Towards a Formalization of pi-calculus Processes in Higher Order Abstract Syntax

Christine Roeckl, Daniel Hirschkoff, Stefan Berghofer

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON
n° 5668

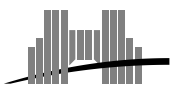
CNRS CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

Towards a Formalization of π-calculus Processes in Higher Order Abstract Syntax

Christine Röckl
Daniel Hirschkoff
Stefan Berghofer

June 2000

Research Report N° 2000-23



École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`

Towards a Formalization of π-calculus Processes in Higher Order Abstract Syntax

Christine Röckl
Daniel Hirschkoff
Stefan Berghofer

June 2000

Abstract

Higher order abstract syntax is a natural way to formalize programming languages with binders, like the π-calculus, because α-conversion and β-reduction are delegated to the meta level of the provers, making tedious substitutions superfluous. However, such formalizations usually lack induction principles, and often give rise to exotic terms. Induction is necessary in syntax analysis, and certain important syntactic properties might be invalid in the presence of exotic terms. The paper introduces *well formedness* predicates for the π-calculus with which exotic terms are excluded and, simultaneously, induction principles are obtained. The principles are then used in formal proofs of vital syntactic properties, mechanized in Isabelle/HOL.

Keywords: mobility, general-purpose theorem proving, higher order abstract syntax, induction, π-calculus

Résumé

La syntaxe abstraite d'ordre supérieur est une technique pour la formalisation de languages comportant des constructions liantes tels que le π-calcul. Grâce à cette technique, l'utilisateur n'a pas à gérer explicitement une notion de substitution, l'α-conversion et la β-réduction faisant intervenir les variables du niveau meta. Cependant, dans une telle approche, l'on ne dispose pas de principe d'induction de manière naturelle, et de plus le langage tel qu'il est formalisé peut englober des termes considérés comme exotiques. Dans cet travail, nous définissons des prédicats de bonne formation pour le π-calcul permettant d'éliminer les termes exotiques et fournissant des principes d'induction. Ceci rend possible la preuve de propriétés syntaxiques essentielles pour le π-calcul, que nous formalisons dans le système Isabelle/HOL.

Mots-clés: mobilité, assistants à la preuve, syntaxe abstraite d'ordre supérieur, induction, π-calcul

1 Motivation

The π-calculus was introduced to model and analyse mobile systems [17, 16]. It is based on synchronous communications, in which a sender $\bar{\mathbf{a}}\mathbf{b}.P$ transmits a message $\mathbf{b}$ to a recipient $\mathbf{a}\mathbf{x}.Q$, yielding a transition

$$\bar{\mathbf{a}}\mathbf{b}.P \mid \mathbf{a}\mathbf{x}.Q \xrightarrow{\tau} P \mid Q\{\mathbf{b}/\mathbf{x}\}$$

Traditionally, the resulting β-reduction is described by a substitution. This can be a tedious task with processes containing binders, like $Q = (\nu\mathbf{b})Q'$, where further substitutions are necessary for resulting α-conversions, $Q\{\mathbf{b}/\mathbf{x}\} =_\alpha (\nu\mathbf{b}')$ $Q'\{\mathbf{b}'/\mathbf{b}, \mathbf{b}/\mathbf{x}\}$. Communication channels and messages both belong to the same type, called *names*. This simplicity gives the π-calculus the power to encode the λ-calculus [15], as well as higher order object oriented and imperative languages [25, 24]. Proofs in the π-calculus, and in particular bisimulation proofs, tend to be very large and tedious, hence machine assistance is necessary to prevent errors. The work at hand is part of a larger project to provide a platform for machine assisted reasoning in and about the π-calculus. We have chosen Isabelle/HOL [21, 19], as it is generic and offers a large range of powerful proof techniques.

General-purpose theorem provers distinguish two levels of reasoning. Upon a *meta logic* that has been provided by the implementors, users can create *object logics* in which they define new data structures and derive proofs. Programming languages or calculi can be formalized, either fully within the object level using a first order syntax, or by exploiting the functional, that is, higher order, mechanisms of the meta level. In a *first order*, or *deep*, embedding, the syntax of the π-calculus is described in terms a recursive datatype of the form $P ::= 0 \mid \bar{\mathbf{a}}\mathbf{b}.P \mid \mathbf{a}\mathbf{b}.P \mid \ldots$. As a consequence, the user can make full use of induction principles, yet has to introduce substitution functions to implement α-conversion and β-reduction on the object level. Several first order formalizations of the π-calculus have been studied in various theorem provers [13, 1, 9, 12, 8]. They have given evidence that proofs about π-calculus processes in first order embeddings are very hard, and that it would be illusive to try to tackle larger proofs. This is due to the calculus being particularly characterized by its binders, *input* and *restriction*, hence a lot of effort has to be invested in tedious substitutions. In a *higher order*, or *shallow*, embedding, on the other hand, the syntax of the π-calculus is described in terms of a recursive datatype of the form $P ::= 0 \mid \bar{\mathbf{a}}\mathbf{b}.P \mid \mathbf{a}x.f_P(x) \mid \ldots$. Here, β-reduction boils down to function application on the meta level, and α-conversion is dealt with by the meta logic as well, freeing the user from a tedious implementation and application of substitution functions. Several higher order formalizations of the π-calculus have been studied in various theorem provers [14, 11, 3]. Unfortunately, with the above datatype not being recursive in a strict sense, there are no suitable induction principles, hence, syntax analysis becomes impossible. Even worse, encodings in higher order abstract syntax may fail to capture precisely the class of processes one is interested in, giving rise to so called *exotic* terms. Consider, for instance,

$$\begin{aligned} f_E &\stackrel{def}{=} \lambda(x : names).\ \text{if } x = \mathbf{a} \text{ then } 0 \text{ else } \mathbf{a}y.0 \\ f_W &\stackrel{def}{=} \lambda(x : names).\ \mathbf{a}y.0 \end{aligned}$$

where f_E is an exotic term, and f_W can be considered as *valid*, or *well formed*.

Three syntactic properties of the π-calculus are essential for a formalization of its labelled transition semantics and bisimulations. This has been pointed out with regard to strong semantics by Honsell, Miculan, and Scagnetto, in [11]. Our own results give evidence that these principles also suffice for the analysis of weak semantics. One of the properties, which Honsell et al. call *extensionality of contexts*, deserves further mention, as it does not hold in the presence of exotic terms: *Two process abstractions are equal, if they are equal for a fresh name.* Consider f_E and f_W from above, and some $\mathbf{b} \neq \mathbf{a}$. Then $f_E(\mathbf{b}) = \mathbf{a}y.0 = f_W(\mathbf{b})$, because the conditional in f_E evaluates to the negative argument. Yet, still $f_E \neq f_W$, because $f_E(\mathbf{a}) \neq f_W(\mathbf{a})$. See also [10] for a discussion.

In this paper, we discuss how exotic process terms can be ruled out and, simultaneously, induction principles can be obtained, by introducing *well formedness* predicates on higher order process terms, making it possible for us to give formal proofs of syntactic principles of the π-calculus. Our approach has been inspired by the work of Despeyroux, Felty and Hirschowitz on higher order embeddings of the λ-calculus [5, 4]. Alternative methodologies for obtaining induction principles in higher order abstract syntax are described in [7, 6]. Although the use of well formedness predicates seems to be a natural choice, it was not obvious at the beginning whether they would suffice to justify the desired syntactic properties. The proofs, which have been fully formalized in Isabelle/HOL[1], apply induction techniques, partly employing coercion from higher order to first order syntax and back within a single proof step. To our knowledge, it is the first time that these non-trivial properties have been derived in a theorem prover.

The paper is organized as follows: In Section 2, we present the necessary background of Isabelle/HOL. In Section 3, we introduce the π-calculus, and describe how it is formalized in our framework. In Section 4, we derive three vital syntactic properties of the π-calculus mentioned above. In Section 5, we discuss some questions related to our results.

2 Isabelle/HOL

We use the theorem prover Isabelle [21], implementing higher order intuitionistic logic on its meta level, and formalize the π-calculus in its instantiation HOL for higher order logic [19]. 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. These heuristics have in common that a provable goal is always transformed into a set of provable

[1] The sources are available at `http://www7.in.tum.de/~roeckl/PI/syntax.shtml`.

subgoals; rules that might yield unprovable subgoals are only applied if none of the resulting subgoals has to be reported to the user as currently unproved.

A major characteristic of Isabelle is that it is generic. This means that new objects must be defined in terms of already existing concepts. Properties of the new objects can then be derived from their definitions by formal proofs. In Isabelle/HOL, the user can define, for instance, recursive datatypes and inductive sets. Isabelle then automatically computes rules for induction and case injection. It should be noted that all these techniques have been fully formalized and verified on the object level, that is, they are a conservative generic extension of Isabelle/HOL [20, 2]. A recent extension of Isabelle/HOL allows function types in datatype definitions to contain strictly positive occurrences of the type being defined [2]. This allows for formalizations of programming languages in higher order abstact syntax, like the one we develop in Section 3 of this paper. Isabelle/HOL implements an extensional equality, $=$, which relates functions if they are equal for all arguments. We employ this equivalence as syntactic equivalence of π-calculus processes.

3 Formalizing Processes

The π-calculus is a value-passing calculus that has been introduced to reason about mobile systems [17, 16]. In the π-calculus, *names* are used both for the communication channels and the values sent along them, allowing processes to emit previously private names, so to create new communication links with the recipients. The π-calculus is particularly characterized by its binding operators *input*, $\mathbf{a}y.P$, and *restriction*, $(\nu x)P$. The former implements the functional aspects of the calculus — apply a process abstraction to a received *name* — whereas the latter characterizes its imperative aspects — create a fresh location, that is, a fresh name. In this section, we describe how the π-calculus can be formalized in higher order abstract syntax, and present well formedness predicates simultaneously ruling out exotic terms and introducing induction principles. We use the datatype from [11, 3] so that our results are comparable to these formalizations.

Names In the semantic analysis of processes one often instantiates process terms with *fresh* names, hence, the type of names has to be at least countably infinite. We do not use a specific type but employ an axiomatic type class *inf_class* comprising all types $\mathcal{T}$ for which there exists an injection from $I\!N$ into $\mathcal{T}$. We neither require nor forbid the existence of a surjection, because for our purpose it is simply relevant that there are infinitely many names, see also our discussion in Section 5. We use $\mathbf{a}, \mathbf{b}, \ldots$ to range over names, and f_a and ff_a to denote *names abstractions*, that is, functions mapping one, respectively two names, to names. In order to make names and meta variables distinguishable, we use bold face letters for the former, as above, and italics for the latter, that is, $x, y, \ldots$.

Processes Processes in the π-calculus are built from *inaction* and the basic mechanisms for the exchange and creation of names, *input*, *output*, and *restriction*, by applying constructors for *choice* (or, *summation*), *parallel composition*, *matching*, *mismatching*, and *replication*. Applying higher order abstract syntax,

$$
\begin{array}{rcl}
fn(0) & = & \emptyset \\
fn(\tau.P) & = & fn(P) \\
fn(\bar{\mathbf{a}}\mathbf{b}.P) & = & \{\mathbf{a},\mathbf{b}\} \cup fn(P) \\
fn(\mathbf{a}x.f_P(x)) & = & \{\mathbf{a}\} \cup fna(f_P) \\
fn((\nu\, x)f_P(x)) & = & fna(f_P) \\
fn(P+Q) & = & fn(P) \cup fn(Q) \\
fn(P \parallel Q) & = & fn(P) \cup fn(Q) \\
fn([\mathbf{a}=\mathbf{b}]P) & = & \{\mathbf{a},\mathbf{b}\} \cup fn(P) \\
fn([\mathbf{a}\neq\mathbf{b}]P) & = & \{\mathbf{a},\mathbf{b}\} \cup fn(P) \\
fn(!P) & = & fn(P)
\end{array}
$$

$$
\begin{array}{rcl}
fna(f_P) & \stackrel{def}{=} & \{\, \mathbf{a} \mid \forall \mathbf{b}.\ \mathbf{a} \in fn(f_P(\mathbf{b}))\,\} \\
fnaa(ff_P) & \stackrel{def}{=} & \{\, \mathbf{a} \mid \forall \mathbf{b}.\ \mathbf{a} \in fna(\lambda x.\ ff_P(\mathbf{b},x))\,\}
\end{array}
$$

$$
\begin{array}{rcl}
db(0,\mathbf{c}) & = & 0 \\
db(\tau.P,\mathbf{c}) & = & db(P,\mathbf{c}) \\
db(\bar{\mathbf{a}}\mathbf{b}.P,\mathbf{c}) & = & db(P,\mathbf{c}) \\
db(\mathbf{a}x.f_P(x),\mathbf{c}) & = & 1 + dba(f_P,\mathbf{c}) \\
db((\nu\, x)f_P(x),\mathbf{c}) & = & 1 + dba(f_P,\mathbf{c}) \\
db(P+Q,\mathbf{c}) & = & max(db(P,\mathbf{c}), db(Q,\mathbf{c})) \\
db(P \parallel Q,\mathbf{c}) & = & max(db(P,\mathbf{c}), db(Q,\mathbf{c})) \\
db([\mathbf{a}=\mathbf{b}]P,\mathbf{c}) & = & db(P,\mathbf{c}) \\
db([\mathbf{a}\neq\mathbf{b}]P,\mathbf{c}) & = & db(P,\mathbf{c}) \\
db(!P,\mathbf{c}) & = & db(P,\mathbf{c})
\end{array}
$$

$$
dba(f_P,\mathbf{c}) \stackrel{def}{=} db(f_P(\mathbf{c}),\mathbf{c})
$$

Table 1: Computing the free names and depth of binders of a process.

we formalize input and restriction by means of *process abstractions* f_P, that is, functions from names to processes. This can be implemented directly in Isabelle/HOL, because in the type of the declaration, processes only occur in a

$$\frac{}{\mathbf{wfp}\,(0)}\ \mathbf{W}_0 \qquad \frac{\mathbf{wfp}\,(P)}{\mathbf{wfp}\,(\tau.P)}\ \mathbf{W}_1 \qquad \frac{\mathbf{wfp}\,(P)}{\mathbf{wfp}\,(\bar{\mathbf{a}}\mathbf{b}.P)}\ \mathbf{W}_2$$

$$\frac{\mathbf{wfpa}\,(f_P)}{\mathbf{wfp}\,(\mathbf{a}y.f_P(y))}\ \mathbf{W}_3 \qquad \frac{\mathbf{wfpa}\,(f_P)}{\mathbf{wfp}\,((\nu\, y)f_P(y))}\ \mathbf{W}_4 \qquad \frac{\mathbf{wfp}\,(P) \quad \mathbf{wfp}\,(Q)}{\mathbf{wfp}\,(P+Q)}\ \mathbf{W}_5$$

$$\frac{\mathbf{wfp}\,(P) \quad \mathbf{wfp}\,(Q)}{\mathbf{wfp}\,(P \parallel Q)}\ \mathbf{W}_6 \qquad \frac{\mathbf{wfp}\,(P)}{\mathbf{wfp}\,([\mathbf{a}=\mathbf{b}]P)}\ \mathbf{W}_7$$

$$\frac{\mathbf{wfp}\,(P)}{\mathbf{wfp}\,([\mathbf{a}\neq\mathbf{b}]P)}\ \mathbf{W}_8 \qquad \frac{\mathbf{wfp}\,(P)}{\mathbf{wfp}\,(!P)}\ \mathbf{W}_9$$

Table 2: Well formed Processes.

positive position.

P	::=	0	*Inaction*
		$\tau.P$	*Silent Prefix*
		$\bar{\mathbf{a}}\mathbf{b}.P$	*Output Prefix*
		$\mathbf{a}x.f_P(x)$	*Input Prefix*
		$(\nu\, x)f_P(x)$	*Restriction*
		$P+P$	*Choice (Summation)*
		$P \parallel P$	*Parallel Composition*
		$[\mathbf{a}=\mathbf{b}]P$	*Matching*
		$[\mathbf{a}\neq\mathbf{b}]P$	*Mismatching*
		$!P$	*Replication*

The above datatype exactly corresponds to those formalized in Coq in [11, 3]. It is obvious that this datatype definition is not recursive in a strict sense, due to the use of process abstractions f_P as continuations of input and restriction. Therefore, induction and case injection are not applicable. Further, it is possible to derive exotic terms in Isabelle/HOL, like f_E from the motivation. We use $P, Q, \ldots$ to range over processes, and f_P and ff_P for process abstractions.

Free and Fresh Names Names occurring in a process which are not in the scope of a binder are called *free*, names in the scope of a binder are called *bound*. In higher order abstract syntax, it is neither necessary nor possible to compute the bound names of a process, because they are represented by meta level variables of the theorem prover. To compute the free names of a process, which are represented by object level variables, we use a primitively recursive function *fn*, see Table 1. Note that for exotic process terms like f_E from Section 1, *fn* and *fna* need not necessarily compute all fresh names; for f_E, for instance, *fna* computes the empty set. However, for all *well formed* processes, *fn* and *fna* yield the expected results. A name is *fresh* in a process or process abstraction if it is not among its free names. This can be formalized in terms of $\mathit{fresh}(a, P)$ iff $a \notin \mathit{fn}(P)$, and $\mathit{fresha}(a, f_P)$ iff $a \notin \mathit{fna}(f_P)$ and $\mathit{freshaa}(a, ff_P)$ iff $a \notin \mathit{fnaa}(ff_P)$, respectively.

Well formedness We introduce *well formedness* predicates in the spirit of [5, 4], with which we simultaneously eliminate exotic processes like f_E given in the motivation, and obtain induction principles which allow us to derive formally

$$\frac{}{\mathbf{wfpa}\,(\lambda x.\,0)}\ \mathbf{W}_0^a \qquad \frac{\mathbf{wfpa}\,(f_P)}{\mathbf{wfpa}\,(\lambda x.\,\tau.f_P(x))}\ \mathbf{W}_1^a \qquad \frac{\mathbf{wfna}\,(f_a) \quad \mathbf{wfna}\,(f_b) \quad \mathbf{wfpa}\,(f_P)}{\mathbf{wfpa}\,(\lambda x.\,\overline{f_a(x)}f_b(x).f_P(x))}\ \mathbf{W}_2^a$$

$$\frac{\mathbf{wfna}\,(f_a) \quad \forall\, b.\ \mathbf{wfpa}\,(\lambda x.\,ff_P(b,x)) \quad \forall\, b.\ \mathbf{wfpa}\,(\lambda x.\,ff_P(x,b))}{\mathbf{wfpa}\,(\lambda x.\,f_a(x)y.ff_P(y,x))}\ \mathbf{W}_3^a$$

$$\frac{\forall\, b.\ \mathbf{wfpa}\,(\lambda x.\,ff_P(b,x)) \quad \forall\, b.\ \mathbf{wfpa}\,(\lambda x.\,ff_P(x,b))}{\mathbf{wfpa}\,(\lambda x.\,(\nu\, y)ff_P(y,x))}\ \mathbf{W}_4^a$$

$$\frac{\mathbf{wfpa}\,(f_P) \quad \mathbf{wfpa}\,(f_Q)}{\mathbf{wfpa}\,(\lambda x.\,f_P(x)+f_Q(x))}\ \mathbf{W}_5^a \qquad \frac{\mathbf{wfpa}\,(f_P) \quad \mathbf{wfpa}\,(f_Q)}{\mathbf{wfpa}\,(\lambda x.\,f_P(x)\parallel f_Q(x))}\ \mathbf{W}_6^a$$

$$\frac{\mathbf{wfna}\,(f_a) \quad \mathbf{wfna}\,(f_b) \quad \mathbf{wfpa}\,(f_P)}{\mathbf{wfpa}\,(\lambda x.\,[f_a(x)=f_b(x)].f_P(x))}\ \mathbf{W}_7^a \qquad \frac{\mathbf{wfna}\,(f_a) \quad \mathbf{wfna}\,(f_b) \quad \mathbf{wfpa}\,(f_P)}{\mathbf{wfpa}\,(\lambda x.\,[f_a(x)\neq f_b(x)].f_P(x))}\ \mathbf{W}_8^a$$

$$\frac{\mathbf{wfpa}\,(f_P)}{\mathbf{wfpa}\,(\lambda x.\,!f_P(x))}\ \mathbf{W}_9^a$$

Table 3: Well formed Process Abstractions.

$$\frac{}{\mathbf{wfna}\,(\lambda x.\,x)}\ \mathbf{W}_1^n \qquad \frac{}{\mathbf{wfna}\,(\lambda x.\,\mathbf{a})}\ \mathbf{W}_2^n$$

$$\frac{}{\mathbf{wfnaa}\,(\lambda(x,y).\,x)}\ \mathbf{W}_3^n \qquad \frac{}{\mathbf{wfnaa}\,(\lambda(x,y).\,y)}\ \mathbf{W}_4^n \qquad \frac{}{\mathbf{wfnaa}\,(\lambda(x,y).\,\mathbf{a})}\ \mathbf{W}_5^n$$

Table 4: Well formed Names Abstractions.

syntactic properties of the π-calculus in Section 4. The predicates are defined inductively, and concern three levels of reasoning: **wfp** defines the set of well formed processes, see Table 2 for the introduction rules, **wfpa** yields the set of well formed process abstractions, see Table 3, and **wfna** and **wfnaa** describe the well formed names abstractions, see Table 4. The rules concerning the binders, that is, W_3, W_4, W_3^a, and W_4^a, are of particular interest. For a restricted or input process to be well formed according to **wfp**, the continuation f_P has to be well formed according to **wfpa**. With f_P possibly containing inputs and/or restrictions itself, this argument could have to be continued ad infinitum. As pointed out in [4], a second order predicates suffices to rule out at least those exotic terms that might render syntactic properties of the original language incorrect in the encoding. Therefore, we argue in W_4^a that a process abstraction over two names can be applied safely if it is well formed according to **wfna** in both its arguments. The process abstraction f_E from the introduction, for instance, is ruled out by this definition. Clearly, for every π-calculus process there is a well formed encoding of it. On the other hand, we will see in Section 4 that our definition of well formedness is discriminating enough to prove all necessary syntactic properties of the π-calculus.

Counting Binders In the proof in Section 4.4, we apply a coercion from higher order syntax to first order syntax by instantiating meta level variables in the scope of the binding operators, with fresh names. In order to be able to provide an amount of fresh names which is sufficient for the proof, we compute the *depth of binders* with a primitively recursive function, *db*, examining process trees in order to determine the maximal number of binders along each path.

$$\frac{\mathit{fresh}(\mathbf{a},\ f_P(\mathbf{b}))}{\mathit{fresha}(\mathbf{a},\ f_P)}\ (\textsc{Mon}) \qquad \frac{\mathit{fresha}(\mathbf{a},\ \lambda x.\, ff_P(\mathbf{b}, x))}{\mathit{freshaa}(\mathbf{a},\ ff_P)}\ (\textsc{Mona})$$

$$\frac{\begin{array}{cc}\mathbf{wfpa}\,(f_P) & \mathbf{wfpa}\,(f_Q) \\ \mathit{fresha}(\mathbf{a}, f_P) & \mathit{fresha}(\mathbf{a}, f_Q)\end{array} \qquad f_P(\mathbf{a}) = f_Q(\mathbf{a})}{f_P = f_Q}\ (\textsc{Ext})$$

$$\frac{\mathbf{wfp}\,(P)}{\exists\, f_P.\ \mathbf{wfpa}\,(f_P)\ \wedge\ \mathit{fresha}(\mathbf{a}, f_P)\ \wedge\ P = f_P(\mathbf{a})}\ (\textsc{Exp})$$

Table 5: Formalizations of *monotonicity*, *extensionality*, and β-expansion.

The definition of *db* is given in Table 1, where $\mathbf{c}$ is an arbitrary name used to instantiate process abstractions. Like *fn*, the function *db* only yields sensible results for well formed processes.

4 Deriving Syntactic Properties

Three syntactic properties of the π-calculus are necessary in bisimulation proofs and for the derivation of weak transition laws. As they all deal with process abstractions, or, process contexts, Honsell, Miculan, and Scagnetto refer to them as the *theory of contexts* [11]. Informally, they can be described as follows:

(Mon) Monotonicity: *If a name* $\mathbf{a}$ *is fresh in an instantiated process abstraction* $f_P(\mathbf{b})$*, it is fresh in* f_P *already.*

(Ext) Extensionality: *Two process abstractions* f_P *and* f_Q *are equal, if they are equal for a fresh name* $\mathbf{a}$*.*

(Exp) β-Expansion: *Every process* P *can be abstracted over an arbitrary name* $\mathbf{a}$*, yielding a suitable process abstraction.*

We prove their formal counterparts, as presented in Table 5, for well formed processes and process abstractions. Recall from the motivation that extensionality only holds for well formed processes. Also in the third law, describing β-expansion, that is, the reverse of β-reduction, we only consider well formed processes and process abstractions, so to make the law strong enough for the semantic analysis of well formed processes. To the best of our knowledge, it is the first time that the three principles have been formally justified in interaction with a theorem prover. The proofs have been conducted in the latest version of Isabelle/HOL[2]. Honsell et al. encode them as axioms in their formalization, reasoning informally for their correctness [11]. Recently, Hofmann has presented another informal justification, using category theory [10].

4.1 Free and Fresh Names

In the proofs of (Ext)and (Exp), we employ the fact that there exist at least countably infinitely many names, see Section 3, so we can always find a fresh

[2]The proof scripts are available at `http://www7.in.tum.de/~roeckl/PI/syntax.shtml`.

name with which to instantiate a process abstraction. The laws (f1) – (f7) formalize these basic properties; their proofs in Isabelle/HOL are standard, and yield scripts of a few lines only.

$$\text{(f1)}\quad \exists\, \mathbf{b}.\ \mathbf{a} \neq \mathbf{b} \qquad \text{(f2)}\quad \frac{\mathit{finite}(A)}{\exists\, \mathbf{b}.\ \mathbf{b} \notin A}$$

$$\text{(f3)}\quad \mathit{finite}(\mathit{fn}(P)) \qquad \text{(f4)}\quad \mathit{finite}(\mathit{fna}(f_P)) \qquad \text{(f5)}\quad \mathit{finite}(\mathit{fnaa}(\mathit{ff}_P))$$

$$\text{(f6)}\quad \frac{\mathbf{wfpa}(f_P) \qquad \mathit{fresha}(\mathbf{a}, f_P) \qquad \mathbf{c} \neq \mathbf{a}}{\mathit{fresh}(\mathbf{a}, f_P(\mathbf{c}))}$$

$$\text{(f7)}\quad \frac{\begin{array}{l}\forall \mathbf{b}.\ \mathbf{wfpa}(\lambda x.\ \mathit{ff}_P(\mathbf{b}, x)) \\ \forall \mathbf{b}.\ \mathbf{wfpa}(\lambda x.\ \mathit{ff}_P(x, \mathbf{b}))\end{array} \qquad \mathit{freshaa}(\mathbf{a}, \mathit{ff}_P) \qquad \mathbf{c} \neq \mathbf{a}}{\mathit{fresha}(\mathbf{a}, \lambda x.\ \mathit{ff}_P(\mathbf{c}, x))}$$

Laws (f6) and (f7) express that a name $\mathbf{a}$ which is fresh for a well formed process abstraction, is necessarily fresh for every instantiation except $\mathbf{a}$. (f6) is proved by induction over **wfpa**, and all cases are proved automatically by Isabelle; (f7) can then be derived as a corollary, by a single call to an automatic tactic.

4.2 Monotonicity

The monotonicity law, see (Mon) in Table 5, is implicitely encoded in our formalization. That is, a name $\mathbf{a}$ is only free in a process abstraction f_P according to *fnaa* if it is free in every instantiation, hence for $\mathbf{a}$ to be fresh in f_P it suffices to present a single name $\mathbf{b}$ so that $\mathbf{a}$ is fresh for $f_P(\mathbf{b})$. The proof in Isabelle requires one call to a standard automatic tactic. Monotonicity can be derived similarly for *freshaa*, see (Mona) in Table 5.

4.3 Extensionality

Two process abstractions should be equal if they are equal for a single fresh name. This variation of extensionality, where usually a universal quantification is used, is natural in the absence of exotic terms, yet does not hold in their presence, see Section 1 for an example. In the formalization of the π-calculus in Coq presented in [11], the Calculus of Constructions guarantees that exotic terms cannot be derived as long as no functions and relations on names can be defined. This allows Honsell et al. to add (Ext) as an axiom, without explicitely requiring the abstractions to be well formed.

We prove (Ext) by induction over one of the two involved well formed processes, f_P, and using case injection for the other, that is, f_Q. Eight out of the ten cases resulting from the induction are purely technical. The interesting cases are those concerning input and restriction, because they involve process abstractions taking two names as arguments. For them, induction yields the following subgoal:

$$\frac{\begin{array}{l}\forall \mathbf{b}, f_Q, \mathbf{a}.\quad \mathbf{wfpa}(f_Q) \ \wedge\ \mathit{fresha}(\mathbf{a}, \lambda x.\ \mathit{ff}_P(\mathbf{b}, x)) \ \wedge\ \mathit{fresha}(\mathbf{a}, f_Q) \ \wedge \\ \qquad\qquad \mathit{ff}_P(\mathbf{b}, \mathbf{a}) = f_Q(\mathbf{a}) \ \longrightarrow\ \lambda x.\ \mathit{ff}_P(b, x) = \lambda x.\ f_Q(x) \\ \forall \mathbf{b}, f_Q, \mathbf{a}.\quad \mathbf{wfpa}(f_Q) \ \wedge\ \mathit{fresha}(\mathbf{a}, \lambda x.\ \mathit{ff}_P(x, \mathbf{b})) \ \wedge\ \mathit{fresha}(\mathbf{a}, f_Q) \ \wedge \\ \qquad\qquad \mathit{ff}_P(\mathbf{a}, \mathbf{b}) = f_Q(\mathbf{a}) \ \longrightarrow\ \lambda x.\ \mathit{ff}_P(x, \mathbf{b}) = \lambda x.\ f_Q(x) \\ \forall \mathbf{b}.\ \mathbf{wfpa}(\lambda x.\ \mathit{ff}_P(\mathbf{b}, x)) \qquad \forall \mathbf{b}.\ \mathbf{wfpa}(\lambda x.\ \mathit{ff}_P(x, \mathbf{b})) \\ \forall \mathbf{b}.\ \mathbf{wfpa}(\lambda x.\ \mathit{ff}_Q(\mathbf{b}, x)) \qquad \forall \mathbf{b}.\ \mathbf{wfpa}(\lambda x.\ \mathit{ff}_Q(x, \mathbf{b})) \\ \mathit{freshaa}(\mathbf{a}, \mathit{ff}_P) \qquad \mathit{freshaa}(\mathbf{a}, \mathit{ff}_Q) \qquad \lambda x.\ \mathit{ff}_P(x, \mathbf{a}) = \lambda x.\ \mathit{ff}_Q(x, \mathbf{a})\end{array}}{\lambda x.\ \mathit{ff}_P(x, \mathbf{c}) = \mathit{ff}_Q(x, \mathbf{c})}$$

The first two premises contain the two induction hypotheses corresponding to instantiations of the first respectively second parameter of ff_P. We make use of both of them by subsequently instantiating the first arguments of ff_P and ff_Q and then the second. Laws (f5) and (f2) from Section 4.1 allow us to choose a name $\mathbf{d}$ which does not occur in $\{\mathbf{a}, \mathbf{c}\} \cup \mathit{fnaa}(ff_P) \cup \mathit{fnaa}(ff_Q)$. We use this name to instantiate the first components of ff_P and ff_Q in the first induction hypothesis, and obtain,

$$\mathbf{wfpa}\,(\lambda x.\, ff_Q(\mathbf{d}, x)) \;\wedge\; \mathit{fresha}(\mathbf{a}, \lambda x.\, ff_P\,(\mathbf{d}, x)) \;\wedge\; \mathit{fresha}\,(\mathbf{a}, \lambda x.\, ff_Q(\mathbf{d}, x)) \;\wedge\; ff_P\,(\mathbf{d}, \mathbf{a}) = ff_Q\,(\mathbf{d}, \mathbf{a}) \;\longrightarrow\; \lambda x.\, ff_P\,(\mathbf{d}, x) = \lambda x.\, ff_Q(\mathbf{d}, x)$$

As all the conditions for the implication can be derived directly from the premises, or applying (f7) and the fact that $\mathbf{d} \neq \mathbf{a}$, this implication can be resolved into a new premise of the form $\lambda x.\, ff_P\,(\mathbf{d}, x) = \lambda x.\, ff_Q(\mathbf{d}, x)$. Similarly, by instantiating the second arguments of ff_P and ff_Q with $\mathbf{c}$ in the second induction hypothesis, we obtain,

$$\mathbf{wfpa}\,(\lambda x.\, ff_Q(x, \mathbf{c})) \;\wedge\; \mathit{fresha}(\mathbf{a}, \lambda x.\, ff_P\,(x, \mathbf{c})) \;\wedge\; \mathit{fresha}\,(\mathbf{a}, \lambda x.\, ff_Q(x, \mathbf{c})) \;\wedge\; ff_P\,(\mathbf{d}, \mathbf{c}) = ff_Q\,(\mathbf{d}, \mathbf{c}) \;\longrightarrow\; \lambda x.\, ff_P\,(x, \mathbf{c}) = \lambda x.\, ff_Q(x, \mathbf{c})$$

The conditions imposed by the implications can be derived like in the above case, this time employing that $\mathbf{c} \neq \mathbf{a}$, yielding the conclusion $\lambda x.\, ff_P\,(x, \mathbf{c}) = \lambda x.\, ff_Q(x, \mathbf{c})$.

In all of the proofs, we have used standard Isabelle proof techniques. Altogether, the proofs of the theorems leading to the extensionality result contain a bit less than 200 lines of code. Note that it was not obvious at the beginning that our well formedness predicate would suffice to prove (Ext), as it does not rule out all exotic terms. From the fact that we have been able to prove *ext*, we can infer that every remaining exotic term is extensionally equal (in the universally quantified sense) to a term which directly corresponds to a process in the π-calculus.

Extensionality for process abstractions taking two names as arguments can be derived from (Ext) if the process abstractions are well formed for all instantiations of their first and second arguments. In the proof, a fresh name is chosen, and (Ext) is instantiated twice, once with that new fresh name, and a second time with the fresh name from the premise, that is, the argument from the proof of (Ext) is replayed, in a proof of about 20 lines of code.

4.4 Beta Expansion

Though seeming fully natural, β-expansion (Exp) has turned out to be the most difficult law to prove. This is because it necessitates an asymmetric treatment of object and meta variables, that is, object variables have to be compared to the name to be abstracted over, whereas meta variables are intended to pass without comparison. Unlike in the proof of (Ext), we cannot directly apply induction, due to the existential quantification in the conclusion. Instead, we encode a methodology that can be used to abstract over a name in a well formed process term without changing its syntactic structure. It remains to prove, by induction, that the process abstraction resulting from such a transformation indeed fulfills the three requirements described in the conclusion of (Exp).

$$
\begin{array}{rcl}
[\![\mathbf{a}, []]\!] & = & \lambda x.\, x \\
[\![\mathbf{a}, (\mathbf{b}, f_a)xs]\!] & = & \mathit{if}\ \mathbf{a} = \mathbf{b}\ \mathit{then}\ f_a\ \mathit{else}\ [\![\mathbf{a}, xs]\!] \\
\\
[\![0,\, xs,\, ys]\!] & = & \lambda x.\, 0 \\
[\![\tau.P,\, xs,\, ys]\!] & = & \lambda x.\, \tau.[\![P,\, xs,\, ys]\!] \\
[\![\bar{\mathbf{a}}\mathbf{b}.P,\, xs,\, ys]\!] & = & \lambda x.\, \overline{[\![a,\, xs]\!](x)}[\![\mathbf{b},\, xs]\!](x).[\![P,\, xs,\, ys]\!](x) \\
[\![\mathbf{a}y.f_P(y),\, xs,\, ys]\!] & = & \lambda x.\, [\![\mathbf{a},\, xs]\!](x)y.\, [\![f_P(\mathit{fst}(ys)),\, (\mathit{fst}(ys),\, (\lambda x.\, y))xs,\, \mathit{tl}(ys)]\!](x) \\
[\![(\nu\, y)f_P(y),\, xs,\, ys]\!] & = & \lambda x.\, (\nu\, y)\, [\![f_P(\mathit{fst}(ys)),\, (\mathit{fst}(ys),\, (\lambda x.\, y))xs,\, \mathit{tl}(ys)]\!](x) \\
[\![P + Q,\, xs,\, ys]\!] & = & \lambda x.\, [\![P,\, xs,\, ys]\!](x) + [\![Q,\, xs,\, ys]\!](x) \\
[\![P \parallel Q,\, xs,\, ys]\!] & = & \lambda x.\, [\![P,\, xs,\, ys]\!](x) \parallel [\![Q,\, xs,\, ys]\!](x) \\
[\![[\mathbf{a} = \mathbf{b}]P,\, xs,\, ys]\!] & = & \lambda x.\, [[\![\mathbf{a},\, xs]\!](x) = [\![\mathbf{b},\, xs]\!](x)][\![P,\, xs,\, ys]\!](x) \\
[\![[\mathbf{a} \neq \mathbf{b}]P,\, xs,\, ys]\!] & = & \lambda x.\, [[\![\mathbf{a},\, xs]\!](x) \neq [\![\mathbf{b},\, xs]\!](x)][\![P,\, xs,\, ys]\!](x) \\
[\![!P,\, xs,\, ys]\!] & = & \lambda x.\, ![\![P,\, xs,\, ys]\!](x)
\end{array}
$$

Table 6: Abstracting over a name in a process.

The transformation We propose a methodology that is based on coercing from higher order to first order syntax and back, using a primitively recursive transformation function $[\![P, xs, ys]\!]$. The two lists, xs and ys, are computed prior to the transformation. The list xs is the *transformation list* telling for every free name in P the names abstraction it is mapped to in the transformation; it contains no element for the name to be abstracted over, but with all other free names $\mathbf{a}$ in P, it associates a constant function $\lambda.\, \mathbf{a}$. The list ys contains as many fresh names as are necessary to instantiate every meta variable in P. When computing ys, we apply $db(P, \mathbf{c})$ for some arbitrary name $\mathbf{c}$, in order to determine the necessary length of ys, see Table 1 for a formal description. The transformation proceeds as follows, refer to Table 6 for its formalization: Every name that is encountered is mapped to the names abstraction denoted in the transformation list xs. Only the name that is to be abstracted over does not occur in xs, hence it is transformed into $\lambda x.\, x$. Whenever the transformation comes across a binder, that is, input or restriction, it instantiates the continuation with the first fresh name from ys, that is, $\mathit{fst}(ys)$, and adds a pair $(\mathit{fst}(ys),\, (\lambda x.\, y))$ to xs, where y is the meta variable given by the binder. When the transformation later encounters the instantiated (object level) name, it abstracts over it again. This methodology — that is, first instantiating and later restoring meta variables in a process abstraction — prevents meta variables from being compared with the object variable to be abstracted over. Note that any such comparison meta and object variables in the transformation function for names, $[\![\mathbf{a}, xs]\!]$, could not be evaluated immediately, and, hence, would necessarily result in an exotic process that would not even be extensionally equal to the intended abstraction.

Well formedness We call an abstraction over a transformation list well formed if it only applies well formed names abstractions (see Table 4 for a

definition):

$$\frac{}{\mathbf{wftrl}\,(\lambda x.\,[])}\;\mathbf{W}_1^t \qquad \frac{\mathbf{wfnaa}\,(ff_a) \quad \mathbf{wftrl}\,(f_{xs})}{\mathbf{wftrl}\,(\lambda x.\,(\mathbf{a}, ff_a(x))f_{xs}(x))}\;\mathbf{W}_2^t$$

The following two theorems prove that the transformation described above produces well formed process abstractions when applied to well formed processes:

$$\frac{\mathbf{wfpa}\,(f_P) \quad \mathbf{wftrl}\,(f_{xs})}{\begin{array}{c}\mathbf{wfpa}\,([\![f_P(\mathbf{c}), f_{xs}(\mathbf{d}), ys]\!]) \wedge \\ \mathbf{wfpa}\,(\lambda x.\,[\![f_P(\mathbf{c}), f_{xs}(x), ys]\!](\mathbf{b}))\end{array}} \qquad \frac{\mathbf{wfp}\,(P) \quad \forall(\mathbf{a}, f_a) \in xs.\,\mathbf{wfna}\,(f_a)}{\mathbf{wfpa}\,([\![P, xs, ys]\!])}$$

The proofs of the two theorems are tedious but purely technical inductions. The main difficulty was to formulate a suitable notion of abstractions over transformation lists, and the first of the two theorems. Note that the second theorem can only be proved as a consequence of the first.

Freshness In order to prove that the transformation really eliminates the intended name $\mathbf{a}$, we choose a name $\mathbf{b} \neq \mathbf{a}$, and derive by two technical inductions,

$$\frac{\mathbf{wfpa}\,(f_P) \qquad \forall(\mathbf{d}, f_d) \in xs.\;\; \mathbf{a} \neq f_d(\mathbf{b}) \qquad \mathbf{a} \neq \mathbf{b}}{\mathit{fresh}(\mathbf{a}, [\![f_P(\mathbf{c}), xs, ys]\!](\mathbf{b}))}$$

$$\frac{\mathbf{wfp}\,(P) \qquad \forall(\mathbf{d}, f_d) \in xs.\;\; \mathbf{a} \neq f_d(\mathbf{b}) \qquad \mathbf{a} \neq \mathbf{b}}{\mathit{fresh}(\mathbf{a}, [\![P, xs, ys]\!](\mathbf{b}))}$$

Again the proof of the second theorem is based on the first. In the proofs, we make extensive use of law (f6), see Section 4.1.

Equality. It remains to show, again by induction, that a reinstantiation of a transformation yields the original process. The proofs make use of the monotonicity and extensionality theorems proved in Sections 4.2 and 4.3, as well as of the well formedness and freshness results from the previous two sections. It is therefore that we have to guarantee, by using *db*, that *ys* contains at least as many names as there are nested binders in a process. We use a predicate **nodups**, to ensure that *ys* does not contain duplicates. The function *fst* maps pairs to their first item; when applied to a list $(a_1, b_1) \ldots (a_n, b_n)$ it returns $a_1 \ldots a_n$.

$$\frac{\begin{array}{c}\mathbf{wfpa}\,(f_P) \qquad \forall(\mathbf{b}, f_b) \in xs.\;\; f_b = \lambda x.\,\mathbf{b} \qquad \mathit{dba}\,(f_P, \mathbf{c}) \leq |ys| \\ \mathit{fna}\,(f_P) \subseteq \{\mathbf{a}\} \cup \mathit{fst}\,(xs) \qquad \mathbf{a} \notin \mathit{fst}\,(xs) \qquad \mathbf{d} \in \mathit{fst}\,(xs) \\ \mathbf{nodups}\,(ys) \qquad ys \cap (\{\mathbf{a}\} \cup \mathit{fst}\,(xs)) = \emptyset\end{array}}{[\![f_P(\mathbf{d}), xs, ys]\!](\mathbf{a}) = f_P(\mathbf{d})}$$

$$\frac{\begin{array}{c}\mathbf{wfp}\,(P) \quad \forall(\mathbf{b}, f_b) \in xs.\;\; f_b = \lambda x.\,\mathbf{b} \quad \mathit{db}\,(P, \mathbf{c}) \leq |ys| \quad \mathit{fn}\,(P) \subseteq \{\mathbf{a}\} \cup \mathit{fst}\,(xs) \\ \mathbf{a} \notin \mathit{fst}\,(xs) \qquad \mathbf{nodups}\,(ys) \qquad ys \cap (\{\mathbf{a}\} \cup \mathit{fst}\,(xs)) = \emptyset\end{array}}{[\![P, xs, ys]\!](\mathbf{a}) = P}$$

The proofs are tedious but purely technical. Whenever a process abstraction is encountered, the first name in *ys* is used as a fresh name, and (EXT) is applied.

The mechanization of the proofs of β-expansion in Isabelle/HOL consist of about 350 lines of code. It has turned out that naive attempts to set up a

proof of β-expansion are bound to fail, as object and meta variables cannot be compared, but meta variables cannot be neglected either. As long as both object (the free names, in our case) and meta variables (the bound names) both play a role in the semantic analysis of a language, similar principles like (MON), (EXT), and (EXP) will have to be derived. We are confident that the techniques presented in this paper can be adapted to related problems with moderate effort.

5 Discussion

In this section, we discuss questions that have arisen during our work.

Why choose a shallow embedding? Binding mechanisms are tedious to deal with in a deep embedding. This is especially hard for the π-calculus, because it is particularly characterized by its two binders, input and restriction. As, for future prospects, we are particularly interested in reasoning about concrete π-calculus processes within our framework, we want to keep the related effort as small as possible.

Why well formed processes? We have decided to introduce a well formedness predicate, in order to rule out exotic terms and, simultaneously, to obtain an induction principle. This gives us the opportunity to reason both *within* and *about* the π-calculus in our formalization.

Why Isabelle/HOL? We have chosen Isabelle/HOL mainly for the following reasons. First, Isabelle implements powerful automatic tactics that facilitate especially prototypical proofs. Second, Isabelle/HOL provides a large database of data structures and theorems about them, for instance, sets and lists. Third, owing to its recent conservative extension, Isabelle is now able to deal with higher order abstract syntax. And, finally, all the features that we exploit, that is, recursion and induction, are generic extensions, that is, have been justified within the object level.

What about other provers? The results presented in this work do not especially rely on Isabelle. A natural question is whether one could adapt them to [11], so that the properties (MON), (EXT), and (EXP) could be formally justified there as well. To do this, it would be necessary to enrich Coq extensional notion of equality, like it is formalized in Isabelle/HOL. With Coq providing powerful induction mechanisms and automatic tactics, we are confident that the methodology presented in this paper could be mechanized there too.

In logical systems like λProlog [18] and Elf [22], encodings naturally exploit higher order abstract syntax, and exotic terms are excluded automatically by the meta logic. On the other hand, these frameworks do not offer adequate induction principles, hence syntactic properties often cannot be derived within the encoding. Recent work attemps to bridge this gap: the theorem prover Twelf [23] implements a meta logic based on Elf which offers a form of automated induction. While it may be possible to adapt the results presented in this paper to a framework like Twelf, it remains an open question how much support these systems can offer in semantic proofs, concerning transition systems and bisimulations.

How many names do we need? Any type with at least countably infinitely many elements fits our formalization. The reason why there cannot be less names is that the proofs of extensionality and β-expansion are based on the creation of fresh names for processes or process abstractions. The situation is less simple in the work of Honsell et al. [11], where the meta level of Coq is employed to guarantee for the absence of exotic terms. A necessary condition for this is that an equality relation on names can only be defined in `Prop`, and not in `Set`. This rules out any inductive type. Honsell et al. suggest to pick the real numbers. The main point, however, is to choose a type in which *any* manipulation of names, be it by functions or by relations, is impossible. This means that when choosing the real numbers, one has to take care that no operator is defined for them in the formalization, otherwise an inconsistency can be derived by applying (EXT) to the exotic terms constructed with the operators.

What about justifying the theory of contexts? This work is not a formal justification of the "theory of contexts" applied in [11]. To do this, we would have to encode the meta level of Coq, that is, the Calculus of Constructions, in a prover, and then employ, for instance, a category theoretical argument, which seems quite illusive. Our work should rather be seen as a formal justification of the three syntactic properties presented in the "theory of contexts", within a shallow formalization of the π-calculus. As such, our work can be related to that of Gordon and Melham [7]. There, an axiomatization of α-conversion in HOL is proposed, which serves as a framework for the derivation of syntax definitions, as well as substitution and induction principles.

Is the theory of contexts really necessary? The three properties presented in the "theory of contexts", and formally justified in this paper, are essential for the semantic analysis of π-calculus processes. The reason is that in transition systems and bisimulation proofs both free and bound names play a role. Recently, Despeyroux has proposed a formalized strong late transition system for a fragment of the π-calculus within a shallow embedding, which applies functions as derivatives in order to reduce the number of instantiations [3]. It will have to be investigated how this formalism can be extended to the full π-calculus, and whether it necessitates a "theory of contexts" in order to reason about semantics or not.

What about other languages? The theory of well formedness has originally been developed for the λ-calculus [5, 4]. In the same way as we have adapted and extended it for reasoning about the π-calculus, it can serve as a means of introducing induction into shallow formalizations of other languages as well. If these languages possess binders of higher order, however, variables have to be used to denote such higher order objects within binding constructs, see [4] for details.

Acknowledgements: We thank Joëlle Despeyroux, Gilles Dowek, Javier Esparza, René Lalement, Tobias Nipkow, and Peter Rossmanith, for helpful comments and discussions. This work has been supported by the PROCOPE project 9723064, "Verification Techniques for Higher Order Imperative Concurrent Languages".

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] J. Despeyroux. A higher-order specification of the π-calculus. In *Proc. TCS'00*, LNCS. Springer, 2000. To appear.

[4] J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order abstract syntax in Coq. In *Proc. TLCA'95*, volume 902 of *LNCS*, pages 124–138. Springer, 1995.

[5] 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.

[6] J. Despeyroux, F. Pfenning, and C. Schürmann. Primitive recursion for higher-order abstract syntax. In *Proc. TLCA'97*, volume 1210 of *LNCS*, pages 147–163. Springer, 1997. An extended version will appear in TCS.

[7] A. Gordon and T. Melham. Five axioms of alpha-conversion. In *Proc. TPHOL'96*, volume 1125 of *LNCS*, pages 173–190. Springer, 1996.

[8] L. Henry-Gréard. Proof of the subject reduction property for a pi-calculus in coq. Technical Report RR-3698, INRIA, 1999.

[9] 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.

[10] M. Hofmann. Semantical analysis of higher-order abstract syntax. In *Proc. LICS'99*, volume 158, pages 204–213. IEEE, 1999.

[11] F. Honsell, M. Miculan, and I. Scagnetto. π-calculus in (co)inductive type theory. *Theoretical Computer Science*, 2000. To appear.

[12] B. Mammass. *Méthodes et Outils pour les Preuve Compositionnelles de Systèmes Paralleèles (in french).* PhD thesis, Pierre et Marie Curie University, Paris, 1999.

[13] T. Melham. A mechanized theory of the π-calculus in HOL. *Nordic Journal of Computing*, 1(1):50–76, 1995.

[14] 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.

[15] R. Milner. Functions as processes. *Journal of Math. Struct. in Computer Science*, 17:119–141, 1992.

[16] R. Milner. *Communicating and Mobile Processes.* Cambridge University Press, 1999.

[17] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.

[18] G. Nadathur and D. Miller. An overview of λprolog. In MIT Press, editor, *Proc. LPC'98*, pages 810–827, 1998.

[19] L. C. Paulson. Isabelle's object-logics. Technical Report 286, University of Cambridge, Computer Laboratory, 1993.

[20] L. C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In *Procs CADE'94*, volume 814 of *LNAI*, pages 148–161. Springer, 1994.

[21] L. C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *LNCS*. Springer, 1994.

[22] F. Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Proc. LICS'89*, pages 313–321. IEEE, 1989.

[23] F. Pfenning and C. Schürmann. System description: Twelf – a meta-logical framework for deductive systems. In *Proc. CAD'99*, volume 1632 of *LNAI*, pages 202–206. Springer, 1999.

[24] C. Röckl and D. Sangiorgi. A π-calculus process semantics of concurrent idealised ALGOL. In *Proc. FOSSACS'99*, volume 1578 of *LNCS*, pages 306–321. Springer, 1999.

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