Level 0/1 Prover: A Tutorial

Alwen Tiu
INRIA Futurs / École polytechnique
tiu@lix.polytechnique.fr

DRAFT – September 9, 2004

1 Introduction

This paper gives an overview of a prototype implementation of a fragment of the logic Linc [5, 8, 13] (also referred to as $FO\lambda^{\Delta\nabla}$ in [5]), which we tentatively call 'Level 0/1 Prover' here. This implementation is part of a larger project, Parsifal, at INRIA Futurs (see `http://www.lix.polytechnique.fr/~dale/parsifal`).

The logic Linc is an extension of first-order intuitionistic logic with a proof-theoretic notion of *definitions* [3, 11, 2, 4], proof rules for induction and co-induction and a new quantifier ∇. Our aim is to use Linc as a logic for specifying and reasoning about various operational semantics of computation systems. It is not necessary to go through the detailed description of the logic in order to use the prover; this is the intent of this tutorial, of course. For those who are interested, details of the logic are available in the related papers on Linc mentioned previously. We assume that users are familiar with Prolog, or better, λProlog, as we follow closely the syntax of both languages, especially λProlog. The current implementation is very much experimental so we choose to restrict the features of the language to purely logical ones, that is, most non-logical features commonly found in Prolog-like languages, e.g., arithmetic, string processing, I/O, etc., are not implemented.

This tutorial is organized as follows. Section 2 gives an overview of the language of the prover and its operational semantics (informally), along with some examples illustrating its differences with λProlog. Section 3 shows how to configure and run the prover. Section 4 discusses the treatment of eigenvariables in Level 0/1 prover, and shows how it is different from the use of eigenvariables in λProlog. This section also shows how to give a *logically sound* interpretation to certain non-logical features in Prolog or λProlog. Section 5 discusses a non-logical feature of Level 0/1 prover which allows users to perform analysis on a logic program. What it does is basically transforming a logic program into a certain abstract syntax, called λ-tree syntax, and vice versa. Section 6 shows how to use this feature to do type checking on logic programs. The type checking program itself would be just another logic program that is fed the λ-tree syntax of another program. Section 7 and Section 8 illustrate the use of Level 0/1 prover to reason about transition systems. The examples cover the specification and verification of Peterson's algorithm (for guaranteeing mutual exclusion), and bisimulation of π-calculus. Section 9 mentions some possible directions for future work.

2 Description of the language

The language of the Level 0/1 prover consists of logical formulas. Formulas are divided into two categories: Level-0 formulas and Level-1 formulas (hence the name of the prover). These categories are defined by the following grammars.

$$\begin{array}{ll} \text{Level 0:} & G := \top \mid \bot \mid A \mid G \wedge G \mid G \vee G \mid \exists x.G \mid \nabla x.G \\ \text{Level 1:} & D := \top \mid \bot \mid A \mid D \wedge D \mid D \vee D \mid G \supset D \mid \exists x.D \mid \nabla x.D \mid \forall x.D \\ \text{atomic:} & A := p\, t_1 \ldots t_n \end{array}$$

where $\top$, $\bot$, $\wedge$, $\vee$, $\supset$, $\exists$, $\forall$ denote the usual logical connectives true, false, conjunction, disjunction, (intuitionistic) implication, existential quantifier, and universal quantifier, respectively. The symbol p ranges over predicate symbols, and t_i ranges over simply typed λ-terms. The ∇-quantifier is a new connective whose intended meaning is to provide a fresh constant within its scope. Its proof search behavior, for Level-0 goal, is similar to that of $\forall$, that is, to provide new eigenvariables during proof search. We have not included the type information in the above syntax, but it should be understood that the formulas are simply-typed, following Church [1], and first-order. That is, given the type o of propositions, the types of the variables in the formulas should not contain o. This typing constraint is not enforced in the current implementation, so users should be aware that although the current prover can accept programs or goals with higher-order variables, the operational behaviors of such programs or goals will fall outside the scope of the logic Linc.

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, while $A \supset (B \supset C)$ is permitted. This use of implication is more restricted than in λProlog. There is actually another restriction that is not captured by the grammar above. That is, in the proof search for a goal formula $B \supset D$, there must not be any occurrence of logic variables in B. We shall come back to this restriction in Section 4.

Unlike λProlog, logic programs in Level 0/1 prover are not defined as logical formulas. Instead, we treat logic programs as a separate syntactic categories, which is called *definitions*. A definition is a collection of *definitional clauses*. A definition clause is written

$$p\, t_1 \ \ldots \ t_n \stackrel{\triangle}{=} B$$

where p is a predicate symbol, $p\, t_1 \ldots t_n$ is the *head* of the definition, and B is the *body* of the definition, which in this case is just a Level-1 formula. Although there is some proof-theoretical significance to this distinction between program clause and definition clause, in most cases, users can ignore this distinction and think of definition clause as simply program clause, and definitions as logic programs.

Concrete syntax The concrete syntax for the logical connectives follow the same syntax as λProlog:

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

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. So in a sense programs in Level 0/1 prover are pure logic programs. However, we will see later that negation-as-failure can be derived from other connectives.

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. The character '%' is used to mark a comment, and lines starting with the character are ignored by the parser of the prover.

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.
```

In the following, we shall often switch between abstract syntax and concrete syntax. Abstract syntax will be often used at the meta-level discussions, while concrete syntax is used when presenting the actual screen output of the prover.

Proof search Proof search for goals involving the logical connectives $\wedge$, $\vee$, $\exists$, $\forall$, *true* and *false* are similar to λProlog. Here is an informal description of the proof search procedure (see [5] for the formal counterpart described in sequent calculus). The proof engine consists of two provers: Level-0 prover (for Level-0 goals) and Level-1 prover (for Level-1 goals).

Recall that the operational meaning of $\nabla x.G$ is that of introducing a new constant whose scope is over G. This means that during proof search, there can be three kinds of variables introduced: logic variables (by the $\exists$ quantifier), eigenvariables (by the $\forall$ quantifier) and ∇ variables (by the ∇ quantifier). Of these three variables, only ∇-variables stay immutable during proof search, the other two can be instantiated. In order to find a proper instantiation to variables, we need to keep track of the order of the introduction of these variables. Let us assign a 'timestamp' to each variable introduced during proof search. The "law" of instantiation order is as follows:

- ∇ variables are not instantiated,
- a logic (or eigen) variable X can be instantiated to a term t if t contains no free variables with timestamp greater than the timestamp of X.

Having this law in mind, in the following description of the proof search procedure, we assume implicitly that appropriate timestamp is assigned to each newly introduced variable. Eigenvariables are normally treated as constants, except on one case (see step 6 in Level-1 prover below).

Level-0 prover:

0 To prove an atomic formula A, select a definition clause $H \stackrel{\triangle}{=} B$, (after renaming all its free variables with some new logic variables) such that A unifies with H, with the unifier θ, and continue proving $B\theta$.

1 To prove $G_1 \wedge G_2$: first call Level-0 prover on G_1, if it returns an answer substitution θ, continue proving $G_2\theta$, otherwise, terminate with failure.

2 To prove $G_1 \vee G_2$: prove either G_1 or G_2.

3 To prove $\exists x.G$: prove $G[X/x]$ where X is a new logic variable.

4 To prove $\nabla x.G$: prove $G[c/x]$ where c is a new ∇ variable.

Level-1 prover (main prover):
steps 0 - 4 are as in Level-0, the additional steps are as follows

5 To prove $\forall x.G$: prove $G[Y/x]$ where Y is a new eigenvariable.

6 To prove $G \supset D$: call Level-0 prover on G. G must not contain any occurrence of logic variables, otherwise the proof search terminates with failure. Prior calling the Level-0 prover, all eigenvariables in G are "marked" as logic variables so that they are subject to instantiation by Level-0 prover. For each answer substitution θ returned by the Level-0 prover, prove $D\theta$. If there is no answer substitution for G, the proof for $G \supset D$ succeeds.

Notice that Step 6 above is a non-trivial departure from the treatment of eigenvariable in λProlog. In λProlog, eigenvariables are not instantiated during proof search. This additional feature allows us to perform *case analyses* on how a (hypothetical) goal might be proved. For example, given a program (definition)

```
p a.
p b.
p c.
```

the following goal is provable

```
?- pi x\ p x => (x = d) => false.
```

That is, if $p\,x$ is provable, then it cannot be the case that x is equal to d, which is true, since p is true only of a, b or c.

Equality We introduce an equality predicate '=' which checks for equality modulo $\beta\eta$-conversion between terms. We can think of this equality is implicitly defined by the following definition

$$X = X \stackrel{\triangle}{=} \top$$

where we use infix notation to write the application of the equality symbol. Its proof search is therefore the same as proof search for defined atoms. That is, in proving a goal $s = t$ the prover would attempt to unify both terms and would either return an answer substitution which is the most general unifier, if it exists, or fail if there is no possible unifier.

If-then-else For convenience, an 'if-then-else' connective is added to the language. Its syntax is as follows

$$\text{if } P\ Q\ R$$

where P is a closed Level-0 formula (that is, it contains no occurrences of logic variables), and Q and R are Level-1 formulas. The formula itself is a Level-1 formula. The 'if-then-else' connective is a *synthetic connective*, that is, it can be defined using the other logical connecitives as follows (we use the symbol $\stackrel{\text{def}}{=}$ to denote meta-level definition)

$$\text{if } P\ Q\ R \stackrel{\text{def}}{=} (P \wedge Q) \vee [(P \supset \bot)) \wedge R].$$

Notice that P appears to the left of the implication in the second disjunct, and hence the restriction on the occurrence of logic variables applies. The proof search behavior of 'if P Q R' is as expected: first try to prove P, if the goal succeeds then continue proving Q, otherwise prove R.

Similarly, negation-as-failure can be defined as

$$(P \supset \bot)$$

where, again, P must not contain any occurrences of logic variables. It can alternatively be defined via 'if-then-else', that is,

$$\text{if } P\ \bot\ \top.$$

Note that this does not capture fully the notion of negation-as-failure as it is commonly found in logic programming since we do not allow logic variables to appear in the goal.

I/O. In addition to the logical constants mentioned above, Level 0/1 has interpreted predicates called 'print', which does the printing of a term or a string to the standard output, and 'input' which input a term from standard input. The argument to 'print' can be a string (enclosed in quotes), e.g., as in

```
?- print "Hello, World!\n".
Hello, World!
Yes
```

or a term, e.g.,

```
?- print (P & Q => R).
(P & Q) => R
Yes
```

Notice that 'print' is a higher-order predicate since it can take a formula as its argument. The proof search behavior of 'print' is the same as 'true', i.e., it always succeeds.

The 'input' predicate takes an arbitrary term (e.g., could be a variable) as the argument. This argument is then matched against the user input term. User can type in any λ-terms as input, and it must be ended with a '.' to indicate the end of input. One can simply bind a variable to a user input term, or do a pattern match. Consider the following examples

```
?- print "Input a term: ",  input L.
Input a term: (a b) => true.
Yes
L = (a b) => true.

?- print "Input a term: ", input (cons A L).
Input a term: (cons a (cons b nil)).
Yes
A = a
L = (cons b nil)
```

The input term can be arbitrary higher-order term, since we do not impose any typing constraints on input. Therefore, one again has to be aware that this use of higher-order input falls outside the scope of the logic Linc. The 'input' statement can actually be used to input terms containing eigenvariables (that is, if one can guess variable names which are used for the eigenvariables), although such use is not well-behaved and is therefore not recommended.

3 Running the prover

The Level 0/1 prover is implemented in Standard ML/New Jersey version 110.46. We assume that users have this or a later version of SML installed. For details concerning the installation of SML, consult the SML website `http://www.smlnj.org`

The binary files for Level01 prover can be downloaded from `http://www.lix.polytechnique.fr/~tiu/linc.htm`. The executable file for the prover is named 'level01'. This is actually a heap-image of SML. We first need to load this image into sml runtime environment prior loading the prover. To load the image file, run SML with the following command

```
sml @SMLload=level01
```

The SML command prompt should appear. Now there are several functions that you can try.

- *To load the prover:* use the following command

  ```
  go <filename>;
  ```

  where <filename> is to be substituted by the file name containing the definition clauses, in quotes. Note that we are still in SML environment, so the statement must end with a semi-colon. For example, to load a file called append.def in the current working directory, type the following.

  ```
  go "append.def";
  ```

  The following prompt will appear after a successful execution of the command

  ```
  ?-
  ```

  indicating that now we are in the Level 0/1 prover.

- *To set the working directory:* all references to files are made with respect to the current working directory, except when the full path is given. To change the working directory, use the 'set_path' command. For example, the following command set the working directory to '/home/user/linc'.

  ```
  - set_path "/home/user/linc";
  ```

  Note that this is done inside the SML interpreter but not inside the Level 0/1 prover.

- *Loading multiple files:* different files can be loaded simultaneously by the command 'golist'. This function takes as argument the list of file names. For example, to load the files "member.def" and "append.def", type the following

  ```
  - golist ["member.def", "append.def"];
  ```

- *Debugging option:* Prior running the prover, users can turn on the debugging feature of the prover. If this feature is set on, a complete execution trace will be shown for each running of the query. To set the debugging on, type the following command in SML runtime environment:

  ```
  - Core.set_debug true;
  ```

  To turn the debugging option off, do the following

  ```
  - Core.set_debug false;
  ```

Merging definition files A definition file can be merged into another definition file via the "include" statement in the latter file. The syntax for the include statement is as follows.

```
include <file>.
```

where <file> is the name of the file to be included. The include-statement in a definition file causes the definition clauses in the included file to be included, at the point where the statement occurs. For instance, suppose we have a definition file called 'member.def' which contains the definition clauses

```
member X (cons X L).
member X (cons Y L) := member X L.
```

Suppose also we would like to use this definition in another definition for a predicate called 'distinct', which checks whether a list is distinct or not. This is simply done by putting an include-statement in the definition file for the 'distinct'-predicate.

```
include "member.def".

distinct nil.
distinct (cons X L) := (member X L => false), distinct L.
```

The parser of Level 0/1 prover actually expands this definition file to

```
member X (cons X L).
member X (cons Y L) := member X L.

distinct nil.
distinct (cons X L) := (member X L => false), distinct L.
```

That is, the include statement is replaced by the definition clauses of the "member.def". The parser also checks for circular and duplicate includes. In particular, an include statement that refers to its own definition file is ignored by the parser, and in case where there are two or more occurrences of the same include statement, the parser evaluates the first occurrence of the include statement and ignore the other similar statements.

4 Eigenvariables, logic variables and ∇

We note in the previous sections that the additional quantifier ∇ gives rise to a 'new' type of variable during proof search (∇-variable). We recall that ∇-variables are not instantiated during proof search, eigenvariables can be instantiated only when doing case analysis (e.g., in hypothetical reasoning), and logic variables can

be instantiated with both eigenvariables or ∇-variables. In addition to this, one has also to take into account the scoping of these variables, which we explained briefly in Section 2 using timestamps. One might imagine that in the most general case, we could have proof search involving unification with these three kinds of variables. Such feature is certainly desirable but the precise nature of this interaction is not well-understood yet. For instance, in proof search for the goal

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

where p is defined as $\{pa, pb, pc\}$, 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 of unification, instead of success. This type of problem is generally referred to *disunification* problem, 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. One can imagine further complication when the problem also involves ∇-variables. For this reason, we currently restrict the unification problem to the following: in unification problem where eigenvariables are to be instantiated (i.e., when doing case analysis), there must not be any occurrence of logic variables. This hence explains our restriction on the hypothetical goal $G \supset D$, where G does not have any occurrences of logic variables.

The following goals highlight the differences between eigenvariables, logic variables, and ∇-variables.

```
?- 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.
```

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 y. 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) above) the unification succeeds. In the last goal, we are trying to prove implicational goal with logic variables, and the system returns an exception.

Comparison with λProlog Setting aside the ∇ quantifier, one might think that the proof search behavior for $\forall$ and $\supset$ connectives in Linc 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 plays 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.

A logical specification of 'bagof' Prolog's non-logical construct 'bagof' can be approximated using standard logical connectives, making use of the proof search behavior of eigenvariables in Level 0/1 prover. It is an approximation in the sense that we are interested only in one aspect of 'bagof', that is, to collect all closed terms that are true of some (ground) predicate. Here we restrict ourselves to unary predicate. To be precise, consider a predicate p, and the predicate 'bagofp L' which is true if for every term 't' in 'L', 'p t' holds. This predicate is defined as follows. The idea is to first generate a list 'L' which *satisfies* 'p', that is, all its elements are true of 'p', and then make sure that this list includes all elements that are true of 'p'. The former is achieved by the predicate 'ground' in the definition below.

```
ground L nil.
ground L R :=
   sigma X\ p X, if (member X L) (R = L)
                     (ground (cons X L) R1, R = (cons X R1)).
bagofp L :=
   ground nil L,
   pi x\ p x => member x L.
```

Suppose 'p' is defined to be true on 'a' or 'b', i.e.,

```
p a.
p b.
```

then a call to bagofp L will bind L to a list consisting of 'a' and 'b':

```
?- bagofp L.
Yes
L = (cons a (cons b nil))
Find another? [y/n] y
Yes
L = (cons b (cons a nil))
```

Notice that there are more than one answer substitution for 'L' but they are all equal modulo permutation of the elements of the lists. In general, if there are n closed terms that are true of 'p', 'bagof L' will give at least $n!$ answers, not taking into account the non-determinacy introduced by the predicates 'ground' and 'member'.

5 Parsing and "unparsing" of λ-tree syntax

Level 0/1 prover has a built-in support for manipulating λ-*tree syntax*, which is basically simply typed λ-terms. This allows the prover to read a term and produce its λ-tree representation and vice versa. This might sound a bit confusing at first reading: we use λ-terms to represent λ-terms itself. But this idea is not new; it is used for example in the *higher-order abstract syntax* encoding technique, or in the encoding of untyped λ-calculus in simply typed λ-calculus. The basic constructors of λ-tree syntax are 'var' to encode variables, 'const' to encode constants, 'abs' to encode abstractions, and 'app' to encode application. For instance, the λ-term

$$\lambda x \lambda y.M\ a\ x\ y$$

is encoded as

$$\text{abs}\ (\lambda x\ \text{abs}(\lambda y\ (\text{app}\ (\text{app}\ (\text{app}\ M\ (\text{const}\ a))\ x)\ y).$$

Notice that abs takes an abstraction as argument.

Now, since a definition itself is represented as λ-term, it can be represented as terms in λ-tree syntax as well. Built-in logical connectives and some non-logical constants have the following counterparts in λ-tree syntax:

$\wedge$	symAnd	$\vee$	symOr	$\supset$	symImp
$\top$	symTrue	$\bot$	symFalse	∇	symNabla
$\exists$	symSigma	$\forall$	symPi	$=$	symEq
if	symIf	print	symPrint	input	symInput
$\triangleq$	def				

A definition clause $H \triangleq B$ will be represented in λ-tree syntax as the term

$$\text{abs}\ \lambda x_1\ \ldots \text{abs}\ \lambda x_n\ (\text{app}\ (\text{app}\ \text{def}\ H')\ B')$$

where H' and B' are the representations of H and B, respectively, and $x_1, \ldots, x_n$ are the free variables in the definition clause. Note that in λ-tree syntax, the formulas with binary connectives are represented using prefix notation, e.g., the formula $A \wedge B$ is represented as 'app (app symAnd A) B'.

Level 0/1 prover provides an interpreted predicate call 'parse' to read a definition file and produce a list of λ-tree terms representing the definition clauses in the file. For example, suppose we have a file 'sum.def' which contains the following definition.

```
% file: sum.def

sum z N N.
sum (s M) N (s L) := sum M N L.

sum1 M N := sigma P\ sum M N P, sum N M P.
```

Then, running the 'parse' on 'sum.def.' gives us:

```
?- parse "sum.def" L.

Yes
L = (cons (abs x1\(app (app def (app (app (app (const sum) (const z)) x1) x1))
           symTrue))
    (cons (abs x1\(abs x2\(abs x3\
           (app (app def (app (app (app (const sum) (app (const s) x3)) x2)
           (app (const s) x1))) (app (app (app (const sum) x3) x2) x1)))))
    (cons (abs x1\(abs x2\
           (app (app def (app (app (const sum1) x2) x1))
           (app symSigma (abs x3\(app (app symAnd
               (app (app (app (const sum) x2) x1) x3))
               (app (app (app (const sum) x1) x2) x3)))))))
     nil)))

Find another? [y/n] y
No.
```

The command 'unparse' does the converse, i.e.,

```
unparse "test.def" L
```

converts the λ-tree syntax in L into actual definition clauses and write them to the file 'test.def'. These two commands allow us to perform analyses on a definition files, e.g., type checking (see Section 6), stratification checking, transformation of definitions, etc.

6 Type-checking definitions

We have mentioned earlier that the Level 0/1 prover has currently no built-in support for type checking. However, using the facilities of parsing and unparsing of λ-terms described in the previous section, we can parse a definition and analyze its λ-tree syntax for type checking. We give an example here how to check whether a definition can be typed in Church's simply typed system.

The typing judgments of Church's system are expressions of the form

$$\Sigma; \Gamma \vdash t : \tau$$

where Σ is a finite set of constants along with their types, Γ is a finite set of pairs of variable with their types and t is a term of type τ. We denote with $x : \alpha$ the pair of variable (or constant) x with its type α. The set Γ and Σ are sets of pairs like $\{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}$, etc. The set Σ together with Γ is called the *context* of

```
member X (cons X L).
member X (cons Y L) := member X L.

typeof L symAnd (ar o (ar o o)).
typeof L symTrue o.
typeof L symSigma (ar (ar T o) o).
typeof L def (ar o (ar o clause)).

typeof L (const z) nat.
typeof L (const s) (ar nat nat).
typeof L (const sum) (ar nat (ar nat (ar nat o))).
typeof L (const sum1) (ar nat (ar nat o)).
typeof L (var X) T := member (pr X T) L.
typeof L (app M N) B := typeof L M (ar A B), typeof L N A.
typeof L (abs M) (ar A B) :=
   pi x\ typeof (cons (pr x A) L) (M (var x)) B.

wellTypedClause L (abs M) :=
   pi x\ wellTypedClause (cons (pr x T) L) (M (var x)).
wellTypedClause L (app M N) :=
   typeof L (app M N) clause.

wellTypedDef nil.
wellTypedDef (cons C Rest) := wellTypedClause nil C, wellTypedDef Rest.

test1 := parse "sum.def" L, wellTypedDef L.

test2 := parse "sum.def" L, wellTypedDef L, unparse "sum1.def" L.

test3 := test2, parse "sum1.def" L, wellTypedDef L.
```

Figure 1: Church's simply typed system

the typing judgment. The type expressions are built from *base types* (i.e., type constants of zero arity) and the binary type constructor $\rightarrow$ (arrow). Church's simply type system is given by the following operational semantics.

$$\frac{}{\Sigma;\Gamma \vdash x : \alpha}\ (x:\alpha) \in \Sigma \cup \Gamma \qquad \frac{\Sigma;\Gamma, x:\alpha \vdash M : \beta}{\Sigma;\Gamma \vdash \lambda x.M : \alpha \rightarrow \beta}\ (x:\alpha) \notin \Sigma \cup \Gamma$$

$$\frac{\Sigma;\Gamma \vdash M : \alpha \rightarrow \beta \quad \Sigma;\Gamma \vdash N : \alpha}{\Sigma;\Gamma \vdash (MN) : \beta}$$

The typing judgment $\Sigma;\Gamma \vdash t : \tau$ is encoded using the predicate 'typeof', which takes three arguments: a list of pairs of typed variables, a term and a type expression. The list is the encoding of the type context. However, we put in the list only the typed variables; the types for the constants are defined as explicit clauses. In this example, we consider checking the typeability of the definition 'sum.def' given in the previous section. We first need to introduce the base types and the types for each constants in the definition. The base types are the type 'o', for denoting propositions, 'clause' for denoting definition clause, and 'nat' for denoting natural numbers. The type constructor $\rightarrow$ is encoded as the constant 'ar'. The pair $x : \tau$ is encoded via the constant 'pr', e.g., $(pr\ x\ \tau)$. The types for the constants are introduced explicitly via definition clauses such as

```
typeof L symAnd (ar o (ar o o)).
typeof L (const z) nat.
```

The complete definition for the simply typed system is shown in Figure 1.

Recall that free variables in a definition clause are abstracted, so in order to do type checking on a definition clause we need to first instantiate this abstraction with concrete variables and then assign types to these variables. This is done in the clause 'wellTypedClause' in Figure 1. Here, we are using eigenvariables (via the universal quantifier) to introduce these new variables. The types for these variables are given as logic variables, which would be instantiated with approriate types if the definition is typeable. However, there could be cases where these type variables are not fully instantiated, so in a sense the definition clauses in Figure 1 actually encodes a sort of type assignment system.

The type inference system is then applied to actual definition file (in this case, it is 'sum.def'), using the 'parse' predicate to read the definition file into λ-tree syntax. The predicate 'test1', 'test2' and 'test3' test the parsing and unparsing commands, and show that these operations preserve typeability.

Note that this technique of implementing type checking is not limited to simply typed system; it can be applied to, for instance, dependent type checking as well.

7 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 (or probably the more interesting case, unreachability) 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
```

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 2. 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 both processes at certain points of the code. Let $pc0$ and $pc1$

```
    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;
       }
    }
```

Figure 2: Peterson's algorithm

```
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).

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).

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

Figure 3: Verifying the correctness of Peterson's algorithm: a declarative approach.

```
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).

member X (cons X L).
member X (cons Y L) := member X L.

ground L P nil.
ground L P M :=
   next P X,
   if (member X L) (M = L) (ground (cons X L) P M).

getnext P L := ground nil P L, pi x\ next P x => member x 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)
   ( % else
      union Visited Frontier Visited1,
      newfrontier Visited1 Frontier Frontier1,
      notreach Visited1 Frontier1 Q
   ).

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

Figure 4: Verifying the correctness of Peterson's algorithm: breath-first search

$$\frac{}{\tau.\mathtt{P} \xrightarrow{\tau} \mathtt{P}}\ \text{TAU} \qquad \frac{}{\bar{x}y.\mathtt{P} \xrightarrow{\bar{x}y} \mathtt{P}}\ \text{OUT} \qquad \frac{\mathtt{P} \xrightarrow{\alpha} \mathtt{P}'}{[x=x]\mathtt{P} \xrightarrow{\alpha} \mathtt{P}'}\ \text{MAT}$$

$$\frac{\mathtt{P} \xrightarrow{\alpha} \mathtt{P}'}{\mathtt{P}+\mathtt{Q} \xrightarrow{\alpha} \mathtt{P}'}\ \text{SUM} \qquad \frac{\mathtt{P} \xrightarrow{\alpha} \mathtt{P}'}{\mathtt{P} \mid \mathtt{Q} \xrightarrow{\alpha} \mathtt{P}' \mid \mathtt{Q}}\ \text{PAR}, \mathrm{bn}(\alpha) \cap \mathrm{fn}(\mathtt{Q}) = \emptyset$$

$$\frac{\mathtt{P} \xrightarrow{\alpha} \mathtt{P}'}{(y)\mathtt{P} \xrightarrow{\alpha} (y)\mathtt{P}'}\ \text{RES}, y \notin \mathrm{n}(\alpha) \qquad \frac{\mathtt{P} \xrightarrow{\bar{x}y} \mathtt{P}'}{(y)\mathtt{P} \xrightarrow{\bar{x}(w)} \mathtt{P}'[w/y]}\ \text{OPEN}, y \neq x, w \notin \mathrm{fn}((y)\mathtt{P}')$$

$$\frac{}{x(y).\mathtt{P} \xrightarrow{x(w)} \mathtt{P}[w/y]}\ \text{INP}-\text{L}, w \notin \mathrm{fn}((y)\mathtt{P})$$

$$\frac{\mathtt{P} \xrightarrow{\bar{x}y} \mathtt{P}' \quad \mathtt{Q} \xrightarrow{x(z)} \mathtt{Q}'}{\mathtt{P} \mid \mathtt{Q} \xrightarrow{\tau} \mathtt{P}' \mid \mathtt{Q}'[y/z]}\ \text{COMM}-\text{L} \qquad \frac{\mathtt{P} \xrightarrow{\bar{x}(w)} \mathtt{P}' \quad \mathtt{Q} \xrightarrow{x(w)} \mathtt{Q}'}{\mathtt{P} \mid \mathtt{Q} \xrightarrow{\tau} (w)(\mathtt{P}' \mid \mathtt{Q}')}\ \text{CLOSE}-\text{L}$$

Figure 5: The late transition rules for π-calculus

be the program counters of $p0$ and $p1$, respectively. The (abstracted) state of the entire system can be represented as the the tuple

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

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'). The state that violates the mutual exclusion property is when 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 3, 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 4 where we use a breath-first-search approach in exploring the state space. This "definition" runs in polynomial time. Here we use a method similar to 'bagof' (see Section 4) for enumerating all possible next states from a given state, that is, the predicate 'ground' and 'getnext' in the figure.

8 Example: π-calculus

We now turn to a different description of transition system. Instead of giving a particular name to each state of the system, we use a language to describe the states and the actions of the system, and use *structured operation semantics* to describe the transition relations between the states described in the language. In this section we consider implementing the specification of the language π-calculus [6]. Terms in this language are called processes, each of which represents a collection of states. This section reviews the encoding of the specifications of one-step transitions and a notion of equivalence between processes, called *open bisimulation* [10], for π-calculus that are given in [14, 13] (see those references for details concerning the soundness and adequacy of the encoding, and other theoretical issues) and implements those specifications directly in Level 0/1 prover.

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

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

$$\frac{}{\tau\ P \xrightarrow{\tau} P}\tau \qquad \frac{}{out\ x\ y\ P \xrightarrow{\uparrow xy} P}\ \text{out} \qquad \frac{P \xrightarrow{A} Q}{match\ x\ x\ P \xrightarrow{A} Q}\text{match} \qquad \frac{P \xrightarrow{A} Q}{match\ x\ x\ P \xrightarrow{A} Q}\text{match}$$

$$\frac{P \xrightarrow{A} R}{P + Q \xrightarrow{A} R}\text{sum} \qquad \frac{Q \xrightarrow{A} R}{P + Q \xrightarrow{A} R}\text{sum} \qquad \frac{P \xrightarrow{A} R}{P + Q \xrightarrow{A} R}\text{sum} \qquad \frac{Q \xrightarrow{A} R}{P + Q \xrightarrow{A} R}\text{sum}$$

$$\frac{P \xrightarrow{A} P'}{P \mid Q \xrightarrow{A} P' \mid Q}\text{par} \qquad \frac{Q \xrightarrow{A} Q'}{P \mid Q \xrightarrow{A} P \mid Q'}\text{par} \qquad \frac{P \xrightarrow{A} M}{P \mid Q \xrightarrow{A} \lambda n(Mn \mid Q)}\text{par} \qquad \frac{Q \xrightarrow{A} N}{P \mid Q \xrightarrow{A} \lambda n(P \mid Nn)}\text{par}$$

$$\frac{\nabla n(Pn \xrightarrow{A} P'n)}{\nu n.Pn \xrightarrow{A} \nu n.P'n}\text{res} \qquad \frac{\nabla n(Pn \xrightarrow{A} P'n)}{\nu n.Pn \xrightarrow{A} \lambda m\ \nu n.(P'nm)}\text{res} \qquad \frac{\nabla y(My \xrightarrow{\uparrow xy} M'y)}{\nu y.My \xrightarrow{\uparrow x} M'}\ \text{open}$$

$$\frac{}{in\ x\ M \xrightarrow{\downarrow x} M}\ \text{in-l} \qquad \frac{P \xrightarrow{\downarrow x} M \qquad Q \xrightarrow{\uparrow xy} Q'}{P \mid Q \xrightarrow{\tau} (My) \mid Q'}\ \text{com-l} \qquad \frac{P \xrightarrow{\uparrow xy} P' \qquad Q \xrightarrow{\downarrow x} N}{P \mid Q \xrightarrow{\tau} P' \mid (Ny)}\ \text{com-l}$$

$$\frac{P \xrightarrow{\downarrow x} M \qquad Q \xrightarrow{\uparrow x} N}{P \mid Q \xrightarrow{\tau} \nu n.(Mn \mid Nn)}\ \text{close-l} \qquad \frac{P \xrightarrow{\uparrow x} M \qquad Q \xrightarrow{\downarrow x} N}{P \mid Q \xrightarrow{\tau} \nu n.(Mn \mid Nn)}\ \text{close-l}$$

Figure 6: The late transition rules for π-calculus in λ-tree syntax.

We use the notation $\mathtt{P}$, $\mathtt{Q}$, $\mathtt{R}$, $\mathtt{S}$ and $\mathtt{T}$ to denote processes. Names are denoted by lower case letters, e.g., a, b, c, d, x, y, z. The occurrence of y in the process $x(y).\mathtt{P}$ and $(y)\mathtt{P}$ is a binding occurrence, with $\mathtt{P}$ as its scope. The set of free names in $\mathtt{P}$ is denoted by $\mathrm{fn}(\mathtt{P})$, the set of bound names is denoted by $\mathrm{bn}(\mathtt{P})$. We write $\mathrm{n}(\mathtt{P})$ for the set $\mathrm{fn}(\mathtt{P}) \cup \mathrm{bn}(\mathtt{P})$. We consider processes to be syntactical equivalent up to renaming of bound names. The operator $+$ denotes the choice operator: a process $P + Q$ can behave either like P or Q. The operator $\mid$ 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 of the strength of π-calculus is that it allows *scope extrusion* of names, and hence it captures certain aspects of mobility of processes.

One-step transition in the π-calculus is denoted by $\mathtt{P} \xrightarrow{\alpha} \mathtt{Q}$, where $\mathtt{P}$ and $\mathtt{Q}$ are processes and α is an action. The kinds of actions are *the silent action* τ, *the free input action* xy, *the free output action* $\bar{x}y$, *the bound input action* $x(y)$ and *the bound output action* $\bar{x}(y)$. The name y in $x(y)$ and $\bar{x}(y)$ is a binding occurrence. Just like we did with processes, we use $\mathrm{fn}(\alpha)$, $\mathrm{bn}(\alpha)$ and $\mathrm{n}(\alpha)$ to denote free names, bound names, and names in α. An action without binding occurrences of names is a *free action*, otherwise it is a *bound action*. The original transition rules for finite late π-calculus is given in Figure 5.

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. We do not assume any inhabitants of type n, therefore in our encoding a free name is translated to a variable of type n, which can later be either universally quantified or ∇-quantified, depending on whether we want to treat a certain name as instantiable or not. In the case of open bisimulation they are universally quantified variables. To encode actions, we use $\tau : a$ (for the silent action), and the two constants $\downarrow$ and $\uparrow$, both of type $n \to n \to a$ for building input and output actions. The free output action $\bar{x}y$, is encoded as $\uparrow xy$ while the bound output action $\bar{x}(y)$ is encoded as $\lambda y\ (\uparrow xy)$ (or the η-equivalent term $\uparrow x$). The free input action xy, is encoded as $\downarrow xy$ while the bound input action $x(y)$ is encoded as $\lambda y\ (\downarrow xy)$ (or simply $\downarrow x$). The process constructors are encoded using the following constants

$$0 : p, \qquad \tau : p \to p, \qquad out : n \to n \to p \to p, \qquad in : n \to (n \to p) \to p,$$
$$+ : p \to p \to p, \quad \mid : p \to p \to p, \quad match : n \to n \to p \to p, \quad \nu : (n \to p) \to p.$$

```
% 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)).
```

Figure 7: 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 (plus (in x y\out x a) (out x a (in x y\z))).
example 3 (in x u\ (plus (taup (taup z)) (taup z))) := true.
example 4 (in x u\ (plus (taup (taup z)) (plus (taup z) (taup (match u y (taup z)))))).
example 5 (nu a\ (par (in x y\z) (out x a z))).
example 6 (nu a\ (plus (in x y\out x a) (out x a (in x y\z)))).
example 7 (taup z).
example 8 (nu x\ (par (in x y\z) (out x a z))).
example 9 (nu x\ out a x z).
example 10 (par (in x y\ z) (nu y\ out x y z)).
example 11 (in x u\ nu y\ ((plus (taup (taup z))
           (plus (taup z) (taup (match u y (taup z)))))))).
```

Figure 8: Some example processes

We use two predicates to encode the one-step transition semantics for the π-calculus. The predicate $\cdot \xrightarrow{\cdot} \cdot$ of type $p \to a \to p \to o$ encodes transitions involving free values and the predicate $\cdot \xrightarrow{\cdot} \cdot$ of type $p \to (n \to a) \to (n \to p) \to o$ encodes transitions involving bound values. The precise translation of π-calculus syntax into simply typed λ-terms is given in the following. We define a function $\langle . \rangle$ which translates process expressions to $\beta\eta$-long normal terms of type p.

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

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

$$\langle \mathtt{P} \xrightarrow{\bar{x}y} \mathtt{Q} \rangle = \langle \mathtt{P} \rangle \xrightarrow{\uparrow xy} \langle \mathtt{Q} \rangle \qquad \langle \mathtt{P} \xrightarrow{\tau} \mathtt{Q} \rangle = \langle \mathtt{P} \rangle \xrightarrow{\tau} \langle \mathtt{Q} \rangle$$
$$\langle \mathtt{P} \xrightarrow{x(y)} \mathtt{Q} \rangle = \langle \mathtt{P} \rangle \xrightarrow{\downarrow x} \lambda y.\langle \mathtt{Q} \rangle \qquad \langle \mathtt{P} \xrightarrow{\bar{x}(y)} \mathtt{Q} \rangle = \langle \mathtt{P} \rangle \xrightarrow{\uparrow x} \lambda y.\langle \mathtt{Q} \rangle$$

The transition rules for late π-calculus in λ-tree syntax is given in Figure 6. Turning this abstract specification of π-calculus into the concrete syntax of Level 0/1 prover is straightforward. For the moment we do not enforce the typing of the specification, hence we do not need to encode directly the base types for names, processes, actions, etc. Type-checking of the following definition clauses can be done using the technique described in Section 5. The constants in the specification are given the following concrete syntax:

$. \xrightarrow{.} .$	one	$. \xrightarrow{.} .$	onep
0	z	τ	tau
out	out	*in*	in
$+$	plus	$\mid$	par
match	match	ν	nu
$\uparrow .$	up	$\downarrow .$	dn

The definition of one-step transition in Level 0/1 prover is given in Figure 7.

We consider some simple examples involving one-step transitions. Figure 8 gives some example processes. The process in example 1 is the encoding of the following

$$x(y).0 \mid \bar{x}a.0.$$

This process can make a bound input transition ($x(y)$), or an output ($\bar{x}a$), or a τ-transition, i.e., as a result of the synchronization between its components. We can query the Level 0/1 prover, given the definition of one-step in Figure 7, to show all possible transitions this process can make:

```
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)).
```

Figure 9: Definition of open bisimulation

```
?- example 1 P, (one P A Q; onep P A Q).
Yes
P = (par (in x x1\z) (out x a z))
A = (up x a)
Q = (par (in x x1\z) z)

Find another? [y/n] y
Yes
P = (par (in x x1\z) (out x a z))
A = tau
Q = (par z z)

Find another? [y/n] y
Yes
P = (par (in x x1\z) (out x a z))
A = (dn x)
Q = x1\(par z (out x a z))

Find another? [y/n] y
No.
```

Note that we need to specify the query in disjunctive form since we distinguish between bound-action transition relation and free-action transition.

Another interesting query is to check whether some process makes no transition. Consider example 0 in Figure 8. It corresponds to the following process

$$\nu x\ [x = a]\tau.0$$

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 in the following query.

```
?- example 0 P, (pi A\pi Q\ one P A Q => false), (pi A\pi Q\onep P A Q => false).
Yes
P = (nu x1\(match x1 a (taup z)))

Find another? [y/n] y
No.
```

Bisimulation 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 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 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 9. 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 bisimPQ is provable then P and Q are late-bisimilar. The encoding also turns out to be equivalent to *open bisimulation* [9], a finer bisimulation relation then late bisimulation (see [14] for details of the encoding and adequacy results).

Let us consider a few examples of checking bisimulation using the definition in Figure 9. The following processes

$$P = \tau.0, \quad Q = \nu x\ (x(y).0 \mid \bar{x}a.0),$$

which correspond to example 7 and 8, respectively, are late bisimilar since the only actions that both process can perform is the action τ. Notice that in Q the scope of x, the input and output channel in P and Q, is restricted within the two subprocesses so it cannot be used to interact with any other processes. Therefore the only action that Q can perform is the internal synchronization (τ). We therefore would expect the outcome of the following query.

```
?- example 7 P, example 8 Q, bisim P Q.
Yes
P = (taup z)
Q = (nu x1\(par (in x1 x2\z) (out x1 a z)))

Find another? [y/n] y
No.
```

We mentioned earlier that the definition in Figure 9 does not capture fully the notion of late bisimulation. Here is a counterexample that shows.

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

This happens to be an example that separates open and late bisimulation [9].

```
?- example 3 P, example 4 Q, bisim P Q.
No.
```

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 we are considering would be

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

These processes correspond to example 3 and 11 in Figure 8. They can be proved bisimilar.

```
?- example 3 P, example 11 Q, bisim P Q.
Yes
P = (in x x1\(plus (taup (taup z)) (taup z)))
Q = (in x x1\(nu x2\(plus (taup (taup z)) (plus (taup z) (taup (match x1 x2 (taup z)))))))

Find another? [y/n] n
No.
```

9 Future work

The current prover implements a fairly restricted fragment of the logic Linc. We consider extending it to richer fragment to include features like, among others, induction and co-induction proof rules (see, e.g.,[13]) 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 studided in, e.g., [12]. This extended feature would allow us, for example, to reason about bisimulation of non-terminating processes. We also consider specifying a tactical language which would allow users to programme their own proof search strategy. 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* [7] 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. Interesting examples include spi-calculus, fusion calculus, *symbolic bisimulation*, barbed bisimulation, etc.

It is also possible to isolate certain features of the implementations of these extended provers, which need not be logical, which can be used to implement application specific (automated) proof checkers. For example, features such as the use of a uniform abstract syntax (i.e., λ-tree syntax) and higher-order pattern unification, the proper treatment of variables and binders, stream-based computation, etc.

Acknowledgments

The Level 0/1 Prover was implemented by the author, based on an implementation of Prolog in SML/NJ by Dale Miller (INRIA Futurs/École polytechnique). The author wishes to thank Gopalan Nadathur and Natalie Linnell (Department of Computer Science and Engineering, University of Minnesota) for providing their source codes of higher-order pattern unification used in the prover.

References

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

[2] J.-Y. Girard. A fixpoint theorem in linear logic. Email to the linear@cs.stanford.edu mailing list, February 1992.

[3] 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.

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

[5] 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.

[6] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Part II. *Information and Computation*, pages 41–77, 1992.

[7] A. Momigliano and F. Pfenning. Higher-order pattern complement and the strict u03bb-calculus. *ACM Trans. Comput. Logic*, 4(4):493–529, 2003.

[8] A. Momigliano and A. Tiu. Induction and co-induction in sequent calculus. In *Proceedings of TYPES 2003 Workshop*, volume 3085 of *LNCS*, pages 293 – 308. Springer, 2003.

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

[10] D. Sangiorgi and D. Walker. *π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[11] P. Schroeder-Heister. Cut-elimination in logics with definitional reflection. In D. Pearce and H. Wansing, editors, *Nonclassical Logics and Information Processing*, volume 619 of *LNCS*, pages 146–171. Springer, 1992.

[12] 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.

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

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