PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT. © ICOT, 1988

Partially Specified Term in Logic Programming for Linguistic Analysis

Kuniaki Mukai

Institute for New Generation Computer Technology
Mita Kokusai-Build. 21F
4-18, Mita 1-Chome, Minato-ku, Tokyo 108 Japan

csnet: mukai%icot.jp@relay.cs.net
uucp:{enea,inria,kddlab,ukc}!icot!mukai

ABSTRACT

This paper describes several aspects of a record-like structure called PST (*Partially Specified Term*), which was introduced into a logic programming language called CIL. The semantic domain for PST consists of infinite trees called PTT with merging as built-in operation. PTT domain differs from the well known one of infinite trees of Colmerauer[9] in that a PTT is of non-fixed arity and have possibly infinite number of branches at a node. Unification grammar formalism is straightforwardly expanded over the domain, which is a natural extension of Definite Clause Grammar. As a new technical result, it is shown that PTT domain is satisfaction complete and compact in the sense of *constraint logic programming schema* [13] with respect to a very simple system of constraints. The PTT/PST theory is a new step towards an integrated domain of syntax and semantics for linguistics analysis[1].

1 Introduction

Record-like structures are basic and essential in linguistic analysis. They have been used widely in computer languages, data bases theories and computational linguistics, and so on. The structure appears in *frame* of Minsky, *attribute-value pairs list*, *property list* of LISP, *functional structure* in LFG[8], *anadic relation* of Pollard[23], *category* in GPSG[24], *assignment* or *state of affairs* in situation theory of Barwise[4], and so on. Thus it is natural to introduce record-like structure into logic programming for such variety of applications. Intuitively, record-like structure (simply, record hereafter) is just defined inductively as a set of ordered pairs written $r = \{a_1/v_1, ..., a_n/v_n\}$, where v_i may be a record again, and a_i are distinct *labels*.

[1]A full version of the paper will appear with the title "A System of Logic Programming for Linguistic Analysis" as an internal technical report.

Why is record so useful? It can be answered that it has many functions in the following senses:

- record, say r as immediately above, is a *partial function* such that $r(a_i) = v_i$.
- r is a *directed graph* or *tree*, which has an edge labeled with a_i from the node r to node v_i.
- r is a *finite state automata*.
- r is a set with *hereditary membership relation*.
- r is an *indexed set*, $\{v_\lambda\}_{\lambda \in I}$, where $I = \{a_1, ..., a_n\}$ is the index set.
- r is an *association list*.
- r is a *list of attribute-value pairs*
- r is an *algebraic structure* which is *associative*, *commutative*, and *idempotent* [11].
- r is an finite or infinite *stream*.
- r is a Herbrand term $f_r(v_1, .., v_n)$ for some appropriate functor f_r. An Herbrand term can be seen as a degenerated representation of a record.

It is clear that these aspects are necessary and useful for linguistic analysis. So the problem is how to introduce a domain of records into logic programming. That is, the main objective of the paper is to propose a domain of records and its theory which meets above interpretations following the CLP(Constraint Logic Programming) schema[12]. A record in the proposed domain is called a PTT(Partially Tagged Tree). A PTT is roughly an infinite tree of Colmerauer[9]. However since record has no arity in nature, it is not obvious to establish expected properties of the domain. In other words, the new domain has a natural order, of which the domain of infinite trees has no counter part. The theory of PTT is described in terms of *compatibility* of two PTTs.

The theoretical results of the paper are:

(1) The PTT domain is *compact.*

(2) The theory of PTT is *satisfaction complete.*

This means that PTT domains can be builtin logic programming which are complete and sound with respect to both its computation rule and Negation as Failure Rule. As a fact, the domain was one of the major motivation of a logic programming language called CIL[17, 20]. Actually, CIL has four year experience in use for natural language processing among other applications, showing the usefulness of the domain as expected. The current version was transplanted on PSI machine. Preparing environmental facilities, CIL has been playing a role of basic language for natural language processing at ICOT[16].

Technical heart of the theory of PTT domain was described in Mukai[18]. For instance, soundness and completeness were formalized and proved there.

This paper reorganizes the theory following CLP schema[13]. Due to the general theory[12], soundness and completeness result of Negation as Failure are obtained automatically from the proof that the domain is *canonical* [12].

Recently there have been many important formulations and studies about record-like structure and they are still on progress. For instance, *feature structure* in unification grammar has been extended so that they have descriptions for disjuntive information and even for negative one [2, 15, 21, 25].

However as far as the author knows, it is not clear how to fit the PTT domain to these theories. The PTT domain seems to be a good test stone for any computational domain theory[11].

Another motivation of PTT domain comes from situation semantics[7]. Central one is to use PTT domains to integrate syntax and semantics processing within a combined framework of situation semantics and constraint logic programming. Particularly, some parallelism is pointed out between P. Aczel's new non-well-founded sets theory called ZFC/AFA[1] and the PST/PTT theory. A model for situation theory was given in Barwise[5, 6] within the universe of ZFC/AFA. These observation strongly suggests possibility of new set theoretical domains for logic programming which are more general and transparent than the traditional Herbrand universe. The PTT/PST theory is a new step towards an integrated domain of syntax and semantics for linguistics analysis.

This paper is organized as follows: In Section 2, an example from simple discourse analysis is described to illustrate some motivations to use PTT domains integrating syntax and semantics using a portion of idea of situation semantics and unification grammar. In Section 3, the syntax and semantics of CIL is summarized as far as PTT is concerned. In Section 4, some built-in utilities of CIL for the PTT are illustrated. In Section 5, several leading ideas are discussed for linguistic analysis in PTT domain putting emphasis on the ideas from situation semantics. Section 6 is for the theory of PTT domain. Unification over the domain is characterized in terms of partial equality theory. Satisfiability is defined so that it is equivalent to unifiability. This is the heart of the idea of PTT domain theory. Using the solution lemma in Mukai[18] essentially, which has a form very similar to the one in Aczel's ZFC/AFA, the domain is proved to be complete. This means that the PTT domain is *canonical*[12]. Moreover, it concludes that logic programming can enjoy *Negation As Failure* rule over this domain. The paper is concluded at Section 7.

2 Using Partially Specified Terms

Before we will describe the language in the sections below, we show an example which illustrates discourse interpretation using situations and feature set. The example program illustrates ideas to use PSTs for linguistic analysis. It includes a simple use of constraint by lazy evaluation. The program expresses a naive idea about the meaning of sentence proposed in some earlier version of situation semantics that the meaning of a sentence is a relation between discourse situations and described situations in Barwise and Perry[7].

Imagine the following discourse piece between two persons, say Jack and Betty:

(1) *Jack: I love you.*
(2) *Betty: I love you.*

The two sentences are same, but interpretations of (1) and (2) are different as in (3) and (4):

(3) *Jack loves Betty.*
(4) *Betty loves Jack.*

This difference is an example of language efficiency [7]. How is this kind of language efficiency analyzed in CIL? We demonstrate the power of PSTs by giving a program to analyze the simplified discourse.

The name of the top level predicate is discourse_constraint. For the query

`?- discourse_constraint([(1),(2)], [X, Y]).`,
the program will produce answer interpretations `X`= (3) and `Y` = (4) for (1) and (2), respectively, as are expected.

In this illustration, suppose simplified discourse constraints (5) and (6):

(5) *The speaker and hearer turn their roles on each sentence utterance.*

(6) *The successive discourse locations are numbered sequentially.*

First, let us see the following clause:

```
(7) discourse_situation(
          {sit/S,sp/I,hr/You,dl/Here,exp/Exp}):-
   member(soa(speaking,(I,Here),yes),S),
   member(soa(addressing,(You,Here),yes),S),
   member(soa(utter,(Exp,Here),yes),S).
```

This clause asserts that an object x, which is parameterized with {sit/S, sp/I, hr/You, dl/Here} and {exp/Exp} is a discourse situation if S has the three state of affairs as indicated in the body of the clause. The membership definition is as usual.

```
(8) discourse_constraint([],[]):-!.
    discourse_constraint([X],[Y]):-!,
        meaning(X,Y).
    discourse_constraint([X,Y|Z], [Mx,My|R]):-
        meaning(X,Mx),
        turn_role(X, Y),
        time_precedent(X, Y),
        discourse_constraint([Y|Z],[My|R]).
```

The first and second arguments are a list of discourse situations and a list of described situations, respectively. The clauses constrain discourse situations and described situations with the rule (5) and (6) above.

The constraint (5) is coded in the clause:

```
(9) turn_role({hr/X,sp/Y},
  {hr/Y,sp/X}@discourse_situation).
```

According to the context of the program, this clause presupposes that the first argument is a discourse situation. The term

```
        {hr/X,sp/Y}@discourse_situation
```

in the second argument place constrains that the actual argument contains both information {hr/X, sp/Y} and some discourse situation which satisfies the constraint defined above.

The constraint (6) is coded in the clause (10):

```
(10) time_precedent({dl/loc(X)},{dl/loc(Y)}):-
constr(X+1=:=Y).
```

The CIL call constr(X+1=:=Y) constrains X and Y with the arithmetic constraint that the latter is greater than the former by one.

The sentence interpretation is described in DCG form. The following clause is an interface between the discourse situation level and sentence level.

```
(11) meaning(X#{exp/E},Y):-
            sentence(E-[],{ip/Y,ds/X}).
```

The sentence model is very simplified. A sentence consists of a noun, verb, and another noun in order. There are only four nouns, i.e., *jack*, *betty*, *i(I)*, *you*. The word *love* is the only verb here. The feature system is taken from GPSG[24]. The control agreement principle is illustrated using subcategorization features. By checking the features agreement between the subject and verb, (12) is legal, but (*13) is illegal.

(12) *I love you.*

(*13) *Jack love you.*

The verb *love* has several semantic parameters: *agent*, *object*, *location*, and so on. The first and last nouns are unified with *agent* and *object* parameters, respectively. The location comes from the given discourse situation parameter. The agreement processing and role unification are coded in the following two clauses (14), (15) using PSTs, where *ip* stands for *interpretation*.

```
(14) sentence({ip/SOA,ds/DS})-->
       noun({ip/Ag,ds/ DS, syncat/{head/F}}),
       verb({ip/SOA, ds/DS, ag/Ag, obj/ Obj,
        syncat/{subcat/F}}),
       noun({ip/Obj, ds/ DS}).
```

```
(15) verb(
  { ip/ soa(love,(X, Y, Loc), yes),
    ds/ {dl/Loc},
    ag/ X,
    obj/Y,
    subcat/{head/{minor/{agr/({plu/P, per/N}:
      (P=(+),N= (@per);
       P=(-),(N=1; N=2)))@agr}}}@category})
     --> [love].              % love
```

The pronoun *I* and proper name *Betty* are described as follows. The agreement features of *I* are the first person and singular. The agreement features of *Betty* are *the third person* and *singularity*. The interpretation of the pronoun *I* is the hearer of the given discourse situation.

```
(16) noun(ip/betty,
          syncat/{head/{minor/{agr/
  {plu/(-),per/3}@agr}}}@category})
        -->[betty].             % Betty
     noun({ip/X, ds/{sp/X},
          syncat/{head/{minor/{agr/
    {plu/(-),per/1}@agr}}@category})
        -->[i]                  % I
```

The system of syntax categories in this example is described as follows:

```
(17) category({bar/ @bar, head/ @head}).
```

This clause says that an object which contains {bar/B, head/H} is a category, where B and H are a bar category and head category.

The following is a category specification by PSTs:

```
(18)    {bar/2,
         head/ {major/ {n/ +, v/ -},
                minor/ {agr/ {per/1, plu/ -},
                        case/ acc     }}}.
```

Take query (19), to the above defined constraint, for example.

```
(19) ?- discourse_constraint(
      {sit/[soa(speaking,  (jack, _), yes),
            soa(addressing,(betty, _),yes)|_],
       exp/ [i,love,you],
       dl/  loc(1)}@discourse_situation,
       {exp/ [i,love,you]}@discourse_situation],
       Interpretation).
```

Note that no parameter other than expression parameter is specified in the second discourse situation in this query. The other parameters are determined by the discourse constraint. Then, the exact output of this query is (20):

```
(20) Interpretation =
        [soa(love,(jack,betty,loc(1)),yes),
         soa(love,(betty,jack,loc(2)),yes)].
```

3 summary of Syntax and Semantics of CIL

3.1 Syntax

Hereafter by first order term, we mean the usual first order term, such like ones in Prolog. We define a class of terms and clauses in CIL by extending the first order term. Let us fix two disjoint sets V of variables and C of constants. For simplicity, C includes atomic symbols, integer constants and functor symbols all together. We follow the convention in Edinburgh Prolog[22] for variables and constants. That is, Abc and _323 are variables and abc and 323 are constants. The following are auxiliary symbols:

{ } , ( ) / :- ;

A class of terms are inductively defined as follows. A variable is a *term*. If f is a constant and $x_1, \ldots, x_n$ are terms with $n \geq 0$ then $f(x_1, \ldots, x_n)$ is a *term*. If $a_1, \ldots, a_n$, are first order terms and $x_1, \ldots, x_n$ are terms then the set

$$\{a_1/x_1, \ldots, a_n/x_n\}$$

is a *term*.

According to this definition, a constant c is a term with $n = 0$. A term of the form $f(x_1, ..., x_n)$ is called a *totally specified term (TST)*. The term $\{a_1/x_1, ..., a_n/x_n\}$ is called a *partially specified term (PST)*. {} is a PST by definition with $n = 0$ and is called the *empty* PST.

We need a subclass consisting of TST terms to introduce *conditioned term*. We assume that the class of conditions is closed with respect to ordinary Boolean combinations: *conjunction*(,), *disjunction*(;), and *negation*(not). PST terms are not allowed to be conditions.

We introduce some special forms of TSTs.

(1) :(x, y) : a *conditioned term*,
(2) @(x, y) : a *conditioned term* (*lazy* version of (1)),
(3) #(x, y) : a *tagged* term,
(4) !(x, y) : a *labeled* term,
(5) ?(x) : a *frozen* term.

TSTs of the form :(x, y), @(x, y), !(x, y), #(x, y) are written in infix notation x:y, x@y, x!y, x#y. Also the TST ?(x) may be written in postfix notation x?.

Several examples of terms follow: [1,2,3], 3+5, and soa(give, {agent/A,recipient/'Jack'},1) are TSTs. {} is the empty PST. Both {agent/father(X), object/O, recipient/X} and {SLOT/X, feature(Y)/Z} are PSTs. X:(man(X),wife_of(X,Y),pretty(Y)) is a conditioned term, and also is Z@(Z>0). S#soa(R,{a/A,b/S},P) is tagged term. Man!name!first is a labeled term. Both X? and (Man!name)? are frozen terms.

3.2 Program Clause

A *program* is a finite set of program clauses, where a *program clause* is a TST. We fix a class of program clauses and define *unit clause, head, body, goal, query* and so on as usual. A program is executed in *top down depth first and from left to right way* as the standard Prolog. As are introduced in the previous section, CIL has various reserved forms of terms. The current CIL treats them as *macros*. They are translated into normal form when the system reads the program clauses. For convenience of explanation, a term t is written informally $t[s]$ if t has a subterm occurrence s. We write $t[s']$ for the term obtained from t by replacing the occurrence s with s'. The rules (1)-(6) below are rewriting rules for the macro expansion. In these rules, r represents a program clause and a represents a TST whose main functor is other than logical connectives (, ; *not*).

[Rules of Expanding Macros]

(1) $r[x@c] \rightarrow r[x : freeze(x, c)]$

(2) $r[x\#y] \rightarrow r[x : (x = y)]$

(3) $r[x!y] \rightarrow r[z : (x = \{y/z\})]$

(4) $h[x : c] : -b \rightarrow h[x] : -solve(c), b$

(5) $h : -b[a[x : c]] \rightarrow h : -b[(a[x], solve(c))]$

(6) $h : -b[a[x?]] \rightarrow h : -b[freeze(x, a[x])]$

Given a program, these rules are applied in *outermost-first* principle. These rules are applied until they become not applicable. It is easy to see that final one does not contain any of ?, @, :, #, !. Thus we can assume that a program contains no part of these reserved forms. Therefore semantics of CIL is reduced to the one for our extended unification $x = y$, suspending *freeze* and *condition interpreter*, *solve*.

In the current implementation, *solve* is an interpreter like CIL itself. *freeze* is the lazy control primitive of Colmerauer. The unification is described below in detail using examples.

4 Built-in Predicates for PST

Built-in functions in CIL are listed below with example uses. PSTs and lazy evaluation are major points of CIL. The other parts follow the standard Prolog specification. Most of what follows in this Section are related to handling PSTs.

In what follow, single upper-case letters such as X, Y, Z are used only for Prolog variables. Greek letters are used for any terms.

4.1 Unification and Copy

The goal $\alpha = \beta$ unifies two terms α and β. The execution $X = \{a/1\}$, $Y = \{b/2\}$, $X = Y$ yields the binding $X = Y = \{a/1, b/2\}$. Similarly, $\{a/b, c/\{d/E\}\}!c!d = h$ yields $E = h$. The next example shows something like *if-filled-demon* in CIL. Note that @*print* is equivalent to $V : freeze(V, print(V))$ where V is a new Prolog variable: $X = \{a/ok\}$, $Y = \{a/@print\}$, $X = Y$ displays *ok*. $X\#\{a/1, b/X!a\} = Y$ yields $X = Y, X = \{a/1, b/1\}$, because the value of a-slot of X is 1. It is easy to produce and represent a circular graph in CIL: The goal $X = \{a/b, c/Y\}$, $Y = \{a/b, c/X\}$, $X = Y$ yields the *circular* graph X, where $X = Y = \{a/b, c/X\}$.

As illustrated later, the notion of parametric objects and handling them seem to be very important in processing natural language semantics. Although much remains to be investigated on the topics, here are four related built-in predicates toward the parametric objects: *fullCopy*, *typeOf*, *createType*, and *instance*.

$fullCopy(\alpha, \beta)$ makes a fresh copy of α and unify it with β. $typeOf(\alpha, type(\beta, \gamma))$ makes a *fullCopy*, (β, γ'), of (α, γ) and then $solve(\gamma')$. $createType(\alpha, \beta, type(\gamma, \delta))$ unifies (γ, δ) with the *fullCopy* of (α, β). $instance(\alpha, \beta)$ performs $fullCopy(\beta, \gamma)$ and *unifies* γ and α.

Here are two examples of queries related to *copying*. The first example will display *ok*:

$$X = \{a/@print, b/X\}, fullCopy(X, Z), Z!b!a = ok?$$

Note that *fullCopy* makes a copy of circular structure.

Take the following execution as the second example: $createType(Y, (Y = 1; Y = 2), T)$, $typeOf(1, T)$, $typeOf(2, T)$. This will display *yes*. Note that T behaves as if it got bound to a type whose extension is the set $\{1, 2\}$.

4.2 Partially Specified Terms (PST)

Here are utilities of CIL for handling PSTs: *getRole*, *locate*, *setOfKeys*, *role*, *partial*, *record*, *buffer*, *glue*, *merge*, *d_merge*, *subpat*, *extend*, *meet*, *frontier*, *match*, *t_subpat*, *t_merge*, and *masked_merge*.

The goal $getRole(\pi, \kappa, \xi)$ unifies the value of "κ-slot" of a record π with ξ, i.e., $\pi!\kappa = \xi$. κ does not need to be ground. $X = \{a/1, b/2\}$, $getRole(X, K, V)$ will produce two sets of bindings in backtracking way: $(K = a, V = 1)$ and $(K = b, V = 2)$.

The goal $locate(\pi, \kappa, \xi)$ performs a similar operation like *getRole* except that κ must be ground. The execution will fail if π has not the argument place, i.e., slot, named κ. $locate(\{a/b\}, a, L)$ produces $L = b$. On the contrary, $locate(\{a/A\}, b, L)$ will fail because there is no *b-slot* in the first argument. $locate(\{a/A\}, a, L)$ produces $A = \_40$ and $L = \_40$, which shows A and L are unified with each other.

$setOfKeys(\pi, \sigma)$ collects the all keys in a given PST π:

$$setOfKeys(\{a/X, b/Y, c/Z\}, S)$$

produces $S = [a, b, c]$.

$role(\kappa, \pi, \xi)$ unifies ξ with the content of κ-slot of π. If κ is not ground then the execution is suspended. An argument place named κ is created in π when π has not the place. For example, the execution of $X = \{a/1, b/2\}$, $role(K, X, 3)$, $K = c$ will produces $K = c$, and $X = \{a/1, b/2, c/3\}$.

$record(\pi, \xi)$ produces a stream ξ which consists of pairs (σ, τ) such that $\pi!\sigma = \tau$. This predicate is similar to *buffer* below. ξ is generated as a stream from π. This predicate is used as a stream generator. For example, $record(\{a/1, b/2\}, R)$ produces $R = [(a, 1), (b, 2)]$. It is useful to attach "consumer" processes to ξ.

$buffer(\pi, \xi)$ is a *buffered stream producer* which converts a PST π to ξ. Unlike *record* predicate, the stream container ξ is assumed to be produced by other process. *buffer* puts in some order each pair (σ, ν) such that $\pi!\sigma = \nu$ on ξ. If ξ is long enough then *end_of_list* is put on ξ just after the final pair in π. If the buffer length is not enough, then *buffer* waits till the buffer ξ grows enough. The execution $buffer(\{a/1, b/3\}, B)$, $B = [A|C]$ will produce $A = (a, 1)$, $B = [(a, 1)|\_1161]$. $buffer(\{a/1, b/3\}, [A, B, C, E])$ will produce $A = (a, 1)$, $B = (b, 3)$, $C = end$, and $E = \_118$, because the buffer is long enough.

The goal $glue(\pi, \tau)$ glues π and τ only at the *joint*. That is, for each common argument place name κ of π and τ, $\pi!\kappa$ and $\tau!\kappa$ are unified with each other. For example,

$$glue(A\#\{a/H\#\{b/1, c/2\}\}, C\#\{a/G\#\{c/B\}\})$$

produces $H = \{b/1, c/2\}$, $A = \{a/\{b/1, c/2\}\}$, $B = 2$, $C = \{a/\{b/1, c/2\}\}$ $G = \{b/1, c/2\}$.

The goal $merge(\pi, \tau)$ merges π to τ. That is, τ is extended minimally so that π is a subpattern of τ. More precisely, for each κ-slot of π, κ-slot of τ is created if necessary and the two fields are unified. For example,

$$merge(X\#\{c/d, a/4\}, Y\#\{a/B\})$$

produces $X = \{a/4, c/d\}$, $B = 4$, $Y = \{a/4, c/d\}$.

The goal $d\_merge(\pi, \tau)$ merges π to τ like $merge$ just above. Unlike $merge$, however, d_merge leaves conflicting fields left unchanged. For instance, $d\_merge(X\# \{c/d, a/4\}, Y\#\{a/5\})$ produces $X = \{a/4, c/d\}$, $Y = \{a/5, c/d\}$. Note that two PSTs have no unifiable a-slots each other.

The goal $subpat(\pi, \tau, \delta)$ tests whether π is a *subpattern* of τ or not. More precisely, the goal succeeds only if each slot name κ of π appears also in τ. δ is a *difference* list, which consists of triples (κ, ξ, η) such that π and τ have the arguments ξ and η with the name κ, respectively. $subpat$ is used in t_subpat below.

The goal $extend(\pi, \tau, \delta)$ extends τ minimally in such a way that π becomes a subpattern of τ. δ is the difference list of π and τ as above. $extend$ performs the same functions as $subpat$ except that τ may be extended.

The goal $meet(\pi, \tau, \delta)$ computes the difference list δ, which consists of all triples (σ, ξ, η) such that π and τ have the arguments ξ and η with the name σ, respectively. For example, $meet(\{a/1, b/2\}, \{b/3, c/4\}, A - [])$ produces $A = [(b, 2, 3)]$.

The goal $frontier(\pi, \tau, \delta)$ computes the difference list between π and τ, where, π and τ are non variable terms. This predicate may fail because of some unmatching functors pairs. For example, $frontier(f(a, g(b)), f(A, B), L - [])$ will produce $A = \_54$, $L = [g(b) = \_75, a = \_54]$, $B = \_75$. On the other hand, $frontier(a, b, L - [])$ will fail.

The goal $match(\pi, \tau, \delta)$ computes the difference list of terms π and τ of any forms. Unlike $frontier$, this predicate always succeeds. For example, $match(a, b, L - [])$ produces $L = [a = b]$.

The goal $t\_subpat(\pi, \tau)$ tests whether π is a subpattern of τ in a *transitive* way. This predicate is intended to be a realization of *hereditary subset relation* in set theory. For example, $t\_subpat(\{a/\{b/Y\}\}, \{b/1, a/\{c/2, b/3\}\})$ succeeds. On the other hand, $t\_subpat(\{a/\{b/Y\}, c/U\}, \{b/1, a/\{c/2, b/3\}\})$ fails.

The goal $t\_merge(\pi, \tau)$ merges π to τ in a transitive way. For instance,

$$t\_merge(\{b/1, a/\{c/2, b/3\}\}, \{a/\{b/Y\}\})$$

produces $Y = 3$.

The goal $delete(\kappa, \pi, \tau)$ deletes the κ-slot of π. More precisely, τ is unified with the record which is the same as π except that it has no κ-slot. For instance,

$$delete(a, \{a/1, b/2\}, O)$$

will produce $O = \{b/2\}$.

The goal $masked\_merge(\pi, \mu, \tau)$ computes π *minus* μ and then merges it to τ. For instance,

$$masked\_merge(\{a/1, b/1, c/1\}, \{a/\_, b/\_\}, U\#\{a/2\})$$

will produce $U = \{a/2, c/1\}$.

5 PST in Linguistic Analysis

In this Section, we illustrate several ideas how to use PST describing linguistic analysis.

5.1 Features Co-occurrence Restriction.

Let us take an example from linguistic constraint on a feature set that if $refl$ feature of X is $(+)$ then the gr feature of X must be sbj. This is an example of constraint called a Feature Co-occurrence Restriction (FCR) in GPSG written :

$$\langle REFL\ +\rangle \Rightarrow \langle GR\ SBJ\rangle.$$

Using $constr$, which is a built-in predicate in CIL, a feature set X is constrained so by a call

$$constr((X!refl = (+) \rightarrow X!gr = sbj)).$$

By effect of this constraint, the following query generates automatically the gr feature in X: the execution of goals $constr((X!refl = (+) \rightarrow X!gr = sbj))$, $X!refl = (+)$, $A = X!gr$ in order produces $A = sbj$, $X = \{refl/(+), gr/sbj\}$.

5.2 Complex Indeterminate

Here complex indeterminate is understood as restricted (= conditioned) parameter, which is written $x : c$ in CIL, where x is a term and c is a condition about x. The following is an example of complex indeterminate, which describes a discourse situation:

$$\begin{array}{l} \{sit/S, sp/I, hr/You, dl/Here, exp/Exp\} : \\ \quad (\ \ member(soa(speaking, (I, Here), yes), S), \\ \quad\quad member(soa(addressing, (You, Here), yes), S), \\ \quad\quad member(soa(utter, (Exp, Here), yes), S)). \end{array}$$

The second illustration is related to semantics of question sentence. The idea is that the meaning of a question sentence is a complex indeterminate of the form $x : c$ and that answering the question is to return the value of x after solving condition c, applying appropriate contexts to contextual parameters in x.

However, although CIL stands for complex indeterminate language, the current implementation has succeeded to catch only small part of this rich notion. Much remains to be done in the future.

5.3 Attitudes in PSTs

An attitude (mental state) is here understood as a pair of a frame and a setting[7]. A frame is a parametric object, and a setting is an *assignment* or *anchor*, that is, binding information.

Let (T, A_1) and (T, A_2) be two mental states with a same frame T and different setting A_1, A_2. These might remind the reader of familiar data structure called a *closure* in LISP or a *molecule* in Prolog of structure sharing implementation. Suppose the following two belief contexts:

(1) *Jack: I believe Taro beats Hanako.*
(2) *Betty: I believe Hanako beats Taro.*

The mental states of (1) and (2) may be represented in (3) and (4), where *beater* and *beaten* are indeterminates. The setting of *jack's* belief is

$beater \mapsto taro,$
$beaten \mapsto hanako$

(3) $believe(jack, \{\ frame/beat(beater, beaten),$
$\qquad beater/taro,$
$\qquad beaten/hanako\})$

Similarly,

(4) $believe(betty, \{\ frame/beat(beater, beaten),$
$\qquad beater/hanako,$
$\qquad beaten/taro\}).$

Since mental states are represented in PSTs, various types of queries about the states are treated by unification over PSTs as follows:
(5) *Who believes taro is the beater?*
(6) $?-believe(X, \{beater/taro\}). \Rightarrow X = jack$
(7) *Who does jack believe is beaten?*
(8) $?-believe(jack, \{beaten/X\}). \Rightarrow X = hanako$
(9) *What does jack believe taro does?*
(10)

$?-believe(jack, M\#\{frame/T\}), getRole(M, A, taro).$
$\Rightarrow A = beater,$
$\quad Z = beat(beater, beaten),$
$\quad M = \{\ frame/beat(beater, beaten),$
$\qquad beater/taro,$
$\qquad beaten/hanako\}$

5.4 Type Theory for Parametric Object

Assume a discourse which the question (1) below presupposes:

(1) *What is Mr. Roll?*

Query (1) may be formalized in some polymorphic type system as (2):

(2) $\exists(X : role, is\_type\_of(\text{“}Mr.Roll\text{”}, X))$ The type inference system will compute a *realizer* X_0 such that “Mr. Roll” is *of-type* X_0. Combined with the situation theory, such a type theoretic approach [10] seems to be a promising approach which will be useful at least to make clear some classes of the discourse understanding problems.

A type inheritance mechanism seems to be easily embedded in CIL by modifying unification over PST. Suppose a type hierarchy is given as a lattice. The idea is very simple. Let P_1 and P_2 be PSTs. Then the desired unification + between PSTs is defined as shown (1):

(1) $(\{type/S_1\} + P_1) + (\{type/S_2\} + P_2) =_{\text{def}}$
$\qquad \{type/(S_1 + S_2)\} + P_1 + P_2,$

where $S_1 + S_2$ means the meet operation in the lattice.

Also Ait-Kaci proposes a similar inheritance mechanism based on his ψ- terms[2, 3]. However sophisticated type inheritance including parametric objects is a further work.

6 Theory of Partially Tagged Trees

A characteristic points of theory of partially tagged trees (PTT) are summarized in this section following CLP schema. First of all, the domain of PTTs and its algebraic properties are introduced. Secondly, a theory of partial equation is introduced over the domain. Unification is formalized in terms of the theory. Thirdly, a simple model theory for the language is defined over the PTT domain in terms of satisfiability relation. There are two main results here: (1) satisfiability and unifiability are equivalent, and (2) our theory of partial equation is *compact*. This result gives good qualification of the PTT domain in logic programming. The following descriptions is not self-contained. A full version will appear elsewhere.

6.1 DAG and PTT

Before going into PTT, we remark some difference between DAG and PTT. DAG in unification grammar and PTT in CIL are very close to each other, as is easily expected from Shieber[14], for instance. However, we point out some difference in that CIL uses PSTs, which use logical variables essentially and that PSTs are interpreted more straightforwardly to be constraints on PTTs as linguistic information. Note that the PTT domain is mathematically more simple than the DAGs domain in that the former is the latter modulo graph isomorphism. Furthermore, owing to using logical variables, the behaviors of CIL are described completely in a constraint logic programming languages schema[12] in particular unification schema. DAG has *structure sharing* property as objective one. On the other hand, PTT can treat the structure sharing property through only meta level notion of sharing variables. However, it is not clear whether structure sharing is an essential linguistic relation or not.

Since a PTT might be infinite, CIL can represent more complex structure than what a (finite) DAG can do. In particular, the PTT domain may be suitable for representing and processing circular situation proposed by Barwise[6] in conjunction with linguistic analysis.

6.2 Partially Tagged Trees

We fix a set $LABEL$ hereinafter. An element in $LABEL$ is called a *label.* $\langle a_1, ..., a_n\rangle$ stands for the string of labels $a_1, ..., a_n$. In particular, $\langle\rangle$ stands for the empty string. The length of the empty string is zero.

We define *concatenation*, $*$, between strings as usual by the following equations:

$$\langle\rangle * x = x,$$

$$x * \langle\rangle = x,$$

$$\langle a_1, ..., a_n\rangle * \langle b_1, ..., b_m\rangle = \langle a_1, ..., a_n, b_1, ..., b_m\rangle.$$

We often identify each label a with the string $\langle a\rangle$ if the context is clear.

A *tree* is a non-empty set T of *finite* strings which is closed under prefix. That is, if $x * y \in T$ then $x \in T$. Each element in T is called a *node* of T. Note that every tree has the empty string $\langle\rangle$.

Let T and x be a tree and a node of T. We define T/x to be the set of nodes y of T such that $x * y$ is in T. Clearly, T/x is a tree. If $y = x * a$ for some label a, y is an *immediate successor* of x. A node in a tree T may have infinite branches. That is, for some node x of T, x may have infinitely many immediate successor nodes in T. A node x of T is a *leaf node* if x has no immediate successor in T. It is clear that for any family of trees both the intersection and union are also tree. For a node x and a tree T, $x * T$ denotes the minimum tree which includes $x * y$ for any node y of T. We write xT for $x * T$ simply.

Definition 1 (PTT) *A partially tagged tree (PTT) is an ordered pair (T, f) of a tree T and a partial function f assigning values to some of leaf nodes of T. The PTT $(\{\langle\rangle\}, \phi)$ is called trivial.*

Let $t = (T, f)$ and x be a PTT and a node of T. The expression t/x denotes the PTT, $(T/x, g)$, where g is a tag function defined by the equation $g(y) = f(x * y)$.

Let $t_1 = (T_1, f_1)$ and $t_2 = (T_2, f_2)$ be PTTs. The pair $t = (T_1 \cup T_2, f_1 \cup f_2)$ is called the *merge of t_1 and t_2* iff t is a PTT. The merge of t_1 and t_2 is written $t_1 + t_2$. We define $t_1 \leq t_2$ if $T_1 \subset T_2$ and $f_1 \subset f_2$.

It is easy to check that the set of PTTs is a commutative, associative and idempotent semigroup with the trivial PTT as the identity with respect to the merge operation.

Whenever we write $x = y$ in what follows, it is presupposed that x is defined iff y is defined.

$\epsilon + t = t.$ (UNIT)
$t + \epsilon = t.$ (UNIT)
$(t_1 + t_2) + t_3 = t_1 + (t_2 + t_3)$ (ASSOCIATIVE)
$t_1 + t_2 = t_2 + t_1$ (COMMUTATIVE)
$t + t = t$ (IDEMPOTENT)
$\langle\rangle t = t.$ (UNIT)
$(\alpha\beta)t = \alpha(\beta t)$ (ASSOCIATIVE)
$\alpha(t_1 + t_2) = \alpha t_1 + \alpha t_2$ (DISTRIBUTIVE)
$\alpha(t_1 + ... + t_\lambda + ...) = \alpha t_1 + ... + \alpha t_\lambda + ...$
(DISTRIBUTIVE)

A set of PTTs is called *consistent* iff any pair of t and t' in the set has the merge $t + t'$.

Proposition 1 *A consistent set of PTTs has the least upper bound with respect to the order $\leq$.*

Also it is clear that the PTT domain is *chain complete* with respect to $\leq$.

6.3 Partially Specified Term

Let $VARIABLE$ and $ATOM$ be disjoint sets. Elements in $VARIABLE$ and $ATOM$ are called *a variable* and *a constant* respectively. Let us $LABEL$ be as set of labels as was introduced in the previous section. We assume that these three sets are disjoint to each other for simplicity. The following auxiliary symbols are used:

$$\{ \} / , ( ) .$$

A *partially specified term (PST)* is defined inductively as follows:

(1) a constant is a PST,

(2) a variable is a PST,

(3) the set of the form $\{(a_1/p_1), ..., (a_n/p_n)\}$ is a PST for any *finite* $n \geq 0$ and distinct labels $a_1, ..., a_n$, provided that all elements $p_1, ..., p_n$ are PSTs.

A PST can be regarded as a *finite* PTT, which may have variables as a tag. So, the definitions of αp and p/α is defined just as in the case of PTT. The notation p/α should not be confused with an ordered pair in a PST.

6.4 Theory of Partial Equation

We characterize the CIL unification in terms of equality axioms. Then, a set of equations between terms, a unification problem is to compute the least super set of the given set which is closed under the axioms.

Axioms of Partial Equations:
In what follows, x, y, z, x_i, and y_i are variables, α is a node, f and g are functors, u and v are any expressions and p and q are PSTs. An *atomic formula* is of the form $u \bowtie v$.

(1) $x \bowtie x$.
if $x \bowtie y$ then $y \bowtie x$.
if $x \bowtie y$ and $y \bowtie z$ then $x \bowtie z$.

(2) if $f \neq g$ then $\neg f(\_,\_,...) \bowtie g(\_,\_,...)$.

(3) $p \bowtie p$.
if $p \bowtie q$ then $q \bowtie p$.
(*There is no Transitive law.*)

(4) if $f(u_1,...,u_n) \bowtie f(v_1,...,v_n)$ then $u_1 \bowtie v_1$ and ... and $u_n \bowtie v_n$.

(5) if $p \bowtie q$ and both p/α and q/α exist then $p/\alpha \bowtie q/\alpha$.

(6) if $x \bowtie p$ and $x \bowtie y$ then $y \bowtie p$.

(7) if $x \bowtie p$ and $x \bowtie q$ then $p \bowtie q$.

Let S be a set of atomic formulas. The *closure* of S is the set of atoms which is derivable from S by these axioms. The three axioms of (1) say that the restriction of the binary relation $\bowtie$ to the variables is an equivalence relation between them.

6.5 Satisfiability

An *assignment* is a partial function which assigns PTTs to variables. The definition of assignment can be extended to PSTs as usual. We define *satisfiability relation*, $\models$, between assignment and a formula in the language. In the following, $dom(\alpha)$ is supposed to be enough large to include all the relevant variables:

(1) $\alpha \models x \bowtie y$ iff $\alpha(x) = \alpha(y)$.

(2) $\alpha \models c \bowtie q$ iff $c = \alpha(q)$.

(3) $\alpha \models x \bowtie p$ iff $\alpha(p/\beta) = \alpha(x)/\beta$ for any β such that p/β is a variable.

(4) $\alpha \models p \bowtie q$ iff $\alpha \models p/a \bowtie q/a$ for any label a such that p/a and q/a exist.

It is easy to see that $\models$ is well-defined and exists. Also, $\models$ is extended, as usual, to more general case in which $dom(\alpha)$ is not enough large. That is, $\alpha \models e$ if $\alpha' \models e$ in the sense above for some extension α' of α.

Definition 2 (PET) *A set of atoms of the language is called a PET if it satisfies the axioms of partial equality.*

Theorem 2 *Any PET is satisfiable.*

This is proved in Mukai[18].

Let Π be a directed family of PETs, $\{T_\lambda\}_{\lambda \in I}$, that is, for any T_{λ_1} and T_{λ_2} there is $\lambda_3 \in I$ such that $T_{\lambda_1} \subset T_{\lambda_3}$ and $T_{\lambda_2} \subset T_{\lambda_3}$.

Lemma 1 *For $\{T_\lambda\}_{\lambda \in I}$ as above, $\bigcup_{\lambda \in I} T_\lambda$ is a PET.*

Lemma 2 *The closure of S is the least PET which contains S.*

Theorem 3 (Compact) *For any constraint $C = \{p_1 \bowtie q_1,...,p_n \bowtie q_n,...\}$, if every finite subset of C is satisfiable then C is satisfiable.*

Proof. Let $F = \bigcup_{d \in pow'(C)} E_d$, where, $pow'(X)$ is the set of *finite* subsets of X, and E_d is the closure of d. Since d is satisfiable E_d is a PET. Further, since $\{E_d\}_{d \in pow'(C)}$ is directed, F is a PET. Therefore, from Mukai[18], F is satisfiable.

The relation $\models$ is extended to *logical consequence* as usual, which is used in the following theorem.

Theorem 4 (Satisfaction Complete) *Let $\mathcal{T}$ be the theory of partial equality, i.e., a set of atomic formulas, then*

$$\neg(\mathcal{T} \models \tilde{\exists} \neg p \bowtie q) \text{ implies } \mathcal{T} \models \tilde{\forall} p \bowtie q,$$

where p and q are any PSTs.

(By the equivalence of satisfiability and unifiability in our theory and PTT domain, the proof is easy.)

Example 1 $\{a/\{d/1\}, b/\{d/2\}\} \bowtie \{a/X, b/X\}$ *This constraint is not satisfiable, that is, for any assignment α it is not the case that*

$$\alpha \models \{a/\{d/1\}, b/\{d/2\}\} \bowtie \{a/X, b/X\}.$$

Theorem 5 *The following conditions (1) and (2) are equivalent:*

(1) p and q are unifiable.

(2) for some α, $\alpha \models p \bowtie q$.

6.6 Unification

Now, unification over the PTT domain is formalized simply as the *closure* operation defined above. The input is a set of atomic formulas (i.e., of the form $p \bowtie q$) The output is the closure of the input, or *FAILURE* if the closure contains a *conflict*.
There is a UNION-FIND like algorithm of unification[18]. The algorithm preserves satisfiability between input and output when computation is successful. Also, a set of constraints is satisfiable if and only if it is unifiable.

7 Concluding Remarks

Logic programming has been strengthened for linguistic analysis in an elegant way by introducing a new canonical domain called PTT domain. However, there remain many problems to be studied further. Original motivation of PTT was to give a computational model for parametric types and objects in situation semantics. A further interest in this direction is to find more implicit representations for this domain. So far, only compatibility relation between PTT has been studied. Also, properties of mixed uses of PTT and Herbrand term remains to investigated. An example is to use the dependent type theory for instance to compile PTT constraint to Herbrand term constraint for more efficient computation. As mentioned in the introduction, relationship between ZFC/AFA domain and PTT domain is an interesting and important theoretical question for foundation of logic programming.

Acknowledgments

The author would like to thank Prof. J. Barwise, Dr. J. Goguen, Dr. C. Pollard, and Dr. J-L. Lassez for their earlier comments to the data domain of PTT introduced in the paper. Also I would like to thank Dr. K. Furukawa for his encouragement to the work.

References

[1] P. Aczel. *Non-well-founded sets.* CSLI lecture note series, 1988.

[2] H. Ait-Kaci. *A Lattice Theoretic Approach to Compuation Based on a Calculus of Partially Ordered Type Structures.* PhD thesis, Computer and Information Science, University of Pennsylvania, 1984.

[3] H. Ait-Kaci and R. Nasr. LOGIN: A logic programming language with built-in inheritance. J. of Logic Programming, Vol.3, No.3, 1986.

[4] J. Barwise. The situation in logic- III: Situations, sets and the axiom of foundation. Technical Report CSLI-85-26, Center for the Study of Language and and Information, 1985.

[5] J. Barwise. Notes on a model of a theory of situations, sets, and propositions. Technical report, CSLI, 1987. Manuscript.

[6] J. Barwise and J. Etchemendy. *The Liar: An Essay on Truth and Circular Propositions.* Oxford Univ. Press, 1987.

[7] J. Barwise and J. Perry. *Situations and Attitudes.* MIT Press, 1983.

[8] J. Bresnan, editor. *The Mental Reprsentation of Grammatical Relation.* Cambridge, Mass.: MIT Press, 1982.

[9] A. Colmerauer. *Prolog II: Reference Manual and Theoretical Model.* Groupe Intelligence Artificielle, Universite d'Aix-Marseille II, 1982.

[10] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), December 1985.

[11] J.A. Goguen and J. Meseguer. Order-sorted algebra I: Partial and overloaded operators, errors and inheritance. Technical report, SRI International and CSLI, 1985.

[12] J. Jaffar and J-L. Lassez. Constraint logic programming. Technical report, IBM Thomas J. Watson Recearch Center, 1986.

[13] J. Jaffar and S. Michaylov. Methodology and implementation of a clp system. In *International Conference on Logic Programming*, 1987.

[14] S.M. Shieber F.C.N. Pereira L. Karttunen and M. Kay. A compilation of papers on unification-based grammar formalisms parts I and II. Technical Report CSLI-86-48, April 1986.

[15] 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.

[16] R. Sugimura H. Miyoshi and K. Mukai. Constraint analysis on Japanese modification. In *Natural Language Understanding and Logic Programming.* North Holland, 1987.

[17] K. Mukai. Horn clause logic with parameterized types for situation semantics programming. Technical Report ICOT-TR-101, ICOT, 1985.

[18] K. Mukai. Anadic tuples in prolog. Technical Report TR-239, ICOT, 1987.

[19] K. Mukai. A system of logic programming for linguistic analysis. Technical Report To Appear, ICOT, 1988.

[20] K. Mukai and H. Yasukawa. Complex indeterminates in prolog and its application to discourse models. *New Generation Computing*, (3(1985)), 1985.

[21] F.C.N. Pereira. Grammars and logics of partial information. In *Proceedings of the Fourth International Conference on Logic Programming.* MIT press, 1987.

[22] F.C.N. Pereira and S.M. Shieber. *Prolog and Natural-Language Analysis.* CSLI, 1987.

[23] Carl J. Pollard. Toward anadic situation semantics. Manuscript, 1985.

[24] G. Gazdar E. Klein G.K. Pullum and I.A. Sag. *Generalized Phrase Structure Grammar.* Cambridge: Blackwell, and Cambridge, Mass.: Harvard University Press, 1985.

[25] G. Smolka. A Feature Logic with Subsorts. LILOG-Report 33, IBM Deutschland GmbH, May 1988.