TR-101

Horn Clause Logic with Paramenterized Types for Situation Semantics Programming

Kuniaki Mukai

February, 1985

Horn Clause Logic with Parameterized Types for Situation Semantics Programming

by

Kuniaki Mukai

Third Research Laboratory
Research Center
Institute for New Generation Computer Technology

Mita Kokusai Building, 21F.
4-28, Mita 1-Chome, Minato-ku, Tokyo 108 Japan

Abstract

In this paper, we give an outline of a programming language. The main motivation behind the language is to describe computational models of discourse understanding [Brady & Berwick 1982] based on situation semantics [Barwise & Perry 1983, Barwise 1984]. The underlying logic of the language is Horn clause logic. Parameterized types and indeterminates are added to logic programming to represent objects in situation semantics. As a basic control feature to process demons, the language uses the control primitive called "freeze" in Prolog II [Colmerauer 1982], also known as bind hook control. The language can be expressed by the term

```
  Horn clause logic
+ parametarized types and indeterminates
+ bind hook control.
```

The language has the clear denotational semantics and operational semantics. We have implemented an experimental version of the language on top of DEC-10 Prolog [Bowen 1981]. Some program examples are included.

Contents

1. Introduction

As is well known, knowledge and contexts play important roles in discourse understanding process models [Brady & Berwick 1982]. We express knowledge and contexts in the form of parameterized types (types for short) and situations, respectively, where situation is taken in the sense of situation semantics [Barwise & Perry 1983, Barwise 1984].

Our observations on discourse understanding and situation semantics are as follow:

1) Creating computational models of discourse understanding is a very difficult problem in natural language processing. The central difficulty is how to develop models for the efficiency of language.
2) Situation semantics is a promising approach which provides a frame-work for discourse understanding.
3) The theory of situation semantics has been formalized as a kind of type theory [Barwise 1984].
4) There are efficient implementations of Horn clause logic on several computers.
5) "Demon" is one of the standard control models of discourse understanding processing.

With these facts in mind, we have written a situation semantics based programming language for the discourse understanding system here at ICOT [ICOT 1985].

The language is called CIL (Complex Indeterminates Language). CIL is built around a Horn clause logic and includes types. CIL has the basic control feature called "freeze" in Polog II [Colmerauer 1981] or bind hook control . In CIL, demons are defined by using "freeze" statements. CIL can be approximately defined by the equation

```
        CIL =  Horn clause logic
             + types and indeterminates
             + bind hook control.
```

CIL is a simple language with easily understandable semantics. We believe that CIL has the capacity to describe the objects in discourse understanding processes by using types, indeterminates and conditions. One of the major reasons for this is that CIL expresses situations as contextual parameters in a uniform manner to describe context-dependent concepts. Using the type of discourse situation, the meanings of the words "I" and "you", for instance, are defined to denote the speaker and the hearer respectively in any given discourse. Actions in plan-goal problems, to take a sophistcated example, could be handled as instances of complex types which have precondition, action body, and postcodition slots. Note that preconditions and postconditions are conditions on contexts. Now, we take context to be the same as situation, which makes CIL a situation semantics oriented programming language.

Our goal is to apply CIL to discourse understanding based on situation semantics as a test domain. So first, let us take a brief look at the principles of discourse understatnding based on situation semantics.

1) Each lexical element has a type.
2) A grammar has a function which synthesizes the type of a sentence from these terminal types according to the parsing tree of the sentence. Types play a smilar role to that of intentional formulas in the semantics of Montague grammar; that is, they conform to Frege's compositionality principle.
3) Let d, c, and e be indeterminates of the types discourse situation, connective situation, and described situation, respectively. A discourse situation has a discourse location, speaker, hearer, and expresssion as parameters. A connective situation involves information such as "who" refers "what" to "what", "who" presupposes "what", "what" are antecedents for pronouns, and other contextual informations. There is a condition M on these parameters d, c, and e such that the meaning of an expression s uttered in the discourse situation d is defined to be the situation type Q, where

   $$Q = [e | M(d,c,e)].$$

4) Discourse understanding is, formally, a condition DU on the sequences of complex indeterminates,

   $$DU(<q1, q2, ..., qn>),$$

   where qi is an instance of the described situation type Qi,

   $$Qi = [ei | M(di,ci,ei)], (1=<i=<n).$$

Thus, our formalized discourse understanding system is given by the two major conditions M and DU. The motivation behind CIL is to provide a simple and efficient programming language for describing both of DU and M. An illusrative version of M and DU will be described in the following section 2.

2. Utterance and Discourse Situation

An illustrative program named "du" is described in detail, which shows some ideas of situation semantics programming. "du" is a fragment of the discourse analysis model DU described in the introduction. This example includes the following features in discourse models:

1) definitions of lexical elements as types,
2) a definition of a grammar including syntax and semantics descriptions, as constraints over these types,
3) the definition of discourse situation as a type,
4) the definition of meaningful option as a type,
5) the simple conversational constraint over discourse that the roles of hearer and speaker change from the one to the other in turn, and
6) the meaning of a sentence depends on the discourse situation.

The last feature will be demonstrated by the sentence "i(=I) love you". The sentence will have two interpretations depending on the discourse situations involved.

The definitions of discourse situation and meaningful option are taken from [Barwise & Perry 1983]. No connective situations are included in the example.

The notations {X¦C} and [H¦T] are used for type and list respectively to avoid confusion between them. These conventions are only in the section. The term [H¦T] is the list whose head and tail are H and T. [] is for the empty list. The form

```
(L,(R, A1, ..., An), P)
```

is for the abstract located fact that the objects A1, ..., An stand (P=1) or do not stand (P=0) in the relation R at the location L.

The type of discourse situation

```
discourse_situation = {S ¦ C}
```

, where

```
C = in(S, (Here, (speaking,I), 1))&
    in(S, (Here, (addressing, You),1))&
    in(S, (Here, (utter, Exp),1)),
```

is translated directly into CIL as follows:

```
dcl(discourse_situation, S ,
      (speaker:I, hearer:You, disc_loc:Here, expression:Exp),
      in(S, (Here, (speaking,I), 1))&
      in(S, (Here, (addressing, You),1))&
      in(S, ((Here, (utter, Exp),1))
        ).
```

It is written as in(S, X) that the set S includes X as a member. The symbols "speaker", "hearer", "disc_loc", and "expression" are the names of the parameters given by the user.

The following is an example description of the type of meaningful option of the sentence.

```
dcl(meaningful_option, sem!SCAT, (ds: ds!SCAT),
      sentence(SCAT, expression!ds!SCAT, [])
     ).
```

The term ds!SCAT referes to the value of the "ds" slot in the complex indeterminate SCAT. Generally speaking, the term of the form S!X refers to the value of the slot S in the complex indeterminate X. A variable can be written for S. The equality "Betty = mother ! Jack" can be read as Betty is the mother of Jack.

The type definition expresses that the meaning of the sentence depends on the discourse situation. The constraint "sentence" defines the syntax and semantics of the example language.

The syntax categeries in the example language are "sentence", "noun", and "verb". The lexicons are "i","you", and "love". Each category is defined by the predicate with the same name of it. Each predicate has three arguments. The first argument is an indeterminate which has all or part of the following slots depending on the predicate.

```
1) ds   : the discourse situation,
2) agent: the agent case,
3) obj  : the object case,
4) sem  : the semantics of the category itself, and
5) sit  : the situation to which the category has some contribution.
```

The last two arguments together as a difference list represent the given sequence of lexicons. Let p and x be a syntax category symbol and a set of features and let h and t be two sequences of words. The atomic formula of the form p(x,h,t) means that the sequence of the words given as the difference between h and t is in the phrase category p and that x is the features of the sequence.

The sentential forms and its semantics which are allowed in the example language are described by the predicate "sentence". Any sentence must be the sequence of a noun, a verb, and another noun in order. The initial part of the right hand side of the definition describes the necessary unifications among the slots in the indeterminates appearing in the rule. The sentence category is assumed to have a discourse situation and a described situation in the "ds" and "sem" slots. The clause of the form H<-B means that B implies H. The symbol "&" is for the logical conjunction.

```
        sentence(S,A,B)<-
                D= ds!S &
                ds!X = D &
                ds!Y = D &
                ds!V = D &
                sem!S = sem!V &
                sem!X = agent!V &
                sem!Y = obj!V &
                noun(X,A,A1) & verb(V,A1,A2) & noun(Y,A2,B).
```

The noun category is assumed to have a context and an object in the "ds" and "sem" slots. The object is the denotation of the noun phrase and appears in the context.

```
        noun(X,A,B)<- i(X,A,B).
        noun(X,A,B)<- you(X,A,B).
```

The verb category is assumed to have several semantic features. The values of these features are in the "sit", "ds", "obj", "agent" slots. This expresses that that the "agent" and "obj" stand in the relation given by the verb in the situation "sit". The discourse situation "ds" provides the location of the circumstance.

```
        verb(X,Y,Z)<- love(X,Y,Z).
```

The followings are the definitions of the lexicons. The first clause defines the pronoun "i" as the speaker of the given discourse situation. The value of the "i" will be unified with the "sem" slot of the indeterminate of the first argument as the semantic value of "i" itself. The others are defined in similar ways.

```
        i(X,[i|A],A)<-
                sem!X = speaker!ds!X.
        you(X,[you|A],A)<-
                sem!X = hearer!ds!X.
        love(X,[love|Z],Z)<-
                in(sit!X, (disc_loc!ds!X,(love, agent!X, obj!X), 1)).
```

The predicate "du" is the constraint over the conversational discourses in the example language. The general form of the use of the predicate is written as "du(X, Y, Z, Exps, MOs)". X and Y are the two participants of the discourse. Z is the discourse location of the beginning of the conversation. Exps is the sequence of all the sentences spoken by X or Y. MOs is a variable for the sequence of the meaningful options of these sentences in the corresponding order. The constraint concerning the conversational roles is described by the predicate "change_role" below.

```
        du(X,Y,Z,Exps,MOs)<-
                instance(DS, discourse_situation,
                        (speaker:X, hearer:Y, disc_loc:Z)) &
                du1(DS, Exps, MOs).
```

The occurrence of the predicate "instance" means that DS is an instance of the type discourse_situation with the speaker X, hearer Y, and disc_loc (for discourse_location) Z.

```
        du1(D,Exp,MO)<-
                \+ Exp= _*_ &
                Exp= expression!D &
                instance(MO*_ , meaningful_option, ds:D).
        du1(D,Exp*R,MO*S)<-
                Exp= expression!D &
                instance(MO*_ , meaningful_option, ds:D)&
                change_role(D,Next) &
                du1(Next,R,S).
```

The symbol "\+" is for negation. The symbol "*" is for the stream constructor. The symbol "_" is for anonymous variables.

The following is the conversational "maxim" describing that the roles of the speaker and the hearer must change with each other in turn every time the speaker has uttered each sentence. The special form is introduced to represent successive discourse locations for convenience. Location is outside of the scope of the paper.

```
        change_role(D, Next)<-
                instance(Next, discourse_situation,
                         (speaker:hearer!D,
                           hearer:speaker!D,
                         disc_loc:next(disc_loc!D)     )).
```

Execution is as follows:

```
          > du(jack,betty,loc01,
          >                       [i,love,you]*[i,love,you],MOs)&
          >  print(MOs).
          (loc01,(love,jack,betty),1)*
          (next(loc01),(love,betty,jack),1)
          success
```

The result shows that the same sentence has the different meanings depending on the discourse situations.

3. Term, Predicate and Clause

CIL is Horn clause based programming language like DEC-10 Prolog for instance. A CIL program consists of Horn clauses and definitions of types. CIL has predicates for "freezing" goals and for instantiating and accessing indeterminates. CIL has a conditional statement like the one in Lisp, but it does not have the "cut" statement.

The characteristic built-in statements of CIL are X^Y ("freeze"), V=A!X (slots accessing), dcl(X,Y,Z,U) (type declaration), and instance(X,Y,Z,U) (instantiation).

3.1 Variable and Indeterminate

A symbol which begins with a capital letter is a variable. An indeterminate in the sense of situation semantics is represented by the special form of internal data structure. The data structure is also called indeterminate. Let us make a convention that a variable is an indeterminate.

3.2 Terms

An integer is a term. An atom symbol is a term. For a functor symbol f and terms a1, ..., an, the form f(a1, ..., an) is a term. Every term falls in one of these categories. The term of the form (A!B) is a notation for the term C such that role_of(A, B, C). The atomic formula role_of(A,B,C) can be read that C is the value of the slot A in the complex structure B.

3.3 Clauses

A term of the form H <- B is called clauses. H is the head of the clause. B is the body of the clause.

3.4 Logical Connectives

Let X and Y be any terms. The termes of the form X & Y, X \/ Y, \+ X , and ~X are called a conjunction, disjunction, negation, and delayed negation.

3.5 Conditional Terms.

Let Ci and Xi be any terms (1=< i =<n and 0<n). The term of the form

```
{C1=>X1, C2=>X2, ..., Cn=>Xn }
```

is called a conditional term. Its reading is that if C1 then X1 else if C2 then X2 else ... else if Cn then Xn. Some conventions are used. For any i<n, Ci=>Xi is written simply as Ci if Xi is "true". Cn=>Xn can be written simply as Xn if Cn="true".

3.6 Logical Constants

The predicate true and false are the system ones which always succeed and fail respectively.

3.7 Freezing

The control primitive "freeze" is a means to describe demons. The system predicate "^" freezes goals. For the variable X and goals Y, the form X^Y freeze the goals Y until the variable X is bound.

3.8 Meta Call

The system predicate solve is a meta predicate. The form solve(X) is to solve the goals bound to the variable X.

3.9 Unification and Slot-filling

The form X=Y is to unify X with Y. The form V=S!X which is the notation for role_of(S,X,V) is to unify V with the value in the slot S of X. The symbol S can be a variable. The form X:= (Y,S,C) unifies X with the complex indeterminate Y|C(S) for the given indeterminate Y and condition C with the slots S.

3.10 Test Predicates

CIL has following system test predicates.

bound(X) : X is bound.
unbound(X) : X is unbound.
indeterminate(X): X is an indeterminate.
atom(X) : X is an atom.
integer(X) : X is an integer.
atomic(X) : X is atomic.
same(X,Y) : X is identical to Y.

3.11 Type and Instantiation

A type can be declared by the predicate dcl and be instantiated by the predicate instance.

dcl(T,X,P,C) : T is the name of the type [X|C], and P are parameters of C. This is the static way of type declaration.

More detailed explanations follow. Let N, X, C be a name, indeterminate, and condition. And let T be the type definition of the form

```
        dcl(N, X, (S1,S2,...,Sn), C)
```

where Si is a slot of the form Ni:Xi (1=<i=<n). The definition T means that N is a name of the type

```
        [X| C].
```

Let T be a type of the form [X|C(X,P1,...,Pn)]. In CIL, the parameters P1, ..., Pn have their "slot names". The user can also supply a name for the type T. Slots in the indeterminate can only be accessed by CIL's built-in procedures. Indetermiates appear to users as abstract objects.

An instance of the type T has a copy C' of the constraint C. The constraint C' is executed as soon as the instance is created. This simple way of solving constraints does not restrict CIL's power since CIL has the primitive "freeze" for suspending goals if necessary until a specified variable is bound.

The basic built-in "instance" has the several forms of uses as follows. Its precise definition is given in the Appendix 2. One way of use is to create an instance through the type definition. The other is to create it dynamically.

0) instance(X) : X is an indeterminate.
1) instance(X,Name): if Name is bound then X is an instance of the type whose name is the value of Name.
2) instance(X,Y): if Y is unbound then this is eqivalent to instance(X,Y,void,true).
3) instance(X,Name,W) : if Name is bound then X is an instance of the type Name with slots W.
4) instance(X,Y,C) : if Y is unbound then this is equivalent to instance(X, Y, void, C).
5) instance(X,Y,W,C) : X is a complex indeterminate with slots W and constrained to the condition C.

3.12 Utilities

"copy" is the predicate for copying terms. The goal copy(X,Y) creates a renamed version of X and unifies with Y. The predicates might be needed for handling multi_environment handling. The goal eval(X, Y) reduces the term X to the equivalent term which has no part of the form (A!B).

3.13 Internal Primitives

The followings are not for user primitives but internal ones of CIL. They work behind the above mentioned functions.

1) pre_instance(X,N,C) : make an instance X of the type named N, and return the condition C given in the definition of the type
2) create_complex(X,S,Y) : make the indeterminate X have the slots S and unify X with the other indeterminate Y.
3) role_of(N,X,V) : the value of the slot N of X is V.
4) slots_of(X,Y) : the slots of X are all Y.
5) merge_role(X, Y) : merge the slots of X with the other slots Y
6) binding(X,Y) : Y is the current value of the indeterminate X.
7) value(X, Y) : Y is the current value of X.

4 Type, Indeterminate and Unification

The semantics of CIL can be described formally along the line which is given by Lloyd [Lloyd 1982] for instance. Both of the denotational and operational semantics are described in the section. The focii are types and indeterminates. They are new objects in logic programming language. The description should be useful for the reader to understand the behavoirs of the new objects. It will be interesting to make clear the relation between the two semantics. The task, however, is the outside of the scope of the work.

4.1 Types and Complex Indeterminates

Situation semantics uses types and indeterminates as a formal basis. Indeterminates are instances of their types and behave as logical variables with slots and constraints. Types are divided into basic types and complex types. Each complex type is represented by a pair of indeterminate and a condition. Complex indeterminates are instances of complex types. We are using the word "indeterminate" for both basic and complex ones. Each type T is supposed to have sufficiently many indeterminates T0, T1, ...

Definitions of both sentence meaning and and discourse understanding were given in the introduction. They are defined in an elegant way using types, indeterminates and conditions.

The Horn clauses in CIL are responsible for executing the conditions. Indeterminates are objects that satisfy the conditions. CIL has two primitive functions which operate on types and indeterminates. One creates indeterminates of given types. The other is the slot-filling function.

Let C be a condition with parameters X0, X1, ... , Xn:

$$C = C(X0, X1, \dots, Xn).$$

The meaning of the condition C is defined to be the set, say A, of all the anchors f such that C(f(X0), f(X1), ..., f(Xn)) holds. The meaning of the type T=[X0|C] is defined to be the set

$$\{ (X0,f) \mid f \text{ is an anchor in } A \}.$$

An element in the set represents an instance of the type T, i.e., a value of an indeterminate of the type T. So we can say that the function of an interpreter for CIL is to create instances of given types.

4.2 Data Domain and Computation Domain

A domain is defined in the section to give the denotational and operational semantics of CIL. It is necessary to extend the usual Herbrand universe to include the so called slot-filling features. this is an extension of that first order terms (Herbrand universe). So, CIL is a natural extention of the usual Horn clause logic programming.

The domain D is called the data domain. Let H be a nonempty set of ground terms which is closed under the operations of sub-term construction and term composition for some fixed set of functor symbols and atomic symbols. Let us fix the Herbrand universe H henceafter.

The data domain D is defined as the smallest extensin of H satisfying the following conditions:

D1) For any x in D and any partial function f from D to D, the pair <x, f> is in D.
D2) D is closed with both of the oprations of taking sub-term and composition.

It is necessary to extend the data domain D for including indeterminates. Lt U be the minimum extension of D satisfying the following conditions:

U1) D is a subset of U.
U2) D includes all indeterminates.
U3) For any x in U and any partial function f from U to U, the pair <x, f> is in U.
U4) U is closed with both of the operations of taking sub-term and composition.

The domain U is called the computation domain. An element of the domain is called a generalized term.

The semantics of the unifying predicate "=" is defined the minimum set E of atomic formulas of the form "="(x, y) for some x and y in the data domain D such that the following conditions hold.

E1) for any x in the Herbrand unverse H, "="(x, x) is in E.
E2) if "="(u, v) is in E and f and g are compatible then "="(<u,f>, <v,g>) is in E. Here, f and g are said to be compatible if the following C1 implies C2 for any x and y in the data domain:

C1) x and y stand in the relation E, and f(x) and f(y) are defined.
C2) f(x) and f(y) stand in the relation E.

It is easy to check that the relation "=" is a equivalence one over the the domain D.

The semantics of the slot-filling predicate "role_of" is defined clearly using the domain D and E. The denotation of the predicate is defined to be the set of all atomic formulas role_of(v, s, x) for some v, s, x in D such that x= <y, f> and f(s)=v hold for some y in D and some function f from D to D.

4.3 Denotational Semantics

Let B be the set of all the atomic formula each argument of which is in the data domain of D. B is called the base set. Let P be a set of clauses in CIL. P is a program. The semantics of the program is defined to be a minimal subset S of the base set B such that following conditions hold:

S1) S includes all the positive ground atomic formulas for any system predicate which are defined to be valid by the system. For instance, the predicate true is in S, but false is not in S.
S2) S is a super set of both the set "=" and "role_of".
S3) For any clause H<-T in the given program and any substitution s, if s(T) is satisfied by S then also s(H) is in S.

A condition is said to be satisfied by the model S if and only if it meets one of the following conditions.

V1) If X is in S then X is satisfied (by S).
V2) If X is not satisfied by S then ~ X is satisfied.
V3) If X and Y are satisfied then X & Y is satisfied.
V4) If X or Y is satisfied then X\/Y is satisfied.
V5) If C1, ..., Ci are not satisfied and C(i+1) and X(i+1) are satisfied for some i=< n, then {C1=>X1, ..., Cn=>Xn} is satisfied.
V6) If G is satisfied then X^G is satisfied.
V7) The atomic formula instance(X,Z,T,G) is satisfied if both of the equality X= <Y, L> and a formula F is satisfied where the triple (Y, S, F) is a copy of (Z, T, G) and the function L is the slots S seen as a function.

4.4 Unification

The unification in CIL is described in the section. The crucial point is the following which comes from the theory of type used by situation semantics. Let T be a type written as T = [X|Q] for a condition Q and an indeterminate X, and let T0 be an instance of T. Let f be an anchor which is defined at both T0 and X then f(T0) and f(X) should be the same.

Let A, F, and V be the set of atoms, functor symbols, and indeterminates. Let U be the computation domain over them. For any subset E of U, an eqivalence relation R over U is called a unifier of E if the following conditions hold.

U1) For any two generalized terms x and y in E, they stand in the relation R(x,y).
U2) Any two generalized terms which have distinct main functors do not stand in the relation R.
U3) For any two generalized terms of the form h(t1, ..., tn) and h(s1, ..., sn) standing in the relatin R, then also ti and si stand in

R for any 1=<i=<n.

U4) For any two indeterminates of the forms <u,f> and <v, g> which stand in the relation R, the indeterminates u and v stand in R and the functions f and g are able to be merged to each other in the following sense. That is, f(x) and g(y) stand in the relation R for any x and y which stand in the relation R and which are in both of the domains of the two functions f and g.

A unifier R of E is called finer than another unifier S of E if for any pair of terms in the computation domain U standing in the relation R, the same terms also stand in S. For any two relations R1 and R2 over U, the relation R is called the "meet" of R1 and R2 if for any x, y in U, R(x,y) is eqivalent to the conjunction of R1(x, y) and R2(x ,y).

Proposition. For any two unifiers of the set E, the meet of them also a unifier of E and is finer than any of the two unifiers.

Proof. It is easily checked that the meet of the two relations satisfies the conditions U1, U2, U3 and U4 above. It is clear from the definition of "meet" that the meet is finer than both of the given two unifiers.

Theorem. For any subset E of the computation domain U, there is at most one finest unifier of E.

Proof. Assume there are two unfiers R1 and R2 which have no finer ones except themselves respectively. From the proposition, the meet of R1 and R2 is a unifier of E which is finer than both of R1 an R2. This is a contradiction.

If slots are neglected, our unification becomes identical to ordinary first order unification. The implemetation in DEC-10 Prolog is incuded in the Appendix 3.

4.5 Operational Semantics

In CIL, a goal is solved just like a Prolog goal, i.e., top_down, depth_first, and left_to_right.

A computation in CIL is formalized as a sequence of computation states. A computation state is a pair of the form (G,A) where G and A are a goal term and an environment. An environment is a pair of an equivalence relation over the computation domain and a set of pairs of indeterminates and demons.

A transition relation "->" between the computation state is defined below. For a given goal term G and initial environment A the computaion is defined to be a maximal sequence of the computaion states (Gi, Ai) (1=<i =<n) for some 1=<n, where G0=G, A0=A, and Gn=true. There is the possibility that there are more than two computation for the given initial computation state.

The intuitive ideas of the computation model as follows. Indeterminates are variables with slots and constraints. For any two indeterminates the unifier invokes frozen goals or merges demons which are attached to them, according to whether they have been instantiated or not. Our concept of unification should be best understood from the examples given in the section 5.

Some function notations are introduced for convenience.

Let unifier(E, X, Y) denote the finest unifier F such that X and Y stand in the relation F and that E is finer than F.

It is said that the demon Y is attached to the indeterminate X in the environment <E, D> if the pair (X, Y) is in D.

Let demon(E, F, D) denote the conjunction of all the demons attached to the indeterminate in D which are unbound in E and bound in F.

Let freeze(D, X, Y) be the union of D and the singleton {(X, Y)}.

Let merge(E, X, S, V) be the unifier unifier(E, (T:U), (S:V)) where X is the indeterminate of the form <Y,f> and f(T)=U for some Y, f, T, and U.

For a generalized term X and an equivalence relation E over the computation domain, let val(X, E) is the set of all the generalized terms which are not indetrminates and stand in the relation E with X.

Let var(X, E) denote the set of all the unbound indeterminates which appear in some term in val(X, E).

Let us start the definition of the transition relation "->" as the minimum relation satisfying the following conditions:

Unify :(X=Y, <E, D>) -> (demon(E, F, D), <F, D>) where F is unifier(E, X, Y).

Filling slot : (role_of(S,X,V), <E, D>) -> (demon(E, F, D), <F, D>) where F is merge(E, X, S, V).

Indeterminate : (X:=(Y, S, C) , A)-> (X=Z & solve(C), A) where Z is the indeterminate <Y, S>.

Instantiation: (instance(Y, X, S, C),A) -> (Y=Y1 & solve(C1), A) if (X1, S1, C1) is a copy of (X, S, C) and Y1 is the pair <X1,S1> for some X1, S1, C1, and Y1.

Conjunction : (X&Y, A) -> (Y, B) if (X, A)-> (true, B).

Disjunction : (X\/Y, A) -> (X, A) and (X\/Y, A) -> (Y, A).

Freezing : (X^Y, <E,D>) -> (true, <E, freeze(D, X, Y)>) if X is unbound in E. (X^Y, <E, D>) -> (Y, <E, D>) if X is bound in E.

Negation : (\+ X, A) -> (true, A) if there are no B such that (X,A)->

(true, B).

Delayed negation : (~ X, <E, D>) ->(suspend(var(X, E), (\+ X)), <E, D>).

Suspending : (suspend(V, G), <E, D>) -> (solve(G), <E, D>) if var(V,E) is empty. (suspend(V,G),<E,D>)->(X^suspend(W,G),<E,D>) if for some indeterminate X and set W, var(V, E) is the union {X} and W and any indeterminate which is equal to X is not in W.me

Conditional {} : ({C1=>X1, ..., Cn=>Xn}, A) -> (Xi, B), where i is the first index such that (Ci, A) -> (true, B).

Meta calling : (solve(X), A) -> (X, A).

Meta calling : (X, <E, D>) -> (G, <E, D>) if the variable X is bound to G in E.

Other instances: similar to the above "instance".

Bound : (bound(X), <E,D>) -> (true, <E, D>) if X is bound in E.

Unbound : (unbound(X), <E, D>) -> (true, <E, D>) if X is not bound in E.

Indeterminate: (indeterminate(X), A) -> (true, A) if X is an indeterminate.

Atom, integer, atomic : similar to the above.

Identical : (same(X,Y), <E, D>) -> (true, <E, D>) if the values of X and Y are the same.

User predicate: (G, A) -> (G=H1 & T1, A) if there is some clause of the form H<-T and H1<-T1 is a copy of the clause.

This is the end of the definition of "->"

5 Programming

Here are some examples of CIL programs. The symbols '>' and '>>' are system and user prompts for inputs respectively.

5.1 Unification and Accessing Slots.

The first example show that the introduction of the logical variable with slots are natural and useful for inferential processes. Notice that the indeterminates Boy and Child in the example are unkown except that they are identical to each other. It is difficult to represent the eqivalent inference in the usual Horn clause logic.

```
> jack= father ! Boy & betty= mother ! Child &
> Boy = Child & Who = father ! Child & print(Who).
jack
success
```

```
        > 1= a!X & 2= a!Y  & X=Y.
        fail
        > 1= a!X & 2= b!Y & X= Y & same(X, Y).
        success
```

It is possible to put the slot name unknown:

```
        > jack= What ! B & Y= father! Z &
          B=betty & B=Z & print((Y,What,Z)).
        (jack,father,betty)
        success
        >
```

Let X and A be a complex indeterminate and a constant. The effect the goal X=A is that the vlaue become the constant A. For any complex indeterminate Y, the effect of the goal X=Y is that X and Y are unified with each other including their slots.

For two indeterminate X and Y, same(X, Y) means only that the current "values" of the two indeterminates are identical to each other. In other words, they may have incompatible sets of slots even when they are anchored to the same value, also the different two classes of indeterminates may have been bound to the same value. Such a case is shown by the following two examples.

```
        > 1= a!X & 2= a!Y & X= Y.
        fail
        > 1= a!X & 2=a!Y &  X=3 & Y=3& same(X, Y).
        success
        > X=a!Y & Y= b!Z &
        > 1=a!A & A= b!B & Z=B & print(X).
        1
        success
```

5.2 Delayed Negation

CIL has two type of negations as failure, i.e., eager_negation (\+) and delayed-negation (~). The latter uses the form of suspend(X,Y), which suspends the goal Y until the term X become to has no unbound indeterminates.

The "diff" statement in Prolog II is defined by using this.

```
        > \+ (X=1).
        fail
        > ~(X = 1).
        success
        >  ~ (X = 1) & X= 2.
        success
        >  ~(X = 1) & X= 1.
        fail
        > ~(f(X,1)=f(2,Y)) & X=2 & Y=1.
        fail
```

5.3 Instantiation of Indeterminates

The predicate "instance" has the following four forms of application for convenience:

```
        instance(X),
        instance(X,Y),
        instance(X,Y,Z), and
        instance(X,Y,Z,U).
```

The uses of each of them are explained in the section 3.

```
        > instance(X,Y,Y=1)&print(X).
        1
        success
```

Two instances of the same type are different to each other initially.

```
        > instance(X,Y,true) & instance(Z,Y,true) &same(X,Y).
        fail
```

The "if_filled_demons" are easily attached to any slot as follows.

```
        > instance(X,Y,a:Z, Z^ print(Z))& 1= a!X.
        1
        success
```

5.4 Freezing Goals and Coroutine Programming

The primitive form of bind hooking is "X^Y". The goal Y is suspended while the variable X is unbound.

```
        >  X^(print(aaa)&nl) & Y^(print(bbb)&nl)&X=Y&Y=1.
        aaa
        bbb

        success
```

The following is a program well known under the name "Eratosthenes' sieve", which generates the prime number sequence 2,3,5,7,11,13,... This example uses the "freeze" statement. "print", "nl", "=:=", "is" have the same functions as the evaluable predicates with same names

```
        primes<- X^primes(X)&integers(2,X).

        primes([P|X])<- print(P)& nl & X^sieve(P,X,Y) & Y^primes(Y).

        sieve(P, [N|R], S)<-
                { (N mod P =:= 0) => R^sieve(P,R,S),
                  S=[N|T]& R^sieve(P,R,T)              }.

        integers(N,[N|R])<- M is N+1 & integers(M,R).
```

The execution is as follows:

```
> primes.
2
3
5
7
.
.
.
```

5.5 Demons Watching Contexts

It is easy to realize both of the conjunctive and disjunctive demons. The predicates "find" and "disj_find" realize demons. The definition of them are in the appendix 2. They are defined by using "freeze" primitives. A demon is a sequence of goals that can be enabled during the execution of some other goals and that watches a given context to detect patterns for which it is responsible. The demon can be frozen and set to resume its control later.

The goal find(X,D,C) means that X is the first element in C such that the condition D(X) holds, where C is a stream of some contextual information. The last element of the stream should be "end".

Let Gi (i=1,...,n) be demons. Each Gi is assumed to have the form (X, D, C) for some indeterminate X, condition D, and stream C. Suppose we want satisfies one of the demons and it is not necessary to satisfy the other demons. This problem is written as disj_find((G1;G2;...;Gn)).

Program execution looks like this;

```
> find(X, X=1&print(ok)&nl, C)& instream(C).
>> 3. 2. 1. end.
ok

success
> disj_find(((X,X=1 & print(aaa)&nl, C);
>             (Y,Y=2 & print(bbb)&nl, C))     ) & instream(C).
>> 3.
>> 2.
bbb
>> 1.
>> end.

success
> disj_find(((X,X=1 & print(aaa)&nl, C);
>             (Y,Y=2 & print(bbb)&nl, C))     ) & instream(C).
>> 3.  end.
fail
```

6. Implementation

In this section, we describe the underlying CIL interpreter implemented on top of DEC-10 Prolog. We call the interpreter REDUCE. The main functions of REDUCE are:

1) to effect "freezing" and "resuming" of procedures, and
2) to define some primitives for complex indeterminates.

Implementation of "freezing" and "resuming" processes is built around the data structure of indeterminates. The REDUCE program is listed in Appendix 3.

6.1 Structure of Indeterminates

Each indeterminate is represented as a term whose main functor is a certain designated symbol 'I'. Slots and frozen goals in the indeterminate are represented as a difference list for easy merging. Each indeterminate contains some "pointers" to others. The set of all indeterminates involved in any computation forms a so called forest under these pointers, which represent the equivalence relation over the set.

6.2 Unifier

Let V be the set of all the indeterminates. Let T be the set of all the non-variable terms. The data structure of the unfier is a forest over V. Each component of the forest is a tree and represents an eqivalence class of V. Each of the classes is supposed to have the representative node, which is the root node of the class seen as a tree. Each indeterminate contains slots (possibly null), frozen goals (possibly null), and the value to which it is anchored. The value is a non-variable term if it is defined. These slots and goals are merged into the representative indeterminate. The term is equal to the one which is assigned to the representative node.

6.3 Freezing and Unification

When any representative indeterminate with frozen goals is unified with a non-variable term, the goals resume their controls. Each indeterminate has a flag for the frozen goals so that they are not executed more than once.

The details of these mechanisms are described under the two predicates "freeze" and "unify" in Appendix 3.

Now, how do we freeze the specified goals ? The form of the "freeze" statement is X^Y, where, generally, the goals Y contain the variable X. For instance, freeze(X,p(X)) is reduced to X = 'I'(_, ..., _, p(X)) in DEC-10 Prolog unification. In other words, we use infinite trees to associate the frozen goals with the specified indeterminate.

Our unifier collects all the goals hooked on to such a variable V appearing in X or Y that V is instantiated through this unification process. For instance, suppose goals p and q are hooked on to variables A and B repectively. Then the goal p&q is returned through the unification f(A,1) with f(1,B). Also, the result of the unification f(A,A) with f(1,1) will be p.

7. Concluding Remarks

We described the simple but powerful language CIL for situation semantics oriented programming. The key idea was to build parametarized types into the language based on unification. Complex indeterminates are implemented as logical variables with named slots and constraints. Also we discussed the generality of CIL's power using the discourse understanding model. We have shown some examples involving contexts to demonstrate the usefulness of CIL for describing discourse models.

The old version of CIL was used in the pragmatics processing part of the discourse understanding system called DUALS [DUALS 1985] on top of DEC-10 Prolog. Situation semantics forms an essential part of the conceptual aparatus of DUALS also.

The target machine on which we plan to implement CIL efficiently is the PSI machine [Uchida & Yokoi 1984], which is now available to us. The PSI has machine primitives for bind hook control and is flexible enough so that we can build parameterized types without loss of efficiency.

Suzuki [Suzuki 1984] suggested a relation between types and frames. CIL seems to be a programming language capable of realizing this relation.

ACKNOWLEDGEMENTS

Dr. F. Pereira of SRI Internatinal & CSLI gave many valuable comments to the work during the invited stay here at ICOT. Also he improved the memory efficiency of the interpreter for the proposed language which has been implemented in DEC-10 Prolog. I would like to thank my colleagues H. Miyoshi, H. Yasukawa, H. Hirakawa, and Y. Tanaka for many stimulating discussions on computational models of discourse. Dr. B. Smith of Xerox PARC & CSLI suggested a situation semantics programming language to us during his short visit here. Also, I would like to thank T. Yokoi, the Head of the Third Laboratory for his useful comments and encouragement.

REFERENCE

[Barwise & Perry 1983] J. Barwise & J. Perry : Situations and Attitudes, MIT Press, 1983.

[Barwise 1975] J. Barwise: Admissible Sets and Structures, Springer Verlag, 1975.

[Barwise 1984] J. Barwise: Lectures on Situation Semantics, Winter Quarter at CSLI, 1984.

[Bowen 1981] D. Bowen: DECsystem-10 Prolog User's Manual, Department of Artificial Intelligence, University of Edingburgh, 1981.

[Brady & Berwick 1982] M. Brady & R. Berwick (eds.): Computaional Models of Discourse, MIT Press, 1982.

[Colmerauer 1982] A. Colmerauer: Prolog II: Reference Manual and Theoretical Model, Internal Report, Groupe Intelligence Artificielle, Universite Aix-Marseille II, 1982.

[ICOT 1985]: (in preparation as an ICOT technical report)

[Furukawa & Yokoi 1984] K. Furukawa & T. Yokoi: Basic Software System, in the Proceedings of the International Conference on Fifth Generation Computer Systems 1984, edited by ICOT, pp47-48, 1984.

[Lloyd 1984] J. W. Lloyd: Foundations of Logic Programming, Technical Report 82/7 (revised March 1984), Department of Computer Science, University of Melbourne.

[Suzuki 1984] H. Suzuki: MAID: a Man-machine Interface for Domestic Affairs, Institute for New Generation Computer Technology, ICOT TM-0058, 1984.

[Uchida & Yokoi 1984] S. Uchida & T. Yokoi: Sequntial Inference Machine: SIM -Progress Report-, in the Proceedings of the International Conference on Fifth Generation Computer Systems 1984, edited by ICOT, pp58-69, 1984.

APPENDIX 1. Summary of Situation Semantics

Our formal basis of situation semantics are in [Barwise 1984]. We list axioms and definitions given there for conveniencies.

-- Set theoretic assumptions are those of KPU [Barwise 1975].

-- Axiom of real situations (1):
Every real situation s determines a set of facts, the set fact(s) of facts f that obtain about in internal structure of s.

-- Axiom of real situations (2):
Every real situation s is completely determined by the set fact(s). That is, if fact(s)=fact(s') then s=s'.

-- Axiom of real facts:
There are located and unlocated facts. A located fact f consists of a sequence x1,...,xn (n>=0) of objects standing, or not standing, in a spatio-temporal relation r at a space time location l. An unlocated fact f consists of a sequence x1,...,xn (n>=0) of objects standing, or not standing, in a certain non-spatio-temporal relation r.

-- Notation of "real fact":
We sometimes describe a located fact f by writing

```
        at l: r,x1,...,xn; 1

        or

        at l: r,x1,...,xn; 0
```

depending on whether or not the fact f is that the objects do or do not stand in the given relation at l. Similarly, we write

```
        r,x1,...,xn; 1

        or

        r,x1,...,xn; 0
```

for unlocated facts. The number 1 or 0 is called the polarity of the fact f and is devoted by pol(f). The other items are called constituents of the fact.

-- Definition of "circumstance":
We define an n-ary (abstract located) circumstance to be a sequence f of the form <l, r, x1, ..., xn, pol> where l is in L, r is an n-place spatio-temporal relation, x1,...,xn are arbitrary objects and pol is 0 or 1. An n-ary (abstract unlocated) circumstance f consists of a sequence <r, x1, ..., xn, pol> ,where r is an n-ary non-spatio-temporal relation. Axiom 3 insures that each fact f can correspond to a unique circumstance f'= abs(f). If a circumstance f' corresponds to a (located or unlocated) fact, then f' is called an abstract (located or unlocated) fact.

-- Definition of "abstract situation":
We define an abstract situation s' to be any set of abstract facts. By axiom 1, we can assign to each real situation s an abstract situation s'= abs(s) by: s'= {abs(f)| f is in fact(s) }. We say that an abstract situation s' corresponds to a real situation s if s'= abs(s); we say that s' correctly classifies s if s' is a subset of abs(s). We say that an abstract situation s' is factual if each f in s' is an abstract fact. On the other hand, we say that s' is actual if s' corresponds to some real situation s.

-- Axiom of factual situation:
Every factual abstract situation s0 is a subset of some actual abstract situation s1.

-- Definition of "coherent":
We say that an abstract situation s is coherent provided:
1) no two circumstances in s differ only in their polarity;
2) there is no object x such that the circumstance < =, x, x, 0> is in s;
3) there is no distinct objects x and y such that the circumstance < =, x, y, 1> (Here we are using "=" to denote the identity relation.)

-- Axiom of actual situation:
Every actual situation is coherent.

-- Axiom of objects:
Every object is the constituent of some fact.

-- Axiom of types:
Among the types of objects are the following basic types of objects:
IND, the type of individual object
RSIT, the type of real situation
SET, the type of set
SIT, the type of (abstract) situation [actually redundant]
LOC, the type of space-time location
POL, the type of the "truth values" 0 and 1
RELn, the type of n-ary relations

-- Axiom of indeterminates:
For each of basic types T, there are indeterminates T1, T2, T3,... which may be "anchored" to things of type T. Distinct types have distinct indeterminates.

-- Definition of anchors:
An anchor is a function h with domain some set of basic indeterminates such that if x, say Tn, is in the domain of h, then h(x) is of type T.

-- Equivalence of Anchors:
Two anchors are equivalent if there is a renumbering of the indeterminates that takes one anchor to the other. We are really interested in anchors only modulo to this equivalence relation.

-- Main assumptions on conditions:
1). Each condition C has some non-empty finite set para(C) of

parameters, a set of basic indeterminates.

2). A condition C is a condition on anchors h with para(C) a subset of the domain of h.

3). A condition C either holds or does not hold of any anchor it is a condition on.

4). Given any finite set X of conditions, there is a condition that C with para(C) the union of all the parameters of X, and such that C holds of h just in case each condition in X holds of h.

5). Conditions are equivalent if there is a renaming of their parameters that takes one to the other; that is, if they put conditions on equivalent anchors.

6). For each condition C and each parameter X of C there is a complex indeterminate X¦C. The parameters of X¦C are those of C, except for X. We impose the following constraint on anchors: if h(X¦C) is defined, then h(X) is defined and equal to h(X¦C), and h meets condition C.

7). Each complex indeterminate X determines a type T(X). Equivalent indterminates determine the same type.

APPENDIX 2: Program Examples

The useful primitives and utilities are defined in the language as below. The fist group of examples is for instantiation of indeterminates. The second one is for demon programming. The third one is for printing complex indeterminates. The final one is an illustrative application of the "type" of "story structure" defined by the language.

(1) Instantiation of Complex Indeterminates

It is fundamental to instantiate objects as indeterminates. The following are several versions of instantiation which are primitives for the user.

```
instance(X)<<- indeterminate(X).

instance(Y, X)<<-
        {bound(X)=> type_of(Y, X, void),
         instance(Y,X,true)             }.

instance(Y, X, Z)<<-
        {bound(X)=> type_of(Y, X, Z),
         copy((X,Z),(X1,Z1)) &
         create_complex(X1,void, Y1) & Y=Y1 & solve(Z1)      }.

instance(Y,X,S,Cond)<<-
        copy((X,S,Cond),(X1,S1,Cond1)) &
        create_complex(X1,S1,Y1) & Y=Y1 & solve(Cond1).

type_of(X,N)<<-pre_instance(X,N,Cond)&solve(Cond).
type_of(X,N,void)<<-type_of(X,N).
type_of(X,N,W)<<-pre_instance(X,N,Cond)&merge_role(X,W)&solve(Cond).

X:=R <<-
        {
         value(R, V)& functor(V, F, 1)=>
                   indeterminate(Z) &
                   arg(1, V, Z) &
                   role_of(F, Z, X),
         R= (Y,S,C)=> create_complex(Y, S, X1)&  X=X1 &solve(C),
         R= (Y,C) => X:= (Y, void, C)
         }.

carg(N,X,Y)<<-value(N,N1)&value(X,X1)&arg(N1,X1,A)&A=Y.

cfunctor(X,F,N)<<-value(X,X1)&functor(X1,F1,N1)&F=F1&N=N1.

mfunctor(X,F,N)<<-value(F,F1)&value(N,N1)&functor(X1,F1,N1)&X=X1.

eval(X,Y)<<-
        {atomic(X)=> Y=X,
         unbound(X)=>Y=X,
         cfunctor(X,F,N)&mfunctor(Y,F,N)&eval(N,X,Y)}.

eval(N,X,Y)<<-
```

```
        {N==0,
         carg(N,X,A)&carg(N,Y,B)&
         eval(A,B) &
         M is N-1 &
         eval(M,X,Y)}.

same(X, Y)<<-
        {unbound(X)=> unbound(Y)&X==Y,
         unbound(Y)=> unbound(X)&X==Y,
         cfunctor(X,F,N)&cfunctor(Y,F,N)&same(N,X,Y)}.

same(N,X,Y)<<-
        {N==0,
         carg(N,X,A)&carg(N,Y,B)&
         same(A,B) &
         M is N-1 &
         same(M, X, Y)}.

copy(X,Y)<<-copy(X,_,Y).

copy(X,M,Y)<<-
        {unbound(X) => map(X,M,Y),
         cfunctor(X,F,N)&mfunctor(Y,F,N)&copy(N, X, M, Y)}.

copy(N,X,M,Y)<<-
        {N==0,
         carg(N,X,A)&
         carg(N,Y,B)&
         copy(A,M,B)&
         J is N-1 &
         copy(J,X,M,Y)}.

map(X,Y,Z)<<-
        {unbound(Y)=> Y= (X-Z)^_ ,
         Y=(U-Z)^V=> {same(X,U), map(X,V,Z)}}.

not(X)<<-{solve(X)=>fail, true}.

undefined(X)<<-unbound(X).
defined(X)<<-bound(X).
```

(2) Demons Watching Streams

The following "when_all" and "when_some" are another variations of the freezing ("^").

```
when_all(X, D)<<-
        {X=[]=> solve(D),
         true=> X=[Y|Z]& Y^when_all(Z, D)       }.

when_some(X,D)<<-hook_or(X,G)& G^D.

hook_or(X,G)<<-
        {X=[],
         X=[Y|Z] => Y^(G=1) & hook_or(Z, G)       }.
```

```
find(X,D,C)<<-
        C^
         {C=A*R=> {X=A & solve(D),
                   find(X,D,R)     },
          true => X=C & solve(D)
             }    .

disj_find(Ds)<<- disj_find(Ds, F).

disj_find(Ds,F)<<-
        {Ds= (H;T) => disj_find(H,F) & disj_find(T, F),
         Ds=(V,D,C) => disj_find(V,D,C,F)              }.

disj_find(V,D,C,F)<<-
        (C^set(Switch,c)) & (F^ set(Switch,f)) &
        Switch ^
             {Switch==f ,
              C=A*R=>
                {V=A & solve(D) & F=1 ,
                 disj_find(V, D, R, F)
                }                ,
              true => V=C & solve(D) & F=1
             }.

set(O,X)<<-
        {bound(O), O=X }.

instream(X)<<- prompt(P, '>> ')&read(Y) &
        { Y==end => X=Y,
          X= (Y*T) & instream(T) } &
        prompt(_, P).

produce(X,Y,Z)<<-
        {unbound(Y)=>Y=X*Z,
         Y=_*U & produce(X,U,Z)
        }.
outstream(X,Y)<<-
        {X=(A*B) => Y= A* Z & outstream(B, Z),
         Y= X
         }.

in(S, X)<<-
        S^
        { S= (X *_ ),
          S= (_ * T) => in(T, X),
          S= X
        }.

has(S, X)<<-
        {
        S= (X*_),
        S= (_*T) => has(T, X),
        S= X
        }.
```

(3) Printing Objects

```
print(X)<<- call(print(X)).

vprint(X)<<- eval(X, V)&print(V).

print_context(X)<<-
        {unbound(X),
        X=(Y*Z)=> print_obj(Y)&print_context(Z),
        print_obj(X)}.

print_obj(O)<<-
        nl&print(O)&print(('::'))&nl&
        {indeterminate(O)=>
                slots_of(O,S)&print_slots(S),
        true}.

print_slots(S)<<-
        {unbound(S),
         S=(H,T) => print_slots(H)&print_slots(T),
         print('  ')&print_slot(S)
         }.

print_slot(K:B)<<-
         {K==context => print('context: ** omitted **'),
          print(K:B) &nl }.

p(X)<<-print(X)&ttyflush.
```

(4) Story Summary Based on Plan_Goal Models

The following is a simplified version of summarizing story program. The program uses several demons who are responsible to detect some their own patterns watching the story context as a stream.

```
dcl(plan_goal, _,
        (
        intention: Intention   ,
        person: Person,
        obstruction: T,
        expected_situation: E,
        planned_situation: PS,
        action: A,
        context: C),
        (
        intend(Person, Intention, C)&
        obstruct(T, Intention, C)&
        expect(Person, E, C)&
        want(Person, PS, C)&
        action(Person, A, C) ) ).

 intend(Person, duty!Person, Context)<- true.

 obstruct(T, I, Context)<-
```

```
        in(Context, (_, (find, Person, T), 1)) &
        in(Context, (_, (surprise, Person, T), 1)).

expect(Person, E, Context)<-
        in(Context, (_, (involve, T, E), 1)).

want(Person,Want, Context)<-
        in(Context, (_,(command,Person,_,Want),1)).

action(Person, Action, Context)<-
        find(Action,
             same(action, type!Action) &
             same(Person, agent!Action),
             Context).

story_summary <-
        instance(S, plan_goal,
         (person:Roll_san,context: Context) ) &

        instance(Roll_san) &
        instance(Fuchigami_san) &
        instance(Captain)&
        instance(Smoke) &
        instance(Fire) &
        instance(Explode ) &
        instance(ToPrepare ) &
        instance(ToInform) &
        instance(Command) &
        instance(Emergency) &
        instance(Gush) &
        instance(Flight) &
        instance(Service) &

        Flight = duty!Captain &
        Sevice = duty!Stewardess &
        Captain = agent!Command &
        action = type!Command &

        S = 'STORY' &
        Roll_san = 'ROLL_SAN' &
        Fuchigami_san = 'FUCHIGAMI_SAN' &
        Command = 'COMMAND' &
        Smoke  =  'SMOKE' &
        Fire  =    'FIRE' &
        Explode  =  'EXPLODE' &
        ToPrepare =     'TO_PREPARE' &
        ToInform=       'TO_INFORM'&
        Emergency=      'EMERGENCY' &
        Gush=           'GUSH' &
        Flight=         'FLIGHT' &

Captain =       Roll_san &
Stewardess =    Fuchigami_san &

outstream(
( Roll_san   *    (_,(gush,Smoke),1) *
  Fuchigami_san  * (_,(find, Roll_san, Gush), 1) *
  Smoke  *    (_,(surprise, Roll_san, Gush), 1) *
  Fire  *     (_,(involve, Fire, Explode),1) *
  Explode  *  (_,(call, Roll_san, Fuchigami_san),1) *
  ToPrepare  *(_,(command, Roll_san, Fuchigami_san, ToPrepare),1) *
  ToInform *  (_,(ask_if, Fuchigami_san, Roll_san, ToInform),1) *
  Emergency * (_,(prepare, Fuchigami_san, Emergency),1)    *
  Gush *
  Flight *
  Command *
  Captain *
  Stewardess ), Context) &

  print_obj(S )&nl.
```

The execution of the goal looks like this:

```
> story_summary.

STORY::
  intention:FLIGHT
  person:ROLL_SAN
  obstruction:GUSH
  expected_situation:EXPLODE
  planned_situation:TO_PREPARE
  action:COMMAND
  context: ** omitted **

success
>
```

APPENDIX 3: Interpreter on DEC-10 Prolog

(1) Operators Declarations
(2) Reducing Goals
(3) Instantiation and Merging
(4) Interpretation of CIL Primitives and Utilities
(5) Calling and Tracing User Rules
(6) Demon
(7) Complex Indeterminates
(8) Unification and Binding
(9) Testing Equality
(10) Printing
(11) CIL Main and Debugging

```
:-public reduce/1.
:-public c/0,cil/0.
```

(1) Operators Declarations

```
:-op(1100, xfy, (<-)).
:-op(1100, xfy, (<<-)).
:-op(960, xfx, (:)).
:-op(930, xfy, (=>)).
:-op(920, xfy, (\/)).
:-op(910, xfy, (&)).
:-op(900, fy, (~)).
:-op(700, xfx, (:=)).
:-op(400, xfy, *).
:-op(150, xfy, (!)).
```

(2) Reducing Goals

```
:-fastcode.
:-mode reduce(+).

reduce(true):-!.
reduce(Goals):-
        reduce_first(Goals, Rest),
        reduce(Rest).

:-mode reduce_first(+,-).
reduce_first(true&P, Q):-!, reduce_first(P,Q).
reduce_first((P&Q)&R, S):-!,reduce_first(P&(Q&R),S).
reduce_first(P&Q, R&Q):-!,
        reduce_first(P, R).
reduce_first(P\/Q, R):-!,
        (reduce_first(P,R);
        reduce_first(Q,R) ).
reduce_first({P=>Q}, Q):-reduce(P),!.
reduce_first({P=>Q,R}, Q):-reduce(P),!.
reduce_first({P,R},true):-reduce(P),!.
reduce_first({_,R},{R}):-!.
reduce_first({P},P):-!.
reduce_first(X=Y,G):-!,unify(X,Y,G-true).
reduce_first(X^Y,G):-!,freeze(X,Y,G).
reduce_first((\+ X), fail):-reduce(X),!,fail.
reduce_first((\+ X), true):-!.
reduce_first(~(X), G):-!,suspend(X,[], (\+ X), G).
reduce_first(suspend(X,Y,Z),G):-!,suspend(X,Y,Z,G).
reduce_first(solve(X),G):-!,reduce_first(X,G).
reduce_first(X,G):-
        binding(X,G),!.
```

(3) Instantiation and Merging

```
reduce_first(pre_instance(X,N,Cond),G):-!,
        value(N,N1),
        instantiate(N1, X1, Cond1),
        unify(Cond,Cond1,G-H),
        unify(X,X1,H-true).
reduce_first(create_complex(X,C,Y),true):-!, create_complex(X,C,Y).
reduce_first(role_of(N,X,V),E):-!,
        representative(X,RX),
        slots_of(RX,Slots),
        merge_slot(Slots, N:V,E-true).
reduce_first(slots_of(X,Slots),E):-!,
        representative(X,RX),
        slots_of(RX,S),
        unify(Slots,S,E-true).
reduce_first(merge_role(X, Roles),E):- !,
        representative(X, Y),
        slots_of(Y,Z),
        merge_slots_to_left(Z,Roles,E-true).
```

(4) Interpretation of CIL Primitives and Utilities

```
reduce_first(binding(X,Y),true):-!,binding(X,Y).
reduce_first(value(X,Y),true):-!,value(X,Y).
reduce_first(unbound(X),Y):-!,unbound(X,Y).
reduce_first(bound(X),Y):-!,bound(X,Y).
reduce_first(indeterminate(X),true):-indeterminate(X),!.
reduce_first(indeterminate(X),true):-!, create_complex(_,void,X).
reduce_first(atom(X),Y):-!,atom(X,Y).
reduce_first(atomic(X),Y):-!,atomic(X,Y).
reduce_first(integer(X),Y):-!,integer(X,Y).
reduce_first(X==Y,true):-!,value(X,U),value(Y,V),U==V.
```

(5) Calling and Tracing User Rules

```
reduce_first(X, G):-
        functor(X,F,N),
        functor(Y,F,N),
        (defined(('<<-'), Y),!,(Y<<-Z)
        ;
        defined(('<-'), Y),!,(Y<-Z)        ),
        unify(X,Y,G-U),
        modify_if_debug(X,Z,U).

reduce_first(X,true):-call(X), !. % Note that "call" has choice points.
```

(6) Demon

```
freeze(X,Y,G):-var(X),!,create_indeterminate(X),freeze_rep(X, Y, G).
freeze(X,Y,G):-indeterminate(X),!,
        representative(X, R),
        freeze_rep(R,Y,G).
freeze(!(X,Y),Z, S&(U^Z)):-!,expand_sugar(!(X,Y),U,S).
freeze(X,Y,Y).

:- mode freeze_rep(+,+,?).
freeze_rep(X,Y,Y):-anchor_of(X,A),nonvar(A),!.
freeze_rep(X,Y,G):-demon_of(X,D),tail(D,(Y&_),G-true).
```

```
:-mode suspend(+,+,?,-).
suspend(X, Y, Z, G):-var(X),!,freeze(X,suspend(X,Y,Z),G).
suspend(X, Y, Z, G):-indeterminate(X),!,suspend_1(X,Y,Z,G).
suspend(X, Y, Z, G):-atomic(X),!,suspend(Y,Z,G).
suspend(!(X,Y),Z,U,S&suspend(V,Z,U)):-!,expand_sugar(!(X,Y),V,S).
suspend(X,Y,Z,G):-functor(X, F, N),suspend([(N,X)|Y],Z,G).

:- mode suspend_1(+,+,?,-).
suspend_1(X,Y,Z,G):-up_of(X,U),nonvar(U),!,suspend_1(U,Y,Z,G).
suspend_1(X,Y,Z,G):-anchor_of(X,A),var(A),!,freeze(X,suspend(X,Y,Z),G).
suspend_1(X,Y,Z,G):-anchor_of(X,A),suspend(A,Y,Z,G).

:-mode suspend(+,?,-).
suspend([], Z, Z):-!.
suspend([(0,_)|R], Z, G):-!,suspend(R, Z, G).
suspend([(I,X)|R], Z, G):- arg(I, X, A), J is I-1,
        suspend(A, [(J,X)|R], Z, G).

:-mode defrost(+,?).
defrost(X,H-H):-flag_of(X,F),nonvar(F),!.
defrost(X,(E&H)-H):-
        flag_of(X,1),
        demon_of(X, D),
        put_end(D, E).

:-mode put_end(?,-).
put_end(X, true):-var(X),!.
put_end(X, D):-indeterminate(X),!,binding(X,B),put_end(B,D).
put_end(X&Y,X&Z):-!,put_end(Y,Z).
put_end(X,X).
```

(7) Complex Indeterminates and Test Predicates

```
% The complex indeterminates are represented by terms of the form of
%
%     'I'(Class,Up,Anchor,Slots,Flag,Demon)

:-mode representative(+,-).
representative(X,X):-up_of(X,U),var(U),!.
representative(X,Y):-up_of(X,U),representative(U,Y).

:- mode value(?,-).
value(F,V):-indeterminate(F),!,binding(F,V).
value(F,F).

:- mode binding(?,-).
binding(X, Y):- indeterminate(X),!,
        representative(X,R),
        anchor_of(R, Y).

:-mode bound(?,-).
bound(X,true):-var(X),!,fail.
bound(X!Y,Z&bound(U)):-!,expand_sugar(X!Y,U,Z).
bound(X,true):-value(X,V),nonvar(V).
```

```
:-mode unbound(?,-).
unbound(X, true):-var(X),!.
unbound(X!Y,Z&unbound(U)):-!,expand_sugar(X!Y,U,Z).
unbound(X, true):-value(X,V), var(V),!.

unbound(X):-indeterminate(X),!,binding(X,Y),var(Y),!.
unbound(X):-var(X).

:-mode atomic(?,-).
atomic(X, _):-var(X),!,fail.
atomic(X!Y,Z&atomic(U)):-!,expand_sugar(X!Y,U,Z).
atomic(X,true):-value(X,V),atomic(V).

:-mode atom(?,-).
atom(X, _):-var(X),!,fail.
atom(X!Y,Z&atom(U)):-!,expand_sugar(X!Y,U,Z).
atom(X,true):-value(X,V),atom(V).

:-mode integer(?,-).
integer(X, _):-var(X),!,fail.
integer(X!Y,Z&integer(U)):-!,expand_sugar(X!Y,U,Z).
integer(X,true):-value(X,V),integer(V).

instantiated(X):-unbound(X),!,fail.
instantiated(X).

indeterminate(X):-nonvar(X),functor(X,'I',_),!.
:- mode class_of(+,-).
class_of(X,Y):- arg(1,X,Y).

:- mode up_of(+,-).
up_of(X,Y):- arg(2,X,Y).

:- mode anchor_of(+,-).
anchor_of(X,Y):-arg(3,X,Y).

:- mode slots-of(+,-).
slots_of(X,Y):-arg(4,X,Y).

:- mode flag_of(+,-).
flag_of(X,Y):-arg(5,X,Y).

:- mode demon_of(+,-).
demon_of(X,Y):-arg(6,X,Y).


:- mode create_indeterminate(-).
create_indeterminate(X):-functor(X,'I',6).

:- mode create_complex(-,+,-).
create_complex(Ind, Slots, X):-!,
        create_indeterminate(X),
        (Slots==void,!,S=_
        ;
        S=(Slots,_)),
        slots_of(X, S),
        create_indeterminate(Ind),
        class_of(Ind,C), class_of(X, C),
        flag_of(Ind,C), flag_of(X,C),
        up_of(Ind,X).

:- mode instantiate(+, -, -).
instantiate(N, X, E&Cond):-
        dcl(N, Ind, Slots, Cond),
        expand_sugar(Ind, R, E),
        create_complex(R, Slots, X).
```

(8) Unification and Binding

```
:- mode expand_sugar(?,-,-).
expand_sugar(X,X,true):-var(X),!,create_indeterminate(X).
expand_sugar(X,R,true):-indeterminate(X),!,representative(X,R).
expand_sugar(!(X,Y),Z,E&role_of(X,U,Z) ):-!,
        create_indeterminate(Z),
        expand_sugar(Y,U,E).
expand_sugar(X, Y, X=Y):- create_indeterminate(Y).

unify(X,Y):-unify(X,Y,_).

unify(X,Y,E):-var(X),!,unify_var(X,Y,E).
unify(X,Y,E):-var(Y),!,unify_var(Y,X,E).
unify(!(X,Y),Z,(P&U=Z&H)-H):-!,expand_sugar(!(X,Y),U,P).
unify(Z, !(X,Y), E):-!,unify(!(X,Y),Z,E).
unify(X,Y,E):-indeterminate(X),!,unify_ind(X,Y,E).
unify(X,Y,E):-indeterminate(Y),!,unify_ind(Y,X,E).
unify(X,Y,W):-
        functor(X,F,N),
        functor(Y,F,N),
        unify_term(0,N,X,Y,W).

:- mode unify_term(+,+,+,+,?).
unify_term(N,N,_,_,P-P):-!.
unify_term(J,M,X,Y,P-Q):-
        N is J+1,
        arg(N,X,A),
        arg(N,Y,B),
        unify(A,B,P-R),
        unify_term(N,M,X,Y,R-Q).

:- mode unify_var(-,?,?).
unify_var(X,Y,P-P):-var(Y),!,X=Y.
unify_var(X,X,P-P):-atomic(X),!.
unify_var(R, X, P-P):-indeterminate(X),!,representative(X,R).
unify_var(X,!(Y,Z),(P&X=U&H)-H):-!,expand_sugar(!(Y,Z),U,P).
unify_var(X,Y,P-P):-create_indeterminate(X),anchor_of(X,Y).

:- mode unify_ind(+,?,?).
unify_ind(X,Y,P-P):-indeterminate(Y),class_of(X,C),class_of(Y,D),C==D,!.
unify_ind(X,Y,E):-indeterminate(Y),!,
        representative(X,R),
        representative(Y,S),
        merge_class(R,S,E).
unify_ind(X,Y,E):-representative(X,R), bind_class(R,Y,E).

:- mode bind_class(+,?,?).
bind_class(X,Y,E):- anchor_of(X,A),var(A),!,
        A=Y,
        defrost(X,E).
bind_class(X,Y,E):-
        anchor_of(X,A),
        unify(A,Y,E).

:-mode merge_class(+,+,?).
merge_class(X,Y,P-Q):-
        anchor_of(X,A),var(A),
        anchor_of(Y,B),var(B),!,
        demon_to_left(X,Y,P-R),
        merge_to_left(X,Y,R-Q).
merge_class(X,Y,P-Q):-
        anchor_of(X,A),var(A),
        anchor_of(Y,B),!,
        defrost(X,P-R),
        merge_to_left(Y,X,R-Q).
merge_class(X,Y,P-Q):-
        anchor_of(X,A),
        anchor_of(Y,B),var(B),!,
        defrost(Y,P-R),
        merge_to_left(X,Y,R-Q).
merge_class(X,Y,P-Q):-
        anchor_of(X,A),
        anchor_of(Y,B),
        unify(A,B,P-R),
        merge_to_left(X,Y,R-Q).

:- mode demon_to_left(+,+,?).
demon_to_left(X,Y, E):-
        flag_of(X,F), flag_of(Y,F),
        demon_of(X,DX), demon_of(Y,DY),
        tail(DX,DY, E).

:- mode merge_to_left(+,+,?).
merge_to_left(X,Y,M):-
        class_of(X,C), class_of(Y,C),
        up_of(Y,X),
        slots_of(X,SX), slots_of(Y,SY),
        merge_slots_to_left(SX,SY,M).

:- mode merge_slots_to_left(+,?,?).
merge_slots_to_left(X,Y,H-H):-unbound(Y),!.
merge_slots_to_left(X,Y,P-Q):-unify(Y,(S,T),P-R),!,
        merge_slots_to_left(X,S,R-R1),
        merge_slots_to_left(X,T,R1-Q).
merge_slots_to_left(X,S,E):-merge_slot(X,S,E).

:- mode merge_slot(+,?,?).
```

```
merge_slot(X,S,P-Q):-
        unify(S,K:U, P-R),
        find_key(K, X, V, R-R1),
        unify(U,V,R1-Q).

find_key(K,X,V,E):-unbound(X),!,unify(X, ((K:V),_), E).
find_key(K,S,V,P-Q):-
        unify(S,(H,T),P-R),!,
        (find_key(K,H,V,R-Q);find_key(K,T,V,R-Q)),!.
find_key(K,S,V,E):-unify(K:V,S,E).

tail(X, Y, E):-unbound(X),!,unify(X,Y,E).
tail(X, Y, P-Q):-unify(X, _&Z, P-R),
        tail(Z, Y, R-Q).
```

(9) Private Portray

```
:-public out_form/2.
:-mode out_form(?,-).

portray(X):-out_form(X,Y),write(Y),fail.
portray(_).

out_form(X,X):-
        (var(X);atomic(X)),!.
out_form(X,Y):-binding(X,B),!,
        out_form(B,Y).
out_form(X,Y):-
        functor(X,F,N),
        functor(Y,F,N),
        out_form(N,X,Y).
out_form(0,_,_):-!.
out_form(N,X,Y):-
        arg(N,X,A),
        arg(N,Y,B),
        out_form(A,B),
        M is N-1,
        out_form(M,X,Y).
```

(10) CIL Main and Debugging

```
:-public portray/1,binding/2.
c:- gcguide(cost,_,100),cil.

cil:-repeat,
        prompt(_, '> '),
        read(X),
        (X=[],! ;
         reduce1(X),fail).

reduce1((X;Y)) :-
        reduce(X),
        print(Y),
        ignore_cr(K),
        (K\==59,!,nl,print(success),nl
        ;
        true).
reduce1((X;Y)) :-!,nl,print(nomore),nl.

reduce1(X):- reduce(X),!,nl,print(success),nl.
reduce1(X):- nl,print(fail),nl.

ignore_cr(Com) :-
   ttygetC(X),
   ((X == 31,!,Com = X) ;
    ([X] == " ",!,ignore_cr(Com)) ;
     X = Com, ignore_cr(_)).

snap(A,B):-snap_flag,!,
         print(A),print(B),print(' ? '),
         ignore_cr(K),
         interpret(K).
snap(_,_).

interpret(31):-!. % <CR>
interpret(B):-"b"=[B],!,cil.
interpret(A):-"a"=[A],!,abort.
interpret(N):-"n"=[N],!,off.

push:-  setof(X, snap_declared(X), D),D\==[],!,
        asserta(trace_saved(D)),
        print('SAVED'(D)),nl.
push:- print('*** NO SPYPOINTS ***'),nl.

pop:-retract(trace_saved(D)), !, on(D), print('UNSAVED'(D)),nl,nl.
pop:-print('*** EMPTY SPY-STACK ***'),nl,nl.

off:-   setof(X, snap_declared(X), D), !,
        abolish(snap_declared,1),
        nochase,
        print('SPY-POINTS OFFED: '),
        print(D),nl,nl.
off:-   nochase,print('*** NO ACTIVE SPY-POINTS ***'),nl.
off([]).
off([X|Y]):-(retract(snap_declared(X)),fail;true), off(Y).

chase:-assert(snap_flag),assert(snap_all_flag).

nochase:-abolish(snap_flag,0), abolish(snap_all_flag,0).

on([]):-!.
on([X|Y]):-!,assert(snap_flag),on1([X|Y]).
on(X):-on([X]).
on1([]).
on1([X|Y]):-assert(snap_declared(X)),on1(Y).

s:-show.
show:-
        active_spy_points(ASP),
        show_active(ASP),nl,
        show_stack.
show_active([]):-
        print('*** NO ACTIVE SPY-POINTS ***'),nl.
show_active(ASP):-
        print('ACTIVE SPY-POINTS: '),
        print(ASP),nl.

show_stack:-empty_stack,!,print('*** EMPTY SPY-STACK ***'),nl.
show_stack:-print('SPY-STACK: '),nl,
            trace_saved(D), print('  '),print(D),nl,fail.
show_stack.

active_spy_points(E):-setof(X, snap_declared(X), E),!.
active_spy_points([]).

empty_stack:- trace_saved(_),!,fail.
empty_stack.
```