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

Towards a Computational Interpretation of Situation Theory

Hideyuki Nakashima*
Electrotechnical Lab. and CSLI †

Hiroyuki Suzuki
Matsushita Electric Industrial Co. Ltd. and CSLI

Per-Kristian Halvorsen
Xerox Palo Alto Research Center and CSLI

Stanley Peters
Stanford University and CSLI

ABSTRACT

This paper describes the design of a programming/knowledge representation language, PROSIT, based on situation theory. Our goal is to provide facilities within the logic programming paradigm for stating theories which allow self-reference, relativization of assertions to situations, direct access to the relationship between situations etc. We provide a computational account of the notion of situated inference, and compare it to other approaches to deduction. The syntax, operational semantics and data structures of the language are presented in detail. The paper concludes with examples of some novel aspects of the query mechanism of PROSIT, as well as the treatment of identity and inheritance.

1 Introduction

This paper describes the design of a programming/knowledge representation language, PROSIT (*Pro*gramming in *Si*tuation *T*heory). As the name implies, the language is based on situation theory (Barwise and Peters 1988; Devlin to appear). The motivation for embarking on the design of another language rests on the unique collection of features which are supported by situation theory:

1. the use of partially specified objects (e.g. situations, parameters) and a general treatment of partial information
2. situations as first class citizens of the theory
3. a formal treatment of informational *constraints*
4. a proper treatment of self-referential expressions

These features make situation theory particularly well suited for the analysis of semantic phenomena in natural language and other instances of communication and information flow. But at the same time these attributes of situation theory put it beyond the comfortable reach of programming environments that provide integral support for knowledge representation, interactive querying and deduction. This is evidenced by the work of Mukai (1985), which extends Prolog to contend with one particular construct introduced by situation theory, viz. *complex indeterminates*, as well a range of other work which addresses the more general questions of how to accommodate partially specified objects in Prolog (Mukai 1987, Aït-Kaci and Lincoln 1988).

The long-term goal of PROSIT is to provide facilities within the logic programming paradigm for stating theories which allow self-reference, relativization of assertions to situations, direct access to the relationship between situations etc. In a logic programming language, a set of clauses can be viewed from two different perspectives. On the one hand, they can be taken as a declarative statement of a "theory". Alternatively, they can be viewed as a program. According to the latter view the clauses specify the computation which will yield a simultaneously satisfying set of bindings for the free variables in the clauses, provided such a binding exists. Given this dual status of the statements of a logic program, situation theory's treatment of objects and operations opens up the possibility of simplifying specification of a range of computations that involve notions such as mutual knowledge, and partial knowledge. (See section 5 for some basic examples of such applications.) This comes in addition to the more obvious advantage to practitioners of situation semantics of having a programming language specifically tailored for the implementation of natural language systems based on situation semantics.

In addition to creating a programming language to accommodate the fundamental objects and operations of

*This author is partially supported by the Science and Technology Agency to visit CSLI.

†Center for the Study of Language and Information, Ventura Hall, Stanford University, Stanford, California 94305 U.S.A.

situation theory, we are also attempting to probe an, as yet, underdeveloped area of situation theory itself, i.e. its deductive features (see also chapter 4 of Fenstad et al. (1987)). We believe that inference is a very important component of intelligent behavior in general, and of successful linguistic communication in particular. This motivates our attempt at providing a computational interpretation of aspects of the logic of situation theory, even though any effort in this direction at this point in time has to be incomplete.

If one thinks of situation theory on the model of set theory, it may seem odd to think of it as generating deductions. However, situation theory includes constraints between facts and situations, such as various forms of involvement relation (e.g. logical involvement ($\Rightarrow_l$), conventional involvement ($\Rightarrow_c$)). This is the logical component of situation theory that drives the computational interpretation of a set of statements.

2 A Computational Perspective on Situation Theory

2.1 Situated Inference

In knowledge representation, it is well known that the form of representations has a great influence on the efficiency of their manipulation. The difference between efficient and inefficient representations sometimes turns out to be the difference between practicality and impracticality of actually performing the desired deductions.

In situation theory the inference system is supposed to be situated, i.e. relativized to a situation—a part of the world. This means not only that the inferences and conclusions drawn are dependent on the situation, but also that the rules themselves are compiled into the situation so that the inference may become more efficient. Most previous theories neglected this fundamental fact, so the representation of knowledge in the systems implementing them took place *sub specie aeternitatis* without any reference to the situation.

Many-sorted logic is one of the exceptions to this general rule. It opens the possibility of more efficient deduction by limiting the domain of variables. However, many-sorted logic is not situated since it must always explicitly state the domain. Moreover, it is efficient only in limiting the domain. In PROSIT, we achieve a similar, but more general, effect by classifying information by situations. Although we are going to give an example in which we form situations according to the domain, we can apply the same technique to classify situations according to spatio-temporal locations, or any other property.

The information that all canaries are yellow has been conventionally represented as

$$\forall x(canary(x) \rightarrow color\_of(x, yellow)). \quad (1)$$

Note that, in this formulation, the truth value of the formula is determined relative to *all* the objects in the total universe of discourse.

If we relativize the first argument of the relation *color_of* to the class of canaries, we can treat *color_of* as a one-place relation, which holds of the color yellow. I.e. $color\_of(yellow)$ is a fact relative to the class of canaries. In PROSIT we represent this as follows:

$$S_{canary} \models color\_of(yellow). \quad (2)$$

This formalization is quite similar to that of many-sorted logic:

$$\forall x/_{canary}\ color\_of(x, yellow). \quad (3)$$

Computationally speaking, the advantage of having situated knowledge is obvious. In the case of inquiring about the color of a particular canary, Tweety, it is not necessary to invoke an inference rule. The answer follows through inheritance.

However, the case for situated knowledge is not only, or even primarily, motivated by computational considerations. The efficiency of language (i.e. the same expression taking on different or specialized interpretations on different occasions of use) is similarly a reflection of the ability to exploit context to provide "missing" information. In other words, all representations of information, including linguistic expressions, depend on situational context for their interpretation. Situated knowledge serves to make more economical linguistic expressions, internal representations, and inference alike.

Situation theory has so far captured only partially the nature of situated inference. To aid in accounting for the behavior of intelligent agents, we wish to actually use situated inference in our reasoning. Thus our system explicitly mentions and manipulates situations. It is not limited to viewing situations from outside; the internals of the system are also organized as situated.

It should be emphasized again that our method of partial evaluation to capture situated knowledge is not limited to domain hierarchies. It can equally well be applied, for example, to the case of temporal reasoning. In temporal reasoning, there are two possibilities given our approach. On the one hand we can consider a situation, s, spanning an interval as in

$$s \models kissing(John, Mary, l). \quad (4)$$

Here an argument role is required to represent the temporal location. Alternatively, we can divide the information in a larger situation into subsituations according to time slices, l, where $l \subseteq s$. This can be represented as

$$l \models kissing(John, Mary). \quad (5)$$

This form, having no temporal parameter, increases the efficiency of certain types of deductions.

2.2 Deduction with Infons and Constraints

PROSIT mirrors situation theory in distinguishing a special type of information that 'generates new facts', called a *constraint*. Constraints are just a special case of *infons*, situation theory's units of information. We restrict attention to situations that respect every constraint they support; in other words, all a situation's infons should agree with the constraints imposed on the situation. In PROSIT if a constraint $\sigma \Rightarrow \tau$ holds in a situation s, then the situation is closed under that constraint, i.e. if σ holds in s, then τ also holds in s. [1] A constraint usually contains variables which are instantiated to particular objects upon application of the constraint. An example of a constraint is:

$$kissing(x, y) \Rightarrow touching(x, y). \qquad (6)$$

Infons and constraints determine the deductive behavior of the system via certain proof rules, which constitute an operational semantics. These proof rules must all be sound, in terms of the situation-theoretic behavior of the objects PROSIT expressions are intended to denote; they need not, however, be complete, in the sense of licensing the deduction of every consequence that validly follows. When infons are asserted, entries are made in the database and the forward-chaining constraints[2] are consulted and applied when matched. Similarly, when forward-chaining constraints are asserted, the database is updated to comply with the constraints. By these mechanisms, the system can conclude a wide range of valid consequences of assertions made to it about what infons hold in various situations, though not every valid consequence.

When a query is made, the system utilizes backward-chaining constraints in a laxer deductive system to try and 'prove' the query from the database. We add to the forward-chaining rules further proof rules that allow the system to induce that the database contains positive instances in support of constraints. For example, if $white(a)$, $white(b)$, $swan(a)$, and $swan(b)$ hold in a situation, and this list includes the only instances of $swan(x)$ proven to hold in that situation, backward-chaining rules will conclude that the constraint $swan(x) \Rightarrow white(x)$ might hold in the situation, or as we shall say that the situation *permits* the constraint. Clearly, the constraint may not hold, and counterexamples to it may even exist. Nevertheless, this is a useful programming mechanism for information processing.

[1] This is analogous to the distinction between necessarily true and contingently true facts discussed, in the context of knowledge representation languages, in Krypton (Brachman and Levesque 1983).

[2] See section 4.4 for an explanation of the different types of constraints in PROSIT.

3 Implementation

3.1 Strategy

PROSIT is implemented as a modification of the Uranus system (Nakashima 1986). Uranus is a logic programming language equipped with a multiple world mechanism. As we will show in the example section (section 5), the multiple world mechanism is similar to the use of situations. Limited modification of the Uranus system made possible a full implementation of PROSIT. The revisions are largely focused on extensions of the unification mechanism (see section 3.2) and introduction of forward chaining.

Since the design of PROSIT is still in an experimental stage, part of PROSIT interpreter is written in core PROSIT itself. As we gain more experience using the language to greater application, we will rewrite the whole code in Lisp to make it run more efficiently.

3.2 Unification

As in other logic programming languages, unification is an important computational component in PROSIT. Our notion of unification is more general than term unification, used in Prolog. For example, unification of two situations is possible.

The basic rules of unification are:

1. A variable is unifiable with any other data structure.
2. A parameter is not unifiable with anything but itself. It is desirable to distinguish the unification of parameters from assertions of equality of parameters. When unifying two sets of infons, (see below), we recursively unify the constituents of the infons in the sets, including any parameters. In this case the parameters should remain distinct. However, one may, after having accumulated information about two parameters eventually discover that they denote the same object, in which case their merger should be possible, e.g. through an explicit assertion of equality.
3. A constant is not unifiable with anything but itself
4. Two infons are unifiable only when their constituents are componentwise unifiable.
5. Unification of two sets of infons yields another set of infons which consists of the set of infons resulting from pairwise unification of the infons in the two sets (Rounds 1988). This is related to taking the intersection of the two sets. On analogy with the operation of intersection, the unification of two sets of infons always succeeds, but the result may be empty (i.e. contain no infons).

3.3 Forward and Backward Chaining

As stated earlier, PROSIT uses both forward and backward chaining rules. Backward chaining in PROSIT is weaker in power than forward chaining. We believe that this distinction is paralleled by a distinction in the deductive capability of humans. We are also weak in foreseeintg all the possible consequences of a given set of facts.

Forward chaining rules are activated whenever a matching new assertion is added. All the consequences are then asserted, which may in turn activate other forward chaining rules recursively. By default, all the consequent assertions are added to the initial situation to which the original assertion is added, unless another situation is explicitly stated in the rule.

Backward chaining rules are activated when a query (a goal) is entered. The mechanism is similar to those used in Prolog implementations. The difference is that there may be super-situations to search for rules/facts to use. The Uranus implementation is used here with only a slight modification, which is necessary because PROSIT uses negative propositions as well as positive ones.

As a natural consequence of the above rules, although backward chaining rules may utilize the result of forward chaining rules, the opposite is not the case. Even if a proposition is proven using backward chaining rules, the result is not asserted and does not trigger any forward chaining rules. The information the system has may be partial and the result of applying backward chaining rules to partial information may be invalidated with presentation of further information. On the other hand, since only definitely true facts are to be asserted, the result of applying forward chaining rules to the assertions is to be persistent. Considering these distinctions, the division of forward and backward chaining rules is more than procedural. Only persistent rules are to be introduced as forward chaining rules.

4 Components of Programs

4.1 Sketch of the Language

Before we go into details about the language, we will sketch the whole structure of a program using extended BNF. Non-terminal symbols are in brackets; terminal symbols are in double-quotes; symbols in braces are optional.

< *program* > ::= (< *assertion* > | < *query* >)$^+$
< *assertion* > ::= {< *label* >} ":" < *infon* >
< *label* > ::= < *parameter* >
< *query* > ::= "?" < *infon* >
< *infon* > ::= < *issue* >|
"(" < *polarity* > < *issue* > ")"
< *polarity* > ::= "yes" | "no"
< *issue* > ::= "(" < *relation* > < *term* >* ")"
< *relation* > ::= < *structural_relation* >|
< *relational_parameter* >
< *structural_relation* > ::= "|=" | "<=" | "=>"|
"<=>" | "=" | "<-" | "->" |
"s*" | "s+"
< *relational_parameter* > ::= < *parameter* >
< *term* > ::= < *variable* > | < *parameter* >
| < *constant* >| < *compound_term* >
< *compound_term* > ::= < *infon* > |
"[" < *variable* > ":" < *infon* > "]"
< *variable* > ::= an atomic symbol beginning
with a capital letter
< *parameter* > ::= an atomic symbol not
beginning with a capital letter
< *constant* > ::= string

4.2 Parameters, Variables and Constants

Parameters are used to represent things in the world, e.g. individuals, situations, relations, infons and propositions. Usually, different parameters correspond to different entities. They are not unifiable, but may be merged by explicit assertions of identity, e.g. `(= x y)` (cf. section3.2).

An infon may not be fully specified. It may contain a pointer to possibly unknown objects as in

$$kissing(Mary, someone). \qquad (7)$$

Expressions, such as *someone* in (7), stand for a particular entity, just like "the person standing next to you" stands for a particular person. We may, at any one point in time, not have enough information to uniquely identify the entity. In PROSIT, we call these expressions/objects *parameters*. These unknown objects should not be confused with variables.

Variables only occur in constraints. In the constraint

```
(=> (kissing X Y) (touching X Y))
```

`X` and `Y` do not stand for something fixed. They are simply a device to connect various mentions of the same entity.

A variable is distinguished by its initial capital letter in this paper. Variables are instantiated to parameters, previously used or not, at the time of application of the constraint. Parameters may also get anchored to constants. A constant is distinguished by surrounding double-quotes "".

A crucial difference between variables and parameters is that the scope of variables is local to the constraint in which they appear while the scope of parameters is global.

4.3 Situations and Infons

Some parameters correspond to real situations. Those parameters are called *situational parameters*. They are associated with sets of *infons*, which classify real situations. An infon is a basic unit of information characterizing relations among parameters. A situation *supports* an infon if the infon is explicitly asserted to hold in the situation, (|= *situation infon*), or can be proved to hold by application of forward-chaining constraints in the situation. In PROSIT, a situation parameter corresponds to the set of infons which it supports.

We say that an infon is *permitted* by a situation if that infon is deduced through application of backward-chaining constraints (<=).

An infon consists of a relation followed by its arguments (possibly none). There may be a yes/no prefix for an infon indicating its *polarity*. If the polarity is omitted, it defaults to yes.[3]

The argument roles of an infon are filled by terms.

Here are examples of infons:

```
(p X [father X])
(yes (p X Y))
(no (p a b))
(=> (p X) (q X))
(p (q X))
```

There are several primitive relations to describe the structure of situations:

- (|= *sit infon*) asserts that *sit* supports *infon*. Statements of this particular form are also called *propositions* (cf. section 4.6).
- (-> *sit1 sit2*) asserts that *sit1* is a super-situation of *sit2*. (This form of statement is also called a *proposition*.) This operation has two computational effects:
  1. *sit1*, viewed as a set of infons, becomes a superset of *sit2* and will constrain all the infons in *sit2*.
  2. As a direct consequence of the above, all the constraints in *sit1* also apply to infons in *sit2*. This is the equivalent of *inheritance* in knowledge representation. See section 5.2 for details.
- (s* *sit1 sit2 sit3*) causes the unification of two sets of infons (the set of those supported by *sit1* and the set supported by *sit2*) and associates the result with the parameter *sit3* (cf. section 3.2).
- (s+ *sit1 sit2 sit3*) makes *sit3* be the union of *sit1* and *sit2*.

[3]The decision to make polarity a prefix is motivated solely by considerations of efficiency of implementation.

An example: Suppose that

```
(|= sit1 (love x "John"))
(|= sit2 (love "Mary" y))
```

then, by

```
(s* sit1 sit2 sit3)
```

we will get

```
(|= sit3 (love "Mary" "John"))
```

Similarly, by

```
(s+ sit1 sit2 sit4)
```

we will get

```
(|= sit4 (love x "John"))
(|= sit4 (love "Mary" y))
```

4.4 Constraints

Constraints are special infons whose relation is one of <=, => and <=>. The argument roles of constraints are filled by types of situations. Types of situation are written in a form identical to infons. One may even think of them as infons, since they designate those situations in which the corresponding infons hold. For example, a constraint

```
(=> (kissing X Y) (touching X Y))
```

consists in a relation, the *involves*-relation, holding between two types of situations: situations of the kissing-type involve situations of the touching-type.

The constraint, (<= *t1 t2*), is used as a *backward chaining* rule: To prove that *t1* holds, we try to prove that *t2* holds in the situation.[4]

Example:

```
(<= (append [cons A X] Y [cons A Z])
    (append X Y Z))
```

[4]Proof by contradiction and as a sub-case of this, resolution, cannot be usefully employed in situation theory. The reason is that the following are all possible: 1) $s \models p$, 2) $s \models \neg p$ and 3) $s \not\models p$ nor $s \not\models \neg p$. Therefore, refuting the case $s \models \neg p$ does not entail $s \models p$.

Backward (forward, or any) chaining, on the other hand, is believed to be a sound proof procedure for situation theory (Plotkin to appear). The behavior here is similar to that of three-valued logic (for example, consult chapter 4 of Fenstad et al.).

The constraint, (=> *t1 t2*), is used as a *forward chaining* rule: Whenever *t1* is asserted, *t2* is also asserted in the situation.

<=> acts as the combination of the both of the above.

In the case of both forward- and backward-chaining constraints, only the first infon is considered to be on the left-hand side.

$$(\Leftarrow i_1 i_2 ... i_n)$$

is interpreted as

$$i_1 \Leftarrow i_2 \wedge ... \wedge i_n.$$

Infons in the right-hand side of constraints are considered to be conjoined.

The decision to break up the single notion of constraint in situation theory into forward-chaining constraints and backward-chaining constraints was motivated by considerations of computational efficiency. Depending on the context, one of the two strategies may be clearly more efficient. The distinction puts the choice of how to "implement" a rule under the programmer's choice. [5]

4.5 Compound Terms

Compound terms are variables with associated restrictions (Nakashima 1985):

[*variable* : *infon*],

e.g.

```
[X : (|= s (father "Adam" X))].
```

For simplicity, we can omit explicit mention of the situation if it is the same as the situation of the surrounding proposition. The "(" and ")" of the infon can also be deleted. Furthermore, when there is no need to explicitly name the variable, i.e., the variable does not appear somewhere else, it may be omitted. Thus, we can write the previous compound term as

```
[father "Adam"]
```

Note that the square brackets distinguish this as a term rather than an infon.

When the relation in a term is not defined it acts as a constructor of a complex data structure (just as terms in Prolog).

4.6 Propositions

4.6.1 Assertions

As assertions, we only accept propositions. Propositions have one of the two forms:

$$situation \models infon$$

$$sit1 \geq sit2$$

and are written as:

label :(|= *situation infon*)

or

label :(-> *sit1 sit2*)

respectively, where *label* is a *propositional parameter* denoting the proposition.

For example,

```
p:(|= world (father "Jon" "Mary"))
q:(|= s (false q))
```

are propositions. Note that the second proposition is self-referential.

It is possible to omit *label*. In this case, it cannot be referred to by name from other propositions.

If *situation* is omitted, it defaults to `world`. Actually, in the two examples above, the supporting relation itself is regarded as an assertion to `world`. We define `world` as a superset of all situations.

Infons in the world are not necessarily inherited by other situations. On the other hand, constraints (represented as infons) in the world apply to all situations. See section 5.2 for more details.

When an assertion is added, forward-chaining rules are applied. No backward-chaining rules are used at this time.

4.6.2 Query

Queries have the following form:

- $?S \models P$ asks if S supports P.
- $?P$ asks if P holds in the world.

[5] Generally speaking, forward-chaining rules are space consuming, while backward-chaining rules are time consuming. Forward chaining causes the addition of propositions to the database, and consequently consumes storage. However, forward chaining provides the advantage that since all the true propositions licensed by forward chaining are added to the database, checking a query requires little computation, just a database lookup.

Backward chaining on the other hand does not cause proven propositions to be added to the database.

The system's response when a query is entered may depend on its understanding of the intention of the user (see section 5 for details). There are several possible actions:

1. Answering by yes/no.
2. Returning one solution, where a solution is a possible anchor for the parameters in the query, and additional solutions on request.
3. Returning all solutions.
4. Returning the most general formula to match the query. This might be the query itself if no further information is found.

In all cases, the basic computation executed upon the entry of a query is done by the application of backward-chaining constraints.

5 Examples

5.1 Treatment of Identity

The role of parameters in situation theory could be seen as a means to keep track of the correspondence between concepts in the mind and real objects in the world (Israel and Perry 1988). This correspondence may depend on one's beliefs. PROSIT provides a treatment of identity between parameters depending on situations, which classify beliefs. This enables us to treat certain phenomena involving belief-contexts.

The renowned Roman orator Cicero's first name is Tully. For someone who knows of this identity, it is easy to answer yes to the question "Is Tully an orator?", but it is impossible for someone who do not know this identity.

In PROSIT we can express the difference between the belief of someone who knows the identity of Cicero and Tully and the belief of someone who does not know this fact. In other words, we allow assertions of situation-dependent equalities between parameters. If a person knows the identity of Cicero and Tully, his belief is classified by the situational parameter s1 where

```
:(|= s1 (= cicero tully))
:(|= s1 (orator cicero))
```

On the other hand, the belief of someone who does not know the identity is classified by s2 where

```
:(|= s2 (orator cicero))
```

In this environment, our system works as follows:

```
?(|= s1 (orator tully))
yes
?(|= s2 (orator tully))
unknown
```

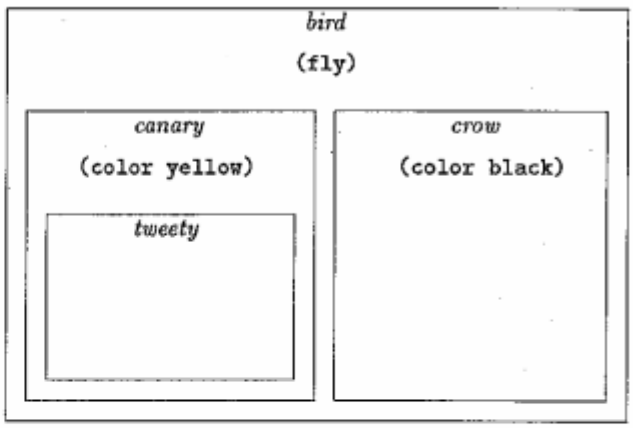

Figure 1: Situations describing concept hierarchy

5.2 Describing Hierarchy and Inheritance

In our model, the world subsumes all other situations. Let S be the collection of all situations. Then,

$$\forall s \in S(s \subseteq \texttt{world})$$

Any constraint supported by `world` is also respected by all sub-situations. Moreover, any constraint supported by a larger situation is respected in its sub-situations.

We can use this partial ordering of situations to express class hierarchies and inheritance through classes.[6] We can view the subset relation on situations as a generalization of the "a kind of" relation on classes.

```
:(|= bird (fly))
:(|= canary (color yellow))
:(-> canary bird)
:(-> tweety canary)
```

Note that `(fly)` and `(color yellow)` state that `(fly)` and `(color yellow)` hold in any situation which is subsumed by *bird* and *canary*, respectively.

If a user asks

```
?(|= tweety (fly)),
```

the system answers "yes" by applying the constraint (backward chaining).

We discussed earlier (in section 2.1) that the above is a situated way of representing knowledge.

In section 2.1, we considered the relationship between situated and unsituated knowledge. In Figure 1 we have individuated 4 different situations: *bird*, where all infons are relativized to the class of birds; *canary*, where all are

[6] This formalization is exactly the same as used in Prolog/KR (Nakashima 1984) and its successor Uranus.

relativized to canaries; *crow*, where all are relativized to crows; and *tweety*, where there is, let us assume, only one individual, Tweety, and all infons are relativized to him (or is it her?). Given this range of situations, we can establish a conversion between situational parameters and predicates, as follows.

```
(<=> (|= world (<= (color X yellow)
                   (canary X)))
     (|= canary (color yellow)))
```

This two-way constraint says that

```
(|= world
    (<= (color X yellow) (canary X)))
```

and

```
(|= canary (color yellow))
```

are equivalent. But procedurally, that is, from the situated-inference point of view, these propositions express two different things:

1. When we are told an unsituated statement:

   ```
   (|= world
       (<= (color X yellow) (canary X)))
   ```

   (and believe it), we should adjust our model of the situation so that it reflects this fact (forward reasoning).

2. When we want to see if

   ```
   (|= world
       (<= (color X yellow) (canary X)))
   ```

   is true, we can just check the situation (backward reasoning).

We could also provide a meta-level constraint to convert situated knowledge into unsituated. It is an important feature of our system that the conversion rule itself is within the scope of the system's descriptive power, due to the fact that situations are themselves first-class objects.

```
(<=> (|= world (<= (Rel X . Args) (S X)))
     (|= S (Rel . Args)))
```

5.3 Query Feature

Consider the following information:

John is Mary's grandfather.
John is Bill's grandfather.

How should the following question be answered:

Are Mary and Bill cousins?

A desirable answer would be:

Yes, if they are not siblings.

Here is how PROSIT handles this situation: First, we assume several constraints on family relationships.

```
:(<=> (grandfather x y)
      (parent x z) (parent z y) (male x))
:(<=> (father X Y) (parent X Y) (male X))
:(<=> (mother X Y) (parent X Y) (female X))
:(<=> (cousin X Y)
      (parent W X) (parent V Y)
      (sibling W V))
:(<=> (cousin X Y) (cousin Y X))
:(<=> (sibling X [Y:(=/= X Y)])
      (parent Z X) (parent Z Y))
:(<=> (sibling X Y) (sibling Y X))
```

Note that the cousin and sibling relations are commutative. Forward chaining of commutative rules must terminate after its first application. In general, for circular chains a loop test is required. This is easily handled given the graph representation of constraint dependency.

Note also the use of a term to test non-equality. Predicates which are only testable (i.e, cannot be used to return any value) should only be used as constraints on variables. They cannot occur on the right-hand side of constraints.

Now, the system is ready to accept the assertion:

```
:(grandfather "John" "Mary")
```

When this is asserted, forward-chaining rules are activated and the following assertions are also added:

```
:(parent "John" z)
:(parent z "Mary")
:(male "John")
```

Note that the link between two z's is maintained. Anchoring one of them should also anchor the other one to the same thing. Similarly, when

```
:(grandfather "John" "Bill")
```

is asserted,

```
:(parent "John" z')
:(parent z' "Bill")
:(male "John")
```

are also added. At last, the question is entered:

```
?(cousin "Mary" "Bill")
```

After trying to prove the above proposition, the term `[z:(=/= z' z)]` remains. By default this relation is true since no two parameters are equal. Nevertheless, there still is a possibility that they are equal. We can define a "careful" equality check, so that

```
(|= careful-mode (=/= z z'))
```

is unknown. This should be enough information to answer: Yes, if `(=/= z z')`.

Acknowledgments

We would like to thank CSLI and the System Development Foundation for providing a cooperative and stimulating environment. We are also grateful to the members of the MOST group at CSLI, especially Syun Tutiya, for detailed comments as well as enlightening discussions.

Appendix A Operational Semantics of PROSIT

In PROSIT assertions associate each situation s with a set of infons. $(\models\ s\ \sigma)$ means that the infon σ is in the set associated with s. PROSIT allows declarations that the subset relation holds for the infons associated with two situations. $(\sqsubseteq\ s\ s')$ represents that the set of infons associated with s is a subset of that for s'.

$(\models^*\ s\ \sigma)$ means that PROSIT will make an affirmative answer for the proposition that σ holds in s if queried.

Convention: A variable in a constraint $(\Rightarrow\ \sigma\ \tau)$, $(\Leftarrow\ \sigma\ \tau)$, or $(\Leftrightarrow\ \sigma\ \tau)$ is bound by the involves relation of the constraint iff no proper subformula is a constraint containing all occurrences of the variable.

Notation: A substitution ρ is a partial function from the set of variables to the set of parameters, constants, and complex expressions. $\sigma[\rho]$ denotes the result of replacing all occurrences in σ of each variable X in the domain of ρ by $\rho(X)$.

$$\frac{(\models\ s\ (=\ a\ b)), (\models\ s\ \sigma)}{(\models\ s\ \sigma[a/b])} \qquad \frac{(\models\ s\ (=\ a\ b)), (\models\ s\ \sigma)}{(\models\ s\ \sigma[b/a])}$$

Note: Here $\sigma[a/b]$ denotes the result of replacing a in σ by b.

$$\frac{(\models\ s\ (\Rightarrow\ \sigma\ \tau)), (\models\ s\ \sigma[\rho])}{(\models\ s\ \tau[\rho])}$$

whenever ρ is defined on the variables bound by the involves relation.

$$\frac{(\models\ s\ (\Leftrightarrow\ \sigma\ \tau)), (\models\ s\ \sigma[\rho])}{(\models\ s\ \tau[\rho])}\ \text{as above.}$$

$$\frac{(\models s\ (and\ \sigma\ \tau))}{(\models s\ \sigma)} \qquad \frac{(\models s\ (and\ \sigma\ \tau))}{(\models s\ \tau)} \qquad \frac{(\models s\ \sigma), (\models s\ \tau)}{(\models s\ (and\ \sigma\ \tau))}$$

$$\frac{(\sqsubseteq\ s\ s'), (\models\ s\ \sigma)}{(\models\ s'\ \sigma)} \qquad \frac{(\sqsupseteq\ s'\ s), (\models\ s\ \sigma)}{(\models\ s'\ \sigma)}$$

$$\frac{(\models\ s\ \sigma)}{(\models^*\ s\ \sigma)}$$

$$\frac{(\models^*\ s\ (=\ a\ b)), (\models^*\ s\ \sigma)}{(\models^*\ s\ \sigma[a/b])} \qquad \frac{(\models^*\ s\ (=\ a\ b)), (\models^*\ s\ \sigma)}{(\models^*\ s\ \sigma[b/a])}$$

$$\frac{(\models^*\ s\ (\Leftarrow\ \tau\ \sigma)), (\models^*\ s\ \sigma[\rho])}{(\models^*\ s\ \tau[\rho])}\ \text{as above.}$$

$$\frac{(\models^*\ s\ (\Leftrightarrow\ \tau\ \sigma)), (\models^*\ s\ \sigma[\rho])}{(\models^*\ s\ \tau[\rho])}\ \text{as above.}$$

$$\frac{(\models^*\ s\ \tau[\rho_1]), \ldots, (\models^*\ s\ \tau[\rho_n])}{(\models^*\ s\ (\Leftarrow\ \tau\ \sigma))}$$

whenever $\rho_1, \ldots, \rho_n$ are all the substitutions defined on the variables bound by the involves relation such that $(\models^*\ s\ \sigma[\rho_i])$ for $1 \leq i \leq n$.

$$\frac{(\models^*\ s\ \tau[\rho_1]), \ldots, (\models^*\ s\ \tau[\rho_n])}{(\models^*\ s\ (\Rightarrow\ \tau\ \sigma))}\ \text{as immediately above.}$$

$$\frac{(\models^*\ s\ \tau[\rho_1]), (\models^*\ s\ \sigma[\rho_1]), \ldots, (\models^*\ s\ \tau[\rho_n]), (\models^*\ s\ \sigma[\rho_n])}{(\models^*\ s\ (\Leftrightarrow\ \tau\ \sigma))}$$

whenever $\rho_1, \ldots, \rho_n$ are defined on the variables bound by the involves relation and $(\models^*\ s\ \tau[\rho])$ or $(\models^*\ s\ \sigma[\rho])$ for any such substitution ρ just in case $\rho = \rho_i$ for some $i = 1, \ldots, n$.

$$\frac{(\models^*\ s\ (and\ \sigma\ \tau))}{(\models^*\ s\ \sigma)} \qquad \frac{(\models^*\ s\ (and\ \sigma\ \tau))}{(\models^*\ s\ \tau)} \qquad \frac{(\models^*\ s\ \sigma), (\models^*\ s\ \tau)}{(\models^*\ s\ (and\ \sigma\ \tau))}$$

REFERENCES

[1] Hassan Aït-Kaci and Patrick Lincoln. *LIFE: a Natural Language for Natural Language.* Technical Report aca-st-074-88, MCC, ACA program, 1988.

[2] Jon Barwise and Stanley Peters. *Naive Situation Theory.* Technical Report to appear, CSLI, 1988.

[3] R. J. Brachman and Hector J. Levesque. Krypton: a functional approach to knowledge representation. *Computer Oct.*, 67–73, 1983.

[4] Keith Devlin. *Logic and Information I: Situation Theory.* to appear, 1989.

[5] J. E. Fenstad, P.-Kr. Halvorsen, T. Langholm, and J. van Benthem. *Situations, Language and Logic.* D. Reidel, 1987.

[6] David Israel and John Perry. *What is Information.* Technical Report, CSLI, 1988.

[7] Kuniaki Mukai. *Anadic Tuples in Prolog.* Technical Report TR-239, ICOT, 1987.

[8] Kuniaki Mukai. *Horn Clause Logic with Parameterized Types for Situation Semantics Programming.* Technical Report TR-101, ICOT, 1985.

[9] Hideyuki Nakashima. Knowledge representation in Prolog/KR. In *Proc. of 1984 International Symposium on Logic Programming*, IEEE, 1984.

[10] Hideyuki Nakashima. Term description: a simple powerful extension to prolog data structures. In *Proc. of IJCAI 85*, pages 708–710, 1985.

[11] Hideyuki Nakashima. Uranus reference manual. *Bulletin of Electrotechnical Laboratory*, 50, 1986.

[12] Gordon Plotkin. Personal communication. 1988.

[13] William Rounds. *Set Values for Unification-Based Grammar Formalisms and Logic Programming.* Technical Report, CSLI, 1988.