TR- 909

Order Sorted Knowledge Representation and Reasoning in Legal Application

Xianchang Wang & K.Nitta & T.Tohzyoh & M.shibasaki

February,1995

Order-Sorted Knowledge Representation and Reasoning in Legal Application

Xianchang Wang[*], *Katsumi Nitta, Satoshi Tojo (MRI), Masato Shibasaki*
Institute for New Generation Computer Technology (ICOT)
1-4-28, Mita 1-Chome, Minato-ku, Tokyo 108 Japan

February 20, 1995

Abstract

In this paper, we give an overview of the order-sorted knowledge representation and reasoning in the formal legal reasoning system HELIC-II developed by ICOT. In section 2, we introduce the ψ-term theory proposed by Ait-Kaci, analyze this theory by *type-based finite automata*, and define the substitution, and unification concepts completed. In section 3, two levels of order sorted knowledge are introduced : One is the object knowledge represented by acyclic and non-acyclic ψ-term; The other is the statement knowledge, the basic formula of order-sorted logic, H-term, which is similar to acyclic ψ-term. Subsequently, two kind of order sorted rules (normal rule and extended rule) have been introduced and form the normal and extended sorted logic program. We discuss the denotational (fix-point) and operational (resolution) semantics of normal SL program without operator **not** and conclude the complete result. Then we discuss the argument theory based on standpoint in extended sorted logical program; At last, we compare our works with other SLP approaches.

Key words: Logic Programming, Type, Legal Reasoning, Argument, ψ-term, H-term

[*]Current address: Department of Computing Science, University of Alberta, Canada T6G 2H1. e-mail: xcwang@cs.ualberta.ca

Contents

1 Introduction

1.1 Motivation and Contents

Historically there are many papers on integrating order-sorted structure into the traditional logic programming such as [2] [8] [9]. Main reasons introducing the order-sorted structure into logic programming are:

1. Inheritance hierarchy structure can express real problems very naturally.

2. After introducing the order-sorted concept, the logic program can become shorter and the inference steps can be decreased. For example, the logic inference rules, like Modus Ponens rule, can be decreased greatly.

3. Order-Sorted logic is an extended and hence more powerful knowledge representation tool than non order-sorted logic. Here more powerful means economical and efficient in knowledge representation, convenient for programming, not in the sense of computability.

4. Many papers (for example, [2] [16]) argue that we can introduce different type of reasoning, such as deductive reasoning (knowledge generalization), analogy reasoning, negative as failure, argument reasoning, etc. in order-sorted logic programs. In this paper, we discuss the argument reasoning, negative as failure reasoning in SLP.

The basic difference of these SLP approaches, in our opinion, lie on how to represent the basic logical symbol (variable, type, function, predicate, label and so on.). According to this idea, most approaches on order-sorted logic programming can be classified into two categories. We call the first one function-based approach, and take C. Beierle et. al's work [2], [27] as typical example. This approach has a clear semantics about order-sorted unification as well as order-sorted resolution. In this approach, unary order-sorted literals can be introduced and hence form the order sorted predicate concept. However this approach has two problems. First, the object representation ability is weaker than ψ-term. Second it can only deal with the inheritance relationship between unary predicates.

We call another approach label-based approach, and take Ait-Kaci et. al's work [8], [9] as typical example. In this approach, the term of FOL formula is replaced by ψ-term which was first appeared in system *Login.* In fact, there are many similar concepts, such as Bob Carpenter et. al's feature structure [3], [17], and Kuniaki Mukai's partially specified term [13]. Although ψ-term expresses the object in the data record structure style and has strong knowledge representation ability, semantics of *Login* (unification, resolusion) has not been clear defined. Recently, an experimental integrated system *Life*, which integrate ψ-term, logic, function and equation together, has been proposed by Ait-Kaci et al. Although in paper [10], three equivalent descriptions about ψ-term, OSF-term, OSF-clause and OSF-graph, have been proposed and the type-theoretic, logic and algebraic renditions of a calculus of order-sorted feature approximations have been given, we still think that $Life$ only discuss the semantics of ψ-term in three different aspects and may provide a first step for integration. The semantic of integrating the logic, function, algebra and equality need be further studied. In this paper, we proposed a new description of ψ-term through type-based finite automata theory. By this concept, we have given a formal definition of substitution and SLD -resolusion of SL program. Finally we shown the equivalent relationship between connected type -based finite automata and OSF-graph of $Life$.

Readers may wonder why we choice *function* and *label* to distinguish the SLP approaches. Mainly there are two reasons. First, most SLP approaches adopt either function-based form or label-based form. Second, it is very difficulty to integrate the real $function$ style and *label* style programming into a unified order-sorted logic environment. Discussion about the theoretical difficulties of this integration can be found in Ait-Kaci's paper [8] in section 6.3.3 on page 339. And it's necessary to point out that, although *Life* try to integrate three orthogonal programming paradigms together, it is still unclear the semantics was completely solved according to our knowledge.

In Ait-Kaci's logic programming language Login [9], predicate symbols are required to have fixed variables and there are no hierarchy inheritance among predicates. This seems too constrained for legal reasoning.

Let's look at the following statements.

Tom hit Jim at class room. ......(1)

Tom hit Jim. ......(2)

For statement (1), we can formally express it by $hit(Tom, Jim, class\text{-}room)$, for statement (2), by $hit(Tom, Jim)$. What is the relationship between these two expressions? We think $hit(Tom, Jim, class\text{-}room)$ contains more information than $hit(Tom, Jim)$, and from a complete statement we can infer an incomplete statement. So from the fact $hit(Tom, Jim, class\text{-}room)$, we can get the statement $hit(Tom, Jim)$. This is reasonable in common sense reasoning. [1]

If we allow predicate have non-fixed arity, then it would be necessary to introduce labels attached to the arguments to express every argument's function in the predicate. For example, in expression *hit(Tom, Jim, class-room), Tom* is the person of subject (or agent) who *hit*, *Jim* is the person of object, who has been hit and *class-room* is the place where action *hit* take place. When we introduce the labels $agent, object, place$ attached to the corresponding arguments, the formal statement would be much easy to understand and contain more information. So the expression *hit(Tom, Jim, class-room)* should be replaced by *hit(agent= Tom, object= Jim, place= class-room).* Such expression is called H-term[2] in HELIC-II.

Suppose $h1, h2$ are two H-terms, if we can judge $h1$ contains more information than $h2$, then we denote this relationship by $h1 <_h h2$. Our philosophy is very simple, if $h1 <_h h2$ and $h1$ is a fact, then $h2$ does also hold.

In this paper we give an overview of the SLP research results in developed formal legal reasoning system HELIC-II[3], which is a formal legal reasoning system based on order-sorted logic programs. Its knowledge representation language describe the legal knowledge include following categories.

1. Type

   Type is the primary component in legal reasoning glossary [21][22]. It contains two classes and the corresponding relations among them. One is 'object' (or 'noun') glossary such as *Mary, person, male*, relationship among objects can be *Mary* $<_n$ *person, male* $<_n$ *person.* The other class is called 'verb'(or 'event', 'predicate') glossary such as *hit, do-violence, punishable, kill, injured,*

[1]According to this idea, both *Tom hit Jim at class-room* and *Tom hit Jim on saturday* can infer *Tom hit Jim*. So we can regard every basic formula h of SLP as an *existential* statement. That is, h is true if and only if there is a fact h' such that h' contains more information then h.

[2]H means Hypothetical Statement

[3]HELIC: means Hypothetical Explanation constructor by Legal Inference with Case by two inference engines.

with-criminal-intent, take-away, relationship among them can be *kill* $<_v$ *hit, hit* $<_v$ *do-violence.* The former is used for ψ-terms, and the later is used for legal knowledge description, H-term.

2. ψ-term and H-term

   ψ-term is a well-defined object. Objects such as '*a person whose age is 30, sex is male*' is formally represented by $person(age = 30; sex = male)$. ψ-term plays the similar rule of traditional logic programs term. H-term defines a legal description about an event.[4] Statement such as '*person Jim watches person Tom hit a male person*' can be represented by H-term *watch (agt1 =Jim, obj =hit (agt1 = Tom, agt2 = person (sex = male)))*. Obviously, H-term plays the same rule of atom in traditional logic programs.

3. Legal rule and priority knowledge.

   There are two kind of (legal) rules. One is normal rule, where logic negation operator $\neg$ appears. The other is called extended rule where negation as failure operator **not** as well as logical negation operator $\neg$ appears. Example of an extended rule is:

   $\neg$ *break-law(agt=x:person)* $\leftarrow$ **not** *break-law(agt=x)*

   Means: If you can not prove person x break the law, then you can assume he does not break the law.

   Each rule has at least one name called *unit*. We can define priority relationship among units. These priority relationship reflects the standard value of legal code. For example, '*New law has priority over old law*'. Standard knowledge is used to determine the best argument among contract arguments.

Detail description about HELIC-II can be referred in [20], [21], [22]. In this paper, we only focus on its theoretical aspects of order-sorted knowledge representation and reasoning.

Compared with the existed works on order-sorted logic programming, main contents and contributions of this paper are:

1. In section 2, we describe the ψ-term theory by type-based finite automata which is more powerful and efficient analyzing tool for ψ-term.[5] Concepts, such as substitution are clear defined and hence form the definite operational semantics, the resolution SLD procedure. Further in the subsection of section 5, we compare the OSF-graph concept proposed in [10] with our type-based finite automata, and get that they are equal. Efficient algorithm for ψ-terms unification have been listed in appendix.

2. In section 3, we extend the order-sorted structure into the predicate domain, and propose the predicate (or verb) lattice signature and define the basic formula H-term of legal programs. H-term is constructed on the *noun* and *verb* signatures. We give a clear semantics of sorted-logic program and, prove the completed result.

3. In section 4, we introduce the standpoint knowledge in our legal program and argument theory based on standpoint. There are two reasons to introduce priority knowledge (standpoint) and legal argument into SLP: First, the legal knowledge base is large-scale, it becomes difficulty to calculate the stable model of extended logic programs; Second a legal knowledge base may contain incompatible information. Argument can help us to get the justifiable conclusion, standpoint knowledge can help us to distinguish which conclusions are more believable and hence form a standard in argument process.

1.2 Two Approaches of SLP

In this section, we briefly introduce the two approaches of SLP and show the relationship between them. Our main conclusion in this section is: label-based SLP has a more powerful knowledge representation ability, function-based SLP has a more clear semantics.

Function-based order-sorted logic programs have the following syntax. The concepts and definitions here are borrowed from Beierle et al's work[2].

[4] In this paper, we don't distinguish the difference between event and general property statement

[5] Prof. Kuniaki Mukai introduce Bob Carpenter's the feature structure [3] to authors. Feature structure is similar to to our type-based finite automata concept.

Definition 1.1 An order-sorted signature σ = *(S, P, F)* consists:

(1) A partially ordered set of sorts $(S, \leq)$ with a least element $\bot$ and a greatest element $\top$. $(S, \leq)$ is called the sort hierarchy. We require that there are no infinitely ascending chains in $(S, \leq)$.

(2) A $(S^* \times S)$-indexed family of sets of function symbols $(F_{w,s})_{w \in S^*, s \in S}$. For $f \in F_{w,s}$ we write $f : s_1, ..., s_n \rightarrow s$ where $w = s1...sn$.

w is called the domain and s the range sort of f. The elements of w are the argument sorts of *f*.

(3) A S^*-indexed family of sets of predicate symbols $(P_w)_{w \in S^*}$. For $p \in P_w$ we write $P : s_1...s_n$ where $w = s_1...s_n$ is called P's argument sorts. □

Generally we assume that S, P, F are pairwise disjoint, each sort has at least one ground term(see the following definition of term). Say σ is a signature with equality if there is a binary relation symbol $\doteq$ with argument sorts $(\top, \top)$, i.e. $\doteq \in P_{\top,\top}$.

Furthermore, σ is a signature with sort predicates if for every sort s in S there is a unary predicate, also denoted by s , with argument sort $\top$, i.e. $s \in P_{\top}$.

Definition 1.2 Given a signature σ = (S, P, F), a family of variable over σ is a variable of V attached with type of S as following:

For every $x \in V, s \in S, s \neq \bot$, $x : s$ is a type variable.

The family of σ-terms $T_\sigma(V)$ over σ, V, called the set of well-sorted terms, is the least s-indexed family of sets such that:

(1) $x : s \in T_\sigma(V)_s$ for every $x \in V, s \in S$.

(2) $c \in T_\sigma(V)_s$ for every $c \in F_{\epsilon,s}$.

(3) $f(t_1, ..., t_n) \in T_\sigma(V)_s$ if $f \in F_{s_1...s_n,s}$ and for $i \in \{1, ..., n\}$, $s'_i \leq s_i$, $t_i \in T_\sigma(V)_{s'_i}$. □

For any well-sorted term t we represent its sort by function *sort(t)*. A well-sorted ground term is a well-sorted term in which no variable occurs.

We can see that if there is no infinite ascending chairs in $(S, \leq)$ then for every type s, except $\bot$ of S, there is at least one well-sorted ground term of s.

Formula in function-based order-sorted logic is defined as usual except the basic formula definition. Basically, if p is a predicate with argument sort $(s_1, ..., s_n)$, $t_1, ..., t_n$ are well-sorted terms with index $(s'_1, ..., s'_n)$ such that for $i = 1, ..., n$, $s'_i \leq s_i$, then $p(t_1, ..., t_n)$ is a atomic formula. Suppose $t1, t2$ are well-sorted terms, then $t1 \doteq t2$ is also basic formula.

In function-based SLP approach, we can also introduce hierarchy among predicates. However, this hierarchy description ability among predicates is limited to unary predicates. In *Hilog* system [15], every logic symbols (predicate, function) may have non-fixed arity. This property seems necessary from the programmer's opinion. *Login* allows object symbol have different arguments, but each predicate has fixed srity. In our system, both the object symbol and predicate symbol can have variable arguments.

The typical theory of label-based SLP is proposed by Ait-Kaci and his collaborators in [8], [9]. In their theory, every term was expressed by a type record structure called ψ-term. An example of ψ-term is:

student (id = name (last = x:string)); domicile = y:address (city = austin); father = person (id = name (last = x:string); domicile =y:address)[6]

Here, *student*, *name*, *string*, *address*, *austin*, *person* are type symbols; *id*, *last*, *domicile*, *city*, *father* are labels; x, y are called tags or variables.

The meaning of above ψ-term is: all the *student* whose *last name* is the same to his or her *father*'s *last name*, whose residence city is *austin* which is also the same of his or her *father*'s.

Compared with function-based approach, label-based approach is more natural and the ψ-term is more accessible to the type record structure. That is one reason we choose label-based order-sorted approach as our starting step in implementing the formal legal reasoning system HELIC-II.

In the rest of this subsection, we will informally show that every function-based sort term can be equally translated to a label-based ψ-term. The formal definition of ψ-term can be referred in section 2. So the

[6] In [9], this ψ-term is written by: $student(id \Rightarrow name(last \Rightarrow x : string); domicile \Rightarrow y : address(city \Rightarrow austin); father \Rightarrow person(id \Rightarrow name(last \Rightarrow x : string); domicile \Rightarrow y : address))$ In [10], this ψ-term is written by: $student(id \Rightarrow name(last \Rightarrow x : string); domicile \Rightarrow y : address(city \Rightarrow austin); father \Rightarrow person(id \Rightarrow name(last \Rightarrow x : \top); domicile \Rightarrow y : \top))$

knowledge base based on function-based SLP can be translated into a knowledge base based on label-based SLP.

Definition 1.3 Suppose $\sigma_f = (S, P, F)$ is a function based signature, V is a variable set, $\leq_f$ is a subsumption relation on S. Now we construct the corresponding label-based signature, called $tr(\sigma_f) = (S1, L1, V)$ such that:

(1). $S1 = S \cup \{$ Fset $\} \cup F$, $(S1, \leq_{S1})$ is defined as: $x \leq_{S1} y$ iff $x = \bot$ or $y = \top$ or $x = y$ or $x, y \in S$ and $x \leq_f y$ or $x \in F$ and $y = Fset$. Generally, we denote $\leq_{S1}$ by $\leq_f$.

(2). $L1 = \{functionname, l_1, ..., l_n, ...\}$.
□

Now let's show how each function-based well-formed term was translated into label based well-formed ψ-term.

Definition 1.4 Suppose σ_f is a function-based signature, $tr(\sigma_f)$ is a label-based signature. We now translate every function-based well-defined term into ψ-term as follows:

(1) $x : s$, where $x \in V$ and $s \in S$. $tr(x : s)$ is:

$x : s$

(2) c, here $c \in F_{\epsilon,s}$. $tr(c)$ is:

$s(function = c)$.

(3) Suppose $f(t_1, ..., t_n)$ is function-based signature and $f : (s_1, ..., s_n) \Rightarrow s$, for $i \in \{1, ..., n\}$, $t_i \in T_{\sigma_f}(V)_{s'_i}$ and $s'_i \leq s_i$, then $tr(f(t_1, ..., t_n)$ is:

s (functionname = f; $l_1 = tr(t_1)$; ...; $l_n = tr(t_n)$). □

For example, suppose f is a function with sort argument $(s1, s2) \Rightarrow s3$, g is a function with sort $(s1 \Rightarrow s2)$, then the function-based term $f(x : s1, g(y : s1))$ can be translated into an acyclic ψ-term

$s3(function = f; l_1 = x : s1; l_2 = s2(function = g; l_1 = y : s1))$.

It is easy to check that:

Suppose t is a function-based term, $\sigma = \{x1 : s1|t1, ..., xn : sn|tn\}$ is a substitution of t under definition of page 173 [2]. Then $tr(\sigma)$ is also a substitution of ψ-term $(tr(t)$. Here $tr(\sigma) = \{x1 : s1|tr(t1), ..., xn : sn|tr(tn)\}$.

Suppose σ is an mgu substitution of two function-based terms *t1, t2*, then we can also get that $tr(\sigma)$ is an mgu of two ψ-terms *tr(t1), tr(t2)*. That is *glb(tr(t1), tr(t2))* = *tr(t1)* $\circ tr(\sigma)$=*tr(t2)*$\circ tr(\sigma)$.

2 Label-based SLP

ψ-term theory was first proposed by Aic-Kaci [9]. We select this theory as the object expression form for the following reasons. Firstly, ψ-term is more natural than function-based order-sorted term. Its appearance is more accessible to the data record structure and hence has the more powerful representation ability. Secondly, we find type-based finite automata is a proper tool to define the ψ-term. In fact type-based finite automata is a date record structure based on type symbols. By type-based finite automata, ψ-term, and operations on ψ-terms, is more easy to understand. Finally, we give the completed *glb* and *lub* algorithms. Same to the conclusion of [9], the algorithms computational complexity is nearly in liner with its input length.

2.1 ψ-term Theory

In this section, we introduce the ψ-term theory. More details can be referred in Ait-Kaci's work [9] [10].

Definition 2.1 Signature

Say $\Sigma_n = (T_n, L_n, V_n)$ is a lattice signature, if T_n, L_n, V_n are parewise disjoint, $< T_n, \leq_n >$ is a lattice in which $\top$ is the maximal element, and $\bot$ the least element. We say T_n is the set of type symbols, its element is always denoted by ti; L_n is a label set, its element is denoted by li; V_n is the variable set, its element is denoted by xi. □

Definition 2.2 ψ-term [9]

We define the ψ-term on signature $\Sigma_n = (T_n, L_n, V_n)$ inductively as follows:

1. $x : ti$ is a ψ-term, if $ti \in T_n$ and $x \in V_n$.

   $x : ti$ is called simple term, when $ti = \bot$, it is called empty term.

2. Suppose $t0 \in T_n$, $x \in V_n$, $t1, ..., tn$ are ψ-terms, $l1, ..., ln$ are labels. If $t0 \neq \bot$, $i = 1, ..., n$, ti is not empty term, then $x : t0[l1 = t1, ..., ln = tn]$ is a ψ-term.

   For $i = 1, ..., n$, ti is called x's subterm under l_i.

□

Suppose $t = x : s(..)$, we define *root-type(t)=s, root-variable(t)=x.*

Example of ψ-terms are:

$t1$=$x0 : person(id = x10 : name(first = x20 : string; last = x21 : string); father = x11 : person(id = x22 : name(last = x21 : string)))$.

$t2$= $x0 : person(father = x2 : person (son = x0 : person))$.

All the variables (such as $x0$, ...) appeared in the ψ-term are called tag symbols in [9]. We can see that some tags may appear more then one time. The first ψ-term express the set of *person* whose *father*'s last name is the same to his last name. It is easy to see that only the tags appear more then one times are useful. For simplicity, we sometimes omit the tags that appear only once in a ψ-term. So ψ-terms appeared in above example can also be written as follows:

t1= person (id= name (first= string; last = x21:string); father = person (id= name (last= x21:string))).

$t2$=$x0 : person(father = person(son = x0 : person))$.

Subterms are defined as usual. We say $x : s$ is a simple term if x is a variable and s is a sort symbol. A simple term is empty if the sort is $\perp$.

Definition 2.3 well-formed ψ-term.

Say ψ-term t is a well-formed ψ-term, iff

For every two subterms of t like $x : t1$, $x : t2$ we must have

1. At least one subterm is simple term.
2. *root-type(x:t1) =root-type(x:t2).*

We denote the set of all the well-formed ψ-term on signature Σ_n by Ψ^{Σ_n}. □

This definition says that if a ψ-term t's two different subterms have the same root-variable, then they must have the same type, and at least one is simple term.

Hence, ψ-term $t1$ $t2$ are well-formed. Following two ψ-terms are also not well-formed.

$t3 = X0 : v0(l1 = X1 : v1; l2 = X1 : v2)$

$t4 = X0 : v0(l1 = X0 : v1(l1 = X1 : v1(l2 = X0)); l2 = X1 : v0(l1 = X2 : v1))$

Suppose t is a well-formed ψ-term, we define $V(t) = \{x|x \text{ is a variable appearing in } t\}$, $L(t) = \{l|l \text{ is a label appearing in } t\}$

Definition 2.4 π_t on well-formed ψ-term t.

Suppose t is a well-formed ψ-term. We define a partial function

π_t: $V(t) \times L(t) \rightarrow V(t)$,

$type_t : V(t) \rightarrow T_n$ as follows:

For every $x \in V(t)$, $l \in L_n$,

$type_t(x)$ =*root-type(x:t1)*, here $x : t1$ is the subterm of t.

$\pi_t(x, l) = y$ iff There is a subterm $y : t1$ such that $y : t1$ is the subterm of x under label l. □

For every ψ-term t, every label string α, every variable x, we define $\pi_t(x, \alpha)$ as following:

$\pi_t(x, \epsilon) = x$

$\pi_t(x, \alpha.l) = \pi_t(\pi_t(x, \alpha), l)$

Suppose *x0=root-variable(t)*, if $\pi_t(x0, \alpha) = y$, we say α can be accepted by ψ-term $t's$ variable y and denote this fact by $V_t(\alpha) = y$. For every two different prefixes α_1, α_2 of α, if $\pi_t(x0, \alpha_1) \neq \pi_t(x0, \alpha_2)$, then we say α is the economic path accepted by y . We denote $Ext(t, y)$ as the set of all the strings accepted by y, and $Ext(t)$ as the set of all strings accepted by any variable. Denote $domain(t, x)$ are all the economic paths attaining x and $domain(t)$ is all the economic paths of ψ-term t.

Definition 2.5 Say ψ-term $t \simeq \perp$ if it has an empty subterm. □

Definition 2.6 Complete Definition of $\leq_t$.

Suppose $t1$, $t2$ are two ψ-terms, we define relationship $t1 \leq_t t2$ iff

1. $t1 \simeq \perp$ or
2. The following two conditions hold.

   2.1. $domain(t2) \subseteq Ext(t1)$.[7]

   2.2. For every $\alpha \in domain(t2)$, we have

   type(root-variable(t1)$, \alpha) \leq_n$ *type(root-variable(t2)*$, \alpha)$ and

   $Ext(t2, \pi_2($ *root-variable(t2)* $, \alpha)) \subseteq$

   Ext(t1, π_1 *root-variable(t1)* $, \alpha)$).

□

The condition 2.1 can be replaced by condition $Ext(t2) \subseteq Ext(t1)$. Compared with the definition 4.10 of [9] in page 311, the subsumption relation between well-formed ψ-terms here is more formal and easy understanding.

[7]This condition can not be replaced by $domain(t2) \subseteq domain(t1)$. A contradict example is $t1 = x0 : person(like = x0)$, $t2 = y0 : person(like = y1 : v(like = y0))$. Although $t1 \leq_t t2$ is reasonable , but $domain(t2) \not\subseteq domain(t1)$, conversely, $domain(t1) \subseteq domain(t2)$.

Definition 2.7 Suppose t is a ψ-term, x, y are variables appeared in t. We say x appears in y's valid scope iff there are two label sequence α, β, $\beta \neq \epsilon$, such that $V_t(\alpha) = y$ and $V_t(\alpha.\beta) = x$. □

Look at ψ-term $t5 = x0 : s0(l1 = x1 : s1(l2 = x2 : s2); l2 = x2 : s2(l3 = x1 : s1))$. It is easy to check *x1, x2* appear in scope of $x0$. But we must be careful that *x1, x2* also appear in the scope of $x1$ and $x2$.

Definition 2.8 Say ψ-term t is acyclic about its variable x, if x does not appear in its own scope, else say t is non-acyclic about variable x;

Say ψ-term t is acyclic, if it is acyclic about every variable, else it is non-acyclic. □

Let's consider another example:

$t6 = X0 : v1(l1 = X0 : v1; l2 = X1 : v2)$

$t7 = X2 : v3(l1 = X3 : v4(l3 = X2 : v3); l3 : X4 : v4)$

$t8 = Z0 : v134(l1 = Z0 : v134; l2 = Z1 : v2); l3 = Z0 : v134)$

Here $v134 = glb(\{v1, v3, v4\})$.

We can see $t8 \leq_t t7$, $t8 \leq_t t6$. In fact $t8$ is the great lower bound of $t6, t7$.

We define $t1 \cong t2$ iff $t1 \leq_t t2$ and $t2 \leq_t t1$. It's easy to conclude that $\cong$ on Ψ^{Σ_n} forms an equivalent relation. Let $\Psi^{\Sigma_n}/\cong$ be the equivalent class, then we have the following important results:

Theorem 2.1 (Refer [9]) *If signature $\Sigma_n = < T_n, L_n, V_n >$ is a lattice signature, then $< \Psi^{\Sigma_n}/\cong, \leq_t >$ is also a lattice.* □

2.2 Type-based Finite Automata Theory

Definition 2.9 Say $Auto = < L, V, \pi, type, x0 >$ is a type-based finite automata on lattice signature $\Sigma_n = < T_n, L_n, V_n >$, if L, V are finite subset of L_n, V_n, $x0 \in V$, π is a partly function from $V \times L \rightarrow V$, and *type* is a function of $V \rightarrow T_n$.

We assume that for every $x \in V$, if *type(x)* $= \perp$ then $V = \{x0\}$ and $L = \{\}$. In this case, we call *Auto* an empty automata. □

Definition 2.10 Say sequence $l1...ln$ ($n >= 1$) is accepted by type-based finite automata $Auto = < L, V, \pi, x_0, type >$, if there is a variable sequence $x_0, x_1, ..., x_n$, such that for every $i, n > i >= 0$, $\pi(x_i, li) = x_{i+1}$. We say $l1...ln$ is accepted by x_n. □

Specially, we say ϵ is accepted by x_0. We denote the set of all the accepted sequence of $Auto = < L, V, \pi, x0, type >$ by $Ext(Auto)$, and denote length of *Auto*, $|Auto|$, is the sum length of L, V, and π, denote the accepted sequence of *Auto* by variable y is *Ext(Auto, y)*.

It is easy to see that,

$$Ext(Auto) = \sum_{x \in V of Auto} Ext(Auto, x).$$

Generally, we assume that if $L \neq \{\}$, then for every variable $x \in V$ of *Auto*$= <L, V, \pi, x0, type>$, Ext (Auto, x) $\neq \{\}$ and for every $l \in L$ there are $x, y \in V$ such that $\pi(x, l) = y$. That is, every node in V lies on a directed path starting at the root node $x0$. In this case we say *Auto* is connected. We denote the set of all the type-based finite automata by CA^{Σ_n}, and the set of all the connected finite automata based on signature Σ_n by CAU^{Σ_n}.

It is easy to conclude that

Theorem 2.2 *Every type-based finite automata can be translated into a well-formed ψ-term; And conversely, every well-formed ψ-term can be translated into a connected type-based finite automata.* □

Let $\Psi : AU^{\Sigma_n} \rightarrow \Psi^{\Sigma_n}$ means translating an automata into a well-formed ψ-term, and Δ: $\Psi^{\Sigma_n} \rightarrow CAU^{\Sigma_n}$, then it is easy to get:

Theorem 2.3 *Suppose $id_{\Psi^{\Sigma_n}}$ is the identity function on domain Ψ^{Σ_n}, $id_{CAU^{\Sigma_n}}$ is the identity function on domain CAU^{Σ_n}, then*

$\Psi \circ \Delta = id_{\Psi^{\Sigma_n}}$

$\Delta \circ \Psi = id_{CAU^{\Sigma_n}}$ □

Example 2.1 Let's consider ψ-term $t1$,

$t1 = x0 : person(id = x10 : name(first = x20 : string, last = x21 : string), farther = x11 : person(id = x22 : name(last = x21 : string)))$.

$t1$'s corresponding type-based finite automata is:

x0	person	id	x10
		father	x11
x10	name	first	x20
		last	x21
x11	person	id	x22
x20	string		
x21	string		
x22	name	last	x21

□

Translating algorithms can be refereed in appendix. Now, we define the relationship between two automates.

Definition 2.11 Suppose $auto_i =< L_i, V_i, \pi_i, type_i, x_i >, i = 1, 2$ are two connected type-based finite automates, we say $auto_1 \leq_t auto_2$ iff there is a map $f : V_2 \to V_1$ such that

1. $f(x_2) = x_1$.
2. For every $x \in V_2$, $type_1(f(x)) \leq_n type_2(x)$.
3. $L_2 \subseteq L_1$ and for every $l \in L_2, \{x, y\} \subseteq V_2$, if $\pi_2(l, x) = y$ then $\pi_1(l, f(x)) = f(y)$. □

The following theorem holds:

Theorem 2.4 *For every two well-formed ψ terms $t1, t2$, two connected automates $auto_1$, $auto_2$, we have*

1. $t1 \leq_t t2$ iff $\Delta(t1) \leq_t \Delta(t2)$.

2. $auto_1 \leq_t auto_2$ iff $\Psi(auto_1) \leq_t \Psi(auto_2)$

□

In the last section, we describe the concept of variable's scope in a well-formed term. Using automata, we can define subterm of a well-formed ψ-term.

Definition 2.12 Suppose t is a well-formed *psi*-term, its automata is $\Delta(t) = < L, V, \pi, type, x0 >$. For every $y \in V$, we say t's y-subterm , denoted by $subterm(t, y)$, is the well-formed ψ-term $\Psi(< L, V, \pi, type, y >)$. □

For example, in well-formed ψ-term $t1$, the $x10$ sub-term of $t1$ is: $x10 : name(first = x20 : string, last = x21 : string)$. The $x2$ sub-term of $t3$ is $x2 : person(son = x0 : person(farther = x2 : person))$.

From definition 2.11, we can see that definition of $\leq_n$ there is much more simple than the definition in section 2.1. From theorem 2.4, we can see that type-based finite automata is another proper tool to define ψ-term.

While we finish this paper, Prof. Kuniaki Mukai introduce authors the feature structure in Bob Carpenter's book [3]. Feature Structure is nearly the same to our type-based finite automata. Interesting readers can also refer [3] for more detail introduction.

2.3 Computation and Complexity Analyze

In this section, we show ψ-terms glb and lub computations and their complexity analyze. Detail algorithms can be refereed in appendix and [9]

In HELIC-II, every ψ-term's inner structure is in fact a type-based finite type automata. Algorithms listed in appendix can help us understand the computing procedure of ψ-terms' greatest lower bound and H-term's least upper bound.

Example 2.2 Consider following two ψ-terms:

$s = X0 : v1(l1 = X0 : v1; l2 = X1 : v2)$

$t = X2 : v3(l1 = X3 : v4(l3 = X2 : v3); l3 = X4 : v4)$

$lub(s, t)$ is $Z0 : s13(l1 = Z1 : s14(l3 = Z0 : s13))$, here $s13 = lub(v1, v3), s14 = lub(v1, v4)$.

$glb(s, t)$ should be

$Z0 : v134(l1 = Z0 : v134; l2 = Z1 : v2; l3 = Z0 : v134)$

Here $v134 = glb(\{v1, v3, v4\})$.

Suppose ψ-term $t9 = X0 : v0(l1 = X0, l2 = X0, l3 = X1 : v1(l4 = X0))$,

The corresponding type-based finite automata is:

X0	vo	l1	X0
		l2	X0
		l3	X1
X1	v1	l4	X0

The well-formed ψ-term:

$t10 = Y0 : v2(l1 = Y1 : v3(l2 = Y0); l2 = Y2 : v4(l3 = Y0; l4 = Y3 : v5); l3 = Y0; l5 = Y3)$.

The corresponding type-based finite automata is:

Y0	v2	l1	Y1
		l2	Y2
		l3	Y0
		l5	Y3
Y1	v3	l2	Y0
Y2	v4	l3	Y0
		l4	Y3
Y3	v5		

$glb(t9, t10)$ is:

$z0 : v_{01234}(l1 = z0; l2 = z0; l3 = z0; l4 = z1 : v5; l5 = z1)$

Here $v01234$ is $glb(\{v0, v1, v2, v3, v4\})$.

z0	v01234	l1	z0
		l2	z0
		l3	z0
		l4	z1
		l5	z1
z1	v5		

lub(t9, t10) = z0: s_{02} (l1 = z1: s_{03} (l2 = z0: s_{02}); l2 = z2: s_{04} (l3= z3: s_{12}), l3= z3: s_{12})

Here, $s_{i1,...,ik} = lub\ \{i1, ..., ik\}$.

□

Main conclusions are:

Proposition 2.5 For every finite automata, $Auto = < L, V, \pi, x0, type >$, every label sequence α,

1. The computational complexity to decide whether α is accepted by $Auto$ is $O(|\alpha| + |Auto|)$.

2. The computational complexity to decide whether α is accepted by a given state c is also $O(|\alpha| + |Auto|)$ □

Theorem 2.6 *The complexity deciding whether a label sequence α belongs to a path of ψ-term t is $O(|\alpha|)$, and The complexity deciding whether two ψ-terms $h1, h2$ have the relation $h1 \leq_t h2$ is $O(|h1| + |h2|)$.* □

Suppose s and t are two terms, we always have the following two questions:

1. Does s equal to $\perp$?

2. Does s equal to t?[8]

From theorem 2.4, above proposition and theorem, we can easily get that

Theorem 2.7 *The complexity deciding whether a ψ term s equals to $\perp$ is $O(|s|)$.*

The complexity of deciding whether two ψ-terms s and t are equivalent is the liner time of the total length of s and t, $O(|s| + |t|)$. □

Theorem 2.8 *[9]*

The complexity of the algorithm computing the greatest lower bound of any two ψ-terms is almost linear of n,[9] where n is the total length of t and s. □

[8]Here, s equals t means $s \cong t$, that is $s \leq_t t$ and $t \leq_t s$. It differs from $s = t$ where s and t are totally equelvenent by syntax

[9]In fact, The complexity is $O(nG(n))$, since for all $n \leq 2^{65536}$, $G(n) \leq 5$, $G(n)$ can be considered as a constant. Here $G(n) =_{df} min\{k | n \leq F(k)\}$. $F(0) = 1$, for all i, $0 \leq i$, $F(i+1) = 2^{F(i)}$. Detail discussion, see [4]

2.4 Substitution and Unification of ψ-terms

Now we turn to ψ-terms substitution and unification problem.

Definition 2.13 Suppose label sequence α can be accepted by t's variable y, then we denote α-subterm of t , denoted by $subterm(t, \alpha)$ is $subterm(t, y)$. □

Definition 2.14 Suppose t is a ψ-term, x, y are two variables of t. if y appears in x's scope, then there must be a least label string α and a label string β such that $\beta \in Ext(t, x)$, $\beta\alpha \in Ext(t, y)$. We say α is a address of y in variable x of t. □

Definition 2.15 [Substitution]

Say $\theta = \{x1/t1, ..., xn/tn\}$ is a well order-sorted substitution of ψ-term t, if

1. $t1, ..., tn$'s variable is disjoint with t's variable. We assume ti's initial variable is xi, initial type is vi.

2. If variable z appears in the scope of any xi $i = 1, ..., n$ of t, then z must also equals to some xi, $i = 1, ..., n$.

3. If variable xi appears in the scope of Xj of t, and the address of xi in xj of t is α, then $subterm(\alpha, tj) = ti$.[10]

4. For xi, suppose *xi-subterm* of t is *subterm (xi, t)*, then $ti \leq_t subterm(xi, t)$.

□

Obviously, we have

For any ψ-term t, suppose θ is t's substitution, then for any *subterm* $t1$ of t, θ must be also $t1$'s substitution.

Definition 2.16 Suppose θ is a substitution of ψ-term t, then we inductively define $t\theta$ as follows:

1. If t's initial variable is xi, then $t\theta$ is ti. else

[10]we assume that if xi appears in the scope xj of ψ-term t, we needn't write $xi|subterm(\alpha, tj)$ in the substitution.

2. Suppose t is $x : v(l1 = t1, ..., lm = tm)$ then $t\theta$ is

   $x : v(l1 = t1\theta, ..., lm = tm\theta)$.

   Here we assume that after xi is first replaced by ti, the following xi should be replaced by $zi : vi$, vi is the top-type of term ti. This assumption insure that if a variable is substituted more then one time in a ψ-term, then the result term must be also well-defined.

□

Theorem 2.9 *For every substitution θ of ψ-term t, we have $t\theta \leq_t t$* □

Proof: We inductively prove above proposition on the depth of the ψ-term.

Step 1. When $t = xi : v$, if $\theta = \{..., xi|ti, ...\}$ then $t\theta = ti \leq_t t$, else $t\theta = t \leq_t t$.

Step 2. Suppose above proposition hold for depths l.

Step 3. Suppose *$t = x1$: $v(l1 = t1, ..., ln = tn)$* in which every *ties* length is not greater than l.

If $\theta = \{..., x|t1, ...\}$, then according the definition we have $t\theta = ti \leq_t t$, else $t\theta = x1 : v(l1 = t1\theta, ..., ln = tn\theta)$. Firstly, according the inductive step, we have $t1\theta \leq_t t1$, ..., $tn\theta \leq_t tn$. Secondly, please notice that $x1$ can not be appeared in the scope of any $ti, i = 1, ..., n$, hence if two address in t has the same variable then the two address in $t\theta$ must have the same variable.

Hence, $t\theta \leq_t t$ and, conclude the theorem.

Obviously, we say a ψ-term set E is unifiable, if there is a unification θ of E such that for every two ψ-term $t1, t2$ of E, $t1\theta = t2\theta \not\simeq \perp$.

Theorem 2.10 *Suppose $t1, t2$ are two ψ-terms, (Generally we assume their variables are disjoint), then $t1, t2$ can be unifiable iff $glb(t1, t2) \not\simeq \perp$.* □

Proof:

$\Leftarrow$ Suppose there is a θ such that $t1\theta = t2\theta \not\simeq \perp$ and $t1\theta \leq_t t1$, $t2\theta \leq_t t2$. Since $t1\theta \leq_t glb(t1, t2)$, we get

$glb(t1, t2) \not\simeq \perp$.

$\Rightarrow$ Suppose $t3 = glb(t1, t2) \not\simeq \perp$, and $t1, t2, t3$ are variable disjoint. Suppose $t1$'s initial variable is x, $t2$'s initial variable is y, it is easy to construct the substitution

θ is $\{x|t3, y|t3\}$.

If $t1 \leq_t t2$ then not for every substitution $\theta, t1\theta \leq_t t2\theta$. For example, $x : bird \leq_t y : animal$, $\theta = \{y : animal/z : ostrich\}$, then $(x : bird)\theta \not\leq_t (y : animal)\theta$.

Definition 2.17 [maximal general unifier]

Suppose E is a ψ-term set, E can be unifiable. For every two substitution θ, β, we define $\theta \leq_t \beta$ iff $E\theta \leq_t E\beta$. Say E's substitution β is the maximal general unifier iff if θ is a unification of E then $\theta \leq_t \beta$. □

Theorem 2.11 *Every unifiable ψ-term set E have one (and only one) maximal general unifier θ.* □

3 Sorted Predicate and SL Program

HELIC-II provides two different signatures. One is noun-signature, denoted by $\Sigma_n = (T_n, L_n, V_n)$. Symbols of T_n is called type or noun-type, symbols of L_n is called noun-label or label, symbols of V_n is called variable or noun-variable. The other is verb-signature, denoted by $\Sigma_v = (T_v, L_v, V_v)$, where symbols of T_v is called verb type or predicate, symbols of L_v is called verb-label or predicate label, symbols of V_v is called verb-variable or H-variable. All the objects following described are based on these two signatures and the subsumption relations.

Notices that these subsumption relations can be regarded as a part of the knowledge stored in legal knowledge base which was written by legal knowledge engineer. Examples of these two relations are:

Some noun-type symbols and their subsumption relations:

adult $\leq_n$ person, child $\leq_n$ person,

japanese $\leq_n$ person,

japan $\leq_n$ country, monthname $\leq_n$ name,

Some verb-type symbols and their subsumption relations:

kill-with-brows $\leq_v$ strike, strike $\leq_v$ violence, kill-with-brows $\leq_v$ kill, kill $\leq_v$ violence.

There are two basic objects in traditional logic programming, term and atom. In HELIC-II, we also have the corresponding two objects, $\psi-$term and H-term. We assume $\psi-$terms are based on Σ_n. Sometimes, we use new symbol to name certain $\psi-$term, for example:

man $=_{df}$ person[sex = male]

japanese-man $=_{df}$ person [sex = male; nationality = japan].

Obviously, *japanese-man* $\leq_n$ *man*, *man* $\leq_n$ *person*.

Different from the ψ-term, H-term is not only defined on verb-signature $\Sigma_v = (T_v, L_v, V_v)$, but also defined on noun-signature $\Sigma_n = (T_n, L_n, V_n)$.

H-term is of the form: $p(c1 = q1, ..., cn = qn)$ or $\neg p(c1 = q1, ..., cn = qn)$ where p is a predicate, qi is a ψ-term or H-term, ci is a label (noun-label if qi is ψ-term or verb-label if qi is H-term). Generally we assume $p, \neg p$ are symbols of T_v, and $lub(p, \neg p) = \mathbf{t}$, $glb(p, \neg p) = \mathbf{f}$, $\neg\mathbf{t} = \mathbf{f}$, $\neg\mathbf{f} = \mathbf{t}$. Further we assume if $p \leq_v q$ then $\neg q \leq_v \neg p$.

There are two kinds of negations in SLP. One is $\neg$, called logic negation; The other is **not**, called negation as failure. A simple example is:

$P = \{\ fly(agt = x : bird) \leftarrow$ **not** $\neg fly(agt = x : bird)$,
$ostrich \leq_n bird$,
$\neg fly(agent = ostrich)$,
$tweety \leq_n bird\ \}$

In this section, we will introduce H-term, its substitution and unification, normal SL program and extended SL programs, and their denotational and operational semantics.

3.1 H-term

Now, we define the acyclic H-term. Informally, H-term is a constrained ψ-term based on signatures Σ_n and Σ_v. The constrained conditions are: H-term is acyclic about its verb-variable, every verb-label should be followed by H-term, and every noun-label should be followed by ψ-term.

Definition 3.1 Suppose $\Sigma_n =< T_n, L_n, V_n >$ is a noun lattice signature, $\Sigma_v =< T_v, L_v, V_v >$ is a verb lattice signature such that Σ_n, Σ_v are disjoint. Now we define H-term of T_v based on T_n as fellows:

1. For every $v \in T_v$, $@x \in V_v$, $@x : v$ is a H-term.

This kind of H-term is called simple H-term, when v is $\mathbf{f}$, we call it a false statement, when v is $\mathbf{t}$, it is called a true simple statement.

2. Suppose $@y0 \in V_v$, $v \in T_v$, $\{l1, ..., lk\} \subseteq L_n$ $\{hl1, ..., hlm\} \subset L_v$, $t1, ..., tk$ are ψ-terms, $h1, ..., hm$ are H-terms, then

$h = @y0 : v\ (l1 = t1, lk = tk, hl1 = h1, ..., hlm = hm)$ is H-term. □

Definition 3.2 Well-defined H-term,

Say H-term h is well-defined, if the following conditions hold:

For every two subterm of h, $x : \alpha$, $x : \beta$:

1. $x : \alpha$ and $x : \beta$ have the same initial type.

2. If x is noun-variable then at least one of $x : \alpha$ and $x : \beta$ is the simple ψ-term.

3. If x is verb-variable, then at least one of the subterms is simple H-term and x can not be appeared in its own scope. □

Here concepts, such as subterm of H-term, variable's scope is defined similar to the ψ-term's. Generally, we don't write all the variables, except the variables appear more then one times. All the H-terms discussed in this paper are assumed well-defined.

Before introducing the formal semantics of H-term, we first explain H-term's meaning informally. Suppose $h1 = hit\ (agent = Tom, object = Jim, place = house1)$, $h2 = hit(agent = Tom, object = Jim, place = house1, time = 1994.7.20\ )$ and $h3 = hit\ (\ agent = Tom, object = Jim)$. $h1$ means *Tom hit Jim at house1*, $h2$ means *Tom his Jim at house1 on 1992.7.20*, $h3$ means *Tom hit Jim*.

Imagine h1 is a fact in the real world, then which statement can be concluded? $h2$ or $h3$? It is easy to see that we can get $h3$. Besides, from $h2$ we can also get $h1$.

The logical relationship between H-terms is expressed by the ordered relation $\leq_h$. Later, we will see that $h2 \leq_n h1, h1 \leq_n h3$, and hence from $h1$ we can get $h3$ instead of $h2$.

Definition 3.3 Say H-term $h \cong \mathbf{t}$ iff $\perp$ appears in h.

Say H-term $h \cong \mathbf{f}$ iff $h \ncong \mathbf{t}$ and $\mathbf{f}$ appears in h. □

For example, $\mathbf{f}(agt = x : person)$ is a false statement about person, so it equals to $\mathbf{f}$; $hit(agt = x : \perp, obj = y : person)$ is a statement whose subject domain is $\perp$ (empty), we regard this kind of statement as true.

Definition 3.4 Suppose $h1 = @x{:}v1(\ l_1 = t_1, ..., l_n = t_n, vl_1 = vt_1, ..., vl_m = vt_m)$, $h2 = @y{:}v1(\ l_{i1} = t'_1, ..., l_{ik} = t'_k, vl_{j1} = vt'_1, ..., vl_{jl} = vt'_l)$ are two H-terms, such that $\{i1, ..., ik\} \subset \{1, ..., n\}$, $\{\ j1, ..., jl\} \subseteq \{1, ..., m\}$, $l1, ..., ln$ are noun-labels, $vl_1, ..., vl_m$ are verb-labels. We define

$h1 \leq_h h2$ iff $h1 \cong \mathbf{f}$ or $h2 \cong \mathbf{t}$ or

1. $v1 \leq_v v2$
2. For every $s = 1, \ldots, k$, $t'_s \leq_t t_{is}$. [11]
3. For every $s = 1, \ldots, l$, $vt_{js} \leq_h vt'_s$.
4. If *variables* appear in two different places of h2 are same, then the variables appear in the corresponding places of h1 must be also same. □

The meaning of $h1 \leq_h h2$ is:

If h1 is a fact in the real world, then h2 is also a fact in the real world.

We denote $h1 \cong h2$ iff $h1 \leq_h h2$ and $h2 \leq_h h1$.

For example, suppose

$ostrich \leq_n bird \leq_n animal$, *fly-by-wings*$<_v fly <_v move$.

Fact: $fly(agent = x : bird)$.

Then we can get:

fly because $fly(agent = x : bird) \leq_h fly$[12]

$move$ because $fly \leq_h move$[13]

$fly(agent = x : ostrich)$ because *fly(agent = x:bird)* $\leq_h$ *fly(agent = x:ostrich)*.[14]

$move(agent = x : bird)$ because *fly(agent = x:bird)* $\leq_h$ *move(agent = x:bird)*

$move(agent = x : ostrich)$ because *move (agent = x:bird)* $\leq_h$ *move (agent = x:ostrich)*

Definition 3.5 Conclusion of a fact set

Suppose F is a H-term set, We say Con(F) is the conclusion set of F, if Con(F) is the least set satisfies the following conditions:

1. $F \subseteq Con(F)$
2. If $h1 \leq_h h2$ and $h1 \in Con(F)$ then $h2 \in Con(F)$ □

Definition 3.6 Say fact set F is inconsistent iff there is a H-terms h such that:

1. $h \in Con(F)$ and $\neg h \in Con(F)$.
2. For every $h1, \neg h2 \in Con(F)$, if $h1 \leq_h h$ and $\neg h2 \leq_h \neg h$ then $h1 \cong h2$. □

Suppose F={ *hit(agt = Tom, obj = Jim, time = sunday)*, $\neg$ *hit(agt =Tom, obj = Jim, place = house1)*}. Although $Con(F)$ contains both *hit (agt = Tom, obj = Jim)*, $\neg$ *hit (agt =Tom, obj =Jim)*, $Con(F)$ is consistent according to above definition.

Definition 3.7 Complete Set, Minimal Set.

Say fact set F is a complete set iff $Con(F) = F$.

Say fact set F is a minimal set iff if $h \in F$ and $h \not\cong h'$ and $h \leq_h h'$ then $h' \notin F$.

Say fact set F can be minimized iff There is a minimal set, defined by $Min(F)$ such that $Min(F) \subseteq F$ and $Con(F) = Con(Min(F))$. Element of $Min(F)$ is called the complete information or fact of F. □

From the inconsistent definition, we can prove that:

Theorem 3.1 *Fact set F is consistent iff $Min(F)$ does not contain contradictory fact.* □

Obviously, not every complete set can be minimized. For example,

Example 3.1 Suppose complete set $F1$ is:

$\{h, h(\,l1= c1), h(\,l1= c1(\,h2= c2)), \ldots, h(\,l1= c1(\ldots(\,ln= cn\,)\ldots)), \ldots\}$

Then $F1$ can not be minimized.

Suppose complete set $F2$ is:

$\{h, h(l1 = c1), \ldots, h(ln = cn), \ldots h(l_i = c_i, l_j = c_j), \ldots, h(l_{i1}) = c_{i1}, \ldots, l_{in} = c_{in}, \ldots\}$

Then $F2$ can not be minimized. □

Theorem 3.2 *For every complete fact set F, if F contains no infinite decreased chain, then F can be minimized.* □

3.2 Unification of H-term

In Prolog, if two atoms can be unified then we can find the most general unifier. For example, $\{p(f(x), z), p(y, a)\}$ is unifiable, its most general unifier is $\theta = \{y/f(x), z/a\}$. How about unification of H-term in typed-based logic programming? From above discussion, we have already known that we can calculate any two ψ-terms greatest lower bound (glb) and least upper bound (lub). In this section, we study the substitution and unification of H-terms.

Suppose H-term $h = p(l1 = x : v1(l1 = y : v2); l2 = y : v2)$, $\theta = \{x|z : v12(l1 = z), y|z\}$, $v12 = glb(v1, v2)$, then θ is a well order-sorted substitution of H-term h, $h\theta = p(l1 = z : v12(l1 = z), l2 = z)$ and $h \leq_h h\theta$.

[11] Someone may prefer that $k = n$ and $l = m$. We don't keep these constraints. We think a H-term h is true iff there is a h', such that $h' \leq_t h$, h' is a fact. In our system, h and $\neg h$ can be both true. This treatment does not lead inconsistency. The definition of inconsistency in type-based logic program can be found in following definition.

[12] fly means something fly.

[13] move means something move.

[14] $fly(agent = x : ostrich)$ means something such that all ostrich can fly.

Definition 3.8 Well Order-Sorted Substitution of H-term h.

Say θ is a well order-sorted substitution of H-term h if

$\theta = \{x1|t1, ..., xn|tn, @y1|h1, ..., @ym|hm\}$

, where xi is noun-variable, $@yj$ is verb-variable, $1 \le i \le n, 1 \le j \le m$, and

1. For every noun-variable xi $(i = 1, ..., n)$, if it appears in h then $ti \le_t$ *subterm(xi, h)*;

   For every verb-variable $@yj$ $(j = 1, ..., m)$, if it appears in h then $subterm(@yj, h)$ $\le_h hj$;

2. If x appears in the scope of xj of h, and the address in xj is add, $(j = 1, ..., n)$ then x must be some xi and *subterm* $(tj, add) = ti$.

   If x appears in the scope of $@yj$ of h, $(j = 1, ..., m)$, and the address in yj is add, then x must be some xi and *subterm (hj, add) = ti.*

   If $@y$ appears in the scope of $@yj$ of h, $(j - 1, ..., m)$, and the address in yj is add then, then $@y$ must be some $@yi$ and *subterm(hj, add) =hi.*

We define $domain(\theta) = \{x1, ..., xn, @y1, ..., @ym\}$, and $\theta(xi) = ti$, $\theta(@yj) = hj$ for $1 \le i \le n, 1 \le j \le m$. □

Definition 3.9 Substitution.

Suppose θ is a well order-sorted substitution of H-term h.

Then we inductively define $h\theta$ as follows:

1. $@y : v\theta = @y : v$ if $@y \notin domain(\theta)$ else $\theta(@y)$, here $@x \in V_v$, $v \in T_v$.

2. Suppose $h = @y : v1(l1 = t1, ..., lm = tm, vl1 = vh1, ..., vln = vhn)$ then

   if $@y \in domain(\theta)$ then $h\theta = \theta(@y)$ else

   $h\theta = @y : v1(l1 = t1\theta, ..., lm = tm\theta, vl1 = vh1\theta, ..., vln = vhn\theta)$

□

Theorem 3.3 *For every H-term h's well order-sorted substitution θ, we have $h \le_h h\theta$.* □

Proof: We inductively prove this statement by H-term's length.

1. $h = @y : v$, then for every θ, it is easy to check that $h \le_h h\theta$.

2. Suppose $h = @y : v(l_1 = \psi_1, ..., l_n = \psi_n, hl_1 = h_1, ..., hl_m = h_m)$.

   Above statement hold for H-terms h_1, ... , h_m .

3. Let θ is $h's$ well order-sorted substitution, then

   (a) if $@y \in domain(\theta)$, then according to the well order-sorted substitution definition, we can get that $h \le_h h\theta = \theta(@y)$.

   (b) if $@y \notin domain(\theta)$ then $h\theta =$ $@y : v(l_1 = \psi_1\theta, ..., l_n = \psi_n\theta, hl_1 = h_1\theta, ..., hl_m = h_m\theta)$.
   We have $h_1 \le_h h_1\theta, ..., h_m \le_h h_m\theta$,
   Notices that $\psi_1\theta \le_t \psi_1, ..., \psi_n\theta \le_t \psi_n$,
   So it is easy to prove:
   $h \le_h h\theta$.

End our inductive proof.

Definition 3.10 Suppose *h1, h2* are two H-terms, and their variables are disjoint. Say they can be unifiable if there is a substitution θ such that $h1\theta = h2\theta \not\approx$ **t**. Say substitution θ is the most general unifiable, if for every substitution β of *h1, h2*, we have $h1\theta = h2\theta \le_h h1\beta = h2\beta$. □

3.3 Horn SLP's Semantics

Definition 3.11 H-terms are atoms or positive literals. Suppose h is an atom, then **not** h is negative literals. Literals are positive or negative literals, and we denote literals by li. Suppose $h0$ is an atom, $l1, ..., ln$ $(0 \le n)$ are literals. If every variable appears in them has the same definition, then $h0 \leftarrow l1, ..., ln$ is called an extended clause; When all li $(1 \le i \le n)$ are atoms, it is called normal clause or Horn clause. A SL program is a set of extended clause and a horn SL program is a set of normal clauses. □

Example 3.2 penal code in normal rules.

pen211: *death-by-negligence-during-business (a-object = @accident)* ←

@work : work (agent = X: person),
negligence (agent = X, object = @accident),

@accident : accident(agent=X), causality (cause = @accident, effect = @death), @death : die(agent=Y:person). □

This rule is adopted from the statement of Japanese legal panel code, Chapter XXVIII, Crime of Inflicting Injury by Negligence, Article 211. A person, who fails to use such care as is required in the performance of profession, occupation or routines and thereby kills or injures another, shall be regarded as death by negligence during business, and punished with penal servitude or imprisonment for not more than five years or a fine of not more than one thousand yen. The same shall apply to a person who, by gross negligence, injures or causes the death of another.

Following we will define extension of a normal program P under assumption set E.

Definition 3.12 $Ext(P, E)$

Let P be a program, E a II-term set. We define the new fact set $Ext(P, E)$ of P under E is:

$Ext(P, E) = \{h0\theta | h0 \leftarrow h1, ..., hn \in P$ and θ is its well order-sorted substitution $\{h1\theta, ..., hn\theta\} \subseteq Con(E)\}$. □

Definition 3.13 Suppose P is a program, we define a fact set sequence as following:

$Ext(P)^0 = \{\}$,

For every $i >= 0$,

$Ext(P)^{i+1} = Con(Ext(P)^i) \cup Ext\ (P, Ext(P)^i)$

The semantic of normal program P, called extension,denoted by $Ext(P)$, is defined by the set:

$Ext(P) = \bigcup_{i=0}^{i=\infty} Ext(P)^i$ □

Definition 3.14 Say program P is well-defined iff $Ext(P)$ can be minimized. □

Example 3.3 Suppose the relationship among noun and verb symbols are:

$Jim \leq_n person, Tom \leq_n person, bird \leq_n animal, c \leq_n bird, fly <_v move$

Suppose

$P1 = \{hit(agt1 = Jim, agt2 = Tom)\}$,

P2$= \{$*fly (agt= x:bird)* $\leftarrow$ *fly(agt= bird(son= x:bird)), fly (agt =bird (son =c))*$\}$.

Then

Ext$(\{\}$*, P1)* $= \{$*hit (agt1 = Jim, agt2= Tom), hit(agt1 = Jim), hit (agt2 = Tom), hit,* **t** $\}$.

Min (Ext$(\{\}$*, P1)* $=$

$\{$ *hit(agt1= Jim, agt2= Tom)* $\}$.

Ext$(\{\}$*, P2)* $= \{$*fly (agt =bird (son= c)), fly (agt =c), move(agt= c), move (agt= bird(son= c)), fly, move,* **t**$\}$.

Min(Ext$(\{\}$*, P2))* $=$

$\{$ *fly(agt= bird(son= c)), fly (agt =c)*$\}$

□

3.4 Resolution Method

Definition 3.15 Let goal G_i be $\leftarrow$ *A1, ..., Am, ..., Ak*, C_{i+1} be $A \leftarrow B1, ..., Bq$[15] and R a computation rule. Then G_{i+1} is derived form G_i and C_{i+1} using mgu θ_{i+1} via R if following conditions hold:

1. A_m is the selected atom given by the computation rule R.

2. $A\theta_{i+1} = A_m\theta_{i+1}$[16]

3. G_{i+1} is the goal $\leftarrow (A1, ..., A_{m-1}, B1, ..., Bq, A_{m+1} ..., A_k)\theta_{i+1}$. □

The definition of SLD-derivation, the properties of Soundness and Completeness of SLD-resolution are same to [18] page 36-56.

Theorem 3.4 Soundness

Let P be a type-based logic program, G is a goal $\leftarrow A1, ..., Ak$*, and R a computation rule. Suppose* $P \cup G$ *has a SLD-refutation of length n via R and* $\theta_1, ..., \theta_n$ *is the sequence of mgu's of SLD-refutation. Then we have* $\bigcup_{j=1}^{j=k} [A_j\theta_1...\theta_n] \subseteq Exp^n(P)$ □

Proof:

We prove this proposition according to the refutation length.

Step 1. When the prove length n is 1. Then it is obvious that k=1 and there is a rule $A \leftarrow$ and substitution θ such that $A\theta = A1\theta$. It is easy to prove that $A1\theta \in Ext^1(P)$, and hence $\{A1\theta\} \subseteq Exp^1(P)$.

Step 2. Suppose above proposition is true for length n.

Step 3. We assume goal $\leftarrow A1, ..., Ak$'s refute length is $n+1$.

Suppose the first goal is $A_m, 1 <= m <= k$, substitution is θ_1, the selected rule is $A \leftarrow B_1, ..., B_q$, then the second goal is

$\leftarrow (A_1, ..., A_{m-1}, B_1, ..., B_q, A_{m+1}, ..., A_k)\theta_1$.

According to the inductive step, we have

[15] We assume that variables of G_i and C_{i+1} are disjoint.

[16] = is syntax equal, not $\cong$.

$\bigcup_{j=1, j\neq m}^{j=k}[A_j\theta_1...\theta_{n+1}] \subseteq Exp^n(P)$

$\bigcup_{j=1,}^{j=q}[B_j\theta_1...\theta_{n+1}] \subseteq Exp^n(P)$

Since $A \leftarrow B_1, ..., B_q$ is a rule and $A\theta_1 = A_m\theta_1$, we consequently get that $A\theta_1...\theta_{n+1} = A_m\theta_1...\theta_{n+1}$. Hence $A_m\theta_1...\theta_{n+1} \in Exp^{n+1}(P)$

Thus we inductively prove above theorem.

Theorem 3.5 Completeness

Let P be a normal program and A an atom. Suppose $A \in Ext(P)$, then there exists an SLD-refutation of $P \cup \{\leftarrow A\}$ with the identity substitution as the answer substitution. □

Hint of Proof:

We inductively prove the following statement:

For any atom $A \in Exp^{n+1}(P)$, there must goal $\leftarrow A$'s refutation, whose length must less or equal to n.

3.5 Extended SLP's Semantics

In this section, we introduce the extended order-sorted logic programs semantics. Many people argue [6], [12] that, it is necessary to introduce two negations into logic programming in order to deal with the 'exception' knowledge in legal field and linguistic field. Like Gelfond and Lifschitz's stable semantics for non-sorted logic program, we propose the stable models for extended SL programs. It is also easy to see that stable model of normal Sl program is the same to its extension model.

Definition 3.16 Suppose $P1, P2$ are two H-term set, we say $P1 \leq_h P2$ iff for every $h \in P1$ there is a element $h' \in P2$ such that $h \leq_h h'$. When $P1 = \{p\}$, we denote $P1 \leq_h P2$ by $p \leq_h P2$. □

Definition 3.17 Stable Model.

Suppose P is an extended SL program. S is a miniable H-term set. We say *Trans(P, S)* is the translating program of P on assumption set S such that $Trans(P, S)$ is the least program satisfied the following condition:

Suppose $r = h_0 \leftarrow h_1, ..., h_n,$ **not** $h'_1, ...,$ **not** $h'_m \in P$, θ is the well order-sorted substitution of h. If for $i = 1, ..., m$ $h'_i\theta \not\leq_h Min(S)$, and $h'_i\theta \not\subseteq Con(S)$ then $h_0\theta \leftarrow h_1\theta, ..., h_n\theta \in Trans(P, S)$.

We denote *Sext(P, S)= Ext(Trans(P, S))*.

We say S is a stable model of P iff $Sext(P, S) = S$. □

Some of the important properties of traditional extended logic program do not hold in extended order-sorted logic program. For example, in traditional logic programming, $Sext(P, S)$ is anti-monotonic about S. But here, $Sext(P, S)$ is neither non-monotonic nor monotonic.

Example 3.4 Suppose

ostrich $\leq_n$ *bird* $\leq_n$ *animal*, *Jim* $\leq_n$ *person* $\leq_n$ *animal*

tweety $\leq_n$ *bird*, *fly-by-machine* $<_v$ *fly* $<_v$ *move*

The program P is:

{ *fly(agent=x:bird)* $\leftarrow$ **not** $\neg$ *fly(agent = x: bird)*,

fly-by-machine(agent=x:person),

$\neg fly(agent = x : ostrich)$ }

Suppose

$S1 = \{\mathbf{f}\}$

S2=Con{$\neg$ *fly(agent=x:ostrich), fly(agent = x:tweety), fly-by-machine (agent =x:person)}*

$Sext(P, S1) = Con\{\neg fly(agent = x : ostrich), fly - by - machine(agent = x : person)\}$

$Sext(P, S2) = S2$ □

Computing the stable extension of a SL program costs very high. So we must find another computing method or modified procedure semantics. One selection is: Argument Theory.

4 Priority Knowledge and Argument Theory

Generally, a legal knowledge base is large scale and may contain many conflict information. In order to select a more reasonable conclusion among incompatible conclusions, priority knowledge is needed. One kind of this priority knowledge is called standpoint, which says that some set of rules have the higher priority than the other set of rules. For example, *New law has priority over old law.* Under this standpoint, we can associate every rule with its promulgated date and dividing the law base according to the date. In this case, dividing the legal knowledge base into unit is necessary.

4.1 Unit and Standpoint

Legal rule base is organized by knowledge unit. Generally under one standpoint, a legal rule belongs to only one unit.

Definition 4.1 [Standpoint]
We define standpoint a computable relationship on units.

Suppose s is a standpoint, *u0, u1* are two units, if $< u0, u1 >\in s$, then we say u0 has a lower priority than u1 and, denote it by $u0 <_s u1$. □

For example, suppose decision-of-supreme-court is unit 1, decision-of-high-court is unit 2, then we define lex-superior is a standpoint, it contains only one element, that is unit 1 has higher priority than unit 2. In legal knowledge base there are many rules owned by the same unit. So from standpoint s, we can define priority relationship among rules.

Definition 4.2 Suppose s is a standpoint, r_i (i=1,2), are rules (normal or extended), r_i belong to unit u_i. Say r2 is higher priority over r1 under standpoint s, denoted by $r1 <_s r2$ if $u10 <_s u20$ and $u20 \not<_s u10$. □

There are many standpoints and people may use different standpoint in practice. Besides, different standpoint stands for different division of legal rules. Following, we give some common standpoints among logic programming field.

The first standpoint is hard-soft standpoint. We divide all rules into two parts. One is soft rule, in which the **not** is appeared in the right body. The rest is hard rule. One standpoint which has been widely used is that hard rule has the high priority than soft rule. It means that a conclusion inferred from hard rule is more believable.

Definition 4.3 Hard-Soft Standpoint (h-s).
Every hard rule has high priorities then soft rule □

We use h-s standpoint to analyze following example.

Example 4.1 $a \leq_n b$, $a \leq_n c$, $d \leq_n b$.
P contains two rules.
r1: $p(l = x : b) \leftarrow$ **not** $\neg ab(l = x : b)$, Generally b has property p.
r2: $\neg p(l = x : c)$, c has the properties $\neg p$. □

We can get $p(l = x : a)$ and $\neg p(l = x : a)$ respectively. In order to get $p(l = x : a)$, we use rule $r1$; In order to infer $\neg p(l = x : a)$, we use rule 2. Since $r2$ keeps high priority, we believe $\neg p(l = x : a)$ instead of $p(l = x : a)$.

Definition 4.4 Suppose r1 and r2 are rules like:
r1: symbol1 h1 $\leftarrow \ldots$
r2: symbol2 h2 $\leftarrow \ldots$
Here, {*symobol1, symbol2*} ={ ¬ } *and symbol1=nil or symbol2=nil.* if $h2 \leq_h h1$ then we say r1 is a more special rule than r2.

Generally, we say special rules should have a more higher priority then general rule. We call this standpoint is *s-g* standpoint. □

Let's analyze another example;

Example 4.2 *fly(agent= x:bird)* ← **not** ¬ *fly(agent= x:ostrich)*
¬ *fly(agent= x:ostrich)* ← **not** *fly(agent= x:ostrich)*
ostrich $\leq_n$ *bird, tweety* $\leq_n$ *ostrich* □

We can see that under standpoint s-g, we can get $\neg fly(x : tweety)$.

Let's conclude this subsection by a simplified example of legal field. In a legal knowledge base, there are are many kinds of standpoint and each one corresponds to different division of the knowledge base. For example, suppose we have following 5 rules where *penal0, penal1, penal3* are formal legal code. *case1* is a case rule, and *fact1* is a fact. Under standpoint of *social-justice*, we place the *case1* rule in the high priorities, so *social-justice*={ {*penal1*} ≤ {*case1*}} . However, under *protection-of-minor* standpoint, we should take the *penal0* rule in the high priorities, so *protection-of-minor* = { {*case1*} ≤ {*penal1*} }.

penal0:: punishable(a-object = X, goal = Y) ← condition-of-crime (a-object =act(agent=X: person), goal = Y: crime).

penal1:: ¬ *punishable(a-object = X) ← act(agent = X: person(age=[0...13])).*

penal2:: condition-of-crime (a-object = @action, goal = crime-of-homicide) ← @action: kill(a-object =act(agent= X:person)).

case1:: punishable(a-object =X, goal= crime-of-homicide) ← kill(agent= X: person(age = 13)) fact

fact1:: kill(agent= tom(age= 13)).

It is easy to check that for query

?- punishable(a-object = tom (age = 13), goal= crime-of-homicide)

The result will be *YES* in the case of *social-justice* standpoint and *NO* in the case of *protection-of-minor*.

In the following section, we will discuss the argument tree in HELIC-II.

4.2 Argument Tree

For a given goal g, our task is to find an *argument tree* to support it.

Definition 4.5 Argument Tree

Given an assumption set S of SL program P, we inductively define argument tree for goal G as following:

For every rule

r=$u0 : p \leftarrow p1, ..., pk$, **not** $p'1$,...,**not** $p'j$, $0 \leq k, 0 \leq j$, Suppose

1. There is a well order-sorted substitution θ of r, such that $p\theta = G$.
2. For $i = 1, ..., k$, tr_i is an argument tree for root node $pi\theta$.
3. For $i = 1, ..., j$, $p'_i\theta \not\leq_h Min(S), p'_i\theta \notin Con(S)$

Then

$tree =< G, \theta, r, \{tr_1, ..., tr_k\}, \{p1\theta, ..., pk\theta\}, \{p'1\theta, ..., p'j\theta\} >$ is an argument tree of goal G.

We define

$$Tr(tree) = \{tree\} \cup \sum_{i=1}^{i=k} Tr(tr_i)$$

$$Sup(tree) = \sum_{i=1}^{i=k} \{p_i\theta\} \cup \sum_{i=1}^{i=k} Sup(tr_i)$$

$$Ass(tree) = \sum_{i=1}^{i=j} \{p'_i\theta\} \cup \sum_{i=1}^{i=k} Ass(tr_i)$$

□

Usually, goal G's argument tree is denoted by $tr(G)$. When $S = \{\mathbf{f}\}$, we say $tr(G)$ is P's skeptical argument tree. When $S = \{\}$, we say $tr(G)$ is P's credulous argument tree.

Theorem 4.1 *For every statement p, $tr(p)$ is an argument tree in program P under assumption set S iff $p \in Sext(P, S)$.* □

Definition 4.6 Say argument tree $tr(p)$ under assumption set F is rational if

1. $Ass(tr(p)) \cap Con(F \cup Sup(tr(p))) = \{\}$
2. For every $q \in Ass(tr(p))$, $q \not\leq_h Min(F \cup Sup(tr(p)))$. □

From this definition, we can see that

Corollary 4.2 Suppose $tr(p)$ is a rational argument tree in program P under assumption F, then $tr(p)$ is also a rational argument tree in program P under assumption set $F \cup sup(tr(p))$. □

Definition 4.7 Rational Assumption Set.

Say assumption set F is a rational assumption set under program P, if for every $p \in F$, p has a rational argument tree under F. Say rational assumption set F is maximal rational assumption set, if for every statement p, if p has a rational argument tree under F, then $p \subseteq F$. □

Corollary 4.3 Assumption set F is a rational assumption set under program P iff $F \in Sext(P, F)$;

Assumption set F is a maximal rational assumption set under program P iff $F = Sext\ (P, F)$ iff F is P's stable model. □

Given a goal G, it may have no one or have many rational argument trees under certain assumption set. If it has many rational argument tree, we think

1. It is impossible and impracticable to list all the possible argument of G.
2. Given an argument tree $tr(G)$ of G, There may be some anti-argument tree $tr(\neg p)$ of G such that p is either G or $tr(G)$'s supporting node. In this case, we need some standards to compare which argument is more powerful.
3. In case 2, if $tr(\neg p)$ is more powerful, then in order to continue to support G, we need to find argument tree of G such that it may be more powerful then $tr(\neg p)$ according the precious standard.
4. Our ideal target is to find the most powerful argument tree of G.

Then what is the most powerful argument tree for a given goal G? In following section, we will give the formal concepts of defeat and so on. We argue that the most argument tree for goal G is the justifiable argument tree.

4.3 Argument Theory

In the following we consider the relationship among any two rational arguments on program P and standpoint s.

Definition 4.8 Directly Defeat

Suppose *tr(p1)*, *tr(p2)* are two arguments, we say *tr(p2)* directly defeats *tr(p1)* if *p2* is a counter argument of *p1*, and the top rule of *tr(p2)* takes higher priority over the top rule of *tr(p1)* under standpoint s. □

Compared with the attack definition in [25], we can see if *tr(p2)* directly defeats *tr(p1)* under standpoint s, then *tr(p2)* attacks *tr(p1)*. However it is not true conversely.

Definition 4.9 Defeat

Suppose *tr(p1)*, *tr(p2)* are two argument trees. We say *tr(p2)* defeats *tr(p1)* under standpoint s if

1. *tr(p2)* directly defeats *tr(p1)* under standpoint s or
2. *tr(p2)* defeat a sub-argument tree of *tr(p1)* under standpoint s . □

Definition 4.10 Suppose *tr(p)* is an argument tree of program P,

1. We say *tr(p)* is justifiable argument under standpoint s, if there is no argument *tr(q)* such that *tr(q)* defeats *tr(p)* under s.

   Say H-term p is a justifiable conclusion if p has a justifiable argument tree.

2. *tr(p)* is defeated under standpoint s, if it is defeated by a justifiable argument under s.

   Say p is defeated if either p has no argument tree or p's every argument tree is defeated.

3. *tr(p)* is merely plausible under standpoint s, if it is neither justifiable nor defeated under s.

   Say p is plausible if p is neither justifiable nor defeated

□

When standpoint s is empty, we simply call argument $tree$ satisfies property X under s by argument tree $tree$ satisfies X.

Example 4.3 Consider following program.

$P=$ { *fly(x:bird)* ← **not** *ab(x: bird)*, ¬ *fly(x:ostrich)*, *ab(x:ostrich)*}.

Suppose the relationship is *ostrich* $\leq_n$ *bird*, *tweety* $\leq$ *bird* , assumption set $F =$ { ¬ *fly(x: ostrich)*, *ab(x:ostrich)*}.

Then *fly(x:tweety)* is a justifiable conclusion, *fly(x: bird)* is a defeated argument. □

Theorem 4.4 *If tr(p) is justifiable under standpoint s, then every sub-argument of tr(p) is also justifiable under s.* □

5 Comparison with others work

In this section, we give a brief comparison with others work include Makoto Haraguchi's order sorted feature legal reasoning system, Ait-Kaci et al's $Login$, $Life$ as well as Prakken's argument theory work.

5.1 Haraguchi's Legal Reasoning System

In paper [16], Makoto Haraguchi proposed a symbol system in which the legal rules are represented. A partially ordered set $(S, \leq)$ is assumed, where S is the set of sort symbols. A function symbol set F is also assumed, in which every function f is associated with a sort specification: $s1, ..., sn \rightarrow s$. A predicate symbol set P is also assumed, in which every predicate p is also associated with a sort specification $s1, ..., sn$.

$(S, \leq)$ correspond to our noun and verb lattice $(T2, \leq_h)$ and $(T1, \leq_t)$. We also assume that $Fset$, being a function symbol set, belongs to our noun set $T2$. Generally we assume $\perp \leq_t f \leq_t Fset \leq_t \top$.

Every variable appeared in [16] is associated with a type. For example, if x is a variable, then $[x]$ is a type.

We can see how translate a well-sorted first order term t of [16] into out ψ-terms.

1. $t = x$, x's type is s, $[t]$=s,

   $trans(t) = x : s$

2. if $f \in F$, $f : s1, , sn \rightarrow s$, t1, ..., tn are well-sorted first order terms and for $i = 1, ..., n$, $[ti] \leq si$, then $t = f(t1, .., tn)$ is also a well-formed first order term.

   t's translation is:

$trans(t) = s(functionname = f, 1 = trans(t1), ..., n = trans(tn))$

Suppose t is a well-sorted term, $\theta = \{ xi / ti \mid xi$ is a variable and ti is a well-sorted term $\}$ is a substitution. Say θ is a substitution of t, if xi appears in t, then $[ti] \leq [xi]$.

Suppose θ is a substitution of well-sorted term t, then $t\theta$ is also well-defined.

It is easy to prove that:

Theorem 5.1 *Suppose θ is a substitution of well-sorted term t. Let $\eta(\theta) = \{xi : [xi] / trans(ti) \mid xi/ti \in \theta\}$, then we have $trans(t\theta) = trans(t)\eta(\theta)$.* □

So substitution of [16] can be equally realized and expressed in our frame.

Suppose E is a set of well-formed terms, we say E can be unifiable if there is a substitution of E, θ such that for every two element $t1$ $t2$ of E, $t1\theta = t2\theta$.

Theorem 5.2 *Suppose E is a set of well-formed terms, if E can be unifiable then the ψ-term set $\{trans(t) | t \in E\}$ can also be unifiable.* □

According to Makoto Haraguchi's arguments, every sort in S can be divided into two categories. One is event type, corresponding to our verb type, another is object type, corresponding to our noun type. For every sort type s, either $s \leq event$ or $s \leq object$. In fact, he also introduce the third object set, label set. It is clear that his idea about the concepts world is similar to ours.

He assumes further that: Every event sort should be associated with only one function. For example, Contact is associated with a function *contract-f:*

person, person, object → contract.

He also introduce the *label* concepts, which equals to our label set of L_n.

For example, term *contract-f(X:person, Y: person, Z: object)* can be rewritten as:

contract-f(agt1= X:person, agt2 = Y:person, obj = Z:object).

This form of writing is more close to our ψ-term:

contact(functionname = contact-f, agt1= X:person, agt2= Y:person, obj= Z:object)

Generally, a well-formed label term $t = f(a1 = s1, ..., an = sn)$ can be transformed to our ψ-term:

$trans(t) = s(functionname = f, a1 = trans(s1), ..., an = trans(tn))$, where $[t] = s$, and s is a event type (corresponding our verb type). Hence if well-formed term t is an event term, like *contract-f (X:person, Y: person, Z: object)*, then $trans(t)$ is a H-term in our paper.

In [16], two predicates, *attr* and *oc* have been introduced. *attr* is called attribute predicate, *oc* stated that some statement has happened. These two predicates can be easily implemented in our frame work.

Suppose t is a H-term (also true for ψ-term), say $attr(t, l, x : s)$ is true iff $type_t(l) = s$ and V-t(l)=x.

Say $oc(t)$ iff t is a fact of our program. (Notice that: We take H-term as true statement).

Now, we conclude the comparison through following example:

Example 5.1 Suppose the event sort hierarchy includes:

contract $\leq$ *justice-act* $\leq$ *lawful-act* $\leq$ *human-act* $\leq$ *event*

A fact in the knowledge base is:

$oc(contract\text{-}f(a, b, c))$

then the query, a well-sorted goal clause $\leftarrow oc(X : lawful\text{-}act)$ will answer 'Yes' and give a substitution:

$\theta = \{ X{:}lawful\text{-}act | contract\text{-}f(a, b, c)\}$.

Above process can be easily implemented in our system:

The verb sort hierarchy includes:

contract $\leq_v$ *justice-act* $\leq_v$ *lawful-act* $\leq_v$ *human-act* $\leq_v$ *event*

A fact in the knowledge base is:

contract (functionname = contract-f, agt1= a, agt2= b, obj= c).

For the query *?X:lawful-act*, the answer is 'yes',and the substitution is:

η ={ *X:lawful-act* | *contract(functionname = contract-f, agt1=a, agt2=b, obj=c)*}.

Notice that *contract(functionname = contract-f, agt1 =a, agt2 =b, obj =c)* $\leq_h$ *X: lawful-act.* □

5.2 Ait Kaci et al's Login and Life

Knowledge Representation proposed in this paper has a close relationship with the work of *Login* where *Prolog*'s first-order term is replaced by ψ-term. An example of *Login* program is:

student $\leq_n$ *person;* {*peter, paul, mary*} $\leq_n$ *student;* { *goodgrade, badgrade*} $\leq_n$ *grade; goodgrade* $\leq_n$ *goodthing;* {*a, b*} $\leq_n$ *goodgrade;* {*c, d, f*} $\leq_n$ *badgrade*

likes(x:person, x); likes(peter, mary); likes(person, goodthing); got(peter, c); got (paul, f); got (mary, a)

happy(x:person) $\leftarrow$ *likes(x, y), got(x, y);*

happy (x:person) $\leftarrow$ *likes (x, y), got (y, goodthing).*

There are two reasons *mary happy*, mary likes goodthing and she got a goodthing (a grade), mary likes herself and herself got a good thing. In [9], a detail procedure of the resolusion has been given why we get $x = mary$ when we inquire ? $-$ $happy(x)$.

We can rewrite above example in our frame. Here we only write the logical rules:

happy(agt=x:person) $\leftarrow$ *likes(agt=x, obj=y:* $\top$*), got(agt=x, obj=y);*

happy(agt=x:person $\leftarrow$ *likes(agt=x, obj=y: person), got(agt=y, obj=goodthing)*

When we inquire $? - happy(agt = x : person)$, obviously, we will get an answer $x = marry$.

We can see that

1. In our work, we also take ψ-term as the first order logic term of *Prolog*. Further, we introduce the order sorted inheritance in predicates, and proposed the H-term concept. We allow predicate symbols can have non-fixed arity and using verb label to distinguish the argument.

2. *Login* gives an informal description about its resolusion method through some examples. The formal resolusion procedure of *Login* is unclear. In this paper, we use type-based finite automata describing ψ-term and give a formal definition about normal Program's resolusion and get the complete result.

3. More importantly, we introduce the operator **not** into SL program, and make the SL program can deal with common sense knowledge and reasoning. Further, defensiable reasoning, argument reasoning have been introduced.

Recently, based on *Login*, a more powerful experimental programming language, $Life$ has been proposed by Hassan Ait-Kaci and Andreas Podelski [10] [11]. $Life$ is an acronym of 'Logic , Inheritance, Function and Equations'. In [10], three different syntactic presentations about ψ-term have been proposed. They are normal OSF-term, rooted solved OSF-clause and OSF-graph. Here, we show that our type-based finite automata has a close relationship with OSF-graph and they are semantically equivalent.

Definition 5.1 Refer Original Definition [10], page 215. This definition is a simplified and equivalent one.

An order-sorted feature graph on signature $\Sigma_n =< L_n, T_n, V_n >$, is a finite directed labeled graph $g = (V, E, \lambda_V, \lambda_E, x0)$, where $V \subset V_n, E \subset V \times V$, $\lambda_V : V \rightarrow T_n$, $\lambda_E : E \rightarrow L_n$. $x0 \in V$ is a distinguished node called the root, such that:

1. each node x of g is labeled by a non-bottom sort, that is $\lambda_V(x) \neq \perp$.

2. no two edges outgoing from the same node are labeled by the same feature, i.e. if $\lambda_E(< x, y >) = \lambda_E(< x, y' >)$ then $y = y'$

3. every node lies on a directed path starting at the root (g is connected). □

Obviously, if $auto =< L, V, \pi, x0, type >$ is a connected type-based finite automata on signature Σ_n, then we can construct a OSF-graph *g-a(auto)=(V, E,* $\lambda_V, \lambda_E, x0)$ such that:

1. $E = \{< x, y > |\{x, y\} \subseteq V$, there is a $l \in L$ such that $\pi(x, l) = y\}$.

2. $\lambda_V = type$.

3. For every $< x, y >\in E$, $\lambda_E(< x, y >) = l$, here $\pi(x, l) = y$.

It is easy to check that for every connected type-based finite automata *auto*, $g - a(auto)$ is OSF-graph. Conversely, for every OSF-graph $g = (V, E, \lambda_V, \lambda_E, x0)$, we can also construct a connected type-based finite automata *a-g(g)=<L, V,* $\pi, x0, type >$, such that:

1. $L = \lambda_E(E)$.

2. $type = \lambda_V$.

3. For every $x \in V$, $l \in L$, $\pi(l, x) = y$ iff $< x, y >\in E$ and $\lambda_E(< x, y >) = l$.

Theorem 5.3 *The correspondence*

g-a: $CAU^{\Sigma_n} \rightarrow D^{\Sigma_n}$ *and*

a-g: $D^{\Sigma_n} \rightarrow CAU^{\Sigma_n}$

between OSF-graphs and connected finite automata are bijections. Namely,

a-g $\circ$ *g-a*$=1_{CAU^{\Sigma_n}}$ *and*

g-a $\circ$ *a-g*$= 1_{D^{\Sigma_n}}$ □

6 Prakken's Argument Theory

In section 5 of this paper, we discuss the argument framework in our legal reasoning system. In fact, there are lots of works on this aspect such as Pollack(1987), Prakken (1991, 1993), Vreeswijk(1991)and Simari and Loui (1992). Main feature on this aspect is the modeling of inconsistency-tolerant and defensiable reasoning as constructing and comparing argument for incompatible conclusion. From the formal aspect, the main differences among these research are:

1. Difference in knowledge, especially the defensiable knowledge representation.

Most of these researches build their knowledge language on FO language. But the difference lies on the defensiable knowledge. For example, in Prakken (1993), the defensiable knowledge is represented by default rule. In our frame, the knowledge is represented by extended order-sorted rule.

2. Difference in definition of incompatible knowledge.

Generally, in non-sorted logic system, it is agreed that if p and $\neg p$ can both be concluded from a knowledge base, we say this knowledge base contains incompatible information. In order sorted knowledge base, especially in our frame work, the situation is a little different because of the degree of information completeness of a statement (H-term). For example, *Jim hit Tom at room one* and *Jim does not hit Tom on Sunday* can respectively conclude *Jim hit Tom* and *Jim does not hit Tom*. Since the later two conclusions come from different, complete and conistency information, so we do not regard them as incompatible conclusions. Hence, although fact set $A = Con(\{\ hit\ (agt = Jim, obj = Tom, place = room\text{-}one), hit(agt = Jim, obj = Tom, time = sunday)\})$ contains $hit(agt = Jim, obj = Tom), \neg\ hit\ (agt = Jim, obj = Tom)$, A is still consistent.

3. Difference in argument definition.

An argument should be defined rational. Here *rational* means difference in different paper. In our opinion, an argument *tree* of program P and assumption set F is rational, (well-defined) if:

Every default assumption should neither be in the conclusion set of assumption and supporting node nor contains more information then the most complete (minimal H-term) statement of assumption and supporting node. That is:

$Ass(tree) \cap Con(F \cup Sup(tree)) \neq \{\}$

For every $q \in Ass(tree)$, $q \not\leq_h Min(F \cup Sup(tr(p)))$.

It should be also noted that our definition about defeat does not conclude the 'floating' conclusion. For example, in program $P = \{\ c \leftarrow p,\ c \leftarrow q,\ p \leftarrow \textbf{not}\ q,\ q \leftarrow \textbf{not}\ p\}$, c is a justifiable conclusion under standpoint $\{\}$.

4. Difference in the classifying of argument tree.

Prakken classify the different kinds of argument trees through justifiable argument, defeated argument and plausible argument. We agree to his proposal and refine the concepts in our frame work. The motivation behind these refinement is that the argument process is the process to reach a stable conclusion. We prove that the stable conclusion of the program must be the justifiable conclusion under empty priority.

7 Conclusion

Now we conclude our paper. Briefly, we review the contents and main conclusions of this paper.

First, we think order-sorted logic programming is a powerful and profit tool to deal with legal problems. Among the many approaches of SLP, we argue that label-based SLP, that is ψ-term based SLP, is more proper because of its knowledge representation ability.

We analyze the ψ-term theory through type based finite automata and analyze the glb and lub algorithms. The complexity is nearly liner of input ψ-term's length, this conclusion is proved by Ait-Kaci. Later, we prove that the connected type-based finite automata is equal to the concept of *Life*'s OSF-graph.

Then we extend the label based approach to the predicate set and proposed the H-term concept. H-term helps us to express the complex statement such as *Tom hit Jim at classroom one on Saturday* and hence get *Tom hit Jim at classroom one.* We analyze the logical relation between these complex statement, and get the priorities according to the information completeness. Simply, we claim that a complete statement can conclude incomplete statement.

Based on these discussion, we form the new SLP program, which a Horn SLP is a SLP without operator **not** , and an extended SLP program contains both two negation operators, negation as failure **not** and negation as logic $\neg$. We first discuss the Horn SLP program's denotational and operation semantics, then extend it to extended SLP program through Gelfond and Lifschitz's stable model.

Then we discuss the priority knowledge, standpoint, among legal rules. By standpoint, we can classified argument into three categories, justifiable, defeated and plausible. It is easy to see that, under standpoint {}, a stable conclusion must be a justifiable conclusion.

Acknowledgment

First author's research has been supported by STA fellowship of Japan. The authors would like to thank Prof. Koichi Furukawa, Prof. Kuniaki Mukai of Keio University and Dr. Kazumasa Yokota of ICOT for giving useful comments on this work.

References

[1] Attardi, G., Simi, M. (1984) 'Metalanguage and Reasoning across Viewpoints', in Proc. 6th ECAI, Pisa, Italy, 315-324.

[2] C. Beierle, U. Hedtstuck, U. Pletat, P.H. Schmitt and J. Siekmann, 'An order-sorted logic for knowledge representation systems', Artificial Intelligence 55 (1992) 149-191.

[3] Bob Carpenter, the Logic of Typed Feature Structure, Cambridge University Press, 1992.

[4] Aho, Hopcroft, Ullman, The Design and Analysis of Computer Algorithms, Addison Wesley.

[5] Chang C.L. and Lee R.C.T., Symbolic Logic and Mechanical Theorem Proving, Academic Press, New York, 1973.

[6] Gelfond, M. and Lifschitz, V., The Stable Model Semantics for Logic Programs, Proceedings of the Fifth Internaitonal Conference and Symposium on Logic Programmming, Kowalski, R. A. and Bowen, K. A. (eds.), Volume 2, 1070-1080.

[7] Peter Jackson, Hun Reichgelt, Frank van Harmelen (eds.), Logic-Based Knowledge Representation, MIT Press Series in Logic Programming, 1989.

[8] Ait-Kaci,H., An Algebraic Semantics Approach to The Effective Resolution of Type Equations. Theoretical Computer Science 45(1986) 293-351

[9] Ait-Kaci, H.,Nasr, R., LOGIN: A Logic Programming Language with Build-in Inheritance. The Journal of Logic Programming, 3, 1986, 185-215.

[10] Ait-Kaci, H. and Podelski Andreas, Towards A Meaning of Life. J. of Logic Programming, 1993:16:195-234.

[11] Ait-Kaci, H. An Overview of Life, Lecture Notes Computer Science, 504, 42-58, 1991.

[12] Roberta A. Kowalski, The Treatment of Negation in Logic Programs for Representing Legislation. Proceedings of the second Interantional Conference on Artificial Intelligence and Law, 11-15.

[13] Kuniaki Mukai, Partially Specified Term in Logic Programming for Linguistic Analysis, Proceedings of the International Conference on FGCS, 1988, edit, ICOT.

[14] Kuniaki Mukai, CLP(AFA): Coinductive Semantics of Horn Clause with Compact Constrints. pp. 179-214, Situation Theory and its Applications, Volume 2, (eds.) Jon Barwise et al. CSLI.

[15] Weidong Chen, Michael Kifer, David S. Warren, HiLog: A first-Order Semantics for Higher-Order Logic Programming Constructs. In 2-nd Intl. Workshop on Database Programming Languages, Norgan Kaufmann, June 1989.

[16] Haraguchi, M., A Reasoning system for Legal Analogy, Technical Report, SSTR-94-1, Department of System Science, Interdisciplinary Graduate School of Science and Technology, Tokyo Institute of Technology. To appear in Machine Intelligence, vol.14, Oxford University Press.

[17] R.T. Kasper and W.C. Rounds, A Logical Semantics for Features Structures, In Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics, 1986.

[18] Lloyd, J.W. Foundation of Logic Programming, Spring-Verlag, 1984.

[19] Marek, Truszczynski, Nonmonotonic Logic, Context-Dependent Reasoning, Springer-Verlag, 1991.

[20] K. Nitta, et al. HELIC-II: Legal Reasoning System on the Parallel Inference Machine, New Generation Computing, 11 (1993) 423-448.

[21] K. Nitta, et al.,Knowledge Representation Language for New HELIC-II, Technical Report, Personal Communication. 1994.

[22] K. Nitta, et al., A Legal Reasoning System: New HELIC-II. Proceedings of the International Symposium on Fifth Generation Computer Systems, 1994.

[23] Pollock 1987, Defeasible Reasoning, Cognitive Science 11 (1987), 481-518.

[24] Henry Prakken, A tool in modeling disagreement in law: preferring the most specific argument. Proceedings ICAIL-1991, Oxford, ACM Press, 1991, 165-174.

[25] Henry Prakken, A Logical Framework for Model Legal Argument, Annuals of Mathematics and Artificial Intelligence, Volume 9, 1993, page 93-132.

[26] Reynolds, J. C., Transformational systems and the Algebraic Structure of Atomic Formulas, In: D. Michie (ed.), Machine Intelligence 5, Edinburgh, U. P., 1970.

[27] Gert Smolka, Hassan Ait-Kaci, Inheritance Hierarchies: Semantics and Unification, Journal of Symbolic Computation (1989)7, 343-370.

[28] Satoshi Tojo, Hiroshi Tsuda, QUIXOTE as a Tool for Natural Language Processing, International Conference on Tools with Artificial Intelligence, 1993, Boston.

[29] G.R. Simari and R. Loui, 'A mathematical treatment of defensiable reasoning and its implementation', Artificial Intelligence 53, (1992) 125-157, Elsevier.

[30] Stoughton, A., Fully Abstract Models of Programming Languages, Research Notes in Theoretical Computer Science, Pitman, 1988.

[31] Vreeswijk 1991, Abstract argument systems, Proceedings of the first World Conference on th fundamentals of AI, Paris, 1991, 501-510.

[32] Kazumasa Yokota, Hiroshi Tsuda, Yukihiro Morita, Specific Features of a Deductive Language QUIXOTE, In the proceedings of ACM SIGMOD'93 Workshop on Combining Declarative and Object-Oriented Databases, 89-99.

[33] Kazumasa Yokota, Hideki Yasukawa, Towards an Integrated Knowledge-Base Management System, Overview of R&D on Databases and Knowledge-Bases in the FGCS Project, Proceedings of the International Conference on fifth Generation Computer Systems 1992, 89-131.

[34] Kazumasa Yokota and Akira Aiba, A New Framework of Very Large Knowledge-Bases —from database and constraint logic programming points of view (draft), In the proceedings of International Conference on Building and Sharing of Very Large-Scale Knowledge Bases'93, 196-200.

A Samples of Query

We assume that for goal $?G$, if answer is no, then $G \notin Sext(P)$ else suppose the return substitution is θ, then $G\theta \in Sext(P)$. Here, we do not consider the H-variable substitution.

The relationship between nouns are:

Jim $\leq_n$ *person, Tom* $\leq_n$ *person, person* $\leq_n$ *animal, ostrich* $\leq_n$ *bird, tweety* $\leq_n$ *bird, bird* $\leq_n$ *animal, Tom* $\leq_n$ *japanese, Tom* $\leq_n$ *male, japanese* $\leq_n$ *person, male* $\leq_n$ *person*

The relationship between verbs are:

fly-by-wings $<_v$ *fly, fly* $<_v$ *move, kill-by-gun* $<_v$ *shot, kill-by-gun* $<_v$ *kill, shot* $<_v$ *action, kill* $<_v$ *action* , *hit* $<_v$ *action*

Following are some simple program and inquire examples.

1. $fly(l = x : bird)\ ?fly(l = x : animal)$ Answer is: Yes, $x : animal/x : bird$

2. $fly(l = x : bird)\ ?fly(l = x : tweety)$ Answer is: Yes.

3. $hit(agt = x : Jim, obj = y : Tom, place = z : class{-}room)\ ?hit(agt = x : Jim, obj = y : Tom)$ Answer is: Yes.

4. $hit(agt = x : Jim), hit(obj = y : Tom)$ $?hit(agt = x : Jim, obj = Tom)$ Answer is: No.

5. $hit(agt = x : Jim, obj = y : Tom)$ $?hit(agt = x : Jim), hit(obj = y : Tom)$ Answer is: Yes.

6. $fly(agt = x : bird)$ $?move(agt = x : bird)$ Answer is: Yes

7. $move(agt = x : animal)$ $?fly(agt = x : animal)$ Answer is: No.

8. $\neg fly(agt = x : ostrich)$, $?\neg fly-by-wing(agt = x : ostrich)$ Answer is: Yes.

9. *know(agt=x:Mary, object = kill-by-gun(agt =Jim, obj =y:Tom))*

   ?know(agt= x:person, object= kill(agt= Jim, obj= y:person)) Answer is: Yes, *x:person / x:mary, y:person / y:Tom*

10. *know(agt= x:person, obj= fly(agt= y:bird))*

    ?know(agt= x:Jim, object= fly(agt= y:tweety)) Answer is: Yes.

11. *know(agt=x:person, obj=fly(agt=y:bird))*

    ?know(agt= x:animal, object= fly(agt= y:animal))

    Answer: Yes, $\theta = (x : animal/x : person, y : animal/y : bird)$

    ?know(agt= x:person, object= move(agt= y:bird))

    Answer: Yes

    ?know(agt= x:animal, object =move(agt= y:ostrich))

    Answer: Yes, $\theta = (x : animal/x : person)$

    ?know(agt= x:person, object =fly-by-wings (agt= y:bird))

    Answer: No.

B A Semantics of ψ-term

As H. Ait-Kaci [9]'s doing, we give here a 'type-as-set' denotational semantics of the ψ-term.

Suppose U is the universe of objects where every type symbols t of T_n is associated with a subset of U.

Suppose I is the interpretation, such that:
I: $< T_n, \leq_n > \rightarrow < 2^U, \subseteq >$
$I[\top] = U$, $I[\bot] = \{\}$ ......(1)
For every $s, t \in T_n$, we have
$s \leq_n t \Rightarrow I[s] \subseteq I[t]$ ......(2)
Furthermore, if glb and lub exist, it is desired that
$I[glb(s,t)] = I[s] \cap I[t]$,
$I[glb(s,t) = I[s] \cup I[t]$. ......(3)
For every label, we assume that every label is associated with a function of following:
$I[l] : U \rightarrow U$
For every label sequence l, we define:
$I[\epsilon] = id$, id is the identification function on U.
$I[l.\alpha] = I[l].I[\alpha]$, here $l \in L_n$, α is a sequence of labels.
The we define the denotational semantics of ψ term such that every ψ-term t is associated with a subset of U.
First we define the semantic of ψ-term without variable.
I(v(l1= t1, ..., ln= tn)) $=\{x \in$ *I(v)* $|$ *I(l1)x* $\in$ *I(t1), ..., I(ln)x* $\subset$ *I(tn)*$\}$. ......(4)
Suppose t is a ψ-term with variable and t' is the ψ-term deleting all variable of t.
I[t]= $\{ x \in$ *I[t']* $|$ *For every l1, l2* $\in$ *domain(t)*, if $V_t(l1) = V_t(l2)$ then $I[l1]x = I[l2]x\}$. ......(5)
It is easy to prove that:

Proposition B.1 Suppose I is an interpretation, If I satisfies the conditions (1) and (2) then
for every ψ-term $t1, t2$, if $t1 \leq_t t2$ then $I[t1] \subseteq I[t2]$. □

C Algorithms

$\Psi(auto)$ is *auto-psi(auto).*

```
=========================
Procedure auto-psi($< L, V, \pi, x0, type >$)
  begin
  call auto-psi1($\{\}$, x0, $< L, V, \pi, x0, type >$)
  end.
Procedure auto-psi1(X, y, $< L, V, \pi, x0, type >$)
  begin
  if $y \in X$ then return y else
    begin
    list $\leftarrow$ $nil$;
    For every $l \in L$ do
    if $\pi(l, y)$ is defined then
    $list \leftarrow append(list,$
                    $l = auto\text{-}psi1(X \cup y, pi(l, y), auto))$
```

```
      return($x : type(x)list$)
      end
    end
  ======================
  $\Delta(t)$ is *psi-auto(t)*.
  ======================
  Procedure psi-auto(t);
    begin
    call psi-auto1($\{\}, t$)
    end
  Procedure psi-auto1(X, t);
    begin
    If $t$ is a simple $\psi$-term, assume $t = x : s$
    then
      if $x \in X$ then return nil
      else return $< \{\}, \{x\}, , \{< x, s >\}, x >$;
    else assume $t = x : s(l1 = t1, ..., ln = tn)$
      init $\leftarrow < \{l1, ..., ln\}, \{x\}, \sum_{i=1}^{i=n} \{< li, x, root\text{-}variable(ti)\ \}, x, \{< x, s >\} >$;
     For every i=1 to n do
       if $psi - auto(X \cup \{x\}, ti)$ is not nil Suppose it is $< A, B, C, D, E >$, then
         init $\leftarrow < init(1) \cup A, init(2) \cup B, init(3) \cup C, init(4) \cup D, init(5) >$.
    Return(init);
    end
  ======================
```

Suppose $Auto1 = < L, S1, \pi 1, s1, Type1 >$, $Auto2 = < L, S2, \pi 2, s2, type2 >$ are type-based finite automata of ψ-term $t1, t2$, how can we calculate $Auto3 = < S, L, \pi, s0, type >$ such that $Auto3$ is the ψ-term $t3 = glb(t1, t2)$'s automata?

First we assume:

1. Every two finite automata states are disjoint.
2. Every state can be regarded as the following structure:

```
  ======================
  record
  tag: variable name
  type:
  subnode: < label, pointer to the corresponding variable>
  core: used to point the core relation
  end
  ======================
  Procedure GLB(s, t);
  pairs $\leftarrow < s, t >$;
  While pairs $\neq \{\}$ do
   begin
   remove $< x, y >$ from pairs;
    u $\leftarrow Find(x)$;
    v $\leftarrow Find(y)$;
    if $u \neq v$ then
      begin
      $\sigma \leftarrow u.type \cap v.type$;
        begin
        UNION(u, v, w);
        w.type $\leftarrow \sigma$;
        for each l $\in$ labels(u) $\cup$ labels(v)
        do
          begin
          if $w = v$
            then Carrylabel(l, u, v)
            else Carrylabel(l, v, u);
          if l $\in labels(u) \cap labels(v)$
            then Pairs $\leftarrow Pairs \cup$
                 $\{< subterm(u, l), subterm(v, l)\}$
          end
        end
      end
   end
  return(Rebuild($Find(s)$))
  end
  ======================
  Find(x)
  begin
   list $\leftarrow$ nil;
   while $x.core = nil$ do
     begin
     add x to the list;
     x $\leftarrow x.core$
     end
   For each w on list do $w.core \leftarrow x$;
   return x
  end
  ======================
  UNION(i, j, w)
  assume count(root(i)) $\leq$ count(root(j))
  otherwise interchange i and j;
  begin
  large $\leftarrow root(j)$;
  small $\leftarrow root(i)$;
  small.core $\leftarrow large$;
  count(large) $\leftarrow$ count(large) + count(small)
  name(large) $\leftarrow w$;
  root(w) $\leftarrow large$
  end
  ======================
  Procedure Carrylabel(l, u, v);
   begin
   if $l \notin labels(v)$
     then v.subnodes $\leftarrow$
          $\cup\{< l, Find(subterm(u, l)) >\}$
   end
  ======================
  Procedure Rebuild(s);
   begin
   classes $\leftarrow \cup_{x \in subterm(s)} \{Find(x)\}$;
   for each x in classes do
     ID(x) $\leftarrow$ Newtagsymbol;
   for each x in classes do
     begin
     node $\leftarrow$ Newtagnode;
```

```
      with node do
        begin
        node.id ← ID(x);
        node.type ← $x.type$;
        subnodes ← $\{< l, ID(Find(y)) >$
$| < l, y > \in x.subnodes\}$;
        node.core ← nil
        end
      end
    end
  return (ID(Find(s)))
  end
  =======================
```

How to compute two well-defined ψ-terms least upper bound?

```
  =======================
  Procedure LUB(s, t);
    begin
    Pairs ← $(s, t)$;
    while Pairs ≠ nil do
      begin
      remove (x, y) from Paris;
      $u \leftarrow Find(x)$;
      $v \leftarrow Find(y)$;
      if $u \neq v$ do
        begin
        $\sigma \leftarrow u.type \vee v.tape$;
        for every label l ∈ $u.label$ do
          begin
          if l ∉ $v.label$ then
            move l-subterm from u
          else Pairs ← Pairs ∪
        $\{< subterm(l, u), subterm(l, v) >\}$;
          end;
        u.type ← $\sigma$;
        v.core ← $u$;
        end
      end
    end
  Rebuild(Find(u))
  end
  ====================
```