A Proof Search Specification of the π-Calculus

Alwen Tiu[1] and Dale Miller[2]

[1] École polytechnique and Penn State University
tiu@lix.polytechnique.fr
[2] INRIA-Futurs and École polytechnique
dale.miller@inria.fr

Abstract. We present a meta-logic that contains a new quantifier ∇ (for encoding "generic judgments") and inference rules for reasoning within fixed points of a given specification. We then specify the operational semantics and bisimulation relations for the finite π-calculus within this meta-logic. Since we restrict to the finite case, the ability of the meta-logic to reason within fixed points becomes a powerful and complete tool since simple proof search can compute this one fixed point. The ∇ quantifier helps with the delicate issues surrounding the scope of variables within π-calculus expressions and their executions (proofs). We shall illustrate several merits of the logical specifications we write: they are natural and declarative; they contain no-side conditions concerning names of variables while maintaining a completely formal treatment of such variables; differences between late and open bisimulation relations are easy to see declaratively; and proof search involving the application of inference rules, unification, and backtracking can provide complete proof systems for both one-step transitions and for bisimulation.

1 Introduction

In order to treat abstractions within expressions and computation declaratively, we shall work within a meta-logic which contains a well understood notion of abstraction: in particular, we shall work with a logic inspired by Church's Simple Theory of Types [4], where terms are actually simply typed λ-terms. Just as it is common to use logic-level application to represent object-level application (for example, the encoding of $P+Q$ is via the meta-level application of the encoding for plus to the encoding of its two arguments), we shall use logic-level abstractions (via λ-abstractions) to encode object-level abstractions. The λ-terms in our setting are thus simply typed and satisfy the usual rules for for α, β, and η-conversion. This style of syntactic encoding has been called *λ-tree syntax* [24]. The term *higher-order abstract syntax* [34] was originally applied to this kind of encoding, but in more recent years, HOAS has come to encompass the use of arbitrary higher-order functions to encode abstractions in syntax. Whatever term one wishes to use to classify our approach here, it is important to understand that λ-abstractions are only intended to form abstractions over syntax and their functional interpretation is limited to providing object-level substitution via β-reduction.

We make use of the ∇-quantifier, first introduced in the logic $FO\lambda^{\Delta\nabla}$ [26], to help encode the notion of "generic judgment" that occurs commonly when reasoning with λ-tree syntax. The ∇ quantifier is used to introduce new elements into a type within a given scope. In particular, a reading of the truth condition for $\nabla x_\gamma.Bx$ is something like: if given a new element, say c, of type γ, then check the truth of Bc. Notice that this is hypothetical reasoning about the datatype γ and it does not require knowing whether or not this type actually contains any members. This is, of course, rather central to the notion of *generic*: something holds generically usually means that it holds for certain "internal" reasons (the structure of an argument, for example) and not for some accident concerning members of the domain. This is quite different from determining the truth of $\forall x_\gamma.Bx$: check that Bt is true for all t in the type γ. If the type is empty, this condition is vacuously true and if the type is infinite, we have an infinite number of checks to make in principle.

It is useful to provide here a high-level comparison between the ∇-quantifier and the "new" quantifier of Gabbay and Pitts [10]. In their set theory foundations, a domain containing an infinite number of names is assumed given. To deal with notions of freshness, renaming of bound variables, substitution, etc, they provide a series of primitives between names and terms that can be used to guarantee that a name does not occur within a term, that one name can be swapped for another, etc. Based on these concepts, they can define a new quantifier that guarantees the selection of a "fresh" name for some specific context. In our approach here, there is no particular class of names: the ∇ quantifier will work at any type. (Later, when we discuss the π-calculus explicitly, we shall assume a type for names since this is required by this particular application.) Also, here types do not need to be infinite or even non-empty. Instead, the meaning of $\nabla x_\gamma.Bx$ is one of explicitly introducing a new object of type γ within a certain scope. Thus, the Gabbay-Pitts approach assumes that the type of names is fixed and closed, while the type used with ∇ is open, in the sense that new members of that type can be constructed by the meta-logic for use within a ∇-bound scope. More specifically, the set of theorems for these two quantifiers is quite different. For example, in the logic considered here, the formulas $\forall x.Bx \supset \nabla x.Bx$ and $\nabla x.Bx \supset \exists x.Bx$ are not theorems, but if ∇ is replaced with the Gabbay-Pitts quantifier, they do hold in their theory.

This distinction between having an open versus closed datatype is also a theme that highlights the differences between intuitionistic and classical logic. The meta-logic in this paper is based on intuitionistic logic, a weaker logic than classical logic. One of the principles missing from intuitionistic logic is that of the excluded middle: that is, $A \vee \neg A$ is not generally provable in intuitionistic logic. Consider, for example, the following formula concerning the variable w:

$$\forall x_\gamma[x = w \vee x \neq w]. \qquad (*)$$

In classical logic, this formula is a trivial theorem. If we think constructively, however, this formula is not trivial and might not be desirable in all cases. If the type of quantification γ is a conventional (closed) datatype, then we might expect to have a decision procedure for equality. For example, if γ is the type for lists, then it is a simple matter to construct a procedure that decides whether or not two members of γ are equal by considering the top constructor of the list and, in the event of comparing two non-empty lists, making recursive calls (assuming a decision procedure is available for the elements of the list). In fact, it is possible to prove in an intuitionistic logic augmented with induction (see, for example, [41]) the formula $(*)$ for such closed datatypes.

If the type γ is not given inductively, as is the usual case for names in intuitionistic formalizations of the π-calculus (see [6, 8, 16] and below), then the corresponding instance of $(*)$ is not provable. Thus, whether or not we allow instances of $(*)$ to be assumed can change the nature of a specification. In fact, we show in Section 5, that if we add to our specification of *open bisimulation* [37] assumptions corresponding to $(*)$, then we get a specification of *late bisimulation*. If we were working with a classical meta-logic, such a declarative presentation of these two bisimulations would not have been so easy to present. To see a similar use for an intuitionistic meta-logic and for open types, see [28], where an intuitionistic logic model allowing for open types is used to help establish completeness theorems for the simply typed λ-calculus.

The authors first presented the meta-logic used in this paper in [26] and illustrated its usefulness with the π-calculus: in particular, the specifications of one-step transitions in Figure 2 and of open bisimulation in Figure 4 also appear in [26], but without proof. In this paper, we state the formal properties of our specifications, provide a specification of late bisimulation and provide a novel comparison between open and late bisimulation. In particular, we show that the difference between open and late bisimulation (apart from the difference that arises from the use of closed and open types discussed above) can be captured by the different quantification of free names using $\forall$ and

∇. The different treatment of free names, that is, whether some free names are instantiable or not, highlights the difference between late and open bisimulation, as noted in [38], where the notion of *distinction* among names is introduced to define the open bisimulation relation. We show in Section 5 that a natural class of distinctions can be captured by the alternation of $\forall$ and ∇ quantifiers, and in the case where we are interested only in checking open bisimilarity modulo empty distinction, the notion of distinction that arises in the process of checking bisimilarity is completely subsumed by quantifier alternation. In Section 6 we outline the automation of proof search based on these specifications, which provides us with symbolic bisimulation procedures.

Since our focus in this paper is on names, scoping of names, dependency of names and distinction of names, we choose to focus on finite π-calculus. For related work on ∇ and infinite process behaviors, see first author's PhD [41] and Section 7 of this paper. Since it does not contribute much to our overall analysis, we only briefly consider early bisimulation in Appendix B.

2 Overview of the logic $FO\lambda^{\Delta\nabla}$

The logic $FO\lambda^{\Delta\nabla}$ (pronounced "fold-nabla") is presented using a sequent calculus that is an extension of Gentzen's system LJ [11] for first-order intuitionistic logic. A *sequent* is an expression of the form $B_1, \ldots, B_n \vdash B_0$ where B_i is a formula and the turnstile $\vdash$ denotes logical entailment. To the left of the turnstile is a multiset: thus repeated occurrences of a formula are allowed. If the formulas $B_0, \ldots, B_n$ contain free variables, they are considered universally quantified outside the sequent, in the sense that if the above sequent is provable than every instance of it is also provable. In proof theoretical terms, such free variables are called *eigenvariables*.

A first attempt at using sequent calculus to capture judgments about the π-calculus could be to use eigenvariables to encode names in π-calculus, but this is certainly problematic. For example, if we have a proof for the sequent $\vdash Pxy$, where x and y are different eigenvariables, then logic dictates that the sequent $\vdash Pzz$ is also provable (given that the reading of eigenvariables is universal). If the judgment P is about, say, bisimulation, then it is not likely that a statement about bisimulation involving two different names x and y remains true if they are identified to the same name z.

To address this problem, the logic $FO\lambda^{\Delta\nabla}$ extends sequents with a new notion of "local scope" for proof-level bound variables (originally motivated in [26] to encode "generic judgments"). In particular, sequents in $FO\lambda^{\Delta\nabla}$ are of the form

$$\Sigma\,;\, \sigma_1 \triangleright B_1, \ldots, \sigma_n \triangleright B_n \vdash \sigma_0 \triangleright B_0$$

where Σ is a *global signature*, i.e., the set of eigenvariables whose scope is over the whole sequent, and σ_i is a *local signature*, i.e., a list of variables scoped over B_i. We shall consider sequents to be binding structures in the sense that the signatures, both the global and local ones, are abstractions over their respective scopes. The variables in Σ and σ_i will admit α-conversion by systematically changing the names of variables in signatures as well as those in their scope, following the usual convention of the λ-calculus. The meaning of eigenvariables is as before, only that now instantiation of eigenvariables has to be capture-avoiding, with respect to the local signatures. The variables in local signatures act as locally scoped *generic constants*, that is, they do not vary in proofs since they will not be instantiated. The expression $\sigma \triangleright B$ is called a *generic judgment* or simply a *judgment*. We use script letters $\mathcal{A}$, $\mathcal{B}$, etc. to denote judgments. We write simply B instead of $\sigma \triangleright B$ if the signature σ is empty. We shall often write the list σ as a string of variables, e.g., a judgment $(x_1, x_2, x_3) \triangleright B$ will be written as $x_1x_2x_3 \triangleright B$. If the list x_1, x_2, x_3 is known from context we shall also abbreviate the judgment as $\bar{x} \triangleright B$.

The logical constants of $FO\lambda^{\Delta\nabla}$ are $\forall$ (universal quantifier), $\exists$ (existential quantifier), ∇, $\wedge$ (conjunction), $\vee$ (disjunction), $\supset$ (implication), $\top$ (true) and $\bot$ (false). The inference rules for the quantifiers of $FO\lambda^{\Delta\nabla}$ are given in Figure 1. The complete set of inference rules can be found in the appendix. Since we do not allow quantification over predicates, this logic is proof-theoretically similar to first-order logic (hence, the letters FO in $FO\lambda^{\Delta\nabla}$).

$$\frac{\Sigma, \sigma \vdash t : \gamma \qquad \Sigma\,;\, \sigma \triangleright B[t/x], \Gamma \vdash \mathcal{C}}{\Sigma\,;\, \sigma \triangleright \forall_\gamma x.B, \Gamma \vdash \mathcal{C}}\ \forall\mathcal{L} \qquad \frac{\Sigma, h\,;\, \Gamma \vdash \sigma \triangleright B[(h\ \sigma)/x]}{\Sigma\,;\, \Gamma \vdash \sigma \triangleright \forall x.B}\ \forall\mathcal{R}$$

$$\frac{\Sigma, h\,;\, \sigma \triangleright B[(h\ \sigma)/x], \Gamma \vdash \mathcal{C}}{\Sigma\,;\, \sigma \triangleright \exists x.B, \Gamma \vdash \mathcal{C}}\ \exists\mathcal{L} \qquad \frac{\Sigma, \sigma \vdash t : \gamma \qquad \Sigma\,;\, \Gamma \vdash \sigma \triangleright B[t/x]}{\Sigma\,;\, \Gamma \vdash \sigma \triangleright \exists_\gamma x.B}\ \exists\mathcal{R}$$

$$\frac{\Sigma\,;\, (\sigma, y) \triangleright B[y/x], \Gamma \vdash \mathcal{C}}{\Sigma\,;\, \sigma \triangleright \nabla x\ B, \Gamma \vdash \mathcal{C}}\ \nabla\mathcal{L} \qquad \frac{\Sigma\,;\, \Gamma \vdash (\sigma, y) \triangleright B[y/x]}{\Sigma\,;\, \Gamma \vdash \sigma \triangleright \nabla x\ B}\ \nabla\mathcal{R}$$

Fig. 1. The introduction rules for quantifiers of $FO\lambda^{\Delta\nabla}$.

During the search for proofs (reading rules bottom up), inference rules for $\forall$ and $\exists$ quantifier place new eigenvariables into the global signature while the inference rules for ∇ place them into the local signature. In the $\forall\mathcal{R}$ and $\exists\mathcal{L}$ rules, raising [23] is used when moving the bound variable x, which can range over the variables in both the global signature and the local signature σ, with the variable h that can only range over variables in the global signature: so as not to miss substitution terms, the variable x is replaced by the term $(h\, x_1 \ldots x_n)$, which we shall write simply as $(h\, \sigma)$, where σ is the list $x_1, \ldots, x_n$ (h must not be free in the lower sequent of these rules). In $\forall\mathcal{L}$ and $\exists\mathcal{R}$, the term t can have free variables from both Σ and σ. This is presented in the rule by the typing judgment $\Sigma, \sigma \vdash t : \tau$. The $\nabla\mathcal{L}$ and $\nabla\mathcal{R}$ rules have the proviso that y is not free in $\nabla x\ B$.

Note that the use of raising in $\forall\mathcal{R}$ and $\exists\mathcal{L}$ means that the eigenvariable introduced might not be of the same type as the quantified variable. This is illustrated in the following example.

$$\dfrac{\dfrac{\{x : \alpha, h : \tau \to \gamma \to \beta\}\,;\, \Gamma \vdash (a : \tau, b : \gamma) \triangleright B\ (h\, a\, b)\ b}{\{x : \alpha\}\,;\, \Gamma \vdash (a : \tau, b : \gamma) \triangleright \forall_\beta y.B\ y\ b}\ \forall\mathcal{L}}{\{x : \alpha\}\,;\, \Gamma \vdash (a : \tau) \triangleright \nabla_\gamma z.\forall_\beta y.B\ y\ z}\ \nabla\mathcal{R}$$

Notice that the quantified variable y is of type β while its corresponding eigenvariable h is raised to the type $\tau \to \gamma \to \beta$, taking into account its possible dependency on $a : \tau$ and $b : \gamma$.

The standard inference rules of logic express introduction rules for logical constants. The full logic $FO\lambda^{\Delta\nabla}$ additionally allows introduction of atomic judgments, that is, judgments which do not contain any occurrences of logical constants. To each atomic judgment, $\mathcal{A}$, we associate a defining judgment, $\mathcal{B}$, the *definition* of $\mathcal{A}$. The introduction rule for the judgment $\mathcal{A}$ is in effect done by replacing $\mathcal{A}$ with $\mathcal{B}$ during proof search. This notion of definitions is an extension of work by Schroeder-Heister [39], Eriksson [7], Girard [12], Stärk [40] and McDowell and Miller [20]. These inference rules for definitions allow for modest reasoning about the fixed points of definitions.

Definition 1. *A* definition clause *is written* $\forall\bar{x}[p\,\bar{t} \stackrel{\triangle}{=} B]$*, where* p *is a predicate constant, every free variable of the formula* B *is also free in at least one term in the list* $\bar{t}$ *of terms, and all variables free in* $p\,\bar{t}$ *are contained in the list* $\bar{x}$ *of variables. The atomic formula* $p\,\bar{t}$ *is called the* head *of the clause, and the formula* B *is called the* body*. The symbol* $\stackrel{\triangle}{=}$ *is used simply to indicate a definitional clause: it is not a logical connective. The predicate* p *occurs strictly positively in* B*, that is, it does not occur to the left of any* $\supset$ *(implication).*

Let $\forall_{\tau_1} x_1 \ldots \forall_{\tau_n} x_n.H \stackrel{\triangle}{=} B$ *be a definition clause. Let* $y_1, \ldots, y_m$ *be a list of variables of types* $\alpha_1, \ldots, \alpha_m$*, respectively. The* raised definition clause *of* H *with respect to the signature* $\{y_1 : \alpha_1, \ldots, y_m : \alpha_m\}$ *is defined as*

$$\forall h_1 \ldots \forall h_n.\bar{y} \triangleright H\theta \stackrel{\triangle}{=} \bar{y} \triangleright B\theta$$

where θ *is the substitution* $[(h_1\,\bar{y})/x_1, \ldots, (h_n\,\bar{y})/x_n]$ *and* h_i*, for every* $i \in \{1, \ldots, n\}$*, is of type* $\alpha_1 \to \ldots \to \alpha_m \to \tau_i$*. A* definition *is a set of definition clauses together with their raised clauses.*

To guarantee the consistency (and cut-elimination) of the logic $FO\lambda^{\Delta\nabla}$, we need some kind of stratification of definition that limits the definition of one predicate to depend negatively on another predicate. We shall not give the full technical details here and but instead refer the interested reader to [26]. All definitions considered in this paper are stratified appropriately and cut-elimination will hold for the logic using them.

The introduction rules for a defined judgment are as follow. When applying the introduction rules, we shall omit the outer quantifiers in a definition clause and assume implicitly that the free variables in the definition clause are distinct from other variables in the sequent.

$$\frac{\{\Sigma\theta\,;\, \mathcal{B}\theta, \Gamma\theta \vdash \mathcal{C}\theta \mid \theta \in CSU(\mathcal{A}, \mathcal{H}) \text{ for some clause } \mathcal{H} \stackrel{\triangle}{=} \mathcal{B}\}}{\Sigma\,;\, \mathcal{A}, \Gamma \vdash \mathcal{C}}\ def\mathcal{L}$$

$$\frac{\Sigma\,;\, \Gamma \vdash \mathcal{B}\theta}{\Sigma\,;\, \Gamma \vdash \mathcal{A}}\ def R, \quad \text{where } \mathcal{H} \stackrel{\triangle}{=} \mathcal{B} \text{ is a definition clause and } \mathcal{H}\theta = \mathcal{A}$$

In the above rules, we apply substitution to judgments. The result of applying a substitution θ to a generic judgment $x_1, \ldots, x_n \triangleright B$, written as $(x_1, \ldots, x_n \triangleright B)\theta$, is $y_1, \ldots, y_n \triangleright B'$, if $(\lambda x_1 \ldots \lambda x_n.B)\theta$ is equal (modulo λ-conversion) to $\lambda y_1 \ldots \lambda y_n.B'$. If Γ is a multiset of generic judgments, then $\Gamma\theta$ is the multiset $\{J\theta \mid J \in \Gamma\}$. In the *def*$\mathcal{L}$ rule, we use the notion of *complete set of unifiers* (CSU) [19]. We denote by $CSU(\mathcal{A}, \mathcal{H})$ the complete set of unifiers for the pair $(\mathcal{A}, \mathcal{H})$, that is, for any substitution θ such that $\mathcal{A}\theta = \mathcal{H}\theta$, there is a substitution $\rho \in CSU(\mathcal{A}, \mathcal{H})$ such that $\theta = \rho \circ \theta'$ for some substitution θ'. In all the applications of *def*$\mathcal{L}$ in this paper, the set $CSU(\mathcal{A}, \mathcal{H})$ is either empty (the two judgments are not unifiable) or contains a single denoting the most general unifier. The signature $\Sigma\theta$ in *def*$\mathcal{L}$ denotes a signature obtained from Σ by removing the variables in the domain of θ and adding the variables in the range of θ. In the *def*$\mathcal{L}$ rule, reading the rule bottom-up, eigenvariables can be instantiated in the premise, while in the *def*R rule, eigenvariables are not instantiated. The set that is the premise of the *def*$\mathcal{L}$ rule means that that rule instance has a premise for every member of that set: if that set is empty, then the premise is proved.

One might find the following analogy with logic programming helpful: if a definition is viewed as a logic program, then the *def*R rule captures backchaining and the *def*$\mathcal{L}$ rule corresponds to *case analysis* on all possible ways an atomic judgment could be proved. In the case where the program has only finitely many computation paths, we can effectively encode *negation-as-failure* using *def*$\mathcal{L}$ [13].

3 Some meta-theory of the meta-logic

We now illustrate how the structural properties of proofs can be used for meta-reasoning about the logical specifications mentioned previously. The general reasoning scheme makes use of the cut rule and the cut-elimination theorem. Cut-elimination says that any proof which makes use of the cut rule can be transformed to a proof without it (a *cut-free* proof). Cut-elimination gives rise

to surprisingly rich structural properties of proofs. One important structural property is that of the invertibility of inference rules. An inference rule of logic is *invertible* if the provability of the conclusion implies the provability of the premise(s) of the rule. The following rules in $FO\lambda^{\Delta\nabla}$ are invertible: $\wedge\mathcal{R}, \wedge\mathcal{L}, \vee\mathcal{L}, \supset \mathcal{R}, \forall\mathcal{R}, \exists\mathcal{L}, def\mathcal{L}$ (see [41] for a proof). Knowing the invertibility of a rule can be useful in determining some structure of a proof. For example, if we know that a sequent $A \vee B, \Gamma \vdash C$ is provable, then by the invertibility of $\vee\mathcal{L}$, we know that it must be the case that $A, \Gamma \vdash C$ and $B, \Gamma \vdash C$ are provable.

Another important property of $FO\lambda^{\Delta\nabla}$ is that concerning the local signatures. Local signatures can be *weakened* without affecting provability.

Proposition 2. *If B is provable and x is not free in B, then $\nabla x B$ is provable.*

Proposition 3. *If $\forall x B$ is provable then $\nabla x B$ is provable.*

One might expect the implication $\forall_\tau x B \supset \nabla_\tau x B$ to hold as well. Notice that if τ is empty this statement would not be expected to be true and, hence, we do not accept it in the core logic.

The converse of Proposition 3 is not true in general. However, it is true for a restricted class of formulas and definitions, called $\mathsf{hc}^{\forall\nabla}$-*formulas* (for Horn clauses with $\forall$ and ∇) and $\mathsf{hc}^{\forall\nabla}$-*definitions*, respectively. A $\mathsf{hc}^{\forall\nabla}$-formula is a formula which does not contain any occurrence of the logical constant $\supset$ (implication). A $\mathsf{hc}^{\forall\nabla}$-definition is a definition whose bodies are $\mathsf{hc}^{\forall\nabla}$-formulas. One of the examples of $\mathsf{hc}^{\forall\nabla}$-definitions is the definition for the one-step transition in Figure 2.

Proposition 4. *Let $\mathcal{D}$ be a $\mathsf{hc}^{\forall\nabla}$-definition and $\forall x G$ be a $\mathsf{hc}^{\forall\nabla}$-formula. Then $\forall x G$ is provable if and only if $\nabla x G$ is provable.*

4 Logical specification of one-step transition

We consider the late transition system for the finite π-calculus as defined in [27], that is, the fragment of π-calculus without recursion (or replication). The syntax of processes is defined as follows

$$\mathtt{P} ::= 0 \mid \bar{x}y.\mathtt{P} \mid x(y).\mathtt{P} \mid \tau.\mathtt{P} \mid (x)\mathtt{P} \mid [x=y]\mathtt{P} \mid \mathtt{P}|\mathtt{Q} \mid \mathtt{P}+\mathtt{Q}.$$

We use the notation $\mathtt{P}$, $\mathtt{Q}$, $\mathtt{R}$, $\mathtt{S}$ and $\mathtt{T}$ to denote processes. Names are denoted by lower case letters, e.g., a, b, c, d, x, y, z. The occurrence of y in the process $x(y).\mathtt{P}$ and $(y)\mathtt{P}$ is a binding occurrence, with $\mathtt{P}$ as its scope. The set of free names in $\mathtt{P}$ is denoted by $\mathrm{fn}(\mathtt{P})$, the set of bound names is denoted by $\mathrm{bn}(\mathtt{P})$. We write $\mathrm{n}(\mathtt{P})$ for the set $\mathrm{fn}(\mathtt{P}) \cup \mathrm{bn}(\mathtt{P})$. We consider processes to be syntactical equivalent up to renaming of bound names.

One-step transition in the π-calculus is denoted by $\mathtt{P} \xrightarrow{\alpha} \mathtt{Q}$, where $\mathtt{P}$ and $\mathtt{Q}$ are processes and α is an action. The kinds of actions are *the silent action* τ, *the free input action* xy, *the free output action* $\bar{x}y$, *the bound input action* $x(y)$ and *the bound output action* $\bar{x}(y)$. The name y in $x(y)$ and $\bar{x}(y)$ is a binding occurrence. Just like we did with processes, we use $\mathrm{fn}(\alpha)$, $\mathrm{bn}(\alpha)$ and $\mathrm{n}(\alpha)$ to denote free names, bound names, and names in α. An action without binding occurrences of names is a *free action*, otherwise it is a *bound action*.

We encode the syntax of process expressions using higher-order syntax as follows. We shall require three primitive syntactic categories: n for names, p for processes, and a for actions, and the constructors corresponding to the operators in π-calculus. We do not assume any inhabitants of type n, therefore in our encoding a free name is translated to a variable of type n, which can later be either universally quantified or ∇-quantified, depending on whether we want to treat a certain name as instantiable or not. (Since the rest of this paper is about the π-calculus, the ∇ quantifier

will from now on only be used at type n.) For instance, in encoding late bisimulation (Section 5) we treat free names as ∇-quantified variables, while in the encoding of open bisimulation they are universally quantified variables. To encode actions, we use $\tau : a$ (for the silent action), and the two constants $\downarrow$ and $\uparrow$, both of type $n \to n \to a$ for building input and output actions. The free output action $\bar{x}y$, is encoded as $\uparrow xy$ while the bound output action $\bar{x}(y)$ is encoded as $\lambda y\ (\uparrow xy)$ (or the η-equivalent term $\uparrow x$). The free input action xy, is encoded as $\downarrow xy$ while the bound input action $x(y)$ is encoded as $\lambda y\ (\downarrow xy)$ (or simply $\downarrow x$). The process constructors are encoded using the following constants

$$\begin{array}{c} 0 : p, \qquad \tau : p \to p, \qquad out : n \to n \to p \to p, \qquad in : n \to (n \to p) \to p, \\ + : p \to p \to p, \quad | : p \to p \to p, \quad match : n \to n \to p \to p, \quad \nu : (n \to p) \to p. \end{array}$$

We use two predicates to encode the one-step transition semantics for the π-calculus. The predicate $\cdot \stackrel{\cdot}{\longrightarrow} \cdot$ of type $p \to a \to p \to o$ encodes transitions involving free values and the predicate $\cdot \stackrel{\cdot}{\longrightarrow\!\!\!\!\searrow} \cdot$ of type $p \to (n \to a) \to (n \to p) \to o$ encodes transitions involving bound values. The precise translation of π-calculus syntax into simply typed λ-terms is given in the following definition.

Definition 5. *The following function $\langle . \rangle$ translates from process expressions to $\beta\eta$-long normal terms of type p.*

$$\begin{array}{llll} \langle 0 \rangle = 0 & \langle \tau.\mathtt{P} \rangle = \tau\ \langle \mathtt{P} \rangle & \langle \bar{x}y.\mathtt{P} \rangle = out\ x\ y\ \langle \mathtt{P} \rangle & \langle x(y).\mathtt{P} \rangle = in\ x\ \lambda y.\langle \mathtt{P} \rangle \\ \langle \mathtt{P} + \mathtt{Q} \rangle = \langle \mathtt{P} \rangle + \langle \mathtt{Q} \rangle & \langle \mathtt{P} | \mathtt{Q} \rangle = \langle \mathtt{P} \rangle \,|\, \langle \mathtt{Q} \rangle & \langle [x = y]\mathtt{P} \rangle = match\ x\ y\ \langle \mathtt{P} \rangle & \langle (x)\mathtt{P} \rangle = \nu\lambda x.\langle \mathtt{P} \rangle \end{array}$$

The one-step transition judgments are translated to atomic formulas as follows (we overload the symbol $\langle . \rangle$).

$$\begin{array}{ll} \langle \mathtt{P} \stackrel{\bar{x}y}{\longrightarrow} \mathtt{Q} \rangle = \langle \mathtt{P} \rangle \stackrel{\uparrow xy}{\longrightarrow} \langle \mathtt{Q} \rangle & \langle \mathtt{P} \stackrel{\tau}{\longrightarrow} \mathtt{Q} \rangle = \langle \mathtt{P} \rangle \stackrel{\tau}{\longrightarrow} \langle \mathtt{Q} \rangle \\ \langle \mathtt{P} \stackrel{x(y)}{\longrightarrow} \mathtt{Q} \rangle = \langle \mathtt{P} \rangle \stackrel{\downarrow x}{\longrightarrow\!\!\!\!\searrow} \lambda y.\langle \mathtt{Q} \rangle & \langle \mathtt{P} \stackrel{\bar{x}(y)}{\longrightarrow} \mathtt{Q} \rangle = \langle \mathtt{P} \rangle \stackrel{\uparrow x}{\longrightarrow\!\!\!\!\searrow} \lambda y.\langle \mathtt{Q} \rangle \end{array}$$

We abbreviate $\nu\lambda x.P$ as simply $\nu x.P$. Notice that when τ is written as a prefix, it has type $p \to p$, and when it is written as an action, it has type a.

The operational semantics of the late transition system for the finite π-calculus is given as a definition, called $\mathbf{D}_\pi$, in Figure 2. In this specification, free variables are schema variables that are assumed to be universally scoped over the inference rule or definitional clause in which it appears. These schema variables have primitive types such as a, n, and p as well as functional types such as $n \to a$ and $n \to p$.

Notice that the complicated side conditions in the original specification of π-calculus [27] are no longer present, as they are now treated directly and declaratively by the meta-logic. For example, the side condition that $X \neq y$ in the open rule is implicit, since X is outside the scope of y and therefore cannot be instantiated with y. However, this is not a feature that is peculiar to $FO\lambda^{\Delta\nabla}$; rather it is a consequence of the use of HOAS in the encoding. The adequacy of our encoding is stated in the following lemma and proposition (their proofs can be found in [41]).

Lemma 6. *The function $\langle . \rangle$ is a bijection.*

Proposition 7. *Let $\mathtt{P}$ and $\mathtt{Q}$ be processes and α an action. Let $\bar{n}$ be a list of free names containing the free names in $\mathtt{P}$, $\mathtt{Q}$ and α. The transition $\mathtt{P} \stackrel{\alpha}{\longrightarrow} \mathtt{Q}$ is derivable in π-calculus if and only if the sequent $.\,;\, . \vdash \nabla\bar{n}.\langle \mathtt{P} \stackrel{\alpha}{\longrightarrow} \mathtt{Q} \rangle$ is provable in $FO\lambda^{\Delta\nabla}$ with the definition $\mathbf{D}_\pi$.*

(tau:) $\tau\ P \xrightarrow{\tau} P \triangleq \top.$ (in:) $in\ X\ M \xrightarrow{\downarrow X} M \triangleq \top.$ (out:) $out\ x\ y\ P \xrightarrow{\uparrow xy} P \triangleq \top.$

(match:) $match\ x\ x\ P \xrightarrow{A} Q \triangleq P \xrightarrow{A} Q.$ $match\ x\ x\ P \xrightarrow{A} Q \triangleq P \xrightarrow{A} Q.$

(sum:) $P + Q \xrightarrow{A} R \triangleq P \xrightarrow{A} R.$ $P + Q \xrightarrow{A} R \triangleq Q \xrightarrow{A} R.$

(sum:) $P + Q \xrightarrow{A} R \triangleq P \xrightarrow{A} R.$ $P + Q \xrightarrow{A} R \triangleq Q \xrightarrow{A} R.$

(par:) $P \mid Q \xrightarrow{A} P' \mid Q \triangleq P \xrightarrow{A} P'.$ $P \mid Q \xrightarrow{A} P \mid Q' \triangleq Q \xrightarrow{A} Q'$

(par:) $P \mid Q \xrightarrow{A} \lambda n(M\,n \mid Q) \triangleq P \xrightarrow{A} M.$ $P \mid Q \xrightarrow{A} \lambda n(P \mid N\,n) \triangleq Q \xrightarrow{A} N.$

(res:) $\nu n.Pn \xrightarrow{A} \nu n.Qn \triangleq \nabla n(Pn \xrightarrow{A} Qn).$ $\nu n.Pn \xrightarrow{A} \lambda m\ \nu n.P'nm \triangleq \nabla n(Pn \xrightarrow{A} P'n).$

(open:) $\nu y.My \xrightarrow{\uparrow X} M' \triangleq \nabla y(My \xrightarrow{\uparrow Xy} M'y).$

(close:) $P \mid Q \xrightarrow{\tau} \nu y.My \mid Ny \triangleq \exists X.P \xrightarrow{\downarrow X} M \wedge Q \xrightarrow{\uparrow X} N$

(close:) $P \mid Q \xrightarrow{\tau} \nu y.My \mid Ny \triangleq \exists X.P \xrightarrow{\uparrow X} M \wedge Q \xrightarrow{\downarrow X} N.$

(com:) $P \mid Q \xrightarrow{\tau} MY \mid Q' \triangleq \exists X.P \xrightarrow{\downarrow X} M \wedge Q \xrightarrow{\uparrow XY} Q'$

(com:) $P \mid Q \xrightarrow{\tau} P' \mid NY \triangleq \exists X.P \xrightarrow{\uparrow XY} P' \wedge Q \xrightarrow{\downarrow X} N$

Fig. 2. Definition clauses for the late transition system.

Since the definition $\mathbf{D}_\pi$ contains essentially Horn clauses, a consequence Proposition 4 is that if any of the ∇-bound variables in $\nabla\bar{n}.\langle \mathtt{P} \xrightarrow{\alpha} \mathtt{Q}\rangle$ are changed to be $\forall$-bound variables, the resulting formula is still provable. The differences between ∇ and $\forall$ are revealed more with non-Horn definitions, such as those for bisimulation.

Given the above adequacy results, we shall omit writing explicitly the function symbol $\langle . \rangle$ when referring to p-term obtained via the translation.

The restriction operator is interpreted at the meta-level as the ∇ quantifier. The use of ∇, instead of $\forall$, allows to prove *negative* statements about the transitions, as illustrated in Example 8. When writing encoded process expressions, we shall use the syntax of π-calculus along with the usual abbreviations: for example, when a name z is used as a prefix, it denotes the prefix $z(w)$ where w is vacuous in its scope; when a name $\bar{z}$ is used as a prefix it denotes the output prefix $\bar{z}a$ for some fixed name a. We also abbreviate $(y)\bar{x}y.P$ as $\bar{x}(y).P$ and the process term 0 is omitted if it appears as the continuation of a prefix. We assume that the operators $\mid$ and $+$ associate the right, e.g., we write $P + Q + R$ to denote $P + (Q + R)$.

Example 8. Consider the process $(y)([x = y]\bar{x}z)$. This process cannot make any transition since the bound variable y denotes a name different from x. One can think of this process as a continuation of some other process which inputs x on some channel, e.g., $a(x).(y)[x = y]\bar{x}z$. We would therefore expect that the following is provable.

$$\forall x \forall z \forall Q \forall \alpha.[((y)[x = y](\bar{x}z) \xrightarrow{\alpha} Q) \supset \bot]$$

This type of statement naturally occurs when one is asking whether two processes are bisimilar (see Section 5), where it is necessary to know what transitions a process can make and what it cannot. The scoping constraint between y and x is captured properly by the alternation of $\forall$ and ∇. Notice that in the above specification, y is inside the scope of x, which means that whatever value we substitute for x cannot be equal to y (since substitution is capture-avoiding). The formal

derivation of the above formula is (ignoring the terminal uses of $\supset\mathcal{R}$ and $\forall\mathcal{R}$):

$$\dfrac{\dfrac{\dfrac{}{\{x,z,Q,\alpha\}\,;\, y \triangleright ([x=y](\bar{x}z.0) \xrightarrow{\alpha} Q) \vdash \bot}\, def\mathcal{L}}{\{x,z,Q,\alpha\}\,;\, . \triangleright \nabla y.([x=y](\bar{x}z.0) \xrightarrow{\alpha} Q) \vdash \bot}\, \nabla\mathcal{L}}{\{x,z,Q,\alpha\}\,;\, . \triangleright ((y)[x=y](\bar{x}z.0) \xrightarrow{\alpha} Q) \vdash \bot}\, def\mathcal{L}$$

The success of the topmost instance of $def\mathcal{L}$ depends on the failure of the unification problem $\lambda y.x = \lambda y.y$. Notice that the scoping of object variables is maintained at the meta-level by the separation of (global) eigenvariables and (locally bound) generic variables. The "newness" of y is internalized as a λ-abstraction and, hence, it is not subject to instantiation.

5 Logical specifications of strong bisimilarity

We consider specifying two notions of bisimilarity, tied to the late transition system: the strong late bisimilarity and the strong open bisimilarity. As we shall see, the essential difference between the specification of late bisimulation and that of open bisimulation is in the presence (or the absence) of the axiom of excluded middle on names. The original definitions of late and open bisimilarity are given in [27, 38]. Here we choose to make the side conditions explicit, instead of adopting the bound variable convention in [38].

Definition 9. Strong late bisimilarity *is the largest symmetric relation, $\sim_l$, such that whenever* $\mathtt{P} \sim_l \mathtt{Q}$,

1. *if* $\mathtt{P} \xrightarrow{\alpha} \mathtt{P}'$ *and α is a free action, then there is* $\mathtt{Q}'$ *such that* $\mathtt{Q} \xrightarrow{\alpha} \mathtt{Q}'$ *and* $\mathtt{P}' \sim_l \mathtt{Q}'$,
2. *if* $\mathtt{P} \xrightarrow{x(z)} \mathtt{P}'$ *and* $z \notin \mathrm{n}(\mathtt{P},\mathtt{Q})$ *then there is* $\mathtt{Q}'$ *such that* $\mathtt{Q} \xrightarrow{x(z)} \mathtt{Q}'$ *and* $\mathtt{P}'[y/z] \sim_l \mathtt{Q}'[y/z]$ *for every name y,*
3. *if* $\mathtt{P} \xrightarrow{\bar{x}(z)} \mathtt{P}'$ *and* $z \notin \mathrm{n}(\mathtt{P},\mathtt{Q})$ *then there is* $\mathtt{Q}'$ *such that* $\mathtt{Q} \xrightarrow{\bar{x}(z)} \mathtt{Q}'$ *and* $\mathtt{P}' \sim_l \mathtt{Q}'$.

Definition 10. *A* distinction *D is a finite symmetric and irreflexive relation on names. A substitution θ* respects *a distinction D if $(x,y) \in D$ implies $x\theta \neq y\theta$. We refer to the substitution θ as a D-*substitution*. Given a distinction D and a D-substitution θ, the result of applying θ to all variables in D, written $D\theta$, is another distinction. We denote with* $\mathrm{fn}(D)$ *the set of free names occuring in D.*

Definition 11. Strong open bisimilarity $\{\sim_o^D |\ D$ *a distinction* $\}$ *is the largest family of symmetric relations such that if* $\mathtt{P} \sim_o^D \mathtt{Q}$ *and θ respects D, then*

1. *if* $\mathtt{P}\theta \xrightarrow{\alpha} \mathtt{P}'$ *and α is a free action, then there is* $\mathtt{Q}'$ *such that* $\mathtt{Q}\theta \xrightarrow{\alpha} \mathtt{Q}'$ *and* $\mathtt{P}' \sim_o^{D\theta} \mathtt{Q}'$,
2. *if* $\mathtt{P}\theta \xrightarrow{x(z)} \mathtt{P}'$ *and* $z \notin \mathrm{n}(\mathtt{P}\theta,\mathtt{Q}\theta)$ *then there is* $\mathtt{Q}'$ *such that* $\mathtt{Q}\theta \xrightarrow{x(z)} \mathtt{Q}'$ *and* $\mathtt{P}' \sim_o^{D\theta} \mathtt{Q}'$,
3. *if* $\mathtt{P}\theta \xrightarrow{\bar{x}(z)} \mathtt{P}'$ *and* $z \notin \mathrm{n}(\mathtt{P}\theta,\mathtt{Q}\theta)$ *then there is* $\mathtt{Q}'$ *such that* $\mathtt{Q}\theta \xrightarrow{\bar{x}(z)} \mathtt{Q}'$ *and* $\mathtt{P}' \sim_o^{D'} \mathtt{Q}'$ *where* $D' = D\theta \cup (\{z\} \times \mathrm{fn}(\mathtt{P}\theta,\mathtt{Q}\theta)) \cup (\{z\} \times \mathrm{fn}(D\theta))$.

Note that we strengthen a bit the condition 3 in Definition 11 to include the distinction $(\{z\} \times \mathrm{fn}(D\theta))$. Strengthening the distinction this way does not change the open bisimilarity, as noted in [38], but in our encoding of open bisimulation, the distinction D is part of the specification and the modified definition above helps us account for names better.

$$
\begin{aligned}
\mathit{lbisim}\ P\ Q \triangleq\ & \forall A \forall P'\,[(P \xrightarrow{A} P') \supset \exists Q'.(Q \xrightarrow{A} Q') \wedge \mathit{lbisim}\ P'\ Q'] \wedge \\
& \forall A \forall Q'\,[(Q \xrightarrow{A} Q') \supset \exists P'.(P \xrightarrow{A} P') \wedge \mathit{lbisim}\ Q'\ P'] \wedge \\
& \forall X \forall P'\,[(P \xrightarrow{\downarrow X} P') \supset \exists Q'.(Q \xrightarrow{\downarrow X} Q') \wedge \forall w.\mathcal{E}w \supset \mathit{lbisim}\ (P'w)\ (Q'w)] \wedge \\
& \forall X \forall Q'\,[(Q \xrightarrow{\downarrow X} Q') \supset \exists P'.(P \xrightarrow{\downarrow X} P') \wedge \forall w.\mathcal{E}w \supset \mathit{lbisim}\ (Q'w)\ (P'w)] \wedge \\
& \forall X \forall P'\,[(P \xrightarrow{\uparrow X} P') \supset \exists Q'.(Q \xrightarrow{\uparrow X} Q') \wedge \nabla w.\mathit{lbisim}\ (P'w)\ (Q'w)] \wedge \\
& \forall X \forall Q'\,[(Q \xrightarrow{\uparrow X} Q') \supset \exists P'.(P \xrightarrow{\uparrow X} P') \wedge \nabla w.\mathit{lbisim}\ (Q'w)\ (P'w)]
\end{aligned}
$$

Fig. 3. Specification of late bisimulation. Here, $\mathcal{E} = \lambda w\ \forall z\ (w = z \vee w \neq z)$.

$$
\begin{aligned}
\mathit{obisim}\ P\ Q \triangleq\ & \forall A \forall P'\,[(P \xrightarrow{A} P') \supset \exists Q'.(Q \xrightarrow{A} Q') \wedge \mathit{obisim}\ P'\ Q'] \wedge \\
& \forall A \forall Q'\,[(Q \xrightarrow{A} Q') \supset \exists P'.(P \xrightarrow{A} P') \wedge \mathit{obisim}\ Q'\ P'] \wedge \\
& \forall X \forall P'\,[(P \xrightarrow{\downarrow X} P') \supset \exists Q'.(Q \xrightarrow{\downarrow X} Q') \wedge \forall w.\mathit{obisim}\ (P'w)\ (Q'w)] \wedge \\
& \forall X \forall Q'\,[(Q \xrightarrow{\downarrow X} Q') \supset \exists P'.(P \xrightarrow{\downarrow X} P') \wedge \forall w.\mathit{obisim}\ (Q'w)\ (P'w)] \wedge \\
& \forall X \forall P'\,[(P \xrightarrow{\uparrow X} P') \supset \exists Q'.(Q \xrightarrow{\uparrow X} Q') \wedge \nabla w.\mathit{obisim}\ (P'w)\ (Q'w)] \wedge \\
& \forall X \forall Q'\,[(Q \xrightarrow{\uparrow X} Q') \supset \exists P'.(P \xrightarrow{\uparrow X} P') \wedge \nabla w.\mathit{obisim}\ (Q'w)\ (P'w)]
\end{aligned}
$$

Fig. 4. Specification of open bisimulation.

The corresponding specifications for late and open bisimulation in $FO\lambda^{\Delta\nabla}$ are given in Figure 3 and Figure 4. The specifications make use of the syntactic equality predicate, which is defined as the definition: $X = X \triangleq \top$. Note that the symbol $=$ here is a predicate symbol written in infix notation. The inequality $x \neq y$ is an abbreviation for $x = y \supset \bot$. Actually the specifications shown in the figures do not readily encode bisimulations, since they do not yet address the notion of distinction among names. The notion of distinction will be addressed later. For the moment it is enough to note that when reasoning about the specification of late bisimulation, we encode free names as ∇-quantified variables whereas in the specification of open bisimulation we encode free names as $\forall$-quantified variables. For example, the processes $Pxy = (x|\bar{y})$ and $Qxy = (x.\bar{y} + \bar{y}.x)$ are late bisimilar. The corresponding encoding in $FO\lambda^{\Delta\nabla}$ would be $\nabla x \nabla y.\mathit{lbisim}\ (Pxy)\ (Qxy)$. The free names x and y should not be $\forall$-quantified, for the obvious reason: in logic we have the implication $\forall x \forall y\ \mathit{lbisim}\ (Pxy)\ (Qxy) \supset \forall z\ \mathit{lbisim}\ (Pzz)\ (Qzz)$. That is, either $\forall x \forall y\ \mathit{lbisim}\ (Pxy)\ (Qxy)$ is not provable, or it is provable and we have a proof of $\forall x\ \mathit{lbisim}\ (Pzz)\ (Qzz)$. In either case we lose the adequacy of the encoding.

Notice that in the specification of late bisimulation, we use the axiom of excluded middle on names while in the open case we do not. For example, the two processes (taken from [37])

$$P = x(u).(\tau.\tau + \tau), \qquad Q = x(u).(\tau.\tau + \tau + \tau.[u = z]\tau)$$

are late bisimilar but not open bisimilar: last bisimulation makes use of doing a case analysis on names. Since the meta-logic $FO\lambda^{\Delta\nabla}$ is intuitionistic, this difference between late and open bisimulation is easily observed. This would not be the case if the meta-logic were classical.

The following theorem states the soundness and completeness of the *lbisim* specification with respect to the notion of late bisimilarity in π-calculus. By soundness we mean that, given a pair of processes `P` and `Q`, if the encoding of the late bisimilarity, $\nabla \bar{n}.\mathit{lbisim}$ `P` `Q`, is provable in $FO\lambda^{\Delta\nabla}$ then the processes `P` and `Q` are late bisimilar. Completeness is the converse. The soundness and completeness of the open bisimilarity encoding is presented in at the end of this section, where we consider the encoding of the notion of distinction in π-calculus. Proofs of these results and their

associated lemmas can be found in the appendices of an extended version of this paper on the authors' web pages.

Theorem 12. *Let* P *and* Q *be two processes and let* $\bar{n}$ *be the free names in* P *and* Q*. The formula* $\nabla\bar{n}.lbisim$ P Q *is provable if and only if* P $\sim_l$ Q*.*

It is well-known that the late bisimulation relation is not a congruence since it is not preserved by the input prefix. Part of the reason why the congruence property fails is that in the late bisimilarity there is no syntactic distinction made between instantiable names and non-instantiable names. This is one of the motivations behind the introduction of the notion of distinction and open bisimulation. The alternation of quantifiers in $FO\lambda^{\Delta\nabla}$ gives rise to a particular kind of distinction, the precise definition of which is given below.

Definition 13. *A* quantifier prefix *is a list* $\mathcal{Q}_1x_1\mathcal{Q}_2x_2\ldots\mathcal{Q}_nx_n$ *for some* $n \geq 0$*, where* $\mathcal{Q}_i$ *is either* ∇ *or* $\forall$*. Let* $\mathcal{Q}\bar{x}$ *be the above quantifier prefix. A* $\mathcal{Q}\bar{x}$-distinction *is the distinction*

$$\{(x_i, x_j), (x_j, x_i) \mid i \neq j \text{ and } \mathcal{Q}_i = \mathcal{Q}_j = \nabla, \text{ or } i < j \text{ and } \mathcal{Q}_i = \forall \text{ and } \mathcal{Q}_j = \nabla\}.$$

Notice that if $\mathcal{Q}\bar{x}$ consists only of universal quantifiers then the $\mathcal{Q}\bar{x}$-distinction is empty. Obviously, the alternation of quantifiers does not capture all possible distinction, e.g., the distinction $\{(x,y),(y,x),(x,z),(z,x),(u,z),(z,u)\}$ does not correspond to any quantifier prefix. However, we can encode the full notion of distinction by explicit encoding of the inequal pairs, as shown later.

Definition 14. *Let* $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ *be a distinction. We define a translation from a distinction* D *to a formula* $[\![D]\!]$ *as follows* $[\![D]\!] = x_1 \neq y_1 \wedge \ldots \wedge x_n \neq y_n$*. If* $n = 0$ *then* $[\![D]\!]$ *is the logical constant* $\top$ *(empty conjunction).*

Theorem 15. *Let* P *and* Q *be two processes, let* D *be a distinction and let* $\mathcal{Q}\bar{x}$ *be a quantifier prefix, where* $\bar{x}$ *contains the free names in* P, Q *and* D*. If the formula* $\mathcal{Q}\bar{x}.([\![D]\!] \supset obisim$ P Q$)$ *is provable then* P $\sim_o^{D'}$ Q*, where* D' *is the union of* D *and the* $\mathcal{Q}\bar{x}$*-distinction.*

Theorem 16. *If* P $\sim_o^D$ Q *then the formula* $\forall\bar{x}.[\![D]\!] \supset obisim$ P Q*, where* $\bar{x}$ *are the free names in* P, Q *and* D*, is provable.*

To conclude this section, we should explicitly compare the two specifications of late bisimulation in Definition 9 and in Figure 3, and the two specifications of late bisimulation in Definition 11 and in Figure 4. Notice that those specifications that rely on logic are written without the need for any explicit conditions on variable names or any need to mention distinctions explicitly. These various conditions are, of course, present in the detailed description of the proof theory of our logic, but it seems to be very desirable to push the details of variable names, substitutions, equalities, etc into logic, where they have elegant and standard solutions. We do not address the *early* bisimulation [27] in this section, but it is not difficult to see that the analysis on names in late bisimilarity carries over to early bisimilarity. In fact, the essential difference between late and early bisimilarity is that in the early bisimilarity the case analysis on names is carried out "one step ahead" of late bisimilarity. This amounts to enlarging the scope of quantified variable w (in the input prefix case). The specification of early bisimilarity is given in the appendix.

6 Automation of proof search

The above specifications for one-step transitions and for late and open bisimulation are not only declarative and natural, an implementation of proof search using them can provide effective and *symbolic* implementation of both one-step transitions and bisimulations. We outline here what is needed to implement the meta-logic presented above.

Unification. Proof search requires unification in a couple of places: one in the implementation of the *def*$\mathcal{L}$ inference rule and one to determine the appropriate terms necessary to instantiate the $\exists$ quantifier in the $\exists\mathcal{R}$ inference rules. In the specifications presented here, unification is always within the L_λ or *higher-order pattern* unification [22] problem. This style of unification, which can be described as first-order unification extended to allow for bound variables and their mobility within terms and proofs, is known to have efficient and practical unification algorithms that compute most general unifiers whenever unifiers exist [32]. The Teyjus implementation [30, 31] of λProlog provides an effective implementation of such unification, as does Isabelle [33] and Twelf [35].

Proof search for one-step transitions. Computing one-step transitions can be done entirely using a conventional, higher-order logic programming language, such as λProlog: since the definition $\mathbf{D}_\pi$ for one-step transitions is Horn, we can use Proposition 4 to show that for the purposes of computing one-step transitions, all occurrences of ∇ in $\mathbf{D}_\pi$ can be changed to $\forall$. The resulting definition is then a logic program for which λProlog provides an effective implementation. In particular, after loading that definition, we would simply ask the query $P \xrightarrow{A} P'$, where P is the encoding of a particular π-calculus expression and A and P' are (meta-level) free variables. Standard logic programming would then systematically bind these two variables to the actions and continuations that P can make. Similarly, if the query was, instead, $P \xrightarrow{A} P'$, logic programming search would systematically return all bound actions (here, A has type $n \to a$) and corresponding bound continuations (here, P' has type $n \to p$).

Proof search for open bisimulation. Proof search for bisimulation is not immediately implemented for bisimulation by, say, λProlog, since neither ∇ nor the case analysis of *def*$\mathcal{L}$ are implemented. None-the-less, the implementation of proof search for open bisimulation is easy to specify. The key steps in a direct implementation of open bisimulation are outlined as follows. (Sequents missing from this outline are trivial to address.) In the following, we use the quantifier prefix $\mathcal{Q}$ to denote either $\forall x$ or ∇x or the empty quantifier prefix.

1. When searching for a proof of $\Sigma\,; \vdash \sigma \triangleright \mathcal{Q}.\mathit{obisim}\ P\ Q$ apply right-introduction rules.
2. If the sequent has a formula on its left-hand sides, then that formula is $\sigma \triangleright P \xrightarrow{A} P'$, where P denotes a particular closed term and A and P' are terms, possibly containing eigen-variables. In this case, select the *def*$\mathcal{L}$ inference rule: the premises of this inference rule will then be either (*i*) the empty-set of premises (which represents the only way that proof search terminates), or (*ii*) a set of premises that are all again of the form of one-step judgments, or (*iii*) the premise contains $\top$ instead of an atom on the left, in which case, we must consider the remaining case that follows (after using the weakening $w\mathcal{L}$ inference rule).
3. If the sequent has the form $\Sigma\,; \vdash \sigma \triangleright \exists Q'[Q \xrightarrow{A} Q' \wedge B(P', Q')]$, where $B(P', Q')$ involves a recursive call to *obisim* and where P' is a closed term, then we must instantiate the existential quantifier with an appropriate substitution. Standard logic programming techniques (as described in step 1 above) can be used to find a substitution for Q' such that $Q \xrightarrow{A} Q'$ is provable (during this search, eigenvariables and locally scoped variables are treated as constants and P and A denote particular closed terms). There might be several ways to prove such a formula and, as a result, there might be several different substitutions for Q'. If one chooses the term T to instantiate Q', then one proceeds to prove the sequent $\Sigma\,; \vdash \sigma \triangleright \mathcal{Q}.\mathit{obisim}\ P'\ T$. If the sequent has the form $\Sigma\,; \vdash \sigma \triangleright \exists Q'[Q \xrightarrow{A} Q' \wedge B(P', Q')]$, one proceeds in the same manner.

Proof search for the first two cases is invertible (no backtracking is needed for those cases). On the other hand, the approach in the third case is not invertible, and backtracking on possibly all choices of substitution term T might be necessary to ensure completeness.

Proof search for late bisimulation. The main difference between doing proof search for open bisimulation and late bisimulation is that in the later, we need to instantiate the formula $\mathcal{E}x$ and explore the cases generated by the $\vee\mathcal{L}$ rule. First, consider a sequent of the form $\Sigma, x\,;\, \mathcal{E}x, \Gamma_x \vdash C_x$, where $\Gamma_x \cup \{C_x\}$ is a set of formulas which may have x free. One way to proceed with search for a proof would be to instantiate $\forall z(x = z \vee x \neq z)$ with, say, a and with b. Thus, we need to consider proofs of the sequent is $\Sigma, x\,;\, x = a \vee x \neq a, x = b \vee x \neq b, \Gamma_x \vdash C_x$. Using the $\vee\mathcal{L}$ rule twice, we are left with four sequents to prove:

1. $\Sigma, x\,;\, x = a, x = b, \Gamma_x \vdash C_x$ which is proved trivially since the equalities are contradictory;
2. $\Sigma, x\,;\, x = a, x \neq b, \Gamma_x \vdash C_x$, which is equivalent to $\Sigma\,;\, \Gamma_a \vdash C_a$;
3. $\Sigma, x\,;\, x \neq a, x = b, \Gamma_x \vdash C_x$, which is equivalent to $\Sigma\,;\, \Gamma_b \vdash C_b$; and
4. $\Sigma, x\,;\, x \neq a, x \neq b, \Gamma_x \vdash C_x$.

In this way, the excluded middle can be used with a set of n items to produce $n + 1$ sequents: one for each member of the set and one extra sequent to handle all other cases (if there are any).

The main issue for implementing proof search with this specification of late bisimulation is to determine at what instances we should make instances of the excluded middle: answering this question would then reduce proof search to one similar to open bisimulation. There seems to be two extreme approaches to take: at one extreme, we can take instances for all possible names that are present in our process expressions: determining such instances is simple but might lead to many more cases to consider than is necessary. Another approach would be more lazy in that we would suggest an instance of the excluded middle only when there seems to be a need to consider that instance. The failure of a $defR$ rule because of a mismatch (at the meta-level) between an eigenvariable and a constant would, for example, suggest that excluded middle should be invoked for that eigenvariable and that constant. The exact details of such schemes is left for future work.

7 Related and future work

There are many papers on topics related to the encoding of the operational semantics of the π-calculus into formal systems. Honsell, Miculan, and Scagnetto [18], for example, encode the π-calculus in Coq and assume that there are an infinite number of global names. They then build formal mechanisms to support notions such as "freshness" within a scope, substitution of names, occurrences of names in expressions, etc. Gabbay [9] does something similar but uses a certain kind of set theory [10] to help develop his formal mechanisms. Hirschkoff [15] also used Coq but employed deBruijn numbers [5] instead of explicit names. In [6], Despeyroux encodes names as parameter in Coq, with additional axioms formalizing, among others, the decidability of equality of names. In all of these projects, formalizing names and their scopes, occurrences, freshness, and substitution is considerable work. In our approach, much of this same work is required, of course, but it is available in rather old technology, particularly, via Church's Simple Theory of Types (where bindings and terms and formulas was put on a firm foundation via λ-terms), Gentzen's sequent calculus and central cut-elimination theorem, Huet's unification procedure for λ-terms, etc. More modern work on proof search in higher-order logics is also available to make our task easier and more declarative.

The encoding of late transition of π-calculus using HOAS and its related advantages have been studied in a number of previous works [17, 25, 6, 18, 36, 26]. Our encoding, presented as a definition

in Figure 2, has appeared in [26]. The material on proof automation in Section 6 clearly seems related to work on *symbolic bisimulation* (for example, see [3, 14]) and work on using unification and logic programming techniques to compute symbolic bisimulations (see, for example, [1, 2]). Since the technologies used to describe these other approaches is rather different than what is described here, a detailed comparison is left for future work.

It is, of course, interesting to consider the general π-calculus where infinite behaviors are allowed (by including ! or recursive definitions). In such cases, one might be able to still do many proofs involving bisimulation if the proof system included induction and co-induction inference rules. Inference rules for induction and co-induction appropriate for the sequent calculus has been presented in [29] and a version of these rules that also involves the ∇ quantifier has been presented in the first author's PhD [41]. The encoding of π-calculus involving ! and a formalization of late bisimulation have been presented in [41]. Open bisimulation, however, has not been studied in this setting. We plan to investigate further how these stronger proof systems can be used establish properties about π-calculus expressions with infinite behaviors.

Specifications of operational semantics using a meta-logic should make it possible to formally prove properties concerning that operational semantics. This was the case, for example, with specifications of the evaluation and typing of simple functional and imperative programming languages: a number of common theorems (determinacy of evaluation, subject-reduction, etc) can be naturally inferred using meta-logical specifications [21]. We plan to investigate using our meta-logic (also incorporating rules for induction and co-induction) for formally proving parts of the theory of the π-calculus. It seems, for example, rather transparent to prove that open bisimilarity is a congruence in our setting.

Acknowledgements. We are grateful to the reviewers of an earlier draft of this paper for their detailed and useful comments.

References

1. S. Basu, M. Mukund, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. M. Verma. Local and symbolic bisimulation using tabled constraint logic programming. In *International Conference on Logic Programming (ICLP)*, volume 2237 of *Lecture Notes in Computer Science*, pages 166–180, Paphos, Cyprus, November 2001. Springer.
2. M. Boreale. Symbolic trace analysis of cryptographic protocols. In *Proceedings of ICALP 2001*, volume 2076 of *LNCS*, pages 667 – 681. Springer-Verlag, 2001.
3. M. Boreale and R. D. Nicola. A symbolic semantics for the π-calculus. *Information and Computation*, 126(1):34–52, April 1996.
4. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
5. N. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indag. Math.*, 34(5):381–392, 1972.
6. J. Despeyroux. A higher-order specification of the π-calculus. In *Proc. of the IFIP International Conference on Theoretical Computer Science, IFIP TCS'2000, Sendai, Japan, August 17-19, 2000.*, Aug. 2000.
7. L.-H. Eriksson. A finitary version of the calculus of partial inductive definitions. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions to Logic Programming*, volume 596 of *Lecture Notes in Artificial Intelligence*, pages 89–134. Springer-Verlag, 1991.
8. M. P. Fiore, G. D. Plotkin, and D. Turi. Abstract syntax and variable binding. In *14th Annual Symposium on Logic in Computer Science*, pages 193–202. IEEE Computer Society Press, 1999.
9. M. J. Gabbay. The π-calculus in FM. In F. Kamareddine, editor, *Thirty-five years of Automath*, volume IV, pages 80–149. Kluwer, 2003.
10. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2001.
11. G. Gentzen. Investigations into logical deductions. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland Publishing Co., Amsterdam, 1969.
12. J.-Y. Girard. A fixpoint theorem in linear logic. Email to the linear@cs.stanford.edu mailing list, February 1992.

13. L. Hallnäs and P. Schroeder-Heister. A proof-theoretic approach to logic programming. ii. Programs as definitions. *Journal of Logic and Computation*, 1(5):635–660, October 1991.
14. M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138(2):353–389, Feb. 1995.
15. D. Hirschkoff. A full formalization of pi-calculus theory in the Calculus of Constructions. In E. Gunter and A. Felty, editors, *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97)*, number 1275 in LNCS, pages 153–169, Murray Hill, New Jersey, Aug. 1997.
16. M. Hofmann. Semantical analysis of higher-order abstract syntax. In *14th Annual Symposium on Logic in Computer Science*, pages 204–213. IEEE Computer Society Press, 1999.
17. F. Honsell, M. Lenisa, U. Montanari, and M. Pistore. Final semantics for the π-calculus. In *Proc. of PROCOMET'98*, 1998.
18. F. Honsell, M. Miculan, and I. Scagnetto. Pi-calculus in (co)inductive type theory. *Theoretical Computer Science*, 253(2):239–285, 2001.
19. G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.
20. R. McDowell and D. Miller. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science*, 232:91–119, 2000.
21. R. McDowell and D. Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Transactions on Computational Logic*, 3(1):80–136, January 2002.
22. D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.
23. D. Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, pages 321–358, 1992.
24. D. Miller. Abstract syntax for variable binders: An overview. In J. Lloyd and et. al., editors, *Computational Logic - CL 2000*, number 1861 in LNAI, pages 239–253. Springer, 2000.
25. D. Miller and C. Palamidessi. Foundational aspects of syntax. In P. Degano, R. Gorrieri, A. Marchetti-Spaccamela, and P. Wegner, editors, *ACM Computing Surveys Symposium on Theoretical Computer Science: A Perspective*, volume 31. ACM, Sep 1999.
26. D. Miller and A. Tiu. A proof theory for generic judgments: An extended abstract. In P. Kolaitis, editor, *Proceedings of LICS 2003*, pages 118 – 127, July 2003.
27. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Part II. *Information and Computation*, pages 41–77, 1992.
28. J. C. Mitchell and E. Moggi. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic*, 51, 1991.
29. A. Momigliano and A. Tiu. Induction and co-induction in sequent calculus. In *Post-proceedings of TYPES 2003 Workshop*, pages 293 – 308. Springer-Verlag LNCS, Vol.3085, 2003.
30. G. Nadathur. A treatment of higher-order features in logic programming. *Theory and Practice of Logic Programming*, To appear.
31. G. Nadathur and D. J. Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of Lambda Prolog. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, pages 287–291, Trento, Italy, July 1999. Springer-Verlag LNCS.
32. T. Nipkow. Functional unification of higher-order patterns. In M. Vardi, editor, *LICS93*, pages 64–74. IEEE, June 1993.
33. L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.
34. F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.
35. F. Pfenning and C. Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.
36. C. Röckl, D. Hirschkoff, and S. Berghofer. Higher-order abstract syntax with induction in Isabelle/HOL: Formalizing the pi-calculus and mechanizing the theory of contexts. In F. Honsell and M. Miculan, editors, *Proc. FOSSACS'01*, volume 2030 of *Lecture Notes in Computer Science*, pages 364–378. Springer, 2001.
37. D. Sangiorgi. A theory of bisimulation for the π-calculus. *Acta Informatica*, 33(1):69–97, 1996.
38. D. Sangiorgi and D. Walker. *π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
39. P. Schroeder-Heister. Rules of definitional reflection. In M. Vardi, editor, *Eighth Annual Symposium on Logic in Computer Science*, pages 222–232. IEEE Computer Society Press, IEEE, June 1993.
40. R. F. Stärk. Cut-property and negation as failure. *International Journal of Foundations of Computer Science*, 5(2):129–164, 1994.
41. A. Tiu. *A Logical Framework for Reasoning about Logical Specifications*. PhD thesis, Pennsylvania State University, May 2004.

Appendix A. Inference rules of $FO\lambda^{\Delta\nabla}$

The "core" inference rules for $FO\lambda^{\Delta\nabla}$ are given in Figure 5. Except for the rules for ∇, these are essentially the inference rules for intuitionistic logic given by Gentzen [11]. The only missing inference rules are those for introducing defined formulas: these are given in Section 2.

$$\frac{}{\Sigma\,;\,\sigma \triangleright B, \Gamma \vdash \sigma \triangleright B}\ init \qquad \frac{\Sigma\,;\,\Delta \vdash \mathcal{B} \qquad \Sigma\,;\,\mathcal{B}, \Gamma \vdash \mathcal{C}}{\Sigma\,;\,\Delta, \Gamma \vdash \mathcal{C}}\ cut$$

$$\frac{\Sigma\,;\,\sigma \triangleright B, \sigma \triangleright C, \Gamma \vdash \mathcal{D}}{\Sigma\,;\,\sigma \triangleright B \wedge C, \Gamma \vdash \mathcal{D}}\ \wedge\mathcal{L} \qquad \frac{\Sigma\,;\,\Gamma \vdash \sigma \triangleright B \qquad \Sigma\,;\,\Gamma \vdash \sigma \triangleright C}{\Sigma\,;\,\Gamma \vdash \sigma \triangleright B \wedge C}\ \wedge\mathcal{R}$$

$$\frac{\Sigma\,;\,\sigma \triangleright B, \Gamma \vdash \mathcal{D} \qquad \Sigma\,;\,\sigma \triangleright C, \Gamma \vdash \mathcal{D}}{\Sigma\,;\,\sigma \triangleright B \vee C, \Gamma \vdash \mathcal{D}}\ \vee\mathcal{L} \qquad \frac{\Sigma\,;\,\Gamma \vdash \sigma \triangleright B}{\Sigma\,;\,\Gamma \vdash \sigma \triangleright B \vee C}\ \vee\mathcal{R} \qquad \frac{\Sigma\,;\,\Gamma \vdash \sigma \triangleright C}{\Sigma\,;\,\Gamma \vdash \sigma \triangleright B \vee C}\ \vee\mathcal{R}$$

$$\frac{\Sigma\,;\,\Gamma \vdash \sigma \triangleright B \qquad \Sigma\,;\,\sigma \triangleright C, \Gamma \vdash \mathcal{D}}{\Sigma\,;\,\sigma \triangleright B \supset C, \Gamma \vdash \mathcal{D}}\ \supset\mathcal{L} \qquad \frac{\Sigma\,;\,\sigma \triangleright B, \Gamma \vdash \sigma \triangleright C}{\Sigma\,;\,\Gamma \vdash \sigma \triangleright B \supset C}\ \supset\mathcal{R}$$

$$\frac{\Sigma, \sigma \vdash t : \gamma \qquad \Sigma\,;\,\sigma \triangleright B[t/x], \Gamma \vdash \mathcal{C}}{\Sigma\,;\,\sigma \triangleright \forall_\gamma x.B, \Gamma \vdash \mathcal{C}}\ \forall\mathcal{L} \qquad \frac{\Sigma, h\,;\,\Gamma \vdash \sigma \triangleright B[(h\ \sigma)/x]}{\Sigma\,;\,\Gamma \vdash \sigma \triangleright \forall x.B}\ \forall\mathcal{R}$$

$$\frac{\Sigma, h\,;\,\sigma \triangleright B[(h\ \sigma)/x], \Gamma \vdash \mathcal{C}}{\Sigma\,;\,\sigma \triangleright \exists x.B, \Gamma \vdash \mathcal{C}}\ \exists\mathcal{L} \qquad \frac{\Sigma, \sigma \vdash t : \gamma \qquad \Sigma\,;\,\Gamma \vdash \sigma \triangleright B[t/x]}{\Sigma\,;\,\Gamma \vdash \sigma \triangleright \exists_\gamma x.B}\ \exists\mathcal{R}$$

$$\frac{\Sigma\,;\,(\sigma, y) \triangleright B[y/x], \Gamma \vdash \mathcal{C}}{\Sigma\,;\,\sigma \triangleright \nabla x\ B, \Gamma \vdash \mathcal{C}}\ \nabla\mathcal{L}(*) \qquad \frac{\Sigma\,;\,\Gamma \vdash (\sigma, y) \triangleright B[y/x]}{\Sigma\,;\,\Gamma \vdash \sigma \triangleright \nabla x\ B}\ \nabla\mathcal{R}(*)$$

$$\frac{}{\Sigma\,;\,\sigma \triangleright \bot, \Gamma \vdash \mathcal{B}}\ \bot\mathcal{L} \qquad \frac{}{\Sigma\,;\,\Gamma \vdash \sigma \triangleright \top}\ \top\mathcal{R} \qquad \frac{\Sigma\,;\,\mathcal{B}, \mathcal{B}, \Gamma \vdash \mathcal{C}}{\Sigma\,;\,\mathcal{B}, \Gamma \vdash \mathcal{C}}\ c\mathcal{L} \qquad \frac{\Sigma\,;\,\Gamma \vdash \mathcal{C}}{\Sigma\,;\,\mathcal{B}, \Gamma \vdash \mathcal{C}}\ w\mathcal{L}$$

(*) provided that y is not free in $\lambda x\ B$.

Fig. 5. The core rules of $FO\lambda^{\Delta\nabla}$.

Appendix B. Specification of strong early bisimilarity

We take the notion of strong early bisimilarity as it is defined in [27] for late transition system. The corresponding specification is given in Figure 6. As noted in [27], the essential difference between late and early bisimulation is in the scope of (universal) quantification of names in the case of bound input transitions. This is reflected in the specification in Figure 6, where not only the scope of quantification is enlarged, but also the scope of the excluded middle axiom on names.

$$\begin{aligned}
ebisim\ P\ Q \triangleq\ & \forall A \forall P'\,[(P \xrightarrow{A} P') \supset \exists Q'.(Q \xrightarrow{A} Q') \wedge ebisim\ P'\ Q'] \wedge \\
& \forall A \forall Q'\,[(Q \xrightarrow{A} Q') \supset \exists P'.(P \xrightarrow{A} P') \wedge ebisim\ Q'\ P'] \wedge \\
& \forall X \forall P'\,[(P \xrightarrow{\downarrow X} P') \supset \forall w.\mathcal{E}w \supset \exists Q'.(Q \xrightarrow{\downarrow X} Q') \wedge ebisim\ (P'w)\ (Q'w)] \wedge \\
& \forall X \forall Q'\,[(Q \xrightarrow{\downarrow X} Q') \supset \forall w.\mathcal{E}w \supset \exists P'.(P \xrightarrow{\downarrow X} P') \wedge ebisim\ (Q'w)\ (P'w)] \wedge \\
& \forall X \forall P'\,[(P \xrightarrow{\uparrow X} P') \supset \exists Q'.(Q \xrightarrow{\uparrow X} Q') \wedge \nabla w.ebisim\ (P'w)\ (Q'w)] \wedge \\
& \forall X \forall Q'\,[(Q \xrightarrow{\uparrow X} Q') \supset \exists P'.(P \xrightarrow{\uparrow X} P') \wedge \nabla w.ebisim\ (Q'w)\ (P'w)]
\end{aligned}$$

where $\mathcal{E} = \lambda x\ \forall y(x = y) \vee (x = y \supset \bot)$.

Fig. 6. Specification of strong early bisimulation.