Implementing a subset of Lambda Prolog in HyperLMNtal

Alimujiang Yasen, Kazunori Ueda

λProlog (Lambda Prolog) is a logic programming language that extends Prolog by incorporating λ-terms and higher-order unification. Putting restrictions on variables in λProlog results in a subset called L_λ. L_λ uses pattern unification and performs a simple version of beta-conversion, while retaining the essence of higher-order unification. This paper describes an implementation of L_λ using HyperLMNtal, a language model based on hierarchical hypergraph rewriting. In this encoding, logic variables and constants are represented by hyperlinks, and proof procedure is modeled with the help of membranes, rule contexts and process contexts. We describe the usage of HyperLMNtal's features, discuss our observation from this encoding and put forward ideas of future improvements of HyperLMNtal.

1 Introduction

The logic programming language λProlog [1] [5] is an extension of Prolog. λProlog supports higher-order logic programming by incorporating λ-terms and higher-order unification. λProlog uses a class of formulas that are called *higher-order hereditary Harrop* (*hohh*) formulas and new features are provided by exploiting the extensions from *first-order horn clauses* (*fohc*) to *hohh*.

LMNtal (pronounced "elemental") is a language model based on hierarchical graph rewriting. Recent years, by incorporating *hyperlink*, LMNtal was extended to HyperLMNtal. Although LMNtal has been modeled many systems successfully [3], the true strength of HyperLMNtal has not been tested.

HyperLMNtal and λProlog are quite different. The main constructs of HyperLMNtal are, unlike many other programming languages, *atoms*, *links* and *membranes* to compose hierarchical graphs and *rewrite rules* to rewrite the graphs. λProlog is a *Logic Programming Language*, which views computation as a proof search. The main constructs are logical formulas. When a *set of formulas* and a *query* are given, it uses *inference rules* to prove the query.

Previous work that used the new features of HyperLMNtal was the encoding of type systems with let polymorphism [8]. As the next stage, we wanted to encode a more complex formal system and λProlog has attracted our attention. We challenge a sophisticated system, L_λ, which is a sublanguage of λProlog. Modeling it, we believe, is complex enough to demonstrate the power of HyperLMNtal. From such experiences, we expect to develop modeling techniques, improve on language implementation, and acquire new ideas for future research.

The newest and most efficient implementation of λProlog is *Teyjus* [7]. The original version of Teyjus implemented higher-order unification of λProlog, but the current version has moved to somewhat simpler *higher-order pattern unification*, which is exactly what we encode in our work.

This paper organized as follows. Section 2 describes *λProlog* briefly. Section 3 describes HyperLMNtal and its features. Section 4 will present the encoding of L_λ in detail. The final section is dedicated to conclusion.

The encoding of L_λ with the example described in this paper is available online at `http://www.ueda.info.waseda.ac.jp/lmntal/demo/hyperlmn/`.

Alimujiang Yasen, Dept. of Computer Science and Engineering, Waseda University.

Kazunori Ueda, Dept. of Computer Science and Engineering, Waseda University.

2 λProlog

λProlog extends Prolog in three points: (i) types, (ii) higher-order terms, and (iii) the syntax of formulas. In this paper, we focus on (ii) and (iii) because they are more relevant to our encoding.

2.1 Terms and formulas

The logical connectives of λProlog, when written in mathematical notation, are $\top$ (truth), $\wedge$ (conjunction), $\vee$ (disjunction), $\supset$ (implication), $\exists$ (existential quantification) and $\forall$ (universal quantification). In λProlog, they are written as follows: the constant $\top$ is written as *true*. The two infix symbols ',' and '&' both denote conjunction, but are used in different situations (we shall see their usage in later examples). The symbol '$\Rightarrow$' denotes implication while ':-' is the converse of implication or 'implied-by'. The semicolon ';' is used to denote disjunction [1].

Quantifiers ∃x and ∀x are written as *pi* $x\backslash$ and *sigma* $x\backslash$ in the concrete syntax. The '\' is used to represent binding operations for both quantification and λ-terms. The term $\lambda x.t$ is written as $x\backslash t$. Curried notation such as $((f\ s)\ t)$ and $(f\ s\ t)$ is used instead of $f(s,t)$. Terms of λProlog can be summarized as follows.

$$t ::= \langle \textit{first-order terms} \rangle \mid x\backslash t \mid t\ t.$$

The following expressions illustrate the concrete syntax for quantified formulas.

```
pi x\ pi k\ memb x (x::k).
pi X\ pi L\ pi K\ pi M\
  append (X::L) K (X::M) :- append L K M.
```

The first one denotes the universal closure of the atomic formula `memb x (x::k)`, while the second one is an example of 'implied-by' formulas. The scope of a bound variable introduced by '\' extends as far to the right as possible, limited only by parentheses. Applying predicate symbols to as many terms as required will produce a term of the propositional type called a *formula*.

2.2 Logic programming and search semantics

Program formulas, *goals or goal formulas* and a *calculus* are essential to describe the semantics of λProlog. The collection of well-formed terms and formulas are called Σ-*terms* and Σ-*formulas*, where Σ stands for a *signature* defining the types of *constants*, which include all nonvariable symbols (including predicates) in this paper. Some of the Σ-formulas used as assertions or axioms are called *program formulas*. Another class of Σ-formulas is *goal formulas*, also called *queries* or *goals*. An attempt will be made to prove goals from a given set of program formulas [1].

The *sequent calculus* is used to construct proofs. In λProlog, a sequent is a triple consisting of a signature Σ, a set P of Σ-formulas, and a Σ-formula G. Such a sequent will be written as $\Sigma; P \longrightarrow G$. The goal formula G will be exposed to the signature-program pair $\langle \Sigma, P \rangle$, which can be proved successfully or fail.

Goal-directed proof search is presented through a collection of rules. Among them, the following are significant.

INSTAN: Reduce $\Sigma; P \longrightarrow \exists_\tau x B$ to $\Sigma; P \longrightarrow B[t/x]$, for some Σ-term t of type τ, the resulting sequent is instantiated by the chosen Σ-term t.

GENERIC: Reduce $\Sigma; P \longrightarrow \forall_\tau x B$ to $c : \tau$, $\Sigma; P \longrightarrow B[c/x]$, where c is a token not in the current signature Σ. c is referred to as a *new constant*.

In λProlog, proof starts with a single goal and a set of program formulas. The rules in Fig. 1 will be applied if the given goal is a non-atomic formula. Thus, it will eventually be reduced to atomic formulas.

In the $\supset R$ rule, the expression P, B_1 denotes the set $P \cup \{B_1\}$ of formulas, and in the $\forall R$ rule, the expression $c : \tau, \Sigma$ denotes the set $\{c : \tau\} \cup \Sigma$ of type declarations. Also, the judgment $\Sigma; \emptyset \Vdash_f t : \tau$ in the $\exists R$ rule is to formalize the requirement that t is a Σ-term of type τ.

When a given goal is an atomic formula or when a non-atomic formula has been reduced to atomic formulas, the rules in Fig. 2 will be used to prove atomic goals.

In the *decide* rule, the sequent $\Sigma; P \xrightarrow{D} A$ indicates that a program formula D is selected from the set of program formulas to derive an atomic goal A. The *initial* rule says that if a selected program formula to derive a goal A is equal to A, then the goal A is proved. The $\supset L$ rule indicates if a selected program

$$\frac{}{\Sigma; P \longrightarrow \top} \quad \top R$$

$$\frac{\Sigma; P \longrightarrow B_1 \qquad \Sigma; P \longrightarrow B_2}{\Sigma; P \longrightarrow B_1 \wedge B_2} \quad \wedge R$$

$$\frac{\Sigma; P \longrightarrow B_1}{\Sigma; P \longrightarrow B_1 \vee B_2} \quad \vee R$$

$$\frac{\Sigma; P \longrightarrow B_2}{\Sigma; P \longrightarrow B_1 \vee B_2} \quad \vee R$$

$$\frac{\Sigma; P, B_1 \longrightarrow B_2}{\Sigma; P \longrightarrow B_1 \supset B_2} \quad \supset R$$

$$\frac{\Sigma; P \longrightarrow B[t/x] \qquad \Sigma; \emptyset \Vdash_f t : \tau}{\Sigma; P \longrightarrow \exists_\tau x B} \quad \exists R$$

$$\frac{c : \tau, \Sigma; P \longrightarrow B[c/x]}{\Sigma; P \longrightarrow \forall_\tau x B} \quad \forall R$$

Fig. 1 Right-introduction rules.

$$\frac{\Sigma; P \xrightarrow{D} A}{\Sigma; P \longrightarrow A} \quad \text{decide}$$

$$\frac{}{\Sigma; P \xrightarrow{A} A} \quad \text{initial}$$

$$\frac{\Sigma; P \xrightarrow{D} A \qquad \Sigma; P \longrightarrow G}{\Sigma; P \xrightarrow{G \supset D} A} \quad \supset L$$

$$\frac{\Sigma; P \xrightarrow{D_1} A}{\Sigma; P \xrightarrow{D_1 \wedge D_2} A} \quad \wedge L$$

$$\frac{\Sigma; P \xrightarrow{D_2} A}{\Sigma; P \xrightarrow{D_1 \wedge D_2} A} \quad \wedge L$$

$$\frac{\Sigma; P \xrightarrow{D[t/x]} A \qquad \Sigma; \emptyset \Vdash_f t : \tau}{\Sigma; P \xrightarrow{\forall_\tau x D} A} \quad \forall L$$

Fig. 2 Rules for backchaining.

formula is a $\supset$ formula, then it will be reduced to two promises as shown.

The two $\wedge L$ rules are about the cases in which a selected program formula is a conjunction. The last rule describes the case when a universally quantified program formula is selected to derive a goal A, in which case it will be instantiated by a chosen Σ-term t and the resulting formula is used to derive A. The τ indicates the type of t; however, types are excluded in our encoding and henceforth we omit types in subsequent definitions.

Goals and program formulas in the setting of *first-order horn clauses* (*fohc*) are given as follows.

$$G ::= \top \mid A \mid G \wedge G \mid G \vee G \mid \exists x G$$
$$D ::= A \mid G \supset D \mid D \wedge D \mid \forall x D$$

Here, A denotes first-order atomic formulas, and goals and program formulas correspond to G and D. In the *fohc* setting, an atomic formula A is not allowed to contain logical constants and all the quantifications are limited to being first order.

In the *hohh*, quantifications are permitted at higher order. In the higher-order setting, the main change is the definition of atomic formulas A. In the *hohh*, these formulas may contain arbitrary occurrences of functions and predicate variables as well as logical symbols [1]. The goal and program formulas of *hohh* are defined by the following syntax.

$$G ::= \top \mid A \mid G \wedge G \mid G \vee G \mid \exists x G \mid D \supset G \mid \forall x G$$
$$D ::= A_r \mid G \supset D \mid D \wedge D \mid \forall x D$$

Here, A indicates atomic formulas and A_r indicates rigid atom(ic formula)s, both in the *Herbrand universe* H_2^Σ of *hohh* defined below. The λ-normal form of a higher-order atomic formula has the form $(h\ t_1\ \ldots\ t_n)$, where h is the head of the formula and $t_1\ \ldots\ t_n$ are terms. The formula is a rigid atom if h is a constant and it is a flexible atom if h is a variable. H_2^Σ is defined to be the set of all well-formed λ-normal terms which do not contain $\supset$, thus, other constants are allowed within atomic formula in the *hohh* setting.

In the *hohh* setting, equality of terms is based on λ-conversion. Therefore, in the *initial* rule, the program for backchaining and the atomic goal are α-convertible formulas. Another change is that, in $\forall L$ and $\exists R$, the term t for instantiation now should be from H_2^Σ.

Besides *fohc* and *hohh*, λProlog has other sub-language such as *first-order hereditary Harrop formulas* (*fohh*) and *higher-order Horn clauses* (*hohc*). These sub-languages are illustrated in Fig. 3. Of these, the L_λ sub-language will be explained in the next section.

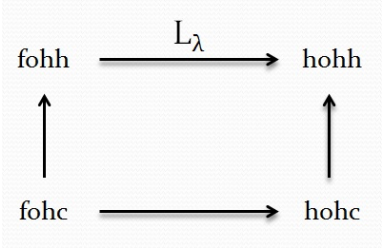

Fig. 3 Sub-languages of λProlog

2.3 L_λ and pattern unification

We need to recall some definitions before describing L_λ and pattern unification.

If a sub-formula A of B occurs on the left of an even number of occurrences of implications in B, A is called a *positive* subformula occurrence. If A occurs on the left of an odd number of occurrences of implication in B, A is called a *negative* subformula occurrence.

A bound variable occurrence in goal formulas is called *essentially universal* if it is bound by a positive occurrence of a universal quantifier, or by a negative occurrence of an existential quantifier, or by a λ-abstraction. It is called *essentially existential* if it is not essentially universal.

A bound variable occurrence in program formulas is classified differently; it is called *essentially universal* if it is bound by a negative occurrence of a universal quantifier, or by a positive occurrence of an existential quantifier, or by a λ-abstraction. It is called *essentially existential* if it is not essentially universal.

The L_λ subset is obtained by putting restriction on *hohh*. First, all quantifications should be over non-predicate types. Second, in each subterm of the shape $x\ t_1\ \ldots\ t_n$, where the head x is an essentially existential quantified variable, the $t_1\ \ldots\ t_n$ must be distinct essentially universally quantified variables bound in the scope of x. As a result of these restrictions, if the head x in a subterm $x\ t_1\ \ldots\ t_n$ is instantiated by a term, the β-conversion, which is performed on the resulting term, always reduces the size of that term [6].

L_λ uses *pattern unification*, an algorithm which can be seen as a generalization of first-order unification and simplification of higher-order unification. Any β-normal term has the shape $\lambda x_1 \ldots \lambda x_p(h\ t_1 \ldots t_q)$ $(p, q \geq 0)$, where the bound variables $x_1 \ldots x_p$ are distinct and $t_1 \ldots t_q$ are also β-normal terms. If the head h is an essentially existential variable, then it is called *flexible*. Otherwise, that term is *rigid*. Therefore, pairs in unification (whose left hand side term and right hand side term are referred to as *lhs* and *rhs*) are classified into one of rigid-rigid, rigid-flexible, flexible-flexible and flexible-rigid. Once a pair of terms is classified, they are unified as follows [1] [5] [6] .

Rigid-rigid. If the rigid-rigid pair has different heads, then there is no unifier. If the pair $\langle c\ s_1 \ldots s_n, c\ t_1 \ldots t_n \rangle$ has the same head, then this pair is replaced by $\langle s_1, t_1 \rangle, \ldots, \langle s_n, t_n \rangle$ or by $\top$ if $n = 0$.

Flexible-flexible. The pair $\langle F\ c_1 \ldots c_n, G\ d_1 \ldots d_m \rangle$, where $c_1 \ldots c_n$ and $d_1 \ldots d_m$ are universally quantified variables within the scope of F and G, which are existentially quantified flexible heads, will be handled according to the two cases.

- **Different head.** If the F and G are different, then the solution will be $[F \mapsto \lambda c_1 \ldots c_n(H\ e_1 \ldots e_\ell)]$ and $[G \mapsto \lambda d_1 \ldots d_m\ (H\ e_1 \ldots e_\ell)]$.
- **Same head.** If the F and G are the same, in order to be unifiable, n and m must be equal. The solution will be $[F \mapsto \lambda c_1 \ldots c_n(H\ e_1 \ldots e_\ell)]$.

Here, H is a new flexible variable and $e_1 \ldots e_\ell$ is a list of variables that are both in the lists $c_1 \ldots c_n$ and $d_1 \ldots d_m$.

Flexible-rigid. For the pair $\langle F\ \ c_1 \ldots c_n, t \rangle$, where F is a flexible head of *lhs* and t is a *rhs* rigid term, perform the following steps.

1. Look for the occurrences of a flexible variable different from F in t. Such a flexible variable appears as the head of a term of the form $G\ d_1 \ldots d_m$. If the $d_1 \ldots d_m$ is not a subset of $c_1 \ldots c_n$, then perform variable elimination to prune variables of $d_1 \ldots d_m$ not appearing in $c_1 \ldots c_n$ as follows. Replace G with the term $\lambda d_1 \ldots d_m(G'\ e_1 \ldots e_\ell)$, where G' is a new flexible variable and $e_1 \ldots e_\ell$ are the variables which are common to $c_1 \ldots c_n$ and $d_1 \ldots d_m$. Let t' be the term obtained from t by the above replacement.
2. Check the occurrences of F in t'. If there are occurrences of F, then this problem

has no unifier.

3. In t', look for the occurrences of variables different from $c_1 \dots c_n$ and universally quantified within the scope of the quantifier binding F. If there are occurrences of such variables, then this problem has no unifier.
4. The solution will be $[F \mapsto \lambda c_1 \dots c_n(t')]$.

Rigid-flexible. Switch the order of pair and solve it as flexible-rigid.

Pattern unification is deterministic and generates a unique solution if successful and always terminates. Also, pattern unification does not use type information; therefore, we omitted types in our encoding.

3 HyperLMNtal Overview

LMNtal views that computation is the manipulation of diagrams, which in our case consists of *atoms*, *links* and *membranes* [2]. Atoms and links form node-labelled undirected graphs, where the links (edges) of an atom (vertex) are totally ordered. Membranes can enclose atoms and other membranes, and can be crossed by links. There are two kinds of links; the first is for one-to-one connectivity and the second is called *hyperlinks* that have multiple connections [4].

Figure 3 shows the simplified syntax of LMNtal, where the two syntactic categories, *link names* (denoted by X_i) and *atom names* (denoted by p) are presupposed. *Processes* are the principal syntactic category and consist of hierarchical graphs and rewrite rules: 0 is an inert process; $p(X_1, \dots, X_m)$ is an atom with arity m; P, P is parallel composition; $\{P\}$ is a *cell* formed by enclosing P by a *membrane* $\{\}$; and T `:-` T is a rewrite rule. The two T's are called the *head* and the *body* of the rule, respectively. A rewrite rule is subject to several syntactic conditions. Most notably, a link name occurring in a rule must occur exactly twice in the rule.

The reserved atom name, `=`, is called a `connector`. The process X`=`Y connects the other end of the link X and the other end of the link Y. A *rule context*, denoted by `@`p, matches a (possibly empty) multiset of all rules within a membrane, while a *process context*, denoted by `$`p, is to match processes other than rules within a membrane.

An abbreviation called a *term notation* is frequently used during LMNtal programming. It allows an atom b without its final argument to occur as the kth argument of a. For instance, `f(a,b)` is the same as `a(A),f(A,B),b(B)`. A list with elements A_i's can be written as X=$[A_1, \dots, A_n]$, where X is the link to the whole list. Some atoms such as '`+`' are written as unary or binary operators [2].

Below we describe some features of *HyperLMNtal* with examples.

Example 1

```
mem{init.  init :- a(X), b(X), tag.}
mem{@m, $m, tag}/ :- mem{@m, $m}.
```

In the first line, a membrane `mem` is created with an atom `init` and a rule. The second line is a rule, which will be executed when the following conditions are satisfied. First, `mem` should contain an atom `tag`. Second, `mem` should satisfy a *stability* constraint: a cell suffixed by a '`/`' in the head of a rule only matches a cell that cannot be reduced further. When `mem` is created, `init` can be reduced to other three atoms, but after that there is no more rule to be executed and `mem` will satisfy the stability constraint. The `@`m rule context will match the rules within `mem`.

Hyperlinks connect multiple points and come with some features.

Example 2

```
H :- new($x)     | B.
H :- new($x,1)   | B.
H :- hlink($x)   | B.
H :- hlink($x,1) | B.
H :- $x == $y    | B.
H :- $x \= $y    | B.
H :- ...         | $x >< $y, B.
```

A hyperlink with a fresh name can be created by a `new` guard constraint (first line). An attribute can be assigned to a hyperlink. The second line shows a hyperlink which is created with an integer attribute 1. To check if an argument of an atom is a hyperlink, the `hlink` constraint is used (third line). `hlink` also can be used to check whether a hyperlink has a particular attribute (fourth line). Equality checking can be performed between two hyperlinks (fifth and sixth lines). *Fusion* maybe the most characteristic operation on hyperlinks (seventh line), in

(Process) $P ::= 0 \mid p(X_1, \ldots, X_m) \mid P, P \mid \{P\} \mid T :- T$
(Process template) $T ::= 0 \mid p(X_1, \ldots, X_m) \mid T, T \mid \{T\} \mid T :- T \mid @p \mid \$p$

Fig. 4 Syntax of LMNtal.

which two hyperlinks `$x` and `$y` are merged into one. In this example, all hyperlinks should occur in the head H of the rule, except for the first two lines.

Our HyperLMNtal implementation features nondeterministic execution, in which it executes all possible paths of reduction, and the output can be displayed either in the text form or as a graph. *StateViewer*, a HyperLMNtal visualizer, displays the state space of the program as a directed graph.

HyperLMNtal also provides other useful features. One of them is *atomic ruleset*. Multiple steps of execution using rules in an atomic ruleset is regarded as a single step of execution. StateViewer displays atomic execution as a single transition between two states in a state space graph.

4 Encoding of L_λ

In this section, we take an L_λ program to compute prenex normal forms of formulas as an example.

```
//program clauses
prenex (atom L) (atom L).
prenex (B && C) D :- prenex B U, prenex C V,
                          prenex (U && V) D.
prenex (all B) (all C) :-
                    pi x\ prenex (B x) (D x).
merge((all B) && (all C)) (all D) :-
           pi x\ merge ((B x) &&(C x)) (D x).
merge((all B) && C) (all D) :-
              pi x\ merge ((B x) && C) (D x).
merge(B && (all C)) (all D) :-
                pi x\ merge (B &&(C x)) (D x).
merge(B && C)(B && C) :-
                    quantfree B, quantfree C.
quantfree atom L.
quantfree B && C.

//goal formula
prenex ((all x\ atom q x x) &&
                (all z\ all y atom q z y)) P.
```

Henceforth, by abuse of terminology, we may use the term *(program) clauses* to refer to program formulas.

The proof search will compute five substitutions for `P` in the goal formula. To keep it short, program clauses for the computation of $\vee$, $\supset$ and $\exists$ are not given, since they are similar to the program clauses for computing $\wedge$ (`&&`) and $\forall$ (`all`).

4.1 Syntactic analysis

There are several basic syntactic entities in L_λ, which have different properties from the prospective of implementation.

Constants. Constants such as logical connectives will keep their original names.

Flexible variables. Flexible variables are not only in the given program clauses and the goal, but also might be created during proof search and are required to have fresh names.

Rigid variables. Rigid variables are also required to have fresh names when they are created.

Free variables. Constants whose equality and inequality are checked in unification are treated as free variables, which are considered to have fixed global names.

Those variables that are involved in pattern unification require that the equality checking can be performed on them. Therefore, hyperlinks are used to represent them.

L_λ	*HyperLMNtal*
constant	*unary atom*
flexible variable	*hyperlink with attribute 1*
rigid variable	*hyperlink with attribute 2*
free variable	*hyperlink with attribute 3*

We have two different levels of logical formulas: L_λ clauses themselves consisting of system built-in logical constants, and logical formulas handled by the L_λ clauses consisting of user-defined logical constants.

Firstly, system built-in logical constants are represented as follows.

L_λ	*HyperLMNtal*
$A \Rightarrow B$	imply(A,B)
$A \,\&\, B$	and(A,B)
A, B	and(A,B)
$A; B$	or(A,B)
A :- B	implied(A,B)

These logical constants are represented by atoms.

User-defined logical constants are represented by hyperlinks as free variables.

L_λ	*HyperLMNtal*
$A \implies B$	[Imply, [A,B]], new(Imply,3)
$A \&\& B$	[And, [A,B]], new(And,3)
A !! B	[Or, [A,B]], new(Or,3)
$neg\ A$	[Neg, A], new(Neg,3)

These free variables are treated as unary or binary functions. That is why *lists* are used to represent logical formulas.

The λ-abstractions are represented with binary atoms. The *lambda*-applications are then represented by a list of two elements.

L_λ	*HyperLMNtal*
$\lambda x.t$	lambda(x, t)
$(\lambda x.t)s$	[lambda(x, t), s]

User-defined quantifications are also treated as free variables. For instance, an L_λ term $all\ x\backslash\ atom\ q\ x\ x$ is represented as:

[All, lambda(X, [Atom, [Q,X,X]])],
new(All,3), new(Atom,3),
new(Q,2), new(X,2).

In our representation, variables are expressed using hyperlinks with attributes and the attributes implicitly state the quantification kind of the variable.

L_λ uses pattern unification, and from observation it is realized that explicitly quantified variables are always rigid variables that are universally quantified. These universally quantified rigid variables involve in the flexible-rigid unification where such quantification becomes necessary to check whether or not the universally quantified rigid variables are quantified within the scope of the quantifier which existentially binds the head of flexible terms on the *lhs* of flexible-rigid pair. Scoping only becomes significant at this stage.

To represent scoping, which is due to explicit quantification, such variables are placed in a list. When representing formulas, we wrap the atomic part of a formula with a special atom named `c`. There are two reasons for this. The first is to perform equality checking between unary atoms. The second is that it is convenient for the inference rules to manipulate complex formulas.

```
prenex (all B)(all D):-
                  pi x\ prenex (B x)(D x).
program(implied(
    c(quantifier=[],
      body(prenex,[[All,[B]],[All,[D]]])),
    c(quantifier=[X],
      body(prenex,[[B,X],[D,X]])))).
```

Here, the first formula is an L_λ program, and the second formula is its HyperLMNtal representation. The `quantifier` list stores these explicitly quantified variables. The program clauses are wrapped by the an atom named `program`.

```
prenex ((all x\ atom q x x) &&
            (all z\ all y atom q z y)) P.
goal(quantifier=[],
    body(prenex,
    [[And,[[All,lambda(X,[Atom,[Q,X,X]])],
    [All,lambda(Z,
    [All,lambda(Y,[Atom,[Q,Z,Y]])])]]],[P]])).
```

Here, the first formula is an L_λ goal formula, and the second formula is its HyperLMNtal representation. The goal formulas are wrapped by the an atom named `goal` (the *new* constraints for creating hyperlinks are omitted).

A λ-term, which is in β-normal form and with the structure $(h\ t_1 \ldots\ t_n)$, is either flexible or rigid, depending on h. Such terms are also viewed as functions, and *lists* are used to represent them:

L_λ	*HyperLMNtal*
$(A\ x)$	[A,X], new(A,1), new(X,2)
(A)	[A], new(A,1)
$(B\ x)$	[B,X], new(B,2), new(X,2)
(B)	[B], new(B,2)

Here, the first line is a flexible term with the argument x, and the second line is a flexible term without arguments. The next two lines are the ex-

pressions of rigid terms.

4.2 Semantic analysis

When a set of programs and a single goal are given, if the goal is a non-atomic formula, then *right-introduction* rules (Fig. 1) simplify the goal until it becomes an atomic formula. Then *backchaining* rules (Fig. 2) try to prove the goal which is an atomic formula. Proof search always starts with a single goal and applying inference rules might generate multiple goals.

The following aspects of proof search should be addressed in our encoding.

Non-determinism. The *decide* rule chooses a program clause from a set of program clauses non-deterministically as long as the predicate name of the goal matches with the predicate name of the program clause. Proof search will look like a tree structure, in which a path of the proof tree represents either a failure or a success.

Copying clause. Every time a clause is selected, it needs to be copied. When making copies, flexible and rigid variables should receive new names, while constants and free variables should preserve their original names.

Presence of loops. There are chances that proof search might stick in a loop, which is an endless path of the proof tree. Using a particular program clause to derive a particular goal will lead to successful unification and then a new goal will be generated, but the new and previous queries are identical up to renaming. Thus, this path will always keep forward and never terminates.

Here are some program clauses (from the given L_λ program) and rules from our encoding:

```
program(c(quantifier=[],
          body(quantfree,[Atom,[L]]))).
program(c(quantifier=[],
  body(quantfree,[And,[[A1],[A2]]]))).
        ...
select_atomic@@ goal(Q1,body($n,L)),
  program(c(Q2,body($m,R))), next :-
  $n=$m | goal(Q1,body($n,L)),
  copy(c(Q2,body($m,R))).
select_implied@@ goal(Q1,body($n,L)),
  program(implied(c(Q2,body($m,R1)),R2)),
  next :-
  $n=$m| goal(Q1,body($n,L)),
  copy(implied(c(Q2,body($m,R1)),R2)).
```

The rules `select_atomic` and `select_implied` (respectively for unit clauses and non-unit clauses) will nondeterministically decide to use a clause if the clause and the goal have the same predicate names. The `$n=$m` constraint ensures such matching. The program wrapped by an atom `copy` will be subject to the `copy` rule below.

HyperLMNtal provides a solution to graph copying. There is a constraint named *hlground*.

```
H :- hlground($g) | B.
H :- hlground($g,1,2) | a($g), b($g).
```

In the first line of code, `$g` is a graph structure, and `hlground` will check if this graph structure is a connected hypergraph with one free link. In the second line, `hlground` checks if the graph `$g` is a hypergraph with one free link consisting of hyperlinks with attributes 1 or 2, and copies it by using fresh names. Hyperlinks whose attributes are different from 1 and 2 will not be followed and retains their original names.

The copying of a program clause is done by the following `copy` rule.

```
copy@@ copy($w) :- hlground($w,1,2) |
  decide($w), program($w).
```

The `decide($w)` will be used to derive goal and the `program($w)` is reserved for future derivations.

Finally, presence of a loop in the proof search can be avoided. That is, we can arrange the conjunction of formulas in such an order that will not lead to a loop.

4.3 Managing procedures

Now, we need to precisely model the proof search process. When a goal is given, L_λ computes single or multiple answer substitutions, depending on the goal and program clauses. The prenex normal form program given earlier in this section generates five answer substitutions for `P`. Starting with a single goal, it might branch into multiple paths by different clauses, trying to prove each of them. When a rule is applied and unification fails, this path will terminate and indicate failure. As long as unification is successful, eventually the *initial* rule will be

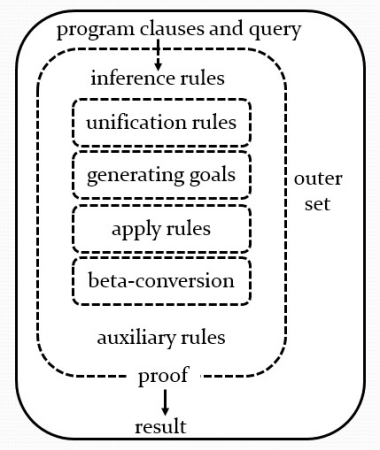

Fig. 5 Components of proof procedure

applied and proof search will end with success.

Non-deterministic execution mode of the Hyper-LMNtal system is used to model proof search, and the whole proof procedure is carefully classified to sub-procedures. Each procedure is packaged as a set of rules (called *rulesets*) placed in membranes as illustrated in Fig. 5. The following are the major rulesets in our encoding.

Outer ruleset. The `outer` membrane has rules to handle control flow, including the detection of finished sub-procedures, the triggering of the next sub-procedures and the copying of data between sub-procedures.

Proof ruleset. It receives program clauses and a goal from the outer ruleset when the proof search starts. This ruleset includes inference rules, unification rules, apply rules, β-conversion rules and auxiliary rules to coordinate them. The `proof` membrane stays active and regulates other sub-procedures until all goals are proved successfully or failed.

Unification ruleset. When unification is required, matching terms will be sent to this ruleset. It solves unification and stops with a unique solution if the unification is successful or indicates failure if the unification fails.

Generating goals ruleset. After unification is completed successfully, this set will be activated to generate new goals, depending on the program clause used to derive the goals.

Apply ruleset. After new goals are generated, `apply` will take substitutions from unification and apply them to all goals in the goal list.

β-conversion ruleset. Applying substitutions to goals might generate λ-application within these goals. This ruleset searches for λ-applications and perform a simple version of β-conversion (that covers the class of λ-applications occurring in L_λ) on them.

Besides, there are many auxiliary rules. The `proof` ruleset is placed within the `outer` ruleset, and all other rulesets are placed within the `proof` ruleset. By using *rule contexts*, *process contexts* and *stability constraints*, sending data and making judgment about the result of each procedure are achieved. Many unary atoms are used to denote the status of sub-procedures. Let's see some code below.

```
unify_no@@ unification{unify_no,$u,@u}/ :-
  unification{$u,@u}, unify_no.
```

This `unify_no` rule is one of the rules inside the `proof` set, which detects unification failure once unification is over (note the '`/`' stability constraint). The `unification` membrane generates a unary atom named `unify_no` when unification failed.

```
apply@@ proof{@p,$p,eq_info,
  eq_info{@e,$e},apply{@a,$a}}/ :-
  proof{@p,$p,temp_list=[],
    eq_info{@e},apply{@a,$a,$e},apply}.

get_subs@@ proof{@p,$p,
  apply{@a,$a,var{$v[]}},apply,
        sub_set{@s,$s}}/ :-
  proof{@p,$p,apply{@a},get_sub,
    sub_set{react{
            '$callback'('atomic_ruleset',
                        2),@s},
        @s,$s,$a}}.
```

These two rules are from the `outer` membrane. The `apply` rule detects if `eq_info` sub-procedure (which collects substitutions after successful unification) is finished and the `proof` membrane is stable. Then by producing an atom `apply` within the `proof` membrane, it sends a signal to start applying substitutions to goals. The `get_subs` rule works in a similar fashion. Now `get_subs` checks if the `apply` membrane is stable (meaning that `apply` has finished the application of substitutions to goals), then moves substitutions to a membrane named `sub_set` (used to store substitutions of unification).

Meanwhile, it defines `sub_set` rules as an *atomic ruleset* (Section 3).

As we have seen from the above example, membranes, rule contexts, process contexts, stability constraints and atoms are used to mark procedures, detect finished procedures, judge ended statues of finished procedure and invoke the next procedure.

The whole process of the computation can be observed by using *StateViewer* of our HyperLMNtal implementation, in which the red nodes represent either failure or successful termination of a search path. To have a clear picture, unification ruleset, apply ruleset and β-conversion rulesets are defined as atomic rulesets. As a result, only the major steps of proof search are displayed in StateViewer while ignoring trivial ones. The proof search of the the prenex normal form program is illustrated in Fig. 6. The leftmost node is the initial state of proof search. The highlighted five red nodes are final states of the successful paths. Many shadowed red states are final states of the failed paths. The arrows are transitions of states.

4.4 Essential features

Hyperlinks are used to represent variables. In pattern unification, there are two kinds of variables, flexible variables which are essentially existentially quantified and rigid variables which are essentially universally quantified. Assigning attributes to hyperlinks reflects their quantification kind. Although we used three different attributes for hyperlinks, except flexible variables given '1' as attributes, other hyperlinks are regarded as rigid variables.

In unification, we need to identify if an essentially universally quantified variable is quantified within the scope of essentially existentially quantified head of the *lhs* term. But luckily, such essentially universally quantified variables are explicitly quantified and such quantification is represented explicitly in our representation. Therefore, can be easily handled. With the features provided with hyperlinks, precise representation of formulas and encoding of pattern unification is made simple.

Fusion of hyperlinks is valuable in some cases. The `β-conversion` ruleset is to perform a simple version of β-conversion. Fusion makes that very easy. An expression $[lambda(x,t),s]$ represents $\lambda x(t).s$, where the binding variable x is a hyperlink. In the cases where s is also a hyperlink, then this β-contraction is done by simply fusing two hyperlinks, x and s. If s is not a hyperlink, we employ a search and replace strategy.

Membranes are frequently used to serve as a set of rules, a set of data or both of them at the same moment. With combination of rule contexts, process contexts, unary atoms and stability constraints, tasks such as exchanging data and program flow are achieved.

StateViewer, a visualization tool of HyperLMNtal, gives a clear picture of the computation. Without it, judging the correctness of modeling, especially in cases where large state spaces generated, would be tedious. To make the visualized state space graph to be more readable, atomic ruleset turned to be invaluable.

5 Conclusion and discussion

A modelling of L_λ proof search in HyperLMNtal has been proposed and described with a running example. Rewrite rules are used to encode the inference rules of L_λ to manipulate programs and goals. Proof search procedure is modeled using a combination of membranes, rule contexts, process contexts and stability constraints.

Experiences from this work have revealed that some improvements could be done for the primitives of HyperLMNtal for graph matching such as the typechecking and equality checking of atoms. These are just some oversight during implementation and are not technically difficult to improve. Currently, our model consists of more than 300 rules, which could be reduced further once the above improvements are made.

It turned out that HyperLMNtal was expressive and efficient for describing L_λ. Our major challenges of describing the syntax and modeling the semantic behavior are well handled through the use of various features, and StateViewer has played an important role to check the result of proof search.

During this work, it is observed that implementing pure higher-order logic programming of λProlog could be a very challenging task. One difficulty will come from the fact that quantification on predicates are allowed in higher-order setting. Another difficulty should be higher-order unification, which is a more complex procedure and for which there is

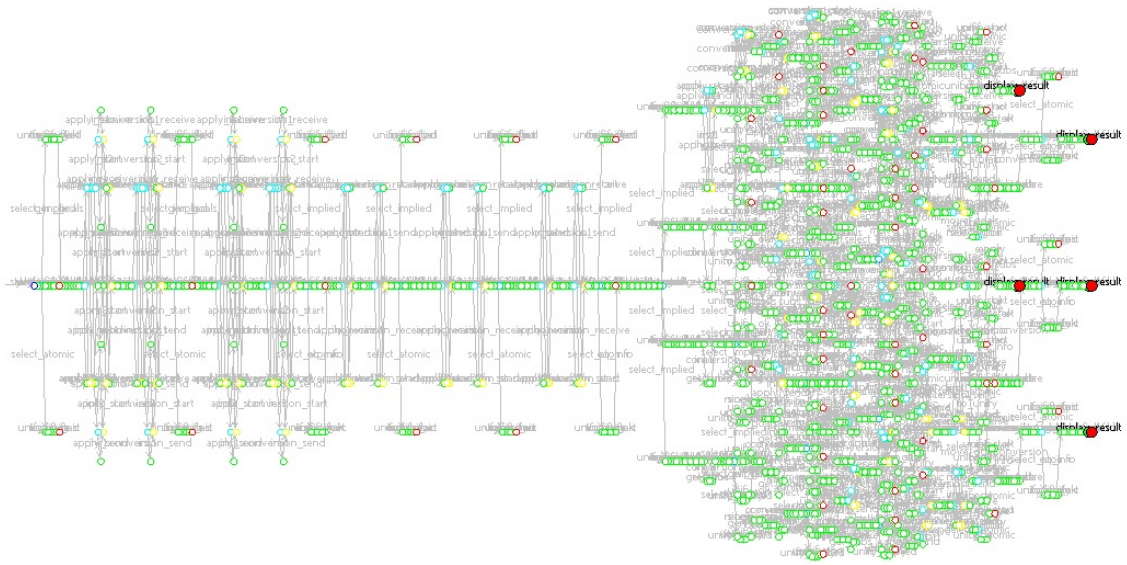

Fig. 6 Proof search illustrated by StateViewer

no guarantee of termination.

Based on the lessons learned, including higher-order features into HyperLMNtal rather than just encoding them will be an interesting future work. Also, enabling HyperLMNtal to handle higher-order unification should be another one of our future work.

Acknowledgment

We are indebted to the developers of HyperLMNtal, Seiji Ogawa and Manabu Meguro, for making the present work possible. This work is partially supported by Grant-In-Aid for Scientific Research ((B) 26280024), JSPS, Japan.

References

[1] Dale Miller and Gopalan Nadathur, Programming with Higher-Order Logic, Cambridge University Press, 2012.

[2] Kazunori Ueda, LMNtal as a Hierarchical Logic Programming Language. Theoretical Computer Science, **410**(46), 2009, pp. 4784-4800.

[3] Kazunori Ueda, Encoding the Pure Lambda Calculus into Hierarchical Graph Rewriting. In Proc. 19th International Conference on Rewriting Techniques and Applications (RTA 2008), LNCS 5117, Springer, 2008, pp. 392-408.

[4] Kazunori Ueda and Seiji Ogawa, HyperLMNtal: An Extension of a Hierarchical Graph Rewriting Model. Künstliche Intelligenz, **26**(1), 2012, pp. 27-36.

[5] Dale Miller, A Logic Programming Langauge with Lambda-Abstraction, Function Variables, and Simple Unification. In Proc. International Workshop on Extensions of Logic Programming, LNCS 475, Springer, 1991, pp. 253-281.

[6] Dale Miller, Abstract Syntax for Variable Binders: An Overview. In Proc. First International Conference on Computational Logic, LNCS 1861, Springer, 2000, pp. 239-253.

[7] Gopalan Nadathur, Teyjus: A λProlog Implementation. `http://teyjus.cs.umn.edu/`.

[8] Alimujiang Yasen and Kazunori Ueda, Encoding Type Systems into HyperLMNtal. In Proc. 2013 JSSST Conference, Tokyo, 2013.