts will allow us to reduce the term $\text{pred } 1$ twice, which contradicts the semantics of linearity because any substitute for a linear variable is assumed to be used and *computed* only once. The restriction on evaluation contexts is important for our technical development in the paper. It has also appeared in the literature. For example, both [13] and [27] require that evaluation does not continue within the components of a pair.

It is clear that LPCF terms in canonical form do not reduce. The following proposition also shows that every closed non-reducible term must be in canonical form. We write $e \not\rightsquigarrow$ when there does not exist a term e' such that $e \rightsquigarrow e'$, and \rightsquigarrow^* denotes the reflexive transitive closure of \rightsquigarrow .

Proposition 1. *Let e be a closed and well-typed term. If $e \not\rightsquigarrow$, then e must be in canonical form.*

PROOF . We prove by induction on the structure of e . As an example, we analyze one case for non-canonical forms:

- $e \equiv e' e''$. Here e' must be closed and not reducible, and by induction, must be canonical: if e' is an abstraction or a fix-point, then the whole term can be reduced; if $e' \in \{\text{succ}, \text{pred}, \text{iszero}\}$, then e'' must be canonical, which will be an integer, hence the whole term can be reduced too.

The other cases are similar, thus omitted. □

Evaluation in LPCF is deterministic and preserves typing.

Lemma 1. *For every well-typed term e , if $e \rightsquigarrow e'$, then $FLV(e') = FLV(e)$.*

$$\begin{array}{c}
\frac{c \in \{\text{true}, \text{false}, 0, 1, 2, \dots\}}{c \xrightarrow{c} \Omega} \quad \frac{\Gamma; \Delta \vdash \langle e_1, e_2 \rangle : \tau_1 \& \tau_2}{\langle e_1, e_2 \rangle \xrightarrow{\text{proj}_i} e_i} \\
\frac{\Gamma; \Delta \vdash \lambda x . e : \tau \quad \emptyset; \emptyset \vdash e' : \tau' \quad \tau \equiv \tau' \multimap \tau'' \text{ or } \tau' \rightarrow \tau''}{\lambda x . e \xrightarrow{\text{@}e'} e[e'/x]} \\
\frac{\Gamma; \Delta \vdash e_1 \otimes e_2 : \tau_1 \otimes \tau_2 \quad \emptyset; x : \tau_1, y : \tau_2 \vdash e : \tau}{e_1 \otimes e_2 \xrightarrow{\otimes e} e[e_1/x, e_2/y]} \\
\frac{e \rightsquigarrow^* e'' \quad e'' \xrightarrow{\alpha} e'}{e \xRightarrow{\alpha} e'}
\end{array}$$

Figure 2: Labeled transition system for LPCF

PROOF . By rule induction on the derivation of $e \rightsquigarrow e'$. □

Proposition 2 (Subject reduction). *If $\Gamma; \Delta \vdash e : \tau$ and $e \rightsquigarrow e'$, then $\Gamma; \Delta \vdash e' : \tau$.*

PROOF . A routine exercise. □

Proposition 3 (Determinacy). *1. If $e \rightsquigarrow^* v$ and $e \rightsquigarrow^* v'$ for some values v and v' , then $v = v'$.*

2. Every well-typed term either converges or all of its reduction do not terminate.

PROOF . It suffices to prove that the evaluation is deterministic, that is, there is at most one reduction rule that applies in any situation. This can be proved by induction on the structure of terms. □

Because the reduction is deterministic in LPCF, for any closed term e , we say e *converges* and write $e \Downarrow$ if it reduces to a value. Conversely, we say e *diverges* if the reduction of e does not terminate and we write $e \Uparrow$. We also define a specific class of terms $\Omega_\tau \stackrel{\text{def}}{=} \text{fix}_\tau(\lambda x . x)$, to represent non-terminating programs.

2.2. A labeled transition system for LPCF

In [13], Gordon defines an explicit labeled transition system in order to illustrate the applicative bisimulation technique in PCF. We follow this idea to define a labeled transition system for LPCF, upon which we can define the notions of traces and trace equivalence and develop our framework.

Transition rules are listed in Figure 2: we make the typing of terms explicit in the rules as the type system plays an important role in LPCF. It should be noticed that transitions are defined in general for LPCF terms, including open terms, but they never introduce new free variables. This is particularly true for @- and \otimes -transitions according to their typing premises.

The last rule in Figure 2 says that term reductions are considered as internal transitions — external transitions are labeled by *actions*. In the sequel, we write $e \xrightarrow{\alpha} e'$

for a single external transition without preceding internal transitions, and make internal transitions explicit when writing $e \rightsquigarrow \dots \rightsquigarrow \xrightarrow{\alpha} e'$.

Intuitively, external transitions represent the way terms interact with environments (or contexts). For instance, a λ -abstraction can “consume” (application of itself to) a term, which is supplied by the environment as an argument, and forms a β -reduction. The first rule says that, what an integer or boolean constant can provide to the environment is the value of itself, and after that it can no more provide any information, hence no external transitions can occur any more. We represent this by a transition, labeled by the value of the constant, into a non-terminating program Ω of appropriate type.

Let s be a finite sequence of actions $\alpha_1 \alpha_2 \dots \alpha_n$ ($n \geq 0$). We write $e \xrightarrow{s}$ if there exist terms e_1, e_2, \dots, e_n such that $e \rightsquigarrow^* \xrightarrow{\alpha_1} e_1 \rightsquigarrow^* \xrightarrow{\alpha_2} e_2 \dots \rightsquigarrow^* \xrightarrow{\alpha_n} e_n$ (the entire sequence including term reductions is called the *full sequence* of s). We also write $\alpha \cdot s$ and $s_1 \cdot s_2$ for the traces obtained by, respectively, prefixing trace s with an action α and concatenating s_1 and s_2 .

Definition 1 (Trace). A trace of LPCF term e is an action sequence s such that $e \xrightarrow{s}$. $\mathcal{Tr}(e)$ for the set of all traces of e , i.e., $\mathcal{Tr}(e) \stackrel{\text{def}}{=} \{s \mid e \xrightarrow{s}\}$. Given two traces s_1 and s_2 , we say s_1 is a subtrace of s_2 if s_1 is a prefix of s_2 when they are viewed as strings. A trace of a LPCF term e is maximal if it is not a subtrace of any other trace of $\mathcal{Tr}(e)$. A computational trace is a maximal trace of the form $s \cdot c$, where c is a boolean or integer constant.

In other words, a computational trace ends with some observable value, while a non-computational trace may end with an action in the forms $@e$, proj_i or $\otimes e$.

The empty trace, denoted by ϵ , can be taken by any program. Meanwhile, if ϵ is the only trace that a term can take, which means the term cannot take any external action, then the term must diverge, i.e., $\mathcal{Tr}(e) = \{\epsilon\}$ iff $e \uparrow$, for any well-typed term e .

Definition 2 (Trace preorder). The trace preorder \sqsubseteq^T between LPCF terms is defined as: $e_1 \sqsubseteq^T e_2$ iff $\mathcal{Tr}(e_1) \subseteq \mathcal{Tr}(e_2)$. Two terms e_1 and e_2 are trace equivalent, written $e_1 \simeq^T e_2$, iff $e_1 \sqsubseteq^T e_2$ and $e_2 \sqsubseteq^T e_1$.

- Lemma 2.**
1. If $\mathcal{Tr}(e_1) = \mathcal{Tr}(e_2) \neq \{\epsilon\}$, then e_1, e_2 must have the same type.
 2. Let e_1, e_2 be two terms of the same type. For any trace s , if $e_1 \xrightarrow{s} e'_1$ and $e_2 \xrightarrow{s} e'_2$, then e'_1 and e'_2 also have the same type.
 3. If $e \rightsquigarrow e'$, then $e' \sqsubseteq^T e$.

PROOF . The first statement can be proved by contradiction; the second one is proved by induction on the length of s ; the third one is a direct consequence of the definition of trace preorder. \square

Note that for LPCF terms the reduction relation \rightsquigarrow is deterministic, so we can strengthen the third clause of Lemma 2 as follows: If $e \rightsquigarrow e'$, then $e' \simeq^T e$. Of course this would not be possible when non-determinism is present, as we will see later on in Section 4.

3. Linear contextual equivalence

Defining a context in a language with linear types must be treated carefully, since holes can hide bound variables and consequently break the typing if the variable is linear [6]. We choose to replace the context hole by an explicit free variable and restrict attention to equivalence between closed terms, so as to avoid extra syntactic machinery.

Intuitively, a linear context is a context where programs under examination will be evaluated and used *exactly once*⁴. In a linear functional language, we can formalize it by a restricted notion of contexts: a *linear context* $\mathcal{C}_{x:\tau}$ in LPCF is a term with a single linear variable x and no non-linear variables, i.e., $\emptyset; x : \tau \vdash \mathcal{C}_{x:\tau} : \sigma$. We often omit the variable and type subscription when it is clear from the text or irrelevant.

Definition 3 (Linear contextual equivalence). We write $e_1 \sqsubseteq^C e_2$ for $e_1, e_2 \in \text{Prog}(\tau)$, if $\mathcal{C}[e_1/x] \Downarrow$ implies $\mathcal{C}[e_2/x] \Downarrow$ for all linear context $\mathcal{C}_{x:\tau}$. The relation \sqsubseteq^C is called the linear contextual preorder between closed programs. Linear contextual equivalence \simeq^C is defined as the symmetrization of \sqsubseteq^C : $e_1 \simeq^C e_2$ iff $e_1 \sqsubseteq^C e_2$ and $e_2 \sqsubseteq^C e_1$.

In [7], the definition of *ground contextual equivalence* (Definition 2.1) says that contexts must be of exponential types, because they are necessary for a program to adopt recursions in the type system presented in that work. In LPCF non-linear function types are primitive, with no exponential types, and the type for fix-point operator indicates that recursions must be taken within non-linear functions. Hence, the above definition admits the requirement of the definition of ground contextual equivalence in [7].

Lemma 3. Let $\mathcal{C}_1, \mathcal{C}_2$ be two linear contexts such that $\emptyset; x : \tau \vdash \mathcal{C}_1 : \sigma$ and $\emptyset; y : \sigma \vdash \mathcal{C}_2 : \sigma'$, then $\mathcal{C}_2[\mathcal{C}_1/y]$ is also a linear context.

PROOF . It can be shown that $\emptyset; x : \tau \vdash \mathcal{C}_2[\mathcal{C}_1/y] : \sigma'$. □

3.1. Linear context transitions

In this subsection we introduce the notion of *linear context transitions* that represent the way a context interact with programs under testing. A linear context transition often eliminates the free variable in the context or transforms it into another variable of a different type (in which case we often use a variable with a different name for the sake of clarity), which indicates that a reduction can occur involving both the candidate program and (a subterm of) the context.

Definition 4 (Linear context transition). A linear context transition is a labeled transition (from a linear context to another linear context or a closed term) of the following

⁴It is more general to consider *affine* contexts where programs are executed *at most once*, but in the current paper we refrain from going that far and leave it as future work.

form:

$$\begin{array}{lcl}
\mathcal{C}[\text{if } x \text{ then } e_1 \text{ else } e_2/y] & \xrightarrow{\text{true}} & \mathcal{C}[e_1/y] \\
\mathcal{C}[\text{if } x \text{ then } e_1 \text{ else } e_2/y] & \xrightarrow{\text{false}} & \mathcal{C}[e_2/y] \\
\mathcal{C}[\text{pred}(x)/y] & \xrightarrow{n} & \mathcal{C}[n'/y] \text{ (} n = n' + 1 \text{ or } n = n' = 0\text{)} \\
\mathcal{C}[\text{succ}(x)/y] & \xrightarrow{n} & \mathcal{C}[n'/y] \text{ (} n' = n + 1\text{)} \\
\mathcal{C}[\text{iszero}(x)/y] & \xrightarrow{n} & \mathcal{C}[\text{true}/y] \text{ (if } n = 0\text{)} \\
\mathcal{C}[\text{iszero}(x)/y] & \xrightarrow{n} & \mathcal{C}[\text{false}/y] \text{ (if } n \neq 0\text{)} \\
\mathcal{C}[\text{proj}_i(x)/y] & \xrightarrow{\text{proj}_i} & \mathcal{C}_y \text{ (} i = 1, 2\text{)} \\
\mathcal{C}[x e/y] & \xrightarrow{\text{@}e} & \mathcal{C}_y \\
\mathcal{C}[\text{let } z_1 \otimes z_2 = x \text{ in } e/y] & \xrightarrow{\otimes e} & \mathcal{C}_y
\end{array}$$

where \mathcal{C} is both a linear context and an evaluation context.

Linear context transitions do not necessarily transform a linear context into another linear context — linear contexts can also be transformed into closed terms, which do not contain any free variables. This particularly happens when the program under testing is a boolean or integer constant, which, after transition, cannot provide any information to the context.

Notice that linear contexts themselves are LPCF terms, so they can also take normal transitions as defined in Figure 2. We have used explicitly distinguished notations for the two kinds of transitions.

Lemma 4 (Transition lemma). *For every linear context $\mathcal{C}_{x:\tau}$ and LPCF program $e \in \text{Prog}(\tau)$ such that $\mathcal{C}[e/x] \not\rightsquigarrow$, a transition from $\mathcal{C}[e/x]$ must be in one of the following two forms:*

- $\mathcal{C}[e/x] \xrightarrow{\alpha} \mathcal{C}'[e/x]$ with $\mathcal{C} \xrightarrow{\alpha} \mathcal{C}'$;
- $\mathcal{C} \equiv x$ and $\mathcal{C}[e/x] \equiv e \xrightarrow{\alpha} e'$.

PROOF . Since $\mathcal{C}[e/x] \not\rightsquigarrow$, it must be in canonical form, then \mathcal{C} must be one of the forms: $x, \mathcal{C}_1 \otimes e', e' \otimes \mathcal{C}_1, \langle \mathcal{C}_1, \mathcal{C}_2 \rangle, \lambda y. \mathcal{C}_1$, where e' is a closed term and $\mathcal{C}_1, \mathcal{C}_2$ are linear contexts with free variable x .

It is clear that if $\mathcal{C} \equiv x$, the transition must be of the second form. In all other forms, it can be easily checked that the transition will be of the first form, with the context \mathcal{C} itself being transformed into another term with the free linear variable x , which forms another linear context. \square

3.2. Linear context reductions

Reductions of linear contexts filled with programs can be classified into several forms, called *linear context reductions* (LCR for short), which characterize the interactions between linear contexts and programs.

Definition 5 (Linear context reduction). *Let $\mathcal{C}_{x:\tau}$ be a linear context and $e \in \text{Prog}(\tau)$ be a LPCF program. A reduction of $\mathcal{C}[e/x]$ (if it is reducible) is called a linear context reduction if it is in one of the following forms:*

- $\mathcal{C}[e/x] \rightsquigarrow \mathcal{C}'[e/x]$, if $\mathcal{C} \rightsquigarrow \mathcal{C}'$;
- $\mathcal{C}[e/x] \rightsquigarrow \mathcal{C}[e'/x]$, if \mathcal{C} is an evaluation context, and $e \rightsquigarrow e'$;
- $\mathcal{C}[e/x] \rightsquigarrow \mathcal{C}'[e'/y]$, if \mathcal{C} is an evaluation context, $e \not\rightsquigarrow$, and $\mathcal{C} \circ^{\alpha} \rightarrow \mathcal{C}'$, $e \xrightarrow{\alpha} e'$ for some external action α .

We often write $\mathcal{C}[e/x] \xrightarrow{\alpha} \mathcal{C}'[e'/y]$ for the third form of linear context reduction, indicating explicitly that the transitions involved are labeled by α .

Linear context reductions are closed under linear evaluation contexts:

Lemma 5. *Let $\mathcal{C}_1, \mathcal{C}_2$ be two linear contexts such that $\emptyset; x : \tau \vdash \mathcal{C}_1 : \sigma$ and $\emptyset; y : \sigma \vdash \mathcal{C}_2 : \sigma'$, and \mathcal{C}_2 also an evaluation context.*

1. If $\mathcal{C}_1 \xrightarrow{\alpha} e$, then $\mathcal{C}_2[\mathcal{C}_1/y] \xrightarrow{\alpha} \mathcal{C}_2[e/y]$.
2. If $\mathcal{C}_1[e/x] \rightsquigarrow e'$ is a linear context reduction, then $\mathcal{C}_2[\mathcal{C}_1[e/x]/y] \rightsquigarrow \mathcal{C}_2[e'/y]$ is also a linear context reduction.

PROOF . Direct consequence of the definition of linear context transitions. \square

The so-called *linear context reduction lemma* below says that, the reduction of a linear context filled with a program in LPCF must be a linear context reduction. This is the core lemma of proving precongruence of trace equivalence w.r.t. linear contexts.

Lemma 6 (Linear context reduction lemma). *For every linear context $\mathcal{C}_{x:\tau}$ and LPCF program $e \in \text{Prog}(\tau)$, if $\mathcal{C}[e/x]$ is reducible, then $\mathcal{C}[e/x] \rightsquigarrow$ must be a linear context reduction.*

PROOF . We prove by induction on the structure of the linear context.

- \mathcal{C} cannot be any constant since it must contain a linear free variable. And it cannot be a normal product, a tensor product or an abstraction, as all these forms cannot be reduced any more, no matter what e is.
- The simplest linear context x is an evaluation context. If e can be reduced, then it is the second case.
- $\mathcal{C} \equiv \text{if } \mathcal{C}' \text{ then } e_1 \text{ else } e_2$, where \mathcal{C}' is another linear context. If $\mathcal{C}'[e/x]$ can be reduced, by induction, it must be in one of the following cases:

- $\mathcal{C}'[e/x] \rightsquigarrow \mathcal{C}''[e/x]$ and $\mathcal{C}' \rightsquigarrow \mathcal{C}''$, then we have that

$$\mathcal{C}[e/x] \rightsquigarrow \text{if } \mathcal{C}''[e/x] \text{ then } e_1 \text{ else } e_2$$

with $\mathcal{C} \rightsquigarrow \text{if } \mathcal{C}'' \text{ then } e_1 \text{ else } e_2$.

- $\mathcal{C}'[e/x] \rightsquigarrow \mathcal{C}'[e'/x]$, $e \rightsquigarrow e'$, and \mathcal{C}' is an evaluation context, then \mathcal{C} is also an evaluation context, hence $\mathcal{C}[e/x] \rightsquigarrow \mathcal{C}[e'/x]$.
- $\mathcal{C}'[e/x] \rightsquigarrow \mathcal{C}''[e'/x]$ and $\mathcal{C}' \circ^{\alpha} \rightarrow \mathcal{C}''$, $e \xrightarrow{\alpha} e'$ for some action α , then

$$\mathcal{C} \circ^{\alpha} \rightarrow \text{if } \mathcal{C}'' \text{ then } e_1 \text{ else } e_2$$

and $\mathcal{C}[e/x]$ can take a similar reduction.

If $\mathcal{C}'[e/x]$ cannot reduce, then it is a canonical boolean term, which is either `true` or `false`, and the only possibility of \mathcal{C}' is the simplest case x , with e being a boolean constant. In this case both \mathcal{C} and e can take the transition $\xrightarrow{\text{true}}$ or $\xrightarrow{\text{false}}$, and the reduction of $\mathcal{C}[e/x]$ falls into the third case.

- $\mathcal{C} \equiv \text{if } e' \text{ then } \mathcal{C}_1 \text{ else } \mathcal{C}_2$, where by typing, both \mathcal{C}_1 and \mathcal{C}_2 are linear contexts. If e' can be reduced ($e' \rightsquigarrow e''$), then $\mathcal{C}[e/x]$ will reduce to

$$\text{if } e'' \text{ then } \mathcal{C}_1[e/x] \text{ else } \mathcal{C}_2[e/x],$$

which is still a linear context. If e' cannot be reduced, then it must be a boolean constant since it must be canonical, then $\mathcal{C}[e/x]$ will reduce to either $\mathcal{C}_1[e/x]$ or $\mathcal{C}_2[e/x]$. Both reductions are the first form of LCR.

- $\mathcal{C} \equiv \text{proj}_i(\mathcal{C}')$, where \mathcal{C}' is a linear context. If $\mathcal{C}'[e/x]$ itself can be reduced, then by induction, it must be in one of the three forms of linear context reduction. In each case, it is easy to see that $\mathcal{C}[e/x]$ will take the same form of reduction.

If $\mathcal{C}'[e/x]$ is not reducible, then it must be of the form $\langle _ , _ \rangle$. There are two cases

- $\mathcal{C}' \equiv \langle \mathcal{C}'_1, \mathcal{C}'_2 \rangle$, where both \mathcal{C}'_1 and \mathcal{C}'_2 are linear contexts, then $\mathcal{C}[e/x] \rightsquigarrow \mathcal{C}_i[e/x]$, which is the first form of linear context reduction.

- $\mathcal{C}' \equiv x$ and $e \equiv \langle e_1, e_2 \rangle$. Now both \mathcal{C} and e can take the transition $\xrightarrow{\text{proj}_i}$:
 $\mathcal{C} \equiv \text{proj}_i(x) \circ \xrightarrow{\text{proj}_i} y$, $e \equiv \langle e_1, e_2 \rangle \xrightarrow{\text{proj}_i} e_i$ and $\mathcal{C}[e/x] \rightsquigarrow e_i = y[e_i/y]$. This is the third form of linear context reduction.

- $\mathcal{C} \equiv \text{let } y_1 \otimes y_2 = \mathcal{C}' \text{ in } e'$. If \mathcal{C}' is reducible, by induction, the reduction of $\mathcal{C}'[e/x]$ must be a linear context reduction, then the reduction of $\mathcal{C}[e/x]$ will be a linear context reduction of the same form.

If $\mathcal{C}'[e/x]$ is not reducible, then there are two cases:

- $\mathcal{C}' \equiv \mathcal{C}'' \otimes e''$ or $\mathcal{C}' \equiv e'' \otimes \mathcal{C}''$, where e'' is a closed term and \mathcal{C}'' is a linear context. Consider the first case without losing generality. $\mathcal{C}[e/x]$ will reduce to $e'[\mathcal{C}''[e/x]/y_1, e''/y_2]$. It is easy to check that $e'[\mathcal{C}''[e/x]/y_1, e''/y_2]$ is also a linear context, then so is $e'[\mathcal{C}''[e/x]/y_1, e''/y_2]$, so the reduction is a linear context reduction of the first form.

- $\mathcal{C}' \equiv x$ and $e \equiv e_1 \otimes e_2$. Now both \mathcal{C} and e can take a $\xrightarrow{\otimes e'}$ transition:
 $\mathcal{C} \equiv \text{let } y_1 \otimes y_2 = x \text{ in } e' \circ \xrightarrow{\otimes e'} z$, $e \equiv e_1 \otimes e_2 \xrightarrow{\otimes e'} e'[e_1/y_1, e_2/y_2]$,
and $\mathcal{C}[e/x] \rightsquigarrow e'[e_1/y_1, e_2/y_2] = z[e_1/y_1, e_2/y_2]$.

- $\mathcal{C} \equiv \text{let } y \otimes z = e' \text{ in } \mathcal{C}'$. It is clear that e' is a closed term and $FLV(\mathcal{C}') = \{x, y, z\}$. If $e' \rightsquigarrow e''$ reduces, then $\mathcal{C}[e/x] \rightsquigarrow \text{let } y \otimes z = e'' \text{ in } \mathcal{C}'[e/x]$. Otherwise, e' must be $e'_1 \otimes e'_2$, then $\mathcal{C}[e/x] \rightsquigarrow \mathcal{C}'[e/x, e'_1/y, e'_2/z]$, with \mathcal{C} reducing to $\mathcal{C}'[e'_1/y, e'_2/z]$, which is a linear context.

- $\mathcal{C} \equiv \mathcal{C}' e'$. Because \mathcal{C}' is a linear context, by induction, if $\mathcal{C}'[e/x]$ can be reduced, then it must be a linear context reduction. As $\mathcal{C}' e'$ is an evaluation context, $\mathcal{C}[e/x]$ will take the same form of linear context reduction as $\mathcal{C}'[e/x]$.

If $\mathcal{C}'[e/x]$ cannot be reduced, then it must be an abstraction. There are two cases:

- $\mathcal{C}' \equiv \lambda y. \mathcal{C}''$ and $x \in FLV(\mathcal{C}'')$, then $\mathcal{C} \equiv (\lambda y. \mathcal{C}'')e' \rightsquigarrow \mathcal{C}''[e'/y]$ and it is easy to check that $\mathcal{C}''[e'/y]$ is a linear context since e' is closed, hence $\mathcal{C}[e/x] \equiv (\lambda y. \mathcal{C}''[e/x])e' \rightsquigarrow \mathcal{C}''[e'/y][e/x]$, which is the first form of linear context reduction.
- $\mathcal{C}' \equiv x$ and $e \equiv \lambda y. e''$ is an abstraction, then $\mathcal{C} \equiv x e' \circ \xrightarrow{\textcircled{e'}} z$ (with z being a fresh linear variable, hence a linear context), $e \equiv \lambda y. e'' \xrightarrow{\textcircled{e'}} e''[e'/y]$, and $\mathcal{C}[e/x] \equiv (\lambda y. e'')e' \rightsquigarrow e''[e'/y] \equiv z[e''[e'/y]/z]$.
- $\mathcal{C} \equiv e' \mathcal{C}'$. If $e' \rightsquigarrow e''$, then $\mathcal{C}[e/x] \rightsquigarrow e''(\mathcal{C}'[e/x])$ with $\mathcal{C} \rightsquigarrow e'' \mathcal{C}'$. If e' cannot be reduced, then it must be canonical, which is either an abstraction or a constant. Because \mathcal{C}' contains a linear variable, according to the typing system, the type of e' can only be a linear function type.

- If $e' \equiv \lambda y. e''$, then

$$\mathcal{C}[e/x] \equiv (\lambda y. e'')(\mathcal{C}'[e/x]) \rightsquigarrow e''[\mathcal{C}'[e/x]/y] = e''[\mathcal{C}'/y][e/x].$$

Also $\mathcal{C} \rightsquigarrow e''[\mathcal{C}'/y]$. Because y is a free linear variable in e'' , e'' is indeed a linear context, so is $e''[\mathcal{C}'/y]$.

- If e' is a constant, because its type must be a linear function type, so it can only be one of $\{\text{pred}, \text{succ}, \text{iszero}\}$. In any case, $e' \mathcal{C}'$ is an evaluation context. If $\mathcal{C}'[e/x]$ reduces, then by induction it must be a linear context reduction, hence $e' \mathcal{C}[e/x]$ can reduce and is a linear context reduction of the same form as of $\mathcal{C}'[e/x]$. If $\mathcal{C}'[e/x]$ cannot reduce, it must be canonical, i.e., an integer n , then $\mathcal{C}' \equiv x$ and $e \equiv n$. Now both \mathcal{C} and e can take a \xrightarrow{n} transition and $\mathcal{C}[e/x]$ will reduce to another integer or a boolean constant, depending on which constant e' is. \square

The linear context reduction lemma is not true if the context is not linear or the language does not have linear types at all, because duplicated use of programs in the context will adopt reductions that cannot be characterized by LCR, particularly when the program itself is reducible, i.e., $\mathcal{C}[e/x] \rightsquigarrow \mathcal{C}[e'/x]$ is not true when $e \rightsquigarrow e'$ and \mathcal{C} makes multiple copies of e . The reduction strategy also interferes, as we have mentioned when defining the operational semantics: introducing improper evaluation contexts like $\langle \mathcal{E}, e \rangle$ breaks the linear context lemma, for the same reason as using non-linear contexts.

3.3. Soundness and completeness of trace equivalence

We show that in LPCF, the trace preorder relation is precongruent with respect to linear contexts. It then enables us to show that trace equivalence actually coincides with linear contextual equivalence.

The following theorem says that trace preorder in LPCF is a precongruence relation with respect to linear contexts. As LPCF is a deterministic language, the proof can be done by induction on (the length of) traces.

Theorem 1 (Linear precongurence of \sqsubseteq^T). *Trace preorder \sqsubseteq^T is a precongurence with respect to linear contexts, i.e., $e_1 \sqsubseteq^T e_2$ implies that $\mathcal{C}[e_1/x] \sqsubseteq^T \mathcal{C}[e_2/x]$ for all linear contexts \mathcal{C}_x .*

PROOF . According to the definition of \sqsubseteq^T , it suffices to show that, for any action sequence s , if $\mathcal{C}[e_1/x] \xrightarrow{s}$, then $\mathcal{C}[e_2/x] \xrightarrow{s}$. We prove by induction on the length of $\mathcal{C}[e_1/x] \xrightarrow{s}$ (note that the transition includes internal transitions, i.e., term reductions). The base case is trivial.

We distinguish two cases.

- $\mathcal{C}[e_1/x] \rightsquigarrow e \xrightarrow{s}$. By the linear context lemma, the reduction must be a linear context reduction, which is one of the following cases:
 - $e \equiv \mathcal{C}'[e_1/x]$ where $\mathcal{C} \rightsquigarrow \mathcal{C}'$. It holds that $\mathcal{C}[e_2/x] \rightsquigarrow \mathcal{C}'[e_2/x]$. By induction, $\mathcal{C}'[e_2/x] \xrightarrow{s}$ since $\mathcal{C}'[e_1/x] \xrightarrow{s}$, hence $\mathcal{C}[e_2/x] \rightsquigarrow \mathcal{C}'[e_2/x] \xrightarrow{s}$, i.e., $\mathcal{C}[e_2/x] \xrightarrow{s}$.
 - $e \equiv \mathcal{C}[e'_1/x]$ where $e_1 \rightsquigarrow e'_1$. We immediately have $e'_1 \sqsubseteq^T e_1 \sqsubseteq^T e_2$ and by induction, $\mathcal{C}[e_2/x] \xrightarrow{s}$ because $\mathcal{C}[e'_1/x] \xrightarrow{s}$.
 - $e \equiv \mathcal{C}'_y[e'_1/y]$ where $\mathcal{C} \circ \xrightarrow{\alpha} \mathcal{C}'_y$ and $e_1 \xrightarrow{\alpha} e'_1$. Since $e_1 \sqsubseteq^T e_2$ and the transitions are deterministic, we have $e_2 \xrightarrow{\alpha} e'_2$ and $e'_1 \sqsubseteq^T e'_2$. It is clear that $e_2 \xrightarrow{\alpha} e'_2$ must be of the form $e_2 \rightsquigarrow^* e''_2 \xrightarrow{\alpha} e'_2$, where $e''_2 \not\rightsquigarrow$. By the definition of linear context reduction, \mathcal{C} must be an evaluation context, hence $\mathcal{C}[e_2/x] \rightsquigarrow^* \mathcal{C}[e''_2/x] \rightsquigarrow \mathcal{C}'_y[e'_2/y]$, and by induction, $\mathcal{C}'_y[e'_1/y] \xrightarrow{s}$ implies $\mathcal{C}'_y[e'_2/y] \xrightarrow{s}$.
- $\mathcal{C}[e_1/x] \xrightarrow{\alpha} e \xrightarrow{s'}$. By Lemma 4, the first transition has two forms:
 - $\mathcal{C} \xrightarrow{\alpha} \mathcal{C}'$ and $e \equiv \mathcal{C}'[e_1/x]$. By induction, $\mathcal{C}'[e_1/x] \xrightarrow{s'}$ implies $\mathcal{C}'[e_2/x] \xrightarrow{s'}$. It follows that $\mathcal{C}[e_2/x] \xrightarrow{\alpha} \mathcal{C}'[e_2/x] \xrightarrow{s'}$.
 - $\mathcal{C} \equiv x$ and $e_1 \xrightarrow{\alpha} e'_1 \equiv e$. Then $e_1 \sqsubseteq^T e_2$ implies that $\mathcal{C}[e_2/x] \equiv e_2 \xrightarrow{\alpha} \xrightarrow{s'}$. □

However, the above proof does not apply in non-deterministic languages as trace preorder does not conform to induction in general. We supply in this section a more general proof to establish linear precongurence of trace preorder, by exploiting the intrinsic features of linear contexts.

Given a term $\mathcal{C}[e/x]$, it may exhibit a trace s by performing a sequence of internal and external transitions. The external transitions are generated either by \mathcal{C} independently or by some interactions between \mathcal{C} and e . We would like to describe these interactions by the concept of context trace.

For every linear context $\mathcal{C}_{x:\tau}$ and LPCF program $e \in \mathcal{P}rog(\tau)$, if $\mathcal{C}[e/x] \xrightarrow{s}$ and $e \xrightarrow{t}$, we define t to be the *context trace* w.r.t. \mathcal{C} and s (also written as (\mathcal{C}, s) -trace), inductively on the full sequence of s , if:

- $t = \epsilon$ when s is empty;

- t is the context trace of e' w.r.t. \mathcal{C} and s when $\mathcal{C}[e/x] \rightsquigarrow \mathcal{C}[e'/x]$ with $e \rightsquigarrow e'$;
- t is the context trace of e w.r.t. \mathcal{C}' and s when $\mathcal{C}[e/x] \rightsquigarrow \mathcal{C}'[e/x]$ with $\mathcal{C} \rightsquigarrow \mathcal{C}'$;
- $t = \alpha \cdot t'$ and t' is the context trace of e' w.r.t. \mathcal{C}' and s when $\mathcal{C}[e/x] \xrightarrow{\alpha} \mathcal{C}'[e'/x]$;
- t is the context trace of e w.r.t. \mathcal{C}' and s' when $\mathcal{C}[e/x] \xrightarrow{\alpha} \mathcal{C}'[e/x]$ with $\mathcal{C} \circ \xrightarrow{\alpha} \mathcal{C}'$ and $s = \alpha \cdot s'$;
- $t = s$ when $\mathcal{C} \equiv x$.

Intuitively, a (\mathcal{C}, s) -trace is a sequence of actions t that a term e can perform by interacting with a context \mathcal{C} and when the term is plugged in the context the compound term $\mathcal{C}[e/x]$ can perform a sequence of actions s .

Lemma 7. *For every linear context $\mathcal{C}_{x:\tau}$ and LPCF program $e \in \text{Prog}(\tau)$, if $\mathcal{C}[e/x] \xrightarrow{s}$, then e has a context trace w.r.t. \mathcal{C} and s .*

PROOF . The definition of context trace is solid by Lemma 6 and Lemma 4, hence it is always feasible to construct the (\mathcal{C}, s) -trace from the full sequence of s — the definition indeed gives the construction. \square

We also write $\mathcal{C}[e/x] \xrightarrow{(s,t)}$ when t is a context trace of e w.r.t. \mathcal{C} and s .

Lemma 8. *For every pair of LPCF traces (s, t) and LPCF programs $e_1, e_2 \in \text{Prog}(\tau)$, if $e_1 \xrightarrow{t}$ and $e_2 \xrightarrow{t}$, then for all linear context $\mathcal{C}_{x:\tau}$, $\mathcal{C}[e_1/x] \xrightarrow{(s,t)}$ implies $\mathcal{C}[e_2/x] \xrightarrow{(s,t)}$.*

PROOF . We prove by induction on the full length of $\mathcal{C}[e_1/x] \xrightarrow{s}$, counting internal transitions.

The base case is trivial. For non-empty traces, we analyze by cases:

- $\mathcal{C}[e_1/x] \rightsquigarrow \mathcal{C}'[e_1/x] \xrightarrow{(s,t)}$ with $\mathcal{C} \rightsquigarrow \mathcal{C}'$. By induction $\mathcal{C}'[e_2/x] \xrightarrow{(s,t)}$, hence $\mathcal{C}[e_2/x] \rightsquigarrow \mathcal{C}'[e_2/x] \xrightarrow{(s,t)}$.
- $\mathcal{C}[e_1/x] \rightsquigarrow \mathcal{C}[e'_1/x] \xrightarrow{(s,t)}$ with $e_1 \rightsquigarrow e'_1$. Clearly $e'_1 \xrightarrow{t}$, so by induction, $\mathcal{C}[e_2/x] \xrightarrow{(s,t)}$.
- $\mathcal{C}[e_1/x] \xrightarrow{\alpha} \mathcal{C}'[e'_1/y] \xrightarrow{(s,t')}$ with $t = \alpha \cdot t'$. Since $e_2 \xrightarrow{t}$, i.e., there exist e'_2 and e''_2 such that $e_2 \rightsquigarrow^* e''_2 \xrightarrow{\alpha} e'_2 \xrightarrow{t'}$, by induction, we have $\mathcal{C}'[e'_2/y] \xrightarrow{(s,t')}$. According to the definition of linear context reduction, \mathcal{C} must be an evaluation context, hence $\mathcal{C}[e_2/x] \rightsquigarrow^* \mathcal{C}[e''_2/x] \xrightarrow{\alpha} \mathcal{C}'[e'_2/y] \xrightarrow{(s,t')}$, i.e., $\mathcal{C}[e_2/x] \xrightarrow{(s,t)}$.
- $\mathcal{C}[e_1/x] \xrightarrow{\alpha} \mathcal{C}'[e_1/x] \xrightarrow{(s',t)}$ with $\mathcal{C} \circ \xrightarrow{\alpha} \mathcal{C}'$ and $s = \alpha \cdot s'$. By induction, $\mathcal{C}'[e_2/x] \xrightarrow{(s',t)}$, which follows that $\mathcal{C}[e_2/x] \xrightarrow{\alpha} \mathcal{C}'[e_2/x] \xrightarrow{(s',t)}$, i.e., $\mathcal{C}[e_2/x] \xrightarrow{(s,t)}$.
- $\mathcal{C} \equiv x$ and $s = t$. Clearly $\mathcal{C}[e_2/x] \equiv e_2 \xrightarrow{t}$, i.e., $\mathcal{C}[e_2/x] \xrightarrow{(s,t)}$.

Lemma 6 and Lemma 4 ensure that the above analysis is comprehensive. \square

PROOF . [Theorem 1] Consider arbitrary linear context \mathcal{C} and trace s such that $\mathcal{C}[e_1/x] \xrightarrow{s}$. By Lemma 7, e_1 has a (\mathcal{C}, s) -trace t , i.e., $e_1 \xrightarrow{t}$, which implies $e_2 \xrightarrow{t}$ since $e_1 \sqsubseteq^T e_2$. By Lemma 8, we have $\mathcal{C}[e_2/x] \xrightarrow{(s,t)}$, which means that $\mathcal{C}[e_2/x] \xrightarrow{s}$. Then it follows that $\mathcal{C}[e_1/x] \sqsubseteq^T \mathcal{C}[e_2/x]$. \square

Remark 1. *The above proof, as well as the last two technical lemmas, does not rely on the determinism of the transition semantics of LPCF. So it can be transplanted to the non-deterministic setting to establish Theorem 4 in the next section.*

Theorem 2 (Soundness of trace equivalence). *In LPCF, it holds that $\simeq^T \subseteq \simeq^C$.*

PROOF . For every well typed linear context \mathcal{C}_x , if $\mathcal{C}[e_1/x] \Downarrow$, i.e. $\mathcal{C}[e_1/x] \rightsquigarrow^* v$ for some canonical term v , then $\mathcal{C}[e_1/x] \rightsquigarrow^* v \xrightarrow{\alpha}$ for some external action α . By the precongruence property of \sqsubseteq^T , Theorem 1, we have $\mathcal{C}[e_1/x] \sqsubseteq^T \mathcal{C}[e_2/x]$. Therefore, there is some term e such that $\mathcal{C}[e_2/x] \rightsquigarrow^* e \xrightarrow{\alpha}$. In order to perform an external action, here e must be a canonical term and it follows that $\mathcal{C}[e_2/x] \Downarrow$. Similarly we can show that if $\mathcal{C}[e_2/x] \Downarrow$, then $\mathcal{C}[e_1/x] \Downarrow$. \square

Theorem 3 (Completeness). *In LPCF, it holds that $\simeq^C \subseteq \simeq^T$.*

PROOF . We first notice that in Definition 3 the relations \sqsubseteq^C and \simeq^C are defined by quantifying over all linear contexts. In fact, it suffices to quantify over the subset of linear contexts that are evaluation contexts (viewing \mathcal{C}_x as $\mathcal{C}[[\]/x]$). In other words, for any two terms of the same type,

(*) if they are distinguished by a linear context, with $\mathcal{C}[e_1/x] \Downarrow$ but $\mathcal{C}[e_2/x] \Uparrow$, then they are also distinguished by an evaluation context \mathcal{C}' with $\mathcal{C} \rightsquigarrow^* \mathcal{C}'$.

This is proved as follows. Suppose $\mathcal{C}[e_1/x] \Downarrow$ but $\mathcal{C}[e_2/x] \Uparrow$. We observe that all reduction sequence starting from \mathcal{C} must terminate in order to ensure $\mathcal{C}[e_1/x] \Downarrow$. So we can proceed by induction on the length of the reduction sequence.

- If \mathcal{C} is already an evaluation context, then we are done by setting \mathcal{C}' to be \mathcal{C} .
- \mathcal{C} cannot be a normal product, a tensor product or an abstraction, as all these forms cannot be reduced any more, and are not able to meet the requirement that $e_2 \Uparrow$.
- For all other cases, if $\mathcal{C} \rightsquigarrow \mathcal{C}_1$ then \mathcal{C}_1 is also a linear context and by determinacy of reduction semantics, Proposition 3, we have $\mathcal{C}[e_1/x] \rightsquigarrow \mathcal{C}_1[e_1/x] \Downarrow$ and $\mathcal{C}[e_2/x] \rightsquigarrow \mathcal{C}_1[e_2/x] \Uparrow$. By induction applied to \mathcal{C}_1 , there exists some evaluation context \mathcal{C}' such that $\mathcal{C}_1 \rightsquigarrow^* \mathcal{C}'$, $\mathcal{C}'[e_1/x] \Downarrow$ and $\mathcal{C}'[e_2/x] \Uparrow$. Hence $\mathcal{C} \rightsquigarrow^* \mathcal{C}'$ and we can find the required \mathcal{C}' .

We now show that, for any terms e_1, e_2 of the same type with $e_1 \simeq^C e_2$ and any action sequence s , if $e_1 \xrightarrow{s}$ then $e_2 \xrightarrow{s}$, which establishes $e_1 \sqsubseteq^T e_2$. Similarly we can prove $e_2 \sqsubseteq^T e_1$ but we shall omit the details.

We proceed by induction on the length of the transition $e_1 \xrightarrow{s}$. The base case is trivial. For the inductive step, we distinguish two cases.

- $e_1 \rightsquigarrow e'_1 \xrightarrow{s}$. Clearly, we can prove, by induction on the structure of context, that $e'_1 \sqsubseteq^C e_1$, then $e'_1 \sqsubseteq^C e_2$. By induction, we obtain that $e_2 \xrightarrow{s}$.
- $e_1 \xrightarrow{\alpha} e'_1 \xrightarrow{s}$. There are a few sub-cases, depending on the form of α .

- $\alpha \equiv n$. Both e_1 and e_2 have type Nat , and $e_1 \xrightarrow{n} \Omega$, so $e_1 \Downarrow$ and $e_1 \equiv n$. Because $e_1 \simeq^C e_2$, $e_2 \Downarrow$ too (otherwise the simple linear context x can distinguish them). We claim that for every possible reduction sequence $e_2 \rightsquigarrow^* e'_2 \not\rightsquigarrow, e'_2 \equiv n$. First, because e_2 has type Nat , by Proposition 1, e'_2 has to be an integer constant. Assume that $e_2 \rightsquigarrow^* m$ and $m \neq n$. Then the context

$$C_x \equiv \text{if } x = n \text{ then } 0 \text{ else } \Omega$$

will distinguish e_1 from e_2 , which contradicts $e_1 \simeq^C e_2$.

Similar is the case where α is a boolean constant.

- $\alpha \equiv @e$. In this case e_1 and e_2 must have a function type, and clearly e_1 is in canonical form: $e_1 \equiv \lambda x.e''_1$ and $e'_1 \equiv e''_1[e/x]$. Because $e_1 \simeq^C e_2$, the reduction of e_2 necessarily terminates and e_2 will be reduced to some canonical form $\lambda x.e''_2$, then $e_2 \xrightarrow{@e} e''_2[e/x]$. We claim that $e''_1[e/x] \sqsubseteq^C e''_2[e/x]$.

Suppose for a contradiction that $e''_1[e/x] \not\sqsubseteq^C e''_2[e/x]$. There exists some linear context \mathcal{C} such that $\mathcal{C}[e''_1[e/x]/y] \Downarrow$ but $\mathcal{C}[e''_2[e/x]/y] \Uparrow$. By property (*) above, we can assume that \mathcal{C} is an evaluation context. Then we can construct another context $\mathcal{C}' := \mathcal{C}[ye/y]$. Clearly $\mathcal{C}'[e_1/y] \Downarrow$ because

$$\mathcal{C}'[e_1/y] \equiv \mathcal{C}[e_1e/y] \equiv \mathcal{C}[(\lambda x.e''_1)e/y] \rightsquigarrow \mathcal{C}[e''_1[e/x]/y] \Downarrow.$$

However, $\mathcal{C}'[e_2/y] \Uparrow$ because

$$\mathcal{C}'[e_2/y] \equiv \mathcal{C}[e_2e/y] \rightsquigarrow^* \mathcal{C}[(\lambda x.e''_2)e/y] \rightsquigarrow \mathcal{C}[e''_2[e/x]/y] \Uparrow.$$

This is a contradiction to $e_1 \sqsubseteq^C e_2$. Therefore the assumption is wrong and we have $e''_1[e/x] \sqsubseteq^C e''_2[e/x]$. By induction, we have $e''_2[e/x] \xrightarrow{s}$ and it follows that $e_2 \xrightarrow{\alpha} e''_2[e/x] \xrightarrow{s}$.

- $\alpha \equiv \text{proj}_1$. In this case e_1, e_2 must have a normal product type, then e_1 is in canonical form $\langle e_{11}, e_{12} \rangle$ and $e'_1 \equiv e_{11}$. The term e_2 can be reduced to a canonical term $\langle e_{21}, e_{22} \rangle$, and then $e_2 \xrightarrow{\text{proj}_1} e_{21}$. We claim that $e_{11} \sqsubseteq^C e_{21}$.

Suppose for a contradiction that $e_{11} \not\sqsubseteq^C e_{21}$. There exists some linear context \mathcal{C} such that $\mathcal{C}[e_{11}/y] \Downarrow$ but $\mathcal{C}[e_{21}/y] \Uparrow$. By property (*), \mathcal{C} can be assumed to be an evaluation context. Then we can construct another context $\mathcal{C}' := \mathcal{C}[\text{proj}_1(y)/y]$. Clearly $\mathcal{C}'[e_1/y] \Downarrow$ because

$$\mathcal{C}'[e_1/y] \equiv \mathcal{C}[\text{proj}_1(e_1)/y] \equiv \mathcal{C}[\text{proj}_1(\langle e_{11}, e_{12} \rangle)/y] \rightsquigarrow \mathcal{C}[e_{11}/y] \Downarrow.$$

However, $\mathcal{C}'[e_2/y] \Uparrow$ because

$$\mathcal{C}'[e_2/y] \equiv \mathcal{C}[\text{proj}_1(e_2)/y] \rightsquigarrow^* \mathcal{C}[\text{proj}_1(\langle e_{21}, e_{22} \rangle)/y] \rightsquigarrow \mathcal{C}[e_{21}/y] \Uparrow.$$

This is a contradiction to $e_1 \sqsubseteq^C e_2$. Therefore the assumption is wrong and we have $e_{11} \sqsubseteq^C e_{21}$. By induction, we have $e_{21} \xrightarrow{s}$ and it follows that $e_2 \xrightarrow{\alpha} e_{21} \xrightarrow{s}$.

The case for $\alpha \equiv \text{proj}_2$ is similar.

- $\alpha \equiv \otimes e$. In this case e_1, e_2 must have a tensor product type, then e_1 is in canonical form $e_{11} \otimes e_{12}$ and $e'_1 \equiv e[e_{11}/x, e_{12}/y]$. The term e_2 can be reduced to a canonical term $e_{21} \otimes e_{22}$, and then $e_2 \xrightarrow{\otimes e} e[e_{21}/x, e_{22}/y]$. We claim that

$$e[e_{11}/x, e_{12}/y] \sqsubseteq^C e[e_{21}/x, e_{22}/y].$$

Suppose for a contradiction that $e[e_{11}/x, e_{12}/y] \not\sqsubseteq^C e[e_{21}/x, e_{22}/y]$. There exists some linear context \mathcal{C} such that $\mathcal{C}[e'_1/z] \Downarrow$ but $\mathcal{C}[(e[e_{21}/x, e_{22}/y])/z] \Uparrow$. By property (*) above, we can assume that \mathcal{C} is an evaluation context. Then we can construct another context $\mathcal{C}' := \mathcal{C}[(\text{let } x \otimes y = z \text{ in } e)/z]$. Clearly $\mathcal{C}'[e_1/z] \Downarrow$ because

$$\begin{aligned} \mathcal{C}'[e_1/z] &\equiv \mathcal{C}[(\text{let } x \otimes y = e_1 \text{ in } e)/z] \\ &\equiv \mathcal{C}[(\text{let } x \otimes y = e_{11} \otimes e_{12} \text{ in } e)/z] \\ &\rightsquigarrow \mathcal{C}[(e[e_{11}/x, e_{12}/y])/z] \Downarrow. \end{aligned}$$

However, $\mathcal{C}'[e_2/z] \Uparrow$ because

$$\begin{aligned} \mathcal{C}'[e_2/z] &\equiv \mathcal{C}[(\text{let } x \otimes y = e_2 \text{ in } e)/z] \\ &\rightsquigarrow^* \mathcal{C}[(\text{let } x \otimes y = e_{21} \otimes e_{22} \text{ in } e)/z] \\ &\rightsquigarrow \mathcal{C}[(e[e_{21}/x, e_{22}/y])/z] \Uparrow. \end{aligned}$$

This is a contradiction to $e_1 \sqsubseteq^C e_2$. Therefore the assumption is wrong and we have $e[e_{11}/x, e_{12}/y] \sqsubseteq^C e[e_{21}/x, e_{22}/y]$. By induction, we have the transition $e[e_{21}/x, e_{22}/y] \xrightarrow{s}$. It follows that $e_2 \xrightarrow{\alpha} e[e_{21}/x, e_{22}/y] \xrightarrow{s}$. \square

4. The non-deterministic linear PCF

In this section we shall extend our language with non-determinism, an important feature that appeared in the example in Section 1. We show that our approach can still be applied to characterize linear contextual equivalence in the non-deterministic setting.

The extension of non-determinism is made in Moggi's computational framework [25], which provides a call-by-value wrapping of imperative features in pure functional languages, using monadic types. We use Moggi's framework also because our original semantics of LPCF is a call-by-name evaluation strategy, while we need the call-by-value evaluation of non-deterministic choice for illustrating interesting effects. Were the original semantics call-by-value, we would not have to use Moggi's framework.

The types of the non-deterministic LPCF (NLPCF for short) are extended by a unary type constructor \top — $\top\tau$ is the type for non-deterministic computations that

return values of type τ if the computations terminate. The language then has extra constructs related to non-determinism:

$e, e', \dots ::= \dots$	$\text{val}(e)$ $\text{bind } x = e \text{ in } e'$ $e \sqcap e'$	Trivial computation Sequential composition Non-deterministic choice
--------------------------	---	---

$\text{val}(e)$ is the trivial computation that returns directly e as a value; $\text{bind } x = e \text{ in } e'$ binds the value of the (non-deterministic) computation e to the variable x and evaluates e' ; $e \sqcap e'$ chooses non-deterministically a computation from e and e' and executes it.

Type assertions for the extra constructs are defined by the following rules:

$$\frac{\Gamma; \Delta \vdash e : \tau}{\Gamma; \Delta \vdash \text{val}(e) : \mathbb{T}\tau} \quad \frac{\Gamma; \emptyset \vdash e_1 : \mathbb{T}\tau_1 \quad \Gamma, x : \tau_1; \Delta \vdash e_2 : \mathbb{T}\tau_2}{\Gamma; \Delta \vdash \text{bind } x = e_1 \text{ in } e_2 : \mathbb{T}\tau_2}$$

$$\frac{\Gamma; \Delta \vdash e_1 : \mathbb{T}\tau_1 \quad \Gamma; \Delta', x : \tau_1 \vdash e_2 : \mathbb{T}\tau_2}{\Gamma; \Delta, \Delta' \vdash \text{bind } x = e_1 \text{ in } e_2 : \mathbb{T}\tau_2} \quad \frac{\Gamma; \Delta \vdash e_i : \mathbb{T}\tau \ (i = 1, 2)}{\Gamma; \Delta \vdash e_1 \sqcap e_2 : \mathbb{T}\tau}$$

The typing for sequential computation must respect the linearity restriction. Also, linear variables appear in both branches of the non-deterministic choice, since eventually only one branch will be executed.

We write $\text{Prog}^{NL}(\tau)$ for the set of programs (closed terms) of type τ in NLPCF.

4.1. Operational semantics

The operational semantics of NLPCF is extended with the following basic reduction rules

$$\begin{aligned} \text{bind } x = \text{val}(e') \text{ in } e &\rightsquigarrow (\lambda x. e)e', \text{ where } e' \not\rightsquigarrow, \\ e_1 \sqcap e_2 &\rightsquigarrow e_i \ (i = 1, 2), \end{aligned}$$

together with the extension for evaluation contexts:

$$\mathcal{E} ::= \dots \mid \text{bind } x = \mathcal{E} \text{ in } e \mid \text{val}(\mathcal{E}).$$

According to linearity, we do not allow evaluation contexts $\mathcal{E} \sqcap e$ and $e \sqcap \mathcal{E}$.

The \sqcap operator behaves like the internal choice in CSP [15]. We can also add the external choice operator \square , together with rules

$$\begin{aligned} e_1 \square e_2 &\rightsquigarrow e'_1 \square e_2, \text{ where } e_1 \rightsquigarrow e'_1, \\ e_1 \square e_2 &\rightsquigarrow e_1 \square e'_2, \text{ where } e_2 \rightsquigarrow e'_2. \end{aligned}$$

In accord with linearity, the typing rule for \square will be different from that of \sqcap :

$$\frac{\Gamma; \Delta_1 \vdash e_1 : \mathbb{T}\tau \quad \Gamma; \Delta_2 \vdash e_2 : \mathbb{T}\tau}{\Gamma; \Delta_1, \Delta_2 \vdash e_1 \square e_2 : \mathbb{T}\tau}$$

Our later development only considers the internal choice operator, but it can be easily adapted to languages with the external choice, with careful treatment of the reduction which can discard linear variables.

Canonical terms of NLPCF, besides the canonical terms of LPCF, now include terms of the form $\text{val}(v)$ where $v \not\rightsquigarrow$. The propositions about canonical form and subject reduction still hold.

Proposition 4. *If e is a NLPCF program and $e \not\rightsquigarrow$, then e must be in canonical form.*

Proposition 5. *In NLPCF, if $\Gamma; \Delta \vdash e : \tau$ and $e \rightsquigarrow e'$, then $\Gamma; \Delta \vdash e' : \tau$.*

The reduction system for NLPCF is non-deterministic and a term does not necessarily reduce to a unique value even if it converges — there is no confluence property in NLPCF. For any closed term e , we say

- e may converge (written as $e \Downarrow$) if there exists a value v such that $e \rightsquigarrow^* v \rightsquigarrow$;
- e must converge (written as $e \Downarrow$) if there is no infinite reduction starting from e , i.e., a reduction of e always terminates;
- e may diverge (written as $e \Uparrow$) if e has an infinite reduction sequence $e \rightsquigarrow e_1 \rightsquigarrow e_2 \rightsquigarrow \dots$;
- e must diverge (written as $e \Uparrow$) if there is no value v such that $e \rightsquigarrow^* v \rightsquigarrow$, i.e., e never reduces to a value.

4.2. Labeled transition system

The labeled transition system for NLPCF is extended by the following rule:

$$\frac{\Gamma; \Delta \vdash \text{val}(e) : \top \tau \quad e \not\rightsquigarrow}{\text{val}(e) \xrightarrow{\top} e}$$

The rule describes how programs of monadic types interact with contexts.

As in LPCF, we can define trace, trace preorder (written as \sqsubseteq^{NT}) and trace equivalence (written as \simeq^{NT}) for NLPCF.

Example 1. *Consider the two programs f_1 and f_2 in Section 1. Both of them have, among many others, the trace $\langle \top, @e, \top, 1 \rangle$ because of the following inference*

$$\begin{array}{ll} f_1 & \equiv \text{val}(\lambda x. \text{val}(0) \sqcap \text{val}(1)) & f_2 & \equiv \text{val}(\lambda x. \text{val}(0)) \sqcap \text{val}(\lambda x. \text{val}(1)) \\ & \xrightarrow{\top} \lambda x. \text{val}(0) \sqcap \text{val}(1) & & \rightsquigarrow \text{val}(\lambda x. \text{val}(1)) \\ & \xrightarrow{@e} (\text{val}(0) \sqcap \text{val}(1))[e/x] & & \xrightarrow{\top} \lambda x. \text{val}(1) \\ & \equiv \text{val}(0) \sqcap \text{val}(1) & & \xrightarrow{@e} \text{val}(1)[e/x] \\ & \rightsquigarrow \text{val}(1) & & \equiv \text{val}(1) \\ & \xrightarrow{\top} 1 & & \xrightarrow{\top} 1 \\ & \xrightarrow{1} \Omega, & & \xrightarrow{1} \Omega. \end{array}$$

The definition of linear context is as in LPCF, so correspondingly we have the following linear context transitions:

$$\mathcal{C}[\text{bind } z = x \text{ in } e/y] \circ \xrightarrow{\top} \mathcal{C}[(\lambda z. e)x'/y]$$

where x' is a fresh variable. The linear context transition lemma still holds:

Lemma 9 (Linear context transition lemma in NLPCF). *For every linear context $\mathcal{C}_{x:\tau}$ and NLPCF program $e \in \text{Prog}^{NL}(\tau)$ such that $\mathcal{C}[e/x] \not\rightsquigarrow$, a transition from $\mathcal{C}[e/x]$ must be in one of the two forms:*

- $\mathcal{C}[e/x] \xrightarrow{\alpha} \mathcal{C}'[e/x]$ with $\mathcal{C} \xrightarrow{\alpha} \mathcal{C}'$;
- $\mathcal{C} \equiv x$ and $\mathcal{C}[e/x] \equiv e \xrightarrow{\alpha} e'$.

PROOF . Similar to the proof of Lemma 4. □

4.3. Linear contextual equivalence in NLPCF

The Morris-style contextual equivalence depends on the notion of convergence, but in NLPCF, we need to choose between the may and must notions of convergence.

The notions of convergence/divergence in NLPCF accordingly leads to the following notions of equivalence relations of programs. Let $e_1, e_2 \in \text{Prog}^{NL}(\tau)$ for arbitrary type τ ,

- $e_1 \simeq^{\Downarrow} e_2$: $e_1 \Downarrow$ if and only if $e_2 \Downarrow$;
- $e_1 \simeq^{\Downarrow\Downarrow} e_2$: $e_1 \Downarrow\Downarrow$ if and only if $e_2 \Downarrow\Downarrow$;
- $e_1 \simeq^{\Uparrow} e_2$: $e_1 \Uparrow$ if and only if $e_2 \Uparrow$;
- $e_1 \simeq^{\Uparrow\Uparrow} e_2$: $e_1 \Uparrow\Uparrow$ if and only if $e_2 \Uparrow\Uparrow$.

It can be easily checked that $\simeq^{\Downarrow} = \simeq^{\Uparrow\Uparrow}$ and $\simeq^{\Uparrow} = \simeq^{\Downarrow\Downarrow}$.

Must convergence equivalence $\simeq^{\Downarrow\Downarrow}$ does not coincide with trace equivalence in a non-deterministic language. If the reduction relation is deterministic or confluent, we can conclude that a term converges as long as it has non-empty traces. However this is not true for must convergence in a non-deterministic language. By observing the traces of a term we can no longer tell whether a term has a non-terminating reduction sequence, since every term can take the empty trace, which by itself can represent divergence. In the contrary, if a term has *only* the empty trace, then we can conclude that the term must diverge.

The linear contextual equivalence in NLPCF is defined based on the notion of may convergence.

Definition 6 (Non-deterministic linear contextual equivalence). *For any type τ , we write $e_1 \sqsubseteq_{\tau}^{NC} e_2$ for $e_1, e_2 \in \text{Prog}^{NL}(\tau)$ if $\mathcal{C}[e_1/x] \Downarrow$ implies $\mathcal{C}[e_2/x] \Downarrow$ for all linear context $\mathcal{C}_{x:\tau}$. The relation \sqsubseteq^{NC} is called non-deterministic linear contextual preorder. Non-deterministic linear contextual equivalence \simeq^{NC} is defined as the symmetrization of \sqsubseteq^{NC} , that is, $e_1 \simeq_{\tau}^{NC} e_2$ iff $e_1 \sqsubseteq_{\tau}^{NC} e_2$ and $e_2 \sqsubseteq_{\tau}^{NC} e_1$.*

The definition of linear context reductions remains the same as in LPCF, except that we are considering the extended transition system for NLPCF. The linear context reduction lemma still holds, from which the precongruence of trace preorder follows, which in turn enables us to prove the soundness of trace preorder with respect to linear contextual equivalence in NLPCF.

Lemma 10 (Linear context reduction lemma in NLPCF). *For every linear context $\mathcal{C}_{x:\tau}$ and NLPCF program $e \in \text{Prog}^{NL}(\tau)$, if $\mathcal{C}[e/x]$ is reducible, then $\mathcal{C}[e/x] \rightsquigarrow$ must be a linear context reduction.*

PROOF . The proof goes as in Lemma 6, by induction on the structure of linear context \mathcal{C} . We show only the cases for new constructs.

- $\mathcal{C} \equiv \text{bind } y = \mathcal{C}' \text{ in } e'$. This is an evaluation context, so if $\mathcal{C}'[e/x]$ reduces, it must be a linear context reduction, then $\mathcal{C}[e/x] \rightsquigarrow$ is a linear context reduction of the same form as of $\mathcal{C}'[e/x] \rightsquigarrow$. If $\mathcal{C}'[e/x]$ does not reduce, which must be canonical of the form $\text{val}(\dots)$, there are two cases:

- $\mathcal{C}' \equiv \text{val}(\mathcal{C}'')$ and $\mathcal{C}''[e/x] \not\rightsquigarrow$. Then

$$\begin{aligned} \mathcal{C}[e/x] &\equiv \text{bind } y = \text{val}(\mathcal{C}''[e/x]) \text{ in } e' \\ &\rightsquigarrow (\lambda y. e') \mathcal{C}''[e/x] \\ &\equiv ((\lambda y. e') \mathcal{C}'')[e/x] \quad (\text{because } x \text{ does not appear freely in } \lambda y. e') \end{aligned}$$

with

$$\mathcal{C} \equiv \text{bind } y = \text{val}(\mathcal{C}'') \text{ in } e' \rightsquigarrow (\lambda y. e') \mathcal{C}'',$$

which is a linear context. The reduction is the first form of LCR.

- $\mathcal{C}' \equiv x$ and $e \equiv \text{val}(e'')$ ($e'' \not\rightsquigarrow$). In this case,

$$\mathcal{C}[e/x] \equiv \text{bind } y = \text{val}(e'') \text{ in } e' \rightsquigarrow (\lambda y. e') e''.$$

It is clear that $\mathcal{C} \equiv \text{bind } y = x \text{ in } e' \circ \xrightarrow{\text{T}} (\lambda y. e') x'$ and $e \equiv \text{val}(e'') \xrightarrow{\text{T}} e''$, so the reduction is the third form of LCR.

- $\mathcal{C} \equiv \text{bind } y = e' \text{ in } \mathcal{C}'$. If $e' \rightsquigarrow e''$, then $\mathcal{C}[e'/x] \rightsquigarrow \text{bind } y = e'' \text{ in } \mathcal{C}'[e/x]$ with $\mathcal{C} \rightsquigarrow \text{bind } y = e'' \text{ in } \mathcal{C}'$. If e' does not reduce, it must be of the form $\text{val}(e'')$, then $\mathcal{C}[e'/x] \rightsquigarrow \mathcal{C}'[e/x][e''/y] \equiv \mathcal{C}'[e''/y][e/x]$, with $\mathcal{C} \rightsquigarrow \mathcal{C}'[e''/y]$, which is a linear context since e'' is closed. In both cases, the reduction is the first form of LCR.
- $\mathcal{C} \equiv \mathcal{C}_1 \sqcap \mathcal{C}_2$. Clearly, both \mathcal{C}_1 and \mathcal{C}_2 are linear contexts, then

$$\mathcal{C}[e/x] \equiv \mathcal{C}_1[e/x] \sqcap \mathcal{C}_2[e/x] \rightsquigarrow \mathcal{C}_i[e/x], \quad (i = 1, 2),$$

with $\mathcal{C} \rightsquigarrow \mathcal{C}_i$. The reduction is the first form of LCR.

- $\mathcal{C} \equiv \text{val}(\mathcal{C}')$. Clearly \mathcal{C}' is a linear context and $\mathcal{C}'[e/x] \rightsquigarrow$ is a LCR, so $\mathcal{C}[e/x] \rightsquigarrow$ is also a LCR of the same form as $\mathcal{C}'[e/x] \rightsquigarrow$. \square

Theorem 4 (Linear precongruence of \sqsubseteq^{NT}). *Trace preorder \sqsubseteq^{NT} is a precongruence with respect to linear contexts, i.e., $e_1 \sqsubseteq^{NT} e_2$ implies that $\mathcal{C}[e_1/x] \sqsubseteq^{NT} \mathcal{C}[e_2/x]$ for all linear contexts \mathcal{C}_x in NLPCF.*

Theorem 4 can be shown by similar arguments in the proof of Theorem 1 that employs context traces, as we anticipated in Remark 1.

Theorem 5 (Soundness of \simeq^{NT}). *In NLPCF, it holds that $\simeq^{NT} \subseteq \simeq^{NC}$.*

PROOF . Assume that $e_1, e_2 \in \mathcal{P}rog^{NL}(\tau)$ are two programs of NLPCF such that $e_1 \simeq^{NT} e_2$. By precongurence, for every linear context $\mathcal{C}_{x:\tau}$, $\mathcal{C}[e_1/x] \simeq^{NT} \mathcal{C}[e_2/x]$. If $\mathcal{C}[e_1/x] \Downarrow$, i.e., $\mathcal{T}r(\mathcal{C}[e_1/x])$ has non-empty traces, then $\mathcal{T}r(\mathcal{C}[e_2/x])$ has non-empty traces too, hence $\mathcal{C}[e_2/x] \Downarrow$. Similarly, if $\mathcal{C}[e_2/x] \Downarrow$, then $\mathcal{C}[e_1/x] \Downarrow$. \square

The above theorem allows us to prove the equivalence of the two functions in Example 1: it is easy to check that both functions have traces $\langle \top, @e, \top, 0 \rangle$ and $\langle \top, @e, \top, 1 \rangle$ (where e is an arbitrary closed NLPCF term of proper type) as well as their subtraces, and they have no other traces.

4.4. Completeness of trace equivalence in NLPCF

The rest of the section is devoted to proving the completeness of trace equivalence with respect to linear contextual equivalence in NLPCF. Unlike the proof of Theorem 3, an induction over the length of traces does not work in a non-deterministic language, therefore we propose a novel proof for completeness.

We begin with constructing *trace-specific* linear contexts which “recognize” the corresponding traces. Given a trace s , we define the s -context $\mathcal{C}_{x:\tau}^s$ by induction on s :

$$\begin{aligned}
\mathcal{C}_{x:\tau}^\varepsilon &\stackrel{\text{def}}{=} \text{val}(x) \\
\mathcal{C}_{x:\text{Nat}}^n &\stackrel{\text{def}}{=} \text{if } x = n \text{ then val}(0) \text{ else } \Omega_{\top\text{Nat}} \\
\mathcal{C}_{x:\text{Bool}}^{\text{true}} &\stackrel{\text{def}}{=} \text{if } x \text{ then val}(0) \text{ else } \Omega_{\top\text{Nat}} \\
\mathcal{C}_{x:\text{Bool}}^{\text{false}} &\stackrel{\text{def}}{=} \text{if } x \text{ then } \Omega_{\top\text{Nat}} \text{ else val}(0) \\
\mathcal{C}_{x:\tau \rightarrow \tau'}^{@e \cdot s} &\stackrel{\text{def}}{=} \text{bind } y = \text{val}(xe) \text{ in } \mathcal{C}_{y:\tau'}^s, \text{ where } \emptyset; \emptyset \vdash e : \tau \\
\mathcal{C}_{x:\tau_1 \& \tau_2}^{\text{proj}_i \cdot s} &\stackrel{\text{def}}{=} \text{bind } y = \text{val}(\text{proj}_i(x)) \text{ in } \mathcal{C}_{y:\tau_i}^s \\
\mathcal{C}_{x:\tau_1 \otimes \tau_2}^{\otimes e \cdot s} &\stackrel{\text{def}}{=} \text{bind } y = \text{val}(\text{let } z_1 \otimes z_2 = x \text{ in } e) \text{ in } \mathcal{C}_{y:\tau'}^s, \\
&\quad \text{where } \emptyset; z_1 : \tau_1, z_2 : \tau_2 \vdash e : \tau' \\
\mathcal{C}_{x:\top\tau}^{\top \cdot s} &\stackrel{\text{def}}{=} \text{bind } y = x \text{ in } \mathcal{C}_{y:\tau}^s
\end{aligned}$$

It can be easily checked that $\emptyset; x : \tau \vdash \mathcal{C}_{x:\tau}^s : \top\tau'$ for some type τ' , if x is a linear variable, and we call it a linear s -context. In particular, if s is a computational trace then τ' is Nat . We shall often omit the type information when it is obvious or irrelevant.

In the definition we do not consider traces $c \cdot s$ with boolean/integer constant c followed by a non-empty trace s , because a valid trace must be taken by a program, while a program that takes the c -transition must be c itself, which no longer takes any external action after the transition ($c \xrightarrow{c} \Omega$).

The following two lemmas show that a program can take a computational trace s if and only if the corresponding linear s -context, when filled in with the program, *may converge*.

Lemma 11. *For every NLPCF program e and every computational trace s , if $e \xrightarrow{s}$ then $\mathcal{C}_x^s[e/x] \Downarrow$, for linear s -context \mathcal{C}_x^s .*

PROOF . Let $e \in \mathcal{P}rog^{NL}(\tau)$ be an arbitrary NLPCF program. We prove by induction on the length of s .

- $s = c$, where c is a boolean or integer constant. We show the case of integer constant; the proof for the boolean constant is similar. If e has the trace $c \cdot s'$, i.e., $e \rightsquigarrow^* c$ and $s' = \epsilon$, it follows that

$$\begin{aligned} \mathcal{C}_x^s[e/x] &\equiv \text{if } e = c \text{ then val}(0) \text{ else } \Omega \\ &\rightsquigarrow^* \text{if } c = c \text{ then val}(0) \text{ else } \Omega \\ &\rightsquigarrow^* \text{val}(0) \Downarrow. \end{aligned}$$

- $s \equiv @e' \cdot s'$. If e has the trace $@e' \cdot s'$, i.e.,

$$e \rightsquigarrow^* \lambda z. e_1 \xrightarrow{@e'} e_1[e'/z] \rightsquigarrow^* e'' \xrightarrow{s'} \Downarrow,$$

with e' a closed term of proper type and $e'' \not\rightsquigarrow$, then it follows that

$$\begin{aligned} \mathcal{C}_x^s[e/x] &\equiv \text{bind } y = \text{val}(e e') \text{ in } \mathcal{C}_y^{s'} \\ &\rightsquigarrow^* \text{bind } y = \text{val}((\lambda z. e_1)e') \text{ in } \mathcal{C}_y^{s'} \\ &\rightsquigarrow \text{bind } y = \text{val}(e_1[e'/z]) \text{ in } \mathcal{C}_y^{s'} \\ &\rightsquigarrow^* \text{bind } y = \text{val}(e'') \text{ in } \mathcal{C}_y^{s'} \\ &\rightsquigarrow^* \mathcal{C}_y^{s'}[e''/y] \end{aligned}$$

Since s' is a shorter trace than s , by induction, we know from $e'' \xrightarrow{s'} \Downarrow$ that $\mathcal{C}_y^{s'}[e''/y] \Downarrow$, therefore $\mathcal{C}_x^s[e/x] \Downarrow$.

- $s = \text{proj}_1 \cdot s'$. If e has the trace $\text{proj}_1 \cdot s'$, i.e.,

$$e \rightsquigarrow^* \langle e_1, e_2 \rangle \xrightarrow{\text{proj}_1} e_1 \rightsquigarrow^* e'_1 \xrightarrow{s'} \Downarrow,$$

with $e'_1 \not\rightsquigarrow$, then it follows that

$$\begin{aligned} \mathcal{C}_x^s[e/x] &\equiv \text{bind } y = \text{val}(\text{proj}_1(e)) \text{ in } \mathcal{C}_y^{s'} \\ &\rightsquigarrow^* \text{bind } y = \text{val}(\text{proj}_1(\langle e_1, e_2 \rangle)) \text{ in } \mathcal{C}_y^{s'} \\ &\rightsquigarrow \text{bind } y = \text{val}(e_1) \text{ in } \mathcal{C}_y^{s'} \\ &\rightsquigarrow^* \text{bind } y = \text{val}(e'_1) \text{ in } \mathcal{C}_y^{s'} \\ &\rightsquigarrow^* \mathcal{C}_y^{s'}[e'_1/y] \end{aligned}$$

Since s' is a shorter trace than s , by induction, we know from $e'_1 \xrightarrow{s'} \Downarrow$ that $\mathcal{C}_y^{s'}[e'_1/y] \Downarrow$, therefore $\mathcal{C}_x^s[e/x] \Downarrow$.

The case $s = \text{proj}_2 \cdot s'$ is similar.

- $s \equiv \otimes e' \cdot s'$. If e has the trace $\otimes e' \cdot s'$, i.e.,

$$e \rightsquigarrow^* e_1 \otimes e_2 \xrightarrow{\otimes e'} e'[e_1/z_1, e_2/z_2] \rightsquigarrow^* e'' \xrightarrow{s'} \Downarrow, \quad (3)$$

with $\emptyset; z_1 : \tau_1, z_2 : \tau_2 \vdash e' : \tau'$ ($\tau \equiv \tau_1 \otimes \tau_2$) and $e'' \not\rightsquigarrow$, then it follows that

$$\begin{aligned} \mathcal{C}_x^s[e/x] &\equiv \text{bind } y = \text{val}(\text{let } z_1 \otimes z_2 = e \text{ in } e') \text{ in } \mathcal{C}_y^{s'} \\ &\rightsquigarrow^* \text{bind } y = \text{val}(\text{let } z_1 \otimes z_2 = e_1 \otimes e_2 \text{ in } e') \text{ in } \mathcal{C}_y^{s'} \\ &\rightsquigarrow \text{bind } y = \text{val}(e'[e_1/z_1, e_2/z_2]) \text{ in } \mathcal{C}_y^{s'} \\ &\rightsquigarrow^* \text{bind } y = \text{val}(e'') \text{ in } \mathcal{C}_y^{s'} \\ &\rightsquigarrow^* \mathcal{C}_y^{s'}[e''/y] \end{aligned}$$

Since s' is a shorter trace than s , by induction, we know from $e'' \xRightarrow{s'}$ that $\mathcal{C}_y^{s'}[e''/y] \Downarrow$, therefore $\mathcal{C}_x^s[e/x] \Downarrow$.

- $s = T \cdot s'$. If e has the trace $T \cdot s'$, i.e.,

$$e \rightsquigarrow^* \text{val}(e') \xrightarrow{T} e' \xRightarrow{s'},$$

with $e' \not\rightsquigarrow$, then it follows that

$$\begin{aligned} \mathcal{C}_x^s[e/x] &\equiv \text{bind } y = e \text{ in } \mathcal{C}_y^{s'} \\ &\rightsquigarrow^* \text{bind } y = \text{val}(e') \text{ in } \mathcal{C}_y^{s'} \\ &\rightsquigarrow^* \mathcal{C}_y^{s'}[e'/y] \end{aligned}$$

Since s' is a shorter trace than s , by induction, we know from $e' \xRightarrow{s'}$ that $\mathcal{C}_y^{s'}[e'/y] \Downarrow$, therefore $\mathcal{C}_x^s[e/x] \Downarrow$. \square

Lemma 12. For any $e \in \text{Prog}^{NL}(\tau)$ and trace s , if $\mathcal{C}_x^s[e/x] \Downarrow$ then $e \xRightarrow{s}$.

PROOF. We prove by induction over the length of s , with an NLPCF program e .

- $s = \epsilon$. It is clear that $\epsilon \in \mathcal{Tr}(e)$.
- $s = c$, where c is a boolean or integer constant. Assume that c is an integer (the case of booleans is similar). Since $\mathcal{C}_x^s[e/x] \equiv \text{if } e = c \text{ then val}(0) \text{ else } \Omega \Downarrow$, it must hold that e may converge and $e \rightsquigarrow^* c$, hence $e \rightsquigarrow^* c \xrightarrow{c}$.
- $s = @e' \cdot s'$, with e' a closed term of proper type. Since

$$\mathcal{C}_x^{@e' \cdot s'}[e/x] \equiv \text{bind } y = \text{val}(e e') \text{ in } \mathcal{C}_y^{s'} \Downarrow,$$

there must be a reduction sequence

$$\begin{aligned} \mathcal{C}_x^{@e' \cdot s'}[e/x] &\rightsquigarrow^* \text{bind } y = \text{val}((\lambda z.e_1)e') \text{ in } \mathcal{C}_y^{s'} \text{ (where } e \rightsquigarrow^* \lambda z.e_1) \\ &\rightsquigarrow \text{bind } y = \text{val}(e_1[e'/z]) \text{ in } \mathcal{C}_y^{s'} \\ &\rightsquigarrow^* \text{bind } y = \text{val}(e'') \text{ in } \mathcal{C}_y^{s'} \text{ (where } e_1[e'/z] \rightsquigarrow^* e'' \text{ and } e'' \not\rightsquigarrow) \\ &\rightsquigarrow^* \mathcal{C}_y^{s'}[e''/y] \end{aligned}$$

and $\mathcal{C}_y^{s'}[e''/y] \Downarrow$, which implies that $e'' \xRightarrow{s'}$ by induction. Clearly, e may converge and $e \rightsquigarrow^* \lambda z.e_1 \xrightarrow{@e'} e_1[e'/z] \rightsquigarrow^* e'' \xRightarrow{s'}$, i.e., $e \xRightarrow{s}$.

- $s = \text{proj}_1 \cdot s'$. Since

$$\mathcal{C}_x^{\text{proj}_1 \cdot s'}[e/x] \equiv \text{bind } y = \text{val}(\text{proj}_1(e)) \text{ in } \mathcal{C}_y^{s'} \Downarrow,$$

there must be a reduction sequence

$$\begin{aligned} \mathcal{C}_x^{\text{proj}_1 \cdot s'}[e/x] &\rightsquigarrow^* \text{bind } y = \text{val}(\text{proj}_1(\langle e_1, e_2 \rangle)) \text{ in } \mathcal{C}_y^{s'} \text{ (where } e \rightsquigarrow^* \langle e_1, e_2 \rangle) \\ &\rightsquigarrow \text{bind } y = \text{val}(e_1) \text{ in } \mathcal{C}_y^{s'} \\ &\rightsquigarrow^* \text{bind } y = \text{val}(e'_1) \text{ in } \mathcal{C}_y^{s'} \text{ (where } e_1 \rightsquigarrow^* e'_1 \text{ and } e'_1 \not\rightsquigarrow) \\ &\rightsquigarrow^* \mathcal{C}_y^{s'}[e'_1/y] \end{aligned}$$

and $\mathcal{C}_y^{s'}[e'_1/y] \Downarrow$, which implies that $e'_1 \xrightarrow{s'} \Downarrow$ by induction. Clearly, e may converge and $e \rightsquigarrow^* \langle e_1, e_2 \rangle \xrightarrow{\text{proj}_1} e_1 \rightsquigarrow^* e'_1 \xrightarrow{s'} \Downarrow$, i.e., $e \xrightarrow{s} \Downarrow$.

The case $s \equiv \text{proj}_2 \cdot s'$ is similar.

- $s = \otimes e' \cdot s'$. Since

$$\mathcal{C}_x^{\otimes e' \cdot s'}[e/x] \equiv \text{bind } y = \text{val}(\text{let } z_1 \otimes z_2 = e \text{ in } e') \text{ in } \mathcal{C}_y^{s'} \Downarrow,$$

there must be a reduction sequence

$$\begin{aligned} \mathcal{C}_x^{\otimes e' \cdot s'}[e/x] &\rightsquigarrow^* \text{bind } y = \text{val}(\text{let } z_1 \otimes z_2 = e_1 \otimes e_2 \text{ in } e') \text{ in } \mathcal{C}_y^{s'} \\ &\quad (\text{where } e \rightsquigarrow^* e_1 \otimes e_2) \\ &\rightsquigarrow \text{bind } y = \text{val}(e'[e_1/z_1, e_2/z_2]) \text{ in } \mathcal{C}_y^{s'} \\ &\rightsquigarrow^* \text{bind } y = \text{val}(e'') \text{ in } \mathcal{C}_y^{s'} \\ &\quad (\text{where } e'[e_1/z_1, e_2/z_2] \rightsquigarrow^* e'' \text{ and } e'' \not\rightsquigarrow) \\ &\rightsquigarrow^* \mathcal{C}_y^{s'}[e''/y] \end{aligned}$$

and $\mathcal{C}_y^{s'}[e''/y] \Downarrow$, which implies that $e'' \xrightarrow{s'} \Downarrow$ by induction. Clearly, e may converge and $e \rightsquigarrow^* e_1 \otimes e_2 \xrightarrow{\otimes e'} e'[e_1/z_1, e_2/z_2] \rightsquigarrow^* e'' \xrightarrow{s'} \Downarrow$, i.e., $e \xrightarrow{s} \Downarrow$.

- $s = \text{T} \cdot s'$. Since

$$\mathcal{C}_x^{\text{T} \cdot s'}[e/x] \equiv \text{bind } y = e \text{ in } \mathcal{C}_y^{s'} \Downarrow,$$

there must be a reduction sequence

$$\begin{aligned} \mathcal{C}_x^{\text{T} \cdot s'}[e/x] &\rightsquigarrow^* \text{bind } y = \text{val}(e') \text{ in } \mathcal{C}_y^{s'} \quad (\text{where } e \rightsquigarrow^* \text{val}(e') \text{ and } e' \not\rightsquigarrow) \\ &\rightsquigarrow^* \mathcal{C}_y^{s'}[e'/y] \end{aligned}$$

and $\mathcal{C}_y^{s'}[e'/y] \Downarrow$, which implies that $e' \xrightarrow{s'} \Downarrow$ by induction. Clearly, e may converge and $e \rightsquigarrow^* \text{val}(e') \xrightarrow{\text{T}} e' \xrightarrow{s'} \Downarrow$, i.e., $e \xrightarrow{s} \Downarrow$. \square

The next two lemmas act as the counterparts of the previous two, but our focus now is on traces that are not computational.

Lemma 13. *If an NLPCF program e has the trace $s \cdot \alpha$ with $e \xrightarrow{s} e' \xrightarrow{\alpha} \Downarrow$ and $e' \not\rightsquigarrow$, then $\mathcal{C}_x^s[e/x] \rightsquigarrow^* \text{val}(e')$.*

PROOF . We first note that s is not a computational trace. Otherwise the program e' derived from a computational trace would be Ω , which cannot make an external action α , a contradiction to the hypothesis that $e' \xrightarrow{\alpha} \Downarrow$.

Let $e \in \text{Prog}^{NL}(\tau)$ be an arbitrary NLPCF program. Similar to the proof of Lemma 11, we prove by induction on the length of s . Here we only consider two example cases.

- $s \equiv \epsilon$. Clearly, it always holds that $e \xrightarrow{\epsilon} e$ and $\mathcal{C}_x^\epsilon[e/x] \equiv \text{val}(e) \rightsquigarrow^* \text{val}(e)$.

- $s \equiv @_{e_1} \cdot s'$. If e has the trace $@_{e_1} \cdot s'$, i.e.,

$$e \rightsquigarrow^* \lambda z. e_2 \xrightarrow{@_{e_1}} e_2[e_1/z] \rightsquigarrow^* e'' \xRightarrow{s'} e',$$

with e_1 a closed term of proper type and $e'' \not\rightsquigarrow$, then it follows that

$$\begin{aligned} \mathcal{C}_x^s[e/x] &\equiv \text{bind } y = \text{val}(e \ e_1) \text{ in } \mathcal{C}_y^{s'} \\ &\rightsquigarrow^* \text{bind } y = \text{val}((\lambda z. e_2)e_1) \text{ in } \mathcal{C}_y^{s'} \\ &\rightsquigarrow \text{bind } y = \text{val}(e_2[e_1/z]) \text{ in } \mathcal{C}_y^{s'} \\ &\rightsquigarrow^* \text{bind } y = \text{val}(e'') \text{ in } \mathcal{C}_y^{s'} \\ &\rightsquigarrow^* \mathcal{C}_y^{s'}[e''/y] \end{aligned}$$

Since s' is a shorter trace than s , by induction, we know from $e'' \xRightarrow{s'} e' \xrightarrow{\alpha}$ that $\mathcal{C}_y^{s'}[e''/y] \rightsquigarrow^* \text{val}(e')$, therefore $\mathcal{C}_x^s[e/x] \rightsquigarrow^* \text{val}(e')$ by transitivity of the relation \rightsquigarrow^* . \square

Lemma 14. *For every NLPCF program $e \in \text{Prog}^{NL}(\tau)$ and trace s that is not computational, if $\mathcal{C}_x^s[e/x] \Downarrow$ then there is some program e' such that $e \xRightarrow{s} e'$ and $e' \Downarrow$.*

PROOF . Similar to the proof of Lemma 12. We prove by induction over the length of s , with an NLPCF program e . Let us consider two cases.

- $s = \epsilon$. Then $\mathcal{C}_x^s[e/x] \equiv \text{val}(e) \Downarrow$. It means that $e \Downarrow$. Clearly we also have $e \xRightarrow{\epsilon} e$.
- $s = T \cdot s'$. Since

$$\mathcal{C}_x^{T \cdot s'}[e/x] \equiv \text{bind } y = e \text{ in } \mathcal{C}_y^{s'} \Downarrow,$$

there must be a reduction sequence

$$\begin{aligned} \mathcal{C}_x^{T \cdot s'}[e/x] &\rightsquigarrow^* \text{bind } y = \text{val}(e'') \text{ in } \mathcal{C}_y^{s'} \quad (\text{where } e \rightsquigarrow^* \text{val}(e'') \text{ and } e'' \not\rightsquigarrow) \\ &\rightsquigarrow^* \mathcal{C}_y^{s'}[e''/y] \end{aligned}$$

and $\mathcal{C}_y^{s'}[e''/y] \Downarrow$, which implies that $e'' \xRightarrow{s'} e'$ and $e' \Downarrow$ by induction. Therefore, $e \rightsquigarrow^* \text{val}(e'') \xrightarrow{T} e'' \xRightarrow{s'} e'$, i.e., $e \xRightarrow{s} e'$. \square

Theorem 6 (Completeness of \simeq^{NT}). *In NLPCF, it holds that $\simeq^{NC} \subseteq \simeq^{NT}$.*

PROOF . Assume that e_1, e_2 are two programs and $e_1 \simeq^{NC} e_2$. Suppose $e_1 \xRightarrow{s} e'$ for some trace s . We distinguish two cases.

- s is a computational trace. By Lemma 11, we have that $\mathcal{C}_x^s[e_1/x] \Downarrow$. Since $e_1 \simeq^{NC} e_2$, it must be the case that $\mathcal{C}_x^s[e_2/x] \Downarrow$. By Lemma 12, it follows that $e_2 \xRightarrow{s} e'$.

- s is not a computational trace. If $s = \epsilon$, we obviously have $e_2 \xrightarrow{s}$. Now suppose that $s = s' \cdot \alpha$, that is $e_1 \xrightarrow{s'} e'_1 \xrightarrow{\alpha}$ for some $e'_1 \not\sim$. By Lemma 13 we have $\mathcal{C}_x^{s'}[e_1/x] \rightsquigarrow^* \text{val}(e')$, which means that $\mathcal{C}_x^{s'}[e_1/x] \Downarrow$. Since $e_1 \simeq^{NC} e_2$, it must be the case that $\mathcal{C}_x^{s'}[e_2/x] \Downarrow$. By Lemma 14, there is some e'_2 such that $e_2 \xrightarrow{s'} e'_2$ and $e'_2 \Downarrow$. By Lemma 2, which also holds for NLPCF, we see that e'_2 has the same type as e'_1 . Since s is not a computational trace, α must be in one of the forms $@e$, proj_i , $\otimes e$ or \top . Depending on the type of e'_1 , in each case there exists some e''_2 such that $e'_2 \not\sim$ and $e'_2 \rightsquigarrow^* e''_2 \xrightarrow{\alpha}$. It follows that $e_2 \xrightarrow{s'} e'_2 \rightsquigarrow^* e''_2 \xrightarrow{\alpha}$, that is $e_2 \xrightarrow{s}$.

Symmetrically, any trace of e_2 is also a trace of e_1 . Therefore, we obtain that $e_1 \simeq^{NT} e_2$. \square

5. Concluding remarks

We have presented a novel approach for characterizing program equivalence in linear contexts, via trace equivalence in appropriate labeled transition systems. The technique is both sound and complete, and as we have shown in the paper, is general enough to be adapted for languages with linear type systems.

Linear contextual equivalence is indeed a restricted notion of program equivalence and one may question its use in practice. As we have explained in the beginning of the paper, it does have applications in security as we can use linearity to limit adversaries' behavior. We also believe that such a notion of program equivalence can be useful in reasoning about programs in systems where only restricted access to resources is allowed, particularly when side effects are present. The result in non-deterministic languages already enables us to prove linear contextual equivalence between non-trivial programs.

We have used both program transitions and context transitions to model the interactions between programs and contexts, and the program/context traces (if combined in a proper way) resemble strategies in game semantics [2, 20] despite our operational treatment of traces. However, it is unclear whether the correspondence can be made between program/context actions in the trace model and player/opponent moves in the game model — the exact connection remains to clarify.

Acknowledgements

We thank the anonymous referees for their helpful comments.

References

- [1] S. Abramsky. The lazy lambda calculus. *Research Topics in Functional Programming*. Addison-Wesley 1990.
- [2] S. Abramsky, G. McCusker. Call-by-value games. In *Proc. of the 11th International Workshop on Computer Science Logic (CSL)*, LNCS 1414. Springer, 1998.

- [3] A. Barber. Dual intuitionistic linear logic. Research report ECS-LFCS-96-347, University of Edinburgh, 1996.
- [4] S. Bellantoni, S. A. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97-110, 1992.
- [5] M. Bundgaard, T. Hildebrandt, J.C. Godskesen. Typing linear and non-linear higher-order mobile embedded resources with local names. Technical Report TR-2007-97, IT University of Copenhagen, 2007.
- [6] G.M. Bierman. Program equivalence in a linear functional language. *J. Funct. Program.* 10(2): 167-190. Cambridge University Press 2000.
- [7] G.M. Bierman, A.M. Pitts, C.V. Russo. Operational properties of Lily, a polymorphic linear lambda calculus with recursion. *Electr. Notes Theor. Comput. Sci.* 41(3): 70-88. 2000.
- [8] I. Cervesato, F. Pfenning. A Linear Logical Framework. *Inf. Comput.* 179(1): 19-75. 2002
- [9] R.L. Crole. Completeness of Bisimilarity for Contextual Equivalence in Linear Theories. *Logic Jnl IGPL* 9(1):27-51, 2001.
- [10] Xinyu Feng. Local rely-guarantee reasoning. In *Proc. of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* , pp. 315-327, ACM 2009.
- [11] O. Goldreich. *The Foundations of Cryptography: Basic Tools*. Cambridge University Press, 2001.
- [12] O. Goldreich. *The Foundations of Cryptography: Basic Applications*. Cambridge University Press, 2004.
- [13] A.D. Gordon. A Tutorial on Co-induction and Functional Programming. In *Glasgow Workshop on Functional Programming*, pp. 78-95. Springer, 1995.
- [14] J. Goubault-Larrecq, S. Lasota, D. Nowak. Logical relations for monadic types. *Mathematical Structures in Computer Science*, 18(6): 1169-1217. Cambridge University Press 2008.
- [15] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [16] M. Hofmann. Safe recursion with higher-types and BCK-algebra. *Annals of Pure and Applied Logic*, 104(1-3): 113-166. 2000.
- [17] M. Hofmann. A Type System for Bounded Space and Functional In-Place Update. *Nord. J. Comput.*, 7(4): 258-289. 2000.
- [18] M. Hofmann. Linear types and non-size increasing polynomial time computation. *Inf. Comput.* 183(1): 57-85. 2003.
- [19] D.J. Howe. Proving congruence of bisimulation in functional programming languages. *Inf. Comput.* 124(2): 103-112. 1996.
- [20] J.M.E. Hyland, C.H.L. Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.* 163(2): 285-408, 2000.

- [21] A.S.A. Jeffrey. A Fully Abstract Semantics for a Non-deterministic Functional Language with Monadic Types. *Theor. Comput. Sci.* 228: 105-150. 1999.
- [22] N. Kobayashi, B.C. Pierce, D.N. Turner. Linearity and the Pi-Calculus *ACM Transactions on Programming Languages and Systems* 21(5):914-947, 1999.
- [23] S. Lasota, D. Nowak, Y. Zhang. On Completeness of Logical Relations for Monadic Types. In *Advances in Computer Science - ASIAN 2006*, LNCS 4435. Springer, 2008.
- [24] J.C. Mitchell. *Foundations for Programming Languages*. MIT press, 1996.
- [25] E. Moggi. Notions of computation and monads. *Inf. Comput.* 93(1): 55-92, 1991.
- [26] D. Nowak, Y. Zhang. A calculus for game-based security proofs. In *Proc. of the 4th International Conference on Provable Security (ProvSec)*, LNCS 6402. Springer, 2010.
- [27] A. M. Pitts. Operationally-Based Theories of Program Equivalence. *Semantics and Logics of Computation*, pp. 241-298. Cambridge University Press, 1997.
- [28] A. M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10: 321-359. Cambridge University Press, 2000.
- [29] A. M. Pitts. Typed operational reasoning. *Advanced Topics in Types and Programming Languages*, chapter 7. The MIT Press, 2005.
- [30] G.D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.* 5: 223-255. 1977.
- [31] G.D. Plotkin. Lambda definability in the full type hierarchy. *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.
- [32] G.D. Plotkin. Second order type theory and recursion. Notes for a talk at Scott Fest, 1993.
- [33] D. Sangiorgi, N. Kobayashi and E. Sumii. Environmental bisimulations for higher-order languages. *ACM Trans. Program. Lang. Syst.* 33(1): 5. ACM 2011.
- [34] R. Statman. Logical relations and the typed lambda calculus. *Information and Control*, 65: 85-97, 1985.
- [35] V. Vafeiadis, M. Parkinson. A marriage of rely/guarantee and separation logic. In *Proc. of the 18th International Conference on Concurrency Theory (CONCUR)*, LNCS, vol. 4703. Springer, 2007.
- [36] D. Walker, K. Watkins. On Regions and Linear Types. In *Proc. of the 6th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM 2001.
- [37] Q. Xu, W.P. de Roever, J. He. The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs. *Formal Asp. Comput.* 9(2): 149-174, 1997.
- [38] N. Yoshida, K. Honda, M. Berger. Linearity and Bisimulation. *J. Log. Algebr. Program.* 72(2): 207-238, 2007.
- [39] Y. Zhang. The computational SLR: a logic for reasoning about computational indistinguishability. *Mathematical Structures in Computer Science*, 20(5): 951-975. Cambridge University Press 2010.