JFP **13** (2): 415–451, March 2003. © 2003 Cambridge University Press
DOI: 10.1017/S0956796802004653 Printed in the United Kingdom

A fully adequate shallow embedding of the π-calculus in Isabelle/HOL with mechanized syntax analysis

CHRISTINE RÖCKL
LAMP–DI–EPFL, INR Ecublens, CH–1015 Lausanne, Switzerland
(*e-mail:* `christine.roeckl@epfl.ch`)

DANIEL HIRSCHKOFF
LIP – ENS Lyon, 46, allée d'Italie, F–69364 Lyon Cedex 7, France
(*e-mail:* `Daniel.Hirschkoff@ens-lyon.fr`)

Abstract

This paper discusses an application of the higher-order abstract syntax technique to general-purpose theorem proving, yielding shallow embeddings of the binders of formalized languages. Higher-order abstract syntax has been applied with success in specialized logical frameworks which satisfy a closed-world assumption. As more general environments (like Isabelle/HOL or Coq) do not support this closed-world assumption, higher-order abstract syntax may yield exotic terms, that is, datatypes may produce more terms than there should actually be in the language. The work at hand demonstrates how such exotic terms can be eliminated by means of a two-level well-formedness predicate, further preparing the ground for an implementation of structural induction in terms of rule induction, and hence providing fully-fledged syntax analysis. In order to apply and justify well-formedness predicates, the paper develops a proof technique based on a combination of instantiations and reabstractions of higher-order terms. As an application, syntactic principles like the theory of contexts (as introduced by Honsell, Miculan, and Scagnetto) are derived, and adequacy of the predicates is shown, both within a formalization of the π-calculus in Isabelle/HOL.

1 Introduction

General-purpose theorem provers like Isabelle/HOL (Paulson, 1994b) or Coq (Barras *et al.*, 2001), offer a wide range of sophisticated automatized proof-strategies which might be beneficial to reasoning within and about programming languages and calculi, once these have been formalized. It is often the case that the abstract syntax of a language or calculus involves binders. For their treatment various methodologies have been proposed, yet their application in practice needs to be further investigated. This paper reports on such an exercise, using higher-order abstract syntax and implementing bound variables on the meta-level of the prover, a technique that so far has obtained rather little attention in general-purpose theorem proving. We have developed our formalization in Isabelle/HOL, which is based on higher-order intuitionistic logic and offers higher-order logic as an object-logic, including formalizations of fixpoints, induction, coinduction, sets, lists, etc. The main motivation for choosing a general tool rather than a specialized logical framework like λProlog (Nadathur & Miller, 1988) or Twelf (Pfenning & Schürmann, 1999) was the broad range of available libraries as well as the automatic proof-support offered by the former. Indeed, although setting up the syntax of languages with binders is already a challenging task of its own, we believe that semantic reasoning based on such a formalization necessitates a degree of generality in libraries and automation of proofs currently only offered by general-purpose theorem provers such as Isabelle or Coq. Our preference for a shallow embedding over a deep one, thus employing the λ-calculus of the theorem prover to implement binders, was motivated by the experience that setting up α-conversion and β-reduction[1] by means of substitution is a time-consuming and error-prone task. As a result, we obtain a formalization that allows us to reason quite naturally about α-equivalence classes of terms. Of course, the greater generality of our proof-environment with respect to specialized logical frameworks does not come for free. There are four problems to be dealt with:

1. The structural induction principles automatically generated by Isabelle/HOL from datatype definitions are too weak for syntax analysis in a higher-order formalization. The natural structural induction principle for a datatype definition $T ::= \ldots \mid f : T_1 \to \ldots \to T_n \to T \mid \ldots$, where we have to assume that neither of the T_i contains T in a negative position, is of the following form: for a term t of type T, a statement $P(t)$ is justified provided that, for all constructors f from the datatype definition, $P(f(t_1, \ldots, t_n))$ can be inferred from the induction-hypothesis that $P(t_i)$ holds for every t_i of type T. As an example, consider such a constructor f of arity $n = 2$, where $T_1 = T$ and $T_2 = \textbf{int} \to T$. One then has to infer $P(f(t_1, t_2))$ from $P(t_1)$ only (as t_2 is not of type T), which is usually too weak; a corresponding hypothesis for t_2 would be necessary. Yet, how should it look like, and how can it be obtained?
2. The search for an adequate induction-principle is further complicated by the presence of *exotic terms*. Isabelle/HOL provides the axiom of choice together with functions that are polymorphic in their result type, such as conditionals. Such constructs can interfere with the datatype, which in turn gives rise to exotic terms. From terms t_1 and t_2 of our example datatype T above and standard conditional, for instance, we can construct an exotic term $f(t_1, \lambda x.\, \text{if } x = 0 \text{ then } t_1 \text{ else } t_2)$. Important syntactic principles are invalid in the presence of exotic terms, including structural induction and specific formulations of extensionality of equality (Hofmann, 1999; Honsell *et al.*, 2001b). We therefore have to find a means of ruling out exotic terms.
3. In the presence of exotic terms, adequacy of the formalization certainly becomes questionable. Having established a procedure separating well-formed terms from exotic ones, one still has to show that no more exotic terms are left (or at least not those doing harm), which intrinsically provides an adequacy proof.

[1] Building on the λ-calculus, we use *"β-reduction"* to refer in general to the mechanism used in the language to instantiate parameters by values.

4. Finally, in syntax analysis one often explicitly refers to bound variables, yet this is not possible in higher-order abstract syntax, where α-equivalence classes of terms are considered rather than simple terms. In order to perform syntax analysis nevertheless, one has to find suitable instantiations of the term classes.

Of course, these four problems are related. For instance, syntactic as well as suitable induction principles for higher-order abstract syntax can be axiomatized if there are no exotic terms (Honsell *et al.*, 2001a, 2001b); weak object-logics guarantee for the absence of exotic terms (Hofmann, 1999; Honsell *et al.*, 2001b; McDowell & Miller, 2001) but necessitate adequacy proofs outside the framework; and strong object-logics provide the power that is necessary for adequacy proofs, but produce exotic terms. So far, work on higher-order abstract syntax has mainly concentrated on logical frameworks that automatically exclude exotic terms (Honsell *et al.*, 2001b; McDowell & Miller, 2001; Schürmann, 2001), providing category-theoretical justifications of their adequacy. In this paper we follow an alternative approach, originally proposed by Despeyroux & Hirschowitz (1994) and Despeyroux *et al.* (1995). Building on a strong object-logic, we have to accept that exotic terms can be derived, but exclude them by means of an inductive well-formedness predicate. This predicate in turn allows us to perform induction over the syntax of well-formed processes and to derive an adequacy proof within the framework. To our knowledge, this is the first full-scale analysis of higher-order abstract syntax within a theorem prover using a two-level well-formedness predicate both deriving adequacy and the underlying theory of contexts ((Despeyroux & Hirschowitz, 1994) present an adequacy proof for an ω-chain of well-formedness predicates, which allows them to give a more succinct adequacy proof without using the theory of contexts, but which entails the use of lists of names.) We have chosen the π-calculus as a paradigmatic object-language; it is a non-trivial example exposing typical features of programming languages with binders, yet it is small enough to be tractable in a formalization like ours. Further it is a *nominal calculus*, that is, processes only appear in positive position in a higher-order abstract syntax. Higher-order languages such as the λ-calculus have to be nominalized before they can be treated similarly in Isabelle/HOL or Coq, for which there exists a standard approach introducing variables, see, for instance, Despeyroux & Hirschowitz (1994).

Contributions

We address the four above-mentioned questions as follows. We specify inductive *well-formedness predicates* on the object-level, which (1) allow us to mimic structural induction by rule induction and (2) rule out exotic terms. All this finally yields (3) an adequacy result stating that our well-formed terms correspond to the α-equivalence classes of a straightforward first-order abstract syntax. To compare our higher-order terms with their first-order counterparts, we (4) instantiate them with fresh parameters which we abstract over again later, to establish the original binding-structure. This coercion technique amounts to selecting a specific representative of the α-equivalence class of a process specified in higher-order abstract syntax. Probably the main contribution of this paper is the exploitation of rule induction and variable coercion to derive within our framework the *theory of contexts* for the π-calculus, which was originally introduced in Honsell *et al.* (2001b). Note that in the adequacy proof we make use of the theory of contexts, and apply essentially the same proof techniques that we have used for its derivation. Further note that we restrict our well-formedness predicates to two levels of abstraction, instead of specifying a chain of predicates. Induction proofs therefore require an elaborate use of fresh parameters, and adequacy is not straightforward. On the other hand, this approach saves us from manipulation of lists of parameters, and further induction over the length of these lists, which would at least significantly increase the size of the proofs, if not make them completely untractable. The material presented in this paper has been fully formalized in Isabelle/HOL. Significant parts of the proof scripts are given in the appendix, so to allow for a reconstruction of the formalization in Isabelle/HOL or related provers.

Related work

Higher-order abstract syntax has been approached from several viewpoints. A category-theoretical background is provided, for instance, in Fiore *et al.* (1999), Hofmann (1999) and Bucalo *et al.* (2001). Specialized logical frameworks are presented, for instance, in McDowell & Miller (2001), Pfenning (1989) and Pfenning & Schuermann (1998). In Honsell *et al.* (2001b), Coq is used as such a logical framework for a formalization of the π-calculus, and the theory of contexts is axiomatized, justified by a model-theoretic adequacy argument, further supported by Hofmann (1999) and Honsell *et al.* (2001a). The present paper presents an alternative proof of the theory of contexts, using well-formedness predicates as they were first proposed in Despeyroux *et al.* (1995). With respect to that work, we give a formal justification of the fact that two-level well-formedness predicates are enough to provide adequacy, and we derive induction schemes for them. As this paper touches diverse areas of research, more pointers to related work are given along the text where appropriate. Parts of the material discussed in this paper have been presented in Röckl *et al.* (2001).

Overview

The paper is organized as follows. Section 2 introduces the basic principles of first-order and higher-order abstract syntax, and relates the two approaches to deep and shallow embeddings in general-purpose theorem provers. Further, some of the necessary features of Isabelle/HOL are described. Section 3 presents the π-calculus along with two formalizations of its syntax in Isabelle/HOL, one in a very straightforward deep embedding and the other in a shallow embedding. Sticking to representations of the π-calculus on paper as closely as possible, the deep embedding serves as a reference for proving the adequacy of the shallow embedding. In section 4 we derive the theory of contexts for our shallow embedding, making extensive use of induction over the well-formed terms and of coercion of bound variables. Section 5 completes the analysis of our shallow embedding by presenting the formalization of an adequacy proof. Section 6 concludes the paper and discusses related topics.

For the reader interested in the Isabelle code of the definitions and proof scripts, the appendix should provide enough material to reconstruct the formalization. Appendix A introduces a specification of parameters, requiring a suitable set to be countably infinite yet leaving open unnecessary details. Appendices B and C present both deep and shallow embeddings; the mechanization of the theory of contexts and the adequacy result are given in Appendix D and E.

2 Formalizing languages with binders

In this section we introduce some terminology from language specification and theorem proving that we will use throughout the paper. In particular, we refer to two taxonomies, one from language specification distinguishing between *first-order abstract syntax* and *higher-order abstract syntax*, and the other from theorem proving providing the terms *deep embedding* and *shallow embedding*. Readers familiar with the terminology and Isabelle/HOL can skip sections 2.1–2.3. Whereas the notions of first-order and higher-order abstract syntax are clearly separable, there are varying degrees of depth and shallowness of embeddings in a theorem prover. These may result in a controversial application of terminology. Formalizing a language using a recursive datatype to represent its syntax is usually referred to as a deep embedding. However, the identification of certain object-variables with meta-variables to implement binders, introduces a form of shallowness into the formalization that clearly distinguishes it from a purely deep embedding. Note that there certainly exist encoding strategies providing higher degrees of shallowness than the way we proceed here, yet we would like to emphasize the fact that such an embedding is indeed shallow for the following reasons: first, a formalization of a higher-order abstract syntax does not necessarily yield a shallow embedding of the binders, as demonstrated by Gordon & Melham (1996) and Gay (2001) (we shall return to this below); second, a shallow embedding of the binding constructs of a language entails specific problems that result precisely from this shallowness.

Historical remark

The term *higher-order abstract syntax* was introduced in Pfenning & Elliot (1988), with its basic principles going back to Church (1940), and since then has mostly been used in the context of logical frameworks like λProlog (Miller, 1991; Miller, 1990; McDowell & Miller, 2001) or Twelf (Pfenning, 1996; Pfenning, 1999; Pfenning & Schürmann, 1999), but also in Coq (Despeyroux & Hirschowitz, 1994; Despeyroux *et al.*, 1995; Despeyroux, 2000; Honsell *et al.*, 2001b).

The distinction between *deep* and *shallow embeddings* was first made in Boulton *et al.* (1992), discussing the formalization of hardware description languages in HOL. There the term shallow is used to describe an embedding of the semantics of the object-language, using the syntactic infrastructure from the underlying meta-level instead of specifying a syntax in terms of a recursive datatype. Since then the term has been used to describe various levels of access to the meta-level, for instance to implement the binding character of quantifiers. The shallow representation of binding as we use it here is discussed, for instance, in Melham (1995).

Table 1. *Some formalizations of the π-calculus in theorem provers and logical frameworks*

	Deep embedding	Shallow embedding of binders
First-order	(Melham, 1995; Aït-Mohamed, 1996; Hirschkoff, 1997; Henry-Gréard, 1999; Röckl, 2001)	—
Higher-order	(Gordon & Melham, 1996; Gay, 2001; Gillard, 2001)	(Miller, 1992; Honsell *et al.*, 1998; Despeyroux, 2000; Honsell *et al.*, 2001b; Miller, 2001; Röckl *et al.*, 2001)

A note on literature

Table 1 gives an overview of formalizations of the π-calculus, specifying datatypes either in first-order abstract syntax or in higher-order abstract syntax. Combining this distinction with the decision to implement bound names in terms of object-variables or meta-variables, this yields either a purely deep embedding or a shallower one. At present we are not aware of any formalization of the π-calculus using a higher degree of shallowness than for the implementation of binders.

2.1 Meta-level and object-level

It is common lore that logical reasoning involves a semantic *meta-logic* within which one argues about a syntactic *object-logic*. To illustrate this distinction, let us consider the following examples:

1. In a model-theoretic argument that Church's Simple Theory of Types is a suitable framework for implementing languages with binders, as is done in Honsell *et al.* (2001), model theory is the meta-logic used to reason about the Simple Theory of Types as an object-logic.
2. It is well possible to reason about higher-order classical logic within higher-order classical logic, where of course one has to make a clear distinction between the meta-logic in which the constructs really have a meaning, and the purely syntactic object-logic.
3. Theorem provers implement a meta-logic which users apply when formalizing an object-logic. Isabelle, for instance, provides higher-order intuitionistic logic, whereas Coq is based on the Calculus of (Co)Inductive Constructions; λProlog is based on higher-order hereditary Harrop formulas, and Twelf implements LF. The implementors of Isabelle further provide a variety of object-logics, such as higher-order logic (Isabelle/HOL) or Zermelo–Fraenkel set-theory (Isabelle/ZF), that can be extended by the user.

Analogously, we speak of a *meta-level* and an *object-level*, the former consisting of the meta-logic, and the latter referring to the object-logic. There are two kinds of variables, two kinds of constants, two kinds of quantifiers, etc. In order to emphasize the distinction, we use $x, y, \dots$ to refer to meta-level variables (*meta-variables* in short), and $\mathbf{x}, \mathbf{y}, \dots$ for object-level variables (or *object-variables*).

Hierarchies

There does not necessarily have to be one single meta-level and one single object-level, but there can be a hierarchy of levels $l_1, \dots, l_n$, where each l_i is an object-level with respect to l_{i-1} and a meta-level with respect to l_{i+1}. Level l_1 represents a meta-level that can be considered as the basic level of reasoning. Let us make this more concrete in terms of our above examples:

1. Honsell *et al.* (2001b) formalize the π-calculus in the calculus of inductive constructions, using the latter as a meta-logic for the former. The above-mentioned model-theoretic justification of the adequacy of the formalization entails a three-level hierarchy (if one identifies the meta-level provided by the Coq system with the Calculus of (Co)Inductive Constructions as treated on paper in Honsell *et al.* (2001a).
2. In this paper, we stay within a two-level hierarchy, using higher-order intuitionistic logic (as provided by Isabelle) as a meta-level and extending higher-order logic on the object-level with suitable constant definitions for the π-calculus.
3. This allows us to formalize in a natural way our complete argument in Isabelle/HOL, sticking to the prover's meta- and object-level. Of course, it is also possible to use theorem provers in a hierarchical way. For example, Gordon & Melham (1996) specify a λ-calculus in the theorem prover HOL, and promote this object-logic as a meta-logic for specifying languages. Gay (2001) follows the same idea in a formalization of the π-calculus in Isabelle/HOL.

2.2 First-order and higher-order abstract syntax, deep and shallow embeddings

When specifying language constructs in a logic, one always has the choice to remain fully on the object-level or to make use of facilities from the meta-level, for instance, its type-system or the λ-calculus it implements. This gives rise to two taxonomies, one relating *first-order* and *higher-order abstract syntax*, and the other dealing with *deep* and *shallow embeddings*.

First-order and higher-order abstract syntax

Recursive datatype definitions of languages with binders can either be given a first-order abstract syntax, such as $T' ::= \dots \mid f : T' \to \mathbf{int} \to T' \to T' \mid \dots$ for our example from the introduction. Here only first-order functions into T' are used. Hence there is no distinction on a syntactic level between binding and non-binding operators. Using higher-order abstract syntax, on the other hand, binders bear higher-order functions, such as $T ::= \dots \mid f : T \to (\mathbf{int} \to T) \to T \mid \dots$ in our example. The main difference is that in a first-order abstract syntax, α-conversion and β-reduction are treated on the object-level, whereas in a higher-order abstract syntax, they are delegated to the meta-level.

Deep and shallow embeddings

Theorem provers provide meta-logics, upon which users can specify their object-logics. Theories remaining fully on the object-level are called *deep embeddings*. *Shallowness* appears whenever some aspects of the theory being formalized are left outside the object-level, possibly through an application of meta-level constructs. For instance, binders can be implemented by identifying certain object-variables with meta-variables (that is, variables bound by the abstraction operator of the meta-logic). Or, meta-level types are used to express types from the object-language. Note that there are various degrees of shallowness. In this paper, we implement the binders of the π-calculus in a shallow way by identifying the object-variables with meta-variables, but otherwise formulate the calculus as a datatype. Shallowness usually results in a higher degree of automation, yet on the other hand disallows direct reasoning about the constructs being manipulated.

Adequacy

Neither for higher-order language specifications nor for shallow embeddings, it is necessarily clear that the formalization precisely corresponds to the object-language. Depending on the axioms and constants of the object-logic, we will see in Section 3.4 that a higher-order abstract syntax can give rise to exotic terms, see also Hofmann (1999), Honsell *et al.* (2001b) and Schuermann (2001). And the more shallow an embedding is, the less clear is its connection with the original language. This *adequacy* problem has to be adressed by establishing the existence of a bijection between the object-language and its formalization. This can either be done in a more abstract way, for instance using a category theoretical argument as is done in Hofmann (1999), or in a more specific way by establishing transformation functions, as we shall do in section 5.

2.3 Isabelle/HOL: an overview

Isabelle is a generic theorem prover, and as such is based on a small kernel implementing higher-order intuitionistic logic as a meta-logic (Paulson, 1994b). On top of that, the theorem prover offers a variety of object-logics, ranging from first-order logic (Isabelle/FOL) and higher-order logic (Isabelle/HOL) to the calculus of constructions (Isabelle/CC) or Zermelo-Fraenkel set-theory (Isabelle/ZF). For a complete overview of the object-logics, see Paulson (1993) and the Isabelle homepage at `http://isabelle.in.tum.de/`.

Proofs in Isabelle are based on unification, and are usually conducted in a backward chaining 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 original 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* heuristically apply the basic tactics to prove given subgoals. 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-analysis. 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 (Berghofer & Wenzel, 1999; Paulson, 1994a). A recent extension of Isabelle/HOL allows function types in datatype definitions to contain strictly positive occurrences of the type being defined (Berghofer & Wenzel, 1999). This allows for formalizations of programming languages using HOAS in a shallow embedding, like the one we discuss in 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 The π-calculus

The π-calculus (Milner 1992, 1999) is a name-passing language based on CCS (Milner, 1989), in which both communication-channels and messages sent along these channels belong to a single type $\mathcal{N}$, called *names*. This identification of channels and values implies that a name received in a communication can be used as a channel in a subsequent communication. As an example, consider the processes $!(ay.\bar{y}c.0)$ and $\bar{a}x.xz.P$. The former represents a very simple procedure that when called along channel a with argument y emits a constant name c along the received name. The replication-operator ! (called "bang") denotes that there is an unlimited number of copies of the process. The latter process is a client asking the procedure to return its result along a name x. Put in parallel, the two processes can engage in the following two communication steps, each of which is marked with the *invisible action* τ (a judgment of the form $P \xrightarrow{\mu} P'$ means that P can evolve to P' by performing some action μ – here τ is a special action standing for an internal communication):

$$!(ay.\bar{y}c.0) \mid \bar{a}x.xz.P \quad \xrightarrow{\tau} \quad !(ay.\bar{y}c.0) \mid \bar{x}c.0 \mid xz.P \quad \xrightarrow{\tau} \quad !(ay.\bar{y}c.0) \mid 0 \mid P\{c/z\}.$$

To ensure that the result is indeed returned to the caller – and not to some interfering process – the π-calculus allows the client to create a local name by means of a *restriction* $(\nu x)(\bar{a}x.xz.P)$, and then share it with the procedure as a *private* means of communication:

$$\begin{aligned} !(ay.\bar{y}c.0) \mid (\nu x)(\bar{a}x.xz.P) \quad &\xrightarrow{\tau} \quad (\nu x)(!(ay.\bar{y}c.0) \mid \bar{x}c.0 \mid xz.P) \\ &\xrightarrow{\tau} \quad (\nu x)(!(ay.\bar{y}c.0) \mid 0 \mid P\{c/z\}). \end{aligned}$$

Table 2. *Free and bound names of processes*

$fn(0) \overset{def}{=} \emptyset$	$bn(0) \overset{def}{=} \emptyset$
$fn(\tau.P,\ !P) \overset{def}{=} fn(P)$	$bn(\tau.P,\ !P) \overset{def}{=} bn(P)$
$fn(\bar{a}b.P,\ [a=b]P,\ [a \neq b]P) \overset{def}{=} \{a,b\} \cup fn(P)$	$bn(\bar{a}b.P,\ [a=b]P,\ [a \neq b]P) \overset{def}{=} bn(P)$
$fn(ax.P) \overset{def}{=} \{a\} \cup (fn(P) - \{x\})$	$bn(ax.P) \overset{def}{=} bn(P) \cup \{x\}$
$fn((\nu x)P) \overset{def}{=} fn(P) - \{x\}$	$bn((\nu x)P) \overset{def}{=} bn(P) \cup \{x\}$
$fn(P+Q,\ P \mid Q) \overset{def}{=} fn(P) \cup fn(Q)$	$bn(P+Q\ P \mid Q) \overset{def}{=} bn(P) \cup bn(Q)$

During the first communication, the scope of the local variable x is extended from the client to the procedure. This *mobility* – i.e. the fact that the binding structure of a process can change during its execution – is a paramount feature of the π-calculus, which allows it to describe higher-order processes (Thomsen, 1990; Sangiorgi, 1992; Amadio, 1993; Sangiorgi, 1996), functions (Milner, 1992; Sangiorgi, 1996), and object-oriented and imperative languages (Walker, 1995; Kleist & Sangiorgi, 1998; Röckl & Sangiorgi, 1999). Further, the π-calculus has served as a basis for concurrent programming (Turner, 1995).

Names and processes

Let $\mathscr{N}$ be a countably infinite set of names. We consider a *monadic* π-calculus, with *output* $\bar{a}b$, *input* ax, and *silent* τ prefixes. Name a in $\bar{a}b$ and ax is usually referred to as the *subject*, b and x are referred to as *object* of a prefix. Processes are built from *inaction* 0 by applying either of the prefixes $\pi.P$, *restriction* $(\nu x)P$, *choice* $P+Q$, *parallel composition* $P \mid Q$, *matching* $[a=b]P$ and *mismatching* $[a \neq b]P$ of names a and b, and *replication* $!P$:

$$P \ ::= \ 0 \mid \tau.P \mid \bar{a}b.P \mid ax.P \mid (\nu x)P \mid P+Q \mid P \mid Q \mid [a=b]P \mid [a \neq b]P \mid !P.$$

Input $ax.P$ and restriction $(\nu x)P$ are the two binders of the π-calculus. As the semantic analysis of processes is often based on the interplay between these two operators, they obviously have a great impact also on a treatment of syntax.

Free and bound names

We use the common notions of *free*, *bound* and *fresh* names. In the π-calculus, a name is bound if it is in the scope of an input-prefix or a restriction, otherwise it is free. A name is *fresh* for a process if it does not occur among its free (and bound) names[2]. Free and bound names can be determined by primitive recursive functions fn and bn, which for the π-calculus are given in Table 2. For our example procedure and client, we obtain free names $fn(!(ay.\bar{y}c.0)) = \{a,c\}$ and $fn((\nu x)(\bar{a}x.xz.P)) = \{a\} \cup (fn(P) - \{x,z\})$, and bound names $bn(!(ay.\bar{y}c.0)) = \{y\}$ and $bn((\nu x)(\bar{a}x.xz.P)) = bn(P) \cup \{x,z\}$.

[2] It is often a matter of taste whether freshness forbids bound names as well. Reasoning in a first-order syntax is often easier if bound names are also excluded. In higher-order syntax, where α-equivalence classes of processes are considered, freshness can only refer to free names. For more information, see sections 2, 3.3 and 3.4.

In the π-calculus, this clear distinction between free and bound names is essential. Consider again the procedure $!ay.\bar{y}c.0$ and an instance $(\nu x)(\bar{a}x.xz.\bar{z}b.0)$ of the client, engaging in the following execution steps:

$$\begin{aligned} !ay.\bar{y}c.0 \mid (\nu x)(\bar{a}x.xz.\bar{z}b.0) \xrightarrow{\tau} \dots &\xrightarrow{\tau} (\nu x)(!ay.\bar{y}c.0 \mid 0 \mid \bar{c}b.0) \\ &\xrightarrow{\bar{c}b} (\nu x)(!ay.\bar{y}c.0 \mid 0 \mid 0). \end{aligned}$$

So far, we have tacitly assumed that $x \neq c$ on a syntactic level. In fact, derivability of the visible output-step relies on the fact that $x \neq c$. This emphasizes the *static* binding-policy of the π-calculus,where α-conversion takes care that previously free names do not become bound by a restriction. In contrast, CHOCS (Thomsen, 1990) is based on *dynamic* binding, where the client can also send c to the procedure, despite the restriction.

Semantics

In the examples above, we have applied a *labelled transition semantics* as introduced in Milner *et al.* (1992). For the π-calculus, *open*, *late* and *early* semantics can be distinguished, which are characterized by the order of the quantifiers in the definitions of bisimulation-based observational equivalences; see Quaglia (1999) for a good overview. Often, *reduction semantics* are presented instead of labelled transition semantics. There, only silent transitions are considered as (reduction) steps, and processes are identified modulo a structural congruence.

3.1 Formalizing the π-calculus

We now embark on our exercise of deriving a fully adequate shallow embedding of the π-calculus in Isabelle/HOL. As pointed out in sections 1 and 2, this is not a straightforward task, because with the theorem prover providing an open world, exotic terms necessarily arise. To eliminate them and simultaneously obtain sufficiently powerful induction-principles, we introduce a well-formedness predicate based on ideas of Despeyroux *et al.* (1995). Then, to be able to prove within Isabelle/HOL that our shallow embedding is fully adequate, we further formalize the π-calculus in a deep embedding, introducing basic notions of substitution and deriving parts of a theory of α-conversion, but only as far as this is necessary to support the adequacy-proof later on. Section 3.3 presents the very straightforward deep embedding, and section 3.4 describes the shallow embedding.

3.2 Formalizing names

Following standard conventions, we have chosen the set $\mathscr{N}$ of names to be at least countably infinite. This is necessary to be able to pick a fresh name at will, or even a set of fresh names, a technique which we make extensive use of in the proofs in sections 4 and 5. Also the theory of contexts itself focuses around fresh names, as will be seen in section 4. In the formalization, we do not restrict ourselves to a specific type but use an axiomatic type-class *inf_class* comprising all types $\mathscr{T}$ for which there exists an injection from $\mathbb{N}$ into $\mathcal{T}$. We neither require nor forbid the existence of a surjection, see also our discussion in section 6. We then can derive that for each type that can be proven to belong to this class, the following holds: given a finite set of elements of such a type, there always exists an element (alternatively, a set or list of elements) not in that set. As the set of free – and that of bound names, if it can be computed – of a process is always finite, this result allows us to pick fresh names whenever necessary. See Appendix A for the Isabelle code.

A note on notation

In this and the following sections, we use $\mathbf{a}, \mathbf{b}, \mathbf{x}, \mathbf{y}, \ldots$ to range over object-level names (with respect to the processes) bound by universal quantifiers[3], and $a, b, x, y \ldots$ to denote meta-level names. In our deep embedding, both free and bound names are represented on the object-level. In our shallow embedding, free names are represented by object-variables and bound names by meta-variables (see also section 2). Further we use f_a and ff_a to denote *name-abstractions*, that is, functions mapping one, respectively two, names to names. *Process-abstractions*, that is, functions from names to processes, are referred to by f_P and ff_P, respectively.

3.3 A deep embedding

In a *deep embedding*, processes are formalized fully on the object-level, including input-prefix and restriction. This yields a straightforward correspondence with the syntax presented at the beginning of section 3:

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

The structural induction-principle computed automatically by Isabelle/HOL is fully viable as a basis for syntax analysis. Finding a suitable notion of substitution is generally a hard task when giving a first-order syntax for a language with binders, and has been the topic of various investigations (deBruijn, 1972; Gabbay & Pitts, 1999; McKinna & Pollack, 1993). As our first-order syntax principally serves the purpose of showing that our higher-order syntax is fully adequate, we stick to a very straightforward formulation of substitution, allowing us to define α-equivalence and α-renaming to normalize processes. Normalization maps terms to a specified representative of their α-equivalence classes. Appendix B presents major parts of the Isabelle code.

[3] Note, however, that these are meta-variables, because object-level quantifers in Isabelle/HOL are formalized in a shallow way.

Free, bound and fresh names

With the deep embedding being a one-to-one translation into an Isabelle datatype of the π-calculus processes introduced in Section 3, the functions from Table 2 computing *free* and *bound* names can be used without modification in the formalization. We use $n(P) = fn(P) \cup bn(P)$ to refer to the complete set of names occurring in P. We consider a name as *fresh* in a process if it occurs neither among its free nor bound names.

Counting binders

To determine whether two processes are α-equivalent, one has to choose a fresh name at each point a binder is encountered along a path in the process tree; in different paths, names can of course be reused. In order to supply sufficiently many fresh names, one has therefore to compute the maximal number of binders along the paths in the process tree. We do this in terms of a straightforward primitively recursive function db_d. For each binder that is encountered, the current value for the subtree is incremented; when two subtrees are joined by choice or parallel composition, the maximum of the two depths is chosen.

Substitution

Following the most straightforward approach, we use a simple substitution that does not perform α-conversion, and prevent name-capture by applying the side-condition that all substitutes be fresh. Our substitution is based on a conditional rewriting of names, following

$$\mathbf{a}\{\mathbf{c}/\mathbf{d}\} \overset{def}{=} \text{if } \mathbf{a} = \mathbf{d} \text{ then } \mathbf{c} \text{ else } \mathbf{a},$$

and is extended recursively for processes. We write $P\{\mathbf{c}/\mathbf{d}\}$ to denote that $\mathbf{c}$ is substituted for $\mathbf{d}$ in P. The definition follows the standard lines, for instance,

$$\begin{aligned}(\bar{\mathbf{a}}\mathbf{b}.P)\{\mathbf{c}/\mathbf{d}\} &\overset{def}{=} \overline{\mathbf{a}\{\mathbf{c}/\mathbf{d}\}}\,\mathbf{b}\{\mathbf{c}/\mathbf{d}\}.P\{\mathbf{c}/\mathbf{d}\}\\ (\mathbf{a}\mathbf{b}.P)\{\mathbf{c}/\mathbf{d}\} &\overset{def}{=} \text{if } \mathbf{b} = \mathbf{d} \text{ then } \mathbf{a}\{\mathbf{c}/\mathbf{d}\}\mathbf{b}.P \text{ else } \mathbf{a}\{\mathbf{c}/\mathbf{d}\}\mathbf{b}.P\{\mathbf{c}/\mathbf{d}\}.\end{aligned}$$

To illustrate the problem of name-capture, consider $\mathbf{ab}.\bar{\mathbf{a}}\mathbf{c}.0$ with $\mathbf{c} \notin \{\mathbf{a}, \mathbf{b}\}$, and suppose that $\mathbf{c}$ is to be replaced by $\mathbf{b}$. Hence, we obtain,

$$\begin{aligned}(\mathbf{ab}.\bar{\mathbf{a}}\mathbf{c}.0)\{\mathbf{b}/\mathbf{c}\} &= \mathbf{ab}.(\bar{\mathbf{a}}\mathbf{c}.0)\{\mathbf{b}/\mathbf{c}\} && \text{because } \mathbf{c} \notin \{\mathbf{a}, \mathbf{b}\}\\ &= \mathbf{ab}.\bar{\mathbf{a}}\mathbf{b}.0\{\mathbf{b}/\mathbf{c}\} && \text{because } \mathbf{c} \neq \mathbf{a}\\ &= \mathbf{ab}.\bar{\mathbf{a}}\mathbf{b}.0,\end{aligned}$$

so that after the transformation, the emitted name $\mathbf{b}$ is bound by the input prefix. Nevertheless, this definition can be used to formulate α-equivalence, α-conversion, and normalization, and hence can serve as a basis for a "proper" notion of substitution. In this paper, we restrict our attention to α-equivalence and normal-forms, because they are sufficient for the adequacy-proof presented in section 5. We write $P\{\mathbf{c}_1/\mathbf{d}_1, \ldots, \mathbf{c}_n/\mathbf{d}_n\}$ for a sequence of substitutions $P\{\mathbf{c}_n/\mathbf{d}_n\}\ldots\{\mathbf{c}_1/\mathbf{d}_1\}$; or simply $P\{xs\}$ for $xs = \{\mathbf{c}_1/\mathbf{d}_1, \ldots \mathbf{c}_n/\mathbf{d}_n\}$.

α-equivalence

We specify two notions of α-equivalence, one very straightforward formalization (written $=_\alpha$), and an implementation of it (written $=_\alpha^{ys}$) that uses lists (such as ys) of fresh names to be instantiated for names underneath binders. For input, the two formalizations yield the following rules:

$$\frac{P\{\mathbf{c}/\mathbf{b}\} =_\alpha P'\{\mathbf{c}/\mathbf{b}'\} \qquad \mathbf{c} \notin n(P) - \{\mathbf{b}\} \qquad \mathbf{c} \notin n(P') - \{\mathbf{b}'\}}{\mathbf{ab}.P =_\alpha \mathbf{ab}'.P'}\,\alpha_3$$

$$\frac{P\{hd(ys)/\mathbf{b}\} =_\alpha^{tl(ys)} P'\{hd(ys)/\mathbf{b}'\}}{\mathbf{ab}.P =_\alpha^{ys} \mathbf{ab}'.P'}\,\alpha_3'$$

It can be shown that the latter implements the former provided ys provides sufficiently many distinct fresh names (the predicate **nodups** captures the fact that a list does not have multiple occurrences of the same name):

$$\frac{P =_\alpha^{ys} P' \qquad db_d(P) \leqslant |ys| \qquad \mathbf{nodups}(ys) \qquad ys \cap (n(P) \cup n(P')) = \emptyset}{P =_\alpha P'}$$

The proof proceeds by rule induction over $P =_\alpha^{ys} P'$. It is a straightforward case-analysis using monotonicity properties of free and bound names with respect to substitutions. In particular, we apply that $n(P\{\mathbf{c}/\mathbf{d}\}) \subseteq \{\mathbf{c}\} \cup n(P)$, which follows by induction on P. The proof has been formalized in Isabelle/HOL, and the proof script consists of about 50 lines of code (see Appendix B).

Normalization

Like for α-equivalence, we specify a very straightforward notion of normalization ($nm(\_,\_)$) as well as an implementation of it ($nm_m(\_,\_)$). By normalization we mean here the process of fixing a canonical (though not unique) representation of first-order terms modulo α-equivalence. We will use the function in the adequacy-proof to map processes to specific representatives of their α-equivalence classes determined by the list of fresh names with which we instantiate bound variables. As both normalization functions depend on our notion of substitution defined above, they only yield correct results if the lists of names they use contain sufficiently many distinct fresh names. As an example, consider the rules for input:

$$\begin{aligned} nm(\mathbf{ab}.P, ys) &\overset{def}{=} \mathbf{a}\, hd(ys).nm(P\{hd(ys)/\mathbf{b}\}, tl(ys)) \\ nm_m(\mathbf{ab}.P, xs, ys) &\overset{def}{=} nm_m(\mathbf{a}, xs)\, hd(ys).nm_m(P, (hd(ys), \mathbf{b})xs, tl(ys)) \end{aligned}$$

By structural induction, we can derive that for a suitable list ys, the normalized process $nm(P, ys)$ coincides with its counterpart $nm_m(P, [], ys)$, and that both are α-equivalent to the original process P:

$$\frac{db_d(P) \leqslant |ys| \qquad \mathbf{nodups}(ys) \qquad ys \cap n(P) = \emptyset}{nm(P, ys) =_\alpha P}$$

$$\frac{db_d(P) \leqslant |ys| \qquad \mathbf{nodups}(ys) \qquad n(P) \cap ys = \emptyset}{nm(P, ys) = nm_m(P, [], ys)}$$

In fact, in both cases, stronger results have to be proved. The difficult part actually

Table 3. *Computing the free names fn_s and depth of binders db_s of a process*

$fn_s(0) = \emptyset$	$db_s(0, \mathbf{c}) = 0$
$fn_s(\tau.P) = fn_s(P)$	$db_s(\tau.P, \mathbf{c}) = db_s(P, \mathbf{c})$
$fn_s(\bar{\mathbf{a}}\mathbf{b}.P) = \{\mathbf{a}, \mathbf{b}\} \cup fn_s(P)$	$db_s(\bar{\mathbf{a}}\mathbf{b}.P, \mathbf{c}) = db_s(P, \mathbf{c})$
$fn_s(\mathbf{a}x.f_P(x)) = \{\mathbf{a}\} \cup fna_s(f_P)$	$db_s(\mathbf{a}x.f_P(x), \mathbf{c}) = 1 + dba_s(f_P, \mathbf{c})$
$fn_s((\nu\, x)f_P(x)) = fna_s(f_P)$	$db_s((\nu\, x)f_P(x), \mathbf{c}) = 1 + dba_s(f_P, \mathbf{c})$
$fn_s(P + Q) = fn_s(P) \cup fn_s(Q)$	$db_s(P + Q, \mathbf{c}) = max(db_s(P, \mathbf{c}), db_s(Q, \mathbf{c}))$
$fn_s(P \mid Q) = fn_s(P) \cup fn_s(Q)$	$db_s(P \mid Q, \mathbf{c}) = max(db_s(P, \mathbf{c}), db_s(Q, \mathbf{c}))$
$fn_s([\mathbf{a} = \mathbf{b}]P) = \{\mathbf{a}, \mathbf{b}\} \cup fn_s(P)$	$db_s([\mathbf{a} = \mathbf{b}]P, \mathbf{c}) = db_s(P, \mathbf{c})$
$fn_s([\mathbf{a} \neq \mathbf{b}]P) = \{\mathbf{a}, \mathbf{b}\} \cup fn_s(P)$	$db_s([\mathbf{a} \neq \mathbf{b}]P, \mathbf{c}) = db_s(P, \mathbf{c})$
$fn_s(!P) = fn_s(P)$	$db_s(!P, \mathbf{c}) = db_s(P, \mathbf{c})$
$fna_s(f_P) \overset{def}{=} \{\, \mathbf{a} \mid \forall \mathbf{b}.\, \mathbf{a} \in fn_s(f_P(\mathbf{b}))\,\}$	$dba_s(f_P, \mathbf{c}) \overset{def}{=} db_s(f_P(\mathbf{c}), \mathbf{c})$
$fnaa_s(ff_P) \overset{def}{=} \{\, \mathbf{a} \mid \forall \mathbf{b}.\, \mathbf{a} \in fna_s(\lambda x.\, ff_P(\mathbf{b}, x))\,\}$	

was to find a suitable strengthenings; the proofs themselves are derived by tedious but straightforward case-analyses:

$$\forall\, xs, ys.\, (db_d(P) \leqslant |ys| \;\wedge\; \mathbf{nodups}(ys) \;\wedge\; ys \cap (n(P) \cup xs) = \emptyset$$
$$\longrightarrow\; nm(P\{xs\}, ys) =_\alpha P\{xs\}).$$
$$\forall xs, ys.\, (db_d(P) \leqslant |ys| \;\wedge\; \mathbf{nodups}(snd(xs)) \;\wedge\; \mathbf{nodups}(ys) \;\wedge\; n(P) \cap (fst(xs) \cup ys) = \emptyset$$
$$\longrightarrow\; nm(P\{xs\}, ys) = nm_m(P, xs, ys)).$$

3.4 A shallow embedding

In our *shallow embedding*, input and restriction make use of *process-abstractions* f_P, that is, functions from names to processes. A corresponding datatype definition can be directly given in Isabelle/HOL, similarly to the specifications used in Despeyroux (2000) and Honsell *et al.* (2001b):

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

However, as pointed out in the Introduction and section 2, the structural induction-principle automatically generated by the prover does not suffice for syntax analysis, and exotic terms can be derived. As typical π-calculus examples, consider the following process-abstractions:

$$f_E \;\overset{def}{=}\; \lambda(x : names).\;\; \text{if}\;\; x = \mathbf{a}\;\; \text{then}\;\; 0\;\; \text{else}\;\; \mathbf{a}y.0,$$
$$f_W \;\overset{def}{=}\; \lambda(x : names).\;\; \mathbf{a}y.0.$$

The term f_E is exotic, because it is built from an object-level conditional (and *not* from a π-calculus conditional, which we represent in terms of matching and mismatching), whereas f_W can be considered as *valid*, or *well-formed*. For the Isabelle sources, see Appendix C.

Table 4. *Well-formed processes*

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

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

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

Free and fresh names, depth of binders

As pointed out in section 2, we are dealing with α-equivalence classes of processes rather than with specific terms. As a consequence, we can compute the set of free names by applying a closure to process abstractions, but we cannot compute bound names. For instance, to compute the free names of an input-prefix $\mathbf{a}x.f_P(x)$, we have to determine those of the process-abstraction f_P first, which we do by considering all possible instantiations: $fna_s\,(f_P) \overset{def}{=} \{\,\mathbf{a} \mid \forall\mathbf{b}.\mathbf{a} \in fn_s\,(f_P(\mathbf{b}))\,\}$. Universal quantification is necessary to eliminate those names that were merely used to instantiate meta-variables during the computation. As an example, consider the process $\mathbf{a}y.\bar{\mathbf{a}}y.0$. Instantiations of $\lambda y.\,\bar{\mathbf{a}}y.0$ with some $\mathbf{b}$ and $\mathbf{c}$ yield $fn_s\,(\bar{\mathbf{a}}\mathbf{b}.0) = \{\mathbf{a},\mathbf{b}\}$ and $fn_s\,(\bar{\mathbf{a}}\mathbf{c}.0) = \{\mathbf{a},\mathbf{c}\}$. The intersection over all such sets clearly preserves the actual set of free names $\{\mathbf{a}\}$. Hence, we obtain $fn_s\,(\mathbf{a}y.\bar{\mathbf{a}}y.0) = \{\mathbf{a}\}$. Table 3 presents the functions used to compute the free names of a process, as well as its depth of binders. Again, the latter will be necessary to determine the amount of fresh names needed for analysing a given process.

Well-formedness

Tables 4, 5 and 6 specify the well-formedness predicates which we apply to rule out exotic terms like f_E, and simultaneously to mimic structural induction by rule induction. Following ideas of Despeyroux *et al.* (195), we use a two-level predicate applying a closure via universal quantification to process-abstractions with binders. As an example, consider the rule for input-prefix:

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

We will see in section 5 that well-formed processes indeed represent a fully adequate encoding of the π-calculus. The reason is that the closure we specify for input-prefix and restriction defines a natural transformation on the domain of those process-abstractions corresponding to terms in a first-order syntax.

4 Deriving the theory of contexts

The *theory of contexts* by Honsell *et al.* (2001b) presents three syntactic properties that are necessary for the semantic analysis of the π-calculus, and has been proved on paper by Hofmann (1999) and Honsell *et al.* (2001a). For the π-calculus, it yields the following intuitive description:

Table 5. *Well-formed process-abstractions*

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

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

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

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

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

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

Table 6. *Well-formed names-abstractions*

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

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

(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 their instantiations with a fresh name* **a** *are equal.*

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

A formal representation involving the well-formedness predicates from the previous section is given in Table 7. Indeed, extensionality of contexts (Ext) only holds for well-formed process-abstractions; applying (Ext) to exotic terms like f_E from section 3.4 yields an inconsistency, because $f_E(\mathbf{b}) = f_W(\mathbf{b})$ for all $\mathbf{b} \neq \mathbf{a}$ yet $f_E(\mathbf{a}) \neq f_W(\mathbf{a})$. The necessity of the well-formedness predicates in (Ext) further yields that (Exp) has to be strengthened by restricting it to well-formed processes as well.

4.1 Proof techniques

In the following, we derive the theory of contexts for the π-calculus within our formalization in Isabelle/HOL. For excerpts of the code see Appendix D. We mimic the missing structural induction-principles in terms of rule induction over the well-formedness predicates. Binders are treated differently on the two levels. In order to prove a property P of well-formed processes $\mathbf{a}x.f_P(x)$ and $(\nu x)f_P(x)$, we refer to the corresponding property P' for well-formed process abstractions f_P; whereas in order to prove P' of well-formed process-abstractions $\lambda y.\,f_a(y)x.f_P(y,x)$ and $\lambda y.\,(\nu x)f_P(y,x)$, we apply the closure given by the universal quantification in rules $\mathbf{W}^a_3$ and $\mathbf{W}^a_4$. Fresh names play a vital role both in the statement of the theory of

Table 7. *Formalizations of* monotonicity, extensionality, *and* β-expansion

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

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

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

contexts as well as in our proof thereof. We use fresh names to uniquely instantiate process-abstractions underneath binders, so that we are able to reabstract over them after analysis. We apply this proof technique extensively in the proof of (Exp), where we use a transformation function which, like the normalization function presented in section 3.3, works by picking up fresh names from a list. Indeed, instantiating a higher-order process with fresh names amounts to converting it to a representative of the α-equivalence class determined by it. Note that (Exp) cannot be proved immediately in our framework, because the existential quantifier in its conclusion yields an induction-hypothesis that is too weak in the case of well-formed process abstractions with binders, where induction is closed by the two universal quantifiers. For this reason, we have to work with a specific transformation function.

4.2 Free and fresh names

The following basic results will be used in the proofs of (Ext) and (Exp) in order to create fresh names, so we summarize them here. 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** (*finite* is a predicate capturing the finiteness of a set). (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.

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

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

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

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

4.3 Monotonicity

The monotonicity law, see (Mon) in Table 7, is implicitely encoded in our formalization. That is, a name **a** is only free in a process-abstraction f_P according to *fnaa*$_s$, 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 7.

4.4 Extensionality

We prove (EXT) by induction over one of the two involved well-formed processes, f_P, and using case-analysis for the other, f_Q, see Appendix D for the Isabelle/HOL proof scripts. Except for input and restriction, the resulting case-analysis is purely technical. For the two binders, induction yields the following subgoal:

(*1st induction hypothesis*)

$$\begin{array}{l}\forall \mathbf{b}, f_Q, \mathbf{a}.\, \mathbf{wfpa}\,(f_Q) \wedge \mathbf{fresha}\,(\mathbf{a}, \lambda x.\, ff_P\,(\mathbf{b}, x)) \wedge \mathbf{fresha}\,(\mathbf{a}, f_Q) \wedge \\ \qquad ff_P\,(\mathbf{b}, \mathbf{a}) = f_Q\,(\mathbf{a}) \longrightarrow \lambda x.\, ff_P(\mathbf{b}, x) = \lambda x.\, f_Q(x)\end{array}$$

(*2nd induction hypothesis*)

$$\begin{array}{l}\forall \mathbf{b}, f_Q, \mathbf{a}.\, \mathbf{wfpa}\,(f_Q) \wedge \mathbf{fresha}\,(\mathbf{a}, \lambda x.\, ff_P\,(x, \mathbf{b})) \wedge \mathbf{fresha}\,(\mathbf{a}, f_Q) \wedge \\ \qquad ff_P\,(\mathbf{a}, \mathbf{b}) = f_Q\,(\mathbf{a}) \longrightarrow \lambda x.\, ff_P(x, \mathbf{b}) = \lambda x.\, f_Q(x)\end{array}$$

$$\begin{array}{ll}\forall \mathbf{b}.\, \mathbf{wfpa}\,(\lambda x.\, ff_P\,(\mathbf{b}, x)) & \forall \mathbf{b}.\, \mathbf{wfpa}\,(\lambda x.\, ff_P\,(x, \mathbf{b})) \\ \forall \mathbf{b}.\, \mathbf{wfpa}\,(\lambda x.\, ff_Q\,(\mathbf{b}, x)) & \forall \mathbf{b}.\, \mathbf{wfpa}\,(\lambda x.\, ff_Q\,(x, \mathbf{b}))\end{array}$$

$$\frac{\mathbf{freshaa}\,(\mathbf{a}, ff_P) \qquad \mathbf{freshaa}\,(\mathbf{a}, ff_Q) \qquad \lambda x.\, ff_P\,(x, \mathbf{a}) \;=\; \lambda x.\, ff_Q\,(x, \mathbf{a})}{\lambda x.\, ff_P\,(x, \mathbf{c}) \;=\; ff_Q\,(x, \mathbf{c})}$$

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 4.2 allow us to choose a name **d** which does not occur in $\{\mathbf{a}, \mathbf{c}\} \cup fnaa_s\,(ff_P) \cup fnaa_s\,(ff_Q)$. 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 conditions for the implication can be derived directly from the premises, or from (f7) and the fact that $\mathbf{d} \neq \mathbf{a}$, this yields a new premise of the form $\lambda x.\, ff_P\,(\mathbf{d}, x) = \lambda x.\, ff_Q(\mathbf{d}, x)$. Then, instantiating the second arguments of ff_P and ff_Q with **c** in the second induction-hypothesis, we obtain,

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

The conditions of 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 proof script code, see Appendix D for more details.

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, see Appendix D.

Table 8. *Abstracting over a name in a process*

$$\begin{aligned}
[\![\mathbf{a}, []]\!] &= \lambda x.\, x \\
[\![\mathbf{a}, (\mathbf{b}, f_a)xs]\!] &= \mathit{if}\ \mathbf{a} = \mathbf{b}\ \mathit{then}\ f_a\ \mathit{else}\ [\![\mathbf{a}, xs]\!] \\[1ex]
[\![0,\ xs,\ ys]\!] &= \lambda x.\, 0 \\
[\![\tau.P,\ xs,\ ys]\!] &= \lambda x.\, \tau.[\![P,\ xs,\ ys]\!] \\
[\![\bar{\mathbf{a}}\mathbf{b}.P,\ xs,\ ys]\!] &= \lambda x.\, \overline{[\![\mathbf{a},\ xs]\!](x)}[\![\mathbf{b},\ xs]\!](x).[\![P,\ xs,\ ys]\!](x) \\
[\![\mathbf{a}y.f_P(y),\ xs,\ ys]\!] &= \lambda x.\, [\![\mathbf{a},\ xs]\!](x)y.\, [\![f_P(\mathit{fst}\,(ys)),\ (\mathit{fst}\,(ys),\ (\lambda x.\ y))xs,\ \mathit{tl}(ys)]\!](x) \\
[\![(\nu\ y)f_P(y),\ xs,\ ys]\!] &= \lambda x.\, (\nu\ y)\ [\![f_P(\mathit{fst}\,(ys)),\ (\mathit{fst}\,(ys),\ (\lambda x.\ y))xs,\ \mathit{tl}(ys)]\!](x) \\
[\![P + Q,\ xs,\ ys]\!] &= \lambda x.\, [\![P,\ xs,\ ys]\!](x) + [\![Q,\ xs,\ ys]\!](x) \\
[\![P \parallel Q,\ xs,\ ys]\!] &= \lambda x.\, [\![P,\ xs,\ ys]\!](x) \parallel [\![Q,\ xs,\ ys]\!](x) \\
[\![[\mathbf{a} = \mathbf{b}]P,\ xs,\ ys]\!] &= \lambda x.\, [[\![\mathbf{a},\ xs]\!](x) = [\![\mathbf{b},\ xs]\!](x)]\,[\![P,\ xs,\ ys]\!](x) \\
[\![[\mathbf{a} \neq \mathbf{b}]P,\ xs,\ ys]\!] &= \lambda x.\, [[\![\mathbf{a},\ xs]\!](x) \neq [\![\mathbf{b},\ xs]\!](x)]\,[\![P,\ xs,\ ys]\!](x) \\
[\![!P,\ xs,\ ys]\!] &= \lambda x.\, ![\![P,\ xs,\ ys]\!](x)
\end{aligned}$$

4.5 Beta-expansion

Though seemingly fully natural, β-expansion (Exp) has turned out to be the trickiest law to prove. The reason for this is twofold: (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 $[\![\_]\!]$ abstracting over a name in a well-formed process. (2) This function necessarily compares all names with the name to be abstracted over, which works well for object-variables, but in a naive implementation would unintentionally 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 $\lambda x.\, x(\mathbf{if}\ \mathbf{a} = y\ \mathbf{then}\ x\ \mathbf{else}\ y).0$.

The transformation We therefore propose a function translating from higher-order to first-order syntax and back. The two lists, xs and ys, in $[\![P, xs, ys]\!]$ 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 in 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 (so that free names are eventually mapped to themselves). List ys contains as many fresh names as are necessary to instantiate every meta variable in P; we compute it with the help of $db_s(P, \mathbf{c})$ (see Table 3 in section 3.4) for some arbitrary name $\mathbf{c}$, and law (f2). The transformation intuitively proceeds as follows (refer to Table 8 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 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. For the Isabelle/HOL code, see Appendix D.

Well-formedness We call an abstraction over a transformation list well-formed if it only applies well-formed names-abstractions (see Table 6 for a definition):

$$\frac{}{\mathbf{wftrl}\,(\lambda x.\ [])}\ \mathbf{W}^t_1 \qquad \frac{\mathbf{wfnaa}\,(ff_a) \quad \mathbf{wftrl}\,(f_{xs})}{\mathbf{wftrl}\,(\lambda x.\ (\mathbf{a}, ff_a(x))f_{xs}(x))}\ \mathbf{W}^t_2$$

The following two derived results show that the transformation described above produces well-formed process-abstractions when applied to well-formed processes:

$$\frac{\mathbf{wfpa}\,(f_P) \quad \mathbf{wftrl}\,(f_{xs})}{\mathbf{wfpa}\,([\![f_P(\mathbf{c}), f_{xs}(\mathbf{d}), ys]\!]) \ \wedge\ \mathbf{wfpa}\,(\lambda x.\ [\![f_P(\mathbf{c}), f_{xs}(x), ys]\!](\mathbf{b}))}$$

$$\frac{\mathbf{wfp}\,(P) \quad \forall(\mathbf{a}, f_a) \in xs.\ \mathbf{wfna}\,(f_a)}{\mathbf{wfpa}\,([\![P, xs, ys]\!])}$$

The proofs of these two theorems are tedious but purely technical inductions. The main difficulty is to formulate a suitable notion of abstraction over transformation-lists (see above). Note that the second theorem is actually a corollary of the first one.

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

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

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

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 4.2.

Equality It remains to show by induction that a reinstantiation of a transformation yields the original process again. The proofs make use of the monotonicity and extensionality theorems proved in sections 4.3 and 4.4, as well as of the well-formedness and freshness results from the previous two sections. For this reason, we have to guarantee by db_s that ys contains at least as many names as there are nested binders in a process. The predicate **nodups** ensures 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 $a_1 \ldots a_n$.

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

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

Table 9. *From first-order to higher-order syntax and back*

$$\begin{array}{rlrl}
[\![0, xs]\!]_{d\to s} & \overset{def}{=} 0 & [\![0, ys]\!]_{s\to d} & \overset{def}{=} 0 \\
[\![\tau.P, xs]\!]_{d\to s} & \overset{def}{=} \tau.[\![P, xs]\!]_{d\to s} & [\![\tau.P, ys]\!]_{s\to d} & \overset{def}{=} \tau.[\![P, ys]\!]_{s\to d} \\
[\![\bar{\mathbf{a}}\mathbf{b}.P, xs]\!]_{d\to s} & \overset{def}{=} \overline{[\![\mathbf{a}, xs]\!]_{d\to s}}[\![\mathbf{b}, xs]\!]_{d\to s}.[\![P, xs]\!]_{d\to s} & [\![\bar{\mathbf{a}}\mathbf{b}.P, ys]\!]_{s\to d} & \overset{def}{=} \bar{\mathbf{a}}\mathbf{b}.[\![P, ys]\!]_{s\to d} \\
[\![\mathbf{ab}.P, xs]\!]_{d\to s} & \overset{def}{=} [\![\mathbf{a}, xs]\!]_{d\to s}b.[\![P, [(b, \mathbf{b})]xs]\!]_{d\to s} & [\![\mathbf{ab}.P, ys]\!]_{s\to d} & \overset{def}{=} \mathbf{a}hd(ys).[\![P, tl(ys)]\!]_{s\to d} \\
[\![(\nu\mathbf{b})P, xs]\!]_{d\to s} & \overset{def}{=} (\nu b)[\![P, [(b, \mathbf{b})]xs]\!]_{d\to s} & [\![(\nu b)P, ys]\!]_{s\to d} & \overset{def}{=} (\nu hd(ys))[\![P, tl(ys)]\!]_{s\to d} \\
[\![P+Q, xs]\!]_{d\to s} & \overset{def}{=} [\![P, xs]\!]_{d\to s} + [\![Q, xs]\!]_{d\to s} & [\![P+Q, ys]\!]_{s\to d} & \overset{def}{=} [\![P, ys]\!]_{s\to d} + [\![Q, ys]\!]_{s\to d} \\
[\![P \mid Q, xs]\!]_{d\to s} & \overset{def}{=} [\![P, xs]\!]_{d\to s} \mid [\![Q, xs]\!]_{d\to s} & [\![P \mid Q, ys]\!]_{s\to d} & \overset{def}{=} [\![P, ys]\!]_{s\to d} \mid [\![Q, ys]\!]_{s\to d} \\
[\![[\mathbf{a}=\mathbf{b}]P, xs]\!]_{d\to s} & \overset{def}{=} [[\![\mathbf{a}, xs]\!]_{d\to s} = [\![\mathbf{b}, xs]\!]_{d\to s}]\,[\![P, xs]\!]_{d\to s} & [\![[\mathbf{a}=\mathbf{b}]P, ys]\!]_{s\to d} & \overset{def}{=} [\mathbf{a}=\mathbf{b}]\,[\![P, ys]\!]_{s\to d} \\
[\![[\mathbf{a}\neq\mathbf{b}]P, xs]\!]_{d\to s} & \overset{def}{=} [[\![\mathbf{a}, xs]\!]_{d\to s} \neq [\![\mathbf{b}, xs]\!]_{d\to s}]\,[\![P, xs]\!]_{d\to s} & [\![[\mathbf{a}\neq\mathbf{b}]P, ys]\!]_{s\to d} & \overset{def}{=} [\mathbf{a}\neq\mathbf{b}]\,[\![P, ys]\!]_{s\to d} \\
[\![!P, xs]\!]_{d\to s} & \overset{def}{=} ![\![P, xs]\!]_{d\to s} & [\![!P, ys]\!]_{s\to d} & \overset{def}{=} ![\![P, ys]\!]_{s\to d}
\end{array}$$

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, see Appendix D for the Isabelle/HOL code. Material from this section has been presented in Röckl *et al.* (2001).

5 Adequacy

In this section, we present a mechanized proof that our shallow embedding is *fully adequate*, that is, that the well-formed processes exactly define α-equivalence classes of processes in a first-order syntax. A similar result is obtained in Despeyroux & Hirschowitz (1994) for two encodings of the λ-calculus, using a more straightforward but tedious to formalize ω-chain of well-formedness predicates. Despeyroux *et al.* (1995) only sketch a justification of the result we obtain here, namely that well-formedness predicates that "stop" at the level of functions taking two arguments are sufficient to obtain adequacy. An overview of the Isabelle/HOL sources is given in Appendix E.

Intuitively, adequacy of two syntaxes means that for every term in one syntax there exists a corresponding term in the other, and vice versa. This can be shown by exhibiting *encoding* and *decoding* functions $[\![\_, \_]\!]_{d\to s}$ from the deep into the shallow embedding and $[\![\_, \_]\!]_{s\to d}$ from the shallow into the deep embedding, and proving that $[\![\_, \_]\!]_{d\to s}$ and $[\![\_, \_]\!]_{s\to d}$ are reverse functions. Table 9 introduces the two functions. Like in the transformation function from the previous section, one has to take care that in the decoding-function meta-names are not involved in comparisons. Again, we therefore apply a transformation-list xs rather than substitution. As a consequence, one can say that $[\![\_, \_]\!]_{s\to d}$ and $[\![\_, \_]\!]_{d\to s}$ are separated representations of the instantiating and the reabstracting parts of $[\![\_, \_, \_]\!]$ from section 4.5.

5.1 From first-order to higher-order syntax

The function $[\![\_, \_]\!]_{d\to s}$ translates processes from the deep embedding into well-formed processes in the shallow embedding. It uses an auxiliary list xs, telling for a process P in $[\![P, xs]\!]_{d\to s}$ how its free names should be transformed. Whenever the function encounters an input or a restriction, the corresponding bound name is added to xs together with a new meta-variable, and bound by Isabelle's functional mechanism (see Table 9 for a complete definition):

$$[\![\mathbf{ab}.P, xs]\!]_{d\to s} \overset{def}{=} [\![\mathbf{a}, xs]\!]_{d\to s} b.[\![P, [(b,\mathbf{b})]xs]\!]_{d\to s} \quad [\![(\nu\mathbf{b})P, xs]\!]_{d\to s} \overset{def}{=} (\nu b)[\![P, [(b,\mathbf{b})]xs]\!]_{d\to s}$$

The encoding of names themselves is rather straightforward: if $\mathbf{a}$ does not occur in xs, it is left unchanged; otherwise, it is mapped to the name accompanying its *first* occurrence, since in non-normalized processes, a name $\mathbf{b}$ can occur under several binders. To illustrate this, consider the process $\mathbf{ab}.\bar{\mathbf{a}}\mathbf{b}.(\nu\mathbf{b})\bar{\mathbf{a}}\mathbf{b}.0$. In it, the $\mathbf{b}$ from the first output is bound by the input, whereas that in the second output is bound by the restriction. This yields the following encoding:

$$\begin{aligned}
[\![\mathbf{ab}.\bar{\mathbf{a}}\mathbf{b}.(\nu\mathbf{b})\bar{\mathbf{a}}\mathbf{b}.0, []]\!]_{d\to s} &= [\![\mathbf{a}, []]\!]_{d\to s} b.[\![\bar{\mathbf{a}}\mathbf{b}.(\nu\mathbf{b})\bar{\mathbf{a}}\mathbf{b}.0, [(b,\mathbf{b})]]\!]_{d\to s} && \text{new meta-name } b\\
&= \mathbf{a}b.\overline{[\![\mathbf{a}, [(b,\mathbf{b})]]\!]_{d\to s}}[\![\mathbf{b}, [(b,\mathbf{b})]]\!]_{d\to s}.[\![(\nu\mathbf{b})\bar{\mathbf{a}}\mathbf{b}.0, [(b,\mathbf{b})]]\!]_{d\to s}\\
&= \mathbf{a}b.\bar{\mathbf{a}}b.(\nu b')[\![\bar{\mathbf{a}}\mathbf{b}.0, [(b',\mathbf{b}), (b,\mathbf{b})]]\!]_{d\to s} && \text{new meta-name } b'\\
&= \mathbf{a}b.\bar{\mathbf{a}}b.(\nu b')\overline{[\![\mathbf{a}, [(b',\mathbf{b}), (b,\mathbf{b})]]\!]_{d\to s}}\\
&\quad\ [\![\mathbf{b}, [(b',\mathbf{b}), (b,\mathbf{b})]]\!]_{d\to s}.[\![0, [(b',\mathbf{b}), (b,\mathbf{b})]]\!]_{d\to s}\\
&= \mathbf{a}b.\bar{\mathbf{a}}b.(\nu b')\bar{\mathbf{a}}b'.0
\end{aligned}$$

Note that the meta-names b and b' are chosen by the theorem prover, and cannot be manipulated by the user. For instance, the user cannot tell whether they are equal to or distinct from some other name, be it on the object-level or on the meta-level.

Well-formedness We can show that every encoding of a deeply embedded process is well-formed:

$$\frac{}{\mathbf{wfp}\,([\![P, xs]\!]_{d\to s})} \qquad\qquad \frac{}{\mathbf{wfp}\,([\![P, []]\!]_{d\to s})}$$

The proof proceeds by induction on the structure of P (in this case, we can indeed apply structural induction, as the generated principle for first-order syntax naturally suffices). With xs replacing object-level names with meta-level names, we have to reason with abstractions f_{xs} over transformation-lists xs. We say that such an f_{xs} is well-formed, if it maps object-level names to well-formed name-abstractions. This can be described by the following inductive predicate:

$$\frac{}{\mathbf{wfnml}\,(\lambda x.\,[])}\,\mathbf{Wfnml1} \qquad\qquad \frac{\mathbf{wfna}\, f_{\mathbf{a}} \quad \mathbf{wfnml}\,(xs)}{\mathbf{wfnml}\,(\lambda x.\,(f_{\mathbf{a}},\mathbf{a})xs)}\,\mathbf{Wfnml2}$$

We show by structural induction on P that for every list f_{xs} with $\mathbf{wfnml}\,(f_{xs})$, the encoding $\lambda x.\,[\![P, f_{xs}x]\!]_{d\to s}$ is a well-formed process abstraction. The proof script consists of less than ten lines, featuring mostly calls to Isabelle's automatic tactic `auto_tac`.

5.2 From higher-order to first-order syntax

The function $[\![\_,\_]\!]_{s\to d}$ translates higher-order processes into representatives of α-equivalence classes of first-order processes. Note that not any arbitrary first-order process can be generated by our function, because it takes the names it uses to instantiate process-abstractions from a list ys. The process $\mathbf{ax}.0 \,|\, \mathbf{by}.0$, for instance, cannot be computed, however the α-equivalent process $\mathbf{ax}.0 \mid \mathbf{bx}.0$ is derivable from $\mathbf{a}x.0 \mid \mathbf{b}y.0$. See Table 9 for an overview, and Appendix E for the Isabelle/HOL code. For the higher-order process $\mathbf{a}b.\bar{\mathbf{a}}b.(\nu b')\bar{\mathbf{a}}b'.0$ resulting from our previous example, consider the resulting higher-order process and a list $[\mathbf{b}, \mathbf{b}']$ of object-level names, we obtain an α-equivalent first-order decoding:

$$\begin{aligned}
[\![\mathbf{a}b.\bar{\mathbf{a}}b.(\nu b')\bar{\mathbf{a}}b'.0, [(\mathbf{b}, \mathbf{b}')]]\!]_{s\to d} &= \mathbf{ab}.[\![\bar{\mathbf{a}}\mathbf{b}.(\nu b')\bar{\mathbf{a}}b'.0, [\mathbf{b}']]\!]_{s\to d} && \text{instantiate } b \text{ with } \mathbf{b}\\
&= \mathbf{ab}.\bar{\mathbf{a}}\mathbf{b}.[\![(\nu b')\bar{\mathbf{a}}b'.0, [\mathbf{b}']]\!]_{s\to d}\\
&= \mathbf{ab}.\bar{\mathbf{a}}\mathbf{b}.(\nu \mathbf{b}')[\![\bar{\mathbf{a}}\mathbf{b}'.0, []]\!]_{s\to d} && \text{instantiate } b' \text{ with } \mathbf{b}'\\
&= \mathbf{ab}.\bar{\mathbf{a}}\mathbf{b}.(\nu \mathbf{b}')\bar{\mathbf{a}}\mathbf{b}'.0
\end{aligned}$$

5.3 Proving Adequacy

Adequacy falls into two parts: (1) whenever a process P from the deep embedding is translated into a shallow process and back, the result is α-equivalent to P, and (2) whenever a well-formed process P from the shallow embedding is translated into a deep process and back, the resulting process is syntactically equal to P. In our formalization, we thus have to prove the following theorems:

$$\frac{db_d(P) \leqslant |ys| \qquad n(P) \cap ys = \emptyset \qquad \mathbf{nodups}(ys)}{[\![[\![P, []]\!]_{d\to s}, ys]\!]_{s\to d} =_\alpha P}$$

$$\frac{\mathbf{wfp}\,(P) \qquad db_s(P)\mathbf{c} \leqslant n \qquad fn_s(P) \cap ys = \emptyset \qquad \mathbf{nodups}(ys)}{[\![[\![P, ys]\!]_{s\to d}, []]\!]_{d\to s} = P}$$

The proofs follow a scheme which is similar to that of β-expansion in section 4.5. We are going to sketch it in the sequel. Excerpts of the Isabelle/HOL proof scripts can be found in Appendix E.

Deep embedding yields deep embedding We prove by induction on the structure of P that for arbitrary xs and ys we have $[\![[\![P, xs]\!]_{d\to s}, ys]\!]_{s\to d} = nm_m(P, xs, ys)$. Then, we use the normalization result from section 3.3 to conclude that for a suitable ys, as assumed in the premises, $[\![[\![P, []]\!]_{d\to s}, ys]\!]_{s\to d} = nm(P, ys)$. Referring to the theorem from section 3.3 relating normalization and α-equivalence, we can then infer that $[\![[\![P, []]\!]_{d\to s}, ys]\!]_{s\to d} =_\alpha P$.

Shallow embedding yields shallow embedding The proof proceeds by the usual two separate structural inductions, one on the well-formedness predicates for process-abstractions, and the other on that for well-formed processes. It makes use of extensionality of contexts (Ext) from section 4.4, so that additional lemmas about the freshness of names are necessary; these are proved by induction on well-formed process abstractions. Otherwise, that is, with usual Leibnitz-equality, the proof would be unmanageable.

6 Discussion

Table 1 in section 2 presents three general ways of specifying the datatype of a language with binders in a theorem prover, determined by suitable combinations of first-order and higher-order abstract syntax with deep and shallow embeddings. This paper deals in particular with higher-order abstract syntax using datatype definitions but a shallow embedding of binders. Within the general-purpose theorem prover Isabelle/HOL we have demonstrated how this approach can be directly related to a straightforward first-order abstract syntax, by establishing translation functions between the two syntaxes. To achieve this, we have developed techniques for syntax analysis in a shallow embedding, mimicking structural induction by rule induction over a well-formedness predicate, and instantiating bound variables with fresh parameters and reabstracting over them. A major result obtained using these techniques is a mechanized proof of the theory of contexts for the π-calculus. We believe that the proof techniques presented in this paper are applicable to other shallow datatype specifications as well.

6.1 First-order and higher-order abstract syntax

The main conceptual difference between first-order and higher-order abstract syntax is that the former yields all the terms of a language whereas the latter specifies α-equivalence classes. As a consequence, first-order datatypes necessitate a notion of substitution to express α-conversion and β-reduction, which are "for free" in higher-order abstract syntax. Formalizing substitutions for a language, especially if it has a lot of operators, is a tedious and error-prone task. As for the π-calculus, a closer look on the literature reveals that the analyses of transitions, bisimulations, and modal logic tacitly assume α-equivalence classes of processes, hence higher-order abstract syntax seems to offer a good support here. Investigations on basic syntactic aspects of programming languages, on the other hand, frequently focus on a theory of α-conversion, which requires explicit reasoning about terms, hence the use of first-order abstract syntax. Various approaches have been proposed to facilitate the setup of abstract syntax (deBruijn, 1972; Gabbay & Pitts, 1999; McKinna & Pollack, 1993). We shall not discuss these further in this work. The search for (as well as the thorough study of) alternatives remains an area of ongoing research. For an overview of existing techniques used in formalizations of the π-calculus, see Gay (2001) and Röckl (2001).

6.2 Higher-order abstract syntax: deep and shallow embeddings

Shallow embeddings of higher-order abstract syntax, like the one studied in this paper, use the functional mechanism of the meta-level of the theorem prover in order to represent binders. Higher-order abstract syntax in a deep embedding, as studied in Gordon & Melham (1996) and applied in Gay (2001) and Gillard (2001), adds an intermediate (pseudo) meta-level to be used as a logical framework for the higher-order abstract syntax specified on top of it. The main advantage of the latter approach with respect to a pure first-order syntax (see Table 1 in section 2) is that $\alpha\beta\eta$-theory has to be derived only for the small intermediate language. As an aside, this approach allows for formalized proofs about the logical framework itself. Usually such proofs are rather conducted on paper. Gordon & Melham

(1996) studied the untyped λ-calculus as a logical framework in HOL. It would be interesting to see how their methodology carries over to analyses of other logical frameworks, especially those underlying existing proof-tools like λProlog (Nadathur & Miller, 1988) or Twelf (Pfenning, 1996).

The principal advantage of that approach over a shallow embedding is that an appropriately chosen logical framework naturally excludes exotic terms. On the other hand, bound variables are still represented on the object-level of the prover, so α-conversions and β-reductions still have to be manipulated by the user, even though on the level of the underlying intermediate logic.

6.3 Logical frameworks and general-purpose theorem provers

Logical frameworks such as λProlog (Nadathur & Miller, 1988) or Twelf (Pfenning, 1996) allow the user to specify higher-order abstract syntax in a *closed world* where exotic terms cannot arise. Miller (1990, 2000, 2001) and McDowell & Miller (1997) study syntactic abstractions as underlying principle, and Despeyroux *et al.* (1997), Pfenning & Schürmann (1998, 1999), Schürmann (2001) and Leleu (1998) examine meta-logics for higher-order abstract syntax in dependently typed settings. Gordon & Melham (1996) present a fully formalized analysis of the untyped lambda-calculus as logical framework. Note that Coq (Barras *et al.*, 1999) can be used as a logical framework as well, as has been demonstrated by Honsell *et al.* (2001b) for the π-calculus. General-purpose theorem provers like Isabelle/HOL or standard extensions of Coq usually provide an *open world*, in which the axiom of choice (or elimination principles in Coq) and constants from the environment may yield exotic terms. Hofmann (1999) and Schürmann (2001) discuss the boundary determining whether exotic terms arise or not. A recent study (Dowek *et al.*, 2001) proposes an extension of first-order logic with binders. This might yield a novel perspective for the development of logics directly supporting the representation of binders.

In this paper, we have dealt with the open world of Isabelle/HOL and have eliminated exotic terms by means of well-formedness predicates, based on ideas from Despeyroux *et al.* (1995) and Despeyroux & Hirschowitz (1994). The use of well-formedness predicates yields *parametrised judgements*, that is, theorems have to be suitably annotated with well-formedness assumptions. This is not necessary in a logical framework. On the other hand, general-purpose theorem provers offer greater support concerning further proofs, in terms of libraries and automatic tactics.

One drawback of tools like Isabelle/HOL or Coq with respect to λProlog or Twelf is that only base types can be abstracted over in a shallow way, hence for the formalization of languages with higher-order binders (like the λ-calculus), a nominalized representation has to be chosen. Yet, such a nominalization can be achieved using standard techniques.

Coq as a logical framework

The use of Coq as a logical framework is based on the fact that datatypes can be specified directly on top of its meta-logic, the calculus of inductive constructions, so no further object-logic is necessary. As pointed out by Hofmann (1999) and Honsell *et al.* (2001a), however, one has to take care not to specify parameters as a recursive type, because this would lead to exotic terms. Note that this rules out the use of natural numbers as names, a choice usually considered natural for the π-calculus. The reason is that, for recursive types, it is possible to tell in Coq's `Set` kind whether two elements are equal or not. As datatype definitions are usually located in `Set`, such a distinction can be exploited to construct a conditional within a function underneath a binder, as used in the introduction to derive exotic terms. However, Coq offers `Prop` as a kind in which to derive proofs. Here a distinction does not cause any problem, and can be exploited for the encoding of transition systems.

6.4 *Exotic terms and well-formedness predicates*

The use of well-formedness predicates to rule out exotic terms entails parametrised judgements, which in turn require some effort from the user to show that a given term is indeed well-formed. The extent of this additional effort in proofs about semantics will have to be further investigated. To hide the predicates in transition rules, for instance, one can use axiomatic type-classes requiring well-formedness implicitly. This, however, does not necessarily reduce the number of proof tasks, as now it has to be shown for each term that it belongs to the type class, that is, that it is well-formed.

Despeyroux *et al.* (1995) and Despeyroux & Hirschowitz (1994) propose two different kinds of well-formedness predicates. In Despeyroux & Hirschowitz (1994), the authors study a well-founded set of predicates, ordered by the number of arguments a function takes. Hence there is a predicate for each level determined by this ordering, implemented for the λ-calculus in Coq in terms of finite lists of arguments. Using the predicates, they are able to specify rule induction principles that mimic structural induction. Being purely syntax-directed, such a chain of predicates naturally corresponds to a family of process contexts one can obtain in a logical framework (Miller, 2001; Honsell *et al.*, 2001a). Later, Despeyroux *et al.* (1995) defined a two-level well-formedness predicate which does not necessitate the use of lists in a formalization, but the authors do not further elaborate on it. In this work, we have shown that this approach indeed yields adequacy, and have derived induction and other syntactic proof-principles which are based on suitable instantiations with fresh names. Such an instantiation amounts to choosing a suitable representative from an α-equivalence class of terms. On a logical level, the use of a two-level predicate is justified by the fact that the universal quantification specifying **wfpaa** in terms of $\forall b.\, \mathbf{wfpa}\,(\lambda x.\, ff_P\,(b,x))$ and $\forall b.\, \mathbf{wfpa}\,(\lambda x.\, ff_P\,(x,b))$ entails a natural transformation. The exact correspondence with the theories presented elsewhere (Fiore *et al.*, 1999; Fiore & Turi, 2001; Hofmann, 1999; Honsell *et al.*, 2001a; McDowell & Miller, 2001; Schürmann, 2001), however, remains to be fully investigated.

Acknowledgements

We thank Stefan Berghofer, Joëlle Despeyroux, Gilles Dowek, Javier Esparza, René Lalement, Tobias Nipkow, and Peter Rossmanith for helpful comments and discussions. We are further indebted to the anonymous referees for providing detailed comments, which we believe have helped us a lot in relating the different approaches towards higher-order abstract syntax.

This work has been supported by the PROCOPE project 9723064, "Verification Techniques for Higher-Order Imperative Concurrent Languages". Most of the formalization was accomplished while the first author was employed at the Technische Universität München.

A Names

We do not fix a set of names *a priori*, but require it to be at least countably infinite, achieving this by an *axiomatic type-class* `inf_class` requiring the existence of an injection from `nat` into the types in it.

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

This allows us to pick a names that is distinct from another name, list of names, or finite set of names:

```
Goal "[| finite (A::(('a::inf_class) set)) ; inj (f::nat=>'a) |] \
  \ ==> EX n. ALL a:A. ALL m. a = f m --> m <= n";
  (* by induction on finite A *)
  by (etac finite_induct 1); ...

Goal "finite A ==> EX (b::('a::inf_class)). b ~: A";
  ... (* as a corollary of the above lemma *)
qed "ex_dist_set";

Goal "finite (A::('a::inf_class) set) ==> EX B. card B = n & finite B & A Int B = {}";
  (* by induction on n referring to ex_dist_set *)
  by (induct_tac "n" 1); ...
qed "ex_n_dist_set";

Goalw [mk_meta_eq set_mem_eq] "EX (b::('a::inf_class)). ~(b mem xs)";
  by (auto_tac ((claset() addIs [ex_dist_set]), simpset())); (* as a corollary of ex_dist_set *)
qed "ex_dist_list";
```

B Deep embedding

We do not fix a set of names *a priori*, but require it to be at least countably infinite, achieving this by an *axiomatic type-class* `inf_class` requiring the existence of an injection from `nat` into the types in it.

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

This allows us to pick a names that is distinct from another name, list of names, or finite set of names:

```
Goal "[| finite (A::(('a::inf_class) set)) ; inj (f::nat=>'a) |] \
  \ ==> EX n. ALL a:A. ALL m. a = f m --> m <= n";
  (* by induction on finite A *)
  by (etac finite_induct 1); ...

Goal "finite A ==> EX (b::('a::inf_class)). b ~: A";
  ... (* as a corollary of the above lemma *)
qed "ex_dist_set";

Goal "finite (A::('a::inf_class) set) ==> EX B. card B = n & finite B & A Int B = {}";
  (* by induction on n referring to ex_dist_set *)
  by (induct_tac "n" 1); ...
qed "ex_n_dist_set";

Goalw [mk_meta_eq set_mem_eq] "EX (b::('a::inf_class)). ~(b mem xs)";
  by (auto_tac ((claset() addIs [ex_dist_set]), simpset())); (* as a corollary of ex_dist_set *)
qed "ex_dist_list";
```

C Shallow embedding

The shallow embedding applies functions from names to processes, yielding α-equivalence classes rather than single processes. Therefore we are not able to compute the bound names of a process. Free names of process abstractions are computed via a closure over all instantiations. Note that for every function and predicate, two instances have to be defined: one for processes and the other for process abstractions.

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

(* free and fresh names *)
consts
  fn      :: (('a::inf_class) procs) => 'a set
  fna     :: (('a::inf_class) => ('a procs)) => 'a set
  fnaa    :: (('a::inf_class) => 'a => ('a procs)) => 'a set
  fresh   :: ('a::inf_class) => ('a procs) => bool
  fresha  :: ('a::inf_class) => ('a => ('a procs)) => bool
  freshaa :: ('a::inf_class) => ('a => 'a => ('a procs)) => bool

primrec
  "fn (.0) = {}"                                "fn (In a fP) = {a} Un fna fP"
  "fn (.t.P) = fn P"                            "fn (Res fP) = fna fP"
  "fn (a<b>.P) = {a,b} Un fn P"                 ...

defs
  fna_def     "fna fP == {a. ALL b. a: fn (fP b)}"
  fnaa_def    "fnaa ffP == {a. ALL b. a: fna (ffP b)}"
  fresh_def   "fresh a P == a ~: fn P"
  fresha_def  "fresha a fP == a ~: fna fP"
  freshaa_def "freshaa a ffP == a ~: fnaa ffP"

(* depth of binders *)
consts
  dob  :: "('a::inf_class) procs => 'a => nat"
  doba :: "(('a::inf_class) => ('a procs)) => 'a => nat"

primrec
  "dob (.0) c = 0"                              "dob (In a fP) c = Suc (doba fP c)"
  "dob (.t.P) c = dob P c"                      "dob (Res fP) c = Suc (doba fP c)"
  "dob (a<b>.P) c = dob P c"                    ...

defs
  doba_def "doba fP c == dob (fP c) c"
```

Well-formedness `wfna` of names abstractions require them to be either the identity or a constant function. Well-formedness `wfpa` on process abstractions uses `wfna` to implement prefixes and the names in matching and mismatching.

```
consts
  WFNA  :: (('a::inf_class) => 'a) set                "wfna fa" == "fa : WFNA"
  WFNAA :: (('a::inf_class) => 'a => 'a) set          "wfnaa ffa" == "ffa : WFNAA"
  WFP   :: (('a::inf_class) procs) set                "wfp P"   == "P : WFP"
  WFPA  :: (('a::inf_class) => ('a procs)) set        "wfpa P"  == "P : WFPA"

inductive WFNA                              inductive WFNAA
    w1 "wfna (%x. x)"  (* identity *)           w1 "wfnaa (%x y. x)"  (* identity *)
    w2 "wfna (%x. a)"  (* constants *)          w2 "wfnaa (%x y. y)"  (* identity *)
                                                w3 "wfnaa (%x y. a)"  (* constants *)
```

```
inductive WFP, WFPA
    Null  "wfp (.0)"                              Res   "wfpa fP ==> wfp (.#x. fP x)"
    Out   "wfp P ==> wfp (a<b>.P)"                Plus  "[| wfp P ; wfp Q |] ==> wfp (P .+ Q)"
    In    "wfpa fP ==> wfp (a[x].fP x)"                 ...

    Null  "wfpa (%x. .0)"
    Out   "[| wfpa fP ; wfna fa ; wfna fb |] ==> wfpa (%x. fa x<fb x>.fP x)"
    In    "[| ALL b. wfpa (ffP b) ; ALL b. wfpa (%x. ffP x b) ; \
            \ wfna fa |] ==> wfpa (%x. fa x[y].ffP y x)"
    Res   "[| ALL b. wfpa (ffP b) ; ALL b. wfpa (%x. ffP x b) |] ==> wfpa (%x. .#y. ffP y x)"
    Plus  "[| wfpa fP ; wfpa fQ |] ==> wfpa (%x. fP x .+ fQ x)"
          ...
```

D Theory of contexts

Monotonicity follows directly from the implementation. *Extensionality of contexts* is proved by induction over well-formedness of one of the two involved process abstractions. To obtain more compact proofs, we treat the induction hypothesis for input and restriction as a theorem of its own (`inst_hyp`).

```
Goal "[| wfna fa ; wfna fb ; fa a = fb a ; \
  \ a ~: fnna fa ; a ~: fnna fb |] ==> fa = fb";
  by (auto_tac (claset() addSEs WFNA.elims, simpset() addsimps [fnna_def]));
qed "ext_na";

Goal "finite ({a, c} Un (fnaa ffP) Un (fnaa ffQ))";
  by (asm_simp_tac (simpset() addsimps [finite_fnaa]) 1);
qed "lemma";

Goal "EX d. d ~= a & d ~= c & d ~: fnaa ffP & d ~: fnaa ffQ";
  by (cut_inst_tac [("a", "a"), ("c", "c"), ("ffP", "ffP"), ("ffQ", "ffQ")] lemma 1);
  by (fast_tac (claset() addSDs [ex_dist_set]) 1);
qed "fresh_ext";

Goal "[| ALL b fQ a. wfpa fQ & ffP b a = fQ a & a ~: fna (ffP b) & a ~: fna fQ --> ffP b = fQ ; \
  \ ALL b fQ a. wfpa fQ & ffP a b = fQ a & a ~: fna (%x. ffP x b) & a ~: fna fQ \
  \ --> (%x. ffP x b) = fQ ; ALL b. wfpa (ffP b) ; ALL b. wfpa (%x. ffP x b) ; \
  \ ALL b. wfpa (ffQ b) ; ALL b. wfpa (%x. ffQ x b) ; a ~: fnaa ffP ; a ~: fnaa ffQ ; \
  \ (%x. ffP x a) = (%x. ffQ x a) |] ==> (%x. ffP x c) = (%x. ffQ x c)";
  (* introduce a fresh name, d *)
  by (cut_inst_tac [("a", "a"), ("c", "c"), ("ffP", "ffP"), ("ffQ", "ffQ")] fresh_ext 1);
  by (Clarify_tac 1);
  (* instantiate first hypothesis: b -> d, fQ -> ffQ d, a -> a *)
  by (eres_inst_tac [("x", "d")] allE 1);
  by (eres_inst_tac [("x", "ffQ d")] allE 1);
  by ((rotate_tac 12 1) THEN (eres_inst_tac [("x", "a")] allE 1)); ...
  (* instantiate second hypothesis: b -> c, fQ -> %x. ffQ x c, a -> d *)
  by (eres_inst_tac [("x", "c")] allE 1);
  by (eres_inst_tac [("x", "%x. ffQ x c")] allE 1);
  by ((rotate_tac 12 1) THEN (eres_inst_tac [("x", "d")] allE 1)); ...
qed "inst_hyp";

Goal "wfpa fP ==> ALL fQ a. wfpa fQ & fP a = fQ a & a ~: fna fP & a ~: fna fQ --> fP = fQ";
  by (etac WFPA.induct 1); ... (* tedious but technical case-study applying inst_hyp *)
qed "lemma";

Goal "[| wfpa fP ; wfpa fQ ; fP a = fQ a ; fresha a fP ; fresha a fQ |] ==> fP = fQ";
  by (auto_tac (claset() addDs [lemma], simpset() addsimps [fresha_def]));
qed "ext_pa";

Goalw [freshaa_def] "[| ALL b. wfpa (ffP b) ; ALL b. wfpa (%x. ffP x b) ; ALL b. wfpa (ffQ b) ; \
  \ ALL b. wfpa (%x. ffQ x b) ; ffP a = ffQ a ; freshaa a ffP ; freshaa a ffQ |] ==> ffP = ffQ";
  ...
  (* introduce a fresh name, d *)
  by (cut_inst_tac [("a", "a"), ("c", "x"), ("ffP", "ffP"), ("ffQ", "ffQ")] fresh_ext 1);
  by (Clarify_tac 1);
  (* apply ext_pa to (ffP x) and (ffQ x) with fresh name d *)
  by (res_inst_tac [("a", "d")] ext_pa 1); ...
  (* apply ext_pa to (%x. ffP x d) and (%x. ffQ x d) with fresh name a *)
  by (dres_inst_tac [("x", "d")] inst_fun_eq 1);
  by (dres_inst_tac [("a", "a"), ("fP", "%x. ffP x d"), ("fQ", "%x. ffQ x d")]
      (rotate_prems 2 ext_pa) 1); ...
qed "ext_paa";
```

We give a constructive proof of *β-expansion*, introducing a transformation function abstracting a process over a given name. Further, we have to introduce a well-formedness predicate `wftrl` for abstractions over lists used in the transformation.

```
types
  'a trlist = "(('a::inf_class) * ('a => 'a)) list"

consts
  mk_trl :: ('a::inf_class) list => 'a => ('a trlist)
  WFTRL  :: (('a::inf_class) => 'a trlist) set           "wftrl fxs" == "fxs : WFTRL"
  trn    :: ('a::inf_class) => 'a trlist => ('a => 'a)
  tr     :: ('a::inf_class) procs => 'a trlist => 'a list => ('a => 'a procs)

primrec
  mtr1 "mk_trl [] a = []"
  mtr2 "mk_trl (x#xs) a = (if x = a then (mk_trl xs a) else (x, (%y. x))#(mk_trl xs a))"

inductive WFTRL
    etrl  "wftrl (%x. [])"
    ctrl  "[| wfnaa ffa ; wftrl fxs |] ==> wftrl (%x. (a, ffa x)#fxs x)"

primrec
  trn1 "trn a [] = (%x. x)"
  trn2 "trn a (x#xs) = (if a = fst x then snd x else (trn a xs))"

primrec
  tr0 "tr (.0) xs ys = (%x. .0)"
  tr2 "tr (a<b>.P) xs ys = (%x. (trn a xs) x<(trn b xs) x>. (tr P xs ys) x)"
  tr3 "tr (In a fP) xs ys = (%x. (trn a xs) x[y]. \
        \ (tr (fP (hd ys)) ((hd ys, (%x. y))#xs) (tl ys)) x)"
  tr4 "tr (Res fP) xs ys = (%x. .#y. (tr (fP (hd ys)) ((hd ys, (%x. y))#xs) (tl ys)) x)"
  tr5 "tr (P .+ Q) xs ys = (%x. (tr P xs ys) x .+ (tr Q xs ys) x)"
      ...
```

The proof then proceeds in four parts: after deriving (1) auxiliary results about well-formed transformation lists, we prove that (2) a suitable transformation indeed eliminates the specified name, (3) yields a well-formed abstraction, (4) which re-instantiated results in the original process. These results can then be combined in the weaker *β*-epansion theorem, requiring merely the existence of a suitable abstraction.

```
Goal "(ALL x:set xs. wfna snd x) --> wftrl (%x. xs)";
  by (induct_tac "xs" 1);
  by (auto_tac (claset() addSIs WFTRL.intrs addSEs WFNA.elims addIs WFNAA.intrs, simpset()));
qed "lemma";

Goal "ALL x:set xs. wfna snd x ==> wftrl (%x. xs)";
  by (fast_tac (claset() addIs [lemma RS mp]) 1);
qed "wfna_wftrl";

Goal "wftrl fxs ==> ALL x:set (fxs a). wfna snd x";
  by (etac WFTRL.induct 1); by (auto_tac (claset() addSEs WFNAA.elims addIs WFNA.intrs,
                simpset()));
qed "wftrl_wfna";

Goal "wftrl fxs ==> wftrl (%x. (a, (%y. b))#fxs c)";
  by (auto_tac (claset() addSIs WFTRL.intrs addIs WFNAA.intrs addSIs [wfna_wftrl, wftrl_wfna],
                simpset()));
qed "wftrl_cons1";

Goal "wftrl fxs ==> wftrl (%x. (a, (%y. x))#fxs c)";
  by (auto_tac (claset() addSIs WFTRL.intrs addIs WFNAA.intrs addSIs [wfna_wftrl, wftrl_wfna],
                simpset()));
qed "wftrl_cons2";
```

```
Goal "wftrl fxs ==> wftrl (%x. (a, (%y. b))#fxs x)";
  by (auto_tac (claset() addSIs WFTRL.intrs addIs WFNAA.intrs, simpset()));
qed "wftrl_cons3";

Goal "wftrl fxs ==> wftrl (%x. (a, (%y. x))#fxs x)";
  by (auto_tac (claset() addSIs WFTRL.intrs addIs WFNAA.intrs, simpset()));
qed "wftrl_cons4";

Goal "wftrl fxs ==> wfna trn a (fxs b)";
  by (etac WFTRL.induct 1);
  by (auto_tac (claset() addSIs WFTRL.intrs addEs WFNAA.elims addIs WFNA.intrs, simpset()));
qed "wftrl_wfna_trn1";

Goal "wftrl fxs ==> wfna (%x. trn a (fxs x) b)";
  by (etac WFTRL.induct 1); by (case_tac "a=aa" 2);
  by (auto_tac (claset() addSIs WFTRL.intrs addEs WFNAA.elims addIs WFNA.intrs, simpset()));
qed "wftrl_wfna_trn2";

(* freshness *)

Goal "wfpa fP ==> ALL xs ys. (ALL x:set xs. a ~= snd x b) & a ~= b --> a ~: fn (tr (fP c) xs ys b)";
  by (etac WFPA.induct 1); ... (* tedious but technical case-study *)
qed "lemma";

Goal "[| wfpa fP ; ALL x:(set xs). a ~= (snd x) b ; a ~= b |] ==> fresh a ((tr (fP c) xs ys) b)";
  by (fast_tac (claset() addDs [lemma] addSss (simpset() addsimps [fresh_def])) 1);
qed "tr_fresh_abs";

Goal "wfp P ==> (ALL x:(set xs). a ~= (snd x) b) & a ~= b --> a ~: fn ((tr P xs ys) b)";
  by (etac WFP.induct 1); ... (* tedious but technical case-study *)
qed "lemma";

Goal "[| wfp P ; ALL x:(set xs). a ~= (snd x) b ; a ~= b |] ==> fresh a ((tr P xs ys) b)";
  by (fast_tac (claset() addDs [lemma] addSss (simpset() addsimps [fresh_def])) 1);
qed "tr_fresh";

(* well-formedness *)

Goal "wfpa fP ==> ALL fxs ys d. wftrl fxs --> wfpa tr (fP c) (fxs d) ys & \
  \ wfpa (%x. tr (fP c) (fxs x) ys d)";
  by (etac WFPA.induct 1); ... (* tedious but technical case-study *)
qed "lemma";

Goal "[| wfpa fP ; wftrl fxs |] ==> wfpa (tr (fP c) (fxs b) ys)";
  by (fast_tac (claset() addDs [lemma]) 1);
qed "tr_wfpa_abs1";

Goal "[| wfpa fP ; wftrl fxs |] ==> wfpa (%x. (tr (fP c) (fxs x) ys) b)";
  by (fast_tac (claset() addDs [lemma]) 1);
qed "tr_wfpa_abs2";

Goal "wfp P ==> (ALL x:set(xs). wfna (snd x)) --> wfpa (tr P xs ys)";
  by (etac WFP.induct 1); ... (* tedious but technical case-study *)
qed "lemma";

Goal "[| wfp P ; ALL x:set xs. wfna snd x |] ==> wfpa tr P xs ys";
  by (fast_tac (claset() addDs [lemma]) 1);
qed "tr_wfpa";

(* equality *)

Goal "[| ALL b xs ys. (ALL x:set xs. snd x = (%y. fst x)) & doba (ffP b) c <= length ys & \
  \ fna (ffP b) <= insert a (set (map fst xs)) & a ~: set (map fst xs) & d : set (map fst xs) & \
  \ nodups ys & set ys Int insert a (set (map fst xs)) = {} --> tr (ffP b d) xs ys a = ffP b d ; \
  \ ALL b. wfpa ffP b ; ALL b. wfpa (%x. ffP x b) ; \
  \ ALL x:set xs. snd x = (%y. fst x) ; Suc (doba (ffP c) c) <= length ys ; \
  \ fnaa ffP <= insert a (set (map fst xs)) ; a ~: set (map fst xs) ; d : set (map fst xs) ; \
  \ nodups ys ; set ys Int insert a (set (map fst xs)) = {} |] \
  \ ==> (%y. tr (ffP (hd ys) d) ((hd ys, (%x. y))#xs) (tl ys) a) = (%y. ffP y d)";
  by (res_inst_tac [("a", "hd ys")] ext_pa 1);
  by (rtac tr_wfpa_abs2 1); ... (* well-formedness required by ext_pa *)
  ...                           (* equality required by ext_pa *)
  ...                           (* freshness required by ext_pa *)
qed "inst_hyp";
```

```
Goal "wfpa fP ==> ALL xs ys. (ALL x:set xs. snd x = (%y. fst x)) & (doba fP c) <= length ys & \
  \ fna fP <= insert a (set (map fst xs)) & a ~: set (map fst xs) & d : set (map fst xs) & \
  \ nodups ys & set ys Int insert a (set (map fst xs)) = {} --> (tr (fP d) xs ys) a = fP d";
  by (etac WFPA.induct 1); ... (* tedious but technical case-study using inst_hyp *)
qed "lemma";

Goal "[| wfpa fP ; ALL x:set xs. snd x = (%y. fst x) ; doba fP c <= length ys ; \
  \ fna fP <= insert a (set (map fst xs)) ; a ~: set (map fst xs) ; d : set (map fst xs) ; \
  \ nodups ys ; set ys Int insert a (set (map fst xs)) = {} |] ==> (tr (fP d) xs ys) a = fP d";
  by (dres_inst_tac [("a", "a"), ("c", "c"), ("d", "d")] lemma 1); by (Auto_tac);
qed "tr_eq_abs";

Goal "[| wfpa fP ; ALL x:set xs. snd x = (%y. fst x) ; Suc (doba fP c) <= length ys ; \
  \ fna fP <= insert a (set (map fst xs)) ; a ~: set (map fst xs) ; nodups ys ; \
  \ set ys Int insert a (set (map fst xs)) = {} |] \
  \ ==> (%y. tr (fP (hd ys)) ((hd ys, (%x. y))#xs) (tl ys) a) = fP";
  by (res_inst_tac [("a", "hd ys")] ext_pa 1);
  by (rtac tr_wfpa_abs2 1); ... (* well-formedness required by ext_pa *)
  ...                         (* equality required by ext_pa *)
  ...                         (* freshness required by ext_pa *)
qed "inst_hyp";

Goal "wfp P ==> ALL xs ys. (ALL x:set xs. snd x = (%y. fst x)) & dob P c <= length ys & \
  \ fn P <= insert a (set (map fst xs)) & a ~: set (map fst xs) & nodups ys & \
  \ set ys Int insert a (set (map fst xs)) = {} --> (tr P xs ys) a = P";
  by (etac WFP.induct 1); ... (* tedious but technical case-study using inst_hyp *)
qed "lemma";

Goal "[| wfp P ; ALL x:set xs. snd x = (%y. fst x) ; dob P c <= length ys ; \
  \ fn P <= insert a (set (map fst xs)) ; a ~: set (map fst xs) ; nodups ys ; \
  \ set ys Int insert a (set (map fst xs)) = {} |] ==> (tr P xs ys) a = P";
  by (dres_inst_tac [("a", "a"), ("c", "c")] lemma 1); by (Auto_tac);
qed "tr_eq";

(* beta-expansion *)

Goal "finite (insert a (fn P))";
  by (cut_inst_tac [("P", "P")] finite_fn 1); by (Auto_tac);
qed "finite_insert_fn";

Goal "wfp P ==> EX fP. wfpa fP & fresha a fP & P = fP a";
  (* create transformation list and store of names *)
  by (cut_inst_tac [("a", "a"), ("P", "P")] finite_insert_fn 1);
  by (dres_inst_tac [("n", "dob P c")] ex_n_dist_set 1);
  ... (* make list out of set o fresh names *)
  by (res_inst_tac [("x", "tr P (mk_trl xs a) xsa")] exI 1); (* instantiate fP *)
  (* well-formedness *)
  by (rtac conjI 1); by (fast_tac (claset() addSIs [tr_wfpa, mk_trl_wfna]) 1);
  (* freshness *)
  ... by (fast_tac (claset() addSIs [tr_fresh, mk_trl_ineq]) 1);
  (* equality *)
  by (rtac (tr_eq RS sym) 1); ...
qed "beta_exp";
```

E Adequacy

Like the proof of β-expansion in Appendix D, our proof of adequacy is constructive in that it specifies translation functions from first-order syntax into higher-order syntax and back. Note the close correspondence between these functions and the transformation function for β-reduction.

```
consts
  foenn :: "('a::inf_class) => ('a * 'a) list => 'a"
  foenc :: "(('a::inf_class) foprocs) => ('a * 'a) list => 'a procs"
  fodec :: "(('a::inf_class) procs) => ('a list) => 'a foprocs"

primrec
  "foenn a [] = a"
  "foenn a (x#xs) = (if a = snd x then fst x else (foenn a xs))"
```

```
primrec
  "foenc (foNull) xs = .0"
  "foenc (foOut a b P) xs = (foenn a xs)<(foenn b xs)>.(foenc P xs)"
  "foenc (foIn a b P) xs = (foenn a xs)[x].(foenc P ((x, b)#xs))"
  "foenc (foRes b P) xs = .#x. (foenc P ((x, b)#xs))"
  "foenc (foPlus P Q) xs = (foenc P xs) .+ (foenc Q xs)"
  ...

primrec
  "fodec (.0) ys = foNull"
  "fodec (a<b>.P) ys = foOut a b (fodec P ys)"
  "fodec (In a fP) ys = foIn a (hd ys) (fodec (fP (hd ys)) (tl ys))"
  "fodec (Res fP) ys = foRes (hd ys) (fodec (fP (hd ys)) (tl ys))"
  "fodec (P .+ Q) ys = foPlus (fodec P ys) (fodec Q ys)"
  ...
```

It is easy to show that the decoding of an encoding yields the normal-form of a process, which is in turn α-equivalent to the original process. The harder part of the proof is to show that the encoding of the decoding of a well-formed higher-order process yields exactly that process again.

```
Goal "fodec (foenc P xs) ys = fonm P xs ys";
  by (res_inst_tac [("x", "xs")] spec 1); by (res_inst_tac [("x", "ys")] spec 1);
  by (induct_tac "P" 1); by (Auto_tac);
qed "fodec_foenc_fonm";

Goal "[| nodups ys ; fodob P <= length ys ; set ys Int fobn P = {} |] \
  \ ==> fodec (foenc P []) ys = fonorm (P, ys)";
  by (dtac fonm_fonorm_gen 1); by (REPEAT (atac 1));
  by (asm_simp_tac (simpset() addsimps [fodec_foenc_fonm]) 1);
qed "fodec_foenc_fonorm";

Goal "[| nodups ys ; fodob P <= length ys ; set ys Int fon P = {} |] \
  \ ==> fodec (foenc P []) ys foalpha P";
  by (forward_tac [fodec_foenc_fonorm] 1); ...
qed "fodec_foenc_foalpha";

Goal "wfpa fP ==> ALL xs zs ys. doba fP c <= length ys & b ~: set ys & \
  \ b ~: fna fP & (ALL x:(set xs). fst x = snd x) & \
  \ (ALL x:(set zs). fst x = snd x) & a ~= b & b ~: snd ‘‘ set xs \
  \ --> b ~: fn (foenc (fodec (fP b) ys) (xs@(a,b)#zs))";
  by (etac WFPA.induct 1); ... (* tedious but technical case-analysis *)
qed "lemma";

Goalw [fresh_def] "[| wfpa fP ; doba fP c <= length ys ; b ~: set ys ; b ~: fna fP ; \
  \ (ALL x:(set xs). fst x = snd x) ; a ~= b |] \
  \ ==> fresh b (foenc (fodec (fP b) ys) ((a,b)#xs))";
  by (dres_inst_tac [("a", "a"), ("b", "b")] lemma 1); by (eres_inst_tac [("x", "[]")] allE 1);
  by (Auto_tac);
qed "foenc_fodec_fresh";

Goal "wfpa fP ==> ALL xs ys b. doba fP c <= length ys & nodups (b#ys) & \
  \ insert b (set ys) Int fna fP = {} & (ALL x:(set xs). fst x = snd x) \
  \ --> (%x. foenc (fodec (fP b) ys) ((x, b)#xs)) = fP";
  by (etac WFPA.induct 1); ... (* tedious but technical case-analysis using foenc_fodec_fresh *)
qed "lemma";

Goal "[| wfpa fP ; doba fP c <= length ys ; nodups (b#ys) ; insert b (set ys) Int fna fP = {} ; \
  \  (ALL x:(set xs). fst x = snd x) |] ==> (%x. foenc (fodec (fP b) ys) ((x, b)#xs)) = fP";
  by (dtac lemma 1); by (eres_inst_tac [("x", "xs")] allE 1); by (Auto_tac);
qed "foenc_fodec_eqa";

Goal "wfp P ==> ALL xs ys. dob P c <= length ys & nodups ys & \
  \ set ys Int fn P = {} & (ALL x:(set xs). fst x = snd x) \
  \ --> foenc (fodec P ys) xs = P";
  by (etac WFP.induct 1); ... (* tedious but technical case-analysis using foenc_fodec_eqa *)
qed "lemma";

Goal "[| wfp P ; dob P c <= length ys ; nodups ys ; set ys Int fn P = {} |] \
  \ ==> foenc (fodec P ys) [] = P";
  by (dtac lemma 1); by (eres_inst_tac [("x", "[]")] allE 1); by (Auto_tac);
qed "foenc_fodec_eq";
```

References

Aït-Mohamed, O. (1996) *Pi-Calculus Theory in HOL*. PhD thesis, Université Henri Poincaré, Nancy, France.

Amadio, R. (1993) On the reduction of CHOCS bisimulation to π-calculus bisimulation. *Proceedings of CONCUR'93: Lecture Notes in Computer Science 715*, pp. 112–126. Springer-Verlag.

Barras, B. et al. (2001) *The Coq proof assistant, version 7.0*. INRIA. available at `http://coq.inria.fr/`.

Barras, B., Boutin, S., Cornes, C., Courant, J., Coscoy, Y., Delahaye, D., de Rauglaudre, D., Filliâtre, J.-C., Giménez, E., Herbelin, H., Huet, G., Laulhère, H., Muñoz, C., Murthy, C., Parent-Vigouroux, C., Loiseleur, P., Paulin-Mohring, C., Saïbi, A. and Werner, B. (1999). *The Coq proof assistant reference manual – version 6.3.1*. Technical report, INRIA.

Berghofer, S. and Wenzel, M. (1999) Inductive datatypes in HOL–lessons learned in Formal-Logic Engineering. *Proceedings of TPHOLs '99: Lecture Notes in Computer Science 1690*, pp. 19–36.

Boulton, R., Gordon, A., Gordon, M., Harrison, J., Herbert, J. and Van Tassel, J. (1992) Experience with embedding hardware description languages in HOL. *Proceedings IFIP TC10/WG 10.2 international conference on theorem provers in circuit design: Theory, practice and experience*. IFIP Transactions, **A-10**, pp. 129–156. Elsevier.

Bucalo, A., Hofmann, M., Honsell, F., Miculan, M. and Scagnetto, I. (2001) *Using functor categories to explain and justify an axiomatisation of variables and schemata in HOAS*. In preparation.

Church, A. (1940) A formulation of the simple theory of types. *J. Symbolic Logic*, **5**, 56–68.

deBruijn, N. G. (1972) Lambda calculus notation with nameless dummies: a tool for automatic formula manipulation, with application to the Curch-Rosser theorem. *Indagationes mathematicae*, **34**, 381–392.

Despeyroux, J. (2000) A higher-order specification of the π-calculus. *Proceedings of IFIP TCS'00: Lecture Notes in Computer Science 1872*, pp. 425–439. Springer-Verlag.

Despeyroux, J. and Hirschowitz, A. (1994) Higher-order abstract syntax with induction in Coq. *Proceedings of LPAR'94: Lecture Notes in Computer Science 822*, pp. 159–173. Springer-Verlag.

Despeyroux, J., Felty, A. and Hirschowitz, A. (1995) Higher-order abstract syntax in Coq. *Proceedings of TLCA'95: Lecture Notes in Computer Science 902*, pp. 124–138. Springer-Verlag.

Despeyroux, J., Pfenning, F. and Schürmann, C. (1997) Primitive recursion for higher-order abstract syntax. *Proceedings of TLCA'97: Lecture Notes in Computer Science 1210*, pp. 147–163. Springer-Verlag.

Dowek, G., Hardin, T. and Kirchner, C. (2001) A completeness theorem for an extension of first-order logic with binders. *Proceedings of MERLIN'01*.

Fiore, M. P. & Turi, D. (2001) Semantics of name and value passing. *Proceedings of LICS'01*, pp. 93–104. IEEE Press.

Fiore, M., Plotkin, G. and Turi, D. (1999) Abstract syntax and variable binding. *Proceedings of the 14th LICS*, pp. 193–202. IEEE Press.

Gabbay, M. J. and Pitts, A. M. (1999) A new approach to abstract syntax involving binders. *Proceedings of the 14th LICS*, pp. 214–224. IEEE Computer Society Press.

Gay, S. (2001) A framework for the formalisation of pi-calculus type-systems in Isabelle/HOL. *Proceedings of TPHOLs '01: Lecture Notes in Computer Science 2152*, pp. 217–232. Springer-Verlag.

Gillard, G. (2001) *Formalization of concurrent and object languages up to alpha-conversion.* PhD thesis, Université Paris 7.

Gordon, A. and Melham, T. (1996) Five axioms of alpha-conversion. *Proceedings of TPHOLs '96: Lecture Notes in Computer Science 1125*, pp. 173–190. Springer-Verlag.

Henry-Gréard, L. (1999) *Proof of the subject reduction property for a pi-calculus in Coq.* Technical report RR-3698, INRIA.

Hirschkoff, D. (1997) A full formalisation of π-calculus theory in the calculus of constructions. *Proceedings of TPHOLs '97: Lecture Notes in Computer Science 1275*, pp. 153–169. Springer-Verlag.

Hofmann, M. (1999) Semantical analysis of higher-order abstract syntax. *Proceedings of LICS'99*, **158**, pp. 204–213. IEEE.

Honsell, F., Lenisa, M., Montanari, U. and Pistore, M. (1998) Final semantics for the pi-calculus. *Proceedings of PROCOMET'98*, pp. 225–243. Chapman & Hall.

Honsell, F., Miculan, M. and Scagnetto, I. (2001a) An axiomatic approach to metareasoning on nominal algebras in HOAS. *Proceedings of ICALP'01: Lecture Notes in Computer Science 2076*, pp. 963–978. Springer-Verlag.

Honsell, F., Miculan, M. and Scagnetto, I. (2001b) π-calculus in (Co)Inductive Type Theory. *Theor. Comput. Sci.* **253**(2), 239–285.

Kleist, J. and Sangiorgi, D. (1998) Imperative objects and mobile processes. *Proceedings of PROCOMET'98*, pp. 285–303. Chapman & Hall.

Leleu, P. (1998) *Induction et Syntaxe Abstraite d'Ordre Supérieur dans les Théories Typées.* PhD thesis, ENPC, Marne la Vallée – France.

McDowell, R. and Miller, D. (1997) A logic for reasoning with higher-order abstract syntax. *Proceedings of the 12th LICS*, pp. 434–445. IEEE Press.

McDowell, R. and Miller, D. (2001) Reasoning with Higher-Order Abstract Syntax in a Logical Framework. *ACM Trans. Computational Logic.* To appear.

McKinna, J. and Pollack, R. (1993) Pure type systems formalized. *Proceedings of TLCA'93: Lecture Notes in Computer Science 664*, pp. 289–305. Springer-Verlag.

Melham, T. (1995) A mechanized theory of the π-calculus in HOL. *Nordic J. Comput.* **1**(1), 50–76.

Miller, D. (1990) An extension to ML to handle bound variables in data structures: preliminary report. *Informal Proceedings of the Logical Frameworks BRA Workshop.* (Available as UPenn CIS technical report, MS-CIS-90-59.)

Miller, D. (1991) A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Logic and Computation*, **1**(4), 497–536.

Miller, D. (1992) The π-calculus as a theory in linear logic: Preliminary results. *Proceedings of ELP'92: Lecture Notes in Computer Science 660*, pp. 242–264. Springer-Verlag.

Miller, D. (2000) Abstract syntax for variable binders: An overview. *Proceedings of Computational Logic 2000: Lecture Notes in Artificial Intelligence 1861*, pp. 239–253. Springer Verlag.

Miller, D. (2001) Encoding Generic Judgments. *Proceedings of MERLIN'01.*

Milner, R. (1989) *Communication and Concurrency.* Prentice-Hall.

Milner, R. (1992) Functions as Processes. *J. Math. Struct. Comput. Sci.* **17**, 119–141.

Milner, R. (1999) *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press.

Milner, R., Parrow, J. and Walker, D. (1992) A calculus of mobile processes. *Infor. & Computation*, **100**, 1–77.

Nadathur, G., & Miller, D. (1988) An overview of λProlog. *Proceedings of LPC'88*, pp. 810–827. MIT Press.

Paulson, L. C. (1993) *Isabelle's object-logics*. Technical report 286, University of Cambridge, Computer Laboratory.

Paulson, L. C. (1994a) A fixedpoint approach to implementing (co)inductive definitions. *Proceedings of CADE'94: Lecture Notes in Artificial Intelligence 814*, pp. 148–161. Springer-Verlag.

Paulson, L. C. (ed). (1994b) *Isabelle: A generic theorem prover: Lecture Notes in Computer Science 828*. Springer-Verlag.

Pfenning, F. (1989) Elf: A language for logic definition and verified metaprogramming. *Proceedings of LICS'89*, pp. 313–321. IEEE.

Pfenning, F. (1996) The practice of logical frameworks. *Proceedings of ICALP'96: Lecture Notes in Computer Science 1059*, pp. 119–134. Springer-Verlag.

Pfenning, F. (1999) Logical and Meta-Logical Frameworks. *Proceedings of PPDP'99*, p. 206. Springer-Verlag.

Pfenning, F. and Elliot, C. (1988) Higher-order abstract syntax. *Proceedings of PLDI '98*, pp. 199–208. ACM Press.

Pfenning, F. and Schürmann, C. (1998) Automated theorem proving in a simple meta-logic for LF. *Proceedings of CADE'98: Lecture Notes in Computer Science 1421*, pp. 286–300. Springer-Verlag.

Pfenning, F. and Schürmann, C. (1999) System description: Twelf – a meta-logical framework for deductive systems. *Proceedings of CADE'99: Lecture Notes in Computer Science 1632*, pp. 202–206. Springer-Verlag.

Quaglia, P. (1999) The π-calculus: Notes on labelled semantics. *Bulletin EATCS*, **68**, 104–114.

Röckl, C. (2001) A First-Order Syntax for the Pi-Calculus in Isabelle/HOL using Permutations. *Proceedings of MERLIN'01*.

Röckl, C. and Sangiorgi, D. (1999) A π-calculus process semantics of concurrent idealised ALGOL. *Proceedings of FOSSACS'99: Lecture Notes in Computer Science 1578*, pp. 306–321. Springer-Verlag.

Röckl, C., Hirschkoff, D. and Berghofer, S. (2001) Higher-Order Abstract Syntax in Isabelle/HOL:. *Proceedings of FOSSACS'01: Lecture Notes in Computer Science 2030*, pp. 364–378. Springer-Verlag.

Sangiorgi, D. (1992) *Expressing mobility in process algebras: First-order and higher-order paradigms*. PhD thesis, University of Edinburgh.

Sangiorgi, D. (1996) π-calculus, internal mobility, and agent-passing calculi. *Theor. Comput. Sci.* **167**(2), 235–274.

Schürmann, C. (2001) Recursion for Higher-Order Encodings. *Proceedings of CSL'01: Lecture Notes in Computer Science 2142*, pp. 585–599. Springer-Verlag.

Thomsen, B. (1990) *Calculi for Higher Order Communicating Systems*. PhD thesis, Imperial College.

Turner, D. (1995) *The Polymorphic Pi-calculus: Theory and Implementation*. PhD thesis, University of Edinburgh.

Walker, D. (1995) Objects in the π-calculus. *Infor. & Computation*, **116**, 253–271.