TR-540

A System of Logic Programming for Linguistic Analysis

by
K. Mukai

April, 1990

A System of 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

March 1, 1990

Abstract

This paper describes unique aspects of a logic programming language called CIL. CIL is an extension of the standard Prolog with motivations for linguistic analysis. CIL handles the record structure and constraint predicates on the structure. The record is understood as an extension of the usual first order ground term. Also parametric records are introduced as a partial description of a record.

The usual unification over the first order terms is extended to that over the records straightforwardly. The unification algorithm is presented whose complex ity is almost linear. Thus some version of the unification grammar formalisms is as well embedded into CIL in a natural way as Definite Clause Grammar is embedded into the standard Prolog.

A combination of the record structure and the constraint adds a simple but powerful feature to logic programming for computational models of linguistic information.

The implementation of record is described in detail. Also the built-in library of CIL for constraint is described.

Several intended uses of CIL are explained with small examples, which includes linguistic analysis based on unification grammar and situation semantics.

1 Introduction

Record is a basic and useful data structure in many fields such as computer languages, databases, computational linguistics, and so on. A record is a recursive structure which consists of pairs of 'attributes' and 'values'. We define record in the present paper to be just a *function*. This definition will be made more precise in Section 3.

*To appear in The SRI Tokyo Series on Advanced Technology– Introduction to Fifth Generation Computers. A preliminary version of the present paper was read at Workshop on Semantic Issues on Human and Computer Languages, organized by S. Peters, D.Israel, and J. Meseguer, at Half Moon Bay, California, March 19-21 1987.

Taken as a function, the records form a partially ordered set with respect to the binary relation that one record is a restriction to the other. If two records are compatible as functions then the set theoretical union of the two records also is a record. The new record is called the *merge* of the two records. This merge operation is a fundamental operation on records.

We introduce parametric records, whose general form is like this:

$$\{a_1/b_1, \ldots, a_n/b_n\}.$$

We call this parametric description a PST for short[1]. The PST is understood as a constraint on records f that

$$f(a_i) = b_i$$

for $1 \leq i \leq n$. Also the equation

$$\{a_1/b_1, \ldots, a_n/b_n\} = \{c_1/d_1, \ldots, c_m/d_m\}$$

is a constraint on records g that $g(a_i) = b_i$ for $1 \leq i \leq n$ and that $g(c_j) = d_j$ for $1 \leq j \leq m$.

By writing $arg(a, f, b)$ for $f(a) = b$, it is easy to see that the theory of record can be developed equivalently as a theory of *arg* relation without introducing the notation of PST.

A PST can be used in almost every context where the standard term can be used in the standard Prolog. The meaning of such uses of PST is understood as a notational convention as follows. The Horn clause, for instance,

$$p(\{a/b\}) : -q(\{c/d\})$$

is translated into the following one:

$$p(x) : -q(y)|\ arg(a, x, b), arg(c, y, d)$$

where, the right part of '|' sign is a constraint, and variables x and y denote records. In general, every Horn clause can be translated into another Horn clause which has no PST occurrences but additional *arg* constraint part instead.

Whichever of *PST* or *arg* is used, the essential point is here that there is *partiality* in the set of the records while it is not in the set of the first order *ground* terms. In fact, there is no built-in merge operations over the first order terms[2].

Here is a list of other well-known data structures which have a similar structure and operations to the records in our sense.

- Association list.
- Attribute-value pairs list.
- Frame structure of Minsky.
- Property list of LISP.
- Record type in the programming languages.

[1]In the history of CIL, the record and the parametric record are called a Partially Tagged Tree (PTT) and a Partially Specified Term (PST), respectively[12]. We use PST also in the present paper while not PTT.

[2]In theory, the order relation between first order general terms that one term is more general than the other is a kind of relation between syntactical objects, i.e., it is a meta level notion.

- Record in database.

Also in the recent literature of computational linguistics, linguistic information is formalized as record-like structures forming a DAG. For instance, the feature structure of GPSG[7] and the functional structure in LFG[8] can be thought of as a kind of the record structure with the merge operation. This observation is one of the strongest motivations of our work.

In addition to these examples, we observe that *partial assignments* play some important role for natural language semantics in the light of situation semantics[6]. Also a merge–like operation is defined on the assignments. Together with relations, partial assignments are the major component of *state of affairs*, which is the basic model of information in situation theory. The point is that since the partial assignment[4] is a function which assigns values to argument places, it is just a record in our sense.

Thus it is natural to introduce record structure into logic programming for such a variety of applications. Although the record and the first order term are essentially different structure in the sense of partiality in the sense mentioned above, fortunately the standard unification can be straightforwardly extended to the record. Therefore, we decided to introduce records into logic programming. This new language is called CIL. PST notations can be used together with the standard notation for the usual first order term in a nested way. CIL has a built–in extension to the usual first order unification for the PSTs. The PST together with built–in constraints is a key aspect of CIL towards a new integrated linguistic analysis framework[18].

The first version of CIL worked 1984 for natural language processing among others. An anaphora processing in CIL is described in Mukai and Yasukawa [15]. The current version described in the present paper, is now applied for Japanese phrase grammar analysis on PSI machines.

A theory of the records is described in Mukai[12, 13, 14]. According to the theory, the domain of the records is an extension of the first order term in such a natural way that the domain of the first order terms is embedded into the domain of records and the soundness and completeness properties of the standard unification theory of the first order terms are preserved also in the record domain.

There are many related works to the record. Aït-Kaci[3], pointing out from a type theoretical point of view that the first order term should be extended to a record-like structure, gives a lattice–theoretic foundation to the record-like structure. Pereira [16] gives a related overview on grammars and logics of partial information. Also see Smolka[21, 20], Johnson[9], Kasper and Rounds[10]. They treat not only merge operation, i.e., conjunction, but also other logical operations such as disjunction and negation. The reader is referred to these articles.

In Section 2, we describe an example from simple discourse analysis, which was also used to illustrate the previous version of CIL in Mukai and Yasukawa[15]. Owing to PSTs, the example in this paper is much more clearly written than the previous one. In Section 3, the domain of records is formally defined. In Section 4, the syntax of the language is defined, giving various reserved forms of terms. In Section 5, the class of programs in the language are defined, and an operational semantics is given. In Section 6, the unification algorithm over the records is given and other built–in constraints are explained. In Section 7, several ideas are proposed for applying the record and constraint to linguistic analysis. The paper is concluded at Section 8. In Appendix A, the whole program which is used in Section 2 is listed for convenience. In Appendix B, the constraint library is explained.

2 Using Partially Specified Terms

In this section, 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 [6].

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

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

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

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

This difference is an example of language efficiency [6]. How is this kind of language efficiency analyzed in CIL? We demonstrate the power of PSTs by giving a program which analyze the simplified discourse. The complete listing of the program is in the appendix.

The name of the top level predicate is discourse_constraint. Roughly speaking, 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 expected.

In this illustration, discourse constraints are simplified as the following (5) and (6):

(5) *The speaker and hearer turn their roles at every 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 declaratively gives a condition in which a parameterized object, say x, is a "discourse situation." The list of parameters consists of **sit/S, sp/I, hr/You, dl/Here** and **exp/Exp**, where the left hand side of / sign of each element in the list is the name of the parameter while the right hand side is the value of the parameter. The body of the clause is the condition for the object x to be a discourse situation in terms of the parameters. That is, x 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.

Next, let's see the discourse constraint clauses:

```
(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 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` so 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 as follows. 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 after GPSG[7]. 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 *singular*. 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 an example of the category specification in PST notation:

```
(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 Record as Function

The record is a key notion of the present paper. So we give here a formal definition of the record. First we define the domain of *finite* records. Let L and C be two sets of *labels* and *constants*, respectively. Let L^* be the set of finite strings over

L, i.e., Kleene's closure. The string also is called a *path*. The domain and range of a function r are ritten $dom(r)$ and $ran(r)$, respectively as usual. A function which has a finite domain is called a *finite* function.

A (finite) *record* is a partial finite function from L whose value is a constant or a record. A little bit more precisely, the record is defined inductively as follows.

- any finite and partial function from L into C is a record.
- if R is a finite set of records then any finite and partial function from L to R is a record.

The null function ϕ, i.e., the empty set, is a record by definition. We write $x \subset^t y$ if $dom(x) \subset dom(y)$ and either $x(a) \subset^t y(a)$ or $x(a) = y(a) \in C$ for any $a \in dom(x)$. It is easy to see that the set R of records is a partially ordered set with respect to $\subset^t$. The *merge* $x + y$ is defined to be the minimum record z such that $x \subset^t z$ and $y \subset^t z$. The merge operation is not totally defined. Under the merge operation $+$ the domain of records is a commutative and idempotent (partial) monoid:

1. $x + x = x$.
2. $x + \phi = x$.
3. $(x + y) + z = x + (y + z)$.
4. $x + y = y + x$.

We write $x!y$ for the *function application* $x(y)$. The rule for the application ! are as follows.

1. $x!\varepsilon = x$.
2. $x!(a\alpha) = (x!a)!\alpha$.

where ε is the empty string. The set of records is a lower semi-lattice with respect to the order $\subset^t$. If $z = x + y$ exists, then z is the supremum of the set $\{x, y\}$. The infimum of $\{x, y\}$ always exists.

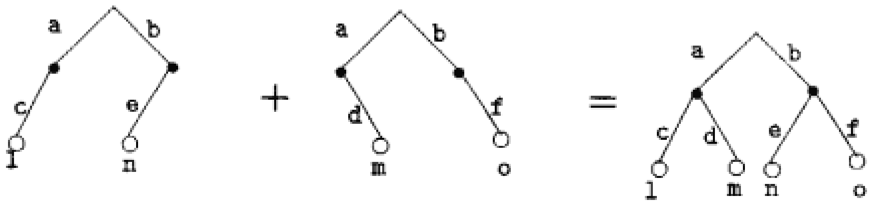

Figure 1: The merge of two records.

It is much interesting and even useful to introduce *infinite* records or equivalently say, *non-well-founded* records. The domain of *infinite* records over L and C is co-inductively defined to be the largest class R such that for any record $f\$R$ the following condition hold:

- $dom(f) \subset L$
- For any $x \in dom(f)$, it is the case either $f(x) \in R$ or $f(x) \in C$.

This definition is justified mathematically in the universe of non-well-founded sets[1].

Actually the domain of CIL is the class of infinite records. However since we treat such records only in terms of PST, which is always finite, our theory will look almost as if it treats only finite records. It should be pointed out that the use of infinite records has even advantage over that of finite ones in that there is always a solution to the constraint such as $x = \{a/x\}$ in the domain of infinite records unlike in the domain of finite records. It is because of the use of the infinite records that there is no need of 'occur check' in our algorithm of the unification which is explained later.

4 Syntax

By *first order term*, we mean the usual first order term such as in Prolog. First of all, we define a class of *terms* and *clauses* of CIL by extending the first order term. Let us fix two disjoint sets *VARIABLE* and *CONSTANT*. For the sake of simplicity, *CONSTANT* includes atomic symbols and integer constants and functor symbols all together. We follow the convention of Edinburgh Prolog[16] for variables and constants. See examples below. Also the following delimiter symbols are used in the language as usual:

```
{ } , ( ) / :- ;
```

A *term* is defined *inductively* as follows:

(1) A variable is a term.

(2) 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.

(3) If $a_1, \ldots, a_n$ are first order terms and $x_1, \ldots, x_n$ are terms then the *set* $\{a_1/x_1, \ldots, a_n/x_n\}$ of *pairs* a_i/x_i is a term.

A constant is a term by (2) with $n = 0$. A term $f(x_1, \ldots, x_n)$ of (2) is called a *totally specified term (TST)* while the term $\{a_1/x_1, ..., a_n/x_n\}$ is called a *partially specified term (PST)*. The empty set $\{\} = \phi$ is called the *empty PST*. Several functors are reserved as follows:

$,(x, y)$: conjunction $x \wedge y$.

$;(x, y)$: disjunction $x \vee y$.

$not(x)$: negation $\neg x$.

$:(x, y)$: an object x with the constraint y.

$@(x, y)$: an object x with the constraint y. (lazy version).

$\#(x, y)$: a tagged term. This is a shorthand of $:(x, x = y)$.

$!(x, y)$: a labeled term. This is a shorthand of $:(z, x = \{y/z\})$, which means the y-component of x.

$?(x)$: a frozen term. The execution of the subgoal which contains $x?$ is suspended while x is unbound.

The binary functors in the list can be used as an infix operator for readability while only some of the unary operators are written as a postfix one. For example, the two notations $:(x, y)$ and $x : y$ are equivalent.

Here are several examples of terms of CIL:

Variable `X   Man   X101   Salary   _325`

Constants `378   'Man'   x1013   abc`

TST `[1, 2, 3, 5]  3+5  f(1, abc, X)`
`soa(give, {agent/A, object/B, recipient/'Jack'}, 1)`

PST `{}  {agent/father(X), object/O, recipient/ X}`
`{f(X)/Y, f(Y)/X}`

Conditioned Term `Z@(Z>0)  X:(man(X), wife_of(X,Y), pretty(Y))`

Tagged Term `X#4  Sit#soa(R, {agent/A, soa/Sit}, P)`

Labeled Term `Man!name!first`

Frozen Term `X?  (Man!name)?`

Conjunction `(X>0, X<10)`

Disjunction `(X>0; X<0)`

Negation `(not X<Y)`

Query `?- print(X?), X=ok.`

5 Operational Semantics

A simplified operational semantics of CIL. The predicate *freeze* and *cut* are explained operationally. Although the actual operational semantics of the current CIL is sensitive like other Prolog implementations to the order of clauses in the program and atomic formulae in the body of a clause, the following semantics treats the program and goal as *sets* for the sake of simplicity. However, this restriction makes a good approximation to the real operational semantics of CIL.

5.1 Program Clause

A *program* is a finite set of program clauses. A *(program) clause* is a pair (h, b) of a term h and a set b of terms. The clause $(h, \{b_1, \ldots, b_n\})$ is written

$$h : -b_1, \ldots, b_n.$$

The term h is called the *head* of the clause while the set b is called the *body* of the clause. A *unit clause* is a program clause whose body is empty. The unit clause (h, ϕ) is written simply h.

A *query* is of the form

$$? - g$$

where g is a set of terms. The set g is called the *goal* of the query. A *subgoal* is an element of the goal.

A program is executed on the *top-down, depth-first and left-to-right* basis like 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 in the program clauses. The *rules of expanding macros* are as follows:

1. $x@c \rightarrow x : freeze(x, c)$.
2. $x\#y \rightarrow x : (x = y)$.
3. $x!y \rightarrow z : (x = \{y/z\})$.
4. $x : c \rightarrow x$. As a side effect, c is moved so that c becomes a subgoal of the clause.
5. $x? \rightarrow x$, and $g \rightarrow freeze(x, g_x^{x?})$, where g is the subgoal in the clause which contains the occurrence $x?$. By g_a^b we means the new term obtained by substituting a for all occurrences of b in the term g.

Given the program, these rules are applied according to *outer-most-first* principle until they become not applicable. For example, the clause

$$p(x\#\{a/y\}) : -b((y!c)?)$$

is rewritten into the clause

$$p(x) : -x = \{a/y\}, y = \{c/v\}, freeze(v, b(v)).$$

In the end of this rewriting process, the final form does not contain any of ?, @, :, #, and !. Thus we assume without loss of generality that a program contains no part of these macros.

5.2 State and Computation

This subsection describes the operational semantics of *freezing*, *melting*, and *negation* of goals. First of all the execution of $freeze(x, g)$ is explained as follows. If x is bound then solve immediately all the constraints in the state which has been attached so far to x including g, otherwise attach g to x suspending them till x is bound.

Before going to the operational semantics, we would like to explain the idea behind it. We see the goal as an expression evaluation of which gives The semantics of a goal is a *set* of constraints which is read disjunctive. a set of normalized constraints. This is an analogy to semantics of context free grammar in which a sentential form (a goal) yields a set of strings which is generated by the form.

There are two useful primitives, one is a choice function which select one constraint from the set of constraints the other is a conditional statement. We can define an operational negation in the language, which is more clear than usual implementation of negation by using '*cut*' and '*fail*'. $(_{(g)})$ selects only one normal constraint among other possible ones to which $C \cap g$ can be reduced by the computation rule, where is C is the current constraint. *cond*(g, g', g'') means "if choose(g) is successful then execute g' else execute g''. A negation *not*(g), for example, is defined *cond*$(g, false, true)$.

Unfortunately, CIL has not these primitives. However with this idea in mind the following explanation might be understood more easily. Roughly speaking *cut* is conceptually understood as *choose*.

We assume a fixed computation rule[11] to select the next subgoal in the body of each program clause. Also we assume a fixed way to select the alternative clause of the program at the choice point. Actually, we assume that the body of the clause is a *list* (not a set) of subgoals and that the program is a *list* of program clauses. The simplest computation rule is to select the subgoal *from-left-to-right* and to select the alternative clauses *from-top-to-down*. Based on these two assumptions we can define the operational *negation* by using *cut* primitive as described below. Unfortunately it is known that this *practical* computation rule is neither sound nor complete in the logical sense. However this problem is not pertinent to our language but still general and active research problem in foundation of logic programming. Hence implementation of logical negation is out side of the present paper. The fixed computation rule is implicit in describing the semantics below for the sake of simplicity and readability.

A *computation state* (state for short) is a triple (g, E, F) of a goal g, a set E of equations of *solved form*, which will be explained later in the next section, and a set F of pairs (v, g') of a variable v and a goal g'. Also E is called an *environment* for variable bindings. F is a set of *frozen goals*. A variable x is *bound* in the state if E has an equation of the form $x = t$ or $t = x$ for some non variable term t. Also the variable is *frozen* in the state if there is a pair (x, g) in F for some goal g.

For the given goal g, the *initial state* is the state (g, ϕ, ϕ). A pair (s, s') of states is called a *basic step* of computation, written $s \rightarrow s'$ if any of the following conditions hold. In the following rules, $\cup$ means the disjoint union if it is not specified explicitly. Also for the sake of simplicity, the rules is given in a non-deterministic way, though actually it is deterministic like the standard Prolog.

resolution: $s = (\{g_1, g_2, \ldots, g_n\}, E, F)$ and $s' = (\{b_1, \ldots, b_m, g_2, \ldots, g_n\}, E', F)$ where there is a fresh copy $a : -b_1, \ldots, b_m$ of some program clause and E' is the solved form of $E \cup \{g_1 = a\}$.

unification: $s = (\{r = r'\} \cup g, E, F)$ and $s' = (g, E', F)$ where E' is the solved form of $E \cup \{r = r'\}$

freeze: $s = (\{freeze(u, g')\} \cup g, E, F)$ and $s' = (g, E, \{(u, g')\} \cup F)$.

melt: $s = (g, E, \{(u, g')\} \cup F)$ and $s' = (g' \cup g, E, F)$, where u is bound in E.

cut: $s = (\{cut\} \cup g, E, F)$ and $s' = (g, E, F)$. As a side effect of this step, all the *alternative branches* in the computation are pruned off. A more precise explanation is given below.

disjunction: $s = (\{(g; g')\} \cup g'', E, F)$ and either $s' = (g \cup g'', E, F)$ or $s' = (g' \cup g'', E, F)$.

negation: $s = (\{not\ g\} \cup g', E, F)$ and either $s' = (g \cup \{cut, fail\}, E, F)$ or $s' = (g', E, F)$.

for $s = (\{fail\} \cup g, E, F)$, s' is undefined.

Note that there is no state s such that $(\{fail\} \cup g, E, F) \Rightarrow s$. An intuitive logical reading of the basic step $(g, E, F) \rightarrow (g', E', F')$ is that $g' \wedge E' \wedge F'$ *implies* $g \wedge E \wedge F$, though this holds only in limited sense because of our computation rule.

A *computation tree* is a tree whose nodes are labeled with states in such a way that if a node N in the tree is a successor of another node N' then $s' \rightarrow s$ must hold, where s and s' are the states on N and N', respectively.

Given the goal, a computation is the construction of a computation tree from the initial computation tree on the *top-down, depth-first and left-to-right* basis. The algorithm of the construction of the tree stops with *success* when it hits upon any state which has the empty goal and empty list of frozen goals, otherwise it stops with failure when there is no alternative state left in the tree which can be expanded further.

It is clear that each subgoal occurrence in the computation has the unique ancestor node which introduced it first. The node is called the *generating ancestor* of the subgoal. Given a computation tree T, a node N in T and an ancestor node

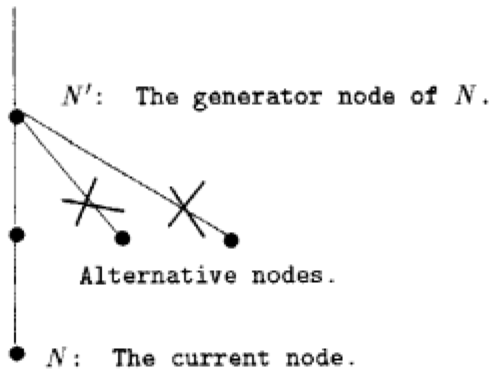

Figure 2: Cutting the alternatives.

N' of N in T, We define $cut(T, N', N)$ to be the computation tree obtained by 'pruning off' all the descendant nodes of N' which is neither an ancestor nor descendant of N. Suppose that the goal of the state at N has the *cut* subgoal and that N' be the generating ancestor on N. In this situation we state more precisely that the execution of the *cut* predicate changes the computation tree T into $cut(T, N', N)$.

6 Built–in Constraints

In this section, we explain the following basic built–in constraints : unification, boolean, arithmetic and (first order) term constraint. These built–ins work as constraint in the sense that the direction of data flow, that is, input and output depends on the context. In addition to them, a meta constraint $freeze(x, g)$ is explained in new light of constraint. The term constraint is written $u = v$ abusing the equality symbol for unification, where u and v are first order terms. Unlike the unification it never instantiates any variables of the equation. For example, the execution $x = a$ constrains x so that only a is the possible value of x.

6.1 Unification and Complexity

In this subsection we extend the standard unification to the records. The input of the algorithm is a set of equations and the output is either the set of solved forms of the input if it exists or undefined otherwise.

The unification algorithm goes as follows: Given a set of equations, repeat the following applicable steps to the set as far as possible until there is no applicable one. When it terminates, check whether there is a *conflicting* equations or not. It is easy to see that any sequence of these steps will always terminate.

(1) if $x = x$ is in the set then remove it.

(2) if $x = y$ is in the set then replace all the occurrences of y with x.

(3) if $u = x$ is in the set for a non variable u then replace it with $x = u$.

(4) if $x = u$ and $x = v$ are in the set for non–variable TSTs u and v then remove one of the equations whose size is not less than the other and add $u = v$.

(5) if $f(u_1, \ldots, u_n) = f(v_1, \ldots, v_n)$ is in the system then replace it with the n equations $u_i = v_i$ for $1 \leq i \leq n$.

(6) if $x = p$ and $x = q$ are in the system for non-variable PSTs p and q then replace them with the single equation $x = p \cup q$.

(7) if $x = u$ is in the system for a PST u and both a/v and $a/w \in u$ for different v and w then replace the equation with the following three equations: $x = (u - \{a/v, a/w\}) \cup \{a/y\}$, $y = v$ and $y = w$ where y is a new variable.

A set S of terms is called a *conflict* if one of the following conditions hold.

- S has both a non variable PST and a non-variable TST.
- S has two non-variable TSTs or constants which have different prime functors to each other.

The equation $u = v$ is called a *conflict* if the set $\{u, v\}$ is a conflict. For example, equations, $1 = 2$, $f(x) = \{a/y\}$, $f(a) = f(a, b)$ are all conflicts. By $unify(E)$, we denote the conflict-free set, only if it exists, of solved forms obtained by applying the unification algorithm to E.

Now we show that our unification problem has a UNION-FIND algorithm[2] in the following way, which is known to be an almost linear complexity algorithm. First of all, we can assume without loss of generality that any argument of terms is either a constant or variable and that no equation has non-variable terms at the both sides. If there is an equation, say $x = \{a/\{b/c\}, d/e\}$, which does not satisfy the assumption, we replace the subterm $\{b/c\}$ with a new variable y, and add the equation $y = \{b/c\}$ to the system. It is clear that by repeating this replacements we obtain in a finite number of steps a system of equations which satisfies the assumption. Also it is clear that the number of generated variables is proportional to the 'size' of the given problem.

Secondly, we transform the unification problem into the UNION-FIND problem by replacing each of the equations with union-find 'commands' according to the following rules.

- $x = u \Rightarrow x = u$.
- $x = \{a_1/u_1, \ldots, a_n/u_n\} \Rightarrow ARG^x_{a_1} = u_1, \ldots, ARG^x_{a_n} = u_n$.
- $x = f(u_1, \ldots, u_n) \Rightarrow FUN_x = f$, $ARG^x_1 = u_1, \ldots, ARG^x_n = u_n$.

where x and y are variables, u, u_i are a variable or constant, ARG^β_α and FUN^β_α are new variables.

Let Ω be the set of variables and constants which appears in the final list of commands. It is easy to see that both the size of Ω and the size of the union-find commands are proportional to the size of the input problem. Since it is well known that UNION-FIND problem has an almost linear algorithm, it follows that our unification problem has an almost linear complexity algorithm. Note that this result does not depend on whether occur-check is performed or not.

6.2 Freeze

The call of the goal $freeze(x, g)$ suspends the execution of g while x is unbound. With $freeze(x, g)$ taken as a constraint, the normalization rule of $freeze$ is given in the following two simple rules: if there are two constraints $freeze(x, g)$, and $freeze(x, h)$ in the current state for some unbound variable x and different goals h and g, then replace them with the constraint $freeze(x, g \cup h)$. For a bound variable x, replace the constraint $freeze(x, g)$ in the current state with $solve(g)$.

6.3 Boolean Constraint

Here are three predicates $and(x,y,z)$, $or(x,y,z)$, and $not(x,y)$ for Boolean algebra. Their declarative meanings are that $z = x \wedge y$, $z = x \vee y$ and $y = \neg x$, respectively. These constraint will bind values to the variables when it turns out that the only one possible solution remains in the state. For instance, suppose that the subgoal $and(x,y,z)$ was called at some time with all the three variables unbound and that now z is bound to the Boolean value *true*. Then due to the constraint $z = x \wedge y$, x and y also are bound to *true* because it is the only possible solution by the truth table. These constraint predicates are implemented based on the *freeze* primitive. In general constraint solving in the current CIL system is driven by the event of variable bindings.

The rule $\alpha \rightarrow \beta$ reads that if the constraint α is in the system then replace it with β.

- $constr(\neg(c),m) \rightarrow constr(c,m'), not(m,m')$.
- $constr(a \wedge b,m) \rightarrow cosntr(a,r), constr(b,s), and(r,s,m)$.
- $constr(a \vee b,m) \rightarrow cosntr(a,r), constr(b,s), or(r,s,m)$.

By these rules, the second argument of *constr* is used as a control parameter for specifying one of three modes of solving constraints. Let c and x be a constraint and an unbound variable. The modes of three calls $constr(c,true)$, $constr(c,false)$, and $constr(c,x)$ may be called *active, passive* and *intermediate mode* respectively. In the passive mode, the constraint solver itself never instantiates variables of the constraint c. The constraints are checked without unification when some variables are bound by other processes. In the active mode, the constraint solver executes equality constraints as unification. In the intermediate mode, the variable is bound to the value immediately when it turns out to be the unique possible value.

6.4 Arithmetic Constraint

As Boolean constraints the arithmetic constraint $add(x,y,z)$ is introduced, which means $x + y = z$. For example, this constraint will bind x to $z - y$ when both of y and z are known.

6.5 Term Constraint

The term constraint is written $constr(u = v, m)$, where u and v are first order terms and m is a variable, *true*, or *false*.

Let m and x be an unbound variable. The constraint is solved by repeating the following rewriting rules.

1. $constr(u = v, true) \rightarrow solve(u = v)$.
2. $constr(a = a, m) \rightarrow solve(m = true)$.
3. $constr(f(t_1,\ldots,t_n) = f(u_1,\ldots,u_n),m) \rightarrow and(m_1,m_2,m_3),\cdots,and(m_{n-1},m_n,m)$, $constr(t_1 = u_1, m_1)$, ..., $constr(t_n = u_n, m_n)$, where m_i is a new variable, for $1 \leq i \leq n$. for $1 \leq i \leq n$.
4. $constr(f(\ldots) = g(\ldots), m) \rightarrow solve(m = false)$,where f and g are different.
5. $constr(x = u, m) \rightarrow constr(v = u, m)$, where x is bound to the term v.

6.6 One Way Unification and Sequential Control

We explain how to allow user-defined constraints. The idea is to use the one way unification for *parameters passing* . The predicate *assign* is a built-in predicate for the one way unification. What is the one way unification? Roughly this is a *pattern matching*. Let us review the difference between the usual unification and *assign*. The standard unification works both ways. For instance, the result of $f(x,1) = f(2,y)$ is $x = 2$ and $y = 1$. On the other hand, *assign* works only a specified way: the result of $assign(f(x,1), f(2,y))$ is $y = 1$ but with x unbound.

Now we define the rewriting rule for *assign*. Let x and v be a variable and non-variable term, respectively. Also let u be any term.

1. $assign(x,u) \rightarrow assign(v,u)$, if x is bound to v.
2. $assign(u,x) \rightarrow assign(u,v)$, if x is bound to v.
3. $assign(u,x) \rightarrow solve(u = x)$, if x is unbound.
4. $assign(a,a) \rightarrow solve(true)$, where a is a constant.
5. $assign(f(t_1,\ldots,t_n), f(u_1,\ldots,u_n)) \rightarrow assign(t_1,u_1), \ldots, assign(t_n,u_n)$.
6. $assign(a,b) \rightarrow solve(fail)$, if a and b are different constants.
7. $assign(f(\ldots),g(\ldots)) \rightarrow solve(false)$, where f and g are different functor symbols.

Note that if x is unbound and u is non-variable, then there is no applicable rule in the list for $assign(x,u)$. In this case we say that the constraint *is suspended*. As a fact, $freeze(x, assign(x,u)))$ is called in the actual implementation of CIL. In general, when a constraint g has been rewritten into g' in the current state by applying the constraint normalization rule in such a way that there is no constraint in g' to be solved, then we say that g *is solved* in the state.

seqand means 'sequential execution'. $seqand(g,g')$ constrains that g' is executed only after g is solved in the sense defined just above.

Here is an example for using the one way unification and sequential 'and'. The standard membership predicate is written in Prolog as usual:

$member(x,[x|\_])$.
$member(x,[\_|y])$:-$member(x,y)$.

Let *mem* be a constraint version of *member*. In the current CIL, the definition of *mem* can be given as the following unit clause:

$$defcon(mem(x,y), seqand(assign(y,[h|t]),(x = h; mem(x,t)))) \qquad (1)$$

where *defcon* is a reserved predicate symbol.

By using *mem*, we explain how the user defined constraint is solved. The call of the constraint $mem(x,y)$ goes as follows. First of all, parameters h and t are assigned values from y by the one way unification, i.e., the pattern matching with the input parameter y, The disjunctive constraint $x = h \vee mem(x,t)$ is solved. If the term constraint $x = h$ is not solved yet, the constraint $mem(x,t)$ is called recursively. If y is unbound, due to the use of *seqand* in the body of the definition the call $mem(x,y)$ suspends not only $assign(y,[h|t])$ but also $(x = h; mem(x,t))$ while y is unbound.

Let us see what happens as the result of the following call in the *active* mode. The problem is to solve the goal (2).

$$constr(mem(x,[a]), true) \qquad (2)$$

By definition (1) of *mem*, this is equivalent to the goal (3).

$$constr(seqand(assign([a],[h|t]),(x = a; mem(x,[]))), true) \tag{3}$$

Firstly by the rewriting rule of *assign*, the constraint $assign([a],[h|t])$ is reduced to $h = a$ and $t = []$. Hence, by the *seqand* rule and the conjunction rule, the constraint (3) is reduced to the three constraints (4), (5), (6) for some new variables v and v'.

$$constr(x = a, v) \tag{4}$$

$$constr(mem(x,[]), v') \tag{5}$$

$$or(v, v', true) \tag{6}$$

The constraint (5) is reduced to $v' = false$. By the *or*–Boolean constraint rule with this binding, the constraint (6) is reduced to $v = true$. Hence from (4), now the problem has been reduced to the goal (7).

$$constr(x = a, true) \tag{7}$$

Finally, we obtain $x = a$ from the last *active* constraint (7). This result is what we expected because the constraint $mem(x,[a])$ declares that there is no other possibility for x than $x = a$.

Note that the constraint reduction goes *without backtracking* as far as *mem* is concerned. It is this point which is different from the standard execution of *member*.

7 Linguistic Analysis

In this Section, we show several ideas how to apply the record and constraint of CIL to describe linguistic analysis.

7.1 Features Co-occurrence Restriction.

We explain a typical intended use of the predicate *constr*. Let us take the following example from linguistic constraint. It says that if *REFL* feature of x is (+) then the *GR* feature of x must be *SBJ*, where x is a feature set. This is called a Feature Co-occurrence Restriction (FCR) in GPSG and written

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

In CIL, the following call of the *constr* predicate make the feature set x have the constraint.

$$constr((x!REFL = (+) \rightarrow x!GR = SBJ)).$$

The feature set x will automatically get the value *SBJ* as the *GR* feature immediately when the value (+) is generated at the *REFL* field of x. That is, the execution of $constr((x!REFL = (+) \rightarrow x!GR = SBJ))$, $x!REFL = (+)$, $a = x!GR$ yields $a = SBJ$, $x = \{REFL/(+), GR/SBJ\}$.

7.2 Record for Partial Assignment

A *state of affairs(soa)* is a triple written

$$\ll R, a, p \gg$$

where R, a, and p are a relation, partial assignment, and polarity, respectively. A relation R is given a set $Arg(R)$ of argument places. A partial assignment for R is a partial function from $Arg(R)$ which assigns objects to argument places. Each argument places has a condition which constrains the object assigned to it by the partial assignment. A partial assignment is *appropriate* if it respects these constraints[4].

Soas play important roles in studying natural language semantics in the light of situation semantics[6]. So does the partial assignment. Note that a partial assignment is just a record in our sense since it is a function. Also merge operation can be defined on the assignments. In other words, the set of partial assignments is homomorphic to the set of the records. Pollard[17] proposed a notion of *anadic relation*, which is defined to be a relation with no fixed arity. Anadic relation is an alternative to the state of affairs, that is, a relation whose arguments may not be fully saturated. Due to our view that record is a function, we can give directly a representation for both soa and anadic relation.

7.3 Complex Indeterminate

A complex indeterminate is an ordered pair (x, c), written $x|c$, of parameter x and condition c on x possibly with other free parameters in c.

For the sake of simplicity, we identify the terminologies, *indeterminate*, *condition*, and *complex indeterminate* to *parameter*, *constraint*, and *conditioned parameter* respectively. This may be safe for the purpose of the present paper.

Here is a remark on complex indeterminate in the history of CIL. A complex indeterminate was represented at earlier time as a triple $h(x, y, z)$, where h is a distinguished functor for parametric object, x is the object to be parameterized, y is a PST for the list of the parameters, and z is a condition on the parameters.

$$h(x, \{age/y\}, (man(x) \wedge age(x, y) \wedge y \leq 30)).$$

We noticed that from implementation point of view it is only convention to separate the prime and the other parameters. By introducing the conditioned term, we achieved more homogeneous representation of parametric objects. Now the above old example is simplified to

$$(\{self/x, age/y\}, (man(x), age(x, y), y \leq 31)).$$

This shift means that by means of using the record we have a uniform representation of complex indeterminates, parametric objects, and feature sets.

In CIL, a complex indeterminate is written $x : c$ or $x@c$. The behavior of the complex indeterminate is specified in the following simple rule:

$$a = (x : c) \Rightarrow unify(a, x), solve(c).$$

$$a = (x@c) \Rightarrow freeze(x, c), unify(a, x).$$

Of course this rule is yet poor to cover the theoretical richness of complex indeterminate. However we must leave it for the further development.

7.4 Multiple Inheritance with Records

As an application of record and unification, we apply it to a simplified model of multiple inheritance system.

First of all, a *class* system T is a set of *classes*. T is assumed to be partially ordered by $\leq$. An *inheritance (function) of* T is a function which assigns PSTs to each class of T. The ordered set T represents an 'is–a', hierarchy and π gives prototype attributes for each class. Let us fix T and π in the rest of this subsection. We write $H(c, x, \nu)$ for the constraint $\{x = \nu(u) | c \leq u \in T\}$. For a class c of T and a variable x, an *instance* x of c written $create(c, x)$ is defined to be a *unifiable* constraint $H(c, x, \nu)$ such that ν is a maximal inheritance function which satisfies the following conditions:

- $\nu(u) \subset^t \pi(u)$ for any $c \leq u$.
- $z = \pi(u)$ and $z = \nu(v)$ are unifiable for any u and v such that $c \leq u \leq v$.

where $\subset^t$ is the order relation defined in Section 3. Note that an instance has ambiguity corresponding to the choice of a maximal constraint which is not unique in general.

Now, let us take a simplified inheritance system, for example, $(\{bird, penguin, swallow\}, \pi)$ as follows:

$penguin \leq bird.$

$swallow \leq bird.$

$\pi(bird) = \{can\_fly/yes\}.$

$\pi(penguin) = \{can\_fly/no\}.$

$\pi(swallow) = \phi.$

This inheritance system can be written in the following CIL clauses.

$is\_a(penguin, bird).$

$is\_a(swallow, bird).$

$bird(\{can\_fly/yes\}).$

$penguin(\{can\_fly/no\}).$

$swallow(\_).$

By definition, there is some inheritance function ν such that

$$\begin{aligned} create(penguin, x) &= H(penguin, x, \nu) \\ &= \{x = \nu(penguin), x = \nu(bird)\} \end{aligned}$$

and that $\nu(bird) = \phi$ and $\nu(penguin) = \{can\_fly/no\}$. Thereby we obtain a penguin instance $x = \{can\_fly/no\}$ as 'common sense reasoning' may expect. Similarly with $create(swallow, y)$ yields $y = \{can\_fly/yes\}$.

7.5 Attitudes in PSTs

We show a simplified idea toward implementation of the attitudes theory in Barwise and Perry[6]. According to them, an attitude (mental state) is a pair of a *frame* and a *setting*. A frame is a parametric object, and a setting is an assignment or anchor. Barwise and Perry solves semantic paradoxes with this representation. Inconsistent beliefs are analyzed in terms of two mental states, say (t, a_1) and (t, a_2) which contain the same frame t but different setting a_1, a_2 respectively.

It is surprising that the proposed data structure is close to that of the *closure* in LISP or the *molecule* in Prolog of structure sharing implementation. The record in CIL gives a natural representation for the mental state as illustrated below. Suppose the following two belief contexts:

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

We represent the mental states of (1) and (2) in (3) and (4), where $beater$ and $beaten$ are indeterminates.

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

We would like to say that basic unification and utilities on records gives a useful model to search for information in the mental state representation given that mental states are represented in records. We show this by giving queries to the above two beliefs.

(5) *Who believes taro is the beater?.*
(6) $? - believe(x, \{beater/taro\})$ yields $x = jack$.
(7) *Who does jack believe is beaten?*
(8) $? - believe(jack, \{beaten/x\})$ yields $x = hanako$.
(9) *What does jack believe taro does?*
(10)$? - m = \{frame/z, a/taro\}, believe(jack, m)$ yields $a = beater$, $z = beat(beater, beaten)$, $m = \{frame/beat(beater, beaten), beater/taro, beaten/hanako\}$. This answer contains information that $taro$ is the $beater$.

7.6 DAG and Record

DAG in unification grammar and record in CIL are very close to each other, as is pointed out in the literature based on intuitive argument, Shieber[19] for instance. We support this closeness by giving a simple explicit translation from DAG into a constraint over records.

Let d be a DAG $(N, A, C, \alpha, \delta, \gamma)$ where N is a set of nodes, A is a set of arrows, C is a set of constants, both δ and γ are functions from A to N, and α is a function from N to C.

The translation $f(d)$ is defined to be the set

$$\{x = u | x \in N, a \in A, \delta(a) = x, \gamma(a) = y, u = \{a/y\}, \text{ or } \alpha(x) = u\}$$

of equations.

$$\begin{tikzcd} & x \arrow[dl, "a"'] \arrow[dd, "c"] \\ y \arrow[dr, "b"'] & \\ & d \end{tikzcd} \qquad \Rightarrow \qquad \begin{aligned} x &= \{a/y\} \\ x &= \{c/d\} \\ y &= \{b/d\} \end{aligned}$$

Figure 3: A DAG as a Constraint

As is seen in the translation, the vocabulary of the constraint language over records are the following: C is the set of *constant* symbols, N is the set of *variable* symbols, and A is the set of *labels* on edges.

It is easy to see that the two DAGs d and d' can be merged if and only if $f(d) \cup f(d') \cup \{x_d = x'_d\}$ are unifiable over the set of finite records, where x_d and x'_d are the *root* node of d and d', respectively. Therefore DAG unification can be seen as a constraint solver over the records.

The notion of *structure sharing* in DAG-based theory is reduced to that of sharing the logical variables in our constraint language. One of the consequence of this translation is that the notion of structure sharing belongs only to the constraint language. That is, it is not a property of the objects. This also seems to raise a question whether structure sharing is an essential linguistic relation or not. The author has not the answer yet.

Since a record may be infinite, it can represent more complex structure than (finite) DAG. In particular, the records domain may be suitable for representing and processing circular situation proposed by Barwise[5] in conjunction with ordinary linguistic information.

8 Concluding Remarks

A dynamic record structure and its unification was introduced into logic programming based on constraint. Several promising ideas of application of the record has been demonstrated for linguistic analysis.

Although record structure is an extension of the standard term by just forgetting the fixed arity, the whole merits of this extension seems more profound than usually thought. In the introduction of the present paper, we has pointed out several source of the power of the record for a variety of application.

Several basic built–in predicates for constraints such as arithmetic, term, and Boolean were described in a uniform way such that even the meta constraint *freeze* was explained in a very simple rewriting rule.

At the current stage, our theory of record lacks several important aspects. For instance, our logical disjunction and negation depends on incomplete computation rules for practical purpose.

We have shown several new ideas how to use records to represent parametric objects motivated from situation semantics and dependent type theory. For instance, we have implemented a toy version of built-in predicates both to define and to instantiate the type. Of course, the current level of treatment is far from satisfiable one. The idea of parametric objects is attractive to represent and handle more sophisticated linguistic information or situated information in more general sense of situation semantics. We would like to develop our system to be able to handle some significant portion of the theory of situated information.

Acknowledgments

Jon Barwise, Joseph Goguen, and Carl Pollard gave me useful comments to the earlier version of the present paper. Koichi Furukawa has been encouraging me by showing his interests and insights in the record structure. Koichi Hasida and Hideki Yasukawa has enhanced the readability of the present paper by giving many comments. Toshio Yokoi and Shunichi Uchida supported this research as the former and current research managers of Second Laboratory of our institute. I would like to thanks them for their various contributions to this work.

References

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

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison–Wesley, 1974.

[3] H. Aït-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.

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

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

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

[8] J.Bresnan and R.Kaplan. Lexical–functional grammar: a formal system for grammatical representation. In J. Bresnan, editor, *The Mental Reprsentation of Grammatical Relation.* Cambridge, Mass.: MIT Press, 1982.

[9] M. Johnson. *Attribute–Value Logic and the Theory of Grammar.* CSLI Lecture Notes 16. Center for the Study of Language and Information, Stanford University, 1987.

[10] R.T. Kasper and W.C. Rounds. A logical semantics for feature structures. In *Proceedings of the 24th Annual Meeting of the ACL, Columbia University.* ACL, 1986. New York, N.Y.

[11] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

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

[13] K. Mukai. Partially specified term in logic programming for linguistic analyis. In *Proceedings of International Conference on the Fifth Generation Computer Systems.* Institute for New Generation Computer Technology, 1988. also appears as ICOT-TM 568, 1988.

[14] K. Mukai. Merge structure with semi-group operation and its unification theory (in japanese). *Journal of Japan Society for Software Science and Technology,* 7(2), 1990. also appears as ICOT-TR 480, 1989.

[15] K. Mukai and H. Yasukawa. Complex indeterminates in prolog and its application to discourse models. *New Generation Computing,* 3(4), 1985.

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

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

[18] S. Ryoichi, H. Miyoshi, and K. Mukai. Constraint analysis on Japanese modification. In *the proceedings of natural language understanding and logic programming.* North–Holland, 1987.

[19] 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, CSLI, April 1986.

[20] G. Smolka. Feature constraint logics for unification grammars. Technical Report IWBS report 93, IBM Deutschland GmbH, 1989.

[21] G. Smolka. Feature logic with subsorts. Technical Report LILOG Report 33, IWBS, IBM Deutschland, Postfach 80 08 80, 7000 Stuttgart 80, W. Germany, May 1989.

A Tiny Discourse Analysis in CIL

```
/* Abbreviations
        sit: situation
        sp:  beaker
        ip: interpretation
        hr: hearer
        dl: discourse location
        exp: expression
        soa: state of affairs
        ag: agent
        ob: object
        syncat: syntactical category
        head: head feature
        subcat: subcategorization */

% Discourse Situation
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).

% Membership
member(X, [X|Y]).
member(X,[Y|Z]):-member(X,Z).

% Discourse Constraint
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]).

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

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

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

% DCG (Definite Clause Grammar)
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}).

% Lexical Items
noun({ip/jack,
     syncat/{head/{minor/{agr/{plu/(-),per/3}@agr}}}@category})
```

```
        -->[jack].          % Jack
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
noun({ip/X,
      ds/{hr/X},
      syncat/{head/{minor/{agr/{plu/@plu, per/2}@agr}}}@category})
        -->[you]            % you
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

% Syntax  Categories
category({bar/ @bar, head/ @head}).
head( {major/ @major, minor/ @minor}).
major({n/ @n, v/ @v}).
minor({agr/ @agr, case/ @case}).
agr({per/ @per, plu/ @plu}).

case(accusative).
case(nominative).
bar(1).
bar(2).
bar(3).
n((+)).
n((-)).
v((+)).
v((-)).
plu((+)).
plu((-)).
per(1).
per(2).
per(3).
```

B Built-in Predicates

Built-in functions in CIL are listed below with example uses. Only relevant predicates to either PSTs or constraint are listed because other ones just follow those of the standard Prolog. By $\alpha(x)$ we means the term t such that $x = t$ is in the environment. For convenience, if x is unbound then $\alpha(x) = x$. Suppose that u be a variable and the equation $u = p$ is in the environment for some PST p. In this situation, by $u(x)$ we means $p(x)$ for convenience.

B.1 Extended Unification

unify(t,u) This call unifies t with u. This is written also $t = u$ for short notation. The execution of the three equations $x = \{a/1\}$, $y = \{b/2\}$, $x = y$ yields $x = y = \{a/1, b/2\}$. Also $\{a/b, c/\{d/x\}\}!c!d = h$ yields $x = h$. The execution of $x = \{a/ok\}$, $y = \{a/@print\}$, $x = y$ displays ok, where $print$ is a built–in output predicate. The execution of $\{a/x, b/c\} = \{a/1, b/y\}$ yields $x = 1$, $y = c$. The execution of $x\#\{a/1, b/x!a\} = y$ yields $x = y, x = \{a/1, b/1\}$. The execution of $x = \{a/b, c/y\}$, $y = \{a/b, c/x\}$, $x = y$ yields, $x = y = \{a/b, c/x\}$. The execution of $x = \{a/b, c/y\}$, $y = \{a/b, c/x\}$ yields $x = \{a/b, c/y\}$, $y = \{a/b, c/x\}$.

assign(u,v,z) This call makes matching u with v by the one way unification. If the matching is successful, then return $z = true$ else return $z = false$ if matching fails, otherwise leaves z unbound. The last case means that the process has been suspended to wait u is getting more instantiated. The variables in u are treated as read-only variables. The execution of $assign(f(x), f(y), z)$ yields $z = true$, $x = y$. The execution of $assign(f(x), f(a), z)$ yields no bindings, i.e., x and y remain unbound. The execution of $assign(f(a), f(x), z)$ yields $x = a$, $z = true$.

B.2 Utilities

same(u,v) This call checks whether u and v are equal at the current state or not. The execution of $same(\{a/\{b/\_, c/\_\}\}, \{a/\{c/1, b/b\}\})$ fails. The execution of $\{a/A\#\{b/\_, c/\_\}\} = \{a/B\#\{c/1, b/b\}\}$, $same(A, B)$ succeeds.

dif(u,v) This call constrains that u and v are different from each other.

fullCopy(u,v) This call makes a fresh copy of u and unifies v with it. Even the frozen conditions accumulated on the variables in u are copied. The execution of $x = \{a/@print, b/x\}$, $fullCopy(x, z)$, $z!b!a = ok$ will display ok on the output screen.

typeOf(u,type(v,w)) This call is equivalent to the execution of $fullCopy((v, w), (u, z))$, $solve(z)$.

createType(u, v, type(w,z)) This call is equivalent the execution of $fullCopy((w, z), (u, v))$. The execution of $createType(y, (y = 1; y = 2), t)$, $typeOf(1, t)$, $typeOf(2, t)$ succeeds.

instance(u,v) This call is equivalent to the execution of $fullCopy(v, w)$, $unify(w, u)$.

B.3 Record Utilities

Here are utilities for handling records.

getRole(u, k, v) This call is equivalent to the following execution:

$$(k = k_1; \cdots; k = k_r), u = \{k/v\}$$

where $dom(\alpha(u)) = \{k_1, \ldots, k_r\}$ in the state.

Less formally said, this call finds the key k in u to return the value v of the key. This is equivalent to the unification $u = \{k/v\}$ in declarative sense. No argument place of u is created. This predicate may have backtrack points. The key k does not need to be ground. This predicate is similar to the predicate *locate* below. In the case that k is known to be ground, the predicate *locate* is more efficient than this. The execution $x = \{a/1, b/2\}$, $getRole(x, k, v)$ yields $k = a$, $v = 1$ as the first solution and then $k = b$, $v = 2$ as the second one.

locate(u,k,v) If k is bound to a ground first order term and $k \in dom(u)$ then solve $u = \{k/v\}$ otherwise fails.

This is similar to the predicate *getRole* above. However k must be ground. The execution will fail if u has not the argument place named k. The execution of $locate(\{a/b\}, a, x)$ yields $x = b$. The execution of $locate(\{a/y\}, b, x)$ fails, where $a \neq b$. The execution of $locate(\{a/y\}, a, x)$ yield the unification $x = y$.

setOfKeys(u, s) This call makes the list of keys in the record u and return it to s. The execution of $setOfKeys(\{a/x, b/y, c/z\}, s)$ yields $z = [a, b, c]$.

role(k, u, v) This is a constraint version of *getRole*, which declares that $getRole(u, k, v)$ is executed when k is bound to a ground first order term. The execution of $x = \{a/1, b/2\}$, $role(k, x, 3)$, $k = c$ yields $k = c$, $x = \{a/1, b/2, c/3\}$.

delete(k,u,v) This call generates the conjunctive constraint of $v = u'$ and ' u' is the restriction to the $dom(u) - \{k\}$' which is to be solved when the value of k is grounded. Intuitively this call deletes the k-field from u. The execution of $delete(a, \{a/1, b/2\}, z)$ yields $z = \{b/2\}$.

partial(u) This call succeeds if u is a record otherwise fails.

record(u,v) Provided that $u = \{a_1/b_1, \ldots, a_n/b_n\}$ in the state, this call generates and solves the following constraints.

$$\begin{aligned} v &= [(a_1, b_1)|v_1] \\ v_1 &= [(a_2, b_2)|v_2] \\ &\vdots \\ v_{n-1} &= [(a_n, b_n)|v_n] \end{aligned}$$

In other words, this call makes the list consisting of pairs (p, w) such that $u = \{p/w\}$ and return it to v. This predicate is similar to $buffer$. v is generated as a stream from the record u. This predicate is used as a stream generator. The execution of $record(\{a/1, b/2\}, r)$ yields $r = [(a, 1), (b, 2)]$. This call is dangerous if there is possibility that the record u grows in the further state.

buffer(u, v) Provided that $u = \{a_1/b_1, \ldots, a_n/b_n\}$ in the state, this call generates the following constraints.

$$v = v_0, v_0 = [w_1|v_1], \cdots, v_{r-1} = [w_r|v_r].$$

$$\text{If } 0 \le j \le n-1 \text{ and } j \le r \text{ then } w_j = (a_{j+1}, b_{j+1}).$$

$$\text{If } n \le r \text{ then } w_n = end\_of\_list.$$

These constraints are solved in such an incremental way that every time when any variable v_i gets instantiated the corresponding unification in the above is performed.

Less formally said, the call $buffer(u, v)$ converts the record u to the buffered list v. This is similar to the predicate *record*. Each pair (k, s) in u, i.e., $u = \{k/r\}$, is put on v as the last element of v while there is room in the list v. The *end_of_list* marker is put when the pairs in u is exhausted. If the current tail of v is unbound then the producing of the rest is suspended. The execution of $buffer(\{a/1, b/3\}, x)$, $x = []$ succeeds. The execution of $buffer(\{a/1, b/3\}, x)$, $x = [y|z]$, $z = [u|v]$ yields $x = [(a, 1), (b, 3)|v]$. The execution of $buffer(\{a/1, b/3\}, [x, y, z, u])$ yields $x = (a, 1)$, $y = (b, 3)$, $z = end\_of\_list$.

glue(u,v) This call executes the unification $u = \{k/z\}$ and $v = \{k/z\}$ for each common key k of u and v. The execution of $x = \{a/1, b/z\}$, $y = \{b/2, c/3\}$, $glue(x, y)$ yields $x = \{a/1, b/2\}$, $y = \{b/2, c/3\}$, $z = 2$.

merge(u, v) This call is just the unification $v = \alpha(u)$. In other words, this call adds to v each element of u. The execution of $x = \{a/1\}$, $y = \{b/2, c/3\}$, $merge(x, y)$ yields $x = \{a/1\}$, and $y = \{a/1, b/2, c/3\}$.

d_merge(u, v) This call is the unification $v = u'$ such that u' is 'a maximal'[3] restriction of u as a function such that the unification $v = u'$ is successful.

In other words, this call adds to v each element of u which is unifiable with the counter part in u if any. The execution of $x = \{a/1, b/2\}$, $y = \{a/3\}$, $d\_merge(x, y)$ yields $x = \{a/1, b/2\}$, $y = \{a/3, b/2\}$.

subpat(u,v,d) This call checks $dom(u) \subset dom(v)$ and unifies d with the list of the triples $(k, u(k), v(k))$, where $k \in dom(u)$.

In other words, this call creates onto d the list of triples (k, r, s) such that (k/r) is in u and (k/s) is in v. If u is not a subrecord of v in the sense that each key of u is also the one of v, this call fails. The execution of $subpat(\{a/1, b/x\}, \{a/y, b/2, c/3\}, z)$ yields $z = [(a, 1, y), (b, x, 2)]$.

extend(u,v,d) This call unifies d with the list of triples $(k, u(k), v(k))$, where $k \in dom(u) \cap dom(v)$. Also this call unifies v with the restriction of u to $dom(u) - dom(v)$.

In other words, this call adds each element of u to v and return in d the difference list between u and v. The execution of $x = \{a/1, b/2\}$, $y = \{b/z, c/3\}$ yields $y = \{a/1, b/z, c/3\}$, $d = [(b, 2, z)]$.

[3] I am not sure that we can safely put "the maximum" for it.

meet(u,v,d) This call creates the triples $(k, u(k), v(k))$ onto the D-list d for $k \in dom(u) \cap dom(v)$. The execution of $meet(\{a/1, b/2\}, \{b/3, c/4\}, x - [])$ yields $x = [(b, 2, 3)]$.

frontier(u,v,d) This call creates the triples $(k, u(k), v(k))$ onto the D-list d for $k \in dom(u) \cap dom(v)$. This predicate fails if $u(k)$ and $v(k)$ are a conflicting pair for some k. It is similar with the case of TST concerned. The execution of $frontier(f(a, g(b)), f(x, y), z - [])$ yields $z = [a = x, g(b) = y]$. The execution of $frontier(a, b, z - [])$ fails.

match(u,v,d) This call creates the triples $(k, u(k), v(k))$ onto the D-list d for $k \in dom(u) \cap dom(v)$ such that $u(k) \neq v(k)$. This predicate always succeeds. The execution of $match(f(a, x, c), f(b, x, z), u - [])$ yields $u = [a = b, c = z]$.

t_subpat(u,v) This call checks whether u is a *hereditary* subrecord of v, i.e, u is a pattern of v, provided that both u and v are ground record. More formally, it is checked that $dom(u) \subset dom(v)$ and for any x in $dom(u)$ it is the case that either $u(x)$ and $v(x)$ are the same first order term or $t_subpat(u(x), v(x))$. This call will fail if it is not the case. This predicate is a transitive version of *subpat*.

The execution of $t\_subpat(\{a/\{b/1\}\}, \{b/1, a/\{c/2, b/3\}\})$ succeeds. The execution of $t\_subpat(\{a/\{b/3\}, c/4\}, \{b/1, a/\{c/2, b/3\}\})$ fails.

t_merge(u,v) This call merges u into v in the transitive way. This predicate is a transitive version of *merge*. In other words, this call generates the constraints $t\_merge(u(a), v(a))$ for each $a \in dom(u) \cap dom(v)$, and execute $v = u'$ where u' is the restriction of u to $dom(u) - dom(v)$.

The execution of $t\_merge(\{b/1, a/\{c/2, b/3\}\}, \{a/\{b/y\}\})$ yields $y = 3$.

masked_merge(u,m,v) This call creates u minus m and then merge them into v. The execution of $v = \{a/2\}$, $masked\_merge(\{a/1, b/1, c/1\}, \{a/\_, b/\_\}, v)$ yields $v = \{a/2, c/1\}$.

B.4 Meta Predicate

bound(u) This call just succeeds if u is already instantiated.

unbound(u) This call just succeeds if u is not bound.

B.5 Control Predicates

freeze(x, g) Suspend the goal g while x is unbound.

freeze(x,y,g) Suspend the goal g while both x and y are unbound.

if(t, g) If $t = true$ then solve g else fails.

if(t,y,z) If t then solve y else solve z.

ifBound(x, g) If x is bound then $solve(g)$.

ifUnbound(x, g) If x is unbound then solve g.

wif(x,y) Suspend the goal y until x is bound. When x is bound, y is called if $x = true$ otherwise fails.

wif(x,y,z) Suspend the possible execution y and z until x is bound. When x is bound, y is called if $x = true$ otherwise z is called.

pv(f,g) If f is unbound then this call makes $f = true$ and solve g else this call just succeeds.

solve(g) Solve g.

when(a, g) Suspend the goal g while a is not ground.