Mixing Finite Success and Finite Failure in an Automated Prover

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

[1] INRIA Lorraine/LORIA
[2] Digital Technology Center and Dept of CS, University of Minnesota
[3] INRIA & LIX, École Polytechnique

Abstract. The operational semantics and typing judgements of modern programming and specification languages are often defined using relations and proof systems. In simple settings, logic programming languages 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 premises, 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 between finite success and finite failure. The implementation techniques for this prover are largely common ones from higher-order 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's architecture and its prototype implementation, Level 0/1, based on the dual interpretation of (finite) success and finite failure in proof search. We show how Level 0/1 provides a high-level and declarative implementation of model checking and bisimulation checking for the (finite) π-calculus.

1 Introduction

The operational semantics and typing judgements of modern programming and specification languages are often defined using relations and proof systems, e.g., in the style of Plotkin's structural operational semantics. In simple settings, higher-order logic programming languages, such as λProlog and Twelf, can be used to provide rather direct and natural interpreters for operational semantics. However, such logic programming languages can provide little more than animation of semantic descriptions: in particular, reasoning about specified languages has to be done outside the system. For instance, checking bisimulation in process calculi needs analyzing 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 the π-calculus [MPW92]), which interferes in a non-trivial way with case analyses. These case analyses cannot be done directly inside the 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 implementation techniques that are common to 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 $FO\lambda^{\Delta\nabla}$ [MT03], which is a logic based on a subset of Church's Simple Theory of Types but extended with fixed points and the ∇ quantifier. In $FO\lambda^{\Delta\nabla}$ quantification over propositions is not allowed but quantifiers can otherwise range over variables of higher-types. Thus the terms of the logic can be simply typed terms, which can be used to encode the λ-*tree syntax* of encoded objects in an operational semantics specification. This style of encoding is a variant of *higher-order abstract syntax* in which meta-level λ-abstractions are used to encode object-level variable binding. The quantifier ∇ is first introduced in [MT03] to help encode the notion of "generic judgment" that occurs commonly when reasoning with λ-tree syntax.

The logical extension to allow fixed points is done through a proof theoretical notion of *definitions* [SH93,Eri91,Gir92,Stä94,MM00]. 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 for which there is a successful computation (proof) of pt, there is a computation (proof) of qt. Towards establishing the truth of this formula, 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 [HSH91,Gir92]. 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 for more details.

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 fail-

ure/success in proof search. Section 4 discusses the treatment of variables in the Level-0/1 prover, in particular, it discusses the issues concerning the interaction between *eigenvariables* and logic variables. Section 5 specifically contrasts the expressiveness of Level-0/1 over what is available in λProlog. Section 6 gives a specification of the operational semantics for the π-calculus and shows how Level-0/1 can turn that specification naturally into a checker for (open) bisimulation. Section 7 provides a specification of modal logic for the π-calculus, which the Level-0/1 prover can use to do model checking for that process calculus. These two specifications involving the π-calculus illustrate the use of the ∇-quantifier to help capture various restrictions of names in the π-calculus. Section 8 discusses the components of proof search implementation and outlines a general implementation architecture for $FO\lambda^{\Delta\nabla}$. Section 9 discusses future work. An extended version of this paper is available on the web, containing more examples and more detailed comparison with logic programming.

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

The logic $FO\lambda^{\Delta\nabla}$ [MT03] (pronounced "fold-nabla") is presented using a sequent calculus that is an extension of Gentzen's system LJ [Gen69] for intuitionistic logic. The first extension to LJ is to allow terms to be simply typed λ-terms and to allow quantification to be over all types not involving the predicate type (in Church's notation [Chu40], the types of quantified variables do not contain o). A *sequent* is an expression of the form $B_1, \ldots, B_n \vdash B_0$ where $B_0, \ldots, B_n$ are formulas and the elongated turnstile $\vdash$ 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 then 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 [MT03] for motivations and examples). In particular, sequents in $FO\lambda^{\Delta\nabla}$ are of the form

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

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

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

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

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

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

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.

The logical constants of $FO\lambda^{\Delta\nabla}$ are $\forall_\gamma$ (universal quantifier), $\exists_\gamma$ (existential quantifier), ∇_γ (nabla quantification), $\wedge$ (conjunction), $\vee$ (disjunction), $\supset$ (implication), $\top$ (true) and $\bot$ (false). The subscript for the three quantifiers is the type of the variable they are intended to bind: in particular, γ can range over any type not containing the predicate type. Usually this type subscript is suppressed. The inference rules for the three quantifiers of $FO\lambda^{\Delta\nabla}$ are given in Figure 1. The introduction rules for propositional connectives are straightforward generalization of LJ: in particular, local signatures are distributed over the subformulas of the main formula (reading the rules bottom-up). The complete set of rules for $FO\lambda^{\Delta\nabla}$ is given in Figure 10 at the end of this paper.

During the search for proofs (reading rules bottom up), the right-introduction rule for $\forall$ and the left-introduction rule for $\exists$ place new variables into the global signature: the left and right introduction rules for ∇ place new variables into the local signature. In the $\forall\mathcal{R}$ and $\exists\mathcal{L}$ rules, raising [Mil92] is used when replacing 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 $\Sigma, \sigma \vdash t : \gamma$. The $\nabla\mathcal{L}$ and $\nabla\mathcal{R}$ rules have the proviso that y is not free in $\nabla x\ B$.

Besides these introduction rules for logical constants, $FO\lambda^{\Delta\nabla}$ additionally allows the introduction of atomic judgments, that is, judgments of the form $\sigma \triangleright A$ where A is an atomic formula. To each atomic judgment, $\mathcal{A}$, we associate a judgment $\mathcal{B}$ called 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 [SH93], Eriksson [Eri91],

Girard [Gir92], Stärk [Stä94] and McDowell and Miller [MM00] and allows 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.*

Let $\forall_{\gamma_1} x_1 \ldots \forall_{\gamma_n} x_n.H \stackrel{\triangle}{=} B$ *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* $\alpha_1 \to \ldots \to \alpha_m \to \gamma_i$*. A* definition *is a set of definition clauses together with their raised clauses.*

The introduction rules for a defined judgment are displayed below. 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.

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

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

In the above rules, we apply substitutions 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 $\mathit{def}\mathcal{L}$ rule, we use the notion of *complete set of unifiers* (CSU) [Hue75]. We denote by $\mathit{CSU}(\mathcal{A}, \mathcal{H})$ a 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 $\rho \in \mathit{CSU}(\mathcal{A}, \mathcal{H})$ 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 a given unification problem. Thus, in the fully general case, $\mathit{def}\mathcal{L}$ may have an infinite number of premises [MM00]. In all the applications of $\mathit{def}\mathcal{L}$ in this paper, however, the terms involved in unification are those of *higher-order patterns* [Mil91,Nip93], that is, terms in which variables are applied only to distinct bound variables. Since higher-order pattern unification is decidable and unary (i.e., the most general unifiers exist if the unification is solvable), the set $\mathit{CSU}(\mathcal{A}, \mathcal{H})$ in this case can be treated as being either empty or containing a single substitution

which is the most general unifier. In this restricted setting, $def\mathcal{L}$ will have a finite number of premises (assuming as we shall that definitions are based on the raising of only a *finite* number of clauses). The signature $\Sigma\theta$ in $def\mathcal{L}$ denotes the 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 $defR$ 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 considered proved.

One might find the following analogy with logic programming helpful: if a definition is viewed as a logic program, then the $defR$ 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 the definition

$$\{pa \stackrel{\triangle}{=} \top,\ \ pb \stackrel{\triangle}{=} \top,\ \ qa \stackrel{\triangle}{=} \top,\ \ qb \stackrel{\triangle}{=} \top,\ \ qc \stackrel{\triangle}{=} \top\},$$

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 $def\mathcal{L}$ [HSH91], e.g., for the above program (definition), the goal $not(pc)$ in logic programming is encoded as the formula $pc \supset \bot$.

Two properties of $FO\lambda^{\Delta\nabla}$ are particularly important to note here. First, if a certain stratification of predicates within definitions is made (so that there is no circularity in defining predicates through negations) then cut-elimination and consistency can be proved [MT05,Tiu04]. Second, many inference rules are known to be *invertible*, in the sense that they 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 $\vee$ [Tiu04]. The invertibility of these rules motivates the choice of the fragment of $FO\lambda^{\Delta\nabla}$ on which the Level-0/1 prover works.

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 formulas which exhibits this duality. We first assume that all predicate symbols are classified as belonging to either level-0 or level-1 (via some mapping of predicates to $\{0, 1\}$). Next consider the following classes of formulas:

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: $A := p\, t_1 \ldots t_n$

Here, atomic formulas A in level 0 formulas must have predicates that have been assigned to level 0. Atomic formulas in level 1 formulas can have predicates of either level 0 and 1. Each definition clause $p\bar{t} \stackrel{\triangle}{=} B$ 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 a level-0 or level-1 formula. In the current implementation, stratification checking and type checking 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 follows the standard logic-programming implementation for Horn clauses: it is actually the subset of λProlog based on Horn clauses but allowing also ∇ quantification in the body of clauses. In this prover, existential quantifiers are instantiated with logic variables, ∇-quantifiers are instantiated with scoped (local) constants (which have to be distinguished from eigenvariables), and $defR$ is implemented via 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 any level-1 eigenvariables as level-0 logic variables.

Step 2 Collect all answer substitutions produced by Step 1 into a lazy stream of substitutions and for each substitution θ in this stream, proceed with proving $G\theta$. For example, if Step 1 fails, then this stream is empty and this step succeeds immediately.

In Step 1, we impose a restriction: the formula G must not contain any occurrences of level-1 logic variables. If this restriction is violated, a runtime exception is returned and proof search is aborted. We shall return to this technical restriction in Section 4. This restriction on the occurrence of logic variable has not posed a problem for a number of applications, e.g., checking bisimulation and satisfiability of modal logic formulas for the π-calculus.

We claim the following soundness theorem for the provers architecture above: If Level-0/1 is given a definition and a goal formula and it successfully claims to have a proof of that goal (that is, the system terminates without a runtime error), then that goal follows from the definition also in the $FO\lambda^{\Delta\nabla}$ logic.

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 an infix backslash, with the body of a λ-abstraction is goes as far to the right as possible, consistent with the existing parentheses: for example, $\lambda x \lambda f.fx$ can be written as `(x\f\ f x)`. The order of precedence for the connectives is as follows (in decreasing order): $\wedge$, $\vee$, $\supset$, $\{\forall, \exists, \nabla\}$. Follow the convention started by Church [Chu40], the bound variable associated to a quantifier is actually a λ-abstraction: for example, the logical expression $\forall x[p(x) \supset q(x)] \wedge p(a)$ can be encoded as the `(pi x\ p x => q x) & (p a)`. Non-logical constants, such as 'not' (negation-as-failure) and '!' (Prolog cut), are not implemented, while we do allow the non-logical constant `print` for printing terms. Finally, we note that the percent sign `%` starts a comment line.

The symbol $\stackrel{\triangle}{=}$ separating the head and the body of a definition clause is written as ':=' 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 formula. 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 never instantiated and are usually represented by λ-abstractions. 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, for example, consider proving the goal

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

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

```
?- 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.

complement problems or *disunification* [LC89], 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 [MP03] the problem is considerably more difficult, and, hence, in the current implementation, we disallow 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 level-0 (i.e., there is no implication in the goal). A non-level 0 goal is given in the third example. Here the unification fails (hence the goal succeeds) because x is bound in the scope of where M is bound. 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 quantifier 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.

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. As we outline below, only in some weak settings can λProlog naturally capture the deduction intended in $FO\lambda^{\Delta\nabla}$.

The $\supset$ connective, for instance, might be defined in λProlog as

```
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 `A` and `B` (thus, containing no eigenvariables), this coincides 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 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: the π-calculus and bisimulation

An implementation of one-step transitions and strong bisimulation for the π-calculus [MPW92] are given in this section. More details on the adequacy of the encodings presented in this section can be found in [TM04,Tiu04]. 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 $\mathtt{P}$, $\mathtt{Q}$, $\mathtt{R}$, $\mathtt{S}$ 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 syntactically 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 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 [MPW92], 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 with a binding occurrences of a name is a *bound action*, otherwise it is a *free 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 3. Figure 4 shows some example processes in λ-tree syntax. The definition clauses corresponding to the operational semantics of π-calculus are given in Figure 5. The original specification of the late semantics of π-calculus can be found in [MPW92]. 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 5 since these are handled directly by the use of λ-tree syntax and the $FO\lambda^{\Delta\nabla}$ logic.

We consider some simple examples involving one-step transitions, using the example processes in Figure 4. We can, for instance, check whether a process is

$\mathrm{z} : p$	$\mathrm{in} : n \to (n \to p) \to p$	$\mathrm{out}, \mathrm{match} : n \to n \to p \to p$
$\mathrm{plus} : p \to p \to p$	$\mathrm{par} : p \to p \to p$	$\mathrm{taup} : p \to p$
$\mathrm{nu} : (n \to p) \to p$	$\mathrm{tau} : a$	$\mathrm{up} : n \to n \to a$
$\mathrm{dn} : n \to n \to a$	$\mathrm{one} : p \to a \to p \to o$	$\mathrm{onep} : p \to (n \to a) \to (n \to p) \to o$

$[\![0]\!] = \mathrm{z}$	$[\![[x = y]\mathtt{P}]\!] = \mathrm{match\ x\ y}\ [\![\mathtt{P}]\!]$
$[\![\bar{x}y.\mathtt{P}]\!] = \mathrm{out\ x\ y}\ [\![\mathtt{P}]\!]$	$[\![x(y).\mathtt{P}]\!] = \mathrm{in\ x}\ \lambda y.[\![\mathtt{P}]\!]$
$[\![\mathtt{P} + \mathtt{Q}]\!] = \mathrm{plus}\ [\![\mathtt{P}]\!]\ [\![\mathtt{Q}]\!]$	$[\![\mathtt{P}\vert\mathtt{Q}]\!] = \mathrm{par}\ [\![\mathtt{P}]\!]\ [\![\mathtt{Q}]\!]$
$[\![\tau.\mathtt{P}]\!] = \mathrm{taup}\ [\![\mathtt{P}]\!]$	$[\![(x)\mathtt{P}]\!] = \mathrm{nu}\ \lambda x.[\![\mathtt{P}]\!]$
$[\![\mathtt{P} \xrightarrow{\tau} \mathtt{Q}]\!] = \mathrm{one}\ [\![\mathtt{P}]\!]\ \mathrm{tau}\ [\![\mathtt{Q}]\!]$	$[\![\mathtt{P} \xrightarrow{\bar{x}y} \mathtt{Q}]\!] = \mathrm{one}\ [\![\mathtt{P}]\!]\ (\mathrm{up\ x\ y})\ [\![\mathtt{Q}]\!]$
$[\![\mathtt{P} \xrightarrow{x(y)} \mathtt{Q}]\!] = \mathrm{onep}\ [\![\mathtt{P}]\!]\ (\mathrm{dn\ x})\ (\lambda y[\![\mathtt{Q}]\!])$	$[\![\mathtt{P} \xrightarrow{\bar{x}(y)} \mathtt{Q}]\!] = \mathrm{onep}\ [\![\mathtt{P}]\!]\ (\mathrm{up\ x})\ (\lambda y[\![\mathtt{Q}]\!])$

Fig. 3. Encoding the π-calculus syntax with λ-tree syntax.

```
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. 4. Several examples processes written in Level-0/1 syntax.

stuck, i.e., no transition is possible from the given process. Consider example 0 in Figure 4 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),
                (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 *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 for every name y, $(\mathtt{P}'[y/z], \mathtt{Q}'[y/z]) \in \mathcal{R}$,

```
onep (in X M) (dn X) M.                    % bound input
one  (out X Y P) (up X Y) P.               % free output
one  (taup P) tau P.                       % tau
one  (match X X P) A Q := one P  A Q.      % match prefix
onep (match X X P) A M := onep P A M.
one  (plus P Q) A R := one P A R.          % sum
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.
one  (par P Q) A (par P1 Q) := one P A P1.  % par
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. 5. Definition of one-step transitions of finite late π-calculus

```
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. 6. Definition of open bisimulation

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 P and Q are *strongly bisimilar* if there is a bisimulation $\mathcal{R}$ such that $(\mathtt{P}, \mathtt{Q}) \in \mathcal{R}$. The above definition is also called *late bisimulation* in the literature.

Consider the definition of the `bisim` predicate Figure 6 that is inspired by the above definition. Notice that the difference between bound-input and bound-output actions is captured by the use of $\forall$ and ∇ quantifiers. This definition provides a sound encoding of late bisimulation, meaning that if bisim P Q is provable then P and Q are late-bisimilar. This encoding turns out to sound and complete for *open bisimulation* [San96], a finer bisimulation relation than late bisimulation (see [TM04] for details of the encoding and adequacy results). The following example, taken from [San96], illustrates 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 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 since our current prover implements intuitionistic logic, this case split based on the excluded middle is not available. 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 4. They can be proved bisimilar.

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

One should compare the above declarative specification and its implementation of symbolic bisimulation checking with that found in, say, [BN96].

7 Example: modal logics for π-calculus

We now consider the modal logics for π-calculus introduced in [MPW93]. 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, we will not need negation as a connective (since $\neg$true $\equiv$ false and $\neg$false $\equiv$ true). The syntax of the object formulas is given by

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

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

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

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

Fig. 7. Translation from modal formula to λ-tree syntax.

bound output modalities. There are other variants of input and output modalities considered in [MPW93] which we do not represent here. For the complete encoding of the modal logics, we refer the interested readers to [Tiu05]. 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 A. Object formulas are considered equivalent up to renaming of bound variables. We shall be concerned with checking whether a process P satisfies a given modal formula 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 7.

```
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. 8. Specification of a modal logic for π-calculus.

The satisfiability relation for the modal logic is encoded as the definition clauses in Figure 8. For the original specification, we refer the interested readers to [MPW93]. The definition in Figure 8 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

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

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 complete encoding of the modal logic is given in [Tiu05] by explicitly introducing axioms for the excluded-middle on name equality, namely, $\forall x \forall y[x = y \vee x \neq y]$.

The definition in Figure 8 serves also as a model checker for π-calculus. For instance, consider the processes 2 and 6 given by in Figure 4. We have seen that the two processes are bisimilar. A characterization theorem given in [MPW93] 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.

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

8 Components of proof search implementation

Implementation of proof search for $FO\lambda^{\Delta\nabla}$ is based on a few simple key components: λ-tree syntax, i.e., data structures for representing objects containing binding, higher-order pattern unification, and stream-based computation. The first two are implemented using the *suspension calculus* [NW98], an explicit substitution notation that allows computations over λ-terms to be realized flexibly and efficiently; further details of the implementation used may be found in [NL05]. 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 9. 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

```
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. 9. The stream datatype in ML.

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 interact 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 exemplified 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.

9 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.,[Tiu04]) and arbitrary stratified definition (i.e., to allow more nesting of implications 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., [SD03]. 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 [MS05]. 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 [Pie03]. 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 [MP03] 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.

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

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

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

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

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

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

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

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

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

Fig. 10. The core rules of $FO\lambda^{\Delta\nabla}$.

Acknowledgements. Support has been obtained for this work from the following sources: from INRIA through the "Equipes Associées" Slimmer, from the ACI grants GEOCAL and Rossignol and from the NSF Grant CCR-0429572 that also includes support for Slimmer.

References

[BN96] Michele Boreale and Rocco De Nicola. A symbolic semantics for the π-calculus. *Information and Computation*, 126(1):34–52, April 1996.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *J. of Symbolic Logic*, 5:56–68, 1940.

[Eri91] Lars-Henrik 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.

[Gen69] Gerhard 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.

[Gir92] Jean-Yves Girard. A fixpoint theorem in linear logic. Email to the linear@cs.stanford.edu mailing list, February 1992.

[HSH91] Lars Hallnäs and Peter Schroeder-Heister. A proof-theoretic approach to logic programming. II. Programs as definitions. *Journal of Logic and Computation*, 1(5):635–660, October 1991.

[Hue75] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[LC89] Pierre Lescanne and Hubert Comon. Equational problems and disunification. *Journal of Symbolic Computation*, 3 and 4:371–426, 1989.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[Mil92] Dale Miller. Unification under a mixed prefix. *J. of Symbolic Computation*, 14(4):321–358, 1992.

[MM00] Raymond McDowell and Dale Miller. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science*, 232:91–119, 2000.

[MP03] Alberto Momigliano and Frank Pfenning. Higher-order pattern complement and the strict λ-calculus. *ACM Trans. Comput. Logic*, 4(4):493–529, 2003.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part II. *Information and Computation*, pages 41–77, 1992.

[MPW93] Robin Milner, Joachim Parrow, and David Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114(1):149–171, 1993.

[MS05] Dale Miller and Alexis Saurin. A game semantics for proof search: Preliminary results. In *Proceedings of the Mathematical Foundations of Programming Semantics (MFPS)*, 2005.

[MT03] Dale Miller and Alwen Tiu. A proof theory for generic judgments: An extended abstract. In *LICS 2003*, pages 118–127. IEEE, June 2003.

[MT05] Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Transactions on Computational Logic*, 6(4), October 2005.

[Nip93] Tobias Nipkow. Functional unification of higher-order patterns. In M. Vardi, editor, *LICS93*, pages 64–74. IEEE, June 1993.

[NL05] Gopalan Nadathur and Natalie Linnell. Practical higher-order pattern unification with on-the-fly raising. In *ICLP 2005: 21st International Logic Programming Conference*, volume 3668 of *LNCS*, pages 371–386, Sitges, Spain, October 2005. Springer.

[NW98] Gopalan Nadathur and Debra Sue Wilson. A notation for lambda terms: A generalization of environments. *Theoretical Computer Science*, 198(1-2):49–98, 1998.

[Pie03] Brigitte Pientka. *Tabled Higher-Order Logic Programming.* PhD thesis, Carnegie Mellon University, December 2003.

[San96] Davide Sangiorgi. A theory of bisimulation for the π-calculus. *Acta Informatica*, 33(1):69–97, 1996.

[SD03] Christoph Sprenger and Mads Dam. On the structure of inductive reasoning: Circular and tree-shaped proofs in the μ-calculus. In A.D. Gordon, editor, *Proceedings, Foundations of Software Science and Computational Structures (FOSSACS), Warsaw, Poland*, volume 2620 of *LNCS*, pages 425–440. Springer-Verlag, 2003.

[SH93] Peter 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.

[Stä94] R. F. Stärk. Cut-property and negation as failure. *International Journal of Foundations of Computer Science*, 5(2):129–164, 1994.

[Tiu04] Alwen Tiu. *A Logical Framework for Reasoning about Logical Specifications.* PhD thesis, Pennsylvania State University, May 2004.

[Tiu05] Alwen Tiu. Model checking for π-calculus using proof search. In Martín Abadi and Luca de Alfaro, editors, *CONCUR*, volume 3653 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2005.

[TM04] Alwen Tiu and Dale Miller. A proof search specification of the π-calculus. In *3rd Workshop on the Foundations of Global Ubiquitous Computing*, 2004.