Emergent Parsing and Generation with Generalized Chart

HASIDA Kôiti
Electrotechnical Laboratory
1-1-4 Umezono, Tukuba, Ibaraki 305, Japan
E-mail: hasida@etl.go.jp

Abstract

A new, flexible inference method for Horn logic program is proposed. It is also a drastic generalization of chart parsing, partial instantiation of clauses in a program roughly corresponding to arcs in a chart. Chart-like parsing and semantic-head-driven generation emerge from this method. With a parsimonious instantiation scheme for ambiguity packing, the parsing complexity reduces to that of standard chart-based algorithms.

1 Introduction

Language use involves very complex interactions among very diverse types of information, not only syntactic one but also semantic, pragmatic, and so forth. It is hence inappropriate to assume any specific algorithm for syntactic parsing or generation, which prescribes particular processing directions (such as left-to-right, top-down and bottom-up) and is biased for specific types of domain knowledge (such as a context-free grammar). To account for the whole language use, we will have to put many such algorithms together, ending up with an intractably complicated model.

A better strategy is to postulate no specific algorithms for parsing or generation or any particular task, but instead a single uniform computational method from which emerge various types of computation including parsing and generation depending upon various computational contexts.

For example, Earley deduction (Pereira & Warren, 1983) is a general procedure for dealing with Horn clauses which gives rise to Earley-like parsing when given a context-free grammar and a word string as the input. Shieber (1988) has generalized this method so as to adapt to sentence generation as well. Those methods fail to give rise to efficient computation for a wide variety of contexts, however, because they prescribe processing directions such as left-to-right for parsing and bottom-up for generation. They also lack a general way of efficient ambiguity packing unlimited to context-free grammars. Hasida (1994a) proposes a more general inference method for clausal form logic programs which accounts for efficient parsing and generation as emergent phenomena. This method prescribes no fixed processing directions, and the way it packs ambiguity is not specific to context-free grammars. However, it is rather complicated and has greater computational complexity than standard algorithms do.

In this paper we propose another inference method for Horn logic programs based on Hasida (1994a), and show that efficient parsing and generation emerge from it. Like that of Hasida (1994a), this method is totally constraint-based in the sense that it presupposes no fixed directions of information flow, but it is more efficient owing to a parsimonious method of instantiation. In Section 2 we define this inference method, which is a generalization of chart parsing, and may be also thought of as a connection method or a sort of program transformation. Section 3 illustrates how efficient parsing and generation emerge from this method without any procedural stipulation specific to the task and the domain knowledge (syntactic constraints). Section 4 introduces a parsimonious instantiation method for ambiguity packing. We will show that owing to this method the efficiency reaches that of the standard algorithms with regard to context-free parsing. Section 5 concludes the paper by touching upon further research directions.

2 Partial Instantiation

A constraint is represented in terms of a Horn clause program such as below.

(a) –p(A,B) –A=a(C).
(b) p(X,Y) –X=a(Y).
(c) p(U,W) –p(U,V) –p(V,W).

Names beginning with capital letters represent variables, and the other names predicates and functors. The atomic formulae following the minus sign are negative (body) literals, and the others are positive (head) literals. A clause without a positive literal is called a **top clause**, whose negation represents a goal (top-level hypothesis), which corresponds to a query in Prolog. For instance, top clause (a) in the above program is regarded as goal $\exists A, B, C\{p(A, B) \wedge A = a(C)\}$. In general, there may be several top clauses. The purpose of computation is to tell whether any goal is satisfiable, and if so obtain an answer substitution for the terms (variables) in a satisfiable goal. We consider the minimal Herbrand models as usual. So the set of answer substitutions for A in the above program is {a(B), a(a(B)), a(a(a(B))), ···}.

A graphical representation of this program is shown in Figure 1. Here each clause is the set of the literals enclosed in a dim closed curve. A link connecting arguments in a clause is the term (variable) filling in

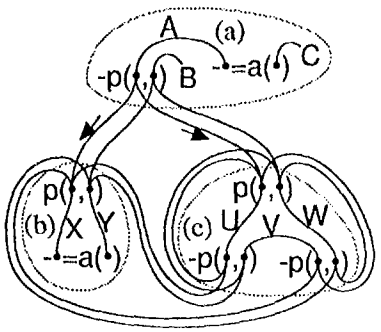

Figure 1: A graphical representation of a program.

those arguments. (It is a hyperlink when there are more than two arguments.) A transclausal link represents the unifiability between two corresponding arguments of two unifiable literals. (Neglect the arrows for a while.)

A **hypothesis** is a conjunction of atomic formulas and bindings. The premise of a clause (i.e., the conjunction of the atomic formulas and bindings which appear as negative literals) is a hypothesis. An **expansion** for a hypothesis is a way of combining (instances of) clauses by resolutions so as to translate the hypothesis to another hypothesis involving bindings only. We will refer to an expansion by the sequence of clauses in the order of leftmost application of resolution using their instances.[1] In the above program, for example, expansion ⟨c, b, b⟩ translates the top-level hypothesis s(A,B) ∧ A=a(C) to a hypothesis A=a(C) ∧ C=a(B). An expansion of a clause is an expansion of its premise. We will simply say 'an expansion' to mean an expansion of the top-level hypothesis. A program represents a set of expansions, and the computation as discussed later is to transform it so as to figure out correct hypotheses while discarding the wrong expansions (those entailing wrong hypotheses).

We say that there is a **dependency** between two terms when those terms are unified in some expansion, and the sequence of terms (including them) mediating this unification is called the **dependency path** of this dependency. In Figure 1, for instance, the dependency between A and X is mediated by dependency path A·X, A·U·X, A·U·U·X, and so on. There is a dependency between C and B, among others, because of the unifiability of the two −•=a(•)s, though this unifiability is not explicitly shown in Figure 1. We say a dependency between two terms is **consistent** when they are not bound by inconsistent bindings. All the dependencies in Figure 1 are consistent.

A **solution** of the program is an expansion in which every dependency is consistent. So the computation we propose in this paper is to transform the given program in such a way that every dependency be consistent. To figure out dependencies, we use a symbolic operation called **subsumption**, and **delete** the parts of the program which contributes to wrong expansions only. For example, suppose there is an inconsistent dependency between terms α and β. We create an instance β' of β by subsumption operations to be discussed shortly, so that every expansion containing an instance of β' contains an instance of a dependency path between α and β. We can then delete the clause containing β' and probably some more parts of the program without affecting the declarative semantics of the program. Below we will define a computational procedure in such a way that the set of the possible expansions eventually represent the set of all the solutions.

Subsumption operation is to create **subsumption relationship**. We regard each part (clause, atomic formula, term, etc.) of a program as the set of its instances, and say that a part ξ of the program **subsumes** another part η to mean that we *explicitly know* that $\xi \supseteq \eta$. We consider that a link is subsumed by δ if and only if one of the terms it links is subsumed by δ. We say term δ is an **origin** of η when η is subsumed by δ. In this paper we consider that every origin is a bound term (the term filling in the first argument of a binding). Let us say that two clauses (or two literals) are **equivalent** when they are of the same form and for each pair of corresponding terms the two terms have the same set of origins.

Subsumption relation restricts the possibility of expansions so that if term η is subsumed by another term δ, then every expansion containing an instance of η must also contain an instance of δ. Subsumption relation is useful to encode structure sharing among expansions. In subsumption-based approaches, a term may subsume several non-unifiable terms and thus the first term is shared among the latters. However, that is impossible in unification-based approaches, where different expansions cannot share the same instance of a term or a clause.

A **partially instantiated clause** is a clause some of whose terms is subsumed by another term in possibly another clause. For instance,

(1) $a(\overline{A_i},Z)\ -b(\overline{A_i},\overline{A_j})\ -c(\overline{A_j},Z).$

is a partial instantiation of the following clause:

(2) a(X,Z) −b(X,Y) −c(Y,Z).

$\overline{A}$ represents a term subsumed by term A.[2] Hereafter we say just 'clause' to refer to both uninstantiated clauses and partially instantiated clauses.

A program consisting of such clauses is a generalization of a chart (Kay, 1980). A chart is a graph whose nodes denote positions between words in a sentence and whose arcs are regarded as context-free rules each instantiated partially with respect to at most two such positions. For instance, an active arc from node i to node j labelled with $[A \rightarrow \bullet\, B \bullet\, C]$ is an instance of rule $A \rightarrow B\,C$ with both sides of B instantiated by positions i and j. This arc approximately corresponds to (1).[3]

[1] Here we mention the order among the literals in a clause just for explanatory convenience. This order is not significant in the computation discussed later.

[2] This notation is problematic because it is unclear whether two occurrences of $\overline{A}$ in a clause denote the same term. In this paper they always do.

[3] However, an arc in a chart does not precisely correspond to a partially instantiated clause derived from a program encoding

A subsumption operation is to extend subsumption relation by possibly creating a partially instantiated clause. A subsumption operation is characterized by the **origin**, the **source**, and the **target**. The origin (let it be δ) is a bound term. The source (σ) and the target (τ) are arguments. σ should already be subsumed by the origin, but τ should not be so. They should be connected through a transclausal link λ. Let the literal containing σ be ρ. Also let the literal containing τ be π, and the clause containing them be Φ. There are two cases for subsumption, and in both cases σ comes to be linked through λ with an argument which is an instance of τ subsumed by δ.

In the first case of subsumption operation, which we call **unfolding**, a partial instantiation Φ' of Φ is created. They are equivalent except that the instance τ' of τ in Φ' is subsumed by δ. After the unfolding, σ is linked through λ to the instance of τ in Φ' instead of the original τ, and accordingly ρ is linked to the instance of π in Φ'. Let τ'' be τ after the unfolding. Then $\tau' \cup \tau'' = \tau$, $\tau' \cap \tau'' = \emptyset$, and $\tau' = \tau \cap \sigma$ hold. This implies $\tau' \subseteq \delta$ and $\tau'' \cap \sigma = \emptyset$. So τ'' and σ are not unifiable.

For instance, the two subsumption operations indicated by the two arrows in Figure 1 are unfoldings. In either case, the origin and the source are both A. The target in the left is X and that in the right is U. We obtain the program in Figure 2 by these operations,

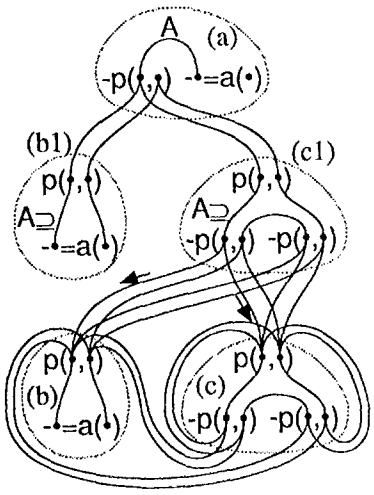

Figure 2: After subsumptions to X and U by A.

where partial instantiation (b1) and (c1) of (b) and (c) have been created, respectively.

In Figure 1, the subsumption operation through the (invisible) link connecting C and Y is not executable now, because the unification represented by this link presupposes the unification of A and X through the dependency paths A·X, A·U·X, A·U·U·X, and so on. That is, it is only when C subsumes an instance (let it be Y′) of Y that subsumption from C to Y′ is possible. (This subsumption is an unfolding without any copy, because then C automatically subsumes Y′.) Same for the subsumption in the opposite direction.

The second case of subsumption operation is called **folding**. It takes place when there is already a literal π' equivalent to π except that its argument τ' corresponding to τ is subsumed by δ. In this case, no new instance of clause is created, but instead link λ is switched so that it links σ with τ' and accordingly ρ is linked with π'. Let τ'' be τ after the unfolding. Then $\tau \cap \tau' = \emptyset$ both before and after the folding, and $\sigma \cap \tau$ is subtracted from τ and added to τ' by the folding. Folding is triggered when there exists literal π' as described above, and unfolding is executed otherwise. If there existed several such π's, folding takes place, creating as many instances of λ and connecting to those π's.

The two subsumption operations indicated in Figure 2 are foldings. Actually, in the left, the p(•,•) in (b1) and that in (b) are equivalent except that the first argument of the former is subsumed by A. So the link with the arrow and the parallel accompanying link are switched up to p(•,•) in (b1). Similarly for the right subsumption. Shown in Figure 3 is the result.

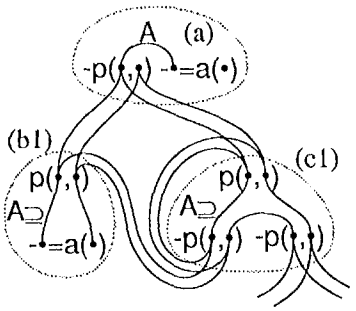

Figure 3: After foldings.

Note that the original program encodes a problem of partial parsing of a string beginning with "a" under the context-free grammar consisting of the following rules.

$$P \rightarrow \mathrm{a}$$
$$P \rightarrow P\ P$$

The result in Figure 3 encodes the infinitely many possible parses of this incomplete sentence. Note also that here the subsumption from C to the instance of Y in (b1) would be possible if C were bound. The next section contains relevant examples.

When a link is subsumed by two terms bound by two inconsistent bindings (such as •=a and •=b), then that link is **deleted**, surrounding clauses possibly being deleted if some of their atomic formulas are linked with no atomic formula any more.

For the sake of simplicity, we mainly consider **input-bound** programs in this paper. We say a program is input-bound when every dependency path between bound terms connects a term in a top clause and one in a non-top clause. The program in Figure 1 and the ones for parsing and generation in the following section are all input-bound programs. For input-bound

a context-free grammar in a standard way. See Section 4 for further discussion.

programs, we have only to consider subsumptions by terms in top clauses: input-driven computation. Also, in input-driven computation for input-bound programs we do not have to worry about duplications of origins by subsumptions.

Both subsumption and deletion preserve the declarative semantics of the program (the set of the solutions), though we skip a detailed proof due to the space limitation. So when they are not applicable any more, every expansion is a solution and vice versa. For input-bound programs, the input-driven computation always terminates within time polynomial as to the size of the program. This is because there are at most n^m partially instantiated clauses derived from a clause with m terms, where n is the size of the input (the number of bound terms in the top clause(s)), and accordingly there are polynomially many transclausal links. Obviously, partially instantiated clauses and new transclausal links are each created in constant time. It is also clear that each folding terminates in polynomial time.

3 Parsing and Generation

Here we show that chart-like parsing and semantic-head-driven generation emerge from the above computational method. We discuss examples of parsing and generation both on the basis of the following grammar.

(3) s(Sem,X,Z) –np(SbjSem,X,Y)
–vp(Sem,SbjSem,Y,Z).

(4) vp(Sem,SbjSem,X,Z)
–v(Sem,SbjSem,ObjSem,X,Y)
–np(ObjSem,Y,Z).

(5) np(Sem,X,Y) –Sem=tom –X="Tom"(Y).

(6) np(Sem,X,Y) –Sem=mary –X="Mary"(Y).

(7) v(Sem,Agt,Pat,X,Y)
–Sem=love(Agt,Pat) –X="loves"(Y).

Since we have already mentioned ambiguity packing in the previous section, below we do not explicitly deal with ambiguity but instead discuss just one sentence structure in both parsing and generation.

Let us first consider parsing of sentence 'Tom loves Mary'. The problem is encoded by the program in Figure 4. The input-driven computation proceeds as shown by the arrows, which represent subsumption operations taking place in the ordering indicated by the labelling numbers. A thick dependency path is processed by successive subsumptions with the same origin. The only subsumption operations executable in the initial situation is the one numbered 1 and after that the one numbered 2, along the thick path between A_0 and X in (5). As the result of these unfoldings, we obtain the following clauses.

(8) s(Sem,$\overline{A_0}$,Z) –np(SbjSem,$\overline{A_0}$,Y)
–vp(Sem,SbjSem,Y,Z).

(9) np(Sem,$\overline{A_0}$,$\overline{A_1}$) –Sem=tom –$\overline{A_0}$="Tom"($\overline{A_1}$).

Of course other partially instantiated clauses may be created here from definition clauses of s other than (3) and those of np other than (5), but we omit them here and concentrate on just one solution.

Now the copy of link with the arrow numbered 3 connected to (9) can mediate subsumption operations. So the subsumption operation indicated that arrow is triggered, though that does not duplicate (9) because A_1 already subsumes the target. The result is already reflected in (9). The subsequent subsumption operations numbered 4, 5, and 6 will yield the following clauses.

(10) s(Sem,$\overline{A_0}$,Z) –np(SbjSem,$\overline{A_0}$,$\overline{A_1}$)
–vp(Sem,SbjSem,$\overline{A_1}$,Z).

(11) vp(Sem,SbjSem,$\overline{A_1}$,Z)
–v(Sem,SbjSem,ObjSem,$\overline{A_1}$,Y)
–np(ObjSem,Y,Z).

(12) v(Sem,Agt,Pat,$\overline{A_1}$,$\overline{A_2}$) –Sem=love(Agt,Pat)
–$\overline{A_1}$="loves"($\overline{A_2}$).

Now the subsumption operations by A_2 are commenced, due to the creation of (12). Accordingly, the following clauses are created, and the parsing is finished.

(13) s(Sem,$\overline{A_0}$,$\overline{A_3}$) –np(SbjSem,$\overline{A_0}$,$\overline{A_1}$)
–vp(Sem,SbjSem,$\overline{A_1}$,$\overline{A_3}$).

(14) vp(Sem,SbjSem,$\overline{A_1}$,$\overline{A_3}$)
–v(Sem,SbjSem,ObjSem,$\overline{A_1}$,$\overline{A_2}$)
–np(ObjSem,$\overline{A_2}$,$\overline{A_3}$).

(15) np(Sem,$\overline{A_2}$,$\overline{A_3}$) –Sem=mary –$\overline{A_2}$="Mary"($\overline{A_3}$).

From the earlier discussion, in the case of context-free parsing the number of clauses created there is $O(n^M)$, where n is the number of the input words and M the maximum number of the occurrences of non-terminal symbols in a context-free rule. This is larger than the space complexity of the standard parsing algorithms, but later we will show how to improve the efficiency so as to be equivalent to the standard algorithms.

No particular order among the subsumption operations is prescribed in the above computation, and so it is not inherently limited to top-down or bottom-up. Note also that the left-to-right processing order among the input words is derived from the definition strong link, rather than stipulated as in Earley deduction, among others. We can account for island-driven parsing as well, by allowing links between bindings to trigger subsumptions more earlier.

Let us next take a look at sentence generation. Consider the program shown in Figure 5. Here the input is semantic structure love(tom,mary). Again the computational process is indicated by the numbered arrows. 6′ takes place after 5, but the order among 6, 7, and 6′ is arbitrary as long as 6 should be before 7. So the only possible subsumption operation in the beginning is the ones by Love, which go through the thick curve connecting Love and the X in (4). This creates the following clause, among others.

(16) v($\overline{\text{Love}}$,$\overline{\text{Tom}}$,$\overline{\text{Mary}}$,X,Y)
–$\overline{\text{Love}}$=love($\overline{\text{Tom}}$,$\overline{\text{Mary}}$) –X="loves"(Y).

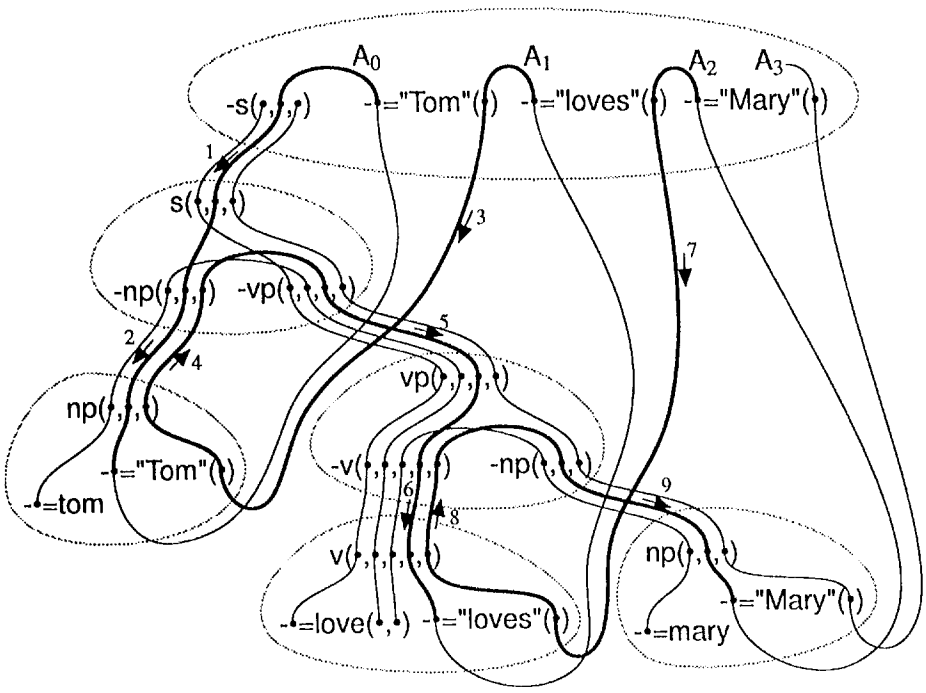

Figure 4: Parsing

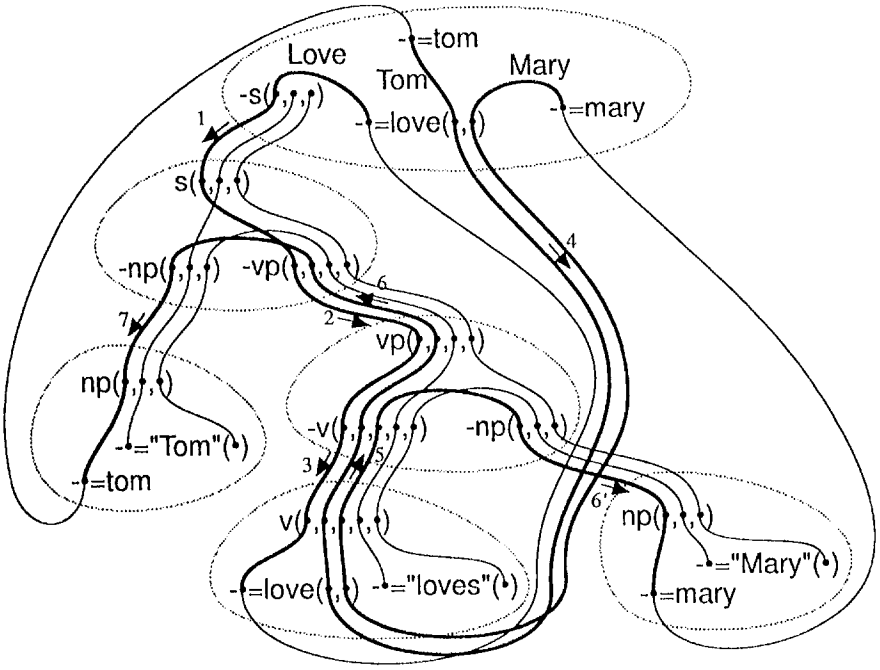

Figure 5: Generation

Now subsumption operations can go through the copies of the other two thick curves. So we are creating the following clauses, among others.

(17) s(Love,X,Z) −np(Tom,X,Y) −vp(Love,Tom,Y,Z).

(18) vp(Love,Tom,X,Z) −v(Love,Tom,Mary,X,Y) −np(Mary,Y,Z).

(19) np(Tom,X,Y) −Tom=tom −X="Tom"(Y).

(20) np(Mary,X,Y) −Mary=mary −X="Mary"(Y).

Note that this generation process amounts to a generalization of semantic-head-driven generation (Shieber, van Noord, & Moore, 1989). The order among the retrievals of semantic heads is the order of subsumption operations by different terms in the input semantic structure, just as with the processing order among words in the case of parsing.[4] Also as in the case of parsing, the computational complexity of such a generation is polynomial with respect to the size of the input semantic structure, provided that the program is input-bound and the computation is input-driven. Although the above example deals with only a single sentence structure, in general cases ambiguity packing naturally takes place just as with parsing of ambiguous sentences.

Under the restriction that the program be input-bound, the grammar cannot employ feature structures prevalent in the current linguistic theories, and also must be semantically monotonic (Shieber et al., 1989)[5] The proposed method can be generalized so as to remove this restriction, though the details do not fit in the allowed space. This generalization makes it possible to deal with feature structures and semantically non-monotonic grammars. Of course the computation is not any more generally guaranteed to terminate (because Horn programs can encode Turing machines), but our method still has a better termination property than more simplistic ones such as Prolog interpreter or Earley deduction. For instance, endless expansion of left recursion or SUBCAT list, which would happen in simple top-down computations, is avoided owing to folding.

4 Incremental Copy

The parsing process discussed above is computationally more complex than chart parsing. Here we improve our method by introducing a more efficient scheme for ambiguity packing and thus reduce the parsing complexity to that of chart parsing, which is $O(n^2)$ for space and $O(n^3)$ for time.

The present inefficiency is due to excessive multiplication of clauses: much more partially instantiated clauses are created than arcs in a chart. So let us suppose that a subsumption operation does not duplicate a whole clause but only some part of it, so that a clause is copied incrementally, as shown in Figure 6. We assume that a subsumption to an argument of a literal copies the term filling in that argument, the literal, and some other literals which mention that term, unless there have already been the terms and literals to be thus created. Subscript i of a literal indicates that it is created by the i-th subsumption operation.

We must ensure that this partial copying be semantically equivalent to the copying of whole clauses. That is a trivial business when there are just one or two literals in the original clause. The case where there are more than three literals reduces to the case where there are exactly three literals, by grouping several literals connected directly (through terms) and treat them as if they were one literal. So below let us consider the case where there are three literals in a clause.

A non-trivial check must be done in such a case as in the lower right of Figure 6. Here you must copy $-r(\bullet,\bullet)_2$ and $-q(\bullet,\bullet)_1$ but not $-q(\bullet,\bullet)$, because $-r(\bullet,\bullet)_2$ is **compatible** with $-q(\bullet,\bullet)_1$ but not with $-q(\bullet,\bullet)$. We say that a set of literals are compatible when there is an instance of the clause which involves an instance of each of those literals. Also, two literals are said to be **heterogeneous** when they have different originals in the original uninstantiated clause. (The original of an original literal is itself.) In general, when a subsumption operation copies two heterogeneous, directly connected literals and creates two directly connected literals, the necessary and sufficient condition for this partial copy to be semantically equivalent to the full-clause copy is obviously that the former two literals be compatible.

When two of the original literals are not connected directly with each other, two heterogeneous literals which have directly connected originals are compatible iff they are also directly connected; we need not consider two literals whose originals are not directly connected, because one subsumption operation does not copy such literals at a time. When all of the three original literals are connected directly with each other, two heterogeneous literals are compatible iff they are connected not only directly but also through another literal heterogeneous to both. In fact, $-r(\bullet,\bullet)_2$ and $-q(\bullet,\bullet)_1$ are connected both through term ξ and through $p(\bullet,\bullet)_2$, but $-r(\bullet,\bullet)_2$ and $-q(\bullet,\bullet)$ are not connected through any instance of the original $p(\bullet,\bullet)$.

In the case of context-free parsing, $O(n^2)$ literals are created, where n is the number of words in the input string, provided that the origins of subsumptions are the positions between the input words only, due to the input-driven computation. Since there are just a constant times more links than literals, the space complexity of context-free parsing hence becomes $O(n^2)$ in our method. The time complexity is $O(n^3)$, because there are $O(n)$ different ways of making each literal. Now the correspondence with chart parsing is more exact. An arc in the chart corresponds to an instantiated literal. For instance, arc $[A \rightarrow \bullet\, B \bullet\, C]$ from node i to node j corresponds to instantiated literal $-b(\overline{A_i},\overline{A_j})$, and $[A \rightarrow \bullet\, B\, C \bullet]$ from node i to node j corresponds to $a(\overline{A_i},\overline{A_j})$. For a context-free rule with more than two symbols in the right-hand side, we can group several literals to one as mentioned above and reduce it to a rule with just two symbols in the right-hand side.

[4] So the semantic-head-driven generation parallels better with left-to-right parsing than with syntactic-head-driven parsing.

[5] The semantic monotonicity is practically same as the input-boundness with regard to semantic structures.

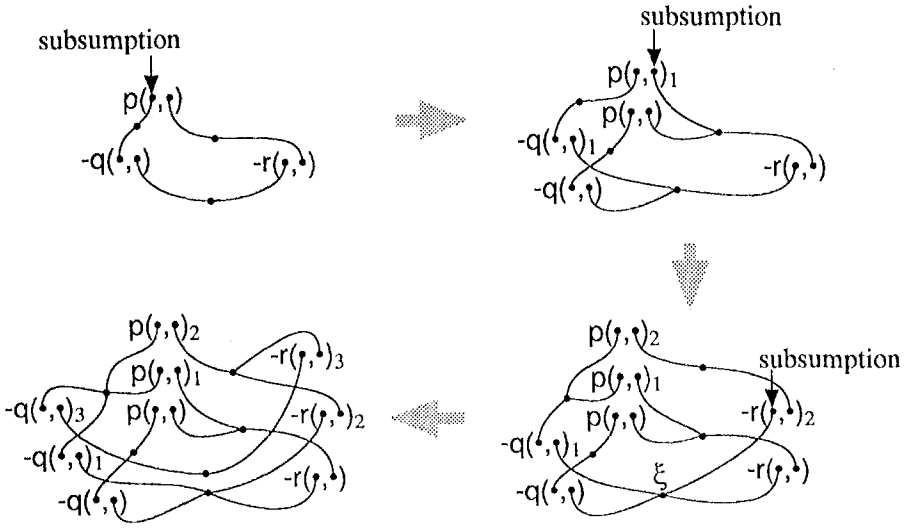

Figure 6: Subsumptions with Incremental Copy

5 Concluding Remarks

We have proposed a flexible inference method for Horn logic programs. The computation based on it is a sort of program transformation, and chart parsing and semantic-head-driven generation are epiphenomena emergent thereof. The proposed method has nothing specific to parsing, generation, context-free grammar, or the like. This indicates that there is no need for any special algorithms of parsing or generation, or perhaps any other aspect of natural language processing.

The idea reported above has already been partially implemented and applied to spoken language understanding (Nagao, Hasida, & Miyata, 1993), and an account of how the roles of speaker and hearer may switch in the midst of a sentence (Hasida, Nagao, & Miyata, 1993). Although this line of work has incorporated a notion of dynamics (Hasida, 1994b) as the declarative semantics to control context-sensitive computation, we are planning to replace dynamics with probability. For input-bound programs together with input-driven computation, it is quite straightforward to define probabilistic semantics as a natural extension of stochastic context-free grammars, among others, because all the body literals are probabilistically independent in that case. We would like to report soon on a general treatment of probabilistically dependent literals while preserving the efficient structure sharing, which will guarantee efficient computation and learning.

Reference

Hasida, K., Nagao, K., & Miyata, T. (1993). Joint Utterance: Intrasentential Speaker/Hearer Switch as an Emergent Phenomenon. In Bajcsy, R. (Ed.), *Proceedings of the 13th International Joint Conference on Artificial Intelligence* Chambéry.

Hasida, K. (1994a). Common Heuristics for Parsing, Generation, and Whatever .... In Strzalkowski, T. (Ed.), *Reversible Grammar in Natural Language Processing*. Kluwer Academic Publisher, Dordrecht.

Hasida, K. (1994b). Dynamics of Symbol Systems. *New Generation Computing, 12*(3). to appear in May 1994.

Kay, M. (1980). Algorithm Schemata and Data Structures in Syntactic Processing. Tech. rep., XEROX Palo Alto Research Center, Palo Alto, California.

Nagao, K., Hasida, K., & Miyata, T. (1993). Understanding Spoken Natural Language with Omni-Directional Information Flow. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*.

Pereira, F. C. N., & Warren, D. H. D. (1983). Parsing as Deduction. In *Proceedings of the 21st Annual Meeting of ACL*, pp. 137–144.

Shieber, S. M. (1988). A Uniform Architecture for Parsing and Generation. In *Proceedings of the 12th International Conference on Computational Linguistics*, pp. 614–619.

Shieber, S. M., van Noord, G., & Moore, R. C. (1989). A Semantic-Head-Driven Generation Algorithm for Unification-Based Formalisms. In *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics*, pp. 7–17.