A Denotational Semantics for the π-Calculus

B. Aziz  G.W. Hamilton

School of Computer Applications
Dublin City University
Dublin 9, Ireland
{baziz,hamilton}@compapp.dcu.ie

Abstract

In this paper, we substantiate Stark's semantic framework for the π-calculus and provide a fully defined denotational semantics. In his categorical framework, Stark defines an abstract domain-theoretic model of name-passing processes based on functor categories. Despite being a sound model, we believe that a more concrete ground is essential for further work involving program analyses like the abstract interpretation analysis. We also give an example of such abstractions.

1 Introduction

Traditionally, the meaning of a process in the π-calculus [15] was specified by means of the structural operational semantics where processes are transformed into a suitable form for communication and then interactions are controlled through a transition system. Despite being intuitively simple for small systems, the structural operational approach does not scale and lacks mathematical rigour. Tasks such as comparing the meaning of different processes are difficult to carry out without the assistance of the complex notions of bisimilarity and equivalence. Also, building certain program analyses, like the abstract interpretation analysis [9, 10], on top of the structural operational semantics becomes hard due to the fact that this semantics is a small-step semantics (Interestingly, [17] deals with this issue).

A mathematically more rigorous alternative is to adopt the denotational approach in defining the meaning of a process. This meaning is given as an element in a proper domain using some semantic function, which maintains properties of the translated process. This approach also simplifies any abstractions over the semantic domain and makes the task of reasoning about programs and proving their correctness much easier than in the case of the structural operational approach.

In this paper, we substantiate Stark’s categorical framework of [18] by providing a fully defined denotational semantics for the π-calculus. In his framework, Stark presents a domain-theoretic model of name-passing processes that is
based on functor categories. The model is shown to be fully abstract and capable of capturing the operational notions of transitions, strong late bisimilarity, and strong equivalence in the $\pi$-calculus. Being of categorical nature, it also provides a syntax-free handling of names, visibility, privacy, and interference.

We give concrete definitions of important process-transforming morphisms, which we believe provide the necessary concrete ground for future work, particularly in the area of abstract interpretation analysis. Such analyses normally require building certain algorithms (or a set of rules), which are themselves built on top of a, possibly, extended semantics. Building any extensions and analyses would necessarily require a concrete ground to start with.

In the course of defining this semantics, we also present a solution to the problem of tracing the origins of names. In [18], the origin of a name is lost as a result of applying $\alpha$-conversion. Experience has shown that this information is essential in a program analysis that involves determining whether confidential data can be leaked [2, 3]. Therefore, we extend the binding mechanism of [18] to be able to keep track of the origin of names by using a variable renaming mechanism similar to the De Bruijn renaming scheme used in $\lambda$-abstractions [5].

The rest of the paper is structured as follows. In Section 2, we review related work in the area of denotational semantics of the $\pi$-calculus. In Section 3, we give an overview of the syntax and the structural operational semantics of the $\pi$-calculus. This is necessary in inspiring the categorical and concrete definitions of Section 4, which are then used in defining the denotational semantics of the $\pi$-calculus in Section 5. In Section 6 we give one example of abstractions that may manipulate the semantics of Section 5. Finally, we conclude in Section 7 and give directions for future work.

## 2 Related Work

Despite the fact that most of the work in the area of the $\pi$-calculus has used the structural operational semantics, there have been a number of attempts to try and define other semantics for the calculus [6, 8, 11, 12, 13]. Most of these have adopted an abstract categorical approach, with little or no concrete ground.

The model presented in [11] is very close to that of [18]. The major difference is that [11] defines the denotational semantics in terms of a metalanguage, extracted from the categorical model. Therefore, it is a more abstract framework that relies on a uniform approach capable of giving semantics to a variety of calculi. However, like [18], the work of [11] only handles strong late bisimulation and lacks definitions for other forms of equivalence, like early and open bisimulation and weak bisimulation, which partially ignores internal actions.

The approach of [13] is not much different either. Two set-theoretic denotational models are presented that are fully abstract with respect to must and may testing. The models are obtained as solutions to domain equations in a functor category.

In [12], a denotational semantics of a higher order version of the $\pi$-calculus is presented, where not only ground type messages like channels can be exchanged,
but also functions into processes. This approach is interesting since it comes closer to the meaning of code mobility. The model is also based on a solution to a single domain equation in a functor category and is shown to be computationally adequate.

The approach of [8] is slightly different than the above. A flexible model for the π-calculus is presented which is based on presheaf categories. Advantages of this approach include support for function spaces, as well as other traditional models like synchronisation, trees, and event structures, as well as a better understanding of bisimulation in the π-calculus. This is because presheaf categories come along with the concept of drawing bisimulation from open maps [14], which yields an easier definition of bisimulation.

Finally, deviating from the denotational approach, [6] defines a Petri net semantics of the π-calculus based on P/T nets with inhibitor arcs and is proven to be sound with respect to early transition semantics.

3 The π-Calculus

The π-calculus [15] is a compact notation used in specifying systems of mobile and distributed processes. The notion of a name is central to the calculus, where a name stands for data as well as communication channels. Mobility of processes is modelled as the movement of links in the virtual world of linked processes. This movement is achieved by allowing processes to send and receive names (including channel names) over communication channels. Hence, a process can dynamically exchange links with other processes during its lifetime.

Simple but expressive, the π-calculus is capable of modelling dynamic changes in systems topology by modelling mobility. It is this and other features, like scoping and non-determinism, that give the π-calculus its power and simplicity making it one of the most popular formalisms used in specifying mobile distributed systems today.

3.1 Syntax

We adopt the standard monadic version of the π-calculus. \( P, Q, \ldots \in \mathcal{P} \) are processes specified by the grammar of Figure 1.

A null process \( 0 \) is an inactive process that is unable to perform any actions. A guarded process \( \pi.P \) performs an atomic action \( \pi \) and continues as \( P \). An atomic action can be an input, output, or internal action. An input action \( x(y).P \) substitutes a message \( z \) received over \( x \) for each free occurrence of \( y \) in \( P \) and continues as \( P[z/y] \). An output action \( \pi(y).P \) sends \( y \) over \( x \) and continues as \( P \). An internal action \( \tau.P \) is not externally observable; the process reacts internally and continues as \( P \). Restriction \((\nu x)P\) creates a fresh name \( x \) within the scope of \( P \). Finally, Replication \(!P\) creates as many parallel copies of \( P \) as required.

The names \( y \) in \( x(y).P \) and \( x \) in \((\nu x)P\) are bound in \( P \). A name is free if it is not bound. The set \( bn(P) \) is the set of bound names and \( fn(P) \) the set of
free names of \( P \). The finite set of names of a process \( P \), \( n(P) \subset N = bn(P) \cup fn(P) \).

### 3.2 Structural Operational Semantics

The original semantics of the \( \pi \)-calculus introduced in [15] was a structural operational semantics inspired by Berry and Boudol’s chemical abstract machine [4]. This semantics relies on two main relations: the structural congruence and the reduction relations. The structural congruence relation \( \equiv \) is defined as the least congruence satisfying the following rules:

- Processes are identified up to the renaming of bound names (\( \alpha \)-conversion).
- \( (P/ \equiv, +, 0) \) and \( (P/ \equiv, |, 0) \) are commutative monoids.
- \( (\nu x)0 \equiv 0 \)
- \( (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P \)
- \( !P \equiv (!P | P) \)
- \( (\nu x)(P | Q) \equiv P | (\nu x)Q, \text{ if } x \notin fn(P) \)

On the other hand, the reaction relation \( \rightarrow \) is defined by the following transition system:

\[
\begin{align*}
\text{(COMM): } & (\pi(z).P) | (x(y).Q) \rightarrow (P | Q[z/y]) & \text{(RES): } & P \rightarrow P' \\
\text{(PAR): } & P \rightarrow P' & \frac{}{(P | Q) \rightarrow (P' | Q)} & \text{(STRUCong): } & P \equiv P' \rightarrow Q' \equiv Q \\
\end{align*}
\]
The axiom of communication (\textsc{Comm}) defines the essence of communication in the $\pi$-calculus. A process willing to send a message over some channel can communicate with another process running in parallel and willing to receive a message over the same channel. The two rules, (\textsc{Par}) and (\textsc{Res}) allow a process to evolve under the parallel composition and restriction operators respectively. Finally, the structural congruence rule (\textsc{StruCong}) is included to allow congruent processes to evolve congruently.

4 A Categorical Framework

Apart from the structural operational semantics, some efforts have followed the denotational approach in defining this semantics. Of these, we have chosen Stark’s categorical framework [18]. The motivation behind this choice is the relative simplicity of the approach adopted in [18] compared to other approaches [11, 12, 13, 8]. The approach is also relevant to previous analyses [2, 3].

4.1 Abstract Concepts

The model Stark presents is mainly motivated by the solution to the following predomain equations, which define a $\pi$-calculus process in terms of the different actions it can perform:

\begin{align*}
\Pi_t &\cong 1 + P(\Pi_{\perp} + In + Out) \quad (1) \\
In &\cong N \times (N \rightarrow \Pi_{\perp}) \quad (2) \\
Out &\cong N \times (N \times \Pi_{\perp} + N \rightarrow \Pi_{\perp}) \quad (3)
\end{align*}

Where $\Pi_{\perp}$ is the lifted domain of processes, $In$ and $Out$ are the domains of input and output actions respectively, and $N$ is the domain of names. $P(-)$ is a power operation and $(N \rightarrow \Pi_{\perp})$ is a non-standard exponential supplying a fresh name to a process.

In order to solve these equations and to capture the fact that a process in the $\pi$-calculus is defined over a finite set of free names that may change during the lifetime of that process, a functor category $\mathcal{B}$ is first chosen. $\mathcal{B}$ is the category of bifinite predomains and continuous maps described in [16] and $\mathcal{I}$ is the category of finite sets of names, $s$, and injections between those sets, $f : s \rightarrow s'$. $\Pi_{\perp}$ is then defined as the functor $\mathcal{I} \rightarrow \mathcal{B}$, where $\Pi_{\perp}$ signifies the domain of processes with free names in $s$, and $\Pi_{\perp}f : \Pi_{\perp}s \rightarrow \Pi_{\perp}s'$ is a relabelling operator. A category $\mathcal{C}$ is then chosen within $\mathcal{B}$ as a full subcategory of functors that are pullback-preserving and bifinite as functors. $\mathcal{C}$ is then shown to be a symmetric monoidal closed category.

Based on this structure, the details of the solution to (1)–(3) above are presented in a categorical context. First, the object of names $N$ is defined within $\mathcal{C}$ as the inclusion taking a set of names $s$ to the discrete predomain $s$. Then $\Pi_{\perp}$, $In$, and $Out$ are obtained by unfolding the above equations and manipulating the definitions of $N$, $P(-)$, and $\rightarrow$ (see [16, 18]). Despite the
soundness of this solution, our main interest remains in defining concretely (i.e. over some $s$) the means by which the finite process elements $p, q, \ldots$ of the domain $\Pi_s$ can be obtained, and ultimately obtaining a concrete semantics of the $\pi$-calculus. To be able to do this, a number of morphisms are first introduced [18]. These are shown in Figure 2 below.

$$
\emptyset : 1 \to \Pi_s \\
\exists : \Pi_s \times \Pi_s \to \Pi_s \\
\parallel : \Pi_s \times \Pi_s \to \Pi_s \\
\new : (N \to \Pi_s) \to \Pi_s \\
\{|\cdot\|\} : (\Pi_s + In + Out)_s \to \Pi_s
$$

Figure 2: Morphisms into $\Pi_s$.

The $\emptyset$ morphism takes the null process $\emptyset$ to become an element of $\Pi_s$ while $\exists$ is a standard powerdomain union representing a non-deterministic choice between two processes. The parallel composition morphism $\parallel$ interleaves two processes, whereas the $\new$ morphism is used to interpret restriction. Finally, the singleton map $\{|\cdot\|\}$ defines elements of the domain $\Pi_s$ in terms of input/output actions. The $\{|\bot\|\}$ is the least element of $\Pi_s$ representing the undetermined process, where $\{|\bot\|\} \sqsubseteq \emptyset$ and $\emptyset$ is incomparable otherwise.

These definitions are still too abstract to be of any use in specifying, concretely, the meaning of a process in the $\pi$-calculus. Also, in their current form, these morphisms cannot be used in further work involving concrete domains. Therefore, we next provide a more concrete ground over which the semantics of the $\pi$-calculus can be built and analysed.

### 4.2 Concrete Definitions

The definitions of the morphisms of Figure 2 can now be substantiated over some $s \in I$. These concrete definitions are inspired by the structural operational semantics of Section 3.2 and will help define the denotational semantics of next section.

It is important to note at this point that name binding is expressed in [18] as a $\lambda$-abstraction of the form $\lambda x.p$, where the name $x$ is bound in $p$. Hence, an input action $x(y).P$ is represented as $in(x, \lambda y.p)$ and a bound output action as $out(x, \lambda y.p)$, where the name $y$ remains within the scope of $p$. The remaining free output and silent actions are represented as $out(x, y, p)$ and $\tau p$, respectively.

The $in$, $out$, and $\tau$ actions do not themselves represent elements of $\Pi_s$, but are rather supplied to $\{|\cdot\|\}$ to yield $p \in \Pi_s$. The behaviour of $\{|\cdot\|\}$, as well as those for $\emptyset$ and $\exists$, is obvious from their abstract definitions, since they do not rely on any equational transformations of a process $p$ over $s$. On the other hand, the two most important concrete definitions are those of $\new$ and
Both of these morphisms may change the structure of a process and the set $s$ of names over which it is defined.

First, we define $\text{new}_s$ by the set of rules of Figure 3. The first rule deals with the case of free output. A bound name $x$ is compared with other names of the output action. If $x$ is equal to the name of the channel $y$, then communication is impossible (hence the null morphism). Alternatively, if $x$ is equal to the name of the free message $z$, the free output becomes bound. Finally, in the case when $x$ is neither equal to $y$ nor $z$, interpreting the restriction should be resumed on the body of the output action $p$.

| $\text{new}_s(\lambda x.\{\text{out}(y,z,p)\})$ | $\emptyset$, if $x = y$
| | $\{\text{out}(y,\lambda x.p)\}$, if $x = z \neq y$
| | $\{\text{out}(y,z,\text{new}_s(\lambda x.p))\}$, otherwise |
| $\text{new}_s(\lambda x.\{\text{out}(y,\lambda z.p)\})$ | $\emptyset$, if $x = y$
| | $\{\text{out}(y,\lambda z.p)\}$, if $x = z \neq y$
| | $\{\text{out}(y,\lambda z.\text{new}_s(\lambda x.p))\}$, otherwise |
| $\text{new}_s(\lambda x.\{\text{in}(y,\lambda z.p)\})$ | $\emptyset$, if $x = y$
| | $\{\text{in}(y,\lambda z.p)\}$, if $x = z \neq y$
| | $\{\text{in}(y,\lambda z.\text{new}_s(\lambda x.p))\}$, otherwise |
| $\text{new}_s(\lambda x.\{\tau(p)\})$ | $\{\tau(\text{new}_s(\lambda x.p))\}$ |
| $\text{new}_s(\lambda x.\emptyset)$ | $\emptyset$ |
| $\text{new}_s(\lambda x.(p_1 \uplus p_2))$ | $\text{new}_s(\lambda x.p_1) \uplus \text{new}_s(\lambda x.p_2)$ |

Figure 3: The definition of $\text{new}_s$

The second and third rules deal with bound output and input actions, respectively. Again the result is $\emptyset$ if $x = y$, else binding the bound parameter does not change anything. Finally, the restriction is resumed on the body $\lambda z.p$ otherwise. The rule for the silent action is inspired by the $(\text{Res})$ rule of the transition system (Section 3.2). Hence, interpreting restriction can still occur even after the process has evolved. Similarly, the rule of the null morphism is inspired by the structural congruence rule of $(\nu x)\emptyset \equiv \emptyset$. Finally, the last rule distributes restriction over the non-deterministic summation morphism.

The definition of $\text{par}_s$ is shown in Figure 4. This definition makes use of two auxiliary operations: $\text{lpar}_s$ and $\text{lcom}_s$. $\text{lpar}_s$ is the prioritised parallel composition; first $p$ does a transition, then $q$ interleaves with the residue. It is non-strict in its right argument to ensure the resulting fixpoint is a fully determined process. One may think of the $\text{lpar}_s$ operation as the $(\text{PAR})$ rule, which states that a process may evolve under parallel composition.

The first case is that when $p$ is capable of reacting internally. In this sit-
ation, \( \text{par}_s \) is interpreted as a process which does an internal reaction and resumes again with the parallel composition between the residue of \( p \) and \( q \). The second case distributes \( \text{par}_s \) over the non-deterministic summation morphism. Finally, in the event that \( p \) is unable to react internally (e.g. it is a guarded process), the result is a null morphism.

The \( \text{lcom}_s \) operation represents communication from \( p \) to \( q \) after which the residues are interleaved. The operation may be thought of in light of the axiom of communication (\( \text{Comm} \)), where it is defined for all the possible cases of free output and input, bound output and input, non-deterministic summation, and the case where communication is impossible between \( p \) and \( q \). Note that in the case of bound output, the name \( y \) is added to \( s \), as this is assumed to be a distinct name not belonging to \( s \).

\[
\text{par}_s(p, q) = \text{par}_s(p, q) \uplus \text{par}_s(q, p) \uplus \text{lcom}_s(p, q) \uplus \text{lcom}_s(q, p)
\]

where,

\[
\text{par}_s(p, q) = \begin{cases} 
\{ \tau(\text{par}_s(p', q)) \}, & \text{if } p = \{ \tau(p') \} \\
\text{par}_s(p_1, q) \uplus \text{par}_s(p_2, q), & \text{if } p = p_1 \uplus p_2 \\
\emptyset, & \text{otherwise}
\end{cases}
\]

and

\[
\begin{align*}
\text{lcom}_s(\{ \text{out}(x, y, p) \}, \{ \text{in}(x, \lambda z. q) \}) &= \{ \tau(\text{par}_s(p, q[y/z])) \} \\
\text{lcom}_s(\{ \text{out}(x, \lambda y. p) \}, \{ \text{in}(x, \lambda z. q) \}) &= \{ \tau(\text{new}_s(\lambda y. \text{par}_s[p, q[y/z]])(p, q[y/z])) \} \\
\text{lcom}_s(p_1 \uplus p_2, q) &= \text{lcom}_s(p_1, q) \uplus \text{lcom}_s(p_2, q) \\
\text{lcom}_s(\{ p \}, \{ q \}) &= \emptyset
\end{align*}
\]

Figure 4: The definition of \( \text{par}_s \)

## 5 Denotational Semantics

Employing the concepts and definitions of last section, we can now give a fully defined denotational semantics for the standard monadic version of the \( \pi \)-calculus. The semantic relation \( \mathcal{C}(P)_s : N \rightarrow P_{\uplus} s \) is defined inductively over a process \( P \), with free names in \( s \), as shown in Figure 5.

The first rule interprets the meaning of the null process as the null morphism. The next three rules interpret the input, free output, and silent actions as the processes resulting from the corresponding actions. We have chosen the free output only to adhere to the syntax of Figure 1, but the case of bound output also applies. The next cases of non-deterministic summation and parallel composition are interpreted with the corresponding morphisms, whereas
a replicated process is interpreted as the fixpoint calculation of the prioritised parallel composition.

One change that has been made to the case of restriction from its original definition in [18] is the use of a mechanism for explicit renaming. In [18], a special underline is used to indicate that a bound name has to be fresh. So, $\lambda x.p$ assumes not only that $x$ is bound to $p$, but is also fresh (which with the aid of $\alpha$-conversion, is also distinct from any other name).

The problem arising from this idea is that it is not possible to trace the origin of names lost as a result of $\alpha$-conversion. Such information, may turn out to be quite important when analysing a process for instances where leakage of confidential data may occur [2, 3]. Therefore, it becomes necessary to keep track of name origins by using a simple, but effective, mechanism for explicit renaming similar to the mechanism used by De Bruijn in numbering the formal parameters in the $\lambda$-abstractions [5].

Assuming that all names are initially distinct, we will rename the first occurrence of a name as $x_0$, the second as $x_1$, and so on. The origin of a name $x_i$ will hence be always obtained from the name stem $x$. The newly created names of the form $x_i$ are also added to the previous set of names as $s + \{x_i\}$ (the subscript was omitted last section because of indifference).

\[
\begin{align*}
C[0]_s &= \emptyset \\
C[x(y).P]_s &= \{ \text{in}(x, \lambda y.C[P]_{s+(y)}) \} \\
C[\pi(y).P]_s &= \{ \text{out}(x, y, C[P]_s) \} \\
C[\tau.P]_s &= \{ \text{tau}(C[P]_s) \} \\
C[P + Q]_s &= C[P]_s \uplus C[Q]_s \\
C[P \mid Q]_s &= \text{par}_{s}(C[P]_s, C[Q]_s) \\
C[(\nu x)P]_s &= \text{new}_{s}(\lambda x_i.C[P[x_i/x]]_s+\{x_i\}) \\
&\text{where, } i = \min\{i \mid i \in \mathbb{N} \land x_i \notin s\} \\
C[\!(P)\!]_s &= \mu p.\text{par}_{s}(C[P \mid P]_s, p)
\end{align*}
\]

Figure 5: Denotational Semantics of the $\pi$-Calculus.

In [18], this semantics is shown to capture the notions of strong late bisimilarity and strong equivalence by showing that the semantics both reflects and preserves transitions in the structural operational semantics.
6 An Abstraction Example

One area of research where the semantics of Figure 5 can be found useful is the abstract interpretation analysis [9, 10] area, where safely approximate properties of the analysed program are extracted based on some abstraction to the semantics of the language. This abstraction is naturally directed to assist the aim of the analysis and may be applied to an extended set of the semantics, where extra information is added relevant to the particular properties the analysis is trying to capture (example of this extra information is the environment of unsafe names of [2]).

One example of such abstractions is the one we adopted throughout [2, 3] where it makes use of integer constraints to restrict the number of names, which can be generated within the semantics. This constraint is needed to ensure the termination of fixpoint calculations whenever replicated processes are involved. For example, the processes $!(\nu x)P$ has the problem that the set of names over which it is defined may grow infinitely. However, using the constraint $k \in \mathbb{N}$, the number of separate instances of the name $x$ is restricted to $k$ (i.e. $x_0 \ldots x_{k-1}$), where $k = 1$ renders a uniform analysis.

The domain of names $N$ of Section 4.1 will now contain names that have been constrained to $k$ instances. This yields a new finite domain of names $N_k$. To arrive at $N_k$, an abstraction function $\alpha_k : \mathbb{N} \times N \rightarrow N_k$ is defined as follows:

$$\alpha_k(x_i) = \begin{cases} x_i, & \text{if } i < k \\ x_{k-1}, & \text{otherwise} \end{cases}$$

This abstraction allows the analysis to be carried out over a finite domain of names and thereby is decidable. It also has the additional advantage of keeping the abstract semantics as close as possible to the concrete (possibly extended) semantics and hence keeping the analysis simple.

7 Conclusion and Future Directions

In conclusion, we have presented a concrete definition of the different morphisms introduced by Stark in his categorical framework [18]. These definitions, we believe, are necessary for any program analysis that rely on the concrete semantics of the $\pi$-calculus. Examples of such analysis include previous work, where we presented abstract interpretation analyses for detecting instances of private information leakage for open systems [2], and with security levels [3].

We hope in future work to define denotational semantics of other models, such as the Spi calculus [1] and the Mobile Ambients [7]. Also, other security analyses may be built, for example, to detect denial of service attacks. Such analyses would require extending the semantics with information about liveness.
References


