Mixing Finite Success and Finite Failure in an Automated Prover

Alwen Tiu[1], Gopalan Nadathur[2], and Dale Miller[3]

[1] INRIA Lorraine
[2] University of Minnesota
[3] INRIA Futurs/École polytechnique

Abstract. The operational semantics and typing of modern programming and specification languages are often defined using relations and proof systems. In simple settings, logic programming can be used to provide rather direct and natural interpreters for such operational semantics. More complex features of specifications such as names and their bindings, proof rules with negative premise, and the exhaustive enumeration of state spaces, all pose significant challenges to conventional logic programming systems. In this paper, we describe a simple architecture for the implementation of deduction systems that allows a specification to interleave both with finite success and finite failure. The implementation techniques for this prover are largely common ones from logic programming, i.e., logic variables, (higher-order pattern) unification, backtracking (using stream-based computation), and abstract syntax based on simply typed λ-terms. We present a particular instance of this prover architecture and its prototype implementation, based on a dual interpretation of (finite) success and failure in proof search. We discuss important differences between this prover and traditional logic programming and present an implementation of bisimulation checking for π-calculus, which cannot be so directly and declaratively done in traditional logic programming languages.

1 Introduction

The operational semantics and typing of modern programming and specification languages are often defined using relations and proof systems, e.g., in the sytle of Plotkin's structural operational semantics. In simple settings, higher-order logic programming such as λProlog or Twelf can be used to provide rather direct and natural interpreters for such operational semantics. However, the use of logic programming in this setting is rarely beyond providing interpreters. In particular, reasoning about the language specifications has to be done outside the logic programming system. For instance, in checking bisimulation in process calculi, one needs to analyze all the transition paths a process can potentially go through. To add to the complication, modern language specifications often make use of complex features such as variable bindings and the notion of *names* (as in process calculi [12]), which interferes in a non-trivial way with case analyses. These case analyses cannot be done directly inside logic programming system, not in a purely logical way at least, even though they are simply enumerations of answer substitutions. In this paper, we describe an extension to logic programming with logically sound features which allow us to do some modest automated reasoning about specifications of operational semantics. This extension is more conceptual than technical, that is, the implementation of the extended logic programming language uses only basic implementation techniques that are largely common ones in logic programming, i.e., logic variables, higher-order pattern unification, backtracking (using stream-based computation) and abstract syntax based on typed λ-calculus.

The implementation described in this paper is based on the logic $FO\lambda^{\Delta\nabla}$ [11]. $FO\lambda^{\Delta\nabla}$ is an extension of the first-order variant of Church's Simple Theory of Types with fixed points and a new quantifier ∇. In $FO\lambda^{\Delta\nabla}$ quantifiers can range over higher-types, but quantification over propositions is not allowed, and hence it is essentially first-order logic with simply typed terms. The simply typed terms of the logic are used as an abstract syntax for encoding arbitrary syntax of object-languages specification. This style of encoding, called *λ-tree syntax*, is a variant of *higher-order abstract syntax* where the λ-abstraction is used only to encode variable binding. The quantifier ∇ is first introduced in [11] to help encode the notion of "generic judgment" that occurs commonly when reasoning with λ-tree syntax. The extension with fixed points is done through a proof theoretical notion of *definitions* [19, 1, 3, 21, 7].

In a logic with definitions, an atomic proposition may be defined by another formula (which may contain the atomic proposition itself). Proof search for a defined atomic formula is done by unfolding the definition of the formula. A provable formula like $\forall X.pX \supset qX$, where p and q are some defined predicates, expresses the fact that for every term t and for every proof (computation) of pt, there is a proof (computation) of qt. If the computation tree associated with p is finite, we can effectively enumerate all its computation paths and check the provability of qt for each path. Note that if the computation tree for p is empty (pt is not provable for any t) then $\forall X.pX \supset qX$ is vacuously true. In other words, *failure* in proof search for pX entails *success* in proof search for $pX \supset qX$. The analogy with *negation-as-failure* in logic programming is obvious: if we take qX to be $\bot$ (false), then provability of $pX \supset \bot$ corresponds to success in proof search for $not(pX)$ in logic programming. This relation between negation-as-failure in logic programming and negation in logic with definitions has been observed in [4, 3]. In the implementation of $FO\lambda^{\Delta\nabla}$, the above observation leads to a neutral view on proof search: If proof search for a goal A returns a non-empty set of answer substitutions, then we have found a proof of A. On the other hand, if proof search for A returns an empty answer set, then we have found a proof for $\neg A$. Answer substitutions can thus be interpreted in a dual way depending on the context of proof search (see Section 3).

The rest of the paper is organized as follows. In Section 2, we give an overview of the logic $FO\lambda^{\Delta\nabla}$. Section 3 describes an implementation of a fragment of $FO\lambda^{\Delta\nabla}$, the Level-0/1 prover, which is based on a dual interpretation of failure/success in proof search. Section 4 discusses the treatment of variables in the Level-1/0 prover, in particular it discusses the issues concerning the interaction between *eigenvariables* and logic variables. Section 7 gives an example of an implementation of π-calculus and bisimulation checker in Level-0/1 prover. This example illustrates the use of the ∇-quantifier in capturing the notion of names in π-calculus. The example is based on a recent work on encoding π-calculus in $FO\lambda^{\Delta\nabla}$ [24]. Section 9 discusses the components of proof search implementation and outlines a general implementation architecture for $FO\lambda^{\Delta\nabla}$. Section 10 discusses future work. An extended version of this paper is avalaible on the web, containing more examples and more detailed comparison with logic programming.

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

The logic $FO\lambda^{\Delta\nabla}$ [11] (pronounced "fold-nabla") is presented using a sequent calculus that is an extension of Gentzen's system LJ [2] for first-order intuitionistic logic. A *sequent* is an expression of the form $B_1, \ldots, B_n \longrightarrow B_0$ where $B_0, \ldots, B_n$ are formulas and the elongated turnstile $\longrightarrow$ is the sequent arrow. 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.* Eigenvariable can be used to encode the dynamics of abstraction in the operational semantics of various languages. However, for reasoning about certain uses of abstraction, notably the notion of *name restriction* in π-calculus, eigenvariables do not capture faithfully the intended meaning of such abstractions. To address this problem, in the logic $FO\lambda^{\Delta\nabla}$ sequents are extended with a new notion of "local scope" for proof-level bound variables (see [11] for motivations and examples). In particular, sequents in $FO\lambda^{\Delta\nabla}$ are of the form

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

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.

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

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

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

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

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 introduction rules for propositional connectives are straightforward generalization of LJ, that is, local signatures are distributed on the subformulas of the main formula (reading the rules bottom-up). The complete set of rules for $FO\lambda^{\Delta\nabla}$ is given in the Appendix. Note that since we do not allow quantification over predicates, this logic is proof-theoretically similar to first-order logic.

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 [9] 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 $\nabla x\ B$.

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 [19], Eriksson [1], Girard [3], Stärk [21] and McDowell and Miller [7]. 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 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.*

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.

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

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

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) [5]. We denote by $CSU(\mathcal{A}, \mathcal{H})$ the complete set of unifiers for the pair $(\mathcal{A}, \mathcal{H})$, that is, for any unifier θ of $\mathcal{A}$ and $\mathcal{H}$, there is a unifier $\rho \in CSU(\mathcal{A}, \mathcal{H})$ such that $\theta = \rho \circ \theta'$ for some substitution θ'. Since we allow higher-order terms in definitions, in certain cases there are no finite CSU's for the corresponding unification problems. However, in all the applications of $def\mathcal{L}$ in this paper, the terms involved in the unification are those of *higher-order pattern* [8, 16], that is, terms in which variables are applied only to distinct bound variables. Since higher-order pattern unification is decidable and has the most general unifier if the unification is solvable, the set $CSU(\mathcal{A}, \mathcal{H})$ in this case is either empty or contains a single substitution which is 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 $def\mathcal{R}$ 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 $def\mathcal{R}$ rule captures backchaining and the *def$\mathcal{L}$* rule corresponds to *case analysis* on all possible ways an atomic judgment could be proved. The latter is a distinguishing feature between the implementation of $FO\lambda^{\Delta\nabla}$ discussed in Section 3 and logic programming. For instance, given a definition $\{pa \stackrel{\triangle}{=} \top, \quad pb \stackrel{\triangle}{=} \top, \quad qa \stackrel{\triangle}{=} \top, \quad qb \stackrel{\triangle}{=} \top, \quad qc \stackrel{\triangle}{=} \top\}$ one can prove $\forall x.px \supset qx$: for all successful "computation" of p, there is a successful computation for q. Notice that by encoding logic programs as definitions, one can effectively encode *negation-as-failure* in logic programming using *def$\mathcal{L}$* [4], e.g., for the above program (definition), the goal $not(pc)$ in logic programming is encoded as the formula $pc \supset \bot$.

Among the properties of $FO\lambda^{\Delta\nabla}$, one that is particularly useful for proof search is the *invertibility* of certain rules. In proof search, invertible rules can always be applied without the need for backtracking. Those rules include *def$\mathcal{L}$*, $\nabla\mathcal{L}$, $\nabla\mathcal{R}$, $\exists\mathcal{L}$, $\forall\mathcal{R}$, the right introduction rules for $\wedge$ and $\supset$, and the left introduction rules for $\wedge$ and $\vee$ (see [22] for a proof). The invertibility of these rules motivates the choice of the fragment of $FO\lambda^{\Delta\nabla}$ and the proof search strategy in the prototype implementation discussed in Section 3.

3 Mixing success and failure in a prover

We now give an overview of an implementation of proof search for a fragment of $FO\lambda^{\Delta\nabla}$. This implementation, called *Level 0/1 prover*, is based on the dual interpretation of finite success and *finite failure* in proof search. In particular, the finite failure in proving a goal $\exists x.G$ should give us a proof of $\neg(\exists x.G)$ and vice versa. We experiment with a simple class of formulae which exhibits this duality. Consider the fragment of $FO\lambda^{\Delta\nabla}$ induced by the following classes of formulae:

Level 0: $G := \top \mid \bot \mid A \mid G \wedge G \mid G \vee G \mid \exists x.G \mid \nabla x.G$
Level 1: $D := \top \mid \bot \mid A \mid D \wedge D \mid D \vee D \mid \exists x.D \mid \nabla x.D \mid \forall x.D \mid G \supset D$
atomic: $A := p\, t_1 \dots t_n$

We shall assume that each predicate symbol belongs either to level-0 or level-1 and that level-0 formula can contain only level-0 predicates. Each definition clause $p\bar{x} \stackrel{\triangle}{=} B$ must be *stratified*, i.e., if p is a level-0 predicate then B should belong to the class level-0, otherwise if p is a level-1 predicate then B can be level-0 or level-1 formula. In the current implementation, stratification checking and typechecking are not implemented, so that we can experiment with a wider range of definitions than those for which the meta-theory is fully developed.

Notice that in the Level-1 formula, the use of implication is restricted to the form $G \supset D$ where G is a Level-0 formula. Therefore, nested implication like $(A \supset B) \supset C$ is not allowed. The Level-0/1 prover actually consists of two separate subprovers, one for each class of formulas. Implementation of proof search for level-0 formula is the standard logic-programming implementation. It is actually a subset of λProlog (substituting $\forall$ for ∇). That is, existentially quantified variables are replaced by logic variables, ∇-quantified variables are replaced with scoped (local) constants (which have to be distinguished from eigenvariables) and *defR* becomes backchaining. For level-1 formulas, the non-standard case is when the goal is an implication, e.g., $G \supset D$. Proof search strategy for this case derives from the following observation: the left-introduction rules for level-0 formulas are all invertible rules, and hence can always be applied first. Proof search for an implicational goal $G \supset D$ therefore proceeds as follows:

Step 1 Run the level-0 prover with the goal G, treating eigenvariables as logic variables.

Step 2 If Step 1 fails, then proof search for $G \supset D$ succeeds. Otherwise, collect all answer substitutions produced in Step 1, and for each answer susbtitution θ, proceed with proving $D\theta$

In Step 1, in the current implementation, we impose a restriction: the formula G must not contain any occurrences of logic variables. If this restriction is violated, an exception is returned and proof search is aborted. We shall return to this technical restriction in Section 4. For several examples that have been studied so far (π-calculus, bisimulation, modal logics) this restriction on the occurrence of logic variable does not seem to pose any problem for the set of goals of interest, e.g., checking bisimulation and satisfiability of modal logic formulas.

Concrete syntax The concrete syntax for Level 0/1 prover follows the syntax of λProlog. The concrete syntax for logical connectives are as follows:

$\top$	`true`	$\bot$	`false`
$\wedge$	`&` (ampersand) or `,` (comma)	$\vee$	`;` (semi-colon)
$\forall$	`pi`	$\exists$	`sigma`
∇	`nabla`	$\supset$	`=>`

The λ-abstraction is represented in the concrete syntax using a backslash, e.g., $\lambda x \lambda f.fx$ is written as `x\f\(f x)`. The order of precedence for the connectives are as follows (in decreasing order): $\wedge$, $\vee$, $\supset$, $\{\forall, \exists, \nabla\}$. Non-logical constants such as 'not' (negation-as-failure) and '!' (Prolog cut) are not implemented. But we allow a non-logical constant "print" which prints a string or term.

The symbol $\stackrel{\triangle}{=}$ separating the head and the body of a definition clause is written ':=' in the concrete syntax. For example, the familiar 'append' predicate for lists can be represented as the following definition.

```
append nil L L.
append (cons X L1) L2 (cons X L3) := append L1 L2 L3.
```

As in λProlog, we use '.' (dot) to indicate the end of a statement. Identifiers starting with a capital letter denote variables and those starting with lower-case letter denote constants. Variables in a definition clause are implicitly quantified outside the clause (the scope of such quantification is over the clause, so there is no accidental mixing of variables across different clauses). A definition clause with the body 'true' is abbreviated with the 'true' removed, e.g., the first clause of append above is actually an abbreviation of `append nil L L := true.`

4 Eigenvariables, logic variables and ∇

The three quantifiers, $\forall$, $\exists$ and ∇, give rise to three kinds of variables during proof search: eigenvariables, logic variables and "variables" generated by ∇. Their characteristics are as follows: logic variables are genuine variables, in that they can be instantiated during proof search. Eigenvariables are subject to instantiation only in proving negative goals, while in positive goals they are treated as scoped constants. Variables generated by ∇ are always treated as constants. In the implementation, ∇-variables are actually represented as λ-abstraction. Eigenvariables and logic variables share similar data structures, and explicit raising is used to encode their dependency on ∇-variables. The interaction between eigenvariables and logic variables is more subtle. Consider the case where both eigenvariables and logic variables are present in a negative goal, e.g.,

$$\forall x.\exists y.(px \wedge py \wedge x = y \supset \bot),$$

where p is defined as $\{pa \stackrel{\triangle}{=} \top, pb \stackrel{\triangle}{=} \top, pc \stackrel{\triangle}{=} \top\}$. In proof search for this formula, we are asked to produce for each x, a y such that x and y are distinct. This is no longer a unification problem in the usual sense, since we seek to cause a failure in unification, instead of success. This type of problem is generally referred to as *complement problems* or *disunification* [6], and its solution is not unique in general, even for the first-order case, e.g., in the above disunification problem, if x is instantiated to a then y can be instantiated with either b or c. In the higher-order case [14] the problem is considerably more difficult, and hence in the current implementation we disallow the occurrences of logic variables in negative goals.

In Figure 2, we show a sample session in Level 0/1 prover which highlights the differences between eigenvariables, logic variables, and ∇-variables. The unification problem in the first two goals can be seen as the unification problem $\lambda x.x = \lambda x.(Mx)$. Notice that there is no difference between $\forall$ and ∇ if the goal is Horn (i.e., there is no implication in the goal). A non-Horn goal is given in the third example. Here the unification fails (hence the goal succeeds) because x is in the scope of M. It is similar to the unification problem $\lambda x.x = \lambda x.M$. Here substitution must be capture-avoiding, therefore M cannot be instantiated with x. However, if we switch the order of quantifer or using application-term (as in (fx) in the fourth goal) the unification succeeds. In the last goal, we are trying to prove implicational goal with logic variables, and the system returns an exception.

Enumerating solutions. The ability to instantiate eigenvariables in the definition-left rule can be used to enumerate all possible solutions to a given level-0 goal. More interestingly, such solutions can actually be explicitly queried. The predicate `enum` in Figure 3 enumerates all solutions to the query pX and stores the resulting answers in a list, e.g.,

```
?- enum L.
Yes
L = (cons c (cons b (cons a nil)))
```

```
?- nabla x\ x = (M x).
Yes
M = x1\x1
Find another? [y/n] y
No.
?- pi x\ x = (M x).
Yes
M = x1\x1
Find another? [y/n] y
No.
?- pi M\ nabla x\ x = M => false.
Yes
Find another? [y/n] y
No.
?- pi f\ nabla x\ x = f x => print "unification succeeded".
unification succeeded
Yes
?- nabla x\ pi y\ x = y => print "unification succeeded".
unification succeeded
Yes
?-  nabla x\ x = (M x) => false.
Error: non-pure term found in implicational goal.
```

Fig. 2. A session in Level 0/1 prover.

There are other solutions which are permutations of the the above list. In general, if there are n answer substitutions to a given goal, `enum` will produce $n!$ answers. Therefore the complexity of the `enum` clause is rather high, and in practice, such enumerations should be used with care.

5 Comparison with λProlog

Setting aside the ∇ quantifier, one might think that the proof search behavior for $\forall$ and $\supset$ connectives in $FO\lambda^{\Delta\nabla}$ can be approximated in λProlog with negation-as-failure. The $\supset$ connective, for instance, can be defined in λProlog as follows

```
imp A B :- not(A, not(B)).
```

If proof search for `A` terminates with failure, then the goal `imp A B` succeeds. Otherwise, for each answer substitution for `A`, if `B` fails then the whole goal fail, otherwise the `not(B)` fails and hence `imp A B` succeeds. For ground terms (with no eigenvariables) `A` and `B`, this co-incides with the operational reading of `A => B` in Level 0/1 prover. The story is not so simple, however, if there are occurrences of eigenvariables in `A` or `B`.

One can sort of see intuitively why the inclusion of eigenvariables in `A` or `B` would cause problem: the eigenvariables in λProlog play a single role as scoped constant, while in Level 0/1 they have dual roles, as constants and as variables

```
p a.
p b.
p c.
member X (cons X L).
member X (cons Y L) := member X L.
acc L R := p Y, (member Y L => false), acc (cons Y L) R.
acc L L := pi y\ p y => member y L.
enum L := acc nil L.
```

Fig. 3. Enumerating solutions.

to be instantiated. However, there is one trick to deal with this, that is, suppose we are to prove $\forall x.Ax \supset Bx$, instead of the straightforward encoding of $\forall$ as 'pi', we may use 'sigma' instead:

```
sigma x\ not (A x, not (B x)).
```

Here the execution of the goal forces the instantiation of the (supposed to be) 'eigenvariable'. The real problem appears when eigenvariables may assume two roles at the same time. Consider the goal

$$\forall x \forall y. x = a \supset y = b$$

where a and b are constants. Assuming nothing about the domain of quantification, this goal is not provable. Now, the possible encodings into λProlog is to use either 'sigma' or 'pi' to encode the quantifier. Using the former, we get

```
sigma x\ sigma y\ not (x = a, not(y = b)).
```

This goal is provable, hence it is not the right encoding. If instead we use 'pi' to encode $\forall$, we get

```
pi x\ pi y\ not (x = a, not (y = b)).
```

This goal also succeeds, since 'x' here will become an eigenvariable and hence it is not unifiable with 'a'. Of course, one cannot rule out other more complicated encodings, e.g., treating $\forall$ as 'pi' in one place and as 'sigma' in others, but it is doubtful that there will be an encoding scheme which can be generalized to arbitrary cases.

6 Example: abstract transition systems

An *abstract transition system* (ats for short) $\mathcal{T}$ is usually defined as the triple

$$(\Lambda, S, \delta)$$

where Λ is a set of *labels* or *actions*, S is a set of *states* and δ is the transition relation, i.e., $S \subseteq S \times \Lambda \times S$. The elements of δ are usually represented graphically as something like $p \xrightarrow{a} q$, where p and q are states and a is an action. Unlabelled transition system can be seen as an ats with a single label. In this section we consider only finite state transition systems, that is, in the above definition, the set S is finite. An (finite) ats can be trivially encoded as follows: each label and state in the ats are encoded as unique constants and the transition relation δ is encoded as a predicate (called 'next' in this example) which takes three arguments, a state, a label and another state. For example, the ats

$$(\{a,b\},\{p,q,r\},\{p \xrightarrow{a} q, p \xrightarrow{b} r, q \xrightarrow{b} p\})$$

is encoded as the following definition:

```
next p a q.
next p b r.
next q b p.
```

Reachability analysis Among the properties of interest about an ats is the reachability of certain states. For example, in modelling an online transaction system, one would want to make sure that no two processes are in the same critical section at the same time. For finite states systems, this can be translated to the problem of unreachability of the states that correspond to this situation.

Let us consider the following unlabelled ats. We remove the label in specifying the transition relation.

```
next p q.
next p z.
next q r.
next r q.
next a b.
```

The reachability condition can be specified as the following definition

```
reach L P P.
reach L P Q :=
   (P = Q => false), sigma R\ next P R, (member R L => false),
   reach (cons P L) R Q.
```

The predicate `reach L P Q` should be read as "the state Q is reachable from P, given the already visited states in L". Reachability of `Q` from `P` is then specified as the formula `reach nil P Q`. We need the list `L` to keep track of visited states, since there can be cycles in the transition relations.

One would probably think that the unreachability predicate can be simply defined as the negation of reachability. However, notice that the reachability predicate defined above is a Level-1 definition, and hence its negation would fall outside the scope of the formulas the current prover can handle. Here we choose to define directly a notion of unreachability: a state Q is not reachable from P if P is different from Q and Q is not reachable from any successor of P. This is formalized as follows:

```
notreach L P Q :=
   (P = Q => false),
   pi R\ next P R => if (member R L) (true) (notreach (cons P L) R Q).
```

Below are some simple queries about the reachability of some states.

```
?- reach nil p q.
Yes

?- notreach nil p a.
Yes

?- notreach nil a b.
No
```

```
    int turn = 0;
    boolean[ ] active = {false, false};
    int i, j;

    void process(int k)
    {
       i = k;
       j = 1 - k;
1:     while (true) {
2:        active[i] = true;
          turn = j;
          while (
3:               active[j] &&
4:               turn == j)
              ;
           // critical section
5:         ...
           // end of critical section
           active[i] = false;
       }
    }
```

Fig. 4. Peterson's algorithm

Peterson's algorithm We shall now apply the reachability analysis to verify the *mutual exclusion* property of a certain concurrent system. More precisely, our system consists of two processes running in parallel, they share certain variables and updates to these variables constitute the critical section for each process. We shall look at a particular algorithm, the Peterson's algorithm, which solves this mutual exclusion problem, and verify that the algorithm is correct. The Peterson's algorithm is given in Figure 4. The system consists of 'process(0)' and 'process(1)' running in parallel. Let $p0$ be the process 'process(0)' and $p1$ be the process 'process(1)'. The variables 'turn' and 'active' are shared by these two processes. The numbers on the left denote the states of the program counters of

```
next (pr 1 A B C D) (pr 2 A 1 C D).
next (pr 2 A B C D) (pr 3 A B C 1).
next (pr 3 A B 0 D) (pr 5 A B 0 D).
next (pr 3 A B 1 D) (pr 4 A B 1 D).
next (pr 4 A B C 0) (pr 5 A B C 0).
next (pr 4 A B C 1) (pr 3 A B C 1).
next (pr 5 A B C D) (pr 1 A 0 C D).
next (pr A 1 B C D) (pr A 2 B 1 D).
next (pr A 2 B C D) (pr A 3 B C 0).
next (pr A 3 0 C D) (pr A 5 0 C D).
next (pr A 3 1 C D) (pr A 4 1 C D).
next (pr A 4 B C 1) (pr A 5 B C 1).
next (pr A 4 B C 0) (pr A 3 B C 0).
next (pr A 5 B C D) (pr A 1 B C D).
```

Fig. 5. A specification of Peterson's algorithm.

```
member X (cons X L).
member X (cons Y L) := member X L.
notreach L P Q :=
    (P = Q => false),
    pi R\ next P R =>
         if (member R L) then (true) (notreach (cons P L) R Q).

mutex := pi x\ pi y\ pi z\ notreach nil (pr 1 1 0 0 0) (pr 5 5 x y z).
```

Fig. 6. Verifying the correctness of Peterson's algorithm: a declarative approach.

```
member X (cons X L).
member X (cons Y L) := member X L.
acc P L R := next P X, (member X L => false), acc P (cons X L) R.
acc P L L := pi x\ next P x => member x L.
% get all reachable next states from P
getnext P L := acc P nil L.

union L nil L.
union L (cons A R) M :=
  if (member A L) (union L R M)
     (union L R M1, M = (cons A M1)).
takenew V nil nil.
takenew V (cons A R) L :=
  if (member A V) (takenew V R L)
     (takenew V R L1, L = (cons A L1)).

newfrontier V nil nil.
newfrontier V (cons P R) NF :=
  getnext P L,
  takenew V L L1,
  newfrontier V R NF1,
  union L1 NF1 NF.

notreach Visited Frontier Q :=
  (member Q Frontier => false),
   if (Frontier = nil) (true)
      (
        print ".",
        union Visited Frontier Visited1,
        newfrontier Visited1 Frontier Frontier1,
        notreach Visited1 Frontier1 Q
   ).
mutex := pi x\ pi y\ pi z\
   notreach nil (cons (pr 1 1 0 0 0) nil) (pr 5 5 x y z).
```

Fig. 7. Verifying the correctness of Peterson's algorithm: breadth-first search

both processes at certain points of the code. Let $pc0$ and $pc1$ be the program counters of $p0$ and $p1$, respectively. The (abstracted) state of the entire system can be represented as the the tuple

$$(pc0, pc1, \text{active}[0], \text{active}[1], \text{turn}).$$

There are a total of 200 states. We shall use '0' and '1' to represent 'false' and 'true', respectively. A state transition is triggered by process execution. For example, in the state $(1,1,0,0,0)$ the possible next states are $(2,1,1,0,0)$ (process $p0$ is executing and sets the value of 'active[0]' to '1' ('true')) and $(1,2,0,1,0)$ (process $p1$ is executing and sets the value of 'active[1]' to '1'). These abstracted states and their transitions are given in Figure 5. The state that violates the mutual exclusion property is the state where both $p0$ and $p1$ are in the critical section, i.e., the state $(5,5,x,y,z)$ where x, y and z are of any value. To verify the correctness of Peterson's algorithm, it is enough to check that the state $(5,5,x,y,z)$ is not reachable from the initial state $(1,1,0,0,0)$.

A straightforward specification of the correctness is given in Figure 6, where we make use of the reachability clause defined previously. However, this definition (read: program) will take too long to execute since it is exponential in the number of states. A better definition (in terms of performance) is the one shown in Figure 7 where we use a breadth-first-search approach in exploring the state space. This "definition" runs in polynomial time. Here we use a method similar to `enum` (see Section 4) for enumerating all possible next states from a given state, that is, the predicate `getnext` in the figure.

7 Example: π-calculus and bisimulation

This section gives an implementation of the specifications of the π-calculus [12] and strong bisimulation. More details on the adequacy of the encodings presented in this section can be found in [24, 22]. We consider only finite π-calculus, 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 `P`, `Q`, `R`, `S` and `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 `P` as its scope. The set of free names in `P` is denoted by $\text{fn}(\mathtt{P})$, the set of bound names is denoted by $\text{bn}(\mathtt{P})$. We write $\text{n}(\mathtt{P})$ for the set $\text{fn}(\mathtt{P}) \cup \text{bn}(\mathtt{P})$. We consider processes to be syntactical equivalent up to renaming of bound names. The operator $+$ denotes the choice operator: a process $P+Q$ can behave either like P or Q. The operator $|$ denotes parallel composition: the process $P|Q$ consists of subprocesses P and Q running in parallel. The process $[x=y]P$ behaves like P if x is equal to y. The process $x(y).P$ can input a name through x, which is then bound to y. The process $\bar{x}y.P$ can output the name y through the channel x. Communication takes place

z : p	in : $n \rightarrow (n \rightarrow p) \rightarrow p$	out : $n \rightarrow n \rightarrow p \rightarrow p$
plus : $p \rightarrow p \rightarrow p$	par : $p \rightarrow p \rightarrow p$	taup : $p \rightarrow p$
nu : $(n \rightarrow p) \rightarrow p$	tau : a	up : $n \rightarrow n \rightarrow a$
dn : $n \rightarrow n \rightarrow a$	one : $p \rightarrow a \rightarrow p \rightarrow o$	onep : $p \rightarrow (n \rightarrow a) \rightarrow (n \rightarrow p) \rightarrow o$

$$\begin{array}{ll}
\llbracket 0 \rrbracket = \mathrm{z} & \llbracket [x=y]\mathtt{P} \rrbracket = \mathrm{match\ x\ y}\ \llbracket \mathtt{P} \rrbracket \\
\llbracket \bar{x}y.\mathtt{P} \rrbracket = \mathrm{out\ x\ y}\ \llbracket \mathtt{P} \rrbracket & \llbracket x(y).\mathtt{P} \rrbracket = \mathrm{in\ x}\ \lambda y.\llbracket \mathtt{P} \rrbracket \\
\llbracket \mathtt{P} + \mathtt{Q} \rrbracket = \mathrm{plus}\ \llbracket \mathtt{P} \rrbracket\ \llbracket \mathtt{Q} \rrbracket & \llbracket \mathtt{P} | \mathtt{Q} \rrbracket = \mathrm{par}\ \llbracket \mathtt{P} \rrbracket\ \llbracket \mathtt{Q} \rrbracket \\
\llbracket \tau.\mathtt{P} \rrbracket = \mathrm{taup}\ \llbracket \mathtt{P} \rrbracket & \llbracket (x)\mathtt{P} \rrbracket = \mathrm{nu}\ \lambda x.\llbracket \mathtt{P} \rrbracket \\
\llbracket \mathtt{P} \xrightarrow{\tau} \mathtt{Q} \rrbracket = \mathrm{one}\ \llbracket \mathtt{P} \rrbracket\ \mathrm{tau}\ \llbracket \mathtt{Q} \rrbracket & \llbracket \mathtt{P} \xrightarrow{\bar{x}y} \mathtt{Q} \rrbracket = \mathrm{one}\ \llbracket \mathtt{P} \rrbracket\ (\mathrm{up\ x\ y})\ \llbracket \mathtt{Q} \rrbracket \\
\llbracket \mathtt{P} \xrightarrow{x(y)} \mathtt{Q} \rrbracket = \mathrm{onep}\ \llbracket \mathtt{P} \rrbracket\ (\mathrm{dn\ x})\ (\lambda y \llbracket \mathtt{Q} \rrbracket) & \llbracket \mathtt{P} \xrightarrow{\bar{x}(y)} \mathtt{Q} \rrbracket = \mathrm{onep}\ \llbracket \mathtt{P} \rrbracket\ (\mathrm{up\ x})\ (\lambda y \llbracket \mathtt{Q} \rrbracket)
\end{array}$$

Fig. 8. Translation from π-calculus syntax to λ-tree syntax.

between two processes running in parallel through the exchanges of messages (names) on the same channel (another name). The restriction operator (), e.g., in $(x)P$, restricts the scope of the name x to P.

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* $\bar{x}(y)$. Since we are working with the *late* transition semantics [12], we shall not be concerned with the free input action. 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 λ-tree 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. The translation from π-calculus processes and transition judgments to λ-tree syntax is given in Figure 8. Figure 10 shows some example processes in λ-tree syntax. The definition clauses corresponding to the operational semantics of π-calculus are given in Figure 9. The original specification of the late semantics of π-calculus can be found in [12]. We note that various side conditions on names and their scopes in the inference rules in the original specification are not present in the encoding in Figure 9. These side conditions are taken care of implicitly by the use of higher-order abstract syntax in the encoding.

We consider some simple examples involving one-step transitions, using the example processes in Figure 10. We can for instance check whether a process is stuck, i.e., no transition is possible from the given process. Consider example 0 in Figure 10 which corresponds to the process $(x)[x = a]\tau.0$. This process clearly cannot make any transition since the name x has to be distinct with respect to the free names in the process. This is specified as follows

```
?- example 0 P, (pi A\pi Q\ one P A Q => false),
% bound input
onep (in X M) (dn X) M.
% free output
one  (out X Y P) (up X Y) P.
% tau
one  (taup P) tau P.
% match prefix
one  (match X X P) A Q := one P A Q.
onep (match X X P) A M := onep P A M.
% sum
one  (plus P Q) A R := one P A R.
one  (plus P Q) A R := one Q A R.
onep (plus P Q) A M := onep P A M.
onep (plus P Q) A M := onep Q A M.
% par
one  (par P Q) A (par P1 Q) := one P A P1.
one  (par P Q) A (par P Q1) := one Q A Q1.
onep (par P Q) A (x\par (M x) Q) := onep P A M.
onep (par P Q) A (x\par P (N x)) := onep Q A N.
% restriction
one  (nu x\P x) A (nu x\Q x) :=  nabla x\ one (P x) A (Q x).
onep  (nu x\P x) A (y\ nu x\Q x y) := nabla x\ onep (P x) A (y\ Q x y).
% open
onep  (nu y\M y) (up X) N := nabla y\ one (M y) (up X y) (N y).
% close
one (par P Q) tau (nu y\ par (M y) (N y)) :=
   sigma X\ onep P (dn X) M & onep Q (up X) N.
one (par P Q) tau (nu y\ par (M y) (N y)) :=
   sigma X\ onep P (up X) M & onep Q (dn X) N.
% comm
one (par P Q) tau (par R T) := sigma X\ sigma Y\ sigma M\ onep P (dn X) M
   & one Q (up X Y) T & (R = (M Y)).
one (par P Q) tau (par R T) := sigma X\ sigma Y\ sigma M\ onep Q (dn X) M
   & one P (up X Y) R & (T = (M Y)).
```

Fig. 9. Definition of one-step transitions of finite late π-calculus

```
example 0 (nu x\ match x a (taup z)).
example 1 (par (in x y\z) (out x a z)).
example 2 (in x u\ (plus (taup (taup z)) (taup z))).
example 3 (in x u\ (plus (taup (taup z))
          (plus (taup z) (taup (match u y (taup z)))))).
example 4 (taup z).
example 5 (nu x\ (par (in x y\z) (out x a z))).
example 6 (in x u\ nu y\ ((plus (taup (taup z))
           (plus (taup z) (taup (match u y (taup z))))))).
```

Fig. 10. Some example processes

```
bisim P Q :=
    (pi A\ pi P1\ one P A P1 => sigma Q1\ one Q A Q1 & bisim P1 Q1) &
    (pi X\ pi M\ onep P (dn X) M => sigma N\ onep Q (dn X) N &
                   pi w\ bisim (M w) (N w)) &
    (pi X\ pi M\ onep P (up X) M => sigma N\ onep Q (up X) N &
                   nabla w\ bisim (M w) (N w)) &
    (pi A\ pi Q1\ one Q A Q1 => sigma P1\ one P A P1 & bisim Q1 P1) &
    (pi X\ pi N\ onep Q (dn X) N => sigma M\ onep P (dn X) M &
                   pi w\ bisim (N w) (M w)) &
    (pi X\ pi N\ onep Q (up X) N => sigma M\ onep P (up X) M &
                   nabla w\ bisim (N w) (M w)).
```

Fig. 11. Definition of open bisimulation

```
   (pi A\pi Q\ onep P A Q => false).
Yes
```

Recall that we distinguish between bound-action transition and free-action transition, and hence there are two kinds of transitions to be verified.

We now consider a notion of equivalence between processes, called *strong bisimulation.* It is formally defined as follows: a relation $\mathcal{R}$ is a bisimulation, if it is a symmetric relation such that for every $(\mathtt{P}, \mathtt{Q}) \in \mathcal{R}$,

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 $(\mathtt{P}', \mathtt{Q}') \in \mathcal{R}$,
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 $(\mathtt{P}'[y/z], \mathtt{Q}'[y/z]) \in \mathcal{R}$ 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 $(\mathtt{P}', \mathtt{Q}') \in \mathcal{R}$.

Two processes $\mathtt{P}$ and $\mathtt{Q}$ are strongly bisimilar if there is a bisimulation $\mathcal{R}$ such that $(\mathtt{P}, \mathtt{Q}) \in \mathcal{R}$.

The above definition is usually referred to as *late bisimulation*, one of the variants of bisimulation existing in the literature. Its encoding in Level 0/1 prover is given in Figure 11. Notice that the difference between bound-input and bound-output actions are captured by the use of $\forall$ and ∇ quantifiers. Actually this definition does not encode fully the notion of late bisimulation, but it is a sound encoding, meaning that if bisim P Q is provable then P and Q are late-bisimilar. The encoding also turns out to correspond to the *open bisimulation* [18], a finer bisimulation relation then late bisimulation (see [24] for details of the encoding and adequacy results). Here is a counterexample that shows the incompleteness with respect to late bisimulation.

$$P = x(u).(\tau.\tau.0 + \tau.0), \qquad Q = x(u).(\tau.\tau.0 + \tau.0 + \tau.[u = y]\tau.0).$$

This happens to be an example that separates open and late bisimulation [18]. This example fails because to prove their bisimilarity, one needs to do case analysis on the input name u above, i.e., whether it is equal to y or not, and our current prover cannot handle such case split (since we are in intuitionistic setting). However, if we restrict the scope of y so that it appears inside the scope of u, then $[u = y]$ is trivially false. In this case, the processes would be $x(u).(\tau.\tau.0 + \tau.0)$ and $x(u).(y)(\tau.\tau.0 + \tau.0 + \tau.[u = y]\tau.0)$, which correspond to example 3 and 6 in Figure 10. They can be proved bisimilar.

```
?- example 2 P, example 6 Q, bisim P Q.
Yes
```

8 Example: modal logics for π-calculus

We now consider the modal logics for π-calculus introduced in [13]. In order not to confuse meta-level ($FO\lambda^{\Delta\nabla}$) formulas (or connectives) with the formulas (connectives) of modal logics under consideration, we shall refer to the latter as object formulas (respectively, object connectives). We shall work only with object formulas which are in negation normal form, i.e., negation appears only at the level of atomic object formulas. As a consequence, we introduce explicitly each dual pair of the object connectives. Note that since the only atomic object formulas are either true or false, by de Morgan duality ¬true ≡ false and ¬false ≡ true. Therefore we are in effect working with positive formulas only. The syntax of the object formulas is given by

$$\begin{aligned} \mathtt{A} ::= {} & \mathrm{true} \mid \mathrm{false} \mid \mathtt{A} \wedge \mathtt{A} \mid \mathtt{A} \vee \mathtt{A} \mid [x = z]\mathtt{A} \mid \langle x = z\rangle\mathtt{A} \\ & \mid \langle\alpha\rangle\mathtt{A} \mid [\alpha]\mathtt{A} \mid \langle\bar{x}(y)\rangle\mathtt{A} \mid [\bar{x}(y)]\mathtt{A} \mid \langle x(y)\rangle^L\mathtt{A} \mid [x(y)]^L\mathtt{A} \end{aligned}$$

Here, α denotes a free action, i.e., it is either τ or $\bar{x}y$. The modalities $[x(y)]^L$ and $\langle x(y)\rangle^L$ are the *late bound-input* modalities, and $\langle\bar{x}(y)\rangle$ and $[\bar{x}(y)]$ are the bound output modalities. There are other variants of input and output modalities considered in [13] which we do not represent here. For the complete encoding of the modal logics, we refer the interested readers to [23]. In each of the formulas (and their dual 'boxed'-formulas) $\langle\bar{x}(y)\rangle\mathtt{A}$ and $\langle x(y)\rangle^L\mathtt{A}$, the occurrence of y in parentheses is a binding occurrence whose scope is $\mathtt{A}$. Object formulas are considered equivalent up to renaming of bound variables. We shall be concerned with checking whether a process $\mathtt{P}$ satisfies a given modal formula $\mathtt{A}$. This satisfiability judgment is written as $\mathtt{P} \models \mathtt{A}$. The translation from modal formulas and judgments to λ-tree syntax is given in Figure 12.

The satisfiability relation for the modal logic is encoded as the definition clauses in Figure 13. For the original specification, we refer the interested readers to [13]. The definition in Figure 13 is not complete, in the sense that there are true assertion of the modal logic which are not provable using this definition alone. For instance, the modal judgment

$$x(y).x(z).0 \models \langle x(y)\rangle^L\langle x(z)\rangle^L(\langle x = z\rangle\mathrm{true} \vee [x = z]\mathrm{false})$$

which basically says that two names are either equal or not equal, is valid, but its encoding in $FO\lambda^{\Delta\nabla}$ is not provable since the meta logic is intuitionistic. A

top : o', bot : o',	and : $o' \to o' \to o'$, or : $o' \to o' \to o'$
boxMatch : $n \to n \to o' \to o'$,	diaMatch : $n \to n \to o' \to o'$,
boxAct : $a \to o' \to o'$,	diaAct : $a \to o' \to o'$,
boxInL : $n \to (n \to o') \to o'$,	diaInL : $n \to (n \to o') \to o'$
boxOut : $n \to (n \to o') \to o'$,	diaOut : $n \to (n \to o') \to o'$
sat : $p \to o' \to o$.	

$[\![\text{true}]\!] = \text{top}$	$[\![\text{false}]\!] = \text{bot}$
$[\![\mathtt{A} \wedge \mathtt{B}]\!] = \text{and } [\![\mathtt{A}]\!]\ [\![\mathtt{B}]\!]$	$[\![\mathtt{A} \vee \mathtt{B}]\!] = \text{or } [\![\mathtt{A}]\!]\ [\![\mathtt{B}]\!]$
$[\![[x = y]\mathtt{A}]\!] = \text{boxMatch x y } [\![\mathtt{A}]\!]$	$[\![\langle x = y\rangle\mathtt{A}]\!] = \text{diaMatch x y } [\![\mathtt{A}]\!]$
$[\![\langle\alpha\rangle\mathtt{A}]\!] = \text{diaAct } \alpha\ [\![\mathtt{A}]\!]$	$[\![[\alpha]\mathtt{A}]\!] = \text{boxAct } \alpha\ [\![\mathtt{A}]\!]$
$[\![\langle x(y)\rangle^L\mathtt{A}]\!] = \text{diaInL x } (\lambda y[\![\mathtt{A}]\!])$	$[\![[x(y)]^L\mathtt{A}]\!] = \text{boxInL x } (\lambda y[\![\mathtt{A}]\!])$
$[\![\langle\bar{x}(y)\rangle\mathtt{A}]\!] = \text{diaOut x } (\lambda y[\![\mathtt{A}]\!])$	$[\![[\bar{x}(y)]\mathtt{A}]\!] = \text{boxOut x } (\lambda y[\![\mathtt{A}]\!])$
$[\![\mathtt{P} \models \mathtt{A}]\!] = \text{sat } [\![\mathtt{P}]\!]\ [\![\mathtt{A}]\!]$	

Fig. 12. Translation from modal formula to λ-tree syntax.

```
sat P top.
sat P (and A B) := sat P A, sat P B.
sat P (or A B) := sat P A ; sat P B.
sat P (boxMatch X Y A) := (X = Y) => sat P A.
sat P (diaMatch X Y A) := (X = Y), sat P A.
sat P (boxAct X A) := pi P1\ one P X P1 => sat P1 A.
sat P (diaAct X A) := sigma P1\ one P X P1, sat P1 A.
sat P (boxOut X A) := pi Q\ onep P (up X) Q => nabla y\ sat (Q y) (A y).
sat P (diaOut X A) := sigma Q\ onep P (up X) Q, nabla y\ sat (Q y) (A y).
sat P (boxInL X A) := pi Q\ onep P (dn X) Q => sigma y\ sat (Q y) (A y).
sat P (diaInL X A) := sigma Q\ onep P (dn X) Q, pi y\ sat (Q y) (A y).
```

Fig. 13. Specification of a modal logic for π-calculus.

complete encoding of the modal logic is given in [23] by explicitly keeping track of the free names introduced by ∇.

The definition in Figure 13 serves also as a model checker for π-calculus. For instance, consider the processes 2 and 6 given by in Figure 10. We have seen that the two processes are bisimilar. A characterization theorem given in [13] states that (late) bisimilar processes satisfy the same set of modal formulas. We consider a particular case here. The modal formula

$$\langle x(y)\rangle^L(\langle\tau\rangle\langle\tau\rangle\text{true} \vee \langle\tau\rangle\text{true})$$

naturally corresponds to the process 2. In the concrete syntax, this formula is written as follows

```
assert (diaInL x (y\ or (diaAct tau (diaAct tau top))
          (diaAct tau top))).
```

We show that both processes 2 and 6 satisfy this formula.

```
datatype 'a cell = delayedcell of unit -> 'a | forcedcell of 'a
type     'a elm = 'a cell
datatype 'a ustream = empty | ustream of 'a * ('a ustream elm ref)
fun getcell(t as ref(delayedcell t')) =
      let val v = t'() in (t := (forcedcell v); v) end
    | getcell(ref (forcedcell v)) = v
fun mkcell t = ref(delayedcell t)
```

Fig. 14. The stream datatype in ML.

```
?- assert A, example 2 P, example 6 Q, sat P A, sat Q A.
Yes
```

9 Components of proof search implementation

Implementation of proof search for $FO\lambda^{\Delta\nabla}$ is based on a few simple key components: the λ-tree syntax, i.e., data structures for representing λ-terms and its parser, higher-order pattern unification and stream-based computation. The first two are implemented using the *suspension calculus* [15], an efficient data structure for representing λ-terms and computing unification on them. We explain the last component briefly. We use streams to store answer substitutions, which are computed lazily, i.e., only when they are queried. The data type for stream in the ML language is shown in Figure 14. Here the type `ustream` is a polymorphic stream. The element of a stream is represented as the data type `cell`, which can be a *delayed cell* or a *forced cell*. A delayed cell stores an unevaluated expression, and its evaluation is triggered by the call to the function `getcell`. A forced cell is an element which is already a value. Elements of a stream are initially created as delayed cells. Note that since an element of a stream can also be a (cell of) stream, we can encode different computation paths using streams of streams. This feature is used, in a particular case, to encode the notion of backtracking in logic programming.

A stream of substitutions for a given goal stores all answer substitutions for the goal. In logic programming, such answer substitutions can be queried one by one by users. Often we are interested in properties that hold for *all* answer substitutions. For instance, in bisimulation checking for transition systems, as we have seen in the π-calculus example, one needs to enumerate all possible successors of a process and check bisimilarity for each successor. In some other examples, information on failed proof search attempts could be of interest as well, e.g., generating counter-model in model checking. This motivates the choice of implementation architecture for $FO\lambda^{\Delta\nabla}$: various fragments of $FO\lambda^{\Delta\nabla}$ are implemented as (specialized) automated provers which interacts with one another. For the current implementation, interaction between provers are restricted to exchanging streams of answer substitutions. A particular arrangement of the interaction between provers that we found quite useful is what we call a $\forall\exists$-interaction. In its simplest form, this consists of two provers, as examplified in the Level-0/1 prover. Recall that in Level-0/1 prover, a proof search session consists of Level-1 calling the Level-0 prover, extracting all answer substitutions, and for each answer substitutions, repeating the calling cycle until the goals are proved. At the implementation level, one can generalize the provers beyond two levels using the same implementation architecture. For instance, one can imagine implementing a "Level-2 prover" which extracts answers from a Level-1 prover and perform some computations on them. Using the example of π-calculus, a Level-2 prover would, for instance, allow for proving goals like "P and Q are not bisimilar". This would be implemented by simply calling Level-1 on this goal and declare a success if Level-1 fails.

10 Future work

The current prover implements a fairly restricted fragment of the logic $FO\lambda^{\Delta\nabla}$. We consider extending it to richer fragments to include features like, among others, induction and co-induction proof rules (see, e.g.,[22]) and arbitrary stratified definition (i.e., to allow more than 1-level implication in goals). Of course, with induction and co-induction proofs, there is in general no complete automated proof search. We are considering implementing a *circular proof* search to automatically generates the (co)inductive invariants. Works along this line has been studied in, e.g., [20]. This extended feature would allow us, for example, to reason about bisimulation of non-terminating processes. Another possible extension is inspired by an on going work on giving a game semantics for proof search, based on the duality of success and failure in proof search. Our particular proof search strategy for Level-0/1 prover turns out to correspond to certain $\forall\exists$- and $\exists\forall$-strategies in the game semantics in [10]. The game semantics studied there also applies to richer fragments of logics. It would be interesting to see if these richer fragments can be implemented as well using a similar architecture as in Level-0/1 prover.

We also plan to use more advance techniques to improve the current implementation such as using tabling to store and reuse subproofs. The use of tabled deduction in higher-order logic programming has been studied in [17]. It seems that the techniques studied there are applicable to our implementation, to the Level-0 prover at least, since it is a subset of λProlog. Another possible extension would be a more flexible restriction on the occurrence of logic variables. The current prover cannot yet handle the case where there is a case analysis involving both eigenvariables and logic variables. Study on a notion of higher-order pattern disunification [14] would be needed to attack this problem at a general level. However, we are still exploring examples and applications which would justify this additional complication to proof search. We also plan to study more examples on encoding process calculi and the related notions of bisimulations.

References

1. L.-H. Eriksson. A finitary version of the calculus of partial inductive definitions. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proc. of the Second*

International Workshop on Extensions to Logic Programming, volume 596 of *LNAI*, pages 89–134. Springer-Verlag, 1991.

2. 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.
3. J.-Y. Girard. A fixpoint theorem in linear logic. Email to the linear@cs.stanford.edu mailing list, February 1992.
4. 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.
5. G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.
6. P. Lescanne and H. Comon. Equational problems and disunification. *Journal of Symbolic Computation*, 3 and 4:371–426, 1989.
7. R. McDowell and D. Miller. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science*, 232:91–119, 2000.
8. D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.
9. D. Miller. Unification under a mixed prefix. *J. of Symbolic Computation*, 14(4):321–358, 1992.
10. D. Miller and A. Saurin. A game semantics for proof search: Preliminary results. Accepted at Mathematical Foundations of Programming Semantics, 2005.
11. D. Miller and A. Tiu. A proof theory for generic judgments: An extended abstract. In *Proceedings of LICS 2003*, pages 118–127. IEEE, June 2003.
12. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Part II. *Information and Computation*, pages 41–77, 1992.
13. R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114(1):149–171, 1993.
14. A. Momigliano and F. Pfenning. Higher-order pattern complement and the strict λ-calculus. *ACM Trans. Comput. Logic*, 4(4):493–529, 2003.
15. G. Nadathur and D. S. Wilson. A notation for lambda terms: A generalization of environments. *Theoretical Computer Science*, 198(1-2):49–98, 1998.
16. T. Nipkow. Functional unification of higher-order patterns. In M. Vardi, editor, *LICS93*, pages 64–74. IEEE, June 1993.
17. B. Pientka. *Tabled Higher-Order Logic Programming.* PhD thesis, Carnegie Mellon University, December 2003.
18. D. Sangiorgi. A theory of bisimulation for the π-calculus. *Acta Informatica*, 33(1):69–97, 1996.
19. 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, June 1993.
20. C. Sprenger and M. Dam. On the structure of inductive reasoning: Circular and tree-shaped proofs in the μ-calculus. In A. Gordon, editor, *Proceedings, Foundations of Software Science and Computational Structures (FOSSACS), Warsaw, Poland*, volume 2620 of *LNCS*, pages 425–440. Springer-Verlag, 2003.
21. R. F. Stärk. Cut-property and negation as failure. *International Journal of Foundations of Computer Science*, 5(2):129–164, 1994.
22. A. Tiu. *A Logical Framework for Reasoning about Logical Specifications.* PhD thesis, Pennsylvania State University, May 2004.

23. A. Tiu. Model checking for π-calculus using proof search. Available online, March 2005.
24. A. Tiu and D. Miller. A proof search specification of the π-calculus. In *3rd Workshop on the Foundations of Global Ubiquitous Computing*, 2004.

Appendix

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

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

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

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

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

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

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

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

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

Fig. 15. The core rules of $FO\lambda^{\Delta\nabla}$.