ON DETERMINING THE CONSISTENCY OF PARTIAL DESCRIPTIONS OF TREES

Thomas L. Cornell
Cognitive Science Program
University of Arizona
Tucson, AZ 85721
cornell@ccit.arizona.edu

Abstract[1]

We examine the consistency problem for descriptions of trees based on remote dominance, and present a consistency-checking algorithm which is polynomial in the number of nodes in the description, despite disjunctions inherent in the theory of trees. The resulting algorithm allows for descriptions which go beyond sets of atomic formulas to allow certain types of disjunction and negation.

INTRODUCTION

In Marcus, Hindle & Fleck (1983), the authors proposed an approach to syntactic tree structures which took the primary structural relation to be remote dominance rather than immediate dominance. Recently, researchers have shown a revived interest in variants of Marcus et al.'s D-Theory, most likely due to the availability of approaches and techniques developed in the study of feature structures and their underlying logics. For example, both Rogers & Vijay-Shanker (1992) and Cornell (1992) present formal treatments of many notions which Marcus et al. (1983) treated only informally and incompletely. Furthermore, work on the psycholinguistic implications of this approach has continued apace (Weinberg 1988; Gorrell 1991; Marcus & Hindle 1990), making all the more necessary sustained foundational work in the theory of description-based tree-building applications (parsers, generators, etc.)

This paper addresses one particular problem that arises in this approach to tree building. As with feature-structures, the essential operation here is the combination of two collections of partial information about the syntactic structure of an expression. It may happen that the two collections to be combined contain contradictory information. For example one might contain the assertion that "node 7 dominates node 12" while the other claims that "node 12 precedes node 7". No tree structure can satisfy both these constraints. The operation of description combination is thus not simple set union, but, like unification, involves taking a least upper bound in a semi-lattice where lub's are not everywhere defined.

Both Rogers & Vijay-Shanker (1992) and Cornell (1992) propose to solve the D-Theoretic consistency problem by using essentially Tableau-based approaches. This can lead to combinatorial explosion in the face of disjunctions inherent in the theory of trees. But as it happens, proof techniques designed to handle general disjunctions are more powerful than we need; the disjunctions that arise from the theory of trees are of a restricted kind which can be handled by strictly polynomial means. We will see that we can efficiently handle richer notions of description than those in the "classical" D-Theory of Marcus, et al. (1983).

D-THEORY AND TREE THEORY

DESCRIPTION LANGUAGE

We will make use of the following description language $\mathcal{L}$. Define the set of basic relation names, **R**, as:

b — "below" (i.e., dominated-by)
d — "dominates"
e — "equals"
f — "follows" (i.e., preceded-by)
p — "precedes"

We define an algebra on relation names as follows.

$(S1 \vee S2)(x,y) =_{def}$ the collection of relation names in either *S1* or *S2*.

$(S1 \wedge S2)(x,y) =_{def}$ the collection of relation names in both *S1* and *S2*.

$S'(x,y) =_{def}$ the collection of relation names

[1] Many thanks to Dick Oehrle, Ed Stabler, Drew Moshier and Mark Johnson for comments, discussion and encouragement. Theirs the gratitude, mine the fault.

not in S.

We then define the full set of compound relation name expressions R^* as the closure of the basic relation names under $\wedge$, $\vee$ and $'$. A *formula* of $\mathcal{L}$ is then an element of R^* applied to a pair of node names. We will often refer to the compound relation name expression in a formula $S(x,y)$ as a *constraint on the pair* x, y. Semantically, we treat $S(x,y)$ as satisfiable if we can assign the denotata of the pair x, y to at least one of the relations denoted by members of S. On this semantics, if $S(x,y)$ is satisfiable and $S \leqslant T$, then $T(x,y)$ is satisfiable as well. Clearly the empty constraint (x,y) is never satisfiable. (Atoms of the form $e(x,y)$ are satisfiable if and only if x and y denote identical members of the domain of discourse. Atoms of the form $b(x,y)$ and $f(x,y)$ are to be considered equivalent to $d(y,x)$ and $p(y,x)$, respectively.)

A *description* is a finite set of formulas. If a description contains only formulas with a basic relation name, we will call it *classical*, since this is the type of description considered in Marcus et al. (1983).

AXIOMS

Note that such structures are not guaranteed to be trees. Therefore we make use of the following fragment of an axiomatization of tree structures, which we will assume in the background of all that follows, and formalize in the next section.

Strictness. Dominance and precedence are strict partial orders, i.e., transitive and irreflexive relations.

Equality. We assume that equality is reflexive, and that we can freely substitute equals for equals.

Exhaustiveness. Every pair of nodes in a tree stand in at least one of the five possible relations. I.e. $R(x,y)$ for all x and y.

Inheritance. All nodes inherit the precedence properties of their ancestors. So if $p(x,y)$ and $d(y,z)$, then $p(x,z)$ as well.

A number of familiar properties of trees follow from the above system. Inheritance assures both the non-tangling of tree branches and the impossibility of upward branching ('V-shaped') configurations. Inheritance, Transitivity, Substitution of equals and Exhaustiveness jointly derive the property of *Exclusiveness*, which states that every pair of nodes is related in *at most* one way. (Note that it is Exclusiveness which assures the soundess of our use of $\wedge$.) A less familiar property, which we will make some use of, is roughly parallel to Inheritance; *Upwards Inheritance* states that if x dominates y and y precedes (follows) z, then x dominates or precedes (follows) z.

Note that this system is not meant to be an axiomatic *definition* of trees; it lacks a Rootedness condition, and it allows infinite and densely ordered structures. It is specifically adapted to the *satisfiability* problem, rather than the *validity* problem. It is relatively straightforward to show that, from any finite atomic $\mathcal{L}$-description satisfying these conditions, we can construct a finite tree or a precedence ordered finite forest of finite trees (which can be extended to a finite tree by the addition of a root node). So this system is complete as far as satisfiability is concerned. Briefly, if a set of formulas satisfies all of the above constraints, then we can (1) construct a new description over the quotient node-space modulo e; (2) list the dominance chains; (3) add a root if necessary; (4) noting that the dominance maximal elements under the root must be totally precedence ordered (they must be ordered and they cannot be dominance ordered or they would not be maximal), we number them accordingly; (5) apply the same procedure to the dominance ideals generated by each of the root's daughters. From the resulting numbering we can construct a "tree domain" straightforwardly. The Inheritance property assures us that dominance chains are non-tangled, so that the ideal generatred by any node will be disjoint from the ideal generated by any node precedence-ordered with respect to the first. Therefore no node will receive two numbers, and, by Exhaustiveness, every node will receive a number.

DEDUCTION WITH DESCRIPTIONS

There is a strong formal parallel among the axioms of Transitivity, Substitution of Equals, and Inheritance: each allows us to reason from a pair of atomic formulas to a single atomic formula. Thus they allow us to reason from classical descriptions to (slightly larger) classical descriptions. Let us refer to these axioms as *generators*. The reason for adopting $\mathcal{L}$ as a description language, rather than the simpler language of Marcus et al. (1983), is that we can now treat the No Upward Branching property ("if x and z both dominate y then x dominates z or z dominates x or they are equal,") and the Upwards Inheritance property as generators. They allow us to reason from pairs of atomic formulas (e.g., $d(x,y)$ and $p(y,z)$) to compound formulas (e.g., $dp(x,z)$). This means that we can express the consequences of any pair of atomic

	b(y,z)	d(y,z)	e(y,z)	f(y,z)	p(y,z)
b(x,y)	b(x,z)	R(x,z)	b(x,z)	f(x,z)	p(x,z)
d(x,y)	bde(x,z)	d(x,z)	d(x,z)	df(x,z)	dp(x,z)
e(x,y)	b(x,z)	d(x,z)	e(x,z)	f(x,z)	p(x,z)
f(x,y)	bf(x,z)	f(x,z)	f(x,z)	f(x,z)	R(x,z)
p(x,y)	bp(x,z)	p(x,z)	p(x,z)	R(x,z)	p(x,z)

Figure 1. Generator Table.

formulas as a formula of $\mathcal{L}$, though possibly a compound formula. They are exhibited in Figure 1. Cells corresponding to axioms in the theory are boxed.

For doing formal deductions we will employ a sequent calculus adapted to our description language $\mathcal{L}$. We assume that sequents are pairs of finite sets of formulas, and we can make the further restriction that formulas on the right of the sequent arrow ("succedents") contain at most a single member. The axioms of the calculus we employ are exhibited in Figure 2, and the connective rules in Figure 3.

Structural Axioms: $\Gamma, A \rightarrow A$

Generator Axioms: $\Gamma, S1(x,y), S2(y,z) \rightarrow S3(x,z)$
for all instances of the generators

Exhaustiveness: $\rightarrow R(x,y)$ *for all x, y*

Figure 2. D-Theory Axioms.

A sequent $[\Gamma \rightarrow \Delta]$ is interpreted as an implication from conjunctions to disjunctions: if everything in Γ is true, then something in Δ must be true. It follows that $[\rightarrow \Delta]$ is invariably true, and $[\Gamma \rightarrow]$ is invariably false. A sequent calculus *proof* is a tree (written right side up, with its root on the bottom) labeled with sequents. The theorem to be proved labels its root, the leaves are labeled with axioms, and all the local subtrees must be accepted by some inference rule. A proof that a description Γ_0 is inconsistent is a proof of the sequent $[\Gamma_0 \rightarrow]$. Note that

$$\frac{\Gamma \rightarrow (x,y)}{\Gamma \rightarrow}$$

is a valid inference, essentially since (x,y) and the empty succedent both express the empty disjunction.

$$L\wedge \frac{\Gamma, S1(x,y) \rightarrow \Delta}{\Gamma, S2(x,y) \rightarrow \Delta} \; (S2 \leqslant S1)$$

$$R\wedge \frac{\Gamma \rightarrow S1(x,y) \qquad \Gamma \rightarrow S2(x,y)}{\Gamma \rightarrow (S1 \wedge S2)(x,y)}$$

$$L\vee \frac{\Gamma, S1(x,y) \rightarrow \Delta \qquad \Gamma, S2(x,y) \rightarrow \Delta}{\Gamma, (S1 \vee S2)(x,y) \rightarrow \Delta}$$

$$R\vee \frac{\Gamma \rightarrow S1(x,y)}{\Gamma \rightarrow S2(x,y)} \; (S2 \leqslant S1)$$

$$L' \; \frac{\Gamma \rightarrow S(x,y)}{\Gamma, S'(x,y) \rightarrow}$$

$$R' \; \frac{\Gamma, S(x,y) \rightarrow}{\Gamma \rightarrow S'(x,y)}$$

Figure 3. D-Theory Inference Rules.

TWO ALGORITHMS

Suppose we are given an input description Γ_0 to check for satisfiability. If it is unsatisfiable, then it contradicts one of the axioms of the tree theory: Exhaustiveness, Reflexivity of Equals, Irreflexivity of Dominance and Precedence, and the Generators. A complete consistency checker must be able to exhaust the consequences of these axioms for Γ_0, monitoring for the false formula (x,y).

Both algorithms take as input a description and an integer indicating the number of node-names constrained by the description. In the Prolog implementations that follow, the description is expected to be a list of constraints in *normal order*, that is, with the first argument lexicographically less than or equal to the second. Thus, assuming we are using integers as node names, the normal order form of $d(2,1)$ will be $b(1,2)$. Furthermore, the description is assumed to be sorted by its node-pairs. This will allow us to use efficient ordered-set manipulations.

For any given set of nodes of size N, we can construct a description which is a filter for violations of Reflexivity, Irreflexivity and Exhaustiveness. We construct F_N to contain for every pair of nodes $x_i, x_j, i,j \leqslant N$, $e(x_i,x_j)$ if $i=j$, and $e'(x_i,x_j)$ (i.e., $bdfp(x_i,x_j)$) if $i \neq j$. We can

$$\mathrm{R}\wedge\ \frac{\mathrm{e}(1,3),\mathrm{d}(1,2),\mathrm{p}(2,3)\rightarrow \mathrm{dp}(1,3)\quad \mathrm{d}(1,2),\mathrm{p}(2,3),\mathrm{e}(1,3)\rightarrow \mathrm{e}(1,3)}{\dfrac{\mathrm{e}(1,3),\mathrm{d}(1,2),\mathrm{p}(2,3)\rightarrow (1,3)}{\mathrm{e}(1,3),\mathrm{d}(1,2),\mathrm{p}(2,3)\rightarrow}}$$

$$\boxed{1}$$

Figure 4. {d(1,3), e(1,3), p(2,3)} is Inconsistent.

$$\boxed{1}$$

$$\mathrm{L}\vee\ \frac{\dfrac{\mathrm{e}(1,3),\mathrm{d}(1,2),\mathrm{p}(2,3)\rightarrow \quad \mathrm{e}(1,3),\mathrm{p}(1,2),\mathrm{p}(2,3)\rightarrow}{\mathrm{e}(1,3),\mathrm{dp}(1,2),\mathrm{p}(2,3)\rightarrow}\quad \mathrm{L}\vee\ \dfrac{\mathrm{f}(1,3),\mathrm{d}(1,2),\mathrm{p}(2,3)\rightarrow \quad \mathrm{f}(1,3),\mathrm{p}(1,2),\mathrm{p}(2,3)\rightarrow}{\mathrm{f}(1,3),\mathrm{dp}(1,2),\mathrm{p}(2,3)\rightarrow}}{\mathrm{ef}(1,3),\mathrm{dp}(1,2),\mathrm{p}(2,3)\rightarrow}$$

Figure 5. {ef(1,3), dp(1,2), p(2,3)} is Inconsistent (A R∧-L∨ Proof).

determine that a description Γ_0 violates either Reflexivity, Irreflexivity or Exhaustiveness simply by taking its pointwise meet $\Gamma_0 \sqcap F_N$.[2] If a description is in violation of the Exhaustiveness condition, then it contains some formula $S(x,y)$ with some part of S not given in R*. In that case, taking its meet with anything $\leqslant \mathrm{R}(x,y)$ will prune away the offending part. Similarly, if a constraint on a reflexive pair of nodes $S(x,x)$ fails to have $e \in S$, then taking its meet with $e(x,x)$ will yield (x,x). Finally, taking the meet of $S(x,y)$, $x \neq y$, with $e'(x,y)$ will yield (x,y) if $S \approx e$; in any case it will have the useful effect of pruning e out of S. Therefore both algorithms begin by constructing F_N and then taking its meet with the input description. This has the extra side effect that any pair of nodes x and y not explicitly constrained in the input will now be explicitly constrained.

EXTEND : TOP-DOWN BACKTRACKING SEARCH FOR CONSISTENT MAXIMAL EXTENSIONS

Given that we have begun by taking $\Gamma_0 \sqcap F_N$, we have only the generators left to check Γ_0 against. We can think of the generator table as defining a function from pairs of atomic formulas to consequences. To use it we must first have atomic formulas.

Def'n: A description Γ' is a *maximal extension* of a description Γ if, for every formula $S(x,y)$ in $\Gamma \sqcap F_N$, Γ' contains $s(x,y)$ for some $s \in S$.

An obvious solution is to enumerate the maximal extensions of $\Gamma_0 \sqcap F_N$ and feed them to the generators. If any such extension passes the generators, then it is satisfiable, and therefore it is a witness to the satisfiability of Γ_0. If the extension is unsatisfiable, then it must violate at least one of the generators. Because a maximal extension is a total assignment of node-pairs to relations, a single application of a well-chosen generator will suffice to derive a contradiction. And so a single pass through the complete set of applicable generators should be sufficient to decide if a given maximal extension is consistent.

Thus, if the input description Γ_0 is inconsistent, then there is a proof of $\Gamma_0 \rightarrow$ in which every branch of the proof ends in a sub-proof like that in Figure 4. There we have the simple description $\{d(1,2), e(1,3), p(2,3)\}$, which gives us $dp(1,3)$, by a generator (Upwards Inheritance, in this case), and $e(1,3)$, by a structural axiom. Combining these by an invocation of R∧ we get the false formula (1,3). The roots of these sub-proofs can be combined using L∨ until we eventually build up the input description on the left, proving $\Gamma_0 \rightarrow$, as in Figure 5.

The following fragment of a Prolog implementation of max_extension/3 can be seen as implementing a backwards chaining search for such a "R∧-L∨" proof. The input to both *extend* and to *close* (see below, next section) is assumed to be an $\mathcal{L}$-description together with an integer giving the number of node-names subject to the description. The node-count is used to construct the appropriate F_N for this description. Note

[2] We can assume that any pair of nodes x, y not explicitly constrained in the input is implicitly constrained by $\mathrm{R}(x,y)$. Of course, $(\mathrm{R} \wedge e') = e'$, so this assumption just amounts to setting unmentioned pairs of (distinct) nodes to $e'(x,y)$.

that, aside from implementing pointwise $\sqcap$, merge_descs/3 checks for the derivation of an empty constraint, and fails if that happens. The real work is then done by extend/3, which is a recursion on an Agenda. The agenda is initialized to the input description. As individual constraints are narrowed, they are added to the agenda so as to implement constraint propagation.

```
max_extension( D0, N, Extension ) :-
    ir_reflexive_rule( N, Filter_N ),
    merge_descs( D0, Filter_N, D1 ),
    Agenda = D1,
    extend( Agenda, D1, Extension ).

extend( [], X, X ).
extend( [C0|Cs], D0, X ) :-
    consequences( C0, D0, Conseqs ),
    meet_rule( Conseqs, D0, D1, NewCons ),
    merge_descs( NewCons, Cs, Agenda1 ),
    extend( Agenda1, D1, X ).
```

Meet_rule/4, in the second clause of extend/3, differs from merge_descs/3 only in (a) sorting its first argument and (b) deriving both the merged description (D1) and a list of those consequences which actually had some effect on D0. Both merge_descs/3 and meet_rule/4 are based on routines for ordered set union from O'Keefe (1990). The main difference is that ordering is defined on the node-pairs of the constraint, rather than on the term expressing the constraint as a whole; equality is defined so that two constraints are equal if they constrain the same node pair, and if two formulas are 'equal' in this sense, then the output contains the meet of their respective relation names expressions. The truly new consequences derived by meet_rule/4 are then added to the remaining agenda (Cs) with another call to merge_descs/3. (If NewCons were merely appended to Cs, we could have two constraints on the same pair of nodes in the agenda at once, either of which may be less tightly constrained than the result of merging the two instances.)

Extend/3 thus both consumes items off the agenda (C0) and adds new items (NewCons). However, each new consequence, if it is truly novel, represents the narrowing of a constraint; since each pair starts with a maximum of four options, clearly we will eventually run out of options to remove; NewCons will be empty, the remaining agenda will eventually be consumed, and the program will halt.

The core of extend/3 is consequences/3, which determines for any given constraint what consequences it has when paired with each of the constraints in the description. Consequences/3 has two clauses; the first handles compound formulas, while the second handles atomic formulas. The second clause of consequences/3 invokes the Splitting Rule, which implements LV.

Note that, instead of exhausting the consequences of the Splitting Rule and then applying the Generator Rule, we apply the Generator Rule whenever we can. This is because it can act to prune away options from its consequents, thus minimizing the combinatorial explosion lurking behind the Splitting Rule. Furthermore, if an application of the Generator Rule does lead to the discovery of an inconsistency, then the program backtracks to its last application of the Splitting Rule, in effect pruning away from its search tree all further consequences of its inconsistent choice.

```
consequences( C, _D, Consequences ) :-
    compound_formula( C ),
    splitting_rule( C, Consequences ).
consequences( C, D, Consequences ) :-
    atomic_formula( C ),
    generator_rule( D, C, Consequences ).

atomic_formula( [_]:(_,_) ).
compound_formula( [_,_|_]:(_,_) ).

splitting_rule( C, [Assumption] ) :-
    C = Rels:Nodes,
    member( R, Rels ),
    Assumption = [R]:Nodes.
```

The heart of consequences/3 is the Generator Rule, implemented as generator_rule/3. It scans the current description for formulas which form a connected pair with its second argument. Note that in all our examples, we have carefully presented inputs to the generators as $S_1(x,y)$, $S_2(y,z)$. Such a combination can be looked up directly in the generator table. However, note that $S_1(x,y)$, $S_2(z,y)$ is no less a connected pair. In order to match it to the generator table, though, we need to invert the second member, giving $S_2^{-1}(y,z)$. This is done by connected_order/4, which succeeds, returning the connected form of the formulas, if they have a connected form, and fails otherwise. If it succeeds, then there is an entry in the generator table which gives the consequence of that connected pair. This consequence (XZ) is then placed in normal order (C3), and added to the output list of consequences.

If C2 is an unconnected atom, or a compound formula, it is skipped. Note that skipping compound formulas does not affect the

completeness of the algorithm. Every agenda item leads a dual life: as an agenda item, and as a member of the current description. The ignored compound formula will eventually be subjected to the Splitting Rule, the result being placed on the agenda. It will then eventually be paired with C2's entry in the description by the Generator Rule. The only difference will be in which formula is the left antecedent and which the right; but that doesn't matter, since they'll be converted to connected form in any case, and their result will be converted to normal order.[3]

```
generator_rule( [], _C, [] ).
generator_rule( [C2|Rest], C1, [C3|Conseqs] ) :-
    atomic_formula( C2 ),
    connected_order( C1, C2, XY, YZ ),
    gen( XY, YZ, XZ ),
    normal_order( XZ, C3 ),
    generator_rule( Rest, C1, Conseqs ).
generator_rule( [C2|Rest], C1, Conseqs ) :-
    atomic_formula( C2 ),
    \+ connected_order( C1, C2, _, _ ),
    generator_rule( Rest, C1, Conseqs ).
generator_rule( [C2|Rest], C1, Conseqs ) :-
    compound_formula( C2 ),
    generator_rule( Rest, C1, Conseqs ).
```

Every rule applied in this procedure is based on a rule in the associated sequent calculus. The Splitting Rule is just L$\vee$; the Meet Rule is R$\wedge$; and the Generator Rule is just the application of an axiom. So there can be little doubt that the algorithm is a sound implementation of a search for a L$\vee$-R$\wedge$ proof of $\Gamma_0 \rightarrow$. That it is complete follows from the fact that consistent maximal extensions are Hintikka sets. In particular, every generator $\Gamma, A, B \rightarrow C$ has the same truth conditions as the set of formulas $\Gamma \cup \{\neg A \vee \neg B \vee C\}$. So a maximal extension is a Hintikka set if it contains either $\neg A$ or $\neg B$ or C for every generator. The exhaustiveness of our search assures this: every pair of constraints is checked at least once to see if it matches a generator. If it does not then the extension must contain either $\neg A$ or $\neg B$. If it does, then the extension contains A and B, and so it must also contain C, or be found inconsistent by the Meet Rule/R$\wedge$.

However, completeness is purchased at the cost of the complexities of exhaustive search. Note that the Splitting Rule is the only source of non-determinism in the program. All of the routines whose definitions were left out are deterministic. The ordered set manipulations are linear in the size of the combined input lists; the sort called by the Meet Rule is just a variant of merge-sort, and so of $N\ logN$ complexity; the many inversions which may have to be done are linear in the length of the constraint list, which is bounded from above by 4, so they can be treated as constant time operations. It is only the Splitting Rule that causes us trouble. The second algorithm attempts to address this problem.

CLOSE: POLYNOMIAL SEARCH FOR A L$\vee$-R$\wedge$ PROOF

The basic design problem to be solved is that the generator table accepts only atomic formulas as inputs, while the description whose consistency is at issue may contain any number of compound formulas. *Extend* solved this problem by 'bringing the description to the generators.' *Close* solves this problem by 'bringing the generators to the description.'

Figure 6 represents a proof that $\{dp(1,2), bf(1,3), dp(2,3)\}$ is inconsistent. Here the leaves are almost entirely drawn from the generator axioms. Only the rightmost leaf invokes a structural axiom. The initial stages of the proof involve combining generators by means of R$\vee$ and L$\vee$ until the two antecedent atoms match a pair of compound atoms found in the input description (in this case $dp(1,2)$ and $dp(2,3)$). Then this 'compound generator' is fed into the R$\wedge$ rule together with the corresponding structural axiom, generating our inconsistency.

Close, like *extend*, implements a backwards chaining search for a proof of the relevant sort. The code for the two algorithms has been made almost identical, for the purposes of this paper. The sole essential difference is that now consequences/3 has only one clause, which invokes the New Generator Rule. The input to new_generator_rule/3 is the same as the input to generator_rule/3: the current description, a constraint looking to be the left antecedent of a generator, and the output consequences. Like the old rule, the new rule searches the current description for a connected formula (now not

---

[3] In fact, every connected pair has two connected forms: $S_1(x,y)$, $S_2(y,z)$ and $T_2(z,y)$, $T_1(y,z)$. Unsurprisingly, in this case the output of the generator table for T_2 and T_1 will be the inverse of what it is for S_1 and S_2. In either case, the output will be placed in normal order before being entered into the description, so we have the required commutativity.

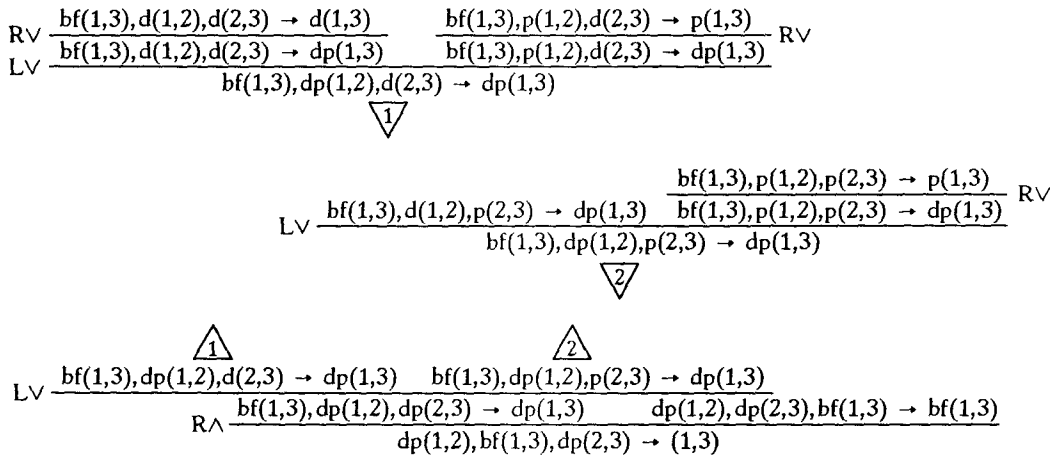

Figure 6. A LV-RA Proof that {dp(1,2), bf(1,3), dp(2,3)} is Inconsistent.

necessarily atomic). From the resulting connected pair it constructs a compound generator by taking the cross product of the atomic relations in the compound formulas (in a double loop implemented in distribute/4 and distrib_1/4), feeding the atomic pairs so constructed to the generator table (in distrib_1/4), and joining each successive generator output. The result is a compound generator whose consequent represents the join of all the atomic generators that went into its construction.

```
new_generator_rule( [], _C, [] ).
new_generator_rule( [C2|Rest], C1, [C3|Cons] ) :-
    connected_order( C1, C2, S1:(X,Y), S2:(Y,Z) ),
    distribute( S1:(X,Y), S2:(Y,Z), []:(X,Z), S3:(X,Z) ),
    normal_order( S3:(X,Z), C3 ),
    new_generator_rule( Rest, C1, Cons ).
new_generator_rule( [C2|Rest], C1, Cons ) :-
    \+ connected_order( C1, C2, _, _ ),
    new_generator_rule( Rest, C1, Cons ).

distribute( []:_, _C2, Cons, Cons ).
distribute( [R1|S1]:XY, S2:YZ, S3a:XZ, S3:XZ ) :-
    distrib_1( S2:YZ, [R1]:XY, S3a:XZ, S3b:XZ ),
    distribute( S1:XY, S2:YZ, S3b:XZ, S3:XZ ).

distrib_1( []:_, _C1, Cons, Cons ).
distrib_1( [R2|S2]:YZ, S1:XY, S3a:XZ, S3:XZ ) :-
    gen( S1:XY, [R2]:YZ, S3b:XZ ),
    ord_union( S3a, S3b, S3c ),
    distrib_1( S2:YZ, S1:XY, S3c:XZ, S3:XZ ).
```

On completion of the double loop, control works its way back to consequences/3 and thence to the Meet Rule, as usual.

Unlike *extend*, *close* is deterministic. Each agenda item is compared to each item in the current description, and that is that. Furthermore, the complexity of the New Generator Rule is not much greater than before: the double loop we have added can only be executed a maximum of $4 \times 4 = 16$ times, so we have increased the complexity of the algorithm, considered apart from the Splitting Rule, by at most a constant factor. The question is: at what cost?

Before we turn to the analysis of *close*, however, note that its output is different from that of *extend*. *Extend* returns a maximal extension, selected non-deterministically. *Close* returns the input description, but with values that could not be part of any solution removed. Essentially, *close* returns the pointwise join of all of Γ_0's consistent maximal extensions.

This action, of joining all the atomic consequences of a pair of constraints, does not preserve all of the information present in the atomic consequences. Consider the following description.

$$\Gamma_0 = \{d(1,2), dp(1,3), dp(2,3)\}$$

Γ_0 is its own closure, and is consistent. However, if we examine its maximal extensions, we note that one of them

$$\Gamma_3 = \{d(1,2), p(1,3), d(2,3)\}$$

is inconsistent. There is nothing in Γ_0 to tell us that one combination of the values it presents is impossible. Note that this may not be essential to proving inconsistency: for Γ_0 to be inconsistent, it would have to be the case that all values in some constraint were ruled out in *all* maximal extensions.

ANALYSIS OF *CLOSE*

We first argue that *close* is indeed finding a LV-RΛ proof of $\Gamma_0 \rightarrow$. Note that in our toy example of Figure 6 only a single 'compound generator' was required to derive the empty solution. In general it may take several compound generators to build a proof of $\Gamma_0 \rightarrow$. Each one functions to remove some of the possibilities from a constraint, until eventually no possibilities are left. Thus we have a LV-RΛ proof of $\Gamma_0 \rightarrow$ if and only if we have a proof of $\Gamma_0 \rightarrow (x,y)$, for some x and y. Let us call such a (not necessarily unique) pair a *critical pair* in the proof of $\Gamma_0 \rightarrow$, and its associated constraint in Γ_0 a *critical constraint*.

It is not at all obvious how to choose a critical constraint beforehand, so *close* must search for it. Every time it calls the New Generator Rule and then calls the Meet Rule to merge in its consequence, it constructs a fragment of a LV-RΛ proof. We could then take the constraint which it finally succeeds in emptying out as the critical constraint, collect the proof fragments having that constraint as their succedent, and plug them together in the order they were generated to supply us with a LV-RΛ proof of $\Gamma_0 \rightarrow$.

So *close* will find a LV-RΛ proof of $\Gamma_0 \rightarrow$, if one exists. It is not clear, however, that such a proof always exists when Γ_0 is unsatisfiable. *Close* is essentially a variant of the path-consistency algorithms frequently discussed in the Constraint Satisfaction literature (Mackworth, 1977; Allen, 1983). It is known that path-consistency is not in general a strong enough condition to ensure completeness. There are, however, special cases where path-consistency techniques are complete (Montanari, 1974).

So far, *close* appears to be complete, (two years of work have failed to turn up a counterexample) but it is unlikely to yield an easy completeness proof. The algorithm presented here is strongly reminiscent of the algorithm in Allen (1983), which is demonstrably incomplete for the temporal reasoning problems to which he applied it. Therefore, if *close* is complete for D-theory, it can only be due to a property of the generator axioms, that is, to properties of trees, as contrasted with properties of temporal intervals. Standard approaches of any generality will almost certainly generalize to the temporal reasoning case.

REFERENCES

Allen, James F. 1983. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM* 26(11): 832-843.

Cornell, Thomas L. 1992. *Description Theory, Licensing Theory and Principle-Based Grammars and Parsers*. UCLA Ph.D. thesis.

Gorrell, P. 1991. Subcategorization and Sentence Processing. In Berwick, R., S. Abney & C. Tenney, eds. *Principle-Based Parsing: Computation and Psycholinguistics*. Kluwer, Dordrecht.

Mackworth, Alan K. 1977. Consistency in Networks of Relations. *Artificial Intelligence* 8: 99-118.

Marcus, Mitchell P., & Donald Hindle. (1990). Description Theory and Intonation Boundaries. In G. T. M. Altman (Ed.), *Cognitive Models of Speech Processing* (pp. 483-512). Cambridge, MA: MIT Press.

Marcus, Mitchell P., Donald Hindle & Margaret M. Fleck. 1983. D-Theory: Talking about Talking about Trees. *Proceedings of the 21st Mtg. of the ACL*.

Montanari, Ugo. 1974. Networks of Constraints: Fundamental Properties and Applications to Picture Processing. *Information Sciences* 7: 95-132.

O'Keefe, Richard A. 1990. *The Craft of Prolog*. Cambridge, MA: MIT Press.

Rogers, James & K. Vijay-Shanker. 1992. Reasoning with Descriptions of Trees. *Proceedings of the 30th Mtg. of the ACL*.

Weinberg, A. 1988. *Locality Principles in Syntax and in Parsing*. MIT Ph.D. dissertation.