A First-Order Syntax for the π-Calculus in Isabelle/HOL using Permutations

Christine Röckl

LAMP – DI – EPFL, INR Ecublens, 1015 Lausanne, Switzerland.
christine.roeckl@epfl.ch

Abstract. A formalized theory of alpha-conversion for the π-calculus in Isabelle/HOL is presented. Following a recent proposal by Gabbay and Pitts, substitutions are modelled in terms of permutations, and alpha-equivalence is defined over all but finitely many names. In contrast to the work by Gabbay and Pitts, however, standard universal and existential quantification are used instead of introducing a new binder.
Further, a classification of the various approaches to formalizing languages with binders is presented. Strengths and weaknesses are pointed out, and suggestions for possible applications are made.

1 Introduction

The π-calculus is a model of higher-order concurrent programming languages [3, 18, 22, 31], and as such is particularly characterized by its binders, *input*, $ax.P$, and *restriction*, $(\nu x)P$. As an example, consider the processes $P \stackrel{def}{=} !az.zx_1.zx_2.P'$ and $C \stackrel{def}{=} (\nu y)\bar{a}y.\bar{y}b.\bar{y}c.C'$. Process P is a procedure[1] that each time it is called along channel a and transmitted two arguments, launches an instantiation of its body P'. Process C, on the other hand, models a client creating a new channel y over which it then transmits its parameters b and c to P. Assuming that z does not occur in P' and y does not occur in C', the following communication can be established between the two processes (modulo strong bisimilarity $\sim$):

$$\begin{array}{lll} P \,|\, C & \stackrel{\tau}{\rightarrow} \; \sim P \,|\, (\nu y)(yx_1.yx_2.P' \,|\, \bar{y}b.\bar{y}c.C') & C \text{ calls } P \text{ along } a, \\ & \stackrel{\tau}{\rightarrow} \; P \,|\, (\nu y)(yx_2.P'\{b/x_1\} \,|\, \bar{y}c.C') & C \text{ transmits } b \text{ to } P', \\ & \stackrel{\tau}{\rightarrow} \; \sim P \,|\, P'\{b/x_1, c/x_2\} \,|\, C' & C \text{ transmits } c \text{ to } P'. \end{array}$$

This simple mechanism of creating and transmitting new names is the source of the expressive power of the π-calculus, but makes reasoning extremely intricate, necessitating an application of interactive theorem-proving.

The first question to be decided in formalizations of languages with binders, like the π-calculus, is which syntax to build on. There exist three general approaches,

[1] The definition of procedures is a typical application of *replication* $!P$, intuitively describing an unlimited number of copies of P. Often replication is used in an *input-bounded* form, $!az.P$, where a can be understood as the name of the procedure.

each of which gives rise to a number of variations: (1) first-order syntax in a deep embedding, (2) higher-order syntax in a deep embedding, and (3) higher-order syntax in a shallow embedding. Note that first-order syntax always yields a deep embedding. Traditionally, deep embeddings are considered to be well-understood and are therefore often applied in syntax analysis and meta-theoretical justifications. Reasoning about concrete systems of larger size, on the other hand, entails cumbersome definitions as well as a diffusing administration of bound variables. Therefore, shallow embeddings have become more and more popular, motivated both by a growing theoretical basis and the fact that they free the user from struggling with bound variables, the latter being particularly essential in reasoning about concrete terms (programs or processes). On the other hand, reasoning about syntax in a shallow embedding either requires a logical framework that is sufficiently weak yet incorporates non-standard axioms ([16, 17] presents such a framework implemented for the π-calculus in Coq), or else a heavy machinery eliminating exotic terms and mimicking the missing structural induction principles by rule induction over well-formedness predicates ([28, 29] presents such a framework for the π-calculus in Isabelle/HOL). Note that the actual choice of a specific syntactic framework heavily relies on the application one has in mind.

In this paper, we study a formalization of the π-calculus in Isabelle/HOL [27, 25] using a first-order syntax[2]. It is part of a larger project aiming at theorem-prover support for reasoning within and about the π-calculus and related languages. The main motivation for the work at hand was to see how conveniently the permutation model can be applied to a formalization of the π-calculus using Isabelle/HOL (so far, Gabbay and Pitts have used Isabelle/ZF as a framework to justify their approach and apply it to the λ-calculus, see [8] for details), and to relate it to approaches investigated previously. The proposition of Gabbay and Pitts [8, 9] is based on two ideas: (1) model substitutions in terms of permutations and (2) reason about *all but finitely many* instantiations. As a slight modification, we do not introduce a new operator for idea (2) but implement it in terms of universal and existential quantification. Note that idea (2) is independent of idea (1); it can be applied together with other notions of substitution alike.

A second aim of this paper is to generally classify frameworks for languages with binders, based on our experience with the π-calculus. That is, we give characterizations of the diverse techniques, discussing their strengths and weaknesses, and specify areas in which they can typically be applied. Although originally obtained for the π-calculus in Isabelle/HOL, these results should be equally valid for formalizations of other languages with binders in comparable frameworks.

Overview In Section 2, we describe the basic features of Isabelle/HOL. In Section 3, we introduce the syntax of the π-calculus and model substitutions by permutations, in Isabelle/HOL. In Section 4, we derive a theory of α-equivalence using *all but finitely many* instantiations. Section 5 discusses the diverse approaches providing guidelines for applications. Section 6 concludes the paper.

[2] The proof-scripts are available at `http://lampwww.epfl.ch/~roeckl/Pi/`. Some extracts can also be found in the appendix.

2 Isabelle/HOL

We use the general-purpose theorem-prover Isabelle [27], based on higher-order intuitionistic logic, and formalize the π-calculus in its instantiation HOL for higher-order logic [25]. Proofs in Isabelle are based on unification, and are usually conducted in a backward-resolution style: the user formulates the goal he/she intends to prove, and then—in interaction with Isabelle—continuously reduces it to simpler subgoals until all of the subgoals have been accepted by the tool. Upon this, the goal can be stored in the theorem-database of Isabelle/HOL to be applicable in further proofs. The prover offers various tactics, most of them applying to single subgoals. The basic resolution tactic `resolve_tac`, for instance, allows the user to instantiate a theorem from Isabelle's database so that its conclusion can be applied to transform a current subgoal into instantiations of its premises. Besides these *classical tactics*, Isabelle offers *simplification tactics* based on algebraic transformations. Powerful *automatic tactics* apply the basic tactics to prove given subgoals according to different heuristics. The automatic solver `auto_tac`, for example, combines classical reasoning and simplification over sets of rules that can be modified by the user. The tactic

```
(auto_tac (claset() addIs [nren_trans], simpset() delsimps [nren_def]))
```

for example, adds the introduction rule `nren_trans` to the set of classical rules, and removes the rewrite rule `nren_def` from the set of simplification rules. When adding definitions and theorems, however, the user has to take care not to provoke unprovable subgoals or infinite loops.

In Isabelle/HOL, the user can define recursive datatypes and inductive sets. The prover then automatically computes rules for induction and case-injection. Note that all these techniques have been fully formalized and verified in Isabelle/HOL, that is, they are a conservative generic extension [2, 26]. Definitions and proofs are usually given on an *object-level* (HOL), but can employ *meta-level* (Isabelle) functions. This gives rise to *shallow embeddings* as opposed to *deep embeddings* fully residing on the object-level [2].

3 The Syntax of the π-Calculus

The π-calculus [23, 24] is the mobile counterpart of CCS, reducing its predecessor to basics by identifying the sort of messages and channels, referring to both as *names*. This gives processes the possibility to declare new names and send them to other processes, thus create private communication lines with them. It is this syntactic simplicity, from which the π-calculus draws the power to model mobile communication systems [23] and higher-order languages [3, 18, 22, 31].

Names Consider an at least countably infinite set of *names*, ranged over by $a, b, \ldots, x, y, \ldots$. In our formalization, we have not chosen a particular type, but axiomatically accept every type that is at least countably infinite. We have

$$
\begin{aligned}
fn(0) &= \emptyset & bn(0) &= \emptyset\\
fn(\tau.P) &= fn(P) & bn(\tau.P) &= bn(P)\\
fn(\bar{a}b.P) &= \{a,b\} \cup fn(P) & bn(\bar{a}b.P) &= bn(P)\\
fn(ax.P) &= \{a\} \cup (fn(P) - \{x\}) & bn(ax.P) &= \{x\} \cup bn(P)\\
fn((\nu x)P) &= fn(P) - \{x\} & bn((\nu x)P) &= \{x\} \cup bn(P)\\
fn(P+Q) &= fn(P) \cup fn(Q) & bn(P+Q) &= bn(P) \cup bn(Q)\\
fn(P \,|\, Q) &= fn(P) \cup fn(Q) & bn(P \,|\, Q) &= bn(P) \cup bn(Q)\\
fn([a=b]P) &= \{a,b\} \cup fn(P) & bn([a=b]P) &= \{a,b\} \cup bn(P)\\
fn([a \neq b]P) &= \{a,b\} \cup fn(P) & bn([a \neq b]P) &= \{a,b\} \cup bn(P)\\
fn(!P) &= fn(P) & bn(!P) &= bn(P)
\end{aligned}
$$

Table 1. Names of processes. Free and bound names are computed by primitively recursive functions. The names of a process are $n(P) \stackrel{def}{=} fn(P) \cup bn(P)$.

adopted this approach already in [28, 29], leaving the possibility for specific instantiations, such as with naturals or reals. Naturals, for instance, can increase automation in certain cases, because fresh names can be computed by determining the maximal name in the involved processes and choosing the successor.

Processes Implementing communications as basic primitive, the calculus possesses *prefixes* $\pi ::= \tau \mid \bar{a}b \mid ax$, which are *silent*, that is, invisible, *output*, and *input* prefixes, respectively. Processes are then built from *inaction*, *prefix*, *restriction*, *choice*, *parallel composition*, *matching*, *mismatching*, and *replication*:

$$P \;::=\; 0 \mid \pi.P \mid (\nu y)P \mid P+Q \mid P \,|\, Q \mid [a=b]P \mid [a \neq b]P \mid !P.$$

The actual syntax may vary from one application to another. Often, mismatching is omitted due to its semantic misbehaviour in various cases, or output prefixes may stand for themselves only, as in asynchronous π-calculi. We formalize the full syntax to be comparable with previous works [17, 28, 29].

Binders The π-calculus has two binders, *input prefix* and *restriction*, for which the notion of boundedness is slightly distinct, however. A name x bound in an input $ax.P$ can be considered as a place-holder for a name b to be received in a communication $ax.P \,|\, \bar{a}b.Q \xrightarrow{\tau} P\{b/x\} \,|\, Q$ later on. A name y bound in a restriction $(\nu y)P$, on the other hand, rather represents a private channel, that is, a name that is only known to P and is inaccessible to all processes running in the environment. On a more nominal level, this exclusiveness could also be explained as y being distinct from any name currently in use.

Free and bound names In order to compute the *free* and *bound names* of a process P, we use standard primitively recursive functions $fn(P)$ and $bn(P)$. The *names* of a process, $n(P)$, are then simply the union of its free and bound names. For instance, $fn(ax.P) = \{a\} \cup (fn(P) - \{x\})$ and $bn(ax.P) = \{x\} \cup bn(P)$. See Table 1 for a complete overview. Note that we are able to compute the

$$
\begin{array}{rcl}
(0)\{x\leftrightarrow y\} & = & 0 \\
(\tau.P)\{x\leftrightarrow y\} & = & \tau.P\{x\leftrightarrow y\} \\
(\bar{a}b.P)\{x\leftrightarrow y\} & = & \overline{a\{x\leftrightarrow y\}}b\{x\leftrightarrow y\}.P\{x\leftrightarrow y\} \\
(ab.P)\{x\leftrightarrow y\} & = & a\{x\leftrightarrow y\}b\{x\leftrightarrow y\}.P\{x\leftrightarrow y\} \\
((\nu b)P)\{x\leftrightarrow y\} & = & (\nu b\{x\leftrightarrow y\})P\{x\leftrightarrow y\} \\
(P+Q)\{x\leftrightarrow y\} & = & P\{x\leftrightarrow y\}+Q\{x\leftrightarrow y\} \\
(P\,|\,Q)\{x\leftrightarrow y\} & = & P\{x\leftrightarrow y\}\,|\,Q\{x\leftrightarrow y\} \\
([a=b]P)\{x\leftrightarrow y\} & = & [a\{x\leftrightarrow y\}=b\{x\leftrightarrow y\}]P\{x\leftrightarrow y\} \\
([a\neq b]P)\{x\leftrightarrow y\} & = & [a\{x\leftrightarrow y\}\neq b\{x\leftrightarrow y\}]P\{x\leftrightarrow y\} \\
(!P)\{x\leftrightarrow y\} & = & !P\{x\leftrightarrow y\}
\end{array}
$$

Table 2. Substitution. Permuting on names, $a\{x\leftrightarrow y\} =$ if $a = x$ then y else if $a = y$ then x else a, substitution can disregard whether a name is free or bound.

bound names only because we are working with a deep embedding. In a shallow embedding, bound names are meta-variables, inaccessible on the object-level.

Substitution The use of first-order syntax entails the need for substitution, in order to (1) define α-equivalence and α-conversion and to (2) instantiate terms in a β-reduction. Focusing on the first area of application, we can use permutations to implement substitution. Table 2 gives the corresponding definitions for the π-calculus. Note that instantiations require a non-injective notion of substitution.

The approach has two elegant properties: (1) in contrast to standard substitutions, permutations are completely symmetric, and (2) permutations can deal with free and bound names alike, owing to their bijectivity. The permutation model allows us to derive the following results without any difficulty:

Lemma 1. *For a process P and names a, b, x, y such that $x, y \notin n(P)$,*

1. $P\{a\leftrightarrow a\} = P$,
2. $P\{a\leftrightarrow b\} = P\{b\leftrightarrow a\}$,
3. $(P\{x\leftrightarrow a\})\{y\leftrightarrow x\} = P\{y\leftrightarrow a\}$.

Proof: The proofs are straightforward applications of Isabelle's automatic tactics and, in item 3, structural induction. □

4 A Theory of α-Equivalence

In this section, we derive a theory of α-equivalence following the second idea of Gabbay and Pitts, which is to underspecify the requirements on names used in instantiations, simply referring to *all but finitely many* of them. This allows us to derive laws telling that it finally suffices to instantiate continuations of binders with a single fresh name. Yet, in contrast to the approach proposed in [8, 9], we do not introduce a new quantifier but specify it on two levels using universal and existential quantification. In particular, we first introduce α-equivalence

$$\frac{}{0 =_\alpha^{\mathcal{F}} 0}\text{nl} \qquad \frac{P =_\alpha^{\mathcal{F}} P'}{\tau.P =_\alpha^{\mathcal{F}} \tau.P'}\text{tau} \qquad \frac{P =_\alpha^{\mathcal{F}} P'}{\bar{a}b.P =_\alpha^{\mathcal{F}} \bar{a}b.P'}\text{out}$$

$$\frac{\forall b \notin \mathcal{F}.\ \ P\{b \leftrightarrow x\} =_\alpha^{\mathcal{F}\cup\{b\}} P'\{b \leftrightarrow x'\}}{ax.P =_\alpha^{\mathcal{F}} ax'.P'}\text{in} \qquad \frac{\forall b \notin \mathcal{F}.\ \ P\{b \leftrightarrow x\} =_\alpha^{\mathcal{F}\cup\{b\}} P'\{b \leftrightarrow x'\}}{(\nu x)P =_\alpha^{\mathcal{F}} (\nu x')P'}\text{res}$$

$$\frac{P =_\alpha^{\mathcal{F}} P' \quad Q =_\alpha^{\mathcal{F}} Q'}{P + Q =_\alpha^{\mathcal{F}} P' + Q'}\text{ch} \qquad \frac{P =_\alpha^{\mathcal{F}} P' \quad Q =_\alpha^{\mathcal{F}} Q'}{P \,|\, Q =_\alpha^{\mathcal{F}} P' \,|\, Q'}\text{par}$$

$$\frac{P =_\alpha^{\mathcal{F}} P'}{[a = b]P =_\alpha^{\mathcal{F}} [a = b]P'}\text{mt} \qquad \frac{P =_\alpha^{\mathcal{F}} P'}{[a \neq b]P =_\alpha^{\mathcal{F}} [a \neq b]P'}\text{mmt} \qquad \frac{P =_\alpha^{\mathcal{F}} P'}{!P =_\alpha^{\mathcal{F}} !P'}\text{rep}$$

Table 3. α-Equivalence wrt. $\mathcal{F}$. We define α-equivalence with respect to a set $\mathcal{F}$ of "forbidden" names, that is, names that must not be used in instantiations.

with respect to a set $\mathcal{F}$ of "forbidden" names, in the definition of which we use universal quantification over *all* names *not* in $\mathcal{F}$. Then we define a notion independent of $\mathcal{F}$ which merely requires the existence of *some finite* $\mathcal{F}$. Usually, one will choose the set of (free) names of the processes that are to be compared. Yet, the underspecification of such a set allows us to derive transitivity of α-equivalence, which is hard (or even impossible) for more specific formulations.

Implementing α-equivalence Table 3 gives an overview of the introduction rules inductively defining α-equivalence with respect to $\mathcal{F}$. Intuitively, $\mathcal{F}$ specifies the (free) names of the processes. Thus, when a fresh name is introduced by an instantiation, it has to be added to $\mathcal{F}$. An equivalence statement $ax.P =_\alpha^{\mathcal{F}} ax'.P'$, for instance, can be derived from $P\{b \leftrightarrow x\} =_\alpha^{\mathcal{F}\cup\{b\}} P'\{b \leftrightarrow x'\}$. A basic result necessary in later proofs is that $\mathcal{F}$ can be augmented arbitrarily:

Lemma 2. *If $P =_\alpha^{\mathcal{F}} P'$, then $P =_\alpha^{\mathcal{F}\cup\mathcal{F}'} P'$.*

Proof: By an easy induction on $=_\alpha^{\mathcal{F}}$, solving all cases by means of the automatic tactic `auto_tac` (see also Appendix C). □

Definition 1 (α-Equivalence). *Two processes P and P' are* α-equivalent, *written $P =_\alpha P'$, if there exists a finite $\mathcal{F}$ such that $P =_\alpha^{\mathcal{F}} P'$.*

Results The underspecification of $\mathcal{F}$ in Definition 2 allows us to derive transitivity of α-equivalence, as well as certain congruence results.

Theorem 1. *For processes P, P' and names $x, y \notin n(P) \cup n(P')$,*

1. *$=_\alpha$ is an equivalence.*
2. *$P =_\alpha P'$ implies $P\{y \leftrightarrow z\} =_\alpha P'\{y \leftrightarrow z\}$.*
3. *$P\{x \leftrightarrow b\} =_\alpha P'\{x \leftrightarrow b'\}$ implies $P\{y \leftrightarrow b\} =_\alpha P'\{y \leftrightarrow b'\}$.*

Proof: The results follow from similar results for $=_\alpha^{\mathcal{F}}$ and the fact that $P =_\alpha^{\mathcal{F}} P'$ implies $P =_\alpha^{\mathcal{F}\cup\mathcal{F}'} P'$ for arbitrary $\mathcal{F}, \mathcal{F}'$ (transitivity). The equivalence results

$$
\begin{array}{rcl}
(0)\{x \leftarrow y\} & = & 0 \\
(\tau.P)\{x \leftarrow y\} & = & \tau.P\{x \leftarrow y\} \\
(\bar{a}b.P)\{x \leftarrow y\} & = & \bar{a}b.P\{x \leftarrow y\} \\
(ab.P)\{x \leftarrow y\} & = & \text{if } b = y \text{ then } ax.P\{x \leftrightarrow y\} \text{ else } ab.P\{x \leftarrow y\} \\
((\nu b)P)\{x \leftarrow y\} & = & \text{if } b = y \text{ then } (\nu x)P\{x \leftrightarrow y\} \text{ else } (\nu b)P\{x \leftarrow y\} \\
(P + Q)\{x \leftarrow y\} & = & P\{x \leftarrow y\} + Q\{x \leftarrow y\} \\
(P \mid Q)\{x \leftarrow y\} & = & P\{x \leftarrow y\} \mid Q\{x \leftarrow y\} \\
([a = b]P)\{x \leftarrow y\} & = & [a = b]P\{x \leftarrow y\} \\
([a \neq b]P)\{x \leftarrow y\} & = & [a \neq b]P\{x \leftarrow y\} \\
(!P)\{x \leftarrow y\} & = & !P\{x \leftarrow y\}
\end{array}
$$

Table 4. α-Conversion. The function $P\{x \leftarrow y\}$ searches for the outermost bound occurrences of y replacing them by x and applying permutation to the continuation. Usually, a fresh x is chosen for that purpose.

are obtained easily using standard automatic tactics. The substitution resulults are a bit harder to prove, using lists of permutations and induction over $=^{\mathcal{F}}_{\alpha}$. □

An immediate consequence of item 3 is that we do not have to consider *all* instantiations with fresh names but only a single one in order to derive α-equivalence of processes with binders. This yields the following characterization:

Theorem 2. *For processes P, P', Q, Q' and $x \notin n(P) \cup n(P')$,*

1. $0 =_\alpha 0$,
2. $P =_\alpha P'$ *implies* $\tau.P =_\alpha \tau.P'$ *and* $\bar{a}b.P =_\alpha \bar{a}b.P'$ *and* $[a = b]P =_\alpha [a = b]P'$ *and* $[a \neq b]P =_\alpha [a \neq b]P'$ *and* $!P =_\alpha !P'$,
3. $P =_\alpha P'$ *and* $Q =_\alpha Q'$ *implies* $P + Q =_\alpha P' + Q'$ *and* $P \mid Q =_\alpha P' \mid Q'$,
4. $P\{x \leftrightarrow b\} =_\alpha P'\{x \leftrightarrow b'\}$ *implies* $ab.P =_\alpha ab'.P'$ *and* $(\nu b)P =_\alpha (\nu b')P'$.

Proof: Follows as a corollary of Theorem 1. □

A theory of α-conversion In practice, a theory of α-equivalence is often complemented by a notion of *α-conversion*. It can be specified by introducing fresh names for certain bound names, applying substitution underneath binders. Table 4 defines a primitively recursive function implementing α-conversion. For it, we can derive the standard laws characterizing α-conversion, including that arbitrary bound names can be eliminated by replacing them with fresh ones.

Theorem 3. *For every process P and names a, b, x with $x \notin n(P)$,*

1. $ax.P\{x \leftrightarrow b\} =_\alpha ab.P$, *and*
2. $(\nu x)P\{x \leftrightarrow b\} =_\alpha (\nu b)P$,
3. $b \notin n(P\{x \leftarrow b\})$,
4. $P\{x \leftarrow b\} =_\alpha P$, *and consequently,*
5. *there exists* P' *such that* $P =_\alpha P'$ *and* $b \notin n(P')$.

Proof: Items 1 and 2 can be derived from Theorem 2 by applying transitivity of permutation and reflexivity of α-equivalence. Items 3 and 4 follow by structural induction on P using items 1 and 2. Item 5 is a direct consequence. □

	first-order/deep	higher-order/deep	higher-order/shallow
adequacy	usually obvious	to be proved	to be proved
substitution	for whole language	for λ-calculus	not necessary
$\alpha\beta\eta$	substitutions	deferred	for free
bound parameters	object-variables accessible	object-variables accessible	meta-variables inaccessible
induction	yes	no	no
exotic terms	no	no	possibly
application	meta-theory justify paper proofs	meta-theory easy results about binders α-conversion	concrete examples easy results about binders
π-calculus	[1, 12, 13, 20, 28]	[10, 11]	[5, 17, 21, 28, 29]

Table 5. General classification. Combining first-order and higher-order syntax with deep and shallow embeddings. First-order syntax always yields a deep embedding.

5 Classifying Formalizations

When it comes to formalizing a language, the first question that naturally arises is which syntax to choose. In this section, we present a classification of the approaches known up to date, based on (mostly own) practical experience. We describe the main features of the approaches, point out strengths and weaknesses with respect to formalizations, and try to give a guideline for possible areas of application. Sticking to the π-calculus as an exemplaric language, we point to formalizations of it within the various schemes. In a second part of this section, we discuss in more detail first-order formalizations; for discussions about higher-order formalizations, see, for instance, [11, 17, 29].

General classification There exist two ways of expressing binders in general-purpose theorem-proving. Following a *deep* strategy, binders are formalized fully within the object-level using object-variables, for free and bound parameters alike. An alternative way is to apply a *shallow* implementation strategy, defining binders in terms of meta-level functions, thus representing bound names by meta-variables, whereas free parameters are further denoted by object-variables. These two implementation strategies can be combined with a first-order or a higher-order syntactic description of the language. Applying first-order syntax, of course, one always obtains a deep embedding, because it does not distinguish between binders and other operators on the syntactic level. Table 5 gives an overview of the three ways of formalization.

(1) The classical way is to remain fully within the object-level of the prover, giving a *first-order syntax* in a *deep embedding*. It is usually close to the way languages are treated on paper, and is therefore traditionally applied in meta-theoretical reasoning. A major inconvenience with respect to both formalization and derivations is that substitutions and α-equivalence have to be defined ex-

plicitely. In particular, reasoning about concrete processes becomes rather cumbersome. Yet, in cases where one wants to reason about bound names—in a theory of α-conversion, for example—the approach is indispensable.

(2) A second line of research applies *higher-order abstract syntax (HOAS)*, yet still within a *deep embedding*. That is, a λ-calculus is formalized along the lines of approach (1) in order to provide a functional mechanism (a pseudo-meta-level) within which binders are then expressed. As a consequence, substitutions and α-equivalence have to be defined only for the (small) underlying calculus and not for the whole (usually large) language itself. A definite advantage of the approach with respect to a shallow embedding is that it allows the user to choose an appropriate meta-logic. However, still both free and bound parameters are expressed by object-variables, which entails substitutions and makes the definition of concrete terms (programs to be analysed) cumbersome.

(3) A third line of research builds on *HOAS* in a *shallow embedding*, using the functional mechanism of the theorem-provers to represent and deal with bound names. In this case, the user does not have to bother about α-conversion and β-reduction at all, but further looses access to the bound names. It is even the case that free and bound names are not merely distinct but incomparable (on the object-level, on which proofs are conducted) with the former being object-variables and the latter meta-variables. The approach is particularly useful to concrete processes, and enjoys increasing popularity also in meta-theoretical reasoning. On the other hand, the meta-levels provided by general-purpose theorem-provers are often so powerful that exotic terms can arise, making an axiomatisation of syntactic properties extremely delicate (see [14, 16, 17, 29]).

Concerning application, deep embeddings are rather suitable in meta-theory, because they are (intuitively) close to reasoning on paper and naturally provide structural induction. Shallow embeddings traditionally head for applications concerning concrete examples, that is, processes or programs. A general problem of the HOAS approach in items (2) and (3) is that it does not easily provide structural induction, which makes syntax analysis difficult. However, with growing effort in deriving induction and syntactic proof principles, HOAS is entering meta-theory as well. Induction can be incorporated in the logical framework [16] or mimicked by rule induction over well-formedness predicates [6, 29]. For a general proof-framework, it might make sense to provide both kinds of syntaxes, with functions translating one syntax into the other and back.

First-order syntax To the best of our knowledge, four approaches have been studied in π-calculus encodings; see Table 6 for an overview. For other languages, further variations have been investigated (see, for instance, [7, 15, 30]).

(1) The most straightforward way is to formalize textual substitution in terms of simple rewrite rules, without taking care of name-capture. Such a definition can then be used to implement α-conversion, so that a "proper" notion of substitution can take care of name-capture, usually applying normalization. A theory of α-equivalence can be obtained along the same lines as done for approach (2) in

	straightforward	permutations [8, 9]	PTS [19]	deBruijn [4]
method	simple renaming, substitution on top	permutations	parameters vs./ variables, two substitutions	nameless variables
$\alpha\beta$	α-equivalence, α-conversion, β-reduction	α-equivalence, α-conversion	α-equivalence, α-conversion, β-reduction	α-equivalence is identity, β-reduction
application	adequacy, (syntax), semantics	(adequacy), syntax	syntax, semantics	semantics
π-calculus	[1, 20, 28]	this paper	[12]	[13]

Table 6. Formalizing first-order syntax. From left to right, the schemes go from "close to definitions on paper" to "more or less an implementation".

this paper. However, the proofs usually require more interaction, because of the asymmetry of the operator, with respect to free and bound parameters.

(2) Recently, Gabbay and Pitts have proposed to use the FM-set model as a basis for reasoning about languages with binders, modelling substitutions in terms of permutations. This approach is characterized by a complete symmetry both of the renaming operation and of the treatment of free and bound parameters, which relies on the bijectivity of permutations. As demonstrated in [8, 9] as well as in the preceeding sections of this paper, the permutation model allows for convenient definitions of α-conversion and α-equivalence, as well as derivations of the respective theories. A practical drawback is that the injectivity of permutations forbids a formulation of β-reduction, hence semantic analysis necessitates the introduction of a second notion of substitution along the lines of approach (1); note that for approach (3), a different datatype definition is often convenient to prevent disjoint summation, and that both approaches (3) and (4) do not need a theory of α-equivalence in the sense of (2).

(3) As a compromise between an intuitive straightforward approach and technical feasibility, McKinna and Pollack propose the use of Pure Type Systems (PTS), based on the idea of explicitely distinguishing between free and bound atoms by means of distinct sorts, which they refer to as *parameters* and *variables*. As a consequence, two notions of substitution have to be defined, a plain textual version for parameters and a capture-avoiding notion for variables. The resulting definitions of substitution are surprisingly simple, yet they do not yield the symmetry properties inherent to the permutation model.

(4) DeBruijn indices are regularly applied to implement functional mechanisms on meta-levels of theorem-provers. Here parameters are replaced by numbers, and α-equivalent terms are represented by one and the same implementation. Capture-avoiding instantiations are then expressed in terms of basic arithmetic operations that can be effectively dealt with by the programming language used for the implementation. When formalizing languages with binders larger than

the λ-calculus, the approach is hard to apply, because its technical orientation makes it intricate and prone to errors. Further, general-purpose theorem provers generally do not offer arithmetic operations as primitives, hence using DeBruijn indices never yields the degree of automation one would like to expect from it.

It seems that in first-order syntax with names, one will always have to formalize two notions of substitution when taking semantic analysis into account. In syntactic analysis, one can do with a single notion of substitution following one of the first three approaches. Following approach (1), one can require that the instantiated parameters be fresh, approach (2) offers a single notion of substitution anyway, and approach (3) only necessitates substitution on variables. Approach (4) is only a restricted basis for syntax analysis, because it identifies α-equivalent terms, and thus can only be applied in reasoning about α-equivalence classes.

6 Conclusion

In this paper, we have pursued two goals: (1) examine meta-theoretical reasoning for the π-calculus on the basis of two recent ideas of Gabbay and Pitts [8, 9], and (2) classify syntactic frameworks for it and related languages.

(1) We have formalized substitution in terms of permutations, and have instantiated continuations of binders with *all but finitely many* names in the definition of α-equivalence. An appealing property of permutations is their symmetry, $P\{a \leftrightarrow b\} = P\{b \leftrightarrow a\}$, simplifying some proofs. Further, due to their bijectivity, permutations allow for an equal treatment of free and bound names, as well as for a convenient derivation of laws necessary to study α-equivalence. This bijectivity yields, on the other hand, that they are incapable of describing β-reduction even in name-passing calculi like the π-calculus. As an example, consider the communication $\bar{a}b.P \,|\, ax.\bar{b}x.0 \xrightarrow{\tau} P \,|\, (\bar{b}x.0)\{b/x\}$. The result should reduce to $P \,|\, \bar{b}b.0$, for which a non-injective substitution is necessary, mapping both b and x to b.

Concerning α-equivalence, we have slightly modified the proposal of Gabbay and Pitts to deal with standard universal and existential quantification instead of having to introduce and develop a theory for a new quantifier. The main idea of the approach—underspecifying the names to be used in an instantiation—is retained, however, allowing for the use of single fresh names in instantiations. The formulation of α-equivalence is independent of the use of permutations, and can hence be combined with other notions of substitution alike.

(2) Languages with binders can be formalized in a deep or in a shallow embedding. While the first is traditionally considered closer to reasoning on paper, the latter is usually more amenable to concrete examples. The choice of a first-order syntax is often motivated by its natural structural induction principles. Higher-order syntax, on the other hand, is generally more convenient in reasoning about binders, and currently there is some effort to derive suitable principles for syntax analysis. Yet, the obtained results have to be adapted to the original syntax afterwards, if they are to justify proofs conducted on paper (returning from equivalence classes to plain terms).

References

1. O. Ait-Mohamed. *Pi-Calculus Theory in HOL*. PhD thesis, Henry Poincarré University, Nancy, 1996.
2. S. Berghofer and M. Wenzel. Inductive datatypes in HOL—lessons learned in Formal-Logic Engineering. In *Proc. TPHOL'99*, volume 1690 of *LNCS*, pages 19–36, 1999.
3. S. Dal Zilio and A. Gordon. Region analysis in a π-calculus with groups. In *Proc. MFCS'00*, volume 1893 of *LNCS*, pages 1–20. Springer, 2000.
4. N. deBruijn. Lambda calculus notation with nameless dummies: a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.
5. J. Despeyroux. A higher-order specification of the π-calculus. In *Proc. TCS'00*, volume 1872 of *LNCS*, pages 425–439. Springer, 2000.
6. J. Despeyroux and A. Hirschowitz. Higher-order abstract syntax with induction in Coq. In *Proc. LPAR'94*, volume 822 of *LNCS*, pages 159–173. Springer, 1994.
7. J. Ford and I. Mason. Establishing a general context lemma in PVS. In *Proc. AWCL'01*, 2001.
8. M. Gabbay. *A Theory of Inductive Definitions with α-Equivalence: Semantics, Implementation, Programming Language.* PhD thesis, Cambridge University, 2000.
9. M. Gabbay and A. Pitts. A new approach to abstract syntax involving binders. In *Proc. LICS'99*, volume 158, pages 214–224. IEEE, 1999.
10. S. Gay. A framework for the formalisation of pi-calculus type systems in Isabelle/HOL. Technical report, University of Glasgow, 2000.
11. A. Gordon and T. Melham. Five axioms of alpha-conversion. In *Proc. TPHOL'96*, volume 1125 of *LNCS*, pages 173–190. Springer, 1996.
12. L. Henry-Gréard. Proof of the subject reduction property for a pi-calculus in Coq. Technical Report RR-3698, INRIA, 1999.
13. D. Hirschkoff. A full formalisation of π-calculus theory in the calculus of constructions. In *Proc. TPHOL'97*, volume 1275 of *LNCS*, pages 153–169. Springer, 1997.
14. M. Hofmann. Semantical analysis of higher-order abstract syntax. In *Proc. LICS'99*, volume 158, pages 204–213. IEEE, 1999.
15. P. Homeier. A proof of the church-rosser theorem for the lambda calculus in higher order logic, 2001. Submitted.
16. F. Honsell, M. Miculan, and I. Scagnetto. An axiomatic approach to metareasoning on systems in higher-order abstract syntax. In *Proc. ICALP'01*, LNCS. Springer, 2001. To appear.
17. F. Honsell, M. Miculan, and I. Scagnetto. π-calculus in (co)inductive type theory. *Theoretical Computer Science*, 253(2):239–285, 2001.
18. C. Jones. A π-calculus semantics for an object-based design notation. In *Proc. CONCUR'93*, volume 715 of *LNCS*, pages 158–172. Springer, 1993.
19. J. McKinna and R. Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3–4):373–409, 1999.
20. T. Melham. A mechanized theory of the π-calculus in HOL. *Nordic Journal of Computing*, 1(1):50–76, 1995.
21. D. Miller. The π-calculus as a theory in linear logic: Preliminary results. In *Proc. ELP'92*, volume 660 of *LNCS*, pages 242–264. Springer, 1992.
22. R. Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 17:119–141, 1992.

23. R. Milner. *Communicating and Mobile Systems: the π-calculus.* Cambridge University Press, 1999.
24. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.
25. L. Paulson. Isabelle's object-logics. Technical Report 286, University of Cambridge, Computer Laboratory, 1993.
26. L. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In *Proc. CADE'94*, volume 814 of *LNAI*, pages 148–161. Springer, 1994.
27. L. Paulson, editor. *Isabelle: a generic theorem prover*, volume 828 of *LNCS*. Springer, 1994.
28. C. Röckl. *On the Mechanized Validation of Infinite-State and Parameterized Reactive and Mobile Systems.* PhD thesis, Technische Universität München, 2001.
29. C. Röckl, D. Hirschkoff, and S. Berghofer. Higher-order abstract syntax with induction in Isabelle/HOL: Formalizing the π-calculus and mechanizing the theory of contexts. In *Proc. FOSSACS'01*, volume 2030 of *LNCS*, pages 364–378. Springer, 2001.
30. R. Vestergaard and J. Brotherston. Formalised first-order confluence proof for the λ-calculus using one-sorted variable names. In *Proc. RTA'01*, volume 2051 of *LNCS*, pages 306–321. Springer, 2001.
31. D. Walker. Objects in the π-calculus. *Information and Computation*, 116(2):253–271, 1995.

A The π-Calculus

Infinity of the set of names is expressed by an axiomatic type-class `inf_class` requiring the existence of an injection from the natural numbers to its members.

```
axclass inf_class < term
  inf_class "EX (f::nat=>'a). inj f"
```

In the datatype for processes, a reference to `inf_class` ensures that names always belong to it. The right-hand annotations specify a concrete syntax.

```
datatype
  'a procs = Null                        (".0" 115)
           | Tau     "('a::inf_class) procs" (".t.(_)" [111] 110)
           | Out     'a 'a ('a procs)    ("_<_>._" [120, 0, 110] 110)
           | In      'a 'a ('a procs)    ("_[_]._" [120, 0, 110] 110)
           | Res     'a ('a procs)       (".#_ _"  [180, 101] 100)
           | Plus    ('a procs) ('a procs) (infixl ".+" 85)
           | Par     ('a procs) ('a procs) (infixl ".|" 90)
           | Mt      'a 'a ('a procs)    (".[_.=_]_" [100, 100, 96] 95)
           | Mmt     'a 'a ('a procs)    (".[_.~=_]_" [100, 100, 96] 95)
           | Repl    ('a procs)          (".!_" [100] 100)
```

B Permutations

Permutation on names is completely symmetric, which carries over to processes.

```
constdefs
  nren :: "[('a::inf_class), 'a, 'a] => 'a" ("n{_<->_}_" [0,0,199] 200)
 "n{x<->y}a == if a=x then y else if a=y then x else a"

consts
  pren :: "[('a::inf_class), 'a, 'a procs] => 'a procs"
          ("p{_<->_}_" [0,0,114] 115)

primrec
  "p{x<->y}.0 = .0"
  "p{x<->y}(.t.P) = .t.p{x<->y}P"
  "p{x<->y}(a<b>.P) = n{x<->y}a<n{x<->y}b>.p{x<->y}P"
  "p{x<->y}(a[b].P) = n{x<->y}a[n{x<->y}b].p{x<->y}P"
  "p{x<->y}(.#b P) = .#n{x<->y}b p{x<->y}P"
  "p{x<->y}(P .+ Q) = p{x<->y}P .+ p{x<->y}Q"
  "p{x<->y}(P .| Q) = p{x<->y}P .| p{x<->y}Q"
  "p{x<->y}(.[a.=b]P) = .[n{x<->y}a.=n{x<->y}b]p{x<->y}P"
  "p{x<->y}(.[a.~=b]P) = .[n{x<->y}a.~=n{x<->y}b]p{x<->y}P"
  "p{x<->y}(.!P) = .!p{x<->y}P"
```

For sample proofs, consider the following, using induction resolved by Isabelle's automatic tactic `auto_tac`:

```
Goal "p{x<->x}P = P";
by (induct_tac "P" 1);
by (Auto_tac);
qed "pren_id";

Goal "{y, z} Int n P = {} --> p{y<->z}p{z<->x}P = p{y<->x}P";
by (induct_tac "P" 1);
by (auto_tac (claset() addIs [nren_trans], simpset() delsimps [nren_def]));
qed "lemma";

Goal "[| y ~: n P ; z ~: n P |] ==> p{y<->z}p{z<->x}P = p{y<->x}P";
by (fast_tac (claset() addIs [lemma RS mp]) 1);
qed "pren_trans";
```

C α-Equivalence

The two levels of α-equivalence, `Al` with respect to `F` and `Alpha` are defined in terms of an inductive set and a constant definition, respectively.

```
consts
  Al    :: "('a procs * ('a::inf_class) set * 'a procs) set"
  Alpha :: "(('a::inf_class) procs * 'a procs) set"

syntax
  "@Al"    :: "['a procs, 'a set, 'a procs] => bool"
               ("_ =a[_] _" [70, 0, 71] 70)
```

```
  "@Alpha" :: "['a procs, 'a procs] => bool"
               ("_ =a _" [70, 71] 70)

translations
  "P =a[F] P'" == "(P, F, P') : Al"
  "P =a P'" == "(P, P') : Alpha"

inductive "Al"
  intrs
    Null  ".0 =a[F] .0"
    Tau   "P =a[F] P' ==> .t.P =a[F] .t.P'"
    Out   "P =a[F] P' ==> a<b>.P =a[F] a<b>.P'"
    In    "ALL x. x ~: S --> p{x<->b}P =a[insert x F] p{x<->b'}P' \
           \ ==> a[b].P =a[F] a[b'].P'"
    Res   "ALL x. x ~: S --> p{x<->b}P =a[insert x F] p{x<->b'}P' \
           \ ==> .#b P =a[F] .#b' P'"
    Plus  "[| P =a[F] P' ; Q =a[F] Q' |] ==> P .+ Q =a[F] P' .+ Q'"
    Par   "[| P =a[F] P' ; Q =a[F] Q' |] ==> P .| Q =a[F] P' .| Q'"
    Mt    "P =a[F] P' ==> .[a.=b]P =a[F] .[a.=b]P'"
    Mmt   "P =a[F] P' ==> .[a.~=b]P =a[F] .[a.~=b]P'"
    Repl  "P =a[F] P' ==> .!P =a[F] .!P'"

defs
  Alpha_def  "Alpha == {(P, P') . EX S. finite S & P =a[F] P'}"
```

For sample proofs, consider the following, using rule induction solved by Isabelle's automatic tactics `auto_tac` and `force_tac`, or using further interaction:

```
Goal "P =a[F] P' ==> P =a[F Un F'] P'";
by (etac Al.induct 1);
by (auto_tac (claset() addSIs Al.intrs, simpset()));
qed "Al_un1";

Goal "P =a[F] P' ==> ALL Q Q' xs xs' . \
 \ P = p{xs}Q & P' = p{xs'}Q' & \
 \ y ~: insert z ((dom xs) Un (dom xs') Un n Q Un n Q') --> \
 \ p{y<->z}P =a[{y, z} Un F] p{y<->z}P'";
by (etac Al.induct 1);
...   (* long chain of interactions *)
by (REPEAT (force_tac
    (claset() addSDs psubst_cases addSIs Al.intrs, simpset()) 1));
qed "lemma";

Goal "[| P =a[F] P' ; y ~: (n P Un n P') |] \
 \ ==> p{y<->z}P =a[{y, z} Un F] p{y<->z}P'";
by (case_tac "y=z" 1);
by (force_tac (claset() addIs [Al_insert], simpset()) 1);
...   (* instantiations *)
by (Force_tac 1);
qed "Al_pren_cong1";
```