Psi-calculi in Isabelle

Jesper Bengtson and Joachim Parrow

Dept. of Information Technology, Uppsala University, Sweden

Abstract. Psi-calculi are extensions of the pi-calculus, accommodating arbitrary nominal datatypes to represent not only data but also communication channels, assertions and conditions, giving it an expressive power beyond the applied pi-calculus and the concurrent constraint pi-calculus.

We have formalised psi-calculi in the interactive theorem prover Isabelle using its nominal datatype package. One distinctive feature is that the framework needs to treat binding sequences, as opposed to single binders, in an efficient way. While different methods for formalising single binder calculi have been proposed over the last decades, representations for such binding sequences are not very well explored.

The main effort in the formalisation is to keep the machine checked proofs as close to their pen-and-paper counterparts as possible. We discuss two approaches to reasoning about binding sequences along with their strengths and weaknesses. We also cover custom induction rules to remove the bulk of manual alpha-conversions.

1 Introduction

There are today several formalisms to describe the behaviour of computer systems. Some of them, like the lambda-calculus and the pi-calculus, are intended to explore fundamental principles of computing and consequently contain as few and basic primitives as possible. Other are more tailored to application areas and include many constructions for modeling convenience. Such formalisms are now being developed en masse. While this is not necessarily a bad thing there is a danger in developing complicated theories too quickly. The proofs (for example of compositionality properties) become gruesome with very many cases to check and the temptation to resort to formulations such as "by analogy with ..." or "is easily seen..." can be overwhelming. For examples in point, both the applied pi-calculus [1] and the concurrent constraint pi-calculus [8] have recently been discovered to have flaws or incompletenesses in the sense that the claimed compositionality results do not hold [5].

Since such proofs often require stamina and attention to detail rather than ingenuity and complicated new constructions they should be amenable to proof mechanisation. Our contribution in this paper is to implement a family of application oriented calculi in Isabelle [12]. The calculi we consider are the so called psi-calculi [5], obtained by extending the basic untyped pi-calculus with the following parameters: (1) a set of data terms, which can function as both communication channels and communicated objects, (2) a set of conditions, for use in conditional constructs such as **if** statements, (3) a set of assertions, used to express e.g. constraints or aliases. We base our exposition on nominal data types and these accommodate e.g. alpha-equivalence classes of terms with binders. For example, we can use a higher-order logic for assertions and conditions, and higher-order formalisms such as the lambda calculus for data terms and channels.

The main difficulty in representing calculi such as the lambda-, pi- or psi-calculi is to find an efficient treatment of binders. Informal proofs often use the Barendregt variable convention [4], that everything bound is unique. This convention provides a tractable abstraction when doing proofs involving binders, but it has recently been proven to be unsound in the general case [16]. Theorem provers have commonly used approaches based on de Bruijn indices [9], higher order abstract syntax, or nominal logic [13]. We use the nominal datatype package in Isabelle [15], and its strategy for dealing with single binders. Recent work by Aydemir et. al. introduce the locally nameless framework [2] which might be an improvement since the infrastructure is small and elegant.

One of our main contributions in the present paper is to extend the strategy to finite sequences of binders. Though it is possible to recurse over such sequences and treat each binder individually the resulting proofs would then become morasses of details with no counterpart in the informal proofs. To overcome this difficulty we introduce the notion of a *binding sequence*, which simultaneously binds arbitrarily finitely many names, and show how it can be implemented in Isabelle. We use such binding sequences to formulate and establish induction and inversion rules for the semantics of psi-calculi. The rules have been used to formally establish compositionality properties of strong bisimilarity. The proofs are close to their informal counterparts.

We are not aware of any other work on implementing calculi of this calibre in a proof assistant such as Isabelle. The closest related work are implementations of the basic pi-calculus, by ourselves [6] and also by others [10, 11, 14]. Neither are we aware of any other general technique for multiple binders, other than the yet unpublished work by Berghofer and Urban which we describe in Section 3.

The rest of the paper is structured as follows. In Section 2 we give a brief account of psi-calculi and how they use nominal data types. Section 3 treats implementation issues related to binding sequences and alpha-conversion. In Section 4 we show how these are used to create our formalisation. In Section 5 we report on the current status of the effort and ideas for further work.

2 Psi-calculi

This section is a brief recapitulation of psi-calculi and nominal data types; for a more extensive treatment including motivations and examples see [5].

2.1 Nominal data types

We assume a countably infinite set of atomic *names* $\mathcal{N}$ ranged over by $a, b, \ldots, z$. Intuitively, names will represent the symbols that can be statically scoped, and also represent symbols acting as variables in the sense that they can be subject to substitution. A *nominal set* [13] is a set equipped with *name swapping* functions written $(a\ b)$, for any names a, b. An intuition is that $(a\ b)\cdot X$ is X with a replaced by b and b replaced by a. A sequence of swappings is called a permutation, often denoted p, where $p \cdot X$ means the term X with the permutation p applied to it. We write p^- for the reverse of p. The *support* of X, written $\mathrm{n}(X)$, is the least set of names A such that $(a\ b) \cdot X = X$ for all a, b not in A. We write $a\#X$, pronounced "a is fresh for X", for $a \notin \mathrm{n}(X)$. If A is a set of names we write $A\#X$ to mean $\forall a \in A\ .\ a\#X$. We require all elements to have finite support, i.e., $\mathrm{n}(X)$ is finite for all X. A function f is *equivariant* if $(a\ b) \cdot f(X) = f((a\ b) \cdot X)$ holds for all X, and similarly for functions and relations of any arity. Intuitively, this means that all names are treated equally.

2.2 Agents

A psi-calculus is defined by instantiating three nominal data types and four operators:

Definition 1 (Psi-calculus parameters). *A psi-calculus requires the three (not necessarily disjoint) nominal data types:*

$\mathbf{T}$	*the (data) terms, ranged over by M, N*
$\mathbf{C}$	*the conditions, ranged over by φ*
$\mathbf{A}$	*the assertions, ranged over by Ψ*

and the four equivariant operators:

$\dot{\leftrightarrow}: \mathbf{T} \times \mathbf{T} \to \mathbf{C}$	*Channel Equivalence*
$\otimes : \mathbf{A} \times \mathbf{A} \to \mathbf{A}$	*Composition*
$\mathbf{1} : \mathbf{A}$	*Unit*
$\vdash \subseteq \mathbf{A} \times \mathbf{C}$	*Entailment*

We require the existence of a substitution function for $\mathbf{T}$, $\mathbf{C}$ and $\mathbf{A}$. When X is a term, condition or assertion we write $X[\widetilde{a} := \widetilde{T}]$ to mean the simultaneous substitution of the names $\widetilde{a}$ for the terms $\widetilde{T}$ in X. The exact requisites of this function will be covered in Section 4.

The binary functions above will be written in infix. Thus, if M and N are terms then $M \dot{\leftrightarrow} N$ is a condition, pronounced "M and N are channel equivalent" and if Ψ and Ψ' are assertions then so is $\Psi \otimes \Psi'$. Also we write $\Psi \vdash \varphi$, "Ψ entails φ", for $(\Psi, \varphi) \in \vdash$.

We say that two assertions are equivalent if they entail the same conditions:

Definition 2 (assertion equivalence). *Two assertions are* equivalent*, written $\Psi \simeq \Psi'$, if for all φ we have that $\Psi \vdash \varphi \Leftrightarrow \Psi' \vdash \varphi$.*

Channel equivalence must be symmetric and transitive, $\otimes$ must be compositional with regard to $\simeq$, and the assertions with $(\otimes, \mathbf{1})$ form an abelian monoid.

In the following $\tilde{a}$ means a finite (possibly empty) sequence of names, $a_1, \ldots, a_n$. The empty sequence is written ϵ and the concatenation of $\tilde{a}$ and $\tilde{b}$ is written $\tilde{a}\tilde{b}$. When occurring as an operand of a set operator, $\tilde{a}$ means the corresponding set of names $\{a_1, \ldots, a_n\}$. We also use sequences of terms, conditions, assertions etc. in the same way.

A *frame* can intuitively be thought of as an assertion with local names:

Definition 3 (Frame). *A* frame *F is a pair $\langle B_F, \Psi_F \rangle$ where B_F is a sequence of names that bind into the assertion Ψ_F. We use F, G to range over frames.*

Name swapping on a frame just distributes to its two components. We identify alpha equivalent frames, so $\mathrm{n}(F) = \mathrm{n}(\Psi_F) - \mathrm{n}(B_F)$. We overload $\mathbf{1}$ to also mean the least informative frame $\langle \epsilon, \mathbf{1} \rangle$ and $\otimes$ to mean composition on frames defined by $\langle B_1, \Psi_1 \rangle \otimes \langle B_2, \Psi_2 \rangle = \langle B_1 B_2, \Psi_1 \otimes \Psi_2 \rangle$ where B_1 is disjoint from $\mathrm{n}(B_2, \Psi_2)$ and vice versa. We also write $\Psi \otimes F$ to mean $\langle \epsilon, \Psi \rangle \otimes F$, and $(\nu b)F$ to mean $\langle b B_F, \Psi_F \rangle$.

Definition 4 (Equivalence of frames). *We define $F \vdash \varphi$ to mean that there exist B_F and Ψ_F such that $F = \langle B_F, \Psi_F \rangle$, $B_F \# \varphi$, and $\Psi_F \vdash \varphi$. We also define $F \simeq G$ to mean that for all φ it holds that $F \vdash \varphi$ iff $G \vdash \varphi$.*

Intuitively a condition is entailed by a frame if it is entailed by the assertion and does not contain any names bound by the frame. Two frames are equivalent if they entail the same conditions.

Definition 5 (psi-calculus agents). *Given valid psi-calculus parameters as in Definition 1, the psi-calculus* agents*, ranged over by $P, Q, \ldots$, are of the following forms.*

$\overline{M}\, N.P$	Output
$\underline{M}(\lambda\widetilde{x})N.P$	Input
$\mathbf{case}\ \varphi_1 : P_1\ [\!]\ \cdots\ [\!]\ \varphi_n : P_n$	Case
$(\nu a)P$	Restriction
$P \mid Q$	Parallel
$!P$	Replication
$(\!\vert \Psi \vert\!)$	Assertion

In the Input $\underline{M}(\lambda\widetilde{x})N.P$ we require that $\widetilde{x} \subseteq \mathrm{n}(N)$ is a sequence without duplicates, and here any name in $\widetilde{x}$ binds its occurrences in both N and P. Restriction binds a in P. An assertion is guarded *if it is a subterm of an Input or Output . In a replication $!P$ there may be no unguarded assertions in P, and in* $\mathbf{case}\ \varphi_1 : P_1\ [\!]\ \cdots\ [\!]\ \varphi_n : P_n$ *there may be no unguarded assertion in any P_i.*

Formally, we define name swapping on agents by distributing it over all constructors, and substitution on agents by distributing it and avoiding captures by binders through alpha-conversion in the usual way. We identify alpha-equivalent agents; in that way we get a nominal data type of agents where the support $\mathrm{n}(P)$ of P is the union of the supports of the components of P, removing the names bound by λ and ν, and corresponds to the names with a free occurrence in P.

$$\textsc{In}\ \frac{\Psi \vdash M \dot{\leftrightarrow} K}{\Psi \rhd \underline{M}(\lambda\tilde{y})N.P \xrightarrow{\underline{K}\,N[\tilde{y}:=\tilde{L}]} P[\tilde{y}:=\tilde{L}]} \qquad \textsc{Out}\ \frac{\Psi \vdash M \dot{\leftrightarrow} K}{\Psi \rhd \overline{M}\,N.P \xrightarrow{\overline{K}\,N} P}$$

$$\textsc{Case}\ \frac{\Psi \rhd P_i \xrightarrow{\alpha} P' \qquad \Psi \vdash \varphi_i}{\Psi \rhd \mathbf{case}\ \tilde{\varphi} : \tilde{P} \xrightarrow{\alpha} P'}$$

$$\textsc{Com}\ \frac{\begin{array}{c}\Psi_Q \otimes \Psi \rhd P \xrightarrow{\overline{M}\,(\nu\tilde{a})N} P' \\ \Psi_P \otimes \Psi \rhd Q \xrightarrow{\underline{K}\,N} Q' \qquad \Psi \otimes \Psi_P \otimes \Psi_Q \vdash M \dot{\leftrightarrow} K\end{array}}{\Psi \rhd P \mid Q \xrightarrow{\tau} (\nu\tilde{a})(P' \mid Q')}\ \tilde{a} \# Q$$

$$\textsc{Par}\ \frac{\Psi_Q \otimes \Psi \rhd P \xrightarrow{\alpha} P'}{\Psi \rhd P|Q \xrightarrow{\alpha} P'|Q}\ \mathrm{bn}(\alpha) \# Q \qquad \textsc{Scope}\ \frac{\Psi \rhd P \xrightarrow{\alpha} P'}{\Psi \rhd (\nu b)P \xrightarrow{\alpha} (\nu b)P'}\ b \# \alpha, \Psi$$

$$\textsc{Open}\ \frac{\Psi \rhd P \xrightarrow{\overline{M}\,(\nu\tilde{a})N} P'}{\Psi \rhd (\nu b)P \xrightarrow{\overline{M}\,(\nu\tilde{a}\cup\{b\})N} P'}\ \begin{array}{l} b \# \tilde{a}, \Psi, M \\ b \in \mathrm{n}(N)\end{array} \qquad \textsc{Rep}\ \frac{\Psi \rhd P \mid !P \xrightarrow{\alpha} P'}{\Psi \rhd !P \xrightarrow{\alpha} P'}$$

Table 1. Structured operational semantics. Symmetric versions of Com and Par are elided. In the rule Com we assume that $\mathcal{F}(P) = \langle B_P, \Psi_P \rangle$ and $\mathcal{F}(Q) = \langle B_Q, \Psi_Q \rangle$ where B_P is fresh for all of Ψ, B_Q, Q, M and P, and that B_Q is similarly fresh. In the rule Par we assume that $\mathcal{F}(Q) = \langle B_Q, \Psi_Q \rangle$ where B_Q is fresh for Ψ, P and α. In Open the expression $\nu\tilde{a} \cup \{b\}$ means the sequence $\tilde{a}$ with b inserted anywhere.

Definition 6 (Frame of an agent). *The* frame $\mathcal{F}(P)$ of an agent P *is defined inductively as follows:*

$$\begin{array}{l}
\mathcal{F}(\underline{M}(\lambda\tilde{x})N.P) = \mathcal{F}(\overline{M}\,N.P) = \mathcal{F}(\mathbf{case}\ \tilde{\varphi} : \tilde{P}) = \mathcal{F}(!P) = \mathbf{1} \\
\mathcal{F}((\!|\Psi|\!)) = \langle \epsilon, \Psi \rangle \\
\mathcal{F}(P \mid Q) = \mathcal{F}(P) \otimes \mathcal{F}(Q) \\
\mathcal{F}((\nu b)P) = (\nu b)\mathcal{F}(P)
\end{array}$$

2.3 Operational semantics

The *actions* ranged over by α, β are of the following three kinds: Output $\overline{M}(\nu\tilde{a})N$, Input $\underline{M}\,N$, and Silent τ. Here we refer to M as the *subject* and N as the *object*. We define $\mathrm{bn}(\overline{M}\,(\nu\tilde{a})N) = \tilde{a}$, and $\mathrm{bn}(\alpha) = \emptyset$ if α is an input or τ.

Definition 7 (Transitions).

A transition *is of the kind* $\Psi \rhd P \xrightarrow{\alpha} P'$, *meaning that in the environment* Ψ *the agent* P *can do an* α *to become* P'*. The transitions are defined inductively in Table 1.*

3 Binding sequences

The main difficulty when formalising any calculus with binders is to handle alpha-equivalence. The techniques that have been used thus far by theorem provers share the trait that they only reason about single binders. This works well for many calculi, but psi-calculi require binding sequences of arbitrary length. For our psi-calculus datatype (Def. 5), a binding sequence is needed in the Input-case where the term $\underline{M}(\lambda\widetilde{x})N.P$ has the sequence $\widetilde{x}$ binding into N and P. The second place sequences are needed is when defining frames (Def 3). Frames are derived from processes (Def. 6) and as agents can have an arbitrary number of binders, so can the frames. The third occurrence of binding sequences can be found in the operational semantics (Table 1). In the transition $\Psi \rhd P \xrightarrow{\overline{M}\,(\nu\widetilde{a})N} P'$, the sequence $\widetilde{a}$ represents the bound names in P which occur in the object N.

In order to formalise these types of calculi efficiently in a theorem prover, libraries with support for sequences of binders have to be added. In the next sections we will discuss two approaches that have been made in this area, first one by us, which we call explicit binding sequences, and then one by Berghofer and Urban which we in this paper will call implicit binding sequences. They both build on the existing nominal representation of alpha-equivalence classes where a binding occurrence of the name a in the term T is written $[a].T$, and the support of $[a].T$ is the support of T with a removed. From this definition, creating a term with the binding sequence $\tilde{a}$ in the term T, written $[\tilde{a}].T$, can easily be done by recursion over $\tilde{a}$. The proof that the support of $[\tilde{a}].T$ is equal to the support of T with the names of $\tilde{a}$ removed is trivial. Similarly, the notion of freshness needs to be expanded to handle sequences. The expression $\tilde{a}\#T$ is defined as: $\forall x \in \texttt{set}\ \tilde{a}.\ x\#T$. This expression is overloaded for when $\tilde{a}$ is either a list or a set.

3.1 Explicit binding sequences

Our approach is to scale the existing single binder setting to sequences. Isabelle has native support for generating fresh names, i.e. given any finite context of names $\mathcal{C}$, Isabelle can generate a name fresh for that context. There is also a distinctness predicate, written `distinct` $\widetilde{a}$ which states that $\widetilde{a}$ contains no duplicates. From these we can generate a finite sequence $\widetilde{a}$ of arbitrary length n where `length` $\widetilde{a} = n$, $\widetilde{a}\#\mathcal{C}$ and `distinct` $\widetilde{a}$ by induction on n.

The term $[a].T$ can be alpha-converted into the term $[b].(a\ b)\cdot T$ if $b\#T$, where we call $(a\ b)$ an alpha-converting swapping. In order to mimic this behaviour with sequences, we lift name swapping to sequence swapping by pairwise composing the elements of two sequences to create an alpha-converting permutation. We will write $(\widetilde{a}\ \widetilde{b})$ for such a composition defined in the following manner:

Definition 8.
$([]\ []) = []$
$((x :: xs)\ (y :: ys)) = (x, y) :: (xs\ ys)$

All theories that construct permutations using this function will ensure that the length of the sequences are equal.

We can now lift alpha-equivalence to support sequences.

Lemma 1. *If* $\texttt{length}\,\tilde{x} = \texttt{length}\,\tilde{y}$, $\texttt{distinct}\,\tilde{y}$, $\tilde{x}\#\tilde{y}$ *and* $\tilde{y}\#T$ *then* $[\tilde{x}].T = [\tilde{y}].(\tilde{x}\ \tilde{y}) \cdot T$.

Proof. By induction on the length of $\tilde{x}$ and $\tilde{y}$.

The distinctness property is a bit stronger than strictly necessary; we only need that the names in $\tilde{x}$ that actually occur in T have a unique corresponding member in $\tilde{y}$. Describing this property formally would be cumbersome and distinctness is sufficient and easy to work with.

Long proofs tend to introduce alpha-converting permutations and it is therefor important to have a strategy for cancelling these. If a term T has been alpha-converted using the swapping $(a\ b)$, becoming $(a\ b) \cdot T$, it is possible to apply the same swapping to the expression where $(a\ b) \cdot T$ occurs. Using equivariance properties, the swapping can be distributed over the expression, and when it reaches $(a\ b) \cdot T$, it will cancel out since $(a\ b) \cdot (a\ b) \cdot T = T$. It can also be cancelled from any remaining term U in the expression, as long as $a\#U$ and $b\#U$. This technique is also applicable when dealing with sequences, where the alpha-converted term has the form $(\widetilde{a}\ \widetilde{b}) \cdot T$, with one important observation. Even though $(a\ b) \cdot (a\ b) \cdot T = T$, it is not generally the case that $p \cdot p \cdot T = T$. To cancel a permutation on a term, its inverse must be applied, i.e. $p^- \cdot p \cdot T = T$. By applying $(\widetilde{a}\ \widetilde{b})^-$ to the expression, the alpha-converting permutation will cancel out. The permutation will also be cancelled from any remaining term U as long as $\widetilde{a}\#U$ and $\widetilde{b}\#U$ since $\widetilde{a}\#U$ and $\widetilde{b}\#U$ implies $(\widetilde{a}\ \widetilde{b}) \cdot U = U$ and $(\widetilde{a}\ \widetilde{b})^- \cdot U = U$

In this setting we are able to fully formalise our theories using binding sequences. The disadvantage is that facts regarding lengths of sequences and distinctness need to be maintained throughout the proofs.

3.2 Implicit binding sequences

Parallel to our work, Berghofer and Urban developed an alternative theory for binding sequences which is also being included in the nominal package. Their approach is to generate the alpha-converting permutation directly using the following lemma:

Lemma 2. *There exists a permutation* p *s.t.* $\texttt{set}\ p \subseteq \texttt{set}\ \tilde{x} \times \texttt{set}(p \cdot \tilde{x})$ *and* $(p \cdot \tilde{x})\#C$.

The intuition is that instead of creating a fresh sequence, a permutation is created which when applied to a sequence ensures the needed freshness conditions. The following corollary makes it possible to discard permutations which are sufficiently fresh:

Corollary 1. *If* $\tilde{x}\#T$, $\tilde{y}\#T$ *and* $\texttt{set}\ p \subseteq \texttt{set}\ \tilde{x} \times \texttt{set}\ \tilde{y}$ *then* $p \cdot T = T$.

From this, a corollary to perform alpha-conversions can be created.

Corollary 2. *If* $\texttt{set}\ p \subseteq \texttt{set}\ \tilde{x} \times \texttt{set}(p \cdot \tilde{x})$ *and* $(p \cdot \tilde{x}) \# T$ *then* $[\tilde{x}].T = [p \cdot \tilde{x}].p \cdot T$.

Proof. since $\tilde{x} \# [\tilde{x}].T$ and $(p \cdot \tilde{x}) \# T$ we have by Cor. 1 that $[\tilde{x}].T = p \cdot [\tilde{x}].T$ and hence by equivariance that $[\tilde{x}].T = [p \cdot \tilde{x}] \,.\, p \cdot T$.

This method has the problem that when cancelling alpha-converting permutations as in section 3.1, the freshness conditions we use to cancel the permutation from the remaining terms are lost since $(p \cdot \widetilde{x}) \# U$ does not imply $(p^- \cdot \widetilde{x}) \# U$. We define the following predicate to fix this.

Definition 9. $\texttt{distinctPerm}\ p \equiv \texttt{distinct}((\texttt{map fst}\ p)@(\texttt{map snd}\ p))$

Intuitively, the `distinctPerm` predicate ensures that all names in a permutation are distinct.

Corollary 3. *If* $\texttt{distinctPerm}\ p$ *then* $p \cdot p \cdot T = T$

Proof. By induction on p.

Thus, by extending Lemma 2 with the condition $\texttt{distintPerm}\ p$ we get permutations p which can be cancelled by applying p again rather than its inverse.

In general, proofs are easier if we know that the binding sequences are distinct. The following corollary helps.

Corollary 4. *If* $\tilde{a} \# \mathcal{C}$ *then there exists an* $\tilde{b}$ *s.t.* $[\tilde{a}].T = [\tilde{b}].T$ *and* $\texttt{distinct}\ \tilde{b}$ *and* $\tilde{b} \# \mathcal{C}$.

Proof. Since each name in $\tilde{a}$ can only bind once in T we can construct $\tilde{b}$ by replacing any duplicate name in $\tilde{a}$ with a sufficiently fresh name.

The advantage of implicit alpha-conversions is that facts about length and distinctness of sequences do not need to be maintained through the proofs. The freshness conditions are the ones needed for the single binder case and the distinctness properties are only needed when cancelling permutations. For most cases, this method is more convenient to work with. There are disadvantages regarding inversion rules, and alpha-equivalence properties that will be discussed in the next section.

3.3 Alpha-equivalence

When reasoning with single binders, the nominal approach to alpha-equivalence is quite straightforward. Two terms $[a].T$ and $[b].U$ are equal if and only if either $a = b$ and $T = U$ or $a \neq b$, $a \# U$ and $U = (a\ b) \cdot T$. Reasoning about binding sequences is more difficult. Exactly what does it mean for two terms $[\widetilde{a}].T$ and $[\widetilde{b}].U$ to be equal? As long as T and U cannot themselves have binding sequences on a top level we know that $\texttt{length}\ \widetilde{a} = \texttt{length}\ \widetilde{b}$, but the problem with the general case is what happens when $\widetilde{a}$ and $\widetilde{b}$ partially share names. As it turns out, this case is not important in order to reason about these types of equalities, but special heuristics are required.

The times where we actually get assumptions such as $[\widetilde{a}].T = [\widetilde{b}].U$ in our proofs are when we do induction or inversion over a term with binders. Typically, $[\widetilde{b}].U$ is the term we start with, and $[\widetilde{a}].T$ is the term that appears in the induction or inversion rule. These rules are designed in such a way that any bound names appearing in the rules can be assumed to be sufficiently fresh. More precisely, we can ensure that $\widetilde{a}\#\widetilde{b}$ and $\widetilde{a}\#U$. If we are working with explicit binding sequences we can also know that $\widetilde{a}$ is distinct. In this case, the heuristic is straightforward. Using the information provided by the induction rule we know using Lemma 1 that $[\widetilde{b}].U = [\widetilde{a}].(\widetilde{a}\ \widetilde{b}) \cdot U$ and hence that $T = (\widetilde{a}\ \widetilde{b}) \cdot U$. From here we continue with the proofs similarly to the single binder case.

When working with implicit sequences the problem is a bit more delicate. These rules have been generated using a permutation designed to ensure freshness conditions and we do not know exactly how $\widetilde{a}$ and $\widetilde{b}$ originally related to each other. We do know that the terms are alpha-equivalent and as such, there is a permutation which equates them. We first prove the following corollary:

Corollary 5. *If $[a].T = [b].U$ then $a \in \texttt{supp}\ T = b \in \texttt{supp}\ U$ and $a\#T = b\#U$.*

Proof. By the definition of alpha-equivalence on terms.

We can now prove the following lemma:

Lemma 3. *If $[\widetilde{a}].T = [\widetilde{b}].U$ and $\widetilde{a}\#\widetilde{b}$ then there exists a permutation p s.t. $\texttt{set}\ p \subseteq \texttt{set}\ \widetilde{a} \times \texttt{set}\ \widetilde{b}$, $\widetilde{a}\#U$ and $T = p \cdot U$*

Proof. The intuition here is to construct p by using Cor. 5 to filter out the pairs of names from $\widetilde{a}$ and $\widetilde{b}$ that do not occur in T and U respectively and pairing together the rest. The proof is done by induction on the length of $\widetilde{a}$ and $\widetilde{b}$.

The problem with this approach is that we do not know how $\widetilde{a}$ and $\widetilde{b}$ are related. If we know that they are both distinct then we can construct p such that $\widetilde{a} = p \cdot \widetilde{b}$ but generally we do not know this. The problematic cases are the ones dealing with inversion, in which case we resort to explicit binding sequences, but for the majority of our proofs Lemma 3 is enough.

4 Formalisation

Psi-calculi are parametric calculi. A specific instance is created by instantiating the framework with dataterms for the terms, assertions and conditions of the calculus. We also require an entailment relation, a notion of channel equality and composition of assertions. Isabelle has good support for reasoning about parametric systems through the use of locales [3].

4.1 Substitution properties

We require a substitution function on agents. Since terms, assertions and conditions of psi-calculi are parameters, a locale is created to ensure that a set of substitution properties hold.

Definition 10. *A term M of type α is a* `substType` *if there is a substitution function* `subst` $:: \alpha \Rightarrow$ `name list` $\Rightarrow \beta$ `list` $\Rightarrow \alpha$ *which meets the following constraints, where* `length` $\tilde{x}$ = `length` $\tilde{T}$ *and* `distinct` $\tilde{x}$

Equivariance:	$p \cdot (M[\tilde{x} := \tilde{T}]) = (p \cdot M)[(p \cdot \tilde{x}) := (p \cdot \tilde{T})]$
Freshness:	*if* $a \# M[\tilde{x} := \tilde{T}]$ *and* $a \# \tilde{x}$ *then* $a \# M$
	if $a \# M$ *and* $a \# \tilde{T}$ *then* $a \# M[\tilde{x} := \tilde{T}]$
	if $\texttt{set}\ \tilde{x} \subseteq \texttt{supp}\ M$ *and* $a \# M[\tilde{x} := \tilde{T}]$ *then* $a \# \tilde{T}$
	if $\tilde{x} \# \tilde{M}$ *then* $M[\tilde{x} := \tilde{T}] = M$
	if $\tilde{x} \# \tilde{y}$ *and* $\tilde{y} \# \tilde{T}$ *then* $M[\tilde{x}\tilde{y} := \tilde{T}\tilde{U}] = (M[\tilde{x} := \tilde{T}])[\tilde{y} := \tilde{U}]$
Alpha-equivalence:	*if* $\texttt{set}\ p \subseteq \texttt{set}\ \tilde{x} \times \texttt{set}(p \cdot \tilde{x})$ *and* $(p \cdot \tilde{x}) \# M$ *then*
	$M[\tilde{x} := \tilde{T}] = (p \cdot M)[(p \cdot \tilde{x}) := \tilde{T}]$

The intuition is that `subst` is a simultaneous substitution function which replaces all occurrences of the names in $\tilde{x}$ in M with the corresponding dataterm in $\tilde{T}$. All that the locale dictates is that there is a function of the correct type which satisfies the constraints. Exactly how it works needs only be specified when creating an instance of the calculus in order to prove that the constraints are satisfied.

These constraints are the ones we need for the formalisation but we have not proven that they are strictly minimal. We leave this for future work.

4.2 The psi datatype

Nominal Isabelle does not support datatypes with binding sequences or nested datatypes. The two cases that are problematic when formalising psi-calculi are the Input case, which requires a binding sequence, and the Case case which requires a list of assertions and processes. The required datatype can be encoded using mutual recursion in the following way.

Definition 11. *The psi-calculi datatype has three type variables for terms, assertions and conditions respectively. In the* `Res` *and the* `Bind` *cases,* `name` *is a binding occurrence.*

```
nominal_datatype (α, β, γ) psi = Output α α (α, β, γ) psi
                               | Input α (α, β, γ) input
                               | Case (α, β, γ) case
                               | Par ((α, β, γ) psi) ((α, β, γ) psi)
                               | Res ≪name≫ ((α, β, γ) psi)
                               | Assert β
                               | Bang (α, β, γ) psi
and (α, β, γ) input            = Term α (α, β, γ) psi
                               | Bind ≪name≫ ((α, β, γ) input)
and (α, β, γ) case             = EmptyCase
                               | Cond γ ((α, β, γ) psi) ((α, β, γ) case)
```

In order to create a substitution function for $(\alpha\ \beta\ \gamma)$ `psi` we create a locale with the following three substitution functions as `substType`*s*.

$$\begin{aligned}
&\texttt{substTerm} &&:: \alpha \Rightarrow \texttt{name list} \Rightarrow \alpha\ \texttt{list} \Rightarrow \alpha\\
&\texttt{substAssert} &&:: \beta \Rightarrow \texttt{name list} \Rightarrow \alpha\ \texttt{list} \Rightarrow \beta\\
&\texttt{substCond} &&:: \gamma \Rightarrow \texttt{name list} \Rightarrow \alpha\ \texttt{list} \Rightarrow \gamma
\end{aligned}$$

These functions will handle substitutions on terms, assertions and conditions respectively. Note that we always substitute names for terms.

The substitution function for `psi` can now be defined in the standard way where the substitutions are pushed through the datatype avoiding the binders. The axioms for `substType` can then be proven for the `psi` substitution function where the axioms themselves are used when the proofs reaches the terms, assertions and conditions.

4.3 Frames

The four nominal morphisms from Def. 1 are also encoded using locales along with their equivariance properties. From this definition, implementing Def. 2 and a locale for our requirements on assertion equivalence $\simeq$ is straightforward. To implement frames, the following nominal datatype is created:

Definition 12.

$$\begin{aligned}
\texttt{nominal\_datatype}\ \beta\ \texttt{frame} = {}&\texttt{Assertion}\ \beta\\
|\ &\texttt{FStep}\ \ll \texttt{name} \gg\ (\beta\ \texttt{frame})
\end{aligned}$$

In order to overload the $\otimes$ operator to work on frames as described in Def. 3 we create the following two nominal functions.

Definition 13.

$$\begin{aligned}
&\texttt{insertAssertion}\ (\texttt{Assertion}\ \Psi)\ \Psi' = \texttt{Assertion}(\Psi' \otimes \Psi)\\
x\#\Psi' \Rightarrow\ &\texttt{insertAssertion}\ (\texttt{FStep}\ x\ F)\ \Psi = \texttt{FStep}\ x\ (\texttt{insertAssertion}\ F\ \Psi')\\
&(\texttt{Assertion}\ \Psi) \otimes G = \texttt{insertAssertion}\ G\ \Psi\\
x\#G \Rightarrow\ &(\texttt{FStep}\ x\ F) \otimes G = \texttt{FStep}\ x\ (F \otimes G)
\end{aligned}$$

The following lemma is then derivable:

Lemma 4. *If* $B_P \# B_Q$, $B_P \# \Psi_Q$ *and* $B_Q \# \Psi_P$
then $\langle B_P, \Psi_P\rangle \otimes \langle B_Q, \Psi_Q\rangle = \langle B_P @ B_Q, \Psi_P \otimes \Psi_Q\rangle$.

The implementations of Defs. 4 and 6 are then straightforward.

4.4 Operational semantics

The operational semantics in Def. 7 is formalised in a similar manner to [6]. Since the actions on the labels can contain bound names which bind into the derivative of the transition, a residual datatype needs to be created which combines the actions with their derivatives. Since a bound output can contain an arbitrary number of bound names, binding sequences must be used here in a similar manner to `psi` and `frame`.

```
nominal_datatype (α, β, γ) boundOutput =
        Output α (α, β, γ) psi
       | BStep ≪name≫ (α, β, γ) boundOutput

datatype α action = Input α α
                  | Tau

datatype (α, β, γ) residual = Free (α action) ((α, β, γ) psi)
                             | Bound α ((α, β, γ) boundOutput)
```

We will use the notation $(\nu\widetilde{a})N \prec P$ for a term of type `boundOutput` which has the binding sequence $\widetilde{a}$ into N and P. We can also write $\Psi \rhd P \longmapsto M(\nu\widetilde{a})N \prec P'$ for $\Psi \rhd P \xrightarrow{\overline{M}\,(\nu\widetilde{a})N} P'$ and similarly for input and tau transitions.

As usual, the operational semantics is defined using an inductively defined predicate. As in [6] rules which can have either free or bound residuals are split into these two cases. We also saturate our rules with freshness conditions to ensure that the bound names are fresh for for all terms outside their scope. This is done to satisfy the vc-property described in [16] so that Isabelle can automatically infer an induction rule, but also to give us as much freshness information as possible when doing induction on transitions. Moreover, all frames are required to have distinct binding sequences. The introduction rules in Table 1 only include the freshness conditions which are strictly necessary and frames with non distinct binding sequences. These can be inferred from our inductive definition using regular alpha converting techniques and Cor. 4.

We will not cover the complete semantics here, just two rules to demonstrate some differences to the pen-and-paper formalisation.

The transition rule Par has the implicit assumption that $\mathcal{F}(Q) = \langle B_Q,\ \Psi_Q\rangle$. When formalising the semantics, one inductive case will look as follows:

$$\textsc{Par}\ \frac{\Psi_Q \otimes \Psi \rhd P \xrightarrow{\alpha} P' \qquad \mathcal{F}(Q) = \langle B_Q,\ \Psi_Q\rangle}{\Psi \rhd P|Q \xrightarrow{\alpha} P'|Q}\ \begin{array}{c} B_Q \# \Psi, P, \alpha, P', Q \\ \texttt{distinct}\ B_Q \end{array}$$

Inferring the transition for P means selecting a specific alpha-variant of $\mathcal{F}(Q)$ as Ψ_Q appears without binders in the inference of the transition. Freshness conditions for B_Q are central for the proofs to hold.

Next consider the rule Open. We want the binding sequence on the transition to behave like a set in that we must not depend on the order of its names. Our formalisation solves this by explicitly splitting the binding sequence in two and placing the opened name in between. By creating a rule which holds for all such splits, we mimic the effect of a set.

$$\textsc{Open}\ \frac{\Psi \rhd P \xrightarrow{\overline{M}\,(\nu\widetilde{a}\widetilde{c})N} P'}{\Psi \rhd (\nu b)P \xrightarrow{\overline{M}\,(\nu\widetilde{a}b\widetilde{c})N} P'}\ \begin{array}{c} b \# \widetilde{a}, \widetilde{c}, \Psi, M \\ b \in \mathrm{n}(N) \\ \widetilde{a} \# \Psi, P, M, \widetilde{c} \\ \widetilde{c} \# \Psi, P, M \end{array}$$

4.5 Induction rules

At the core of any nominal formalisation is the need to create custom induction rules which allow the introduced bound names to be fresh for any given context. Without these, the user is forced to do manual alpha-conversions throughout the proofs and such proofs will differ significantly from their pen and paper counterparts, where freshness is just assumed. An in depth description can be found in [16]. Very recent additions to the nominal package generate induction rules where the user is allowed to choose a set of name which can be arbitrarily fresh for each inductive case. In most cases, this set will be the set of binders present in the rule.

Standard induction Isabelle will automatically create a rule for doing induction on transitions of the form $\Psi \rhd P \longmapsto Rs$, where Rs is a residual. In nominal induction the predicate to be proven has the extra argument $\mathcal{C}$, such that all bound names introduced by the induction rule are fresh for $\mathcal{C}$. Thus, the predicate has the form $\texttt{Prop}\ \mathcal{C}\ \Psi\ P\ Rs$. This induction rule is useful for very general proofs about transitions, but we often need proofs which are specialised for input, output, or tau transitions. We create the following custom induction rules:

Lemma 5.

$$\dfrac{\begin{array}{c}\Psi \rhd P \xrightarrow{\underline{M}\,N} P' \\ \vdots\end{array}}{\texttt{Prop}\,\mathcal{C}\,\Psi\,P\,M\,N\,P'} \quad\Bigg|\quad \dfrac{\begin{array}{c}\Psi \rhd P \xrightarrow{\overline{M}\,(\nu\widetilde{a})N} P' \\ \vdots\end{array}}{\texttt{Prop}\,\mathcal{C}\,\Psi\,P\,M\,((\nu\widetilde{a})(N \prec P'))} \quad\Bigg|\quad \dfrac{\begin{array}{c}\Psi \rhd P \xrightarrow{\tau} P' \\ \vdots\end{array}}{\texttt{Prop}\,\mathcal{C}\,\Psi\,P\,P'}$$

Proof. Follows immediately from the induction rule generated by Isabelle.

The inductive steps for each rule have been left out as they are instances of the ones from the automatically generated induction rule, but with the predicates changed to match the corresponding transition.

These induction rules work well only as long as the predicate to be proven does not depend on anything under the scope of a binder. Trying to prove the following lemma illustrates the problem.

Lemma 6. *If* $\Psi \rhd P \xrightarrow{\overline{M}\,(\nu\widetilde{a})N} P'$, $x\#P$ *and* $x\#\widetilde{a}$ *then* $x\#N$ *and* $x\#P'$

Proof. By induction over the transitions of the form $\Psi \rhd P \xrightarrow{\overline{M}\,(\nu\widetilde{a})N} P'$.

The problem is that none of the induction rules we have will prove this lemma in a satisfactory way. Every applicable case in the induction rule will introduce its own bound output term $(\nu\widetilde{b})N' \prec P''$ where we know that $(\nu\widetilde{b})N' \prec P'' = (\nu\widetilde{a})N \prec P'$. What we need to prove relates to the term P', what the inductive hypotheses will give us is something related to the term P'' where all we know is that they are part of alpha-equivalent terms.

Proving this lemma on its own is not too difficult but in every step of every proof of this type, manual alpha-conversions and equivariance properties are needed. The following induction rule solves this problem.

$$\dfrac{\begin{array}{c}\Psi \rhd P \xrightarrow{\overline{M}\,(\nu\widetilde{a})N} P' \\ \forall \Psi\ P\ M\ \widetilde{a}\ N\ P'\ \widetilde{b}\ p\ \mathcal{C}.\ \begin{pmatrix}\widetilde{a} \# \widetilde{b}, \Psi, P, M, \mathcal{C}\ \wedge\ \widetilde{b} \# N, P'\ \wedge \\ \texttt{set}\ p \subseteq \texttt{set}\ \widetilde{a} \times \texttt{set}\ \widetilde{b}\ \wedge \\ \texttt{Prop}\ \mathcal{C}\ \Psi\ P\ M\ \widetilde{a}\ N\ P' \longrightarrow \\ \texttt{Prop}\ \mathcal{C}\ \Psi\ P\ M\ \widetilde{b}\ (p \cdot N)\ (p \cdot P')\end{pmatrix} \\ \vdots\end{array}}{\texttt{Prop}\ \mathcal{C}\ \Psi\ P\ M\ \widetilde{a}\ N\ P'}$$

The difference between this rule and the output rule in Lemma 5 is that the predicate in Lemma 5 takes a residual $(\nu\widetilde{a})N \prec P'$ as one argument and the predicate in this rule takes $\widetilde{a}$, N and P' as three separate ones. By disassociating the binding sequence from the residual in this manner we have lost the ability to alpha-convert the residual, but we have gained the ability to reason about terms under the binding sequence. The extra added case in the induction rule above (beginning with $\forall \Psi\ P\ M\ \ldots$) is designed to allow the predicate to mimic the alpha-conversion abilities we have lost. When proving this induction rule, Lemma 3 is used in each step to generate the alpha-converting permutation, `Prop` is proven in the standard way and then alpha-converted using the new inductive case.

With this lemma, we must prove that the predicate we are trying to prove can respect alpha-conversions. The advantage is that it only has to be done once for each proof. Moreover, the case is very general and does not require the processes or actions to be of a specific form.

Using this induction rule will not allow us to prove lemmas which reason directly about the binding sequence $\widetilde{a}$. The new inductive case swaps a sequence $\widetilde{a}$ for $\widetilde{b}$ but as in Lemma 3, we do not know exactly how these sequences relate to each other.

Induction with frames A very common proof strategy in the psi-calculus is to do induction on a transition of a process which has a specific frame. Trying to prove the following lemma illustrates this.

Lemma 7. *If $\Psi \rhd P \xrightarrow{\underline{M}\,N} P'$, $\mathcal{F}(P) = \langle B_P, \Psi_P \rangle$, $X \# P$ and $B_P \# X, \Psi, P, M$ then there exists a K. s.t. $\Psi \otimes \Psi_P \vdash M \leftrightarrow K$ and $X \# K$*

Proof. By induction on the transition $\Psi \rhd P \xrightarrow{\underline{M}\,N} P'$. The intuition of the proof is that K is the subject in the process P.

This lemma suffers from the same problem as Lemma 6 – every inductive step will generate a frame alpha-equivalent to $\langle B_P,\ \Psi_P \rangle$ and many tedious alpha-conversions have to be done to prove the lemma. Moreover, some of our lemmas need to directly reason about the binding sequence of the frame. A similar induction rule as for output transitions can be created to solve the problem.

$$\dfrac{\begin{array}{l}\Psi \rhd P \xrightarrow{\underline{M}\, N} P' \\ \mathcal{F}(P) = \langle B_P,\ \Psi_P\rangle \\ \quad \texttt{distinct}\ B_P \\ \forall \Psi\ P\ M\ N\ P'\ B_P\ \Psi_P\ p\ \mathcal{C}. \left(\begin{array}{l}(p \cdot B_P) \# \Psi, P, M, \mathcal{C}, N, P', B_P\ \wedge B_P \# \Psi_P \wedge \\ \texttt{set}\ p \subseteq \texttt{set}\ B_P \times \texttt{set}(p \cdot B_P) \wedge \\ \texttt{Prop}\ \mathcal{C}\ \Psi\ P\ M\ N\ P'\ B_P\ \Psi_P \longrightarrow \\ \texttt{Prop}\ \mathcal{C}\ \Psi\ P\ M\ N\ P'\ (p \cdot B_P)\ (p \cdot \Psi_P)\end{array}\right) \\ \quad\quad \vdots\end{array}}{\texttt{Prop}\ \mathcal{C}\ \Psi\ P\ M\ N\ P'\ B_P\ \Psi_P}$$

This lemma requires that the binding sequence B_P is distinct. This added requirement allows the alpha converting case to relate the sequence B_P to $p \cdot B_P$ allowing for a larger class of lemmas to be proven. Our semantics require all frames to have distinct binding sequences making this added requirement unproblematic.

A corresponding lemma has to be created for output transitions as well, but since frames only affect subjects as far as input and output transitions are concerned, this induction rule does not have to use the same mechanism for the bound names in the residual as for the ones in the frame.

After introducing these custom induction rules, we were able to remove thousands of lines of code which were only dealing with alpha-conversions.

5 Conclusions and future work

Nominal Isabelle has proven to be a very potent tool when doing this formalisation. Its support for locales has made the formalisation of parametric calculi such as psi-calculi feasible and the nominal datatype package handles binders elegantly.

Psi-calculi require substantially more infrastructure than the pi-calculus [6]. The reason for this is mainly that binding sequences are a very new addition to the nominal package, and many of the automatic rules are not fully developed. Extending the support for binding sequences will require a fair bit of work, but we believe that the custom induction rules that we have designed can be created automatically as they do not use any intrinsic properties of psi-calculi.

We are currently working on extending our framework to include weak bisimulation and barbs. We also plan to work on typed psi calculi where we aim to make the type system as general and parametric as psi calculi themselves.

The source files for this formalisation can be found at:
`http://www.it.uu.se/katalog/jesperb/psi.tar.gz`.

Acknowledgments We want to convey our sincere thanks to Stefan Berghofer for his hard work on expanding the nominal package to include the features we have needed for this formalisation.

References

1. M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proceedings of POPL '01*, pages 104–115. ACM, Jan. 2001.
2. B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 3–15, New York, NY, USA, 2008. ACM.
3. C. Ballarin. Locales and locale expressions in isabelle/isar. In *TYPES*, pages 34–50, 2003.
4. H. P. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.
5. J. Bengtson, M. Johansson, J. Parrow, and B. Victor. Psi-calculi: Mobile processes, nominal data, and logic. Technical report, Uppsala University, 2009. Submitted. See `http://user.it.uu.se/~joachim/psi.pdf`.
6. J. Bengtson and J. Parrow. Formalising the pi-calculus using nominal logic. In *Proceedings of FoSSaCS 2007*, volume 4423 of *LNCS*, pages 63–77. Springer, 2007.
7. S. Berghofer and C. Urban. Nominal inversion principles. In *TPHOLs '08: Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, pages 71–85, Berlin, Heidelberg, 2008. Springer-Verlag.
8. M. G. Buscemi and U. Montanari. Open bisimulation for the concurrent constraint pi-calculus. In S. Drossopoulou, editor, *Proceedings of ESOP 2008*, volume 4960 of *LNCS*, pages 254–268. Springer, 2008.
9. N. G. de Bruijn. Lambda calculus notation with nameless dummies. a tool for automatic formula manipulation with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.
10. D. Hirschkoff. A full formalisation of pi-calculus theory in the calculus of constructions. In *TPHOLs '97: Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*, pages 153–169, London, UK, 1997. Springer-Verlag.
11. F. Honsell, M. Miculan, and I. Scagnetto. π-calculus in (co)inductive type theory. *Theoretical Comput. Sci.*, 253(2):239–285, 2001.
12. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
13. A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.
14. C. Röckl and D. Hirschkoff. A fully adequate shallow embedding of the π-calculus in Isabelle/HOL with mechanized syntax analysis. *J. Funct. Program.*, 13(2):415–451, 2003.
15. C. Urban. Nominal techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, May 2008.
16. C. Urban, S. Berghofer, and M. Norrish. Barendregt's variable convention in rule inductions. In *CADE-21: Proceedings of the 21st international conference on Automated Deduction*, pages 35–50, Berlin, Heidelberg, 2007. Springer-Verlag.