Higher-Order Abstract Syntax with Induction in Isabelle/HOL: Formalizing the π-Calculus and Mechanizing the Theory of Contexts

Christine Röckl[1], Daniel Hirschkoff[2], and Stefan Berghofer[1]

[1] Fakultät für Informatik, Technische Universität München, D-80290 München
Email: {roeckl,berghofe}@in.tum.de

[2] LIP – École Normale Supérieure de Lyon, 46, allée d'Italie, F-69364 Lyon Cedex 7
Email: Daniel.Hirschkoff@ens-lyon.fr

Abstract. *Higher-order abstract syntax is a natural way to formalize programming languages with binders, like the π-calculus, because α-conversion, instantiations and capture avoidance are delegated to the meta-level of the provers, making tedious substitutions superfluous. However, such formalizations usually lack structural induction, which makes syntax-analysis impossible. Moreover, when applied in logical frameworks with object-logics, like Isabelle/HOL or standard extensions of Coq,* exotic *terms can be defined, for which important syntactic properties become invalid.*

The paper presents a formalization of the π-calculus in Isabelle/HOL, using well-formedness *predicates which both eliminate exotic terms and yield structural induction. These induction-principles are then used to derive the* Theory of Contexts *fully within the mechanization.*

1 Motivation

The π-calculus was introduced to model and analyse mobile systems [18,17]. In it, communication channels and messages belong to the same sort, called *names*. This simplicity gives the π-calculus the power to encode the λ-calculus [16], as well as higher-order object-oriented and imperative languages [27,26]. Communications are synchronous, that is, a sender $\bar{\mathbf{a}}\mathbf{b}.P$ transmits a message $\mathbf{b}$ to a recipient $\mathbf{ax}.Q$, in a transition $\bar{\mathbf{a}}\mathbf{b}.P \,|\, \mathbf{ax}.Q \xrightarrow{\tau} P \,|\, Q\{\mathbf{b}/\mathbf{x}\}$. Usually, a substitution is applied to describe that $\mathbf{b}$ replaces $\mathbf{x}$ in Q. This can be tedious for processes with binders, like $Q = (\nu\mathbf{b})Q'$, where a further substitution is necessary to avoid name-capture: $Q\{\mathbf{b}/\mathbf{x}\} =_\alpha (\nu\mathbf{b}')Q'\{\mathbf{b}'/\mathbf{b}, \mathbf{b}/\mathbf{x}\}$. Higher-order abstract syntax builds upon a functional view, considering binders as abstractions with respect to an underlying λ-calculus to which the replacement of names and capture-avoidance are delegated. The above transition could thus be rewritten as $\bar{\mathbf{a}}\mathbf{b}.P \,|\, \mathbf{a}x.f_Q(x) \xrightarrow{\tau} P \,|\, f_Q(\mathbf{b})$, where the function $f_Q(x) = (\nu b)f_{Q'}(b, x)$ corresponds to the process Q, and the process $f_Q(\mathbf{b})$ represents $Q\{\mathbf{b}/\mathbf{x}\}$.

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 [22,20], as it is generic and offers a large range of powerful proof-techniques.

Formalizing the π-calculus. General-purpose theorem provers distinguish two levels of reasoning. Upon a *meta-logic* 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 (*deep embedding*), or by exploiting the functional mechanisms of the meta-level (*shallow embedding*).

• Following the classical way, the syntax of the π-calculus is described in terms of a recursive datatype $P ::= 0 \mid \bar{\mathbf{a}}\mathbf{b}.P \mid \mathbf{ab}.P \mid \ldots$, and substitution functions are introduced explicitely by the user. These deep embeddings of first-order syntax allow the user to make full use of structural induction, which is a vital proof-tool in syntax analysis. Several formalizations of this kind have been studied in various theorem provers [14,1,9,12,8]. They give evidence that proofs about π-calculus processes in deep embeddings are generally hard, and that it would be tedious to try to tackle larger proofs. The reason is that the π-calculus is particularly characterized by its binders, *input* and *restriction*; hence, a lot of effort goes into intricate reasoning about substitutions.

• In contrast, higher-order abstract syntax used in a shallow embedding, builds on a recursive datatype of the form $P ::= 0 \mid \bar{\mathbf{a}}\mathbf{b}.P \mid \mathbf{a}x.f_P(x) \mid \ldots$, where f_P is a function mapping names to π-calculus processes. Here, the capture-avoiding replacement of names are dealt with automatically by the meta-level of the theorem-prover, freeing the user from a tedious implementation and application of substitutions. Shallow embeddings of the π-calculus have been studied in Coq and λProlog [15,11,3]. Unfortunately, higher-order datatypes are not recursive in a strict sense, due to the functions in the continuations of binders. As a consequence, plain structural induction does not work, making syntax-analysis impossible. Even worse, in logical frameworks with object-level constructors, so-called *exotic* terms can be derived. Consider, for example,

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

The term f_E is *exotic*, because it is built from an object-level conditional (not from a π-calculus conditional, which we represent in terms of matching and mismatching), and does not correspond to any process in the usual syntax of the π-calculus, whereas f_W can be considered as *valid*, or *well-formed*.

• A third variation uses higher-order abstract syntax, but within a deep embedding, using a first-order formalization of a λ-calculus as a (pseudo) "meta-level" for embedding the π-calculus thereupon. As a consequence, substitutions do not have to be defined for the (large) language, but only for the (smaller) λ-calculus, while still reasoning entirely on the object-level. For formalizations of the π-calculus, this approach has been followed in [7,6]. Up to this point, however, these frameworks have not been tested in larger-scale syntax-analyses.

The theory of contexts. The *theory of contexts* for the π-calculus consists of three syntactic properties that are essential for a semantic analysis of processes in a shallow embedding. It was introduced in the shape of axioms by Honsell, Miculan, and Scagnetto to reason about strong transitions of π-calculus processes in a in Coq formalization, with justification on paper [11,10]. Our own experience gives evidence that the three properties are sufficient for weak transitions as well[1]. 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.

Outlook of the paper. In this paper, we present a shallow embedding of the π-calculus in Isabelle/HOL using inductive *well-formedness predicates* which rule out exotic terms and, simultaneously, allow us to perform structural induction on π-calculus processes. Our work was inspired by a similar approach for shallow embeddings of the λ-calculus in Coq, by Despeyroux, Felty, and Hirschowitz [5,4]. As a result, we are able to derive the theory of contexts fully mechanically within Isabelle/HOL. The resulting formalization thus provides a generic framework for the semantic analysis of the π-calculus (for instance, transitions, bisimulations), as well as of concurrent and mobile systems modelled within the π-calculus.

The paper is organized as follows: In Section 2, we give some 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 the theory of contexts. In Section 5, we discuss some questions related to our results.

2 Isabelle/HOL

We use the general-purpose theorem-prover Isabelle [22], implementing higher-order intuitionistic logic on its meta-level, and formalize the π-calculus in its instantiation HOL for higher-order logic [20]. 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

[1] Technically, properties of weak transitions are usually derived from corresponding properties of strong transitions by induction on the number of silent steps.

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 subgoals; rules that might yield unprovable subgoals are only applied if they succeed in terminating the proof of a subgoal.

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 [2,21]. 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, and was introduced to reason about mobile systems [17,18]. 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 and 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 present a shallow embedding of the π-calculus, and present inductive well-formedness predicates that simultaneously rule out exotic terms and provide structural induction. We use the same datatype as [3,11], so that our results are comparable to these formalizations.

Names In semantic analysis, processes are often instantiated with *fresh* names; hence, the type of names has to be at least countably infinite. Also the theory of contexts hinges on fresh names. We do not commit ourselves to a specific type but use 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, 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, that is, $x, y, \ldots$, for the latter.

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*,

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

$$
\mathit{fna}(f_P) \overset{def}{=} \{\,\mathbf{a} \mid \forall \mathbf{b}.\, \mathbf{a}\in \mathit{fn}(f_P(\mathbf{b}))\,\} \qquad \mathit{dba}(f_P,\mathbf{c}) \overset{def}{=} \mathit{db}(f_P(\mathbf{c}),\mathbf{c})
$$

$$
\mathit{fnaa}(ff_P) \overset{def}{=} \{\,\mathbf{a} \mid \forall \mathbf{b}.\, \mathbf{a}\in \mathit{fna}(\lambda x.\, ff_P(\mathbf{b},x))\,\}
$$

Table 1. Computing the free names *fn* and depth of binders *db* of a process.

mismatching, and *replication*. In a shallow embedding, 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 positive position.

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

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 which are not in the scope of a binder are called *free*, whereas 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-variables of the theorem-prover. Free names are represented by object-variables, and we compute them

$$\frac{}{\mathbf{wfp}\,(0)}\ \mathbf{W}_0 \qquad \frac{\mathbf{wfp}\,(P)}{\mathbf{wfp}\,(\tau.P)}\ \mathbf{W}_1 \qquad \frac{\mathbf{wfp}\,(P)}{\mathbf{wfp}\,(\bar{\mathbf{a}}\mathbf{b}.P)}\ \mathbf{W}_2 \qquad \frac{\mathbf{wfpa}\,(f_P)}{\mathbf{wfp}\,(\mathbf{a}y.f_P(y))}\ \mathbf{W}_3$$

$$\frac{\mathbf{wfpa}\,(f_P)}{\mathbf{wfp}\,((\nu\,y)f_P(y))}\ \mathbf{W}_4 \qquad \frac{\mathbf{wfp}\,(P) \quad \mathbf{wfp}\,(Q)}{\mathbf{wfp}\,(P+Q)}\ \mathbf{W}_5 \qquad \frac{\mathbf{wfp}\,(P) \quad \mathbf{wfp}\,(Q)}{\mathbf{wfp}\,(P \parallel Q)}\ \mathbf{W}_6$$

$$\frac{\mathbf{wfp}\,(P)}{\mathbf{wfp}\,([\mathbf{a}=\mathbf{b}]P)}\ \mathbf{W}_7 \qquad \frac{\mathbf{wfp}\,(P)}{\mathbf{wfp}\,([\mathbf{a}\neq\mathbf{b}]P)}\ \mathbf{W}_8 \qquad \frac{\mathbf{wfp}\,(P)}{\mathbf{wfp}\,(!P)}\ \mathbf{W}_9$$

Table 2. Well-formed Processes.

$$\frac{}{\mathbf{wfpa}\,(\lambda x.\,0)}\ \mathbf{W}^a_0 \qquad \frac{\mathbf{wfpa}\,(f_P)}{\mathbf{wfpa}\,(\lambda x.\,\tau.f_P(x))}\ \mathbf{W}^a_1 \qquad \frac{\mathbf{wfna}\,(f_a) \quad \mathbf{wfna}\,(f_b) \quad \mathbf{wfpa}\,(f_P)}{\mathbf{wfpa}\,(\lambda x.\,\overline{f_a(x)}f_b(x).f_P(x))}\ \mathbf{W}^a_2$$

$$\frac{\mathbf{wfna}\,(f_a) \quad \forall\,b.\,\mathbf{wfpa}\,(\lambda x.\,ff_P(b,x)) \quad \forall\,b.\,\mathbf{wfpa}\,(\lambda x.\,ff_P(x,b))}{\mathbf{wfpa}\,(\lambda x.\,f_a(x)y.ff_P(y,x))}\ \mathbf{W}^a_3$$

$$\frac{\forall\,b.\,\mathbf{wfpa}\,(\lambda x.\,ff_P(b,x)) \quad \forall\,b.\,\mathbf{wfpa}\,(\lambda x.\,ff_P(x,b))}{\mathbf{wfpa}\,(\lambda x.\,(\nu\,y)ff_P(y,x))}\ \mathbf{W}^a_4$$

$$\frac{\mathbf{wfpa}\,(f_P) \quad \mathbf{wfpa}\,(f_Q)}{\mathbf{wfpa}\,(\lambda x.\,f_P(x)+f_Q(x))}\ \mathbf{W}^a_5 \qquad \frac{\mathbf{wfpa}\,(f_P) \quad \mathbf{wfpa}\,(f_Q)}{\mathbf{wfpa}\,(\lambda x.\,f_P(x) \parallel f_Q(x))}\ \mathbf{W}^a_6$$

$$\frac{\mathbf{wfna}\,(f_a) \quad \mathbf{wfna}\,(f_b) \quad \mathbf{wfpa}\,(f_P)}{\mathbf{wfpa}\,(\lambda x.\,[f_a(x)=f_b(x)].f_P(x))}\ \mathbf{W}^a_7 \qquad \frac{\mathbf{wfna}\,(f_a) \quad \mathbf{wfna}\,(f_b) \quad \mathbf{wfpa}\,(f_P)}{\mathbf{wfpa}\,(\lambda x.\,[f_a(x)\neq f_b(x)].f_P(x))}\ \mathbf{W}^a_8$$

$$\frac{\mathbf{wfpa}\,(f_P)}{\mathbf{wfpa}\,(\lambda x.\,!f_P(x))}\ \mathbf{W}^a_9$$

Table 3. Well-formed Process-Abstractions.

with 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 the free names as one might expect; for f_E, for instance, *fna* computes the empty set. For all *well-formed* processes, however, *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 **fresh** (a, P) iff $a \notin fn(P)$, and **fresha** (a, f_P) iff $a \notin fna(f_P)$ and **freshaa** (a, ff_P) iff $a \notin fnaa(ff_P)$, respectively.

Well-formedness We introduce *well-formedness* predicates with which we simultaneously eliminate exotic processes like f_E from Section 1, and obtain structural induction. 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. Rules W_3, W_4, W^a_3, and W^a_4, concerning the binders, 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. However, a second-order predicate suffices to rule

$$\frac{}{\mathbf{wfna}\,(\lambda x.\,x)}\ \mathbf{W}^n_1 \qquad \frac{}{\mathbf{wfna}\,(\lambda x.\,\mathbf{a})}\ \mathbf{W}^n_2$$

$$\frac{}{\mathbf{wfnaa}\,(\lambda(x,y).\,x)}\ \mathbf{W}^n_3 \qquad \frac{}{\mathbf{wfnaa}\,(\lambda(x,y).\,y)}\ \mathbf{W}^n_4 \qquad \frac{}{\mathbf{wfnaa}\,(\lambda(x,y).\,\mathbf{a})}\ \mathbf{W}^n_5$$

Table 4. Well-formed Names-Abstractions.

out at least those exotic terms that might render syntactic properties of the original language incorrect in the encoding, see Section 5 for a discussion. The process abstraction f_E from the introduction, for instance, is ruled out as exotic by **wfpa**. We are thus able to derive in Section 4 the validity of the theory of contexts for the set of well-formed processes and abstractions.

Counting Binders In the proof in Section 4.4, we use coercion from higher-order syntax to first-order syntax by instantiating meta-variables with fresh names. In order to provide a sufficient amount of fresh names, we statically compute the *depth of binders* with a primitively recursive function, *db*; for a formal definition, see Table 1. The function computes the maximal number of binders along each path in the process-tree, instantiating process-abstractions with an auxiliary name **c**. Like *fn* above, *db* only yields sensible results for well-formed processes.

4 Deriving Syntactic Properties in Isabelle/HOL

We now turn to a formal derivation of the *theory of contexts* [11] for well-formed processes. It consists of three general syntactic properties of languages with binders, which can be described intuitively as follows:

(MON) Monotonicity: *If a name* **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* **a**.

(EXP) β-Expansion: *Every process* P *can be abstracted over an arbitrary name* **a***, yielding a suitable process-abstraction.*

A formal description is depicted in Table 5, for well formed processes and process abstractions. Recall from Section 1 that extensionality only holds for well-formed process-abstractions. Also in the third law (EXP), describing β-expansion, we only consider well-formed processes and process-abstractions. The reason is that, although β-expansion holds for arbitrary processes and abstractions over them, we want to strengthen it as much as possible, so that it can be used together with (EXT) in the semantic analysis of processes.

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

$$\frac{\mathbf{wfpa}\,(f_P) \quad \mathbf{wfpa}\,(f_Q) \quad \mathbf{fresha}\,(\mathbf{a}, f_P) \quad \mathbf{fresha}\,(\mathbf{a}, f_Q) \quad f_P(\mathbf{a}) = f_Q(\mathbf{a})}{f_P = f_Q}\ (\textsc{Ext})$$

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

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

4.1 Free and Fresh Names

In the proofs of (Ext) and (Exp), we rely on the fact that there exist at least countably infinitely many names—see Section 3—so we can always find a fresh name with which to instantiate a process abstraction. Laws (f1)–(f7) formalize these basic properties; their proofs in Isabelle/HOL are standard, and yield scripts of a few lines only.

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

$$\text{(f3)}\ \mathit{finite}\,(\mathit{fn}\,(P)) \qquad \text{(f4)}\ \mathit{finite}\,(\mathit{fna}\,(f_P)) \qquad \text{(f5)}\ \mathit{finite}\,(\mathit{fnaa}\,(ff_P))$$

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

$$\text{(f7)}\ \frac{\forall \mathbf{b}.\ \mathbf{wfpa}\,(\lambda x.\ ff_P\,(\mathbf{b}, x)) \quad \forall \mathbf{b}.\ \mathbf{wfpa}\,(\lambda x.\ ff_P\,(x, \mathbf{b})) \quad \mathbf{freshaa}\,(\mathbf{a}, ff_P) \quad \mathbf{c} \neq \mathbf{a}}{\mathbf{fresha}\,(\mathbf{a}, \lambda x.\ ff_P\,(\mathbf{c}, x))}$$

Laws (f6) and (f7) express that a name **a** which is fresh for a well-formed process-abstraction, is necessarily fresh for every instantiation except **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 **a** is only free in a process-abstraction f_P according to *fnaa*, if it is free in every instantiation; hence for **a** to be fresh in f_P, it suffices to present a single name **b** as a witness for which **a** is fresh in $f_P(\mathbf{b})$. 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, as the counter-example in Section 1 shows.

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

$$\frac{\begin{array}{l} \forall \mathbf{b}, f_Q, \mathbf{a}.\ \ \mathbf{wfpa}\,(f_Q) \ \wedge\ \mathbf{fresha}\,(\mathbf{a}, \lambda x.\, ff_P\,(\mathbf{b}, x)) \ \wedge\ \mathbf{fresha}\,(\mathbf{a}, f_Q) \ \wedge \\ \qquad\qquad ff_P\,(\mathbf{b}, \mathbf{a}) = f_Q\,(\mathbf{a}) \ \longrightarrow\ \lambda x.\, ff_P(b, x) = \lambda x.\, f_Q(x) \\ \forall \mathbf{b}, f_Q, \mathbf{a}.\ \ \mathbf{wfpa}\,(f_Q) \ \wedge\ \mathbf{fresha}\,(\mathbf{a}, \lambda x.\, ff_P\,(x, \mathbf{b})) \ \wedge\ \mathbf{fresha}\,(\mathbf{a}, f_Q) \ \wedge \\ \qquad\qquad ff_P\,(\mathbf{a}, \mathbf{b}) = f_Q\,(\mathbf{a}) \ \longrightarrow\ \lambda x.\, ff_P(x, \mathbf{b}) = \lambda x.\, f_Q(x) \\[1ex] \qquad \forall \mathbf{b}.\ \mathbf{wfpa}\,(\lambda x.\, ff_P\,(\mathbf{b}, x)) \qquad\qquad \forall \mathbf{b}.\ \mathbf{wfpa}\,(\lambda x.\, ff_P\,(x, \mathbf{b})) \\ \qquad \forall \mathbf{b}.\ \mathbf{wfpa}\,(\lambda x.\, ff_Q\,(\mathbf{b}, x)) \qquad\qquad \forall \mathbf{b}.\ \mathbf{wfpa}\,(\lambda x.\, ff_Q\,(x, \mathbf{b})) \\[1ex] \mathbf{freshaa}\,(\mathbf{a}, ff_P) \qquad \mathbf{freshaa}\,(\mathbf{a}, ff_Q) \qquad \lambda x.\, ff_P\,(x, \mathbf{a}) \ =\ \lambda x.\, ff_Q\,(x, \mathbf{a}) \end{array}}{\lambda x.\, ff_P\,(x, \mathbf{c}) \ =\ ff_Q\,(x, \mathbf{c})}$$

The first two premises are the induction-hypotheses corresponding to instantiations of the first (respectively second) parameter of ff_P. We use 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 $\{\mathbf{a}, \mathbf{c}\} \cup \mathit{fnaa}\,(ff_P) \cup \mathit{fnaa}\,(ff_Q)$. Instantiating the first components of ff_P and ff_Q in the first induction hypothesis, we obtain,

$$\begin{array}{l} \mathbf{wfpa}\,(\lambda x.\, ff_Q(\mathbf{d}, x)) \ \wedge\ \mathbf{fresha}\,(\mathbf{a}, \lambda x.\, ff_P\,(\mathbf{d}, x)) \ \wedge\ \mathbf{fresha}\,(\mathbf{a}, \lambda x.\, ff_Q(\mathbf{d}, x)) \ \wedge \\ ff_P\,(\mathbf{d}, \mathbf{a}) = ff_Q\,(\mathbf{d}, \mathbf{a}) \ \longrightarrow\ \lambda x.\, ff_P\,(\mathbf{d}, x) = \lambda x.\, ff_Q(\mathbf{d}, x). \end{array}$$

As all the hypotheses for the implication can be established directly from the premises, or from (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,

$$\begin{array}{l} \mathbf{wfpa}\,(\lambda x.\, ff_Q(x, \mathbf{c})) \ \wedge\ \mathbf{fresha}\,(\mathbf{d}, \lambda x.\, ff_P\,(x, \mathbf{c})) \ \wedge\ \mathbf{fresha}\,(\mathbf{d}, \lambda x.\, ff_Q(x, \mathbf{c})) \ \wedge \\ ff_P\,(\mathbf{d}, \mathbf{c}) = ff_Q\,(\mathbf{d}, \mathbf{c}) \ \longrightarrow\ \lambda x.\, ff_P\,(x, \mathbf{c}) = \lambda x.\, ff_Q(x, \mathbf{c}). \end{array}$$

The conditions of the implications can be derived like in the above case, this time employing that $\mathbf{c} \neq \mathbf{d}$, 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 proof-script 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

$$
\begin{aligned}
\llbracket \mathbf{a}, [] \rrbracket &= \lambda x.\, x \\
\llbracket \mathbf{a}, (\mathbf{b}, f_a)xs \rrbracket &= \mathit{if}\ \mathbf{a} = \mathbf{b}\ \mathit{then}\ f_a\ \mathit{else}\ \llbracket \mathbf{a}, xs \rrbracket \\[4pt]
\llbracket 0,\, xs,\, ys \rrbracket &= \lambda x.\, 0 \\
\llbracket \tau.P,\, xs,\, ys \rrbracket &= \lambda x.\, \tau.\llbracket P,\, xs,\, ys \rrbracket \\
\llbracket \bar{\mathbf{a}}\mathbf{b}.P,\, xs,\, ys \rrbracket &= \lambda x.\, \overline{\llbracket a,\, xs \rrbracket(x)}\llbracket \mathbf{b},\, xs \rrbracket(x).\llbracket P,\, xs,\, ys \rrbracket(x) \\
\llbracket \mathbf{a}y.f_P(y),\, xs,\, ys \rrbracket &= \lambda x.\, \llbracket \mathbf{a},\, xs \rrbracket(x)y.\, \llbracket f_P(\mathit{fst}(ys)),\, (\mathit{fst}(ys),\, (\lambda x.\, y))xs,\, \mathit{tl}(ys) \rrbracket(x) \\
\llbracket (\nu\, y) f_P(y),\, xs,\, ys \rrbracket &= \lambda x.\, (\nu\, y)\, \llbracket f_P(\mathit{fst}(ys)),\, (\mathit{fst}(ys),\, (\lambda x.\, y))xs,\, \mathit{tl}(ys) \rrbracket(x) \\
\llbracket P + Q,\, xs,\, ys \rrbracket &= \lambda x.\, \llbracket P,\, xs,\, ys \rrbracket(x) + \llbracket Q,\, xs,\, ys \rrbracket(x) \\
\llbracket P \parallel Q,\, xs,\, ys \rrbracket &= \lambda x.\, \llbracket P,\, xs,\, ys \rrbracket(x) \parallel \llbracket Q,\, xs,\, ys \rrbracket(x) \\
\llbracket [\mathbf{a} = \mathbf{b}]P,\, xs,\, ys \rrbracket &= \lambda x.\, [\llbracket \mathbf{a},\, xs \rrbracket(x) = \llbracket \mathbf{b},\, xs \rrbracket(x)]\llbracket P,\, xs,\, ys \rrbracket(x) \\
\llbracket [\mathbf{a} \neq \mathbf{b}]P,\, xs,\, ys \rrbracket &= \lambda x.\, [\llbracket \mathbf{a},\, xs \rrbracket(x) \neq \llbracket \mathbf{b},\, xs \rrbracket(x)]\llbracket P,\, xs,\, ys \rrbracket(x) \\
\llbracket !P,\, xs,\, ys \rrbracket &= \lambda x.\, !\llbracket P,\, xs,\, ys \rrbracket(x)
\end{aligned}
$$

Table 6. Abstracting over a name in a process.

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 an Isabelle proof-script of about 20 lines of code.

4.4 Beta Expansion

Though seeming fully natural, β-expansion (EXP) has turned out to be the trickiest law to prove. The reason for this is two-fold: (1) Unlike in the proof of (EXT), we cannot directly apply induction, due to the existential quantification in the conclusion. Instead, we encode a primitively recursive translation-function $\llbracket\_\rrbracket$ abstracting over a name in a well-formed process. (2) This function has to compare all names with the name to be abstracted over, which works well for object-variables, but in a naive implementation, could accidentally replace every meta-variable with a conditional. As a result, every well-formed process with binders would be transformed into an exotic process-abstraction. For example, an abstraction $\mathbf{a}y.0$ over $\mathbf{a}$ would result in $\lambda x.\, x(\textbf{if } \mathbf{a} = y \textbf{ then } x \textbf{ else } y).0$.

The transformation We therefore propose a function coercing from higher-order to first-order syntax and back. The two lists, xs and ys, in $\llbracket P, xs, ys \rrbracket$ are computed prior to the transformation. List xs is the *transformation list* telling for every free name in P the names-abstraction it shall be mapped to during the transformation; except for the name to be abstracted over, it associates a constant function $\lambda x.\, \mathbf{a}$ with every free name $\mathbf{a}$ in P. List ys contains as many fresh names as are necessary to instantiate every meta variable in P; we compute it with the help of $db(P, \mathbf{c})$ (see Table 1) for some arbitrary name $\mathbf{c}$, and law (f2). The transformation intuitively 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, $fst\,(ys)$, and adds a pair $(fst\,(ys),\, (\lambda x.\, y))$ to xs, where y is the meta-variable given by the binder. When the transformation later encounters the instantiated (object-level) name, it thus 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.

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:

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

The proofs of the two theorems are tedious but purely technical inductions. The main difficulty was to formulate a suitable notion of abstraction over transformation-lists (see above), 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,

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

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

Again the proof of the second theorem is based on that of the first. In the proofs, we make extensive use of law (f6) from 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. Therefore, 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 $(a_1, b_1) \dots (a_n, b_n)$ it returns $a_1 \dots a_n$.

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

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

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 proof-script code.

4.5 General Evaluation and Further Work

In this section, we have formally derived the theory of contexts for a shallow embedding of the π-calculus in Isabelle/HOL, using structural induction as provided by a well-formedness predicate for processes and process-abstractions. A similar theory of contexts can be applied to every languages with binders for which free names play a role in the semantic analysis.

Applying (Ext) from the theory of contexts, we have further derived adequacy of the encoding in Isabelle/HOL [25] with respect to a first-order formalization of the π-calculus. An interesting fact is that it would have been hardly possible to derive adequacy without the previous establishment of the theory of contexts.

All proof-scripts referred to in this section, including adequacy, are available at `http://www7.in.tum.de/~roeckl/PI/syntax.shtml`.

5 Discussion

Why well-formed processes? Like for the replacement of names, there are two principal ways of ruling out exotic terms. [15,11] rely on λProlog and Coq to produce the set of well-formed processes. They hence delegate well-formedness to the meta-level of the provers. In our formalization, we do not rely on Isabelle to rule out exotic terms, but do this explicitely on the object-level by means of inductive predicates. As a consequence, we can mimic structural induction—which is generally non-existent in shallow embeddings—by rule-induction over the well-formedness predicates. This gives us a powerful tool for syntax analysis, which the formalizations in [15,11] do not possess. As a result, we can reason both *within* and *about* the π-calculus. Further, we are not restricted to specialized inductive frameworks for making (Ext) valid, but could adapt our framework to any general-purpose theorem-prover allowing for shallow embeddings.

What about other provers than Isabelle/HOL? In principle, our mechanization can be replayed on any theorem prover offering shallow embeddings, recursive functions, and inductive sets. There is no need that the framework rules out exotic terms automatically. Adapting our proofs to Coq would require modifying the notion of equality, by adding an extensionality axiom, and stating decidability of equality on names: both properties come indeed for free in Isabelle/HOL. In principle, one could then—and would have to, in that case (See Section 1 and [10])—add well-formedness predicates to the formalization presented in [11] and formally derive (Mon), (Ext), and (Exp).

In logical frameworks like λProlog [19] and Elf [23], 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 [24] implements a meta-logic based on Elf which offers a form of automated induction. Similarly, McDowell and Miller propose $FO\lambda^{\Delta\mathbb{N}}$ [13], a meta-logic where induction over natural numbers can be used to reason on a subset of simply typed λ-calculus. While it may be possible to adapt the results presented in this paper to such frameworks, it remains an open question how much support these systems can offer in semantic proofs about 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 used as an inductive logical framework, fully stripped of an object-logic. Still, to guarantee for the absence of exotic terms, it is necessary that the ability to compare names is only defined in `Prop` and not in `Set`. This rules out in particular any inductive type for names.

What about justifying the theory of contexts? This work is not a mechanized justification of the work of Honsell, Miculan, and Scagnetto [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 theory of contexts within a shallow embedding 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 principles for substitution and induction.

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 transition-system for a fragment of the π-calculus within a shallow embedding, which reduces the number of instantiations by modelling derivatives in terms of functions [3]. More precisely, she presents a shallow embedding of Milner's transition system for the π-calculus based on abstractions and concretions [16]. It will have to be investigated how this formalism can be extended to the full π-calculus, and whether a theory of contexts is also necessary to reason about semantics or not.

Is HOAS worth it? So far, work on higher-order abstract syntax for the π-calculus has focused on introducing the main concepts [15,11] or presenting fundamental applications [11,3]. The work at hand belongs to the first category, in that it provides a framework for reasoning within the π-calculus in a shallow embedding. It should not be regarded as a case-study for the application of HOAS to the π-calculus; we rather hope it might contribute to a language-independent method of syntax analysis in shallow embeddings, in terms of the general concepts and proofs described in the previous sections. The practical aspects of applying HOAS to the π-calculus will have to be further examined by large-scale case-studies based on frameworks like the ones presented in [11] and in this paper.

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. Aït-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. S. Gay. A framework for the formalisation of pi-calculus type-systems in Isabelle/HOL. Technical report, University of Glasgow, 2000.
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. R. McDowell and D. Miller. Reasoning with higher-order abstract syntax in a logical framework. *Transactions on Computational Logic*, 2000. to appear.
14. T. Melham. A mechanized theory of the π-calculus in HOL. *Nordic Journal of Computing*, 1(1):50–76, 1995.
15. D. Miller. Specification of the pi-calculus. available at `http://www.cse.psu.edu/~dale/lProlog/examples/pi-calculus/toc.html`.
16. R. Milner. Functions as processes. *Journal of Math. Struct. in Computer Science*, 17:119–141, 1992.
17. R. Milner. *Communicating and Mobile Processes.* Cambridge University Press, 1999.
18. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.
19. G. Nadathur and D. Miller. An overview of λprolog. In MIT Press, editor, *Proc. LPC'98*, pages 810–827, 1998.
20. L. C. Paulson. Isabelle's object-logics. Technical Report 286, University of Cambridge, Computer Laboratory, 1993.
21. L. C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In *Procs CADE'94*, volume 814 of *LNAI*, pages 148–161. Springer, 1994.
22. L. C. Paulson, editor. *Isabelle: a generic theorem prover*, volume 828 of *LNCS*. Springer, 1994.
23. F. Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Proc. LICS'89*, pages 313–321. IEEE, 1989.
24. 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.
25. C. Röckl. *On the Mechanized Validation of Infinite-State and Parameterized Reactive and Mobile Systems.* PhD thesis, Technische Universität München, 2001. Submitted.
26. 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.
27. D. Walker. Objects in the π-calculus. *Information and Computation*, 116:253–271, 1995.