COMMON HEURISTICS FOR PARSING, GENERATION, AND WHATEVER ...

HASIDA, Kôiti
Institute for New Generation Computer Technology (ICOT)
Mita Kokusai Bldg. 21F, 1-4-28 Mita, Minato-ku, Tokyo 108 JAPAN
Tel: +81-3-3456-3069, E-mail: hasida@icot.or.jp

ABSTRACT

This paper discusses general heuristics to control computation on symbolic constraints represented in terms of first-order logic programs. These heuristics are totally independent of specific domains and tasks. Efficient computation for sentence parsing and generation naturally emerge from these heuristics, capturing the essence of standard parsing procedures and semantic head-driven generation. Thus, the same representation of knowledge, including grammar and lexicon, can be exploited in a multi-directional manner in various aspects of language use.

1 Introduction

One lesson to learn from the repeated failure to design large AI systems in general is that the information flow in the cognitive systems is too complex and diverse to stipulate in the design of these AI systems. To capture this diversity of information flow, therefore, AI systems must be designed at a more abstract level where direction of information flow is not explicit.

This is where *constraint* paradigm comes in. Since constraints do not stipulate the direction of information flow or processing order, constraint-based systems could be tailored to have tractable complexity, unlike procedural systems, which stipulate information flow and thus quickly become too complex for human designers to extend or maintain.

Naturally, the key issue in the constraint-based approach is how to control information flow. A very general control schema independent of any specific domain or task is vitally necessary for the success of this approach.

The present paper introduces a system of constraint in a form of logic program, and a set of very general heuristics to control symbolic operation on the constraints. The symbolic operations here are regarded as transforming logic programs. They are quite permissive operations as a whole, allowing very diverse information processing involving top-down, bottom-up and other directions of information flow. The heuristics control this computation so that only relevant information should be exploited and the resulting representation should be compact. Parsing and generation of sentences are shown to be efficiently done under these heuristics, and a standard parsing algorithm and the semantic head-driven generation [8] naturally emerge thereof.

The rest of the paper is organized as follows. Section 2 describes the syntax of our system of constraint. Section 3 defines the symbolic computation on these constraints, and proposes a set of general heuristics to control computation. Section 4 and Section 5 show how sentence parsing and generation are executed efficiently by those heuristics. Finally, Section 6 concludes the paper.

2 Constraint Network

A program is a set of *clauses*. A clause is a set of *literals*. A *literal* is an *atomic constraint* with a sign in front of it. The sign is a '+', '-', or nil. A literal with a sign '+' or nil is called a *positive literal* and one with a sign '-' is a *negative literal*. An atomic constraint is an *atomic formula* such as `p(X,Y,Z)`, a *binding* such as `X=f(Y)`, a *feature specification* such as `a(X,Y)`, or an *equality* such as `X=Y`. Names beginning with capital letters represent variables, and the other names predicates and functions. A feature specification may be regarded as an atomic formula with a special binary predicate called a *feature*. A feature is a partial function from the first argument to the second argument: that is, if `a` is a feature and both `a(X,Y)` and `a(X,Z)` hold, then `Y=Z` must also hold. The other atomic constraints may be understood in the standard fashion. The atomic constraints other than equalities are called *proper atomic constraints*.

A clause is written as a sequence of literals it contains followed by a semicolon. The order among literals is not significant. So (1) is a clause, which may also be written as (2).

(1) `-p(U,Y) +q(Z) -U=f(X) -X=Z;`

(2) `+q(Z) -p(f(Z),Y);`

A clause containing a literal with null sign is a *definition clause* of the predicate of that literal. A predicate having definition clauses are called *defined predicate*, and its meaning is defined in terms of completion based on the definition clauses. For instance, if the definition clauses of predicate `p` are those in (3), the declarative meaning of `p` is given by (4).

(3) `p(X) -q(X,a);  p(f(X)) -r(X);`

(4) $\forall A\{p(A) \Leftrightarrow \{\exists Y(q(A,Y) \wedge Y = a) \vee \exists X(A = f(X) \wedge r(X))\}\}$

A predicate which is not a defined predicate is called a *free predicate*. There is a special 0-ary defined predicate `true`. Its definition clauses are called *top clauses*. A top clause corresponds to the query clause of Prolog, although the latter has `false` instead of `true`.

Programs are regarded as constraint networks. For instance, the following program is a network as in Figure 1.

```
(i)   true -member(a,X);
(ii)  member(A,[A|S]);
(iii) member(A,[B|S]) -member(A,S);
```

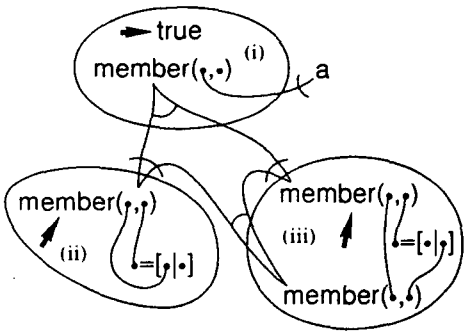

Figure 1: Constraint Network

In graphical representations like Figure 1, a '•' often represents an argument of an atomic constraint. There are two types of *nodes*: arguments, and proper atomic constraints. An argument is involved in at most one proper atomic constraint, but in any number of equalities. An argument bound to a constant is identified with that constant. That is, the first argument of a binding •=a, for instance, is represented simply by a. A *link*, represented as a curve, connects two nodes. For any two (possibly the same) nodes, there is at most one link connecting them. A link connecting two arguments is an equality between them. A link connecting two proper atomic constraints is called an *inference link*. No link connects an argument and an atomic constraint. Although often not explicitly shown, an inference link accompanies equalities between the corresponding arguments of the two proper atomic constraints. A *clausal domain* of clause Φ is the part of the constraint network consisting of the atomic constraints referred to as literals in Φ except equalities concerning constants. A clausal domain is depicted by a closed curve enclosing the included atomic constraints. The short thick arrows indicate the references to the atomic constraints as positive literals in clauses. A *predicate domain* of predicate π consists of all the proper atomic constraints with π (binding `X=f(Y)` is regarded as having binary free predicate `=f`, for instance), inference links among them, and equalities accompanying these inference links.

The *instantiation possibilities* of the constraint network is defined by regarding nodes and links as sets. Those sets are disjoint of each other. An instance of an argument corresponds to an individual in the domain of interpretation, and an instance of an atomic constraint corresponds to an atomic proposition. Constants (bindings to constants) and 0-ary atomic formulas are singleton sets. A link δ between nodes α and β stands for a symmetric relation. That is, $\delta = R \cup R^{-1}$ for some relation $R \subseteq \alpha \times \beta$. We call $\{x \in \alpha | \exists y\, x \delta y\}$ the *α-domain* of δ. Every link in a clausal domain or the predicate domain of a defined predicate is of the form $R \cup R^{-1}$ for some bijection R. Let Δ be the transitive closure of the union of all the links. $x \Delta y$ means that x and y correspond to the same object in the domain of interpretation if x and y belong to arguments, and that they correspond to the same atomic proposition if they belong to proper atomic constraints. We say that node α *subsumes* node β when $\alpha/\Delta \supseteq \beta/\Delta$; that is, for every $y \in \beta$ there exists $x \in \alpha$ such that $x \Delta y$. For each pair of a proper atomic constraint α and an argument β of α, there is a bijection ρ from α to β, such that $x \rho y$ holds iff $y \in \beta$ is an argument of $x \in \alpha$. ρ is called a *role assignment*.

Consider a part $\mathcal{P}$ of the constraint network and the minimum equivalence relation including the links and the role assingments in $\mathcal{P}$. A *layer* of $\mathcal{P}$ is an equivalence class with respect to this relation. A *splitting domain* is a part $\mathcal{S}$ of the network in which every link is of the form $R \cup R^{-1}$ where R is the union of $(\alpha \cap \mathcal{L}) \times (\beta \cap \mathcal{L})$ over all the layers $\mathcal{L}$ of $\mathcal{S}$ and α and β are the two endnodes of that link. Thus, if a link in a splitting domain splits into two links sharing an endnode α and having disjoint α-domains, then the entire splitting domain splits into two separate splitting domains each containing one of these two links. The clausal domains and predicate domains are assumed to be splitting domains.

A *joint* is a part of a node which connects the node with a link or more. Figure 2 shows some joints. The figures below illustrate the instantiation possibilities of the networks shown above by depicting each node as an ellipse enclosing its instances, and each link as a bundle of curves representing the pairs belonging to the link. A joint J of node α is depicted as an arc convex towards α crossing the links involved in J. A joint involving just one link, as in (a) and (b), is called a *unitary joint*, and one containing several links, as in (c) and (d), is called a *multiple joint*. Distinct links involved in the same multiple joint on node α have disjoint α-domains. A joint is *partial* if it stretches out of the involved links, as in (b) and (d), and *total* otherwise, as in (a) and (c). The union of α-domains of the links involved in the same joint on node α is equal to α. A total unitary joint as in (a) is not explicitly shown as an arc. Partial joints on node α are *complementary* when the union of the α-domains of the links involved in them is α. Complementary joints are indicated by a dashed arc crossing these links. So the union of the α-domains of the three links is α in (e). When node α and β are connected by link δ and the joint of β involving δ is total and unitary, α and δ are said to *dominate* β.

The initial structures of predicate domains are shown in Figure 3. Such structures, as well as the other structures, will change as computation proceeds.

3 Computation

Here we introduce a method of symbolic computation together with some general control heuristics for controlling computation. There are two types of symbolic operation: *subsumption* and *deletion*. Here we chiefly

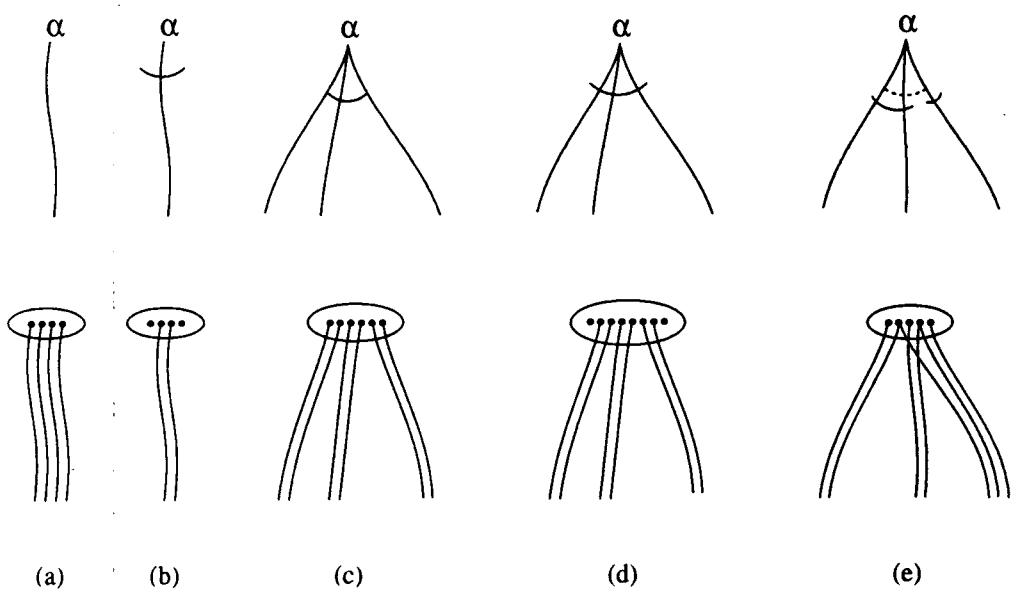

Figure 2: Joints between Nodes and Links

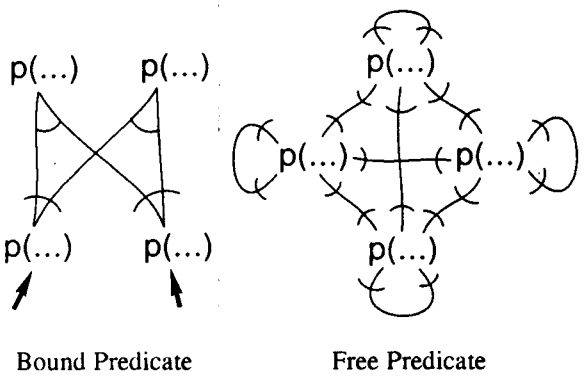

Figure 3: Predicate Domains

concern ourselves with subsumption.

3.1 Subsumption

'Subsumption' means two things: *subsumption relation*, which we defined above, and *subsumption operation*, which we discuss below.

The purpose of a subsumption is to let information flow from a node. A node α may have *probes*. α is called the *origin* of these probes. Each probe is placed on a link and directed towards an endnode. The origin of a probe subsumes the node behind the probe. Probes transmit information of their origins across the network via subsumptions. The origin of probes has its *scope*. The scope of node α is the part $\mathcal{S}$ of the constraint network satisfying the following conditions.

- $\mathcal{S}$ is a connected graph containing α.
- A node β is behind a probe π on link δ and with origin α, iff β is in $\mathcal{S}$ but δ is not.
- α subsumes every node in $\mathcal{S}$.

So the scope of α may be illustrated as in Figure 4, where arrows are probes, which just cover the boundary

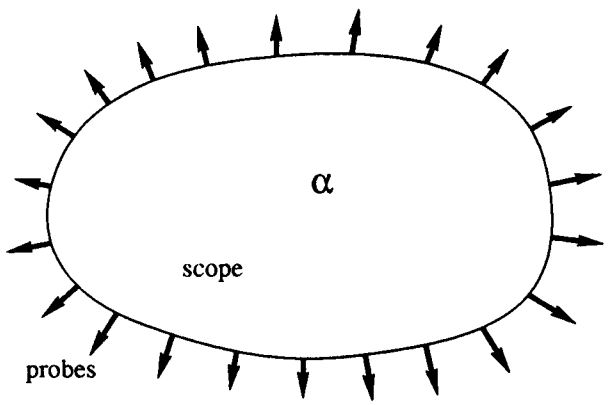

Figure 4: Scope of Node

of the scope.

Every node α can just once create probes on all the links connected to α so that α is behind these probes. Subsumption extends the scope of a node by advancing probes, while preserving the instantiation possibilities of the network described above. We consider a *subsumption from node iota to node ξ along link δ*. ι, ξ, and δ are called the *input node*, the *target node*, and the *axis*, respectively, of this subsumption. The joint J of ξ involving δ is called the *target joint*. This subsumption extends the scopes of the origins of the probes on δ directed towards ξ. It proceeds as follows.

First, the set Π of the probes on δ towards ξ is detached from δ, and δ is shifted from J to another joint J', as illustrated in Figure 5. J' is a copy of J and is on a node ξ' which is a copy of ξ. J' and ξ' may be created here and now, but may also have been made in a previous subsumption, as mentioned below. Below we proceed so as to make $\xi_0 = \xi_1 \cup \xi' \wedge \xi_1 \cap \xi' = \emptyset$

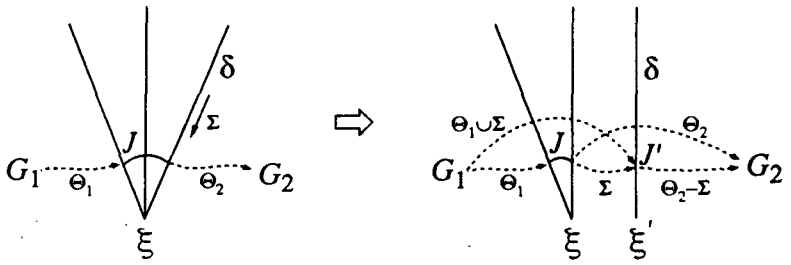

Figure 5: Shifting of Link and Augmentation of Foldability

true, where ξ_0 and ξ_1 stand for ξ before and after this subsumption, respectively.

A joint may be *foldable* to another joint by a set of origins of probes. Each joint involved here, called a *foldable joint*, is one obtained by copying zero or more times a multiple joint in the initial state of computation. Typically, a foldable joint is one involving links in the predicate domain of a defined predicate. No joint just created is foldable to any joint. For any joint G and set Θ of nodes, there is at most one joint H such that G is foldable to H by Θ.

Let Σ be the set of origins of the probes in Π. If J is foldable, then for each joint G the foldability relation extends in the following way, as illustrated in Figure 5, where the foldability relation is depicted by dashed arrows.

- J is foldable to J' by Σ.
- If G is foldable to J by Θ, then G is foldable to J' by $\Theta \cup \Sigma$.
- If J is foldable to G by Θ such that $\Theta \supset \Sigma$, then J' is foldable to G by $\Theta - \Sigma$.

If there has already been a joint to which J is foldable by Σ, then J' is that joint, ξ' is the node on J', J' becomes a total multiple joint, and the foldability relation remains unchanged. Otherwise, J' and ξ' are newly created, δ dominates ξ', and the foldability relation is augmented. We call the former case *folding*, and the latter *unfolding*.

If α is a proper atomic constraint or an argument of a proper atomic constraint, then $\hat{\alpha}$ stands for the set whose elements are this proper atomic constraint and its arguments; otherwise $\hat{\alpha} = \{\alpha\}$.

In the case of unfolding, each node ν in $\hat{\xi}$ is copied to ν', and each link σ ($\sigma \neq \delta$) connecting ν and η is copied to σ' connecting ν' and some node η'. η' is the copy of η if $\eta \in \hat{\xi}$ and $\eta' = \eta$ otherwise. Relevant Joints are copied accordingly so as to preserve the instantiation possibilities of the network.

There are two cases, *splitting* and *non-splitting*, about how to create σ'. In the former, it is guaranteed that no layer of the splitting domain including σ before the copy overlaps with both ν and ν' after the copy. Such a guarantee is obtained iff $\sigma = R \cup R^{-1}$ for some bijection R or (inclusive) δ and σ belong to the same splitting domain. There is no such guarantee in the non-splitting case.

In the splitting case, as is illustrated in Figure 6, the η-domains of σ and σ' are disjoint when $\eta' = \eta$.

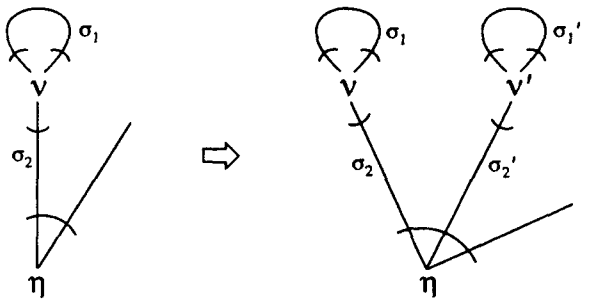

Figure 6: Copy of Links (Splitting)

In the non-splitting case, as is illustrated in Figure 7, if σ was a loop, ν and ν' is connected by an additional link representing a relation pertaining to the layers overlapping both ν and ν'. Further if σ was involved in a multiple joint of η, then a subsumption along σ to η must be done before creating σ'; otherwise the right instantiation possibilities cannot be represented.

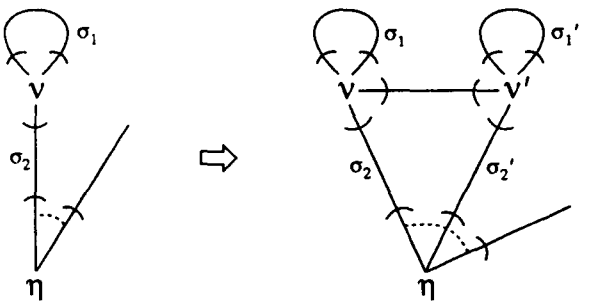

Figure 7: Copy of Links (Non-Splitting)

In both splitting and non-splitting cases, the probes that ν had, if any, are deleted, and ν and ν' are licensed to generate new probes. Then every remaining probe on σ is copied to a probe on σ', towards ν', and the same origin. Further, each probe in Π is advanced through ξ' onto every link τ ($\neq \delta$) connected with ξ' so that ξ' should be behind the probe. If there is another probe on τ towards ξ' and with the same origin, then both probes are deleted.

Finally, in both folding and unfolding, if δ dominated ξ before this subsumption, ξ is deleted because it has become the empty set now. This deletion propagates across links and nodes until possibly non-empty sets

are encountered: that is, until you come across partial or multiple joints of remaining nodes.[1] Now the subsumption is done.

To properly split splitting domains, we must augment this subsumption procedure so that a probe may carry, instead of origin, some information about which layers of the relevant splitting domain are involved in the node behind the probe. Such probes are transmitted from proper atomic constraints to their arguments and vice versa. A link is deleted if it contains two probes with opposite directions and associated with disjoint sets of layers. Further details are omitted due to the space limitation.

So far we have discussed subsumption in general. Below we describe the particularities of subsumptions along equalities and subsumptions along inference links.

A subsumption along an equality is triggered by a *dependency* between arguments. We say that there is a dependency between two arguments, when they *compete* with each other and are connected by a *dependency path*. Nodes α and β compete with each other when they are the first arguments of

- two bindings (as in ξ=f($\bullet$) and η=g($\bullet$)),
- a binding and a feature specification, or
- two feature specifications with the same feature.

A dependency path connecting α and β is a sequence $\delta_1\delta_2\cdots\delta_n$ of *strong equalities* such that the endpoints of δ_i are α_{i-1} and α_i $(1 \leq i \leq n)$, δ_i and δ_{i+1} are involved in different joints of α_i one of which is total $(1 \leq i < n)$, $\alpha_0 = \alpha$ and $\alpha_n = \beta$. An equality is strong when it belongs to a clause or the predicate domain of a defined predicate, or when a subsumption has taken place along that equality.

A probe π on an equality δ might trigger a subsumption to advance π, when there is a dependency between the origin α of π and another node β and δ is included in a dependency path connecting α and β.

Suppose the scope of α includes another node β competing with α. If the proper atomic constraints A and B, each involving α and β as the first argument, respectively, are connected by an inference link δ, then δ *absorbs* B, as shown in Figure 8. That is, the joint

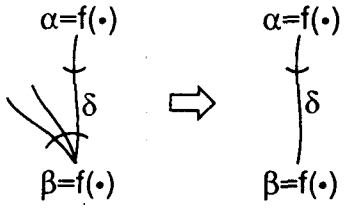

Figure 8: Absorption by Link

of B involving δ is modified so that β dominates β, because A has turned out to subsume B. Any other inference link involved in this joint is deleted, because it has turned out to be the empty set. Of course each equality accompanying δ must absorb its endnode in B at the same time. If there is no inference link between A and B, then B is deleted. Deletions of links and nodes propagate so long as the empty set is encountered, as said before.

A subsumption along an inference link may be triggered by *cost* assigned to the input node. Each literal in a clause may be assigned a cost. Similarly to the *assumability cost* of Hobbs et al. [5], the cost of a literal corresponds to the difficulty to abductively assume its negation. For instance, if you want to assume atomic constraint α by using a clause backwards whereas the cost of the literal $\neg\alpha$ in this clause is not zero, then you are to do something in order to cancel the cost. In this sense, an atomic constraint with a cost is regarded as a *goal* to achieve, and the abductive usage of the clause which gives rise to the goal is regarded as the *motivation* to set up that goal. A cost may be canceled by making the atomic constraint subsume another which is more readily believable. That is, a goal is fulfilled when it is established by some reason other than its motivation.

The input node of a subsumption along an inference link is the goal atomic constraint in the rest of the paper.[2] Such a subsumption eliminates the cost if the target node has been derived from the top clause without recourse to that very subsumption. Otherwise the cost is inherited into the clause which contains the output node. In a Horn clause normally used with all the atomic constraints therein being true, the head literal inherits the cost from a body atomic constraint, and the body atomic constraints inherit the cost from the head literal. We neglect the cost inheritance among body atomic constraints.

3.2 Heuristics

Subsumptions along equalities and those along inference links both encompass top-down and bottom-up information flow. Some heuristics are necessary to control such an otherwise promiscuous system of computation so that more relevant pieces of information should be exploited with greater preferences.

Each heuristic for a subsumption along an equality is that one of the following conditions raises the preference of such a subsumption.

(H1) The origin of a probe on the axis is close to (typically included in) the top clause or is a constant.

(H2) A dependency path involving the axis and connecting an argument with the origin of a probe on the axis is short.

Both these conditions are regarded as indicating that the transmitted information (about the origin) is highly relevant to the destination of this transmission. In this connection, a subsumption along an equality is unlikely to happen if the axis belongs to the predicate domain of a free predicate and the target joint is partial, since the conveyed information would not be very relevant to the target node.

[1] This combination of copy and deletion is vacuous and thus may be omitted in actual implementation for the unfolding cases. The deletion of probes in the splitting case may also be avoided in such a situation.

[2] Subsumptions for checking consistency need not be triggered by cost.

As for subsumptions along inference links, the following conditions each raise the preference.

(H3) Corresponding arguments of the input node and the target node are connected via short dependency paths with the same node. (That is, those arguments are 'shared.')

(H4) The target node has already been derived from the top clause.

(H3) raises the possibility for instances of the two arguments to coincide in the domain of interpretation. (H3) amounts to a generalization (or relaxation) of the condition on which an inference link absorbs one of its endnodes. (H4) guarantees that the subsumption in question will lead to an immediate elimination of the cost of the input node. Probably (H4) could be relaxed to be a graded condition.

4 Parsing

Let us consider a simple case of context-free parsing based on the following grammar.

$$P \rightarrow a$$
$$P \rightarrow P\,P$$

A parsing based on this grammar is formulated by the program as follows.

```
(5) true -p(A0,B) -A0=[a|A1] -A1=[a|A2] ···;
    (Φ) p([a|X],X);
    (Ψ) p(X,Z) -p(X,Y) -p(Y,Z);
```

Depicted in Figure 9 are the four types of clauses cre-

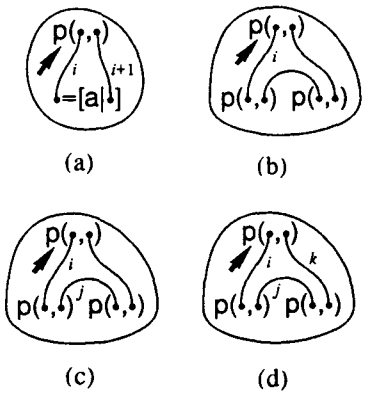

Figure 9: Clauses Produced through Parsing

ated by this parsing process. A •=[a|•] is a shorthand representation for a •=[•|•] plus an equality between the second argument and (the argument bound by) a. (a) is a copy of clause Φ in (5), and the other clauses are copies of Ψ. A label i of a link means that the relevant part of the network is in the scope of argument A_i. The reason why only these types of clauses are generated is that in this case every dependency arises between a •=[a|•] in the top clause and another •=[a|•] somewhere else and the first argument of the former is the origin of the subsumptions to resolve that dependency.

A strict proof will be obtained by mathematical induction. Since the number of these clauses is $O(n^3)$ due to (d) and each of them may be generated in a constant time, the time complexity of the entire parsing is $O(n^3)$, where n is the sentence length. Each clause is guaranteed to be generated in a constant time, because each foldability test can be performed in a constant time, as discussed later. By employing a general optimization technique, we can eliminate the clauses of type (d), so that the space complexity is reduced to $O(n^2)$. Thus, our general control scheme naturally gives rise to standard parsing procedures such as Earley's algorithm and chart parsing.

(5) is graphically represented as Figure 10. We

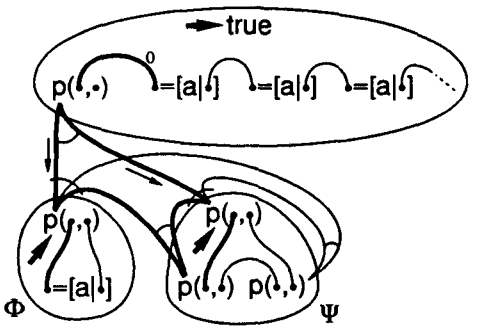

Figure 10: Parsing (1)

omit the links involved in the predicate domain of a free predicate, until they are modified as in Figure 8. Thus no links among •=[a|•]s are shown in Figure 10. Here is a dependency between the first •=[a|•] in the top clause and the •=[a|•] in Φ, as indicated by the dependency paths, which consist of thick links. To let information flow from the top clause following the above heuristic (H1), we are to do the two subsumptions indicated by the two thin arrows.

Those subsumptions copy Φ to Φ_1 and Ψ to Ψ_1, resulting in Figure 11. For expository convenience, we

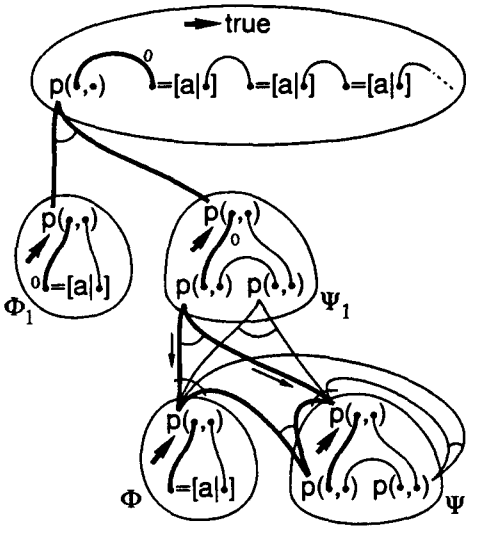

Figure 11: Parsing (2)

assume here without loss of generality that copying of a clause produces a separate clause rather than one sharing atomic constraints with the original clause. Note that the first argument of the •=[a|•] in Φ_1 is subsumed by A_0.

Computation goes on into the same direction, and the two subsumptions are to happen as shown in Figure 11. Folding takes place this time, and the result is to shift the two inference links upwards, as in Figure 12. Now the first •=[a|•] in the top clause dominates

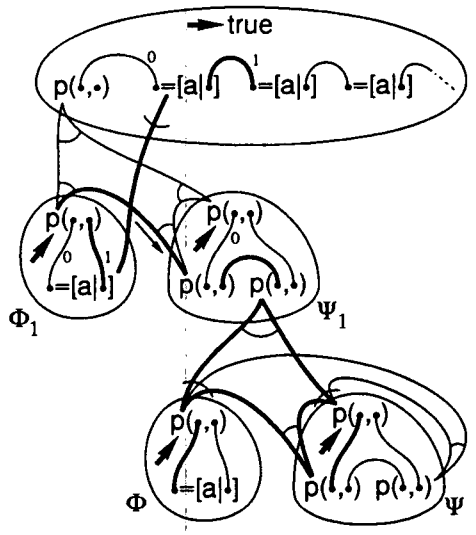

Figure 12: Parsing (3)

the •=[a|•] in Φ_1 as indicated by the inference link between them, because, as indicated by number 0 in Φ_1, the first argument of the former is within the scope of the first argument of the latter. Now the equality in the right-hand side of Φ_1 is within the scope of A_1, as indicated in the figure. This subsumption also engenders a new set of dependencies between the first argument of the second •=[a|•] in the top clause and that of •=[a|•] in Φ, as indicated again by thick links in Figure 12. By executing the indicated subsumption following (H1), Ψ_1 is copied to Ψ_2, so that we obtain Figure 13. Further advancing subsumptions as shown there, we get Figure 14. Computation goes on in the similar way.

As mentioned above, we are able to assume that each foldability test is performed in a constant time. This assumption is justified by, for instance, sorting the foldability information from each joint in the chronical order of the first subsumption which advanced probes with the relevant origin. In the present parsing example, this order happens to be the increasing order of the suffix i of A_i.

It is straightforward to integrate such a phrase-structure parsing with computation on internal structures of grammatical categories represented in terms of feature bundles, for instance. See [2, 3] for further details in this regard. Note that the above derivation of the parsing process is more general than the parsing-as-deduction approaches [6, 7], because it is free from stipulation of the left-to-right and to-down processing direction and also from task-dependency with regard to parsing or context-free grammar.

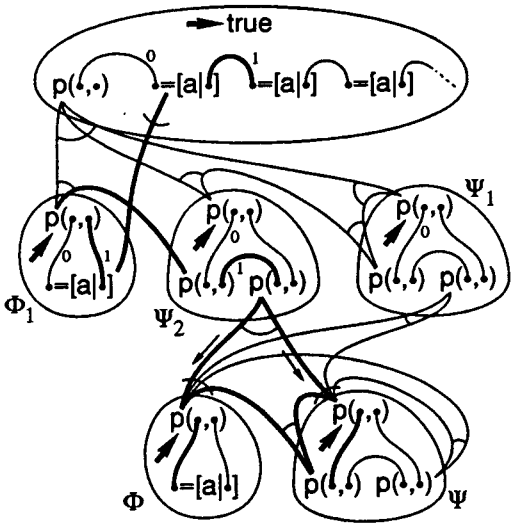

Figure 13: Parsing (4)

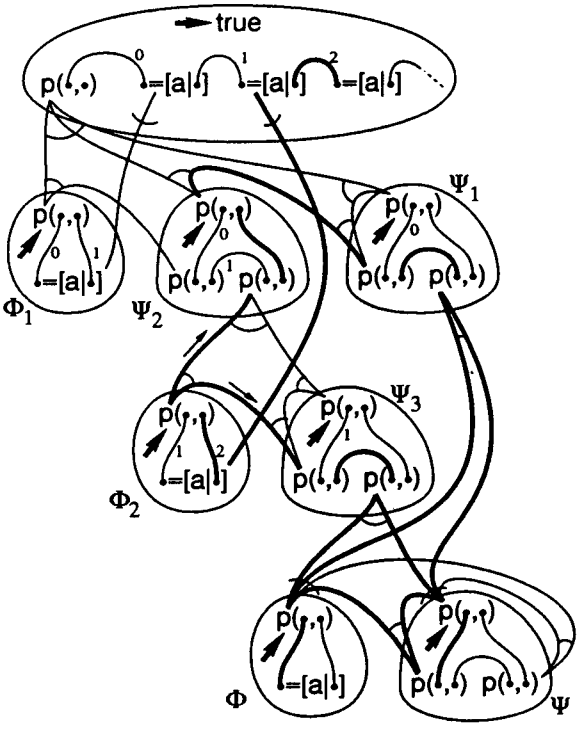

Figure 14: Parsing (5)

5 Generation

Here we consider how to verbalize the following semantic content in English.

S⊨⟨⟨laughed, kim⟩⟩

This means that Kim laughed, based on Situation Theory [1]. That is, in some situation S there is an event which is of the sort laughed and whose agent is kim. So a sentence we might want to generate is 'Kim laughed.' S may be interpreted as, for instance, the speaker's model of the hearer's model of the world. A state of affairs ⟨⟨laughed, kim⟩⟩ will be regarded as variable L1 constrained by two feature specifications rel(L1,laughed) and agt(L1,kim).

The initial state of computation could be formulated in terms of a program including the following clauses, among much more others.

```
(A) true -s(SEM,W0,W1) -S⊨SEM -say(W0)
         -S⊨L1$, -rel(L1,laughed)$
         -agt(L1,kim)$ ···;
(B) s(SEM,X,Z) -np(SBJSEM,X,Y)
         vp(SEM,SBJSEM,Y,Z);
(C) np(kim,X,Y) -X=['kim'|Y]$;
(D) vp(L,AGT,X,Y) -X=['laughed'|Y]$
         -rel(L,laughed) -agt(L,AGT);
```

say(W0) means that the utterance beginning at W0 should be actually uttered. S⊨SEM and S⊨L1 seperately exist in (A), because the next utterance need not directly refer to L1. For instance, one can mean that Kim laughed by saying 'Do you know that Kim laughed?' instead of just 'Kim laughed,' or by doing something other than utterance. One might even just give up the goal and say something quite different.

A '$' attached to an atomic constraint represents a cost, so that the atomic constraint is a goal. The three goals in (A) altogether amount to a macroscopic goal to make a state of affairs ⟨⟨laughed, kim⟩⟩ hold in situation S.

What we would like to demonstrate below is again that the control heuristics described in Section 3 tend to trigger the right operations depending upon the computational context, provided that the current goal is to be reached by some linguistic means; that is, by eventually uttering some sentence. Below we pay attention to only one maximal consistent structure of the sentence at a time just for the sake of simplicity, but the actual generation process may involve OR-parallel computation similar to that in parsing of the previous section.

Figure 15 graphically represents clauses (A) and (C). A proper atomic constraint with a binary predicate, possibly together with equalities involving the two arguments, is represented here as an arrow from (an argument equalized with) the first argument to (an argument equalized with) the second argument. Links in predicate domains are selectively displayed for expository simplicity.

The most probable operations to take place here are subsumptions involving one of these three goals. There should be innumerable combinations for such

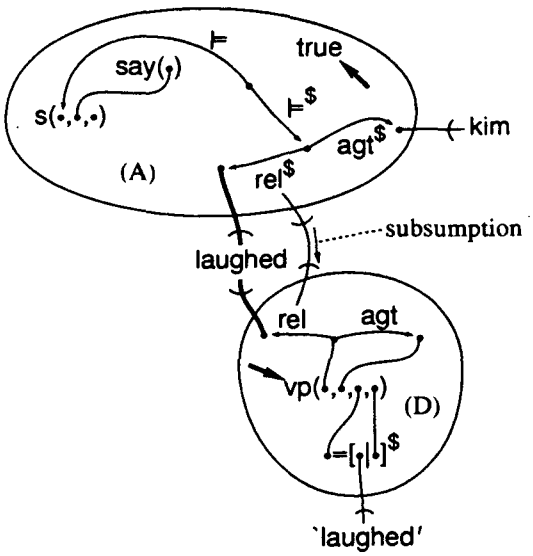

Figure 15: Generation (1)

subsumptions, because the speaker's lexicon must include a large number of atomic constraints of the form •⊨•, rel(•,•) and agt(•,•), even though subsumptions with extralinguistic parts of the constraints are excluded due to the above provision that the current goal is to be fulfilled by linguistic means.

However, two of such subsumptions are preferred to the others. One is the subsumption concerning the two •⊨•s in (A), and the other is from the rel(•,•) in (A) to that in (D). In both cases, the two atomic constraints share the same argument for the same argument place, which raises the preference due to (H3). Let us tentatively choose just the latter subsumption in this particular presentation. No big difference would follow in the long run, even if the former subsumption or both were chosen instead.

By the subsumption concerning the two rel(•,•)s, we obtain the structure shown in Figure 16. We have

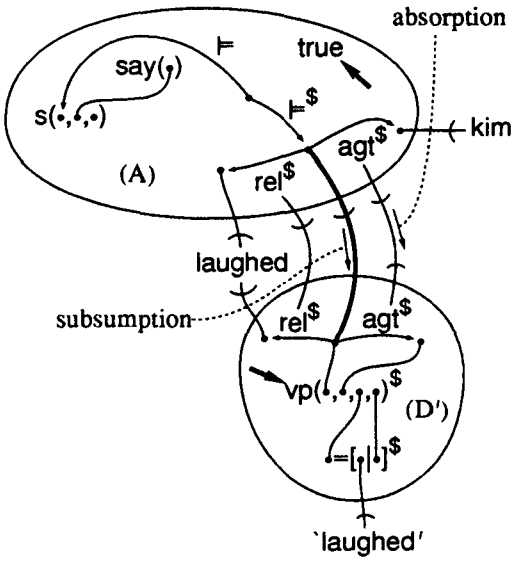

Figure 16: Generation (2)

copied clause (D) to (D'), because the `rel(•,•)` in (A) is a goal. Now in Figure 16, `vp(•,•,•,•)` in (D') is a goal, by inheriting the cost from `rel(•,•)` of (A). The cost of `•=[•|•]` in (D') is inherent, as indicated in (D). Now the most probable next computation is the sequence of subsumptions along the thick link(s) constituting a dependency path. Following the heuristic (H1), those subsumptions propagate from the top clause. After that, the inference link between the two `agt(•,•)`s absorbs the one in (B).

This gives us Figure 17. (D') has not been duplicated here, because the above subsumptions did not actually duplicate any clause. In this context, the subsumption concerning the two `vp(•,•,•,•)`s is possible, since the one in (D') is a goal. Due to (H3), this subsumption is more preferable than the others concerning two `vp(•,•,•,•)`s, because their first arguments are both connected to `kim` (that is, the first argument of `•=kim`) via short dependency paths. As a result, (B) is copied to (B') and the `vp(•,•,•,•)` in (B') is dominated by that in (D'), as in Figure 18.

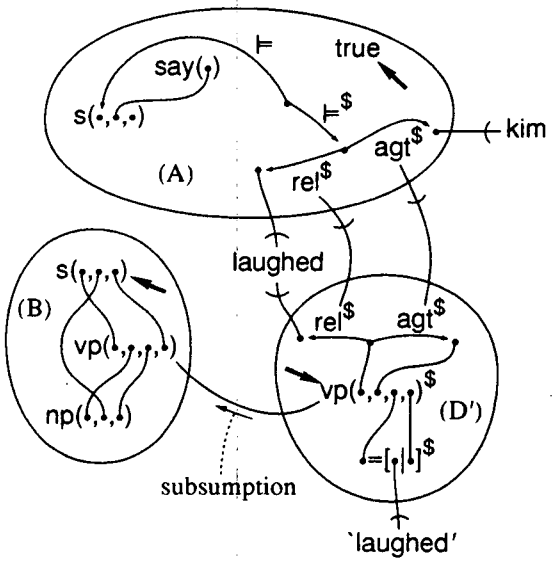

Figure 17: Generation (3)

Now that `s(•,•,•)` in (B') is a new goal, it is caused to subsume another `s(•,•,•)` in (A). According to (H4), this subsumption is particularly preferable because (A) is the top clause. On the other hand, the subsumption from the first argument of `np(•,•,•)` in (B') to the first argument of `np(•,•,•)` in (C) could take place here, to resolve the cyclic dependency about `kim` referred to from (A) and (C). This subsumption is the most probable operation concerning this dependency in this context, because it is along the shortest relevant dependency path. We assume that the direction of this subsumption is downwards, as indicated in Figure 18. It will be the same in the long run if it were in the opposite direction.

The mentioned computation in Figure 18 takes us to Figure 19. We have a new top clause (A'), which shares most part of itself with (A), except the copy of `s(•,•,•)`. Some of the previous goals have disappeared due to the subsumption concerning `s(•,•,•)`s. Now the remaining goals are `•=[•|•]`s in (C') and (D')

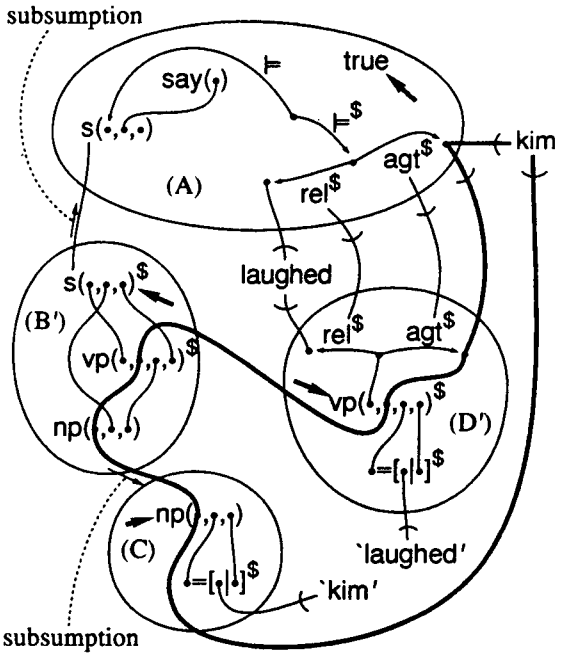

Figure 18: Generation (4)

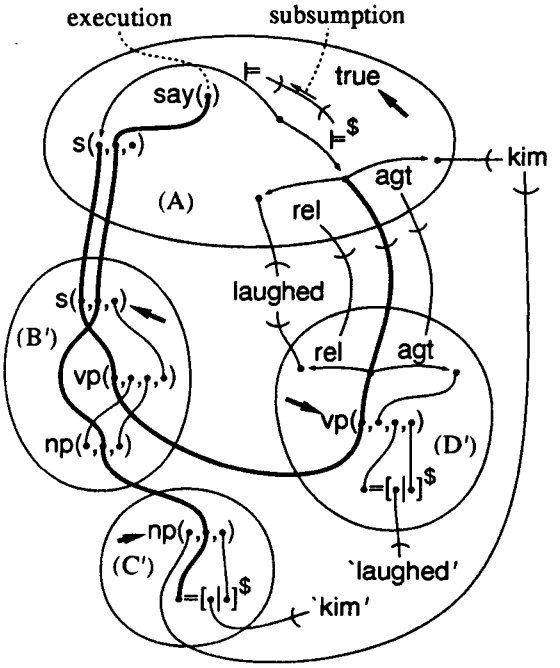

Figure 19: Generation (5)

and the •⊨• in the intersection of (A) and (C'). We might do a subsumption concerning the two •⊨•s, because they share both the arguments. This subsumption could have happened earlier, of course, particularly ever since both arguments came to be shared in Figure 16 via (B) and (D'). As mentioned before, however, it would have caused no essential difference eventually. At the same time we could execute the procedure say(•) to realize the goal •=[•|•] in (C'). It is reasonable to assume that this computation is triggered by the fact that the argument of say(•) subsumes the first argument of this •=[•|•]. This heuristic for firing procedures looks generally applicable not only to utterance but also to every other output procedure.

Thus we move to a new computational context in Figure 20. The execution of say(•) has created a new

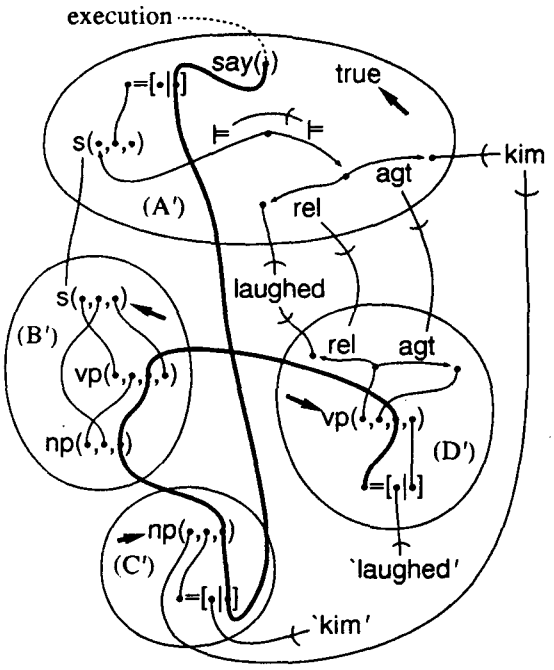

Figure 20: Generation (6)

•=[•|•], so that 'Kim' has been spoken aloud. This •=[•|•] dominates the •=[•|•] in (C'), as indicated by the thick link. Generation of 'Kim laughed' completes if say(•) is executed one step further.

Note that this generation process captures the bottom-up feature of semantic head-driven generation [8], especially when we move from Figure 15 through Figure 18. The subsumption concerning the arguments of np(•,•,•)s happening between Figure 18 and Figure 19 captures the top-down aspect as well.

6 Concluding Remarks

We have introduced a set of general heuristics for controlling symbolic computation on logic constraints, and demonstrated that sentence parsing and generation are attributed to these heuristics. In the above presentation, parsing is for the most part based on truth maintenance (resolution of dependencies among arguments) controlled by heuristics (H1) and (H2), whereas generation is more dependent on goal satisfaction controlled by (H3) and (H4). In more realistic cases, however, both processes would involve both kinds of computation in a more intertwined way.

A related nice feature of our framework is that, in principle, all the types of constraints — syntactic, semantic, pragmatic and extralinguistic — are treated uniformly and integrated naturally, though a really efficient implementation of such an integrated system requires further research. In this connection, we have undertaken to study how to implement the above heuristics in a more principled and flexible way, based on a notion of *potential energy* [4], but the present paper lacks the space for discussing the details.

In this paper we have discussed only task-independent aspects of control heuristics. Our conjecture is that we will be able to dispense with domain-dependent and task-dependent control heuristics altogether. The domain/task-dependent characteristics of information processing will be captured in terms of the assignment of energy functions to the relevant constraints. The resulting system will still be free from stipulation of the directions of information flow, allowing multi-directional information processing, since neither the symbolic component nor the analog component (that is, energy) of the constraint refers explicitly to information flow.

References

[1] Barwise, J. (1990) *The Situation in Logic*, CSLI Lecture Notes No. 17.

[2] Hasida, K. (1990) 'Sentence Processing as Constraint Transformation,' *Proceedings of ECAI '90.*

[3] Hasida, K. and Tsuda, H. (1991) 'Parsing without Parser,' *International Workshop on Parsing Technologies*, pp. 1-10, Cancun.

[4] Hasida, K. (in preparation) *Potential Energy of Combinatorial Constraints.*

[5] Hobbs, J., Stickel, M., Martin, P., and Edwards, D. (1988) 'Interpretation as Abduction,' *Proceedings of the 26th Annual Meeting of ACL*, pp. 95-103.

[6] Pereira, F. C. N. and Warren, D. H. D. (1983) 'Parsing as Deduction,' *Proceedings of the 21st Annual Meeting of ACL*, pp. 137-144.

[7] Shieber, S. M. (1988) 'A Uniform Architecture for Parsing and Generation,' *Proceedings of the 12th COLING*, pp. 614-619.

[8] Shieber, S. M., van Noord, G., and Moore, R. C. (1989) 'A Semantic-Head-Driven Generation Algorithm for Unification-Based Formalisms,' *Proceedings of the 27th Annual Meeting of ACL*, pp. 7-17.