Revised version of T. F. Melham, 'A Mechanized Theory of the π-calculus in HOL', Technical Report No. 244, University of Cambridge Computer Laboratory (January 1992). Submitted for publication in the *Nordic Journal of Computing*.

A Mechanized Theory of the π-calculus in HOL

T. F. Melham

Department of Computing Science
University of Glasgow
Glasgow, Scotland, G12 8QQ

Abstract: *The π-calculus is a process algebra for modelling concurrent systems in which the pattern of communication between processes may change over time. This paper describes the results of preliminary work on a definitional formal theory of the π-calculus in higher order logic using the HOL theorem prover. The ultimate goal of this work is to provide practical mechanized support for reasoning with the π-calculus about applications.*

Introduction

The π-calculus [17, 18] is a process algebra proposed by Milner, Parrow and Walker for modelling concurrent systems in which the pattern of interconnection between processes may change over time. This paper describes work on a mechanized formal theory of the π-calculus in higher order logic using the HOL theorem prover [8]. The main aim of this work is to construct a practical and sound theorem-proving tool to support reasoning about applications using the π-calculus, as well as metatheoretic reasoning about the π-calculus itself.

Four general principles have been adopted in this project. First, a purely definitional approach is taken to describing the π-calculus in logic. New notation concerned with the π-calculus is added to the logic not by postulating arbitrary axioms to give meaning to it, but rather by defining it in terms of existing expressions of the logic that already have the required semantics. Second, proofs in the π-calculus are automated wherever feasible, with a view to eventually using the system to reason about applications. In practice, this means writing efficient derived inference rules in HOL for proving decidable classes of propositions, such as the α-equivalence of two terms in the calculus. The third principle is to make the HOL proofs as robust as possible, in the sense that they should run without major modification even when minor changes are made to the π-calculus itself. The hope is that this will facilitate experimental investigations in HOL of minor variants of the calculus. Finally, the π-calculus as mechanized in HOL is intended to be as nearly identical as possible to the calculus as described in the papers [17, 18]. The aim is to avoid simplifying the calculus merely in order to make the job of mechanizing it easier. One point at which we have compromised this last principle is discussed in section 4.3.

1 The HOL system

The HOL system [8] is a mechanized proof-assistant for generating proofs in higher order logic. HOL is based on the LCF approach to interactive theorem proving and has many features in common with the LCF systems developed at Cambridge [21] and Edinburgh [9]. Like LCF, the HOL system supports secure theorem proving by representing its logic in the strongly-typed functional programming language ML. Propositions and theorems of the logic are represented by abstract data types, and interaction with the theorem prover takes place by executing ML procedures that operate on values of these data types. Because HOL is built on top of a general-purpose programming language, the user can write arbitrarily complex programs to implement proof strategies. Furthermore, because of the way the logic is represented in ML, such user-defined proof strategies are guaranteed to perform only valid logical inferences.

1.1 Higher order logic

The version of higher order logic supported by the HOL theorem prover is based on Church's formulation of simple type theory [4]. For the purposes of this paper, the logic can be viewed as a typed extension of the conventional syntax of predicate calculus in which functions may be curried and one may quantify over functions. The notation is illustrated by the theorem shown below.

$$\vdash \forall x\, f.\, \exists fn.\, (fn\ 0 = x) \wedge \forall n.\, fn{:}num{\rightarrow}num\ (n{+}1) = (f\ (fn\ n))\ n$$

This says that functions can be defined on the natural numbers such that they satisfy primitive-recursive defining equations (the quantified variables f and fn range over functions). We adopt the convention that italic identifiers (e.g. x, x_1, fn) are variables and sans serif identifiers (e.g. a, F, Tau) and non-alphabetical symbols (e.g. $\supset$, $=$, $\forall$) are constants.

The HOL logic extends Church's formulation in two significant ways: the syntax of types includes the polymorphic type discipline developed by Milner for the LCF logic PPλ [9], and the primitive basis of the logic includes explicitly-stated rules of definition for consistently extending the logic with new constants and new types. Because this second feature of the logic is particularly relevant to the approach taken to embedding the π-calculus in HOL, the rules of definition in the HOL logic are very briefly introduced below. A full description of these rules and details of the rest of the logic, including a set-theoretic semantics, can be found in [8].

1.1.1 Primitive rules of definition

The HOL user community has a strong tradition of taking a purely definitional approach to using higher order logic, and this is the way in which the logic is used in the present work on the π-calculus. The advantage of this approach, as opposed to the axiomatic method, is that the primitive rules of definition admit only sound extensions to the logic, in the sense that they preserve the property of the logic having a (standard) model. Making definitions is therefore guaranteed not to introduce inconsistency. The disadvantage is that these rules admit only definitions that satisfy certain very restrictive rules of formation. Definitions expressed in any other form must always be justified by deriving them from equivalent, but possibly rather complex, primitive definitions.

The primitive basis of the HOL logic includes three rules of definition: the rule of *constant definition*, the rule of *constant specification*, and the rule of *type definition*. A constant definition is simply an equational axiom of the form $\vdash \mathsf{c} = t$ that introduces a new constant c as an object-language abbreviation for a closed term t. Also admitted by this rule are curried or paired function definitions of the forms

$$\vdash \mathsf{f}\ v_1\ v_2\ \ldots\ v_n = t \qquad \text{and} \qquad \vdash \mathsf{f}(v_1, v_2, \ldots, v_n) = t$$

Among the side-conditions of the rule of constant definition, the details of which are not relevant here, is the condition that the constant being defined may not occur on the right-hand side of its defining equation. This rules out, at least as primitive, all recursive definitions—including inconsistent ones like $\vdash \mathsf{c} = \neg\mathsf{c}$.

The rule of constant specification allows one to introduce a new constant into the logic as an atomic name for a quantity already known to exist. By this rule of definition, one may infer from a theorem of the form $\vdash \exists x.\, P[x]$ a theorem $\vdash P[\mathsf{c}]$, where c is a new constant symbol. This simply introduces c as an object-language name for an existing value x for which $P[x]$ holds.

The third primitive rule of definition in HOL is the rule of type definition. Suppose that σ is a type and $P{:}\sigma{\to}\mathit{bool}$ is the characteristic predicate of some useful nonempty subset of the set denoted by σ. A type definition introduces a new type constant τ to name this subset of σ. From the theorem $\vdash \exists x{:}\sigma.\, P\ x$, one may infer by the rule of type definition the existence of a bijection from the values of a new type τ to the set of values that satisfy P:

$$\vdash \exists f{:}\tau{\to}\sigma.\, (\forall x\ y.\, (f\ x = f\ y) \supset (x = y)) \wedge (\forall x.\, P\ x = (\exists y.\, x = f\ y))$$

This definitional theorem introduces the new type constant τ to name the nonempty set of values whose properties are determined by the choice of predicate P. The requirement that $\vdash \exists x.\, P\ x$ ensures that there is at least one value of type τ. This restriction is necessary because the HOL logic does not allow empty types. The rule of type definition can also be used to define new type operators; one can define, for example, the Cartesian product $\sigma_1 \times \sigma_2$ of two types, or the type of finite lists $(\alpha)\mathit{list}$. See [12] for a series of detailed examples.

1.1.2 Derived rules of definition

The primitive rules outlined above disallow the direct use of many commonly-used principles of definition—for example, the definition of functions by primitive recursion. The general-purpose language ML, however, provides a facility in HOL for implementing derived rules of definition; using ML, one can write programs that automatically generate the proofs that justify the legitimacy of derived forms of definition. The built-in HOL derived rules of definition include recursive concrete type definitions and primitive recursive function definitions over these types, as well as certain forms of inductive definition. The details of the primitive definitions that underlie these rules are hidden from the user, and their ML implementations are highly optimized. So these derived principles of definition may just be regarded as primitive by most users of the system.

The HOL mechanization of the π-calculus is a purely definitional theory in higher order logic. It relies heavily on the derived principles of definition available in HOL, which are therefore briefly explained as they used in the sections that follow. Details of these derived rules can be found in the HOL system documentation [8] or the papers [12, 14].

2 A sketch of the π-calculus

This section provides a summary overview of the π-calculus in just enough detail for a reader familiar with (for example) CCS [15] to follow the HOL mechanization described in later sections. For full details of the π-calculus and for motivational discussion, the reader should consult the papers by Milner, Parrow and Walker [17, 18]. The summary presented here is based on the material in these papers, and of course no claim to originality in respect of the ideas in this section is made by the present author.

2.1 Syntax of the calculus

Let $\mathcal{N}$ be an infinite set of *names*, which in the π-calculus are used both as variables and as data values, as well as names of the ports or communication links between processes. The syntax of *agents* in the π-calculus is defined by

$$
\begin{array}{llll}
P & ::= & \mathbf{0} & \text{inaction} \\
 & \mid & \overline{x}y.P & \text{output } y \text{ on } x \text{ then } P \\
 & \mid & x(y).P & \text{input } z \text{ on } x \text{ then } P\{z/y\} \\
 & \mid & \tau.P & \text{do silent } \tau \text{ then } P \\
 & \mid & (x)P & \text{restrict scope of } x \\
 & \mid & [x{=}y]P & \text{if } x = y \text{ then } P \text{ else } \mathbf{0} \\
 & \mid & P_1 \mid P_2 & P_1 \text{ and } P_2 \text{, in parallel} \\
 & \mid & P_1 + P_2 & P_1 \text{ or } P_2 \\
 & \mid & A(x_1, \ldots, x_n) & \text{defined agent}
\end{array}
$$

where P, P_1, P_2 range over agents, x, x_1, ..., x_n, y range over names, and A ranges over n-ary *agent identifiers*. The forms $x(y).P$ and $(y)P$ introduce variable binding into the calculus; the prefixes '$x(y)$' and '(y)' bind the name y in P. If an occurrence of a name y is not bound, it is called *free*. The set of names that occur free in an agent P is written $\mathrm{fn}(P)$, and the set of names bound in an agent P is written $\mathrm{bn}(P)$. The set of names of an agent P, written $\mathrm{n}(P)$, is defined to be the union of $\mathrm{fn}(P)$ and $\mathrm{bn}(P)$.

Agent identifiers provide the π-calculus with both abbreviations for classes of agents and recursion. Each n-ary agent identifier A is equipped with a defining equation of the form

$$A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P$$

where the set of all names that appear free in P is a subset of $\{x_1, \ldots, x_n\}$. A defining equation or a set of such equations may be recursive and hence may introduce agents with infinite behaviour.

The meaning of agents is very briefly summarized as follows. The agent $\mathbf{0}$ does nothing. The agent $\overline{x}y.P$ emits the name y on the output port x and then behaves like P. The agent $x(y).P$ receives a name z on the input port x and then behaves like $P\{z/y\}$, where '$P\{z/y\}$' denotes the result of substituting z for every free occurrence of y in P, with change of bound names if necessary to avoid capture of z. The agent $\tau.P$ performs the silent action τ and then behaves like P. In the agent $(x)P$, the name x is made local to P by the binding prefix '(x)'. The agent $[x{=}y]P$ behaves like P if x and y are the same name and otherwise behaves like $\mathbf{0}$. As in CCS, the agent $P_1 \mid P_2$ represents the parallel composition of P_1 and P_2, and the agent $P_1 + P_2$ behaves like either P_1 or P_2. Finally, the defined agent $A(x_1, ..., x_n)$ behaves like the corresponding instance of the right-hand side of the defining equation for the n-ary agent identifier A.

2.2 The transitional semantics

As in CCS, agents in the π-calculus are given a transitional semantics based on labelled transitions of the form $P \xrightarrow{\alpha} Q$, which can be read 'P can perform the action α and then evolve into Q'. There are four types of action:

$$
\begin{array}{llll}
A & ::= & \tau & \text{silent action} \\
 & \mid & \overline{x}y & \text{free output action} \\
 & \mid & x(y) & \text{input action} \\
 & \mid & \overline{x}(y) & \text{bound output action}
\end{array}
$$

The silent action arises from internal communication within an agent, as well as from agents of the form $\tau.P$. Output-prefixed agents such as $\overline{x}y.P$ give rise to the free output action $\overline{x}y$, and input prefixed agents $x(y).P$ to the input action $x(y)$. Bound output actions of the form $\overline{x}(y)$ arise from output actions that export a name outside its current scope.

$$\text{TAU-ACT:}\quad \frac{}{\tau.P \xrightarrow{\tau} P} \qquad\qquad \text{OUTPUT-ACT:}\quad \frac{}{\overline{x}y.P \xrightarrow{\overline{x}y} P}$$

$$\text{INPUT-ACT:}\quad \frac{}{x(z).P \xrightarrow{x(w)} P\{w/z\}} \quad w \notin \mathrm{fn}((z)P)$$

$$\text{SUM:}\quad \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'} \qquad\qquad \text{MATCH:}\quad \frac{P \xrightarrow{\alpha} P'}{[x{=}x]P \xrightarrow{\alpha} P'}$$

$$\text{IDE:}\quad \frac{P\{y_1,\ldots,y_n/x_1,\ldots,x_n\} \xrightarrow{\alpha} P'}{A(y_1,\ldots,y_n) \xrightarrow{\alpha} P'} \quad A(x_1,\ldots,x_n) \stackrel{\mathrm{def}}{=} P$$

$$\text{PAR:}\quad \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \quad \mathrm{bn}(\alpha) \cap \mathrm{fn}(Q) = \{\}$$

$$\text{COM:}\quad \frac{P \xrightarrow{\overline{x}y} P' \quad Q \xrightarrow{x(z)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'\{y/z\}} \qquad\qquad \text{CLOSE:}\quad \frac{P \xrightarrow{\overline{x}(w)} P' \quad Q \xrightarrow{x(w)} Q'}{P \mid Q \xrightarrow{\tau} (w)(P' \mid Q')}$$

$$\text{RES:}\quad \frac{P \xrightarrow{\alpha} P'}{(y)P \xrightarrow{\alpha} (y)P'} \quad y \notin \mathrm{n}(\alpha)$$

$$\text{OPEN:}\quad \frac{P \xrightarrow{\overline{x}y} P'}{(y)P \xrightarrow{\overline{x}(w)} P'\{w/y\}} \quad \begin{array}{l} y \neq x \\ w \notin \mathrm{fn}((y)P') \end{array}$$

Figure 1: Transition rules for the π-calculus.

The following notation, which is introduced in [18], is used in defining the transition relation for the π-calculus. The set of bound names of an action α is written $\mathrm{bn}(\alpha)$, and the set of free names of an action α is written $\mathrm{fn}(\alpha)$. The meaning of this notation is defined by

$$\mathrm{bn}(\alpha) = \begin{cases} \{y\} & \text{if } \alpha = x(y) \text{ or } \overline{x}(y) \\ \{\} & \text{otherwise} \end{cases}$$

and

$$\mathrm{fn}(\alpha) = \begin{cases} \{x\} & \text{if } \alpha = x(y) \text{ or } \overline{x}(y) \\ \{x, y\} & \text{if } \alpha = \overline{x}y \\ \{\} & \text{if } \alpha = \tau \end{cases}$$

The set of names of an action $\mathrm{v}(\alpha)$ is defined to be the union of $\mathrm{bn}(\alpha)$ and $\mathrm{fn}(\alpha)$. An expression of the form '$P\{y_1,\ldots,y_n/x_1,\ldots,x_n\}$' denotes the result of simultaneously substituting y_i for x_i for $1 \leq i \leq n$ in P, with change of bound names as required to avoid capture.

The transition relation $P \xrightarrow{\alpha} Q$ itself is defined inductively by the rules shown in figure 1, together with additional symmetric rules for the operators $|$ and $+$. More precisely, the three-place relation $\longrightarrow \subseteq (agent \times action \times agent)$ is defined to be the smallest set closed under these rules, where '$(P, \alpha, Q) \in \longrightarrow$' is written '$P \xrightarrow{\alpha} Q$'. The details of the transition rules are not relevant here; they are shown in full merely to give the reader a general idea of the size and complexity of the calculus.

2.3 Bisimulation and Equivalence

As in CCS, equivalence of agents in the π-calculus is defined using the notion of a bisimulation between agents. A binary relation S is a *strong simulation* if $P\ S\ Q$ implies that

1. If $P \xrightarrow{\alpha} P'$ and $\alpha = \tau$ or $\alpha = \overline{x}y$, then for some Q', $Q \xrightarrow{\alpha} Q'$ and $P'\ S\ Q'$.

2. If $P \xrightarrow{x(y)} P'$ and $y \notin \mathrm{n}(P) \cup \mathrm{n}(Q)$, then for some Q', $Q \xrightarrow{x(y)} Q'$ and for all w, $P'\{w/y\}\ S\ Q'\{w/y\}$.

3. If $P \xrightarrow{\overline{x}(y)} P'$ and $y \notin \mathrm{n}(P) \cup \mathrm{n}(Q)$, then for some Q', $Q \xrightarrow{\overline{x}(y)} Q'$ and $P'\ S\ Q'$.

A *strong bisimulation* is a strong simulation S whose inverse is also a strong simulation. The relation $\dot{\sim}$ is defined to be the largest strong bisimulation, so that two agents P and Q are *strongly bisimilar* (written '$P \dot{\sim} Q$') if $P\ S\ Q$ for some strong bisimulation S.

The algebraic theory of bisimilarity for agents in the π-calculus is based on the definitions given above. Many of the algebraic laws correspond to similar or identical laws in CCS. For example the following equations for summation hold:

$$\begin{array}{rcll} P + \mathbf{0} & \dot{\sim} & P & \text{zero} \\ P + P & \dot{\sim} & P & \text{idempotence} \\ P_1 + P_2 & \dot{\sim} & P_2 + P_1 & \text{commutativity} \\ P_1 + (P_2 + P_3) & \dot{\sim} & (P_1 + P_2) + P_3 & \text{associativity} \end{array}$$

The equational theory also includes an analogue to the CCS expansion law.

Strong bisimilarity is not preserved by substitution of names for free names. Equivalence is therefore defined to be strong bisimilarity under all substitutions. For any substitution $\sigma : \mathcal{N} \to \mathcal{N}$, '$P\sigma$' denotes the result of simultaneously substituting $\sigma(z)$ for all free z in the agent P, changing bound variables as necessary to avoid captures. Two agents P and Q are *strongly equivalent* (written '$P \sim Q$') if $P\sigma \dot{\sim} Q\sigma$ for all substitutions σ. The algebraic theory of equivalence is similar (but not identical) to the theory of bisimilarity.

For a presentation of the full algebraic theory, detailed proofs of soundness and of completeness for finite agents, and for a discussion of other notions of equivalence for the π-calculus, see the papers [17, 18].

3 Mechanizing the π-calculus in HOL

One possible approach to mechanizing a formal system in HOL is to translate its syntactic objects directly into appropriate denotations in higher order logic. This approach is exemplified by Mike Gordon's work on mechanizing Hoare logic [7]. Meaning is given to partial correctness statements in Hoare logic by translating them into propositions of higher order logic that capture their intended semantics. For example, the partial correctness statement

{X=n} X:=X+1 {X=n+1}

is translated into the assertion that the following relation holds of any pair of initial and final states s_1 and s_2:

$$\forall n.\ ((s_1\ \mathtt{X} = n) \wedge (s_2 = \lambda v.\ (v{=}\mathtt{X} \Rightarrow (s_1\ \mathtt{X}) + 1 \mid s_1\ v))) \supset (s_2\ \mathtt{X} = n + 1)$$

Program variables (e.g. X) are represented by constants of a specially-defined logical type *var*, and states are modelled by total functions from program variables to natural numbers. Partial correctness statements are represented directly by their denotations in logic; with sufficient parser and pretty-printer support, these can be made to look like assertions in Hoare logic (see [7]).

The advantage of this approach is that the embedded formal system inherits a certain amount of syntactic infrastructure from the underlying logic. For example, λ-abstraction and β-reduction in higher order logic can be used to simulate variable binding and substitution in the language being mechanized. The result is a system particularly well suited to reasoning about applications, since the HOL system provides highly optimized proof support for these basic syntactic notions. This is sometimes called the *shallow embedding* approach to mechanizing another formal system in HOL [2].

The disadvantage of direct translation is that it does not allow metatheoretic reasoning about the embedded formal system to be carried out within higher order logic itself. For example, a proposition that makes reference to the embedded language as a whole cannot be expressed in the logic; it can be stated only as a metatheorem about classes of logical assertions and hence cannot be proved in HOL.

One goal of the present work is to support formal metatheoretic reasoning about the π-calculus itself, as well as reasoning in the calculus about applications. For example, one might wish to prove properties of a programming language semantics given by a translation into the π-calculus; such properties are typically meta-theoretical in nature with respect to the target language and its semantics. A different approach is therefore taken to mechanizing the π-calculus in HOL. The language of agents is embedded as an object (or, more specifically, as a defined type) within the logic, rather than metalinguistically translated into terms of the logic. Higher order logic is thus used as a formal metalanguage whose objects are the process or agent expressions of the π-calculus. Meaning is then given to these expressions by defining the labelled transition semantics, strong bisimulation, and the relations $\dot{\sim}$ and $\sim$ within the logic itself. This is an example of a so-called *deep embedding* of a formal system in HOL.

A similar approach is taken by Camilleri [3] in his formalization of CSP in higher order logic, by Back and von Wright [1] in their work on mechanized program transformation in HOL, and in the present author's work on reasoning about circuit models [14]. All this work, however, contrives to avoid explicit definitions of substitution, essentially by inheriting it from higher order logic. In this respect, it differs from the present formalization of the π-calculus, in which all syntactic operations over the embedded language of agent expressions are defined within the logic and can therefore be mentioned explicitly in propositions of the logical metalanguage.

4 Embedding the syntax of agents in HOL

This section outlines a definitional HOL theory of the language of agent expressions in the π-calculus. For clarity of notation, as well as for fidelity to the presentation in [17, 18], the theory makes use of a predefined logical type $(\alpha)set$, values of which are sets of elements of type α. This type is defined formally in the built-in HOL 'set theory' library, which contains a substantial collection of basic theorems about sets. Also provided by the library are parser and pretty-printer support for naming finite sets by enumeration, for example by writing '$\{a, b, c\}$', and for the set specification notation '$\{x \mid \phi(x)\}$'. Sets written in these notations should be regarded as metalinguistic abbreviations; they are expanded by the HOL term parser into logical terms that denote the appropriate values.

4.1 Representing names in logic

An obvious way to represent names in higher order logic is to model the set of names $\mathcal{N}$ by a logical type. The only property required of $\mathcal{N}$ is that it must be infinite, so that bound names can always be changed to avoid capture of names by the binding constructs '$x(y).P$' and '$(y)P$' when a substitution is done. Names can therefore be represented in logic by any type that contains an infinite number of distinct values, for example the type of natural numbers.

But rather than develop the theory with a particular fixed representation for names, the set of names $\mathcal{N}$ is represented by a type variable 'α'. An infinite set of names is then assumed by working (when necessary) under the hypothesis that there exists a choice function $ch{:}(\alpha)set{\to}\alpha$ which for any finite set of names S yields a name not in S:

$$\forall S.\ \mathsf{Finite}\ S \supset \neg(ch\ S \in S)$$

This *infinity hypothesis* is required only for the proofs of certain theorems about the π-calculus whose truth depends on the ability to change bound names during substitution. The assumption that there exists a choice function ch with the above property is provably equivalent in HOL to the alternative hypothesis

$$\exists f{:}\alpha{\to}\alpha.\ (\forall x\ y.\ (f\ x = f\ y) \supset (x = y)) \wedge (\exists y.\ \forall x.\ \neg(f\ x = y))$$

This asserts of the type α that it has no more elements than some proper subset of α. That is, it asserts of α that it satisfies the conventional definition of an infinite set.

Using a type variable to represent the set of names results in a 'polymorphic' theory of the π-calculus in HOL. The entire theory can be specialized for a particular application by choosing an (infinite) application-specific logical type to model names, instantiating the type variable α to this type, and discharging the resulting infinity hypothesis wherever it appears. This is not an atomic operation in the HOL system, but it is not hard to program in ML. The only part that cannot be automated is proving the existence of a choice function for the type selected to represent names.

4.2 Defining the syntax of agents

The formal language of agents in the π-calculus is embedded in HOL by defining a logical type $(\alpha)agent$, values of which represent agent expressions with names of type α. The primitive rule of type definition, as was already mentioned, allows new types to be introduced into the logic only as names for subsets of already existing types. So to define a type of agent expressions a rather complex encoding into values of an existing logical type is required.

The HOL system, however, provides a derived principle of definition that automates all the formal inference necessary to define an arbitrary concrete recursive type in higher order logic [12]. The user supplies a specification of the required type in a form similar to a datatype declaration in Standard ML [22]. The system then constructs an appropriate encoding for values of the required type, defines the type using this encoding and the primitive rule of type definition, and automatically proves an abstract characterization of the newly-defined type. The details of the definition are hidden from the user, who may regard this derived principle of *recursive type definition* just as if it were primitive.

Using the derived rule of recursive type definition, the language of agent expressions is embedded in logic by the type $(\alpha)agent$ specified by

$$
\begin{array}{lcll}
agent & ::= & \mathsf{Zero} & \mathsf{Zero} \text{ represents } \mathbf{0} \\
 & | & \mathsf{Out}\ \alpha\ \alpha\ agent & \mathsf{Out}\ x\ y\ P \text{ represents } \overline{x}y.P \\
 & | & \mathsf{In}\ \alpha\ \alpha\ agent & \mathsf{In}\ x\ y\ P \text{ represents } x(y).P \\
 & | & \mathsf{Tau}\ agent & \mathsf{Tau}\ P \text{ represents } \tau.P \\
 & | & \mathsf{Res}\ \alpha\ agent & \mathsf{Res}\ x\ P \text{ represents } (x)P \\
 & | & \mathsf{Match}\ \alpha\ \alpha\ agent & \mathsf{Match}\ x\ y\ P \text{ represents } [x{=}y]P \\
 & | & \mathsf{Comp}\ agent\ agent & \mathsf{Comp}\ P_1\ P_2 \text{ represents } P_1 \mid P_2 \\
 & | & \mathsf{Plus}\ agent\ agent & \mathsf{Plus}\ P_1\ P_2 \text{ represents } P_1 + P_2 \\
 & | & \mathsf{Repl}\ agent & \mathsf{Repl}\ P \text{ represents } !P
\end{array}
$$

This equation specifies a concrete recursive type with nine constructors, each of which (except Repl, which is explained later) corresponds to one of the forms of agent expression in the π-calculus syntax presented in section 2.1. Given this specification, the rule of recursive type definition automatically finds a representation for the required type $(\alpha)agent$ and makes an appropriate primitive type definition for it. The system also makes an appropriate constant definition for each of the specified constructors. Plus, for example, becomes a constant of type

$$(\alpha)agent\to(\alpha)agent\to(\alpha)agent$$

introduced by means of the primitive rule of constant definition. Likewise, Res becomes a constant function that maps a value of type α representing a name to a value of type $(\alpha)agent$ representing an agent, and so on.

The result is a single theorem which provides a complete and abstract characterization of the type $(\alpha)agent$ and forms the basis for all further reasoning about it. The theorem asserts the admissibility of defining functions over agents by primitive recursion:

$$
\begin{array}{l}
\vdash \forall e\ f_0\ f_1\ f_2\ f_3\ f_4\ f_5\ f_6\ f_7. \\
\quad \exists! fn{:}(\alpha)agent\to\beta. \\
\qquad fn\ \mathsf{Zero} = e\ \wedge \\
\qquad \forall x_0\ x_1\ a.\ fn(\mathsf{Out}\ x_0\ x_1\ a) = f_0\ (fn\ a)\ x_0\ x_1\ a\ \wedge \\
\qquad \forall x_0\ x_1\ a.\ fn(\mathsf{In}\ x_0\ x_1\ a) = f_1\ (fn\ a)\ x_0\ x_1\ a\ \wedge \\
\qquad \forall a.\ fn(\mathsf{Tau}\ a) = f_2\ (fn\ a)\ a\ \wedge \\
\qquad \forall x\ a.\ fn(\mathsf{Res}\ x\ a) = f_3\ (fn\ a)\ x\ a\ \wedge \\
\qquad \forall x_0\ x_1\ a.\ fn(\mathsf{Match}\ x_0\ x_1\ a) = f_4\ (fn\ a)\ x_0\ x_1\ a\ \wedge \\
\qquad \forall a_1\ a_2.\ fn(\mathsf{Comp}\ a_1\ a_2) = f_5\ (fn\ a_1)\ (fn\ a_2)\ a_1\ a_2\ \wedge \\
\qquad \forall a_1\ a_2.\ fn(\mathsf{Plus}\ a_1\ a_2) = f_6\ (fn\ a_1)\ (fn\ a_2)\ a_1\ a_2\ \wedge \\
\qquad \forall a.\ fn(\mathsf{Repl}\ a) = f_7\ (fn\ a)\ a
\end{array}
$$

This is an abstract characterization of the language of agents in logic which is both succinct and complete, in the sense that it completely determines the structure of agent expressions up to isomorphism. It can be viewed as slight extension of the *initiality* property by which structures are characterized in the 'initial algebra' approach to specifying abstract data types [5].

4.2.1 Primitive recursion and induction over agents

As was discussed in section 1.1.1, function constants that satisfy recursive defining equations are not directly definable by the primitive rule for constant definitions. To define such a constant, one must first prove that there in fact exists a total function that satisfies the required recursive equation. The HOL system, however, has a built-in derived principle of *primitive recursive function definition*, which automates existence proofs for primitive recursive functions defined over concrete recursive types such as $(\alpha)agent$.

Given the characterizing theorem for $(\alpha)agent$ and the primitive recursive defining equations for a function over agents, this rule automatically proves the existence of a total function that satisfies these equations. A constant is then introduced by a constant specification to name this total function. The details of the proofs are hidden from the user, who for all practical purposes can simply regard this derived principle of recursive function definition as part of the primitive basis of the logic.

The HOL system also has a built-in derived inference rule for proving a structural induction theorem for any concrete recursive type. Given the recursion theorem for $(\alpha)agent$ shown above, this rule automatically proves a theorem that states the validity of structural induction on agent expressions. This induction theorem can, in turn, be used with another built-in proof tool to automatically construct a HOL *tactic* for interactive goal-directed proofs by structural induction on agents. (See any one of [8, 9, 21] for an explanation of tactics.) As one might expect, this tactic is invaluable for proving many of the basic syntactic theorems about the π-calculus in HOL.

4.3 Agent identifiers and replication

The π-calculus syntax shown in section 2.1 includes defined agents of the form $A(x_1, \ldots, x_n)$, where A is an n-ary agent identifier. Agent identifiers, together with their defining equations, supply the π-calculus both with object-language abbreviations for agent expressions and with recursion. The latter is the essential function of agent identifiers; without them, there is no way to express infinite behaviour. In the HOL mechanization, however, agent identifiers are replaced by an alternative way of providing unbounded behaviour, namely the *replication* of agents. This difference represents the only significant point at which the principle that the HOL theory should be as close as possible to the calculus as presented in [17, 18] has been compromised.

The replication of an agent P is written '$!P$' and is represented in logic by '$\mathsf{Repl}\ P$'. The agent $!P$ can be thought of as the parallel composition of as many instances of P as desired. Informally,

$$!P = \underbrace{P \mid P \mid \cdots \mid P}_{n \text{ copies}} \mid !P$$

This is reflected in the following transition rule for replication

$$\text{REPL:} \quad \frac{P \mid !P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}$$

which states that whatever action can be performed by the parallel composition of an agent P with the replication $!P$ can also be done by the replication $!P$ itself. In the HOL theory of the π-calculus, this rule replaces the agent identifier rule IDE shown in figure 1.

Replacing agent identifiers by replication considerably simplifies the HOL mechanization; it avoids the need to parameterize the entire theory by sets of defining equations and to work under well-formedness hypotheses for these equations. But for many applications, recursive agent definitions are likely to be more direct and natural to use than replication. The theory aims, therefore, to recover at least some of the utility of agent identifiers. The merely abbreviatory role of agent identifiers can just be transferred to ordinary constant definitions in the logic. But the expressive power of recursive defining equations can be regained only at the cost of developing some special-purpose proof support. The aim is eventually to support recursive agent definitions by a method similar to that by which recursive function definitions are automated in HOL. Preliminary experiments indicate that this approach is feasible, but to date little work has been done on this in the HOL mechanization. For an explanation of how at least some recursive definitions can be encoded using replication, see Milner's tutorial [16].

Using replication departs from the formulation in the original exposition of the π-calculus. In subsequent work, however, replication has in any case largely replaced agent identifier definitions as the chosen primitive for infinite behaviours. Replication is used in the *polyadic* π-calculus [16], a generalization allowing simultaneous communication of several names, as well as more recent formulations of the original π-calculus.

4.4 Elementary syntactic theory

Having defined the type $(\alpha)agent$ in logic, it is straightforward, if somewhat tedious, to develop the elementary theory of the syntax of agents in HOL. This comprises the various definitions and theorems about free and bound names, substitution, and α-equivalence of agents needed for later proofs—matters that are covered in a mere page or so in the paper [18], but which naturally take considerably longer to treat formally. The following sections outline the HOL theory of free and bound names and substitution; the definition of α-equivalence is omitted.

4.4.1 Free and bound names

Development of the theory begins with defining the function constants Fn, Bn and N. These have the logical type $(\alpha)agent{\to}(\alpha)set$ and correspond to the functions fn, bn and n described above in section 2.1. The definitions use some of the infrastructure provided by the HOL set theory library, namely the basic operations of set union and set difference, as well as notation for specifying finite sets by enumeration. The functions themselves are primitive recursive over the type of agent expressions $(\alpha)agent$. They can therefore be defined simply by supplying the required defining equations to the HOL derived rule of recursive function definition. The recursive definition of Fn, for example, is given by the theorem

$$
\begin{array}{l}
\vdash \mathsf{Fn}\ \mathsf{Zero} = \{\} \wedge \\
\quad \forall x\ y\ P.\ \mathsf{Fn}(\mathsf{Out}\ x\ y\ P) = \{x, y\} \cup (\mathsf{Fn}\ P) \wedge \\
\quad \forall x\ y\ P.\ \mathsf{Fn}(\mathsf{In}\ x\ y\ P) = \{x\} \cup ((\mathsf{Fn}\ P) - \{y\}) \wedge \\
\quad \forall P.\ \mathsf{Fn}(\mathsf{Tau}\ P) = \mathsf{Fn}\ P \wedge \\
\quad \forall x\ P.\ \mathsf{Fn}(\mathsf{Res}\ x\ P) = (\mathsf{Fn}\ P) - \{x\} \wedge \\
\quad \forall x\ y\ P.\ \mathsf{Fn}(\mathsf{Match}\ x\ y\ P) = \{x, y\} \cup (\mathsf{Fn}\ P) \wedge \\
\quad \forall P\ Q.\ \mathsf{Fn}(\mathsf{Comp}\ P\ Q) = (\mathsf{Fn}\ P) \cup (\mathsf{Fn}\ Q) \wedge \\
\quad \forall P\ Q.\ \mathsf{Fn}(\mathsf{Plus}\ P\ Q) = (\mathsf{Fn}\ P) \cup (\mathsf{Fn}\ Q) \wedge \\
\quad \forall P.\ \mathsf{Fn}(\mathsf{Repl}\ P) = \mathsf{Fn}\ P
\end{array}
$$

which is proved automatically by this derived rule, as outlined above in section 4.2.1. The definitions of Bn and N are similar.

A collection of theorems about free and bound names in the π-calculus has been proved in HOL from the definitions of Fn, Bn and N . These theorems are mostly very simple and their proofs trivial; two illustrative examples are

$$
\begin{array}{l}
\vdash \forall P.\ \mathsf{Finite}(\mathsf{Fn}\ P) \\
\vdash \forall P.\ \mathsf{N}\ P = \mathsf{Fn}\ P \cup \mathsf{Bn}\ P
\end{array}
$$

Both theorems are proved by structural induction on the agent P using the tactic discussed above in section 4.2.1. The significance of the first theorem has to do with the need to change bound names to avoid capture during substitution. A fresh name is sometimes needed, distinct from all the names free in a given agent P, and this is possible only if the set $\mathsf{Fn}(P)$ is finite. The second theorem merely states that the function N, which is defined recursively in HOL, satisfies the more direct definition used in [18].

4.4.2 Substitution

One of the more complex definitions in the syntactic theory is the definition of simultaneous substitution of names for free occurrences of names in an agent. The complexity is due, of course, to the name binding constructs of the π-calculus. Bound names sometimes have to be changed to avoid the capture of names introduced by substitution. Furthermore, the present theory of substitution is designed with future use for applications in mind, so bound names are changed only when strictly necessary. This further complicates the definition.

To formalize substitution for the π-calculus in logic, a function

$$\mathsf{Sub} : \underbrace{(\alpha{\to}(\alpha)set{\to}\alpha)}_{\text{choice function}} \to \underbrace{(\alpha)agent}_{\text{agent}} \to \underbrace{(\alpha{\to}\alpha)}_{\text{name mapping}} \to \underbrace{(\alpha)agent}_{\text{result}}$$

is defined by primitive recursion on agents. The function Sub takes two arguments in addition to the agent in which the substitution is to be done. One is a name mapping $s{:}\alpha{\to}\alpha$, which specifies the particular substitution of names for names required. The other argument is a choice function $ch{:}\alpha{\to}(\alpha)set{\to}\alpha$, which is used in the body of the definition of substitution to generate fresh names wherever a change of bound names is required. It is assumed that the choice function has the property that for any name n and finite set of names S, the name $ch\ n\ S$ is not an element of S. This is expressed by

$$\forall S.\ \mathsf{Finite}\ S \supset \forall n.\ \neg(ch\ n\ S \in S)$$

which is taken as a hypothesis, if necessary, in proofs involving substitution—as was discussed above in section 4.1. In general, the choice function is assumed to take both a name n and a set S as arguments. This is done so that application-specific instances of the choice function can, if desired, generate a name not in S by taking some variant of the name n.

The primitive recursive definition of Sub in HOL is given by the theorem shown below. The notation '`let` $v = t_1$ `in` t_2' used in this definition is a metalinguistic abbreviation supported by the HOL parser and pretty-printer. It expands into a term provably equivalent to $(\lambda v.\, t_2)\ t_1$.

$$
\begin{array}{l}
\vdash \forall ch\ s.\ \mathsf{Sub}\ ch\ \mathsf{Zero}\ s = \mathsf{Zero}\ \wedge \\
\quad \forall ch\ x\ y\ P\ s.\ \mathsf{Sub}\ ch\ (\mathsf{Out}\ x\ y\ P)\ s = \mathsf{Out}\ (s\ x)\ (s\ y)\ (\mathsf{Sub}\ ch\ P\ s) \wedge \\
\quad \forall ch\ x\ y\ P\ s.\ \mathsf{Sub}\ ch\ (\mathsf{In}\ x\ y\ P)\ s\ = \\
\qquad\qquad\qquad \mathtt{let}\ vs = \mathsf{Image}\ s\ ((\mathsf{Fn}\ P) - \{y\})\ \mathtt{in} \\
\qquad\qquad\qquad \mathtt{let}\ y' = (y \in vs \Rightarrow ch\ y\ vs \mid y)\ \mathtt{in} \\
\qquad\qquad\qquad\quad \mathsf{In}\ (s\ x)\ y'\ (\mathsf{Sub}\ ch\ P\ (\lambda n.\ (n{=}y) \Rightarrow y' \mid s\ n)) \wedge \\
\quad \forall ch\ P\ s.\ \mathsf{Sub}\ ch\ (\mathsf{Tau}\ P)\ s = \mathsf{Tau}\ (\mathsf{Sub}\ ch\ P\ s) \wedge \\
\quad \forall ch\ y\ P\ s.\ \mathsf{Sub}\ ch\ (\mathsf{Res}\ y\ P)\ s = \\
\qquad\qquad\quad \mathtt{let}\ vs = \mathsf{Image}\ s\ ((\mathsf{Fn}\ P) - \{y\})\ \mathtt{in} \\
\qquad\qquad\quad \mathtt{let}\ y' = (y \in vs \Rightarrow ch\ y\ vs \mid y)\ \mathtt{in} \\
\qquad\qquad\qquad \mathsf{Res}\ y'\ (\mathsf{Sub}\ ch\ P\ (\lambda n.\ (n = y) \Rightarrow y' \mid s\ n)) \wedge \\
\quad \forall ch\ x\ y\ P\ s.\ \mathsf{Sub}\ ch\ (\mathsf{Match}\ x\ y\ P)\ s = \mathsf{Match}\ (s\ x)\ (s\ y)\ (\mathsf{Sub}\ ch\ P\ s) \wedge \\
\quad \forall ch\ P\ Q\ s.\ \mathsf{Sub}\ ch\ (\mathsf{Comp}\ P\ Q)\ s = \mathsf{Comp}\ (\mathsf{Sub}\ ch\ P\ s)\ (\mathsf{Sub}\ ch\ Q\ s) \wedge \\
\quad \forall ch\ P\ Q\ s.\ \mathsf{Sub}\ ch\ (\mathsf{Plus}\ P\ Q)\ s = \mathsf{Plus}\ (\mathsf{Sub}\ ch\ P\ s)\ (\mathsf{Sub}\ ch\ Q\ s) \wedge \\
\quad \forall ch\ P\ s.\ \mathsf{Sub}\ ch\ (\mathsf{Repl}\ P)\ s = \mathsf{Repl}\ (\mathsf{Sub}\ ch\ P\ s)
\end{array}
$$

The definition is straightforward, except for the defining equations for the input prefix In and restriction Res. For all the other constructors, the function Sub simply maps the substitution recursively down through an agent, applying the mapping s wherever free names occur. The input prefix and restriction constructs 'In $x\ y\ P$' and 'Res $y\ P$', however, both bind the name y. It may therefore be necessary to change this bound name to a fresh name y', in order to avoid capture of names when the substitution s is applied to P. The definition ensures that bound names are changed only when necessary, namely when y occurs in the image of the function s on the set of all names (other than y itself) that occur free in P. In this case, the bound name is changed to a new name y' which is generated by the choice function ch and which, under the infinity hypothesis for ch, does not occur in this set. Any free occurrences of y in P are also changed to y'.

4.4.3 Theorems about substitution

A number of general theorems about substitution are needed for proofs about the π-calculus. The content of these theorems is mostly predictable, and a full list of theorems need not be given here. In proving these theorems in the HOL system, care was taken to restrict dependence on the infinity hypothesis for the choice function to only those theorems for which it is really needed. For example, one of the theorems proved in HOL states that the identity substitution leaves agents unchanged:

$$\vdash \forall P\ ch.\ \mathsf{Sub}\ ch\ P\ (\lambda x.x) = P$$

This proposition holds for any function ch whatsoever, and the theorem therefore does not include the infinity hypothesis for ch as an assumption. By contrast, the following theorem

$$\begin{aligned}&\vdash \forall ch.\ (\forall S.\ \mathsf{Finite}\ S \supset \forall n.\ \neg(ch\ n\ S \in S))\ \supset\\ &\qquad \forall P\ s.\ \mathsf{Fn}\ (\mathsf{Sub}\ ch\ P\ s) = \mathsf{Image}\ s\ (\mathsf{Fn}\ P)\end{aligned}$$

states that the set of names that occur free in an agent after substitution with a name mapping s is the same as the image of the function s on the original set of free names. This holds only if the choice function ch correctly generates new bound names chosen from an infinite set of names α. In this theorem, the infinity hypothesis is essential.

4.4.4 Substitution for a single name

Simultaneous substitution of names for names is needed for only certain parts of the theory developed in [17, 18]. In the absence of agent identifiers, full simultaneous substitution is not needed for defining the transition relation, strong bisimulation and the relation $\dot{\sim}$. Substitution for a single name will suffice.

Substitution of x for y in the agent P, written '$P\{x/y\}$' in the notation of section 2.1, is formalized by the constant definition

$$\vdash \forall ch\ P\ x\ y.\ \mathsf{Sub1}\ ch\ P\ (x,y) = \mathsf{Sub}\ ch\ P\ (\lambda n.\ (n = y) \Rightarrow x \mid n)$$

where substitution for a single name is defined in terms of a simultaneous substitution in which the name mapping is the identity function on all names but one. Theorems about the special case of substitution for a single name are (mostly) straightforward to prove in HOL, given this definition of Sub1 and the more general theory of simultaneous substitution.

5 Formalizing the transitional semantics

The theory outlined above provides all the syntactic infrastructure needed to define and reason about the transitional semantics for the π-calculus in logic. This section describes how the labelled transition relation on which this semantics is based is defined in HOL and gives a sketch of the theory developed from this definition.

5.1 Representing actions in HOL

The transition system for the π-calculus shown in section 2.2 is based on four kinds of actions. These are represented in logic by values of the type $(\alpha)action$, which is specified by

$$
\begin{array}{llll}
action & ::= & \mathsf{tau} & \mathsf{tau} \text{ represents } \tau \\
 & | & \mathsf{fo}\ \alpha\ \alpha & \mathsf{fo}\ x\ y \text{ represents } \overline{x}y \\
 & | & \mathsf{in}\ \alpha\ \alpha & \mathsf{in}\ x\ y \text{ represents } x(y) \\
 & | & \mathsf{bo}\ \alpha\ \alpha & \mathsf{bo}\ x\ y \text{ represents } \overline{x}(y)
\end{array}
$$

and which is defined automatically using the same derived rule of (recursive) type definition used to define the type of agents. The concrete type $(\alpha)action$ specified by this equation has four constructors. One of these, namely **tau**, is a constant representing the distinguished action τ; the other three are functions of type $\alpha{\to}\alpha{\to}(\alpha)action$ that map a pair of names to the representation of an action.

Given this specifying equation for the type of actions, the derived rule of type definition automatically proves the following characterizing theorem for the type $(\alpha)action$:

$$
\begin{array}{l}
\vdash \forall e\ f_0\ f_1\ f_2.\ \exists! fn{:}(\alpha)action{\to}\beta. \\
\quad fn\ \mathsf{tau} = e\ \wedge \\
\quad \forall x_0\ x_1.\ fn(\mathsf{fo}\ x_0\ x_1) = f_0\ x_0\ x_1\ \wedge \\
\quad \forall x_0\ x_1.\ fn(\mathsf{in}\ x_0\ x_1) = f_1\ x_0\ x_1\ \wedge \\
\quad \forall x_0\ x_1.\ fn(\mathsf{bo}\ x_0\ x_1) = f_2\ x_0\ x_1
\end{array}
$$

This theorem asserts that functions over the type $(\alpha)action$ can be uniquely defined by cases on the four different kinds of actions in the π-calculus. It is straightforward to use this theorem in conjunction with the derived principle of (primitive recursive) function definition to define logical counterparts to the functions fn, bn and n on actions introduced in section 2.2. For example, the definition of a function $\mathsf{fn}{:}(\alpha)action{\to}(\alpha)set$ that corresponds to fn is just

$$
\begin{array}{l}
\vdash \mathsf{fn}\ \mathsf{tau} = \{\}\ \wedge \\
\ \ \forall x\ y.\ \mathsf{fn}(\mathsf{fo}\ x\ y) = \{x, y\}\ \wedge \\
\ \ \forall x\ y.\ \mathsf{fn}(\mathsf{in}\ x\ y) = \{x\}\ \wedge \\
\ \ \forall x\ y.\ \mathsf{fn}(\mathsf{bo}\ x\ y) = \{x\}
\end{array}
$$

The definitions of functions **bn** and **n** corresponding to bn and n are similar. Given these definitions and the characterizing theorem for the type $(\alpha)action$, it is trivial to develop a basic theory of actions for the π-calculus in HOL.

5.2 Defining the labelled transition relation

In the paper [18], the transition relation $\longrightarrow$ is defined inductively by the rules reproduced in the present paper in figure 1. In the mechanized theory of the π-calculus this relation is also defined inductively, using a derived principle of *inductive predicate definition* implemented in HOL [13]. Given the user's specification of a desired set of rules, this derived principle of definition automatically proves the existence of the relation inductively defined by them. More precisely, the system constructs a term that explicitly denotes the smallest relation closed under the rules specified by the user. HOL then introduces (via a constant specification) a constant to name this relation. The result is a collection of automatically proved theorems stating that the newly-defined relation is in fact closed under the required rules, together with an additional theorem asserting that it is the smallest such relation.

To define the transition relation using this derived principle of inductive definition, the user just enters the transition rules shown in figure 1 as a list of pairs of the form

$$(\,\langle \textit{list of premises}\rangle, \langle \textit{conclusion}\rangle\,)$$

Each pair consists of a list of the premises of a rule, including any side conditions, and its conclusion. There is one such pair for each of the transition rules, including all symmetric forms. The premises and conclusions are stated using the HOL representation of agents and actions and (where necessary) the notation for free and bound names and substitution defined in the syntactic theory described above.

Given this user-supplied specification of the rules, the system constructs a logical statement of each transition rule in the form of an implication of conclusion by premises. These express what it means for a 3-place relation

$$R{:}(\alpha)\mathit{agent}{\to}(\alpha)\mathit{action}{\to}(\alpha)\mathit{agent}{\to}\mathit{bool}$$

to be closed under each of the rules. The assertion that the relation R is closed under the left-hand symmetric form of the SUM rule, for example, is expressed in logic by the implication shown below.

$$\forall P\; a\; P'.\; R\; P\; a\; P' \supset \forall Q.\; R\; (\mathsf{Plus}\; P\; Q)\; a\; P'$$

Likewise, the translation into higher order logic of the OPEN rule is

$$\begin{array}{l} \forall P\; x\; y\; P'\; w. \\ \quad R\; P\; (\mathsf{fo}\; x\; y)\; P' \wedge \neg(y{=}x) \wedge \neg w \in \mathsf{Fn}(\mathsf{Res}\; y\; P') \;\supset \\ \qquad R\; (\mathsf{Res}\; y\; P)\; (\mathsf{bo}\; x\; w)\; (\mathsf{Sub1}\; ch\; P'\; (w, y)) \end{array}$$

This logical formulation of the OPEN rule illustrates an explicit use of the defined syntactic notions of substitution and the set of free names in an agent. The translations into logic of the remaining rules are similar to these examples.

The definition made by HOL of the transition relation is based on this translation of the rules into logical implications. The conjunction of all these implications asserts the closure of an arbitrary three-place relation R under the transition rules of the π-calculus, and the labelled transition relation itself is just defined to be the intersection of all such relations. More precisely, the derived HOL rule of inductive definition makes a constant specification for the relation

$$\mathsf{Trans} : (\alpha{\to}(\alpha)set{\to}\alpha) \to (\alpha)\mathit{agent} \to (\alpha)\mathit{action} \to (\alpha)\mathit{agent} \to \mathit{bool}$$

which is logically equivalent to the following constant definition:

$$\begin{array}{l} \vdash \mathsf{Trans}\; ch\; P\; a\; Q = \\ \quad \forall R{:}(\alpha)\mathit{agent}{\to}(\alpha)\mathit{action}{\to}(\alpha)\mathit{agent}{\to}\mathit{bool}. \\ \qquad \langle R \textit{ is closed under the rules}\rangle \supset R\; P\; a\; Q \end{array}$$

This definition states that there is a transition from the agent P to the agent Q labelled by the action a exactly when P, a and Q are in the intersection (i.e. '$\forall$') of every relation R closed under the transition rules for the π-calculus. The relation Trans must take the choice function ch as an argument, since substitution is employed in stating closure under the rules.

The final result of making the automatic inductive definition sketched above (and all the user actually sees) is a set of theorems that state the transition rules for the defined relation Trans, together with an additional theorem stating that Trans is the smallest relation closed under these rules. The following theorems for the left-hand SUM rule and the OPEN rule, for example, are among the theorems proved automatically by the system:

$$\vdash \forall ch\ P\ a\ P'.\ \mathsf{Trans}\ ch\ \ P\ a\ P' \supset \forall Q.\ \mathsf{Trans}\ ch\ (\mathsf{Plus}\ P\ Q)\ a\ P'$$

$$\begin{array}{l}\vdash \forall ch\ P\ x\ y\ P'\ w. \\ \quad \mathsf{Trans}\ ch\ P\ (\mathsf{fo}\ x\ y)\ P' \wedge \neg(y{=}x) \wedge \neg w \in \mathsf{Fn}(\mathsf{Res}\ y\ P') \ \supset \\ \quad\quad \mathsf{Trans}\ ch\ (\mathsf{Res}\ y\ P)\ (\mathsf{bo}\ x\ w)\ (\mathsf{Sub1}\ ch\ P'\ (w,y))\end{array}$$

There are sixteen such theorems for the π-calculus with replication in place of agent identifiers, one for each transition rule including symmetric forms. The additional theorem stating that Trans (actually, that $\mathsf{Trans}\ ch$) is the smallest relation closed under the rules, which is also derived automatically by the rule of inductive predicate definition, has the form

$$\begin{array}{l}\vdash \forall ch.\ \forall R{:}(\alpha)agent{\to}(\alpha)action{\to}(\alpha)agent{\to}bool. \\ \quad \langle R\ \textit{is closed under the rules}\rangle \ \supset \\ \quad\quad \forall P\ a\ Q.\ \mathsf{Trans}\ ch\ P\ a\ Q \supset R\ P\ a\ Q\end{array}$$

This *rule induction* theorem for Trans is essential for proving properties of the transition relation by induction on the depth of inference. By appeal to an appropriate instance of this theorem, one may reduce proving that some property $R[P, a, Q]$ holds of all a-labelled transitions from P to Q to showing that this property is preserved by the transition rules for the π-calculus.

5.3 Proof tools associated with the transition relation

Associated with the derived rule of inductive predicate definition are several general-purpose proof tools for reasoning about inductively defined relations in HOL. The most important of these is a tactic for interactive goal-directed proofs by rule induction. This tactic mechanizes the inductive form of argument outlined above; given the rule induction theorem for Trans and a hypothesis to be proved of the form

$$\forall P\ a\ Q.\ \mathsf{Trans}\ ch\ P\ a\ Q \supset R[P, a, Q]$$

the rule induction tactic reduces the task of proving this hypothesis to proving that the property expressed by '$R[P, a, Q]$' is preserved by the rules that inductively define Trans. Many of the proofs about the π-calculus in [18] are done by induction on the depth of inference, so this tactic is of primary importance in mechanizing these proofs in HOL.

Other proof tools associated with the transition relation include a set of HOL tactics for proving that specific labelled transitions hold between agents of the calculus. For example, one of these tactics can be used to reduce the task of proving that $\mathsf{Trans}\ ch\ (P + Q)\ a\ P'$ to proving that $\mathsf{Trans}\ ch\ P\ a\ P'$. These tactics are constructed automatically by the system from the theorems stating the transition rules for Trans. There is also an automatic proof procedure for deriving an exhaustive case analysis theorem for the transition system:

$$\begin{array}{l}\vdash \mathsf{Trans}\ ch\ P\ a\ Q\ \ = \\ \quad (P{=}\mathsf{Tau}\ Q \wedge a{=}\mathsf{tau}) \vee \\ \quad (\exists x\ y.\ P{=}\mathsf{Out}\ x\ y\ Q \wedge a{=}\mathsf{fo}\ x\ y) \vee \\ \quad (\exists P'\ Q'.\ P{=}\mathsf{Plus}\ P'\ Q' \wedge \mathsf{Trans}\ ch\ P'\ a\ Q) \vee \ldots\end{array}$$

This theorem may be loosely paraphrased as follows:

if $\vdash P \xrightarrow{a} Q$, then this follows from
 the TAU-ACT rule, or
 the OUTPUT-ACT rule, or
 the PLUS rule, or...

This fact is used to mechanize arguments about the transition system of the kind that are typically accompanied by an explanation of the form 'if ..., then by a shorter inference ...'.

5.4 Theorems about the transition relation

The theorems and proof tools described above provide the logical infrastructure necessary to develop the HOL theory of the labelled transition relation for the π-calculus. This theory consists of a collection of simple facts about the transition relation formalized by Trans. One example is the following lemma about free and bound names, which shows how dependence on the infinity hypothesis propagates to the level of transitions:

$$\begin{aligned}&\vdash \forall ch.\ (\forall S.\ \mathsf{Finite}\ S \supset \forall n.\ \neg(ch\ n\ S \in S)) \supset \\ &\qquad \forall P\ a\ P'.\ \mathsf{Trans}\ ch\ P\ a\ P' \supset (\mathsf{Fn}\ P' \subseteq (\mathsf{Fn}\ P \cup \mathsf{bn}\ a)) \wedge (\mathsf{fn}\ a \subseteq \mathsf{Fn}\ P)\end{aligned}$$

This is one in a series of lemmas for the proof that α-equivalence is a strong bisimulation presented in the paper [18]. The HOL proof was done using the rule induction tactic described above; this is very natural, since the theorem to be proved is an implication of precisely the form one can infer using the rule indiction theorem for Trans. The HOL proof closely follows the detailed proof given in [18], which proceeds by induction on the depth of inference.

Other theorems that have been proved in HOL about the labelled transition system include various equivalences between transitions, for example

$$\vdash \forall ch\ P\ a\ Q\ R.\ \mathsf{Trans}\ ch\ (\mathsf{Plus}\ P\ Q)\ a\ R = \mathsf{Trans}\ ch\ (\mathsf{Plus}\ Q\ P)\ a\ R$$

$$\vdash \forall ch\ P\ a\ Q.\ \mathsf{Trans}\ ch\ (\mathsf{Plus}\ P\ \mathsf{Zero})\ a\ Q = \mathsf{Trans}\ ch\ P\ a\ Q$$

$$\vdash \forall ch\ P\ x\ a\ Q.\ \mathsf{Trans}\ ch\ (\mathsf{Match}\ x\ x\ P)\ a\ Q = \mathsf{Trans}\ ch\ P\ a\ Q$$

One can also prove that certain transitions are impossible, as in

$$\vdash \forall ch\ P\ a.\ \neg(\mathsf{Trans}\ ch\ \mathsf{Zero}\ a\ P)$$

Simple theorems of this kind follow directly from the rules defining the relation Trans and the case analysis theorem discussed in the preceding section. They are easy to prove, and the proofs are very regular and could be completely automated in HOL.

6 Defining bisimulation and equivalence

Once the substitution function Sub1 and the transition relation Trans have been defined, it is straightforward to express the concept of a strong simulation in logic. The following definition is a direct translation into higher order logic of the definition given in section 2.3.

$$
\begin{array}{l}
\vdash \mathsf{Sim}\ ch\ S = \\
\quad \forall P\ Q.\ S\ P\ Q\ \supset \\
\qquad \forall P'.\ \mathsf{Trans}\ ch\ P\ \mathsf{tau}\ P'\ \supset \\
\qquad\quad \exists Q'.\ \mathsf{Trans}\ ch\ Q\ \mathsf{tau}\ Q' \wedge S\ P'\ Q' \wedge \\
\qquad \forall x\ y\ P'.\ \mathsf{Trans}\ ch\ P\ (\mathsf{fo}\ x\ y)\ P'\ \supset \\
\qquad\quad \exists Q'.\ \mathsf{Trans}\ ch\ Q\ (\mathsf{fo}\ x\ y)\ Q' \wedge S\ P'\ Q' \wedge \\
\qquad \forall x\ y\ P'.\ \mathsf{Trans}\ ch\ P\ (\mathsf{in}\ x\ y)\ P' \wedge \neg(y \in (\mathsf{N}\ P \cup \mathsf{N}\ Q))\ \supset \\
\qquad\quad \exists Q'.\ \mathsf{Trans}\ ch\ Q\ (\mathsf{in}\ x\ y)\ Q' \wedge \\
\qquad\qquad\quad \forall w.\ S\ (\mathsf{Sub1}\ ch\ P'\ (w, y))\ (\mathsf{Sub1}\ ch\ Q'\ (w, y)) \wedge \\
\qquad \forall x\ y\ P'.\ \mathsf{Trans}\ ch\ P\ (\mathsf{bo}\ x\ y)\ P' \wedge \neg(y \in (\mathsf{N}\ P \cup \mathsf{N}\ Q))\ \supset \\
\qquad\quad \exists Q'.\ \mathsf{Trans}\ ch\ Q\ (\mathsf{bo}\ x\ y)\ Q' \wedge S\ P'\ Q'
\end{array}
$$

This defines '$\mathsf{Sim}\ ch\ S$' to mean 'the relation S is a strong simulation'. The predicate Sim must take the choice function ch as a parameter because its definition depends on substitution.

Given this definition, the bisimilarity relation $\dot{\sim}$ between agents is defined in HOL by the simple constant definition

$$\vdash \mathsf{Bisim}\ ch\ P\ Q = \exists S.\ S\ P\ Q \wedge \mathsf{Sim}\ ch\ S \wedge \mathsf{Sim}\ ch\ (\lambda x\ y.\ S\ y\ x)$$

This says that two agents P and Q are bisimilar if $S\ P\ Q$ holds for any strong bisimulation S; it uses (higher-order) existential quantification over relations to define '$\mathsf{Bisim}\ ch$' to be the largest strong bisimulation. Once again, the decision to use a type variable to model the set of names means that the choice function must appear as a parameter to Bisim.

Finally, strong equivalence is defined to be bisimilarity under all substitutions of names for names. In HOL, we just define

$$\vdash \mathsf{Equiv}\ ch\ P\ Q = \forall s{:}\alpha \to \alpha.\ \mathsf{Bisim}\ \ (\mathsf{Sub}\ ch\ P\ s)\ \ (\mathsf{Sub}\ ch\ Q\ s)$$

Notice that universal quantification over substitution functions is used in this definition; *higher-order* logic makes a direct definition completely straightforward. As usual, the choice function ch becomes a parameter.

7 The algebraic theory

Having defined strong bisimulation and equivalence in HOL, one may then proceed to develop the algebraic theory presented in [17, 18] as a collection of theorems about the relations Bisim and Equiv. Proofs have been completed in HOL for many of the simpler equivalences in this theory, but work on the theory is still in progress. Some examples of the theorems proved are the laws for summation shown above in section 2.3. These are expressed in logic by the theorems

$$\vdash \forall ch\ P.\ \mathsf{Bisim}\ ch\ (\mathsf{Plus}\ P\ \mathsf{Zero})\ P$$

$$\vdash \forall ch\ P.\ \mathsf{Bisim}\ ch\ (\mathsf{Plus}\ P\ P)\ P$$

$$\vdash \forall ch\ P\ Q.\ \mathsf{Bisim}\ ch\ (\mathsf{Plus}\ P\ Q)\ (\mathsf{Plus}\ Q\ P)$$

$$\vdash \forall ch\ P\ Q\ R.\ \mathsf{Bisim}\ ch\ (\mathsf{Plus}\ P\ (\mathsf{Plus}\ Q\ R))\ (\mathsf{Plus}\ (\mathsf{Plus}\ P\ Q)\ R)$$

These theorems were proved in HOL in the same way that the corresponding laws are proved in [18], namely by explicitly producing an appropriate strong bisimulation in each case. For example, the bisimulation used to prove the commutative law of summation is presented in [18] as

$$\{(P_1 + P_2,\ P_2 + P_1) \mid P_1, P_2 \text{ agents}\} \cup \mathbf{Id}$$

where **Id** is the identity relation on agents. In the HOL proof, the same relation is written

$$\lambda P.\, \lambda Q.\, (P = Q) \vee \exists P'\, Q'.\, (P = \mathsf{Plus}\ P'\ Q') \wedge (Q = \mathsf{Plus}\ Q'\ P')$$

Formally, this term becomes the witness supplied for the existentially quantified variable in an instance of the definition of Bisim. The proof that this is indeed a strong bisimulation proceeds essentially by rewriting, making extensive use of the theory of the transition system discussed above in section 5.2. Several other laws are may be proved in HOL in exactly the same way—that is, by exhibiting an appropriate bisimulation.

Following [18], many of the laws for equivalence may be easily derived in HOL from corresponding laws for bisimilarity. For example, to prove

$$\vdash \forall ch\ P.\, \mathsf{Equiv}\ ch\ (\mathsf{Plus}\ P\ \mathsf{Zero})\ P$$

we merely use the theorem-prover's built-in rewriting facility to rewrite with the definitions of Equiv and Sim, transforming this proposition into

$$\vdash \forall ch\ P\ s.\, \mathsf{Bisim}\ ch\ (\mathsf{Plus}\ (\mathsf{Sub}\ ch\ P\ s)\ \mathsf{Zero})\ (\mathsf{Sub}\ ch\ P\ s)$$

This is just an instance of the identity law for bisimilarity already proved, and so the desired result follows immediately.

Once all the laws have been proved, we will have the theory of strong equivalence for agents available as a collection of (essentially) equational theorems in HOL. We will then be able to use these theorems to reason about applications in this theory. In the simplest case, such reasoning could consist in just interactively guiding HOL's rewriting tools to use the laws to show that two particular agents—describing, say, an implementation and a specification—are equivalent. One could also investigate more automatic proof strategies based on algebraic manipulation.

But because the *definition* of equivalence is also available, we could do equivalence proofs by directly exhibiting bisimulations as well. We might even employ a mixture of the two proof styles, using both algebra and simulation as necessary. Furthermore, we could also prove that two agents are *not* equivalent in this framework; the case analysis theorem presented in section 5.3 lets us reason directly about possible transitions. More generally, the full range of classical proof techniques—induction, proof by contradiction, equational reasoning—is potentially available in such a system. Both automatic and semi-automatic (user guided) approaches to proof can be implemented. The result is a rather powerful and flexible framework for both practical use and theoretical experiments.

8 Concluding remarks

This paper has outlined work in progress on a mechanized formal theory of the π-calculus in higher order logic using the HOL system. This theory is still far from complete—the expansion law is yet to be derived, for example—and it is still too early to tell if the goals mentioned in the introduction can be achieved. But the results obtained so far seem to indicate that some measure of success is possible. Once the theory is complete, we intend to test it on a realistic application. It would also be interesting to compare the practical utility of the HOL mechanization with a proof system for the π-calculus implemented using a more general logical framework, such as Isabelle [20] or the Edinburgh Logical Framework [10].

The research most closely related to the theory described in this paper is Monica Nesi's work on a theory of CCS in HOL [19]. This work parallels ours; essentially the same techniques are used to define the syntax and transitional semantics of CCS and to derive rules for observation congruence. A modal logic for CCS (a variant of Hennessy-Milner logic [11]) is also included in Nesi's theory. One of the main technical differences between the two formalizations is that the CCS theory has managed to avoid the difficulties connected with substitution. In particular, although Nesi's theory includes recursively-defined processes $\mathsf{rec}\, X.\, E$, and hence includes bound process variables and substitutions, it is (informally) assumed that bound variables are chosen so that captures do not occur. By contrast, the present theory deals with the possibility of free variable capture explicitly and formally.

A very different approach to providing theorem-proving support for the π-calculus is that of the Mobility Workbench [24]. This is a special-purpose tool for automated reasoning about equivalences between agents. Given two agents P and Q, the system attempts to construct a bisimulation that relates them; this is done by incrementally generating the state spaces of P and Q at the same time as building the bisimulation relation. This gives a decision procedure for equivalence in a certain class of agents with *finite control* (similar to finite state systems in CCS). The exact equivalence employed is Sangiorgi's *open bisimulation* relation [23].

The basic strategy of proving equivalences by constructing bisimulations is, of course, also technically possible in the HOL mechanization—it was used 'manually' in the proofs discussed in section 7, for example. It would interesting to see if algorithms of the kind employed in the Mobility Workbench could be adapted for the HOL framework, or even if some hybrid system could be constructed. (Such an investigation may require the HOL theory to be revised to employ open bisimulation.) A HOL tool based on this idea should, in principle, be more powerful than the more specialized Workbench; for example, it should be possible in such a system to combine algebraic reasoning with the construction of bisimulation relations, perhaps in a semi-automated way. Furthermore, one could also reason about π-calculus agents without finite control. The automatic parts of any HOL-based tool are, however, likely to be considerably slower than the more specialized system.

As a further development of this work, a HOL mechanization of the polyadic π-calculus [16] should be considered. The formulation of this calculus employs the notion of *structural congruence* to separate the laws dealing with the structure of groups of agents from those describing how these agents interact. The former are just postulated as equational axioms, whereas the latter are derived from a reduced set of transition rules. In the corresponding HOL theory, one would need to derive the axiomatic component formally; the most direct approach would be to take a quotient using an appropriately-defined equivalence relation on terms.

Much of the HOL theory outlined in this paper is concerned with syntax, and in particular with the fundamental ideas of variable binding and substitution. As well as being rather dull, these technicalities are notoriously easy to make mistakes about. A general solution to these problems is one of the aims of Andy Gordon's work on representing syntax in a mechanized logic [6]. The goal is to define a general theory of syntax and to construct tools to automatically define specific syntaxes in logic and to reason about them. There is, therefore, some hope that theory developments of the kind described in the present paper can be made considerably easier in future.

Acknowledgements

I am grateful to Professor Robin Milner for explaining how agent identifiers could be replaced by replication. Thanks are also due to Monica Nesi and Yves Bertot, who carefully read an early draft of this paper and found several typographical errors, and to Konrad Slind for valuable comments on the theory and its presentation. Some preliminary studies for this work were done jointly with Mike Gordon at the University of Cambridge Computer Laboratory.

References

[1] R. J. R. Back and J. von Wright, 'Refinement Concepts Formalised in Higher Order Logic', *Formal Aspects of Computing*, Vol. 2, No. 3 (July-September 1990), pp. 247–272.

[2] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van Tassel, 'Experience with embedding hardware description languages in HOL', in *Theorem Provers in Circuit Design: Theory, Practice and Experience: Proceedings of the IFIP WG10.2 International Conference, Nijmegen, June 1992*, edited by V. Stavridou, T. F. Melham, and R. T. Boute (North-Holland, 1992), pp. 129–156.

[3] A. J. Camilleri, 'Mechanizing CSP Trace Theory in Higher Order Logic', *IEEE Transactions on Software Engineering*, Vol. 16, No. 9 (September 1990), pp. 993–1004.

[4] A. Church, 'A Formulation of the Simple Theory of Types', *The Journal of Symbolic Logic*, Vol. 5 (1940), pp. 56–68.

[5] J. A. Goguen, J. W. Thatcher, and E. G. Wagner, 'An initial algebra approach to the specification, correctness, and implementation of abstract data types', in *Current Trends in Programming Methodology*, edited by R.T. Yeh (Prentice-Hall, 1978), Vol. IV, pp. 80–149.

[6] A. Gordon, 'A Mechanisation of Name-carrying Syntax up to Alpha-conversion' in *Higher-order logic theorem proving and its applications, Proceedings 1993*, Lecture Notes in Computer Science, Vol. 780 (Springer-Verlag, 1994).

[7] M. J. C. Gordon, 'Mechanizing Programming Logics in Higher Order Logic', in: *Current Trends in Hardware Verification and Automated Theorem Proving*, edited by G. Birtwistle and P.A. Subrahmanyam (Springer-Verlag, 1989), pp. 387–439.

[8] M. J. C. Gordon and T. F. Melham, eds. *Introduction to HOL: A theorem proving environment for higher order logic* (Cambridge University Press, 1993).

[9] M. J. Gordon, A. J. Milner, and C. P. Wadsworth, *Edinburgh LCF: A Mechanised Logic of Computation*, Lecture Notes in Computer Science, Vol. 78 (Springer-Verlag, 1979).

[10] R. Harper, F. Honsell, and G. Plotkin, 'A Framework for Defining Logics', Report no. ECS-LFCS-87-23, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh (March 1987).

[11] M. Hennessy and R. Milner, 'Algebraic Laws for Nondeterminism and Concurrency', *Journal of the ACM*, Vol. 32, No. 1 (January 1985), pp. 137–161.

[12] T. F. Melham, 'Automating Recursive Type Definitions in Higher Order Logic', in *Current Trends in Hardware Verification and Automated Theorem Proving*, edited by G. Birtwistle and P. A. Subrahmanyam (Springer-Verlag, 1989), pp. 341–386.

[13] T. Melham, 'A Package for Inductive Relation Definitions in HOL', in *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, Davis, August 1991*, edited by M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley (IEEE Computer Society Press, 1992), pp. 350–357.

[14] T. F. Melham, 'Using Recursive Types to Reason about Hardware in Higher Order Logic', in *Proceedings of the IFIP WG 10.2 Working Conference on the Fusion of Hardware Design and Verification*, edited by G. J. Milne (North-Holland, 1988), pp. 51–75.

[15] R. Milner, *Communication and Concurrency* (Prentice Hall, 1989).

[16] R. Milner, 'The Polyadic π-Calculus: a Tutorial', Report no. ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh (October 1991).

[17] R. Milner, J. Parrow, and D. Walker, 'A Calculus of Mobile Processes, I', *Information and Computation*, Vol. 100, No. 1 (September, 1992), pp. 1–40.

[18] R. Milner, J. Parrow, and D. Walker, 'A Calculus of Mobile Processes, II', *Information and Computation*, Vol. 100, No. 1 (September, 1992), pp. 41–77.

[19] M. Nesi, 'A Formalization of the Process Algebra CCS in Higher Order Logic', Technical report no. 278, Computer Laboratory, University of Cambridge (December 1992).

[20] L. C. Paulson, 'Isabelle: The Next 700 Theorem Provers', in *Logic and Computer Science*, edited by P. Odifreddi (Academic Press, 1990), pp. 361–386.

[21] L. C. Paulson, *Logic and Computation: Interactive Proof with Cambridge LCF*, Cambridge Tracts in Theoretical Computer Science 2 (Cambridge University Press, 1987).

[22] L. C. Paulson, *ML for the Working Programmer* (Cambridge University Press, 1991).

[23] D. Sangiorgi, 'A Theory of Bisimulation for the π-calculus', in *CONCUR'93: 4th International Conference on Concurrency Theory, Hildesheim, August 1993, Proceedings*, edited by E. Best, Lecture Notes in Computer Science, Vol. 715 (Springer-Verlag, 1993), pp. 127–142.

[24] B. Victor and F. Moller, 'The Mobility Workbench: A Tool for the π-calculus', Report no. ECS-LFCS-94-285, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh (February 1994).