The Universal Process

Yuxi Fu

BASICS, Department of Computer Science
Shanghai Jiaotong University, Shanghai 200030, China

September 4, 2014

Abstract

A universal process of a process calculus is one that, given the Gödel index of a process of a certain type, produces a process equivalent to the encoded process. This paper demonstrates how universal processes can be formally defined and how a universal process of the value-passing calculus can be constructed. The existence of such a universal process in a process model can be explored to implement higher order communications, security protocols, and programming languages in the process model. A process version of the S-m-n theorem is stated to showcase how to embed the recursion theory in a process calculus.

1 Introduction

The classic recursion theory [Rog87, Soa87] is based on two fundamental observations. The first is that there is an effective function ϕ^k that enumerates all the k-ary recursive functions. By fixing an enumeration function we can write ϕ^k_i for $\phi^k(i)$, the i-th k-ary recursive function. The number i is called the *Gödel number*, or the *Gödel index* of the recursive function. The effectiveness of ϕ^k_i comes in both directions. One can effectively calculate a unique number from a given recursive function. One can also effectively recover a unique recursive function from a given number. The S-m-n Theorem states that for all k_0, k_1 there is a total $(k_0{+}1)$-ary recursive function $\mathsf{s}^{k_0}_{k_1}(z, x_1, \ldots, x_{k_0})$ such that $\phi^{k_0+k_1}_k(i_1, \ldots, i_{k_0}, j_1, \ldots, j_{k_1}) \simeq \phi^{k_1}_{\mathsf{s}^{k_0}_{k_1}(k,i_1,\ldots,i_{k_0})}(j_1, \ldots, j_{k_1})$ for all numbers $k, i_1, \ldots, i_{k_0}, j_1, \ldots, j_{k_1}$. The equality $\simeq$ means that either both sides are defined and they are equal or neither side is defined. The second important observation is that there exists a $(k{+}1)$-ary universal function $\mathcal{U}^k$ that, upon receiving an index j of a k-ary recursive function f and k numbers $i_1, \ldots, i_k$, evaluates $\mathsf{f}(i_1, \ldots, i_k)$. In other words, $\mathcal{U}^k(j, i_1, \ldots, i_k) \simeq \phi^k_j(i_1, \ldots, i_k)$. The existence of such a universal function depends crucially on Gödelization. It is by Gödelization that we can see a number both as a datum and a program. The S-m-n Theorem and the universal functions are the foundational tools in recursion theory. The practical counterpart of a universal function is a general purpose computer. The central idea of the von Neumann structure of such a computer is that of the stored program, which is essentially the same thing

as Gödelization. From the point of view of programming, a universal function is an interpreter that works by interpreting a datum as a program. Again this is the idea of Gödelization.

Recursion theory plays a foundational role in computation theory and the theory of programming languages. It makes one think why in the theory of process calculus, or more generally in concurrency theory, the fundamental technique of Gödelization has not been utilized so far. One possible explanation is that concurrent computations are often distributed. For processes scattered at different locations the notion of a centralized universal process may sound alien. In retrospect however, the absence of any universal process has been unfortunate. The π-calculus [MPW92], and CCS [Mil89] as well, was proposed with the intention to be the 'λ-calculus' for concurrent computation. Yet in the theory of process calculus there still lacks a notion comparable to that of decidability/undecidability. There are now some interesting techniques that allow one to prove negative results in say π-calculus [BGZ03, BGZ04, Pal03, GSV04, FL10]. However they do not offer a method as general as the reduction method in recursion theory. To develop a theory of solvability/unsolvability for the π-calculus, the ideas and the techniques of recursion theory are instructive. In programming theory, there have been quite a few papers on implementing variants of π, or substantial extensions of them, on current computing platforms. But there has been little discussion on how to implement a concurrent programming language in the π-calculus. It's understandably so since the idea of a universal process (or general interpreter) is indispensable in any such implementation. If we are serious about the promise that the π-calculus is to concurrent computation what the λ-calculus is to functional computation, we should look at implementation issues of concurrent programming languages in the π-calculus.

The above discussion leads to the conclusion that in both theory and practice there is a genuine need for a process theory that goes beyond the classic recursion theory of function. The theory of process calculus currently fails to meet that need. What can we do to improve the situation? A natural thing to do is to look at how Gödelization can be carried out in process calculi and how universal processes can be constructed. Gödelization *is* a problem for a process calculus that cannot even code up the natural numbers in a way that supports the interpretations of the computable functions within the calculus. We need to confine our attention to complete models. Intuitively a complete process calculus is one that is expressive enough to admit good use of Gödelization. Now suppose $\mathbb{M}$ is a complete model. What does a universal process of $\mathbb{M}$ look like? In the general case it is unlikely that there is a single $\mathbb{M}$-process capable of simulating all $\mathbb{M}$-processes. A π-process for example only refers to a finite number of global names. There is no way for it to simulate a π-process that uses strictly more global names. We must accept that a universal process of a process calculus should consist of a countable family of processes. Luckily we seldom need a single all powerful universal process. In most applications it suffices to have a collection of processes, each acting as a universal process for a set of processes of a certain type. A type for example could be a finite set of names. Then a process is of that type if the names it contains all appear in that set. If we think of it, having to use a restricted version of universal process does not really stop us from deriving any solvability/unsolvability results in $\mathbb{M}$. If something is solvable in $\mathbb{M}$, it is solvable by an $\mathbb{M}$-process of some type. If it is not solvable, it is not solved by any $\mathbb{M}$-process of any type.

We will look at Gödelization and the notion of universal process in $\mathbb{VPC}$, a self-contained version of the value-passing calculus. The reason to start with this particular model is that it is closer to recursion theory than all the other process calculi [Fu13]. The contribution of this paper is the introduction of a formal definition of universal process and the construction of a universal process for $\mathbb{VPC}$. The significance of the existence of a universal process is emphasized by illustrating a number of applications. The technique developed in this work is expected to play a key role in the study of process theory and programming theory implemented on process models.

The paper is structured as follows. Section 2 reviews the necessary background on $\mathbb{VPC}$ and the observational theory of processes. Section 3 provides the formal definition of universal process and demonstrates how to construct a universal process in $\mathbb{VPC}$. Section 4 outlines three major applications of universal process. Section 5 formalizes the process version of S-m-n Theorem. Section 6 discusses some future research topics.

Before engaging in the technicalities in the rest of the paper, we should comment on the presentation style of this paper. We shall not spell out all technical details of our constructions, and consequently nor shall we formally establish the correctness of the constructions. We will make full use of the fact that $\mathbb{VPC}$ is complete. This is very much like what recursion theoreticians make use of Church-Turing Thesis since the publication of Post's pioneering paper [Pos44]. If one has not built up enough confidence in exploiting the completeness of process models this way, one is advised to consult [Fu13, Fu14b, Fu14a, FL10] in which sufficient technical details can be found.

2 Preliminary

In this section we define the semantics of the value-passing calculus, fix the notion of process equality used in this paper, and explain in what sense the value-passing calculus is complete.

2.1 VPC

Value-passing calculi [Hoa85, Mil89, HI93a, HI93b, HL95] have been studied in various contexts. In most of these studies, the value domains are left open-ended. A recent work that provides a self-contained account of the value-passing calculi is [Fu13]. Since our value-passing calculus is going to be the source model whose programs are to be interpreted by a universal process, an open-ended attitude is inadequate. At the same time we hope to avoid the formality of [Fu13] for clarity. Fortunately there is a standard theory we can refer to. The value domain of our value-passing calculus is taken to be Presburger Arithmetic [Pre29] (an English translation of the original paper can be found in [Sta84]). This is the sub-theory of Peano Arithmetic defined by the constant 0, the unary function s and the binary function '+'. By overloading notations, we shall abbreviate $\mathsf{s}^k(0)$ to k and $\mathsf{s}^k(x)$ to $x + k$. For our purpose the most attractive property of Presburger Arithmetic is the decidability of its first order theory. There is a terminating procedure that decides the validity of every first order formula of Presburger Arithmetic [Pre29, Mon76, End01]. This is a crucial property if a value-passing

calculus is seen as a programming model. The absence of the multiplication operator does not affect the power of our model since the operator can be implemented in the value-passing calculus [Fu13].

Let N be the set $\{0, \mathsf{s}(0), \mathsf{s}^2(0), \ldots\}$, ranged over by i, j, k, and V be the set of natural number variables, ranged over by x, y, z. The set T of *value terms*, ranged over by s, t, is constructed from the numbers, the variables, and the binary operator '+'. The notation T^0 stands for the set of closed terms. The set B of first order *logical formulae*, ranged over by φ, consists of the formulas constructed from the terms, the logical operators $\bot, \top, \wedge, \vee, \Rightarrow, \exists, \forall$ and the binary relations $<, =$. We write $\vdash \varphi$ if φ is a theorem of Presburger Arithmetic.

Let $\mathcal{N}$ be the set of names, ranged over by a, b, c, d, e, f, g, h. The set of the finite $\mathbb{VPC}$-terms is defined by the following BNF:

$$T \quad := \quad \mathbf{0} \mid a(x).T \mid \overline{a}(t).T \mid T \mid T \mid (c)T \mid \textit{if } \varphi \textit{ then } T.$$

The $\mathbb{VPC}$-processes, denoted by P, Q, are the $\mathbb{VPC}$-terms that contain no free variables. The name c in $(c)T$ is a local name. A name is global if it is not local. The semantics of the finite $\mathbb{VPC}$-terms is given by the following labeled transition system, where α ranges over the action set $\{a(i), \overline{a}(i) \mid a \in \mathcal{N}, i \in \mathsf{N}\} \cup \{\tau\}$.

Action

$$\frac{}{a(x).T \xrightarrow{a(i)} T\{i/x\}} \qquad \frac{}{\overline{a}(t).T \xrightarrow{\overline{a}(i)} T} \vdash t = i.$$

Composition

$$\frac{S \xrightarrow{\alpha} S'}{S \mid T \xrightarrow{\alpha} S' \mid T} \qquad \frac{S \xrightarrow{a(i)} S' \quad T \xrightarrow{\overline{a}(i)} T'}{S \mid T \xrightarrow{\tau} S' \mid T'}$$

Localization

$$\frac{T \xrightarrow{\alpha} T'}{(c)T \xrightarrow{\alpha} (c)T'} \; c \text{ is not in } \alpha.$$

Condition

$$\frac{T \xrightarrow{\alpha} T'}{\textit{if } \varphi \textit{ then } T \xrightarrow{\alpha} T'} \vdash \varphi.$$

We shall use standard notations like $\Longrightarrow$ and $\overset{\alpha}{\Longrightarrow}$. The recursion mechanism of a value-passing calculus can be defined in a number of ways. They are not completely equivalent in terms of expressive power [BGZ03, BGZ04, Pal03, GSV04, FL10]. The infinite behaviors of our model $\mathbb{VPC}$ is introduced by equationally defined terms. A *parametric definition* is given by the equation

$$D(x_1, \ldots, x_k) = T, \tag{1}$$

where $x_1, \ldots, x_k$ are parameter variables. In this paper we require that T does not contain any free variable not in $\{x_1, \ldots, x_k\}$. The instantiation of $D(x_1, \ldots, x_k)$ at value

terms $t_1, \ldots, t_k$, denoted by $D(t_1, \ldots, t_k)$, is $T\{t_1/x_1, \ldots, t_k/x_k\}$. In addition to the finite terms, $\mathbb{VPC}$ also has instantiated terms of the form $D(t_1, \ldots, t_k)$, where $t_1, \ldots, t_k$ are value terms. The parametric definition (1) is generally recursive in the sense that T may contain instantiated occurrences of $D(x_1, \ldots, x_k)$. It may also contain instantiated occurrences of some $D'(y_1, \ldots, y_j)$ given by another parametric definition. The operational semantics of $D(t_1, \ldots, t_k)$ is defined by the following rule:

$$\frac{T\{t_1/x_1, \ldots, t_k/x_k\} \xrightarrow{\alpha} T'}{D(t_1, \ldots, t_k) \xrightarrow{\alpha} T'} \; D(x_1, \ldots, x_k) = T.$$

Now suppose $D(x) = \overline{c}(0) \,|\, (c)(\overline{c}(x) \,|\, \overline{c}(x) \,|\, c(z).D(z{+}1))$. Then the following reductions are admissible:

$$\begin{aligned} D(1) &\xrightarrow{\tau} \overline{c}(0) \,|\, (c)(\overline{c}(1) \,|\, D(1+1)) \\ &\xrightarrow{\tau} \overline{c}(0) \,|\, (c)(\overline{c}(1) \,|\, \overline{c}(0) \,|\, (c)(\overline{c}(2) \,|\, D(2{+}1))). \end{aligned}$$

In this example the global name c in the component $\overline{c}(0)$ gets captured every time the parametric definition is unfolded.

An alternative to parametric definition is replication. The syntax for the replication terms is given by

$$T \quad := \quad \ldots \mid\, !a(x).T \mid\, !\overline{a}(t).T.$$

The operational semantics of the replicator is defined by the following transitions:

$$\frac{}{!a(x).T \xrightarrow{a(i)} T\{i/x\} \,|\, !a(x).T} \qquad \frac{}{!\overline{a}(t).T \xrightarrow{\overline{a}(i)} T \,|\, !\overline{a}(t).T} \; \vdash t = i.$$

We will denote by $\mathbb{VPC}^!$ the value-passing calculus with the replicator.

The replicator is a derived operator in $\mathbb{VPC}$. The term $!a(x).T$ for example is equal to the instantiation $D(x_1, \ldots, x_k)$ where

$$D(x_1, \ldots, x_k) = D(x_1, \ldots, x_k) \,|\, a(x).T$$

and $\{x_1, \ldots, x_k\}$ is the set of the free variables appearing in $a(x).T$. We shall freely use the replication operator in $\mathbb{VPC}$. The calculus $\mathbb{VPC}^!$ cannot simulate everything in $\mathbb{VPC}$ though [Fu13]. In the latter one can implement a recursive algorithm in a top-to-bottom fashion. This is often not the case in the former. The best $\mathbb{VPC}^!$ can do is to use stack explicitly to manage the recursive calls in a bottom-up manner. The important thing for us is that all recursive functions can be implemented in $\mathbb{VPC}^!$ [Fu13].

The following abbreviations will be used

$$\begin{aligned} a.T &\stackrel{\text{def}}{=} a(x).T, \;\; \text{where } x \text{ does not appear in } T, \\ \overline{a}.T &\stackrel{\text{def}}{=} \overline{a}(0).T. \end{aligned}$$

We occasionally write for example $t(x)$ to indicate that t contains the variable x. Accordingly we write $t(s)$ for the term obtained by substituting s for x. The notations

$\varphi(x), \varphi(s)$ and $T(x), T(s)$ are used similarly. We sometimes use the two leg if command defined as follows:

$$\textit{if } \varphi \textit{ then } S \textit{ else } T \stackrel{\text{def}}{=} \textit{if } \varphi \textit{ then } S \mid \textit{if } \neg\varphi \textit{ then } T.$$

For clarity we will write

case t **of**
$\varphi_0(z) \Rightarrow T_0(z)$;
$\vdots$
$\varphi_{k-1}(z) \Rightarrow T_{k-1}(z)$;
$\varphi_k(z) \Rightarrow T_k(z)$
end case

for the nested if statement *if* $\varphi_0(t)$ *then* $T_0(t)$ *else if* $\ldots$ *else if* $\varphi_k(t)$ *then* $T_k(t)$. The auxiliary notation *let* $x = t$ *in* T stands for $T\{t/x\}$. This is useful when t is a long expression and x occurs in T several times.

For a complete treatment of $\mathbb{VPC}$ the reader should consult [Fu13].

2.2 Equality and Expressiveness

The definition of a universal process must refer to a process equality. The choice of such an equality is not entirely orthogonal to the existence of a universal process. It is conceivable that some sort of universal process exists with respect to a weak equality, whereas it is impossible to have a universal process with respect to a stronger equality. To present our result in its strongest form, we shall introduce a number of properties that we believe best describe the *correctness* of our universal processes. The following account follows the general methodology of [Fu14b]. The description given here is however self-contained. In this section we assume that $\mathbb{M}$ is a process calculus and $\mathcal{R}$ is a binary relation on the set of $\mathbb{M}$-processes. The notation $\mathcal{R}^{-1}$ will stand for the reverse relation of $\mathcal{R}$.

A universal process is a generalization of a universal Turing machine. Upon receiving a number the latter simulates the Turing machine encoded by the number. But how about the correctness of the simulation? The answer is provided by the operational interpretation of the Church-Turing Thesis [vEB90]. A sound translation of one computation model to another is a bisimulation of computation steps *à la* Milner [Mil89] and Park [Par81]. Moreover if we take nondeterministic computation into account the translation ought to be a branching bisimulation of van Glabbeek and Weijland [vGW89]. The reader is referred to [Fu14a] for a formal study of nondeterministic computation in a process algebraic setting.

Definition 1. *$\mathcal{R}$ is a* bisimulation *if the following clauses are valid:*

1. *If $Q\mathcal{R}^{-1}P \xrightarrow{\tau} P'$ then one of the following statements is valid:*
   (i) *$Q \Longrightarrow Q'$ for some Q' such that $Q'\mathcal{R}^{-1}P$ and $Q'\mathcal{R}^{-1}P'$.*

(ii) $Q \Longrightarrow Q''\mathcal{R}^{-1}P$ *for some* Q'' *such that* $\exists Q'.Q'' \xrightarrow{\tau} Q'\mathcal{R}^{-1}P'$.

2. *If* $P\mathcal{R}Q \xrightarrow{\tau} Q'$ *then one of the following statements is valid:*

   (i) $P \Longrightarrow P'$ *for some* P' *such that* $P'\mathcal{R}Q$ *and* $P'\mathcal{R}Q'$.

   (ii) $P \Longrightarrow P''\mathcal{R}Q$ *for some* P'' *such that* $\exists P'.P'' \xrightarrow{\tau} P'\mathcal{R}Q'$.

A universal process must be sensitive to divergence. It would be unacceptable to interpret all processes by divergent processes. The following definition is from [Pri78]. It is the best formalization of the termination preserving property that goes along with the bisimulations [vGLT09, Fu14b].

Definition 2. $\mathcal{R}$ *is* codivergent *if the following statements are valid:*

1. *If* $P\mathcal{R}Q \xrightarrow{\tau} Q_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} Q_i \xrightarrow{\tau} \ldots$, *then* $\exists P'.\exists k \geq 1.P \overset{\tau}{\Longrightarrow} P'\mathcal{R}Q_k$.

2. *If* $Q\mathcal{R}^{-1}P \xrightarrow{\tau} P_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} P_i \xrightarrow{\tau} \ldots$, *then* $\exists Q'.\exists k \geq 1.Q \overset{\tau}{\Longrightarrow} Q'\mathcal{R}^{-1}P_k$.

A universal process should also respect interactability. The barbedness of Milner and Sangiorgi [MS92] poses a minimal condition. We say that a process P is observable, notation $P\Downarrow$, if $\Longrightarrow\!\!\xrightarrow{\alpha}$ for some $\alpha \neq \tau$. It is unobservable, notation $P\not\Downarrow$, if it is not observable.

Definition 3. $\mathcal{R}$ *is* equipollent *if* $P\Downarrow \Leftrightarrow Q\Downarrow$ *whenever* $P\mathcal{R}Q$.

The equipollence condition does not make much sense unless some kind of closure property is available. Concurrent composition and localization are the most fundamental operators in concurrency theory. The former makes global interaction possible while the latter localizes such possibility. Hence the following definition.

Definition 4. $\mathcal{R}$ *is* extensional *if the following statements are valid:*

1. *If* $P\mathcal{R}Q$, *then* $(c)P\mathcal{R}(c)Q$ *for every* $c \in \mathcal{N}$;

2. *If* $P_0\mathcal{R}Q_0$ *and* $P_1\mathcal{R}Q_1$, *then* $(P_0 \mid P_1)\mathcal{R}(Q_0 \mid Q_1)$.

In [Fu14b] it is argued that these properties give a model-independent characterization of process equality.

Definition 5. *The absolute equality* $=_{\mathbb{M}}$ *is the largest relation on the set of the* $\mathbb{M}$*-processes that satisfies the following:*

1. *It is reflexive;*

2. *It is extensional, equipollent, codivergent and bisimilar.*

It is easy to convince oneself that $=_{\mathbb{M}}$ is well defined. So we have $=_{\mathbb{VPC}}$ and $=_{\mathbb{VPC}^!}$. We will often omit the subscript.

The abstract definition of $=_{\mathbb{M}}$ makes it difficult to work with. We need a characterization of $=_{\mathbb{M}}$ that relies neither on the equipollence condition nor on the extensionality condition. In practice it is sufficient to have an external bisimilarity $\simeq_{\mathbb{M}}$ satisfying $\simeq_{\mathbb{M}} \subseteq =_{\mathbb{M}}$.

Definition 6. *A codivergent bisimulation $\mathcal{R}$ is an $\mathbb{M}$-bisimulation if whenever $Q\mathcal{R}P \xrightarrow{\alpha} P'$ for some $\alpha \neq \tau$ then $Q \Longrightarrow Q'' \xrightarrow{\alpha} Q'\mathcal{R}P'$ and $P\mathcal{R}Q''$ for some Q', Q''. The $\mathbb{M}$-bisimilarity $\simeq_{\mathbb{M}}$ is the largest $\mathbb{M}$-bisimulation.*

Both $\simeq_{\mathbb{VPC}} \subseteq =_{\mathbb{VPC}}$ and $\simeq_{\mathbb{VPC}^!} \subseteq =_{\mathbb{VPC}^!}$ hold. We shall make use of these facts in the correctness proofs.

By making use of the congruence = we can define semantically the one step *deterministic* computation $P \to P'$ as an internal action $P \xrightarrow{\tau} P'$ such that $P' = P$, and the one step *nondeterministic* computation $P \xrightarrow{\iota} P'$ as an internal action $P \xrightarrow{\tau} P'$ such that $P' \neq P$. The distinction between the two classes of internal actions is important to appreciate the working mechanism of the universal process. The reflexive and transitive closure of $\to$ is denoted by $\to^*$.

If we consider interpreter rather than universal process, we need to relate a process of one model to a process of another model. In other words we need to talk about 'equality' between the processes from two different process calculi. This way of looking at the expressiveness relationship between two process calculi leads immediately to the following definition [Fu14b].

Definition 7. *A binary relation $\propto$ from the set of $\mathbb{M}$-processes to the set of $\mathbb{N}$-processes is a* subbisimilarity *if it renders true the following statements:*

- *$\propto$ is total and sound;*
- *$\propto$ is extensional, equipollent, codivergent and bisimilar.*

$\mathbb{M}$ is subbisimilar to *$\mathbb{N}$, notation $\mathbb{M} \sqsubseteq \mathbb{N}$ or $\mathbb{N} \sqsupseteq \mathbb{M}$, if there is a subbisimilarity, a witness of $\mathbb{M} \sqsubseteq \mathbb{N}$, from $\mathbb{M}$ to $\mathbb{N}$.*

The reflexivity of Definition 5 is turned into the totality and soundness of Definition 7. Totality means that for each $\mathbb{M}$-process P there is an $\mathbb{N}$-process Q such that $P \propto Q$. The soundness is the condition that $Q =_{\mathbb{N}} Q'$ whenever $P =_{\mathbb{M}} P'$, $P \propto Q$ and $P' \propto Q'$. Intuitively $\mathbb{M} \sqsubseteq \mathbb{N}$ means that $\mathbb{N}$ is at least as expressive as $\mathbb{M}$. The relation $\sqsubseteq$ is stronger than most of the expressiveness relations discussed in the literature [BGZ03, BGZ04, Gor08, Pal03, Fu14b], which makes the correctness of our interpreter more convincing.

2.3 Completeness

Both $\mathbb{VPC}$ and $\mathbb{VPC}^!$ are Turing complete. There are several interpretations of Turing completeness in the literature on process calculus [BGZ04, MP05, FL10]. The general requirement on Turing completeness of a process model $\mathbb{M}$ can be summarized as follows:

- There is an encoding $[\![\_]\!]$ of the natural numbers in $\mathbb{M}$.
- There is an interpretation $[\![\_]\!]$ of recursive functions [Rog87] in $\mathbb{M}$ such that for every k-ary computable function $\mathsf{f}(x_1, \ldots, x_k)$ and all numbers $i_1, \ldots, i_k$ the following operational property holds: If $\mathsf{f}(i_1, \ldots, i_k)$ is defined then

$$[\![i_1]\!] \mid \ldots \mid [\![i_k]\!] \mid [\![\mathsf{f}(x_1, \ldots, x_k)]\!] \stackrel{\tau}{\Longrightarrow} \approx [\![\mathsf{f}(i_1, \ldots, i_k)]\!],$$

where $\overset{\tau}{\Longrightarrow}$ is the transitive closure of $\overset{\tau}{\longrightarrow}$; if $\mathsf{f}(i_1, \ldots, i_k)$ is undefined then

$$[\![i_1]\!] \mid \ldots \mid [\![i_k]\!] \mid [\![\mathsf{f}(x_1, \ldots, x_k)]\!] \overset{\tau}{\Longrightarrow} \approx \Omega,$$

where Ω is a divergent process whose only action is $\Omega \overset{\tau}{\longrightarrow} \Omega$, and $\approx$ is one of the termination preserving weak equalities. A criticism to this level of completeness is that the input numbers $[\![i_1]\!], \ldots, [\![i_k]\!]$ are not necessarily picked up properly by $[\![\mathsf{f}(x_1, \ldots, x_k)]\!]$, and the result number $[\![\mathsf{f}(x_1, \ldots, x_k)]\!]$ is not sent to any intended receiver. The evolution from say $[\![i_1]\!] \mid \ldots \mid [\![i_k]\!] \mid [\![\mathsf{f}(x_1, \ldots, x_k)]\!]$ to $[\![\mathsf{f}(i_1, \ldots, i_k)]\!]$ could be too liberal.

The Turing completeness of an interaction model $\mathbb{M}$ means that an *outsider* can see that the recursive functions can be coded up using $\mathbb{M}$-processes. It is an external completeness. A stronger notion of completeness, a much more useful one in practice, is internal completeness. Intuitively the internal completeness of $\mathbb{M}$ means that the *insiders*, the $\mathbb{M}$-processes, are aware of the fact that they can compute all the computable functions. A formal treatment of this kind of completeness is provided in [Fu14b]. In this paper it suffices to say that the completeness of $\mathbb{M}$ boils down to the following:

- For each name a and each number i there is an encoding $[\![i]\!]_a$ of i at a.
- For all $k \geq 0$ and all names $a_1, \ldots, a_k, b$, there is an encoding function $[\![\_]\!]^b_{a_1,\ldots,a_k}$ such that for every k-ary recursive function $\mathsf{f}(x_1, \ldots, x_k)$ the following statement is valid: For all natural numbers $i_1, \ldots, i_k$, $\mathsf{f}(i_1, \ldots, i_k)$ is defined if and only if

$$[\![i_1]\!]_{a_1} \mid \ldots \mid [\![i_k]\!]_{a_k} \mid [\![\mathsf{f}(x_1, \ldots, x_k)]\!]^b_{a_1,\ldots,a_k} \underbrace{\xrightarrow{\iota} \ldots \xrightarrow{\iota}}_{k \text{ times}} =_{\mathbb{M}} [\![\mathsf{f}(i_1, \ldots, i_k)]\!]_b,$$

  and $\mathsf{f}(i_1, \ldots, i_k)$ is undefined if and only if

$$[\![i_1]\!]_{a_1} \mid \ldots \mid [\![i_k]\!]_{a_k} \mid [\![\mathsf{f}(x_1, \ldots, x_k)]\!]^b_{a_1,\ldots,a_k} \underbrace{\xrightarrow{\iota} \ldots \xrightarrow{\iota}}_{k \text{ times}} =_{\mathbb{M}} \Omega.$$

The class $\{[\![i]\!]_a\}_{i \in \mathsf{N}, a \in \mathcal{N}}$ provides an encoding of the natural numbers in $\mathbb{M}$. The process $[\![i]\!]_a$ is ready to deliver the number i to a process at channel a. The process $[\![\mathsf{f}(x_1, \ldots, x_k)]\!]^b_{a_1,\ldots,a_k}$ inputs the numbers $i_1, \ldots, i_k$ sequentially at $a_1, \ldots, a_k$ in k steps, after which it becomes some process, say M, equal to $[\![\mathsf{f}(i_1, \ldots, i_k)]\!]_b$. The only action of $[\![\mathsf{f}(i_1, \ldots, i_k)]\!]_b$ is to deliver the result to whichever process wants a number at channel b. It should be remarked that M may perform a finite sequence of deterministic computations to simulate the computation of $\mathsf{f}(i_1, \ldots, i_k)$. All the intermediate states of this simulation are equal to each other.

Both $\mathbb{VPC}$ and $\mathbb{VPC}^!$ are complete in the above sense. In both models the encoding $[\![i]\!]_a$ is defined by $\overline{a}(i)$.

The practical implication of the completeness of $\mathbb{VPC}$ and $\mathbb{VPC}^!$ is that we may make use of a process without explicitly defining it. Let's explain this point by examples. Suppose $\mathsf{f}(x_1, \ldots, x_{k_0})$, $\mathsf{g}(y_1, \ldots, y_{k_1})$ are computable functions. Then we may assume that *if* $\mathsf{f}(x_1, \ldots, x_{k_0}) = \mathsf{g}(y_1, \ldots, y_{k_1})$ *then* T is a $\mathbb{VPC}$-term with the obvious semantics. In fact it can be defined by the following term

$$(c)(d)([\![\mathsf{f}(x_1, \ldots, x_{k_0})]\!]^c \mid [\![\mathsf{g}(y_1, \ldots, y_{k_1})]\!]^d \mid c(x).d(y).\mathit{if}\ x = y\ \mathit{then}\ T),$$

where $[\![\mathsf{f}(x_1,\ldots,x_{k_0})]\!]^c$ is the $\mathbb{VPC}$-process that outputs the value of $\mathsf{f}(x_1,\ldots,x_{k_0})$ at c, whose existence is guaranteed by the completeness of $\mathbb{VPC}$. The process $[\![\mathsf{g}(y_1,\ldots,y_{k_1})]\!]^d$ is similar. In the same fashion we may think of $\overline{a}(\mathsf{f}(x_1,\ldots,x_k)).T$ as the $\mathbb{VPC}$-term $(c)([\![\mathsf{f}(x_1,\ldots,x_k)]\!]^c \mid c(z).\overline{a}(z).T)$. More generally let $\psi(x_1,\ldots,x_k)$ be a semi-decidable property and $\chi_\psi(x_1,\ldots,x_k)$ be the partial characteristic function of ψ. According to definition $\chi_\psi(x_1,\ldots,x_k)$ is the recursive function that returns '1' at input sequence $i_1,\ldots,i_k$ when the property holds and diverges otherwise. Now we may regard *if* ψ *then* T the same as *if* $\chi_\psi = 1$ *then* T. If ψ is a decidable property, then *if* ψ *then* T *else* T' can be interpreted as

$$\textit{if } \chi_\psi = 1 \textit{ then } T \mid \textit{if } \chi_\psi \neq 1 \textit{ then } T'.$$

To simplify notation we shall use more liberal terms like

$$A(j).\textit{if } j = 1 \textit{ then } T(j). \tag{2}$$

In the above term the generalized prefix operation $A(j)$ is understood as an arithmetic operation. After the result j has been calculated, the term *if* $j = 1$ *then* $T(j)$ is ready to fire. By the completeness the term in (2) can be implemented in $\mathbb{VPC}$.

In the rest of the paper we shall use the internal completeness of $\mathbb{VPC}$ extensively in the manner just described.

For a systematic development of the equality theory, the expressiveness theory and the completeness theory from which the definitions given in Section 2.2 and Section 2.3 are imported, the reader is referred to [Fu14b]. The completeness of $\mathbb{VPC}$ is formally established in [Fu13].

3 Universal Process

This section presents the major contribution of this paper, which is to construct a universal process for $\mathbb{VPC}$. Since a universal process is a special case of an interpreter, we will first give a formal definition of the latter. We then define an interpreter of $\mathbb{VPC}^!$ in $\mathbb{VPC}$. Finally we modify the definition of the interpreter to produce the desired universal process of $\mathbb{VPC}$. We hope that this two step definition offers a clearer presentation of our methodology.

Suppose $\mathbb{L}, \mathbb{M}$ are complete models. We intend to formalize the relationship saying that $\mathbb{M}$ is capable of interpreting all the $\mathbb{L}$-processes *within* $\mathbb{M}$. Informally an *interpreter* of $\mathbb{L}$ in $\mathbb{M}$ is an $\mathbb{M}$-process such that after inputting a Gödel number of an $\mathbb{L}$-process it behaves like the $\mathbb{L}$-process represented by the number. A prerequisite for the existence of such an interpreter is that $\mathbb{M}$ should be at least as expressive as $\mathbb{L}$. This is because if $\mathbb{L} \not\propto \mathbb{M}$ then there is an $\mathbb{L}$-process whose interactive behavior cannot be simulated by any $\mathbb{M}$-processes. When this is the case there cannot be any interpreter of $\mathbb{L}$ in $\mathbb{M}$. We conclude that every interpreter of $\mathbb{L}$ in $\mathbb{M}$ is based on an expressiveness relation $\propto$ from $\mathbb{L}$ to $\mathbb{M}$.

What is expected of an interpreter? There is no point for it to simulate a term containing free variables. But it is expected to be able to manipulate bound variables since they only act as placeholders. An interpreter can deal with a finite number of global names. But no interpreter can store an infinite number of global names. The

issue concerning local names is more tricky. Different models have different naming policies. Some models admit dynamic creation of local names, others do not. So in general an interpreter must know the number of distinct local names appearing in a process in order to simulate it properly. Talking about the number of distinct names, we would like to emphasize that $(b)(\overline{a}b.\overline{b} \mid (b)(c)\overline{a}b.\overline{a}c.\overline{b}.\overline{c})$ contains two, not three, distinct local names, although semantically there are three local names. This static view is important for Gödelization. Let $\mathcal{N}^*$ be the set of finite lists of names, ranged over by $\jmath$. The notation $a \in \jmath$ means that a appears in $\jmath$. We have the following description of an interpreter:

> An interpreter of $\mathbb{L}$ in $\mathbb{M}$ is a family $\{\mathcal{I}_c^{i,J}\}_{i\in\mathsf{N},J\in\mathcal{N}^*,c\notin J}$ of $\mathbb{M}$-processes such that, for all $i \in \mathsf{N}$, $\jmath \in \mathcal{N}^*$ and $c \in \mathcal{N}$ such that $c \notin \jmath$, after picking up a Gödel number at channel c the process $\mathcal{I}_c^{i,J}$ can simulate all $\mathbb{L}$-processes that have at most i distinct local names and contain no more global names than those appearing in $\jmath$.

We will write $\mathcal{I}_c^{i,a_1\ldots a_k}$ if $\jmath$ is the list $a_1 \ldots a_k$. The superscript i is often omitted. The interpretation of a number by $\mathcal{I}_c^{i,J}$ differs from the interpretation of the same number by $\mathcal{I}_c^{i,J'}$ in that they have different interfaces. However $\mathcal{I}_c^{i,J}$ and $\mathcal{I}_c^{i',J}$ may produce the same interpretation of a number if the number encodes a process that has at most $\min\{i, i'\}$ local names.

Now suppose k is the Gödel index of an $\mathbb{L}$-process P whose set of global names is a subset of $\{a_1, \ldots, a_j\}$ and whose number of local names is no more than i. Let $\{[\![\_]\!]_c\}_{c\in\mathcal{N}}$ be an indexed encoding function of the natural numbers in $\mathbb{M}$. The process $\mathcal{I}_c^{i,a_1\ldots a_j}$ must satisfy the following property: There exists a unique Q such that

$$[\![k]\!]_c \mid \mathcal{I}_c^{i,a_1\ldots a_j} \xrightarrow{\iota} Q \propto^{-1} P, \tag{3}$$

where $\propto$ is the subbisimilarity the interpretation is based upon and $\propto^{-1}$ is the inverse relation of $\propto$. After a single step interaction with $[\![k]\!]_c$ the process $\mathcal{I}_c^{i,a_1\ldots a_j}$ becomes an $\mathbb{M}$-version of P under $\propto$. Since there may be many subbisimilarities from $\mathbb{L}$ to $\mathbb{M}$ and possibly infinite number of encodings of the natural numbers into $\mathbb{M}$, it is more precise to define an interpreter of $\mathbb{L}$ in $\mathbb{M}$ as the tuple $\langle\{\mathcal{I}_c^{i,J}\}_{i\in\mathsf{N},J\in\mathcal{N}^*,c\notin J}, [\![\_]\!], \propto\rangle$ that satisfies (3). This completes the formal definition of interpreter.

Let's write $\mathbb{L} \in \mathbb{M}$ if there is an interpreter of $\mathbb{L}$ in $\mathbb{M}$. We may think of $\mathbb{L} \in \mathbb{M}$ as an internal version, or a programming version, of $\mathbb{L} \sqsubseteq \mathbb{M}$. In the terminology of programming language, $\mathbb{L} \in \mathbb{M}$ says that $\mathbb{L}$ can be implemented in $\mathbb{M}$.

The distinction between a translation and an implementation should now be clear. A translation is a reduction from a source model to a target model. It is a meta theoretical operation. An implementation is a family of processes in the target model that is capable of reproducing a process of the source model at will.

3.1 Gödel Index

We avail ourselves of an effective bijective function

$$\langle \_, \ldots, \_ \rangle_k : \underbrace{\mathsf{N} \times \ldots \times \mathsf{N}}_{k \text{ times}} \to \mathsf{N},$$

$$
\begin{aligned}
[\![\mathbf{0}]\!]_{\mathrm{i}} &\stackrel{\text{def}}{=} 0,\\
[\![a(x).T]\!]_{\mathrm{i}} &\stackrel{\text{def}}{=} 7 * \langle \varsigma(a), \varsigma(x), [\![T]\!]_{\mathrm{i}} \rangle + 1,\\
[\![\overline{a}(t).T]\!]_{\mathrm{i}} &\stackrel{\text{def}}{=} 7 * \langle \varsigma(a), [\![t]\!]_{\varsigma}, [\![T]\!]_{\mathrm{i}} \rangle + 2,\\
[\![T \mid T']\!]_{\mathrm{i}} &\stackrel{\text{def}}{=} 7 * \langle [\![T]\!]_{\mathrm{i}}, [\![T']\!]_{\mathrm{i}} \rangle + 3,\\
[\![(c)T]\!]_{\mathrm{i}} &\stackrel{\text{def}}{=} 7 * \langle \varsigma(c), [\![T]\!]_{\mathrm{i}} \rangle + 4,\\
[\![\mathit{if}\ \varphi\ \mathit{then}\ T]\!]_{\mathrm{i}} &\stackrel{\text{def}}{=} 7 * \langle [\![\varphi]\!]_{\varsigma}, [\![T]\!]_{\mathrm{i}} \rangle + 5,\\
[\![!a(x).T]\!]_{\mathrm{i}} &\stackrel{\text{def}}{=} 7 * \langle \varsigma(a), \varsigma(x), [\![T]\!]_{\mathrm{i}} \rangle + 6,\\
[\![!\overline{a}(t).T]\!]_{\mathrm{i}} &\stackrel{\text{def}}{=} 7 * \langle \varsigma(a), [\![t]\!]_{\varsigma}, [\![T]\!]_{\mathrm{i}} \rangle + 7.
\end{aligned}
$$

Figure 1: Gödel Index of $\mathbb{VPC}^!$-Term.

whose inverse function is composed of the unary functions $(\_)_0, \ldots, (\_)_{k-1}$. For clarity we sometimes write for instance $z_{i,j}$ for $((z)_i)_j$ when no confusion arises. We assume that $\langle 0, 0, \ldots, 0 \rangle_k = 0$, and we often omit the subscript in $\langle \_, \ldots, \_ \rangle_k$. For convenience we assume that the unary pairing function is the identity function and the 0-ary pairing function is the constant 0.

By abusing notations, let ς denote both a bijective function from the set $\mathcal{N}$ of names to N and a bijective function from the set V of variables to N. Using ς as an oracle function, we can define an effective bijective function from the set T of terms to N and an effective bijective function from the set B of formulas to N. We will denote both by $[\![\_]\!]_{\varsigma}$ and omit the obvious structural definition.

Using the standard technique the Gödel number of a $\mathbb{VPC}^!$-term is defined by the function $[\![\_]\!]_{\mathrm{i}}$ introduced in Figure 1. The function $[\![\_]\!]_{\mathrm{i}}$ is a *bijection* between the set of the $\mathbb{VPC}^!$-terms and the set of the natural numbers. It should be emphasized that we prohibit the use of α-conversion when we are assigning Gödel numbers to the $\mathbb{VPC}^!$-terms. The encodings of say $a(x).a(x).\overline{b}(x)$ and $a(x).a(y).\overline{b}(y)$ are different, even though they are treated as syntactically the same term when α-conversion is admitted.

We get another set of Gödel indices if we use an oracle function different from ς. The definition of our interpreter does not depend on any particular choice of such a function. For a $\mathbb{VPC}^!$-process P using k global names $a_1, \ldots, a_k$ and i local names, a *normal index* of P is the one in which the global names $a_1, \ldots, a_k$ are indexed by $1, 2, \ldots, k$ and the local names are indexed by $k+1, k+2, \ldots, k+i$.

3.2 A VPC Interpreter

We now define an interpreter $\{\mathcal{I}_d^{i,J}\}_{i \in \mathsf{N}, J \in \mathcal{N}^*, d \notin J}$ of $\mathbb{VPC}^!$ in $\mathbb{VPC}$. The definition of $\mathcal{I}_d^{i,a_1 \ldots a_k}$ is given as follows:

$$
\mathcal{I}_d^{i,a_1 \ldots a_k} \quad = \quad d(z).(h)(\mathcal{P}_{\mathrm{i}}(z) \mid h(z).\mathcal{S}_{\mathrm{i}}(z)). \tag{4}
$$

The interpreter executes the two subroutines sequentially.

- If z is the Gödel number of a process, say A, of the right type, the *parser* $\mathcal{P}_{\mathrm{i}}(z)$ transforms z to a normal Gödel index z' that codes up a process α-convertible

case z **of**
$\mathsf{r}_7(z)=0 \;\Rightarrow\; c(x).(\mathit{if}\ x=1\ \mathit{then}\ \overline{e}\ \mathit{else}\ \overline{c}(x-1));$
$\mathsf{r}_7(z)=1 \;\Rightarrow\; \mathit{if}\ L_1(\mathsf{d}_7(z)_0)\ \mathit{then}\ \mathcal{G}_{\mathrm{i}}(\mathsf{d}_7(z)_2, \langle \mathsf{d}_7(z)_1+1, v\rangle);$
$\mathsf{r}_7(z)=2 \;\Rightarrow\; \mathit{if}\ L_2^1(\mathsf{d}_7(z)_1)\ \mathit{then}\ \mathit{if}\ L_2^0(\mathsf{d}_7(z)_0)\ \mathit{then}\ \mathcal{G}_{\mathrm{i}}(\mathsf{d}_7(z)_2, v);$
$\mathsf{r}_7(z)=3 \;\Rightarrow\; c(x).(\overline{c}(x+1)\,|\,\mathcal{G}_{\mathrm{i}}(\mathsf{d}_7(z)_0, v)\,|\,\mathcal{G}_{\mathrm{i}}(\mathsf{d}_7(z)_1, v));$
$\mathsf{r}_7(z)=4 \;\Rightarrow\; \mathit{if}\ L_4(\mathsf{d}_7(z)_0)\ \mathit{then}\ \mathcal{G}_{\mathrm{i}}(\mathsf{d}_7(z)_1, v);$
$\mathsf{r}_7(z)=5 \;\Rightarrow\; \mathit{if}\ L_5(\mathsf{d}_7(z)_0)\ \mathit{then}\ \mathcal{G}_{\mathrm{i}}(\mathsf{d}_7(z)_1, v);$
$\mathsf{r}_7(z)=6 \;\Rightarrow\; \mathit{if}\ L_6(\mathsf{d}_7(z)_0)\ \mathit{then}\ \mathcal{G}_{\mathrm{i}}(\mathsf{d}_7(z)_2, \langle \mathsf{d}_7(z)_1+1, v\rangle);$
$\mathsf{r}_7(z)=7 \;\Rightarrow\; \mathit{if}\ L_7^1(\mathsf{d}_7(z)_1)\ \mathit{then}\ \mathit{if}\ L_7^0(\mathsf{d}_7(z)_0)\ \mathit{then}\ \mathcal{G}_{\mathrm{i}}(\mathsf{d}_7(z)_2, v)$
end case.

Figure 2: Grammar Checker $\mathcal{G}_{\mathrm{i}}(z, v)$.

to A, and then releases z' through the channel h. If z is illegitimate the parser aborts the interpretation. In other words $\mathcal{I}_d^{i,a_1\ldots a_k}$ chooses to interpret the index of a process of a wrong type as an index for $\mathbf{0}$.

- The *simulator* $\mathcal{S}_{\mathrm{i}}(z')$ operates on the received normal Gödel number and simulates the process indexed by the number.

The interpretation makes use of the following recursive functions $\mathsf{r}_7, \mathsf{d}_7$:

$$\mathsf{r}_7(z) \stackrel{\text{def}}{=} \begin{cases} 0, & \text{if } z = 0, \\ i, & \text{if } 1 \le i \le 7 \text{ and } \exists j.z = 7 * j + i, \end{cases}$$

$$\mathsf{d}_7(z) \stackrel{\text{def}}{=} \begin{cases} 0, & \text{if } z = 0, \\ j, & \text{if } \exists i{\in}\{1,\ldots,7\}.z = 7 * j + i. \end{cases}$$

The operation carried out by the parser is purely arithmetical. So it can be implemented in $\mathbb{VPC}$. Let's however take a look at an outline of the following implementation:

$$(g_1 \ldots g_{k+i})(c)(e)(\prod_{j=1}^{k+i} \overline{g_j}(0)\,|\,\overline{c}(1)\,|\,\mathcal{G}_{\mathrm{i}}(z,0)\,|\,e.\mathcal{N}_{\mathrm{i}}(z,0)),$$

where $\prod_{j=1}^{k+i} \overline{g_j}(0)$ stands for $\overline{g_1}(0)\,|\,\ldots\,|\,\overline{g_{k+i}}(0)$. The *grammar checker* $\mathcal{G}_{\mathrm{i}}(z, v)$ is defined in Fig. 2. It aborts the interpreter if any one of the following happens:

- The number of the indices of global names is more than k.
- The number of the indices of local names is more than i.
- There is an index for a free variable.

At the name c is recorded the number of the concurrent components the parser has encountered. Initially there is only one component, which explains the presence of $\overline{c}(1)$. The parser ends successfully if in the end the value at c is 0. The names $g_1, \ldots, g_k$ are used to store the indices for the global names and the names $g_{k+1}, \ldots, g_{k+i}$ for the

local names. If j_1 is the first index for a local name $\mathcal{G}_{\mathrm{i}}(z, v)$ encounters, the grammar checker stores $j_1 + 1$ at g_{k+1}. This can be done by invoking $g_{k+1}(x).\overline{g_{k+1}}(j_1 + 1)$. Similarly if j_2 is the second index of a local name $\mathcal{G}_{\mathrm{i}}(z, v)$ encounters, then $\mathcal{G}_{\mathrm{i}}(z, v)$ stores $j_2 + 1$ at g_{k+2}. In completely the same fashion, the grammar checker stores the Gödel numbers that represent the global names at $g_1, \ldots, g_k$ in the order they are discovered. The second parameter v of $\mathcal{G}_{\mathrm{i}}(z, v)$ codes up the bound variables already discovered. If for example the bound variables are $x_1, \ldots, x_{k'}$ encountered in that order then v would be $\langle \varsigma(x_{k'}) + 1, \langle \varsigma(x_{k'-1}) + 1, \ldots, \langle \varsigma(x_1) + 1, 0\rangle \ldots\rangle\rangle$. The Boolean functions $L_1, L_2^1, L_2^0, L_4, L_5, L_6, L_7^1, L_7^0$ check if the number of names is under the limit or if all variables are bound. We now explain how $\mathcal{G}_{\mathrm{i}}(z, v)$ works.

- $\mathsf{r}_7(z) = 0$. If the number of concurrent components becomes zero, end $\mathcal{G}_i$ successfully and initiate the *normalizer* $\mathcal{N}_{\mathrm{i}}(z, 0)$.

- $\mathsf{r}_7(z) = 1$. The number $\mathsf{d}_7(z)_1$ is the index of a bound variable. The process $L_1(\mathsf{d}_7(z)_0)$ needs to make sure that if $\mathsf{d}_7(z)_0$ is neither the index of a local name nor an index of a global name that has been recorded before, then the number stored at g_k must be 0. If $L_1(\mathsf{d}_7(z)_0)$ succeeds, the number $\mathsf{d}_7(z)_0 + 1$ is stored at the least g_j that has not been used. After $L_1(\mathsf{d}_7(z)_0)$ has ended successfully, the number $\mathsf{d}_7(z)_1 + 1$ is added to the list of the indices of the bound variables already parsed and is passed down recursively.

- $\mathsf{r}_7(z) = 2$. The subroutine $L_2^1(\mathsf{d}_7(z)_1)$ checks if the term represented by the number $\mathsf{d}_7(z)_1$ contains an unknown variable. It aborts if it encounters an index that does not appear in the tuple encoded by v. If $L_2^1(\mathsf{d}_7(z)_1)$ succeeds, $L_2^0(\mathsf{d}_7(z)_0)$ checks the legitimacy of the encoding $\mathsf{d}_7(z)_0$. The process L_2^0 works in the same manner as the process L_1.

- $\mathsf{r}_7(z) = 3$. The counter at c is incremented by 1 since one concurrent component is split into two.

- $\mathsf{r}_7(z) = 4$. The subroutine $L_4(\mathsf{d}_7(z)_0)$ checks if the number $\mathsf{d}_7(z)_0 + 1$ is the same as the number stored at some g_j, where $k+1 \leq j \leq k+i$. If the answer is positive, $L_4(\mathsf{d}_7(z)_0)$ succeeds; otherwise it checks if the number at g_{k+i} is 0. If the answer to the latter query is negative, it aborts; otherwise it succeeds after it has stored the number $\mathsf{d}_7(z)_0 + 1$ at the appropriate g_j.

- $\mathsf{r}_7(z) = 5$. The subroutine $L_5(\mathsf{d}_7(z)_0)$ checks if the number $\mathsf{d}_7(z)_0$ codes up a well formed formula. Specifically it needs to make sure that the formula coded up by the number does not contain any free variable.

- $\mathsf{r}_7(z) = 6$, $\mathsf{r}_7(z) = 7$. These cases are similar to the cases $\mathsf{r}_7(z) = 1$, $\mathsf{r}_7(z) = 2$ respectively.

After $\mathcal{G}_{\mathrm{i}}(z, 0)$ has ended successfully, it starts the process $\mathcal{N}_{\mathrm{i}}(z, 0)$.

Using the values stored at g_j's, $\mathcal{N}_{\mathrm{i}}(z, 0)$ transforms Gödel index z to a normal Gödel index z' and then releases z' at h. An implementation of $\mathcal{N}_{\mathrm{i}}$ is given in Fig. 3. The operation $Find(j, w, y)$ returns as the value of y the j' such that j is the number stored at $g_{j'}$. If j appears in w, then look for the index j' in $\{k+1, \ldots, k+i\}$; otherwise look for

$$
\begin{array}{l}
\textbf{case } z \textbf{ of}\\
\quad \mathsf{r}_7(z){=}0 \;\Rightarrow\; \overline{h}(0);\\
\quad \mathsf{r}_7(z){=}1 \;\Rightarrow\; Find(\mathsf{d}_7(z)_0, w, y).(b)(b(x).\overline{h}(7{*}\langle y, \mathsf{d}_7(z)_1, x\rangle{+}1) \,|\, (h)(h(u).\overline{b}(u) \,|\, \mathcal{N}_{\mathrm{i}}(\mathsf{d}_7(z)_2, w)));\\
\quad \mathsf{r}_7(z){=}2 \;\Rightarrow\; Find(\mathsf{d}_7(z)_0, w, y).(b)(b(x).\overline{h}(7{*}\langle y, \mathsf{d}_7(z)_1, x\rangle{+}2) \,|\, (h)(h(u).\overline{b}(u) \,|\, \mathcal{N}_{\mathrm{i}}(\mathsf{d}_7(z)_2, w)));\\
\quad \mathsf{r}_7(z){=}3 \;\Rightarrow\; (b_1b_2)(b_1(x_1).b_2(x_2).\overline{h}(7{*}\langle x_1, x_2\rangle{+}3)\\
\qquad\qquad\qquad \,|\, (h)(h(u).\overline{b_1}(u) \,|\, \mathcal{N}_{\mathrm{i}}(\mathsf{d}_7(z)_0, w)) \,|\, (h)(h(u).\overline{b_2}(u) \,|\, \mathcal{N}_{\mathrm{i}}(\mathsf{d}_7(z)_1, w)));\\
\quad \mathsf{r}_7(z){=}4 \;\Rightarrow\; (b)(b(x).\overline{h}(7{*}\langle \mathsf{d}_7(z)_0, x\rangle{+}4) \,|\, (h)(h(u).\overline{b}(u) \,|\, \mathcal{N}_{\mathrm{i}}(\mathsf{d}_7(z)_1, \langle \mathsf{d}_7(z)_0, w\rangle)));\\
\quad \mathsf{r}_7(z){=}5 \;\Rightarrow\; (b)(b(x).\overline{h}(7{*}\langle \mathsf{d}_7(z)_0, x\rangle{+}5) \,|\, (h)(h(u).\overline{b}(u) \,|\, \mathcal{N}_{\mathrm{i}}(\mathsf{d}_7(z)_1, w)));\\
\quad \mathsf{r}_7(z){=}6 \;\Rightarrow\; Find(\mathsf{d}_7(z)_0, w, y).(b)(b(x).\overline{h}(7{*}\langle y, \mathsf{d}_7(z)_1, x\rangle{+}6) \,|\, (h)(h(u).\overline{b}(u) \,|\, \mathcal{N}_{\mathrm{i}}(\mathsf{d}_7(z)_2, w)));\\
\quad \mathsf{r}_7(z){=}7 \;\Rightarrow\; Find(\mathsf{d}_7(z)_0, w, y).(b)(b(x).\overline{h}(7{*}\langle y, \mathsf{d}_7(z)_1, x\rangle{+}7) \,|\, (h)(h(u).\overline{b}(u) \,|\, \mathcal{N}_{\mathrm{i}}(\mathsf{d}_7(z)_2, w)))\\
\textbf{end case}
\end{array}
$$

Figure 3: Normalizer $\mathcal{N}_{\mathrm{i}}(z, w)$.

$$
\begin{array}{l}
\textbf{case } z \textbf{ of}\\
\quad \mathsf{r}_7(z){=}0 \;\Rightarrow\; \mathbf{0};\\
\quad \mathsf{r}_7(z){=}1 \;\Rightarrow\; Nth(\mathsf{d}_7(z)_0, y).a_y(x).\mathcal{S}_{\mathrm{i}}([x/\mathsf{d}_7(z)_1]\mathsf{d}_7(z)_2);\\
\quad \mathsf{r}_7(z){=}2 \;\Rightarrow\; Nth(\mathsf{d}_7(z)_0, y).\overline{a_y}(val(\mathsf{d}_7(z)_1)).\mathcal{S}_{\mathrm{i}}(\mathsf{d}_7(z)_2);\\
\quad \mathsf{r}_7(z){=}3 \;\Rightarrow\; \mathcal{S}_{\mathrm{i}}(\mathsf{d}_7(z)_0) \,|\, \mathcal{S}_{\mathrm{i}}(\mathsf{d}_7(z)_1);\\
\quad \mathsf{r}_7(z){=}4 \;\Rightarrow\; Nth(\mathsf{d}_7(z)_0, y).(a_y)\mathcal{S}_{\mathrm{i}}(\mathsf{d}_7(z)_1);\\
\quad \mathsf{r}_7(z){=}5 \;\Rightarrow\; if\ val(\mathsf{d}_7(z)_0)\ then\ \mathcal{S}_{\mathrm{i}}(\mathsf{d}_7(z)_1);\\
\quad \mathsf{r}_7(z){=}6 \;\Rightarrow\; Nth(\mathsf{d}_7(z)_0, y).!a_y(x).\mathcal{S}_{\mathrm{i}}([x/\mathsf{d}_7(z)_1]\mathsf{d}_7(z)_2);\\
\quad \mathsf{r}_7(z){=}7 \;\Rightarrow\; Nth(\mathsf{d}_7(z)_0, y).!\overline{a_y}(val(\mathsf{d}_7(z)_1)).\mathcal{S}_{\mathrm{i}}(\mathsf{d}_7(z)_2)\\
\textbf{end case}
\end{array}
$$

Figure 4: Simulator $\mathcal{S}_{\mathrm{i}}(z)$.

the j' in $\{1, \ldots, k\}$. In w are stored the local names that might appear in the term being processed. This additional complexity is due to the fact that a number could denote both a local name and a global name. By making use of dynamic binding, a recursive invocation of $\mathcal{N}_{\mathrm{i}}(z, w)$ collects through the localized channel h the normal encoding of a subterm and passes it to its caller using the local channel b.

The simulator $\mathcal{S}_{\mathrm{i}}(z)$, defined in Fig. 4, simulates the $\mathbb{VPC}^!$-process coded up by z in an on-the-fly fashion. The arithmetical operations referred to in Fig. 4 are described below:

- The notation $[x/\mathsf{d}_7(z)_1]\mathsf{d}_7(z)_2$ stands for the Gödel number obtained from $\mathsf{d}_7(z)_2$ by substituting x, which must have been instantiated by an input action at the moment this operation is executed, for $\mathsf{d}_7(z)_1$.

- The notation $val(\mathsf{d}_7(z)_1)$ denotes the result of evaluating the term expression coded up by $\mathsf{d}_7(z)_1$. Similarly $val(\mathsf{d}_7(z)_0)$ denotes the result of evaluating the formula coded up by $\mathsf{d}_7(z)_0$. Notice that when the evaluation operations start, neither $\mathsf{d}_7(z)_0$ nor $\mathsf{d}_7(z)_1$ contains any variables.

The prefix operation $Nth(\mathsf{d}_7(z)_0, y)$ returns j as the value of y if $\mathsf{d}_7(z)_0$ is stored at g_j, where $1 \leq j \leq k + i$. Some comments on $\mathcal{S}_{\mathrm{i}}(z)$ are as follows:

- $\mathsf{r}_7(z)$=1. The continuation $a_y(x).\mathcal{S}_{\mathrm{i}}([x/\mathsf{d}_7(z)_1]\mathsf{d}_7(z)_2)$ is an abbreviation of

  if y=1 *then* $a_1(x).\mathcal{S}_{\mathrm{i}}([x/\mathsf{d}_7(z)_1]\mathsf{d}_7(z)_2) \mid \ldots \mid$ *if* y=k *then* $a_k(x).\mathcal{S}_{\mathrm{i}}([x/\mathsf{d}_7(z)_1]\mathsf{d}_7(z)_2)$.

  In case $\mathsf{r}_7(z)$=2, the subterm $\overline{a_y}(val(\mathsf{d}_7(z)_1)).\mathcal{S}_{\mathrm{i}}(\mathsf{d}_7(z)_2)$ is defined similarly.

- $\mathsf{r}_7(z)$=4. This case deserves special attention. This is where we have to use $\mathbb{VPC}$. An implementation of the recursive call of the simulator in $\mathbb{VPC}^!$ would render the localization operator (a_y) detached from the body to which the operator should apply.

It is remarkable that the interpreter uses only one dummy variable x. No confusion among the bound variables can ever arise.

Although we have not supplied all the details of the interpretation, the key ingredients that support the following claim have all been spelled out.

Theorem 8. $\mathbb{VPC}^! \in \mathbb{VPC}$.

Proof. The argument given here rests on the completeness of $\mathbb{VPC}$ [Fu13] and our trust in the Church-Turing Thesis. We summarize the main points below:

- The encoding of the natural numbers is given by the class $\{\overline{a}(k)\}_{k\in\mathsf{N},a\in\mathcal{N}}$. This encoding is correct with respect to the absolute equality $=_{\mathbb{VPC}}$ [Fu13]. All the number theoretical operations involved in the definition of $\mathcal{I}_d^{i,J}$ are computable. It follows that these operations are all definable in $\mathbb{VPC}$.

- The relation $\propto_{\mathrm{i}}$: $\mathbb{VPC}^! \to \mathbb{VPC}$ is basically the structural embedding composed with the equality relation $=_{\mathbb{VPC}}$, where the replication operator in $\mathbb{VPC}^!$ is interpreted by the derived replication operator in $\mathbb{VPC}$.

- The interpreter $\{\mathcal{I}_d^{i,J}\}_{i\in\mathsf{N},J\in\mathcal{N}^*,d\notin J}$ is defined in (4). To establish (3) it is sufficient to demonstrate that the following relation, denoted by $\Im$, is a subbisimilarity.

$$\left\{ (P, (d)(\overline{d}(k) \mid \mathcal{I}_d^{i,J})) \;\middle|\; \begin{array}{l} P \text{ is a } \mathbb{VPC}^! \text{ process, all global names of } P \\ \text{are in } J, \text{ the number of local names of } P \text{ is} \\ \text{no more than } i,\ k \text{ is an index of } P, \text{ and } d \notin J. \end{array} \right\}; =_{\mathbb{VPC}} .$$

  It is easy to argue informally that the definition of the simulator in Fig. 4 renders true the following statements:

  - Since all the actions of P are also actions of $(d)(\overline{d}(k) \mid \mathcal{I}_d^{i,J})$, the following explicit bisimulation property is valid whenever $P\Im Q$:
    - If $P \xrightarrow{\alpha} P'$ then $Q \to^* \xrightarrow{\alpha} Q' \Im^{-1} P'$ for some Q'.
    - If $Q \xrightarrow{\alpha} Q'$ and $\alpha \neq \tau$ or $Q \xrightarrow{\iota} Q'$, then $\exists P'.P \xrightarrow{\alpha} P' \Im Q'$.
    - If $Q \to Q'$ then either $P\Im Q'$ or $\exists P'.P \to P'\Im Q'$.

– $\Im$ is codivergent since the extra number theoretical manipulations do not introduce any divergence.

These properties are enough for us to conclude that $\Im$ is a subbisimilarity.

This completes the proof sketch. □

3.3 Universal VPC Process

A self-interpreter for $\mathbb{M}$ is an interpreter of $\mathbb{M}$ in $\mathbb{M}$. Such an interpreter is based on a subbisimilarity from $\mathbb{M}$ to $\mathbb{M}$. In general there is more than one subbisimilarity from $\mathbb{M}$ to $\mathbb{M}$ [Fu14b], among which the absolute equality $=_{\mathbb{M}}$ offers a canonical relation in the sense that every process is interpreted by itself. An interpreter of $\mathbb{M}$ in $\mathbb{M}$ based on the absolute equality $=_{\mathbb{M}}$ is called a *universal process* for $\mathbb{M}$. We will write $\langle\{\mathcal{U}_c^{i,J}\}_{i\in\mathsf{N},J\in\mathcal{N}^*,c\notin J}, [\![\_]\!]\rangle$ for a universal process of $\mathbb{M}$. For all i, J, c the process $\mathcal{U}_c^{i,J}$ must satisfy the following property: If P is of the right type and k is a Gödel index of P then a unique Q exists such that

$$[\![k]\!]_c \,|\, \mathcal{U}_c^{i,J} \xrightarrow{\iota} Q =_{\mathbb{M}} P. \tag{5}$$

The aim of this subsection is to modify the interpreter constructed in the previous subsection to obtain a universal process for $\mathbb{VPC}$.

We need to explain how parametric definitions are treated. Now every $\mathbb{VPC}$-term refers to only a finite number of parametric definitions. Suppose the parametric definitions appearing in T are given by the following equations:

$$\begin{array}{rcl} D_1(x_{11},\ldots,x_{1i_1}) & = & T_1, \\ & \vdots & \\ D_k(x_{k1},\ldots,x_{ki_k}) & = & T_k, \end{array} \tag{6}$$

for some $k \geq 0$. When $k = 0$ we understand that T contains no occurrence of any parametric definition. The Gödel index $[\![T]\!]_{\mathfrak{u}}$ of T is defined as follows:

$$[\![T]\!]_{\mathfrak{u}} \stackrel{\text{def}}{=} \langle k, \langle [\![T]\!]_{\mathfrak{d}}, [\![T]\!]_{\mathfrak{p}}\rangle\rangle. \tag{7}$$

The components of (7) have the following readings:

- k is the number of parametric definitions in (6). According to our notational convention $[\![T]\!]_{\mathfrak{d}}$, which is a k-ary tuple, is 0 when $k = 0$.
- The index $[\![T]\!]_{\mathfrak{d}}$ codes up all the parametric definitions in (6). It is given by

$$[\![T]\!]_{\mathfrak{d}} \stackrel{\text{def}}{=} \begin{array}{c} \langle\, \langle i_1, \langle \varsigma(x_{11}),\ldots,\varsigma(x_{1i_1}), [\![T_1]\!]_{\mathfrak{p}}\rangle\rangle, \\ \vdots \\ \langle i_k, \langle \varsigma(x_{k1}),\ldots,\varsigma(x_{ki_k}), [\![T_k]\!]_{\mathfrak{p}}\rangle\rangle\, \rangle. \end{array} \tag{8}$$

  This is not a self-referential definition since the function $[\![\_]\!]_{\mathfrak{p}}$ is independent from the function $[\![\_]\!]_{\mathfrak{u}}$.

$$
\begin{array}{rcl}
[\![\mathbf{0}]\!]_{\mathfrak{p}} & \stackrel{\text{def}}{=} & 0, \\
[\![a(x).T]\!]_{\mathfrak{p}} & \stackrel{\text{def}}{=} & 6 * \langle \varsigma(a), \varsigma(x), [\![T]\!]_{\mathfrak{p}} \rangle + 1, \\
[\![\overline{a}(t).T]\!]_{\mathfrak{p}} & \stackrel{\text{def}}{=} & 6 * \langle \varsigma(a), [\![t]\!]_{\varsigma}, [\![T]\!]_{\mathfrak{p}} \rangle + 2, \\
[\![T \mid T']\!]_{\mathfrak{p}} & \stackrel{\text{def}}{=} & 6 * \langle [\![T]\!]_{\mathfrak{p}}, [\![T']\!]_{\mathfrak{p}} \rangle + 3, \\
[\![(c)T]\!]_{\mathfrak{p}} & \stackrel{\text{def}}{=} & 6 * \langle \varsigma(c), [\![T]\!]_{\mathfrak{p}} \rangle + 4, \\
[\![\mathit{if}\ \varphi\ \mathit{then}\ T]\!]_{\mathfrak{p}} & \stackrel{\text{def}}{=} & 6 * \langle [\![\varphi]\!]_{\varsigma}, [\![T]\!]_{\mathfrak{p}} \rangle + 5, \\
[\![D_j(t_{j1}, \ldots, t_{jn_j})]\!]_{\mathfrak{p}} & \stackrel{\text{def}}{=} & 6 * \langle j, \langle n_j, \langle [\![t_{j1}]\!]_{\varsigma}, \ldots, [\![t_{jn_j}]\!]_{\varsigma} \rangle \rangle \rangle + 6.
\end{array}
$$

Figure 5: Gödel Index of $\mathbb{VPC}$-Term.

- The structural definition of $[\![\_]\!]_{\mathfrak{p}}$ is given in Fig. 5. The only thing worth mentioning is that the index of $D_j(t_{j1}, \ldots, t_{ji_j})$ contains the information about the equation in which D_j is defined, the number of parameters of D_j, and all the terms that instantiate the parameters.

At top level the indices of all $\mathbb{VPC}$-terms are of the form (7). One may think of $[\![T]\!]_{\mathfrak{p}}$ as the main program and $[\![T]\!]_{\mathfrak{b}}$ as the subroutines necessary when executing the program.

Our universal process $\{\mathcal{U}_d^{i,J}\}_{i \in \mathsf{N}, J \in \mathcal{N}^*, d \notin J}$ for $\mathbb{VPC}$ is defined as follows:

$$
\mathcal{U}_d^{i,a_1 \ldots a_k} \stackrel{\text{def}}{=} d(z).(h)(\mathcal{P}_{\mathfrak{u}}(z) \mid h(z).(e)(\mathcal{D}_{\mathfrak{u}}((z)_0, z_{1,0}) \mid \mathcal{S}_{\mathfrak{u}}(z_{1,1}))) \tag{9}
$$

for all $i, a_1, \ldots, a_k, d$ such that $d \notin \{a_1, \ldots, a_k\}$. The process $\mathcal{U}_d^{i,a_1 \ldots a_k}$ is similar to $\mathcal{I}_d^{i,a_1 \ldots a_k}$ defined in (4). We leave out the definition of the parser $\mathcal{P}_{\mathfrak{u}}(z)$ since it is similar to $\mathcal{P}_{\mathfrak{i}}(z)$ and it is purely arithmetical. The process $\mathcal{D}_{\mathfrak{u}}((z)_0, z_{1,0})$ is an instantiation of the parametric definition $\mathcal{D}_{\mathfrak{u}}(x, y)$ given by the following equation:

$$
\begin{array}{rcl}
\mathcal{D}_{\mathfrak{u}}(x, y) & = & !e(v).\mathit{if}\ 1 \leq (v)_0 \leq x\ \mathit{then\ let}\ w = (v)_0 - 1 \\
& & \mathit{in\ let}\ u = y_{w,0}\ \mathit{in}\ \mathcal{S}_{\mathfrak{u}}([v_{1,1,u-1}/y_{w,1,u-1}] \ldots [v_{1,1,0}/y_{w,1,0}]y_{w,1,u}).
\end{array}
$$

The first parameter of $\mathcal{D}_{\mathfrak{u}}(x, y)$ indicates the number of the mutually dependent equations. The second parameter is the Gödel index of these parametric definitions that takes the form of (8). The definition $\mathcal{D}_{\mathfrak{u}}(x, y)$ is essentially the interpretation of $[\![T]\!]_{\mathfrak{b}}$. It is able to simulate all the parametric definitions that are encoded in y. Again this is possible because the simulation is on-the-fly. The simulator $\mathcal{S}_{\mathfrak{u}}(z)$ in (9) is defined in Fig. 6. The recursive functions $\mathsf{r}_6, \mathsf{d}_6$ are similar to those of $\mathsf{r}_7, \mathsf{d}_7$ respectively. In case $\mathsf{r}_6(z) = 6$ the subroutine $\mathcal{D}_{\mathfrak{u}}((z)_0, z_{1,0})$ is invoked with the parameter $\mathsf{d}_6(z)$.

By an argument similar to the one given in the proof of Theorem 8, one can convince oneself of the validity of the following result.

Theorem 9. $\mathbb{VPC} \in \mathbb{VPC}$.

4 Application

The existence of a universal process can be seen as an expressiveness criterion. There cannot be any universal process for CCS [Mil89] or the process-passing calculus [San92,

case z **of**
$\mathsf{r}_6(z)=0 \;\Rightarrow\; \mathbf{0}$;
$\mathsf{r}_6(z)=1 \;\Rightarrow\; \mathit{Nth}(\mathsf{d}_6(z)_0, y).a_y(x).\mathcal{S}_{\mathrm{u}}([x/\mathsf{d}_6(z)_1]\mathsf{d}_6(z)_2)$;
$\mathsf{r}_6(z)=2 \;\Rightarrow\; \mathit{Nth}(\mathsf{d}_6(z)_0, y).\overline{a_y}(\mathit{val}(\mathsf{d}_6(z)_1)).\mathcal{S}_{\mathrm{u}}(\mathsf{d}_6(z)_2)$;
$\mathsf{r}_6(z)=3 \;\Rightarrow\; \mathcal{S}_{\mathrm{u}}(\mathsf{d}_6(z)_0) \mid \mathcal{S}_{\mathrm{u}}(\mathsf{d}_6(z)_1)$;
$\mathsf{r}_6(z)=4 \;\Rightarrow\; \mathit{Nth}(\mathsf{d}_6(z)_0, y).(a_y)\mathcal{S}_{\mathrm{u}}(\mathsf{d}_6(z)_1)$;
$\mathsf{r}_6(z)=5 \;\Rightarrow\; \mathit{if}\ \mathit{val}(\mathsf{d}_6(z)_0)\ \mathit{then}\ \mathcal{S}_{\mathrm{u}}(\mathsf{d}_6(z)_1)$;
$\mathsf{r}_6(z)=6 \;\Rightarrow\; \overline{e}(\mathsf{d}_6(z))$
end case

Figure 6: Simulator $\mathcal{S}_{\mathrm{u}}(z)$.

Tho95] since neither is complete [Fu14b]. But once an interaction model does admit some sort of universal process, a whole range of new applications are available. In this section we sketch three of them.

4.1 Process Passing as Value Passing

The most valuable contribution of a universal process is that it allows a receiving process to interpret a number as a process. This is a very useful property when applying $\mathbb{VPC}$ to tackle practical programming issues. But is it necessary to pass a process from one location to another? If the process that appears in the target environment is completely the same as the one sent from the source environment, a positive answer to the question can hardly be convincing. What is useful in practice is that the source process sends an *abstraction* parameterized over names, and the process on the receiving end instantiates the parameters of the abstraction by its local names. This way of introducing the higher order feature in the π-calculus is adopted in [San92, San93]. We follow the same approach to extend $\mathbb{VPC}$. Our higher order $\mathbb{VPC}$ has the following grammar:

$$T := \ldots \mid X(a_1, \ldots, a_j) \mid A(a_1, \ldots, a_j) \mid a(X{:}\langle i, j\rangle).T \mid \overline{a}(A{:}\langle i, j\rangle).T,$$

where only the higher order terms are indicated. An abstraction A is a term whose global names are abstracted away. A process with j global names can be turned into an abstraction with j parameters. If for example T is a term with the global names $c_1, \ldots, c_j$ then $\lambda c_1 \ldots c_j.T$ is an abstraction. In the above syntax $A : \langle i, j\rangle$ indicates that A is an abstraction with i local names and j parameters. The term $a(X{:}\langle i, j\rangle).T$ is a higher order input, in which $\langle i, j\rangle$ provides the typing information of the abstraction variable X, $\overline{a}(A{:}\langle i, j\rangle).T$ is in higher order output form, $X(a_1, \ldots, a_j)$ is an instantiation of the abstraction variable X at $a_1, \ldots, a_j$ and $A(a_1, \ldots, a_j)$ an instantiation of the abstraction A at $a_1, \ldots, a_j$. The instantiation of the abstraction $\lambda c_1 \ldots c_j.T$ at $a_1, \ldots, a_j$ is syntactically the same as the term $T\{a_1/c_1 \ldots, a_j/c_j\}$. Instantiations must be type correct. Formally the names in the higher order $\mathbb{VPC}$ are typed in the same way as the channels in the higher order π-calculus are typed [San92, San93]. We ignore the type system in the present light weight treatment. In addition to the operational semantics

of $\mathbb{VPC}$ the higher order $\mathbb{VPC}$ has the following semantic rules:

$$\frac{}{a(X{:}\langle i,j\rangle).T \xrightarrow{a(A)} T\{A/X\}} \qquad \frac{}{\overline{a}(A{:}\langle i,j\rangle).T \xrightarrow{\overline{a}(A)} T} \qquad \frac{S \xrightarrow{a(A)} S' \quad T \xrightarrow{\overline{a}(A)} T'}{S \mid T \xrightarrow{\tau} S' \mid T'}$$

In the higher order input rule A must be of the same type as the variable X.

We can now explain how to simulate the operational semantics of the higher order $\mathbb{VPC}$ in the first order $\mathbb{VPC}$. Let υ stand for a partial function from the set of abstraction variables to V that is injective on its finite domain of definition. The notation $\upsilon[Z{\to}z]$ refers to the function that is the same as υ except that it sends Z onto z. The nontrivial part of the encoding is given below:

$$\begin{aligned}
[\![X(a_1,\ldots,a_j)]\!]_\upsilon &= (d)(\overline{d}([\varsigma(a_1)/v_1,\ldots,\varsigma(a_j)/v_j]\upsilon(X)) \mid \mathcal{U}_d^{i,a_1\ldots a_j}),\\
[\![a(X{:}\langle i,j\rangle).T]\!]_\upsilon &= a(x).[\![T]\!]_{\upsilon[X\to x]}, \text{ where } x \text{ is chosen such that it is not in } T,\\
[\![\overline{a}(A{:}\langle i,j\rangle).T]\!]_\upsilon &= \overline{a}([\![A]\!]_\upsilon).[\![T]\!]_\upsilon.
\end{aligned}$$

The operation $[\varsigma(a_1)/v_1,\ldots,\varsigma(a_j)/v_j](\_)$ replaces in $\upsilon(X)$ the indexes $v_1,\ldots,v_j$ of j global names by the indexes of $a_1,\ldots,a_j$ respectively. Notice that since we are dealing with processes containing neither first order variables nor higher order variables, by the time the on-the-fly interpretation reaches $\upsilon(X)$, it has already been instantiated by a number. Thus $v_1,\ldots,v_j$ can be safely calculated from $\upsilon(X)$. The encoding of a higher order $\mathbb{VPC}$ process is given by $[\![P]\!]_\emptyset$, where $\emptyset$ is the nowhere defined function.

The translation of the higher order $\mathbb{VPC}$ into the first order $\mathbb{VPC}$ is notably different from the translation of the higher π into the first order π [San93]. For one thing the simulation of the higher order communication in $\mathbb{VPC}$ is achieved by code transmission, whereas in the π scenario the simulation is implemented via access control. An advantage of our encoding is that it allows the 'user' to exert tight control over the executions. The user may wish to terminate the simulation after a certain amount of time, bounded by a time complexity function. This can be implemented by incorporating into the encoding a timer that counts the number of simulation steps already performed.

In practice it is the code transmission, rather than the programme transmission, that is widely used.

4.2 Communication Security

The completeness of an interaction model means that any encryption/decryption algorithm can be implemented in it and an encrypted text can be passed from one process to another. By exploiting that fact, a universal process can offer an effective way to enhance the security of communications. Suppose party A intends to send a piece of programme to party B through a public channel. There is no way to prevent anyone from eavesdropping on the communication channels. What party A can do is to encrypt the Gödel number of the programme before sending it to party B. After receiving the number, party B decodes the number to recover the Gödel index. It then places the encoded programme in its private environment and puts it into action by invoking a universal process. Far more complicated scenarios can be designed along this line of thinking.

The point is that the existence of a universal process allows one to implement the well known security protocols in $\mathbb{VPC}$ to enhance communication security. This is a more traditional approach compared to the one that introduces explicit operators to model security protocols in process calculi [AG99].

4.3 Programming Paradigm

If $\mathbb{VPC}$ is seen as a machine model, what would be an implementation of a higher order programming language in $\mathbb{VPC}$? If $\mathbb{VPC}$ is seen as a programming language, what kind of programming paradigm does it support? The issue of constructing a higher order programming language on top of a process model has been studied by several research groups, see the references in [SWU10]. This is not the right place to overview the existing approaches and tools. What we are going to do is to propose a new programming paradigm that makes essential use of the universal processes. To explain our idea we introduce a new process model referred to as $\mathbb{PL}$. This is essentially the same as $\mathbb{VPC}$. But instead of sending and receiving numbers, a $\mathbb{PL}$ process sends and receives strings. The reason for a string-passing process calculus is that it provides the right level of abstraction to study programming theory in process algebra. A piece of program is nothing but a string, which can be parsed, type checked and executed.

Let Σ be a finite set of *symbols* and Σ^* be the set of finite *strings* over Σ. We will write ϱ for a string variable and ℓ for a *string term* constructed from strings, string variables and string operations. We write ψ for a boolean formula about strings. For simplicity we see a symbol as a string of length one. We choose a set of basic string operations on Σ^* and a set of basic logic operations on Σ^*. We obviously need the head, tail and length operations, as well as the binary prefix relation among others. The particular choice of the operations is not our concern.

The set of the $\mathbb{PL}$-terms is defined by the following BNF:

$$T \quad := \quad \mathbf{0} \mid a(\varrho).T \mid \overline{a}(\ell).T \mid T \mid T' \mid (c)T \mid \mathit{if}\ \psi\ \mathit{then}\ T \mid D(\ell_1, \ldots, \ell_k).$$

The operational semantics of $\mathbb{PL}$ is obtained from the labeled transition system of $\mathbb{VPC}$ by substituting strings for numbers. The model $\mathbb{PL}$ is easily seen to be complete. In fact there is clearly an effective bijection between Σ^* and N. Using this bijection it ought to be easy to construct two subbisimilarities to support the claim that $\mathbb{PL} \sqsubseteq \mathbb{VPC}$ and $\mathbb{VPC} \sqsubseteq \mathbb{PL}$. From the practical point of view it is convenient to assume that Σ contains the proper subset $\Sigma_{\mathsf{N}} = \{0, \mathsf{succ}, +, \times\}$. The calculus $\mathbb{PL}_{\mathsf{N}}$ defined in terms of the symbol set Σ_{N} is as expressive as $\mathbb{VPC}$. We may think of $\mathbb{PL}_{\mathsf{N}}$ as a machine model on which $\mathbb{PL}$ is implemented, by which we mean that there is an interpreter of $\mathbb{PL}$ in $\mathbb{PL}_{\mathsf{N}}$.

Now the general framework has been set up, let's explain how programming languages can be implemented in our model. Suppose $\mathfrak{L}$ is a concurrent, typed programming language, which could be as sophisticated as a full-fledged programming language or as simple as the concurrent language studied in [Mil89]. The *definition* of $\mathfrak{L}$ is given by a parser $\{\mathfrak{P}_c\}_{c\in\mathcal{N}}$. Let Pr be an $\mathfrak{L}$ program. Then

$$\overline{c}(Pr) \mid \mathfrak{P}_c \xrightarrow{\iota} L$$

for some L. The process indicates acceptance if Pr is a well defined $\mathfrak{Q}$ program, it refuses if Pr violates the $\mathfrak{Q}$ grammar. The *implementation* of $\mathfrak{Q}$ is given by an executor $\{\mathfrak{E}_c^{i,J}\}_{i\in\mathsf{N},J\in\mathcal{N}^*,c\notin J}$, which is feasible by the technique developed in this paper. For a legitimate $\mathfrak{Q}$ program Pr of the right type one must have that

$$\overline{c}(Pr)\,|\,\mathfrak{E}_c^{i,J} \xrightarrow{\iota} I$$

for some I that implements Pr in $\mathbb{PL}$. This oversimplified account should be enough to give the reader a taste of our methodology.

Different programming paradigms are supported by different process models. If one intends to study the object oriented features, the right model is the π-calculus. Since π is also complete [Fu14b], everything carried out in this paper can be repeated for π. It should not be difficult to internalize, as it were, Walker's meta translation of an object oriented language [Wal91, Wal95] in the fashion advocated here. What we get is an implementation, rather than a translation, of the object oriented language in π.

5 S-m-n Theorem

In this section we explain how to do recursion theory by taking a look at $\mathbb{VPC}$ version of the S-m-n Theorem. The challenge here is actually how to formulate it correctly. We know from the recursion theory that S-m-n Theorem is about partial evaluation. There is an effective way to transfer the index of a $(k_0{+}k_1)$-ary effective function $\mathsf{f}(x_1,\ldots,x_{k_0},y_1,\ldots,y_{k_1})$ and the inputs $i_1,\ldots,i_{k_0}$ to the index of the k_1-ary effective function $\mathsf{f}(i_1,\ldots,i_{k_0},y_1,\ldots,y_{k_1})$. If we are ever to have a recursion theory of $\mathbb{VPC}$ processes, we must start by answering the question of what the right $\mathbb{VPC}$ counterpart of a recursive function is. It is not hard to see that the most natural generalization of a recursive function is a parametric definition of the form

$$D(z_1,\ldots,z_k) = T. \tag{10}$$

For numbers $i_1,\ldots,i_j$, where $j \leq k$, we will write $D(i_1,\ldots,i_j,z_{j+1},\ldots,z_k)$ for the $(k{-}j)$-ary parametric definition $D'(z_{j+1},\ldots,z_k)$ given by

$$D'(z_{j+1},\ldots,z_k) = T\{i_1/z_1,\ldots,i_j/z_j\}.$$

Suppose i is the number of local names of T and $\jmath$ is the list of the global names in T. We say that the $D(z_1,\ldots,z_k)$ defined in (10) is a k-ary parametric definition of type $[i,\jmath]$. Two k-ary parametric definitions, say $D_0(z_1,\ldots,z_k)$ and $D_1(z_1,\ldots,z_k)$, are equal if for all numbers $i_1,\ldots,i_k$, one has $D_0(i_1,\ldots,i_k) =_{\mathbb{VPC}} D_1(i_1,\ldots,i_k)$.

By recycling the encoding in Section 3.1 we can Gödelize the set of the k-ary parametric definitions of type $[i,\jmath]$. Our technique to derive a universal process, as developed in Section 3, helps define a universal process $\mathcal{U}_z^{[i,\jmath][k]}(z_1,\ldots,z_k)$ for the set of the k-ary parametric definitions of type $[i,\jmath]$. Here the word 'process' is not very precise since $\mathcal{U}_z^{[i,\jmath][k]}(z_1,\ldots,z_k)$ is a parametric definition rather than a process. The parameter z is made an index, suggesting that $\mathcal{U}_z^{[i,\jmath][k]}(z_1,\ldots,z_k)$ should be thought of as the z-th

k-ary parametric definition of type $[i, j]$. The defining property of $\mathcal{U}_z^{[i,j][k]}(z_1, \ldots, z_k)$ requires that for each number j, say the index of $D(z_1, \ldots, z_k)$, the equality

$$\mathcal{U}_j^{[i,j][k]}(z_1, \ldots, z_k) =_{\mathbb{VPC}} D(z_1, \ldots, z_k)$$

holds. Now we can state the S-m-n Theorem.

Theorem 10. *Suppose k_0, k_1 are natural numbers. There is a total $(k_0{+}1)$-ary recursive function $\mathsf{s}_{k_1}^{k_0}(z, x_1, \ldots, x_{k_0})$ such that for all numbers $j, i_1, \ldots, i_{k_0}$ the following equality holds:*

$$\mathcal{U}_j^{[i,j][k_0+k_1]}(i_1, \ldots, i_{k_0}, y_1, \ldots, y_{k_1}) =_{\mathbb{VPC}} \mathcal{U}^{[i,j][k_1]}_{\mathsf{s}_{k_1}^{k_0}(j,i_1,\ldots,i_{k_0})}(y_1, \ldots, y_{k_1}).$$

Proof. The proof follows the standard argument. Given the index of a $(k_0{+}k_1)$-ary parametric definition $D''(x_1, \ldots, x_{k_0}, y_1, \ldots, y_{k_1})$ of type $[i, j]$, one can effectively construct the k_1-ary parametric definition $D''(i_1, \ldots, i_{k_0}, y_1, \ldots, y_{k_1})$ of the same type, from which we get its Gödel index effectively. This defines a total recursive function. □

The S-m-n Theorem helps import results in recursion theory to process theory. The famous Rice Theorem [Rog87] is one such result.

Theorem 11. *Suppose $\mathcal{B}$ is a set of k-ary parametric definitions that satisfies the following properties:*

1. *$\mathcal{B}$ is not empty;*
2. *there is some k-ary parametric definition that is not in $\mathcal{B}$;*
3. *$\mathcal{B}$ is closed under the absolute equality.*

Then the set $\{j \mid \mathcal{U}_j^{[i,j][k]}(x_1, \ldots, x_k) \in \mathcal{B}$ for some $[i, j]\}$ is undecidable.

Proof. Without loss of generality, suppose $\Omega \notin \mathcal{B}$. By condition (1) the set $\mathcal{B}$ contains some k-ary parametric definition $D(x_1, \ldots, x_k)$. Let W contain the number $\langle i_1, \ldots, i_k\rangle$ if the $\langle i_1, \ldots, i_k\rangle$-th k-ary recursive function is definable at $i_1, \ldots, i_k$. It is well known that W is recursive enumerable but not decidable. Let the $(k{+}1)$-ary parametric definition $D'(z, x_1, \ldots, x_k)$ be defined by

$$D'(z, x_1, \ldots, x_k) \quad = \quad \textit{if } z \in W \textit{ then } D(x_1, \ldots, x_k).$$

Suppose $D'(z, x_1, \ldots, x_k)$ is of type $[i, j]$. Then some number j exists such that

$$\mathcal{U}_j^{[i,j][k+1]}(z, x_1, \ldots, x_k) =_{\mathbb{VPC}} D'(z, x_1, \ldots, x_k).$$

According to Theorem 10 some binary total recursive function s exists such that

$$\mathcal{U}^{[i,j][k]}_{\mathsf{s}(j,z)}(x_1, \ldots, x_k) =_{\mathbb{VPC}} \mathcal{U}_j^{[i,j][k+1]}(z, x_1, \ldots, x_k).$$

Using (3) it is clear that $k \in W$ if and only if $D'(k, x_1, \ldots, x_k) =_{\mathbb{VPC}} D(x_1, \ldots, x_k)$ if and only if $\mathcal{U}^{[i,j][k]}_{\mathsf{s}(j,k)}(x_1, \ldots, x_k) \in \mathcal{B}$. So $\mathcal{B}$ cannot be decidable. □

There is nothing new about the above proof. But at least it demonstrates that the type constraint $[i, j]$ of $\mathcal{U}^{[i,j][k]}$ is not much of a restriction.

A simple consequence of the Rice Theorem is about the unobservable processes.

Corollary 12. *The set of the unobservable processes is undecidable.*

6 Future Work

The idea of designing universal processes was discussed in [AMS97] in the framework of CCS. Due to the limitation of the model, a universal process for CCS is static in the sense that it must preload the Gödel number of the CCS process to be simulated since it can never dynamically input any number. In order to simulate the branching structure of a process, the universal process for CCS must introduce divergence. A recent work is reported in [BLT11], where the authors studied universal Reactive Turing Machines. A universal Reactive Turing Machine either introduces divergence, or places restriction on the maximum branching degree of the Reactive Turing Machine being simulated, both divergence and branching degree being of semantic nature. A universal machine for Reactive Turing Machines is also static in the above sense since the transmission of the description of a machine, which is a string of symbols, to the universal machine should not be interrupted. The advantage of our universal process is it is dynamic and does not impose any *semantic* constraints on any processes to be simulated.

The constructions of the universal processes in other complete models can be similarly given. In π-calculus for example the numbers can be coded up using the following inductively defined name indexed functions:

$$\begin{aligned} [\![0]\!]_a &= \overline{a}(c).\overline{c}(b).\overline{c}(e).\overline{e}e, \\ [\![i{+}1]\!]_a &= \overline{a}(c).\overline{c}(b).\overline{c}(e).[\![i]\!]_b. \end{aligned}$$

It is more or less a formality to apply the approach of this paper to generate a universal process of π. We leave it as future work to investigate how such a universal process can be exploited in a fruitful way.

Not every complete model seems to have a universal process though. At the time of writing we do not yet know if $\mathbb{VPC}^!$ has a universal process. Our preliminary study of this problem has not led to any definite answer. In a more general scenario, the technique to prove or disprove that a particular process model has a universal process may well reveal some fundamental property of the model.

The idea of universal process can be further exploited. Two important directions are outlined below.

- At the theoretical level, it is interesting to look at a recursion theory of process. It is well known that the bulk of the classic recursion theory of functions can be developed from the universal functions, the S-m-n theorem and Recursion Theorem [Rog87, Soa87]. It remains to see however how far a recursion theory of $\mathbb{VPC}$ can take us.

- At the programming level, it is worth the effort to study programming theory in a systematic way. More generally we can look at the class of process calculi with universal processes. These are models well equipped to model programming features. The significance of these models to programming theory is yet to be investigated.

Is it possible for a universal process to be a single process rather than a family of processes? A drastic approach to address the issue is to introduce a bijective naming

function $\nu_{(\_)} : \mathsf{N} \to \mathcal{N}$ that a $\mathbb{VPC}$ process may make use of. The function $\nu_{(\_)}$ gives an enumeration $\nu_0, \nu_1, \ldots$ of the names. It can be extended to a function from T to the set $\{\nu_t \mid t \in \mathsf{T}\}$ of name expressions. Now the grammar of $\mathbb{VPC}_\nu$ can be defined by the following BNF:

$$T \quad := \quad \mathbf{0} \mid \nu_t(x).T \mid \overline{\nu_t}(t).T \mid T \mid T \mid (\nu_i)T \mid \textit{if } \psi \textit{ then } T \mid D(t_1, \ldots, t_k).$$

An example of a $\mathbb{VPC}_\nu$ process is

$$\nu_0(x).\nu_1(y).\textit{if } x = 2y \textit{ then } \overline{\nu_x}(y).$$

It is clear from this example that $\mathbb{VPC}_\nu$ admits mobile computing to a certain degree. Notice that the match operator $[\nu_t{=}\nu_{t'}]T$ is definable in $\mathbb{VPC}_\nu$. The new model lacks the power, and the trouble as well, introduced by relocating local names. The virtue of $\mathbb{VPC}_\nu$ is that it has a single universal process $\mathcal{U}_c$ that is capable of dealing with indices of all $\mathbb{VPC}_\nu$ processes. This is rendered possible by the naming function which produces a canonical indexing for all the names whatsoever. Further study on $\mathbb{VPC}_\nu$ is necessary before we can evaluate its theoretical and practical relevance to process theory.

Acknowledgments This work has been supported by NSFC (60873034, 61033002). Xiaojuan Cai's idea about the on-the-fly simulations has been very instructive to this work. Sandy Harris and Huan Long have helped in improving the quality of this paper.

References

[AG99] M. Abadi and A. Gordon. A calculus for cryptographic protocols: The spi calculus. *Informaiton and Computation*, 148(1):1–70, 1999.

[AMS97] H. Andersen, S. Mørk, and M. Sørensen. A universal reactive machine. In *Proc. CONCUR 1997*, volume 1243 of *Lecture Notes in Computer Science*, pages 89–103, 1997.

[BGZ03] N. Busi, M. Gabbrielli, and G. Zavattaro. Replication vs recursive definitions in channel based calculi. In *Proc. ICALP'03*, volume 2719 of *Lecture Notes in Computer Science*, pages 133–144, 2003.

[BGZ04] N. Busi, M. Gabbrielli, and G. Zavattaro. Comparing recursion, replication and iteration in process calculi. In *Proc. ICALP'04*, volume 3142 of *Lecture Notes in Computer Science*, pages 307–319, 2004.

[BLT11] J. Baeten, B. Luttik, and P. Tilburg. Reactive turing machine. In *Proc. FCT 2011*, volume 6914 of *Lecture Notes in Computer Science*, pages 348–359, 2011.

[End01] J. Enderton. *A Mathematical Introduction to Logic*. Harcourt/Academic Press, 2001.

[FL10] Y. Fu and H. Lu. On the expressiveness of interaction. *Theoretical Computer Science*, 411:1387–1451, 2010.

[Fu13] Y. Fu. The value-passing calculus. In *Theories of Programming and Formal Methods*, Lecture Notes in Computer Science 8051, pages 166–195, 2013.

[Fu14a] Y. Fu. Nondeterministic structure of computation. *To appear in Mathematical Structures in Computer Science*, 2014.

[Fu14b] Y. Fu. Theory of interaction. *Theoretical Computer Science*, revision submitted (`http://basics.sjtu.edu.cn/~yuxi/`), 2014.

[Gor08] D. Gorla. Towards a unified approach to encodability and separation results for process calculi. In *CONCUR 2008*, Lecture Notes in Computer Science 5201, pages 492–507, 2008.

[GSV04] P. Giambiagi, G. Schneider, and F. Valencia. On the expressiveness of infinite behavior and name scoping in process calculi. In *FOSSACS 2004*, Lecture Notes in Computer Science 2987, pages 226–240, 2004.

[HI93a] M. Hennessy and A. Ingólfsdóttir. Communicating processes with value-passing and assignment. *Journal of Formal Aspects of Computing*, 5:432–466, 1993.

[HI93b] M. Hennessy and A. Ingólfsdóttir. A theory of communicating processes with value-passing. *Information and Computation*, 107:202–236, 1993.

[HL95] M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138:353–369, 1995.

[Hoa85] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mon76] J. Monk. *Mathematical Logic*. Springer-Verlag, New York, 1976.

[MP05] S. Maffeis and I. Phillips. On the computational strength of pure ambient calculi. *Theoretical Computer Science*, 330:501–551, 2005.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–40 (Part I), 41–77 (Part II), 1992.

[MS92] R. Milner and D. Sangiorgi. Barbed bisimulation. In *Proc. ICALP'92*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695, 1992.

[Pal03] C. Palamidessi. Comparing the expressive power of the synchronous and the asynchronous π-calculus. *Mathematical Structures in Computer Science*, 13:685–719, 2003.

[Par81] D. Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science*, volume Lecture Notes in Computer Science 104, pages 167–183. Springer, 1981.

[Pos44] Post. Recursively enumerable sets of positive integers and their decision problems. *Bulletin of the American Mathematical Society*, 50:284–316, 1944.

[Pre29] M. Presburger. Über die vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervortritt. In *Warsaw Mathematics Congress*, volume 395, pages 92–101, 1929.

[Pri78] L. Priese. On the concept of simulation in asynchronous, concurrent systems. *Progress in Cybernatics and Systems Research*, 7:85–92, 1978.

[Rog87] H. Rogers. *Theory of Recursive Functions and Effective Computability*. MIT Press, 1987.

[San92] D. Sangiorgi. *Expressing Mobility in Process Algebras: First Order and Higher Order Paradigm*. PhD thesis, Department of Computer Science, University of Edinburgh, 1992.

[San93] D. Sangiorgi. From π-calculus to higher order π-calculus–and back. In *Proc. TAPSOFT'93*, volume 668 of *Lecture Notes in Computer Science*, pages 151–166, 1993.

[Soa87] R. Soare. *Recursively Enumerable Sets and Degrees, a study of computable functions and computably generated sets*. Springer, 1987.

[Sta84] R. Stansifer. Presburger's article on integer arithmetic: Remarks and translation. Technical report, Technical report, Computer Science Department, Cornell University, 1984.

[SWU10] P. Sewell, P. Wojciechowski, and A. Unyapoth. Nomadic pict: Programming languages, communication infrastructure overlays, and semantics for mobile computation. *ACM Tansactions on Programming Languages and Systems*, 32:1–63, 2010.

[Tho95] B. Thomsen. A theory of higher order communicating systems. *Information and Computation*, 116:38–57, 1995.

[vEB90] P. van Emde Boas. Machine models and simulations. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Algorithm and Complexity, volume A*, pages 65–116. Elservier, 1990.

[vGLT09] R. van Glabbeek, B. Luttik, and N. Trčka. Branching bisimilarity with explicit divergence. *Fundamenta Informaticae*, 93:371–392, 2009.

[vGW89] R. van Glabbeek and W. Weijland. Branching time and abstraction in bisimulation semantics. In *Information Processing'89*, pages 613–618. North-Holland, 1989.

[Wal91] D. Walker. π-calculus semantics for object-oriented programming languages. In *Proc. TACS '91*, volume 526 of *Lecture Notes in Computer Science*, pages 532–547, 1991.

[Wal95] D. Walker. Objects in the π-calculus. *Information and Computation*, 116:253–271, 1995.