Lexical Constraints in DATR

Lionel Moser
School of Cognitive & Computing Sciences
University of Sussex
Brighton, U.K.
email: lionelm@cogs.sussex.ac.uk

February 1992

Abstract

DATR contains no special features to support testing of equality, negation, disjunction, or multiple inheritance. Nevertheless, given an appropriate interpretation it is possible, within DATR's existing syntax and semantics, to represent these operations. In this paper we review the technique known as *negative path extension*, and show how it can be used to reconstruct negation, disjunction, and equality testing. We then show how these operations can be used to define what are essentially meta-level constraints on DATR lexical derivation.

Appendix of DATR Examples

Introduction

DATR is an 'untyped' inheritance network representation language, in the sense that all extensional values are of the same basic type (sequences of atoms), and there is no restriction on what possible extensional values can be reasonably derived for a given path.[1] 'Typed' languages, on the other hand constrain an entity to have (or represent) some value of the appropriate type. Characters, strings, integers, and integer subranges are familiar examples of types. 'Typed feature structures' are those which are obliged to satisfy some set of 'type constraints', typically a logical formula consisting of disjunctions, negations, equalities and logical connectives. While DATR has no special facilities to support any of the types of constraints discussed in this paper, all can be expressed within this paradigm, and more – notably various types of multiple inheritance (Evans *et al.* 1991, Moser 1992b).

We begin by developing a technique known as *negative path extension* (introduced in Evans *et al.* (1991) and Moser (1991)), and then use it, along with the observation that the set of symbols in

The type of constraints with which we are concerned in this paper are intra-lexical – that is, they constrain a single lexical entry, not some projection of it (*e.g.*, a phrase). They may be used for consistency checking in the lexicon, but they are *not* constraints on composite feature structures. An external constraint grammar[2] (such as HPSG (Pollard and Sag 1987, Pollard and Sag 1991)) imposes constraints on relations between feature structure constituents of separate feature structures;[3] DATR is designed only to represent lexical items, so we may say that any constraints evaluable in DATR are intra-lexical-item constraints. These are not without value, as certain types of constraints are inheritable within and apply to the lexical

then the specific exception was *not* matched. The DATR rule of path extension may be stated as follows: A path p queried at node `A` extends the longest defined path at `A` which is a prefix of p. This 'longest-defined-prefix-wins' rule derives the form of inference known in DATR as *default extension*. For example, from the node definition for `Bird`,

```
Bird: <eats> == yes.
```

we can derive the following statements:[6]

```
Bird: <eats> = yes
      <eats worms> = yes.
```

The first follows from the definition of that particular path, while the second extends (by default) its longest defined prefix (`<eats>`). The statement

```
Bird: <swims> = yes.
```

is not derivable, since there is no prefix of the path `<swims>`

```
A: <x> == w x y.

B: <y> == a.

D: <> == <A B:<y> c>
   <w> == z
   <w x> == yes.
```

Notice in the above example that the domains of path constituents and values are not disjoint. The values `w`, `x`, `y`, and `a` at nodes `A` and `B` are interpreted as path constituents at node `D`, when the evaluable path `<A B:<y> c>` is instantiated. This evaluable path specifies non-local inheritance from nodes `A` and `B`. The derivation of `D:<x> = yes` is shown in Figure 1.

Initial query	Derived value	Justification
`D:<x> =`	`D:< A:<x> B:<y x> c x>`	`D:<> == <A B:<y> c>`

`=`	`D:<w x y a c x>`	evaluable path instantiation, using
		`A:<x> = w x y.`
		`B:<y x> = a.`

`=`	`yes`	`D:<w x> == yes.`

Figure 1: Derivation of `D:<x> = yes`.

Equality

Negative path extension is essentially a test for inequality against a specific case or cases. Using NPE we can define a test for equality against a given constant. From

```
Equal_13: <13> == true
          <> == false.
```

we can derive the following statements:

```
Equal_13: <13> = true
          <14> = false.
```

The first statement derives from the fact that the path `Equal:<13>` is defined (to have value `true`), while the second does not extend path `<13>`, so extends only the next shortest selector, the empty path (`<>`), which has value `false`.

Equality as a local constraint

An equality test can be used to define constraints on feature values. We begin by illustrating a simple local constraint – *i.e.*, a constraint self-imposed at the defining node.

Node `C` constrains the value of its path `<x>` to be `13`, using the definition of `Equal_13` given above:

```
C: <constraint1> == Equal_13:<<x>>
   <constraint2> == Equal_13:<<y>>
   <x> == 13
   <y> == 14.
```

The evaluation of path `C:<constraint1>` tests whether `C` satisfies the constraint that its value for path `<x>` extend `<13>`. Similarly for `C:<constraint2>` and path `<y>`, yielding the two following theorems:

```
C: <constraint1> = true
   <constraint2> = false.
```

Their derivations are shown in Figures 2 and 3, respectively.

Initial query	Derived value	Justification
C:<constraint1> =	Equal_13:<C:<x>>	C:<constraint1> == Equal_13:<<x>>
=	Equal_13:<13>	evaluable path instantiation using C:<x> = 13.
=	true	Equal_13:<13> = true.

Figure 2: Derivation of C:<constraint1> = true

Initial query	Derived value	Justification
C:<constraint2> =	Equal_13:<C:<y>>	C:<constraint2> == Equal_13:<<y>>
=	Equal_13:<14>	evaluable path instantiation using C:<y> = 14.
=	false	Equal_13:<14> = false.

Figure 3: Derivation of C:<constraint2> = false

Generalizing equality

Since there may be a number of atomic tests for equality in the hierarchy, rather than create a new node for each comparison we can bundle all of the atomic comparisons in a single node, using the particular test value as a selector path prefix. A single node can be used to test for all such cases of equality:

```
Equal:
  <13 13> == true
  <13> == false
  <singular singular> == true
  <singular> == false
  <> == 'unexpected selector'.
```

Now we test a path against a value by using the known constant as a selector:

```
Equal:<13 <x>> = true   (if <x> extends <13>)
Equal:<13 <x>> = false  (if <x> does not extend <13>)
```

The definition of node C is modified as follows:

```
C: <constraint1> == Equal:<13 <x>>
   <constraint2> == Equal:<13 <y>>
   <x> == 13
   <y> == 14.
```

We still have the following derivable statements:

```
C: <constraint1> = true
   <constraint2> = false.
```

Given conservative intuitions about the notion of equality, it is perhaps anomalous that the following theorems are also derivable, by default extension:

```
Equal: <13 13 hi mom> = true
       <13 13 21> = true.
```

To restrict equality to a better approximation of our intuitive notion we could insert a delimiter after the test value, which can be matched in the selector path, and modify the definition of `C` accordingly:[7]

```
Equal:
  <13 13 ;> == true
  <13> == false
  <singular singular ;> == true
  <singular> == false
  <> == 'unexpected selector'.

C: <constraint1> == Equal:<13 <x> ;>
   <constraint2> == Equal:<13 <y> ;>
   <constraint3> == Equal:<13 <z> ;>
   <x> == 13
   <y> == 14
   <z> == 13 14.
```

This gives the desired result: the test for equality will fail precisely when the test argument path (or test value) are not equal. Thus we have have the following theorems:

```
C: <constraint1> = true
   <constraint2> = false
   <constraint3> = false.
```

However, if we delimit the selector as well as the test value, we can use *any* path as selector, subject to its value being in the set of selectors. Doing this, we have:[8]

```
Equal:
   <13 ; 13 ;> == true
   <13 ;> == false
   <singular ; singular ;> == true
   <singular ;> == false
   <> == 'unexpected selector'.


C: <constraint1> == Equal:<13 ; <x> ;>
   <constraint2> == Equal:<13 ; <y> ;>
   <constraint3> == Equal:<<x> ; <z> ;>
   <x> == 13
   <y> == 14
   <z> == 13 14.
```

Parametrization using DATR variables

By defining a DATR variable, `$terminal`, which enumerates the atoms over which the logical operators are defined, we can make the definition of `Equal` quite concise. For the examples given thus far, an adequate definition would be:

```
#vars $terminal: 1 2 3 13 14 singular true false.
```

The restatement of `Equal` in terms of `$terminal` is rather elegant:

```
Equal:
   <$terminal ; $terminal ;> == true
   <$terminal ;> == false
   <> == 'unexpected selector'.
```

The symbol `;`, which lies outside the descriptive domain of the theory, is not enumerated in `$terminal`. Other symbols which are not enumerated by `$terminal` may also be used (*e.g.*, as path constituents), so long as they do not appear in paths on which the logical operators are queried.[9]

A constraint logic

Thus far we have used only a trivial constraint, testing a value against a single atomic constant. In order to define conjunctive and disjunctive constraints, we require a logic for combining conjuncts and disjuncts. If the constraints were to be applied only to fully-specified lexical entries, in which every path expected to have a value does indeed have one, then classical logic would seem appropriate. On the other hand, if we wish to state constraints which *might* be applicable, then we might also want to have some third logical value, interpreted as 'indeterminate'. One example, mentioned above, was the possibility of testing such constraints as determiner-noun compatibility, where one of the operands was supplied from a system external to the lexicon. If the constraint were tested in the absence of a required value, it would be neither satisfied nor violated. We

In general, we can use function composition to define any logical expression. Material implication, for example, could be defined by the expression:

```
D5: <c4> == Or:<Not:<c1> <c2> ;>.
```

`D5:<c4>` evaluates to `true` whenever `<c1>` is false or `<c2>` is true, precisely when `<c1>` $\Rightarrow$ `<c2>` is true. This can be isolated at a node as follows:

```
Implies: <> == true
         <true false> == false.
```

and then the constraint at `D5:<c4a>` can be defined as:

```
D5: <c4> == Implies:<<c1> <c2>>.
```

Disjunctive constraints such as `D6:<c5>` test an extensional value for membership in a set, *e.g.*, `<person>` $\in \{1, 2, 3\}$:

```
D6: <c5> == Or:<Equal:<<person> ; 1 ;>
                Equal:<<person> ; 2 ;>
                Equal:<<person> ; 3 ;>
                ;>
    <person> == 2.
```

Subset relations

The generalization of membership of an atom in a list is set inclusion, where *every* atom in a list is a member of a second list. Specifically, if we let $X = x_1 x_2 \ldots x_n$ and $Y = y_1 y_2 \ldots y_m$, then we define `Subset:<` $x_1\ x_2\ \ldots\ x_n$ `;` $y_1\ y_2\ \ldots\ y_m$ `;>` to be `true` whenever every x_i is some y_j, and `false` otherwise. Before defining `Subset`, we first make use of path-to-value conversion (Moser 1992a) to extract the sequence of atoms $y_1 y_2 \ldots y_m$ `;`:

```
Arg2: <> == <scan_to_;>
   <scan_to_; $terminal> == () <scan_to_;>
   <scan_to_; ;> == () <copy_to_;>
   <copy_to_; $terminal> == $terminal <copy_to_;>
   <copy_to_; ;> == ().
```

`Arg2` is essentially a two-state automaton, with states `scan_to_;` and `copy_to_;`. It produces the sequence of symbols between the first and second `;` by: starting in state `scan_to_;`, where it (a) scans (and removes) terminals up to the first `;`, outputting nothing (which we illustrate as the empty list `()`); (b) jumps to state `copy_to_;` on input `;`, again outputting nothing. From its second state it has two edges, one traversed on terminals, which it copies to the output, and another traversed on symbol `;`, in which case it outputs nothing and halts.[12]

Typical theorems of `Arg2`, given a suitable definition of `$terminal`, are:

```
Arg2: <2 3 ; 1 4 ;> = 1 4
      <1 2 3 2 1 2 3 ; 1 4 3 2 1 5 ;> = 1 4 3 2 1 5.
```

We define $x_0 x_1 \ldots x_n \subseteq y_0 y_1 \ldots y_m$ recursively in terms of n: if $n = 0$ then it's true; if $n = 1$ then it's true if $x_1 \in Y$; and if $n > 1$ then it's true if $x_1 \in Y$ and $x_2 \ldots x_n \subseteq Y$. We again make use of the assumption that `$terminal2` is defined identically to `$terminal`, allowing us to form a cross-product of terminals.

```
Subset:
   <;> == true
   <$terminal ;> == Member
   <$terminal $terminal2> == And:<Member:<$terminal ; Arg2 ;>
                                 Subset:<$terminal2> ;>.
```

The following are example theorems of `Subset`:

```
Subset: <3 1 ; 1 2 3 4 ;> = true
        <3 1 ; 1 2 4 5 5 ;> = false.
```

Constraints can be defined in terms of `Member` and `Subset`, just as was done using `And`, `Or` and `Not`:

```
F1: <c5> == Subset:< <x> ; <y> ;>
    <c6> == Member:< <person> ; 1 2 3 ;>.
```

Constraint inheritance

All of the constraints we have shown so far have been locally defined. For illustrative purposes this is fine, but of course the usual case is that some more general class constrains its subclasses and instances. This can be done by defining the constraints in terms of 'global inheritance', where the test values are defined with respect to the global context:

[12] Evans & Gazdar (1990) discuss formally the class of DATR theories which are equivalent to finite state automata.

```
Parent1:
  <constraints> == And:< <c1> <c2> <c3> ;>
  <c1> == Member:<"<person>" ; 1 2 3 ;>
  <c2> == Member:<"<gender>" ; male female neuter ;>
  <c3> == Not:<And:<Equal:<"<person>" ; 3 ;>
                   Equal:<"<number>" ; singular> >>.

Child1: <> == Parent1
   <person> == 3
   <number> == plural
   <gender> == female.
```

Here the constraints are defined so that the values to be tested are inherited from the context node. `Child1` inherits `<constraints>` (by default, in this case) from `Parent1`, and `Parent1` inherits `<person>` (as well as `<gender>` and `<number>`) from whichever node originated the query.

It might be desirable, depending upon the grammatical theory, to define constraints which apply to nodes lacking some constrained attribute. For example, there might be general constraints on feature `<gender>`, but some members of the class to which the constraint applies might be unspecified for gender. Since in DATR one cannot query paths which have no definition, we might supply a default of the form `<> == undef` at an appropriate node (perhaps at each node) such that any path otherwise undefined evaluates to this value. `undef` is assumed to be a symbol lying outside the descriptive domain of the theory. We could then apply constraints only when the paths have values lying within the theory's descriptive domain. We first define, for convenience, `Is_defined:<`*val*`>` to be true whenever *val* does not extend `undef`:

```
Is_defined: <> == true
            <undef> == false.
```

We now define constraints so that they are not violated by the absence of a test value. Equally, we can test whether an obligatory path *is* defined:

```
Parent2: <> == undef
   <c6> == Implies:<Is_defined:<"<person>"
                   Member:<"<person>" ; 1 2 3 ;> >
   <c7> == Is_defined:<"<spelling>">.
```

`Parent2:<c6>` is reminiscent of GPSG Feature Co-occurrence Restrictions (Gazdar *et al.* 1985) (although in GPSG such constraints are grammatical, or extra-lexical, so would fall in the class constraints merely stated as constants in a DATR representation). In HPSG (Pollard and Sag 1987, Pollard and Sag 1991) constraints on the well-formedness of feature structures hold at both lexical and grammatical levels, since lexical entries and their projections are both represented as feature structures, or signs.[13] Adopting the terminology of Carpenter (1990), a feature structure is said to be *well-typed* if (i) the value of every feature is appropriate (*i.e.*, satisfies the constraints on that feature's value); and (ii) every feature defined is appropriate (with respect to a type hierarchy). Furthermore, a feature structure is *totally well-typed* if (iii) every appropriate feature is defined.[14] Such constraints are not extra-lexical – they apply equally to both lexical and phrasal signs. It is (a subset of) those which apply to the structure of lexical signs which could be embedded into lexical entries by local definition or inheritance. In HPSG the lexicon is considered to be a structured hierarchy, with default inheritance providing all of the generalizable structure of particular word classes. Embedding such constraints (as are representable) in the definitions of word classes would facilitate testing that lexical entries themselves are well-formed.

[13] The primary operation in HPSG in unification, and lexical entries are the predefined building blocks.

[14] In this case there is no possibility of constraint violation 'by default' – if a feature is appropriate and undefined, the constraints of the type hierarchy have been violated.

number of atoms, we have expressed constraints as tests over an enumeration of them. We used DATR variables to

[Pollard and Sag 1987] C. Pollard and I.A. Sag. *Information-Based Syntax and Semantics: Volume I – Fundamentals*. CSLI Lecture Notes Series, No. 13. Chicago University Press, Chicago, 1987.

[Pollard and Sag 1991] C. Pollard and I.A. Sag. *Information-Based Syntax and Semantics: Volume II – Agreement, Binding, and Control*. Manuscript, circulated at the Third European Summer School in Language, Logic and Information, Saarbrucken, Germany, August 1991.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                      %
% File:            clogic.dtr                                          %
% Purpose:         Define logic for working with constraints.          %
% Author:          Lionel Moser, December  1991                        %
% Documentation:   HELP *datr                                          %
% Related Files:   lib datr; args.dtr                                  %
% Version:         7.00                                                %
%     Copyright (c) University of Sussex 1991.  All rights reserved.   %
%                                                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% $terminal must be defined elsewhere if loading this file from some
% other file.
%#vars $terminal: alpha beta gamma 1 2 3 undef.

%#load 'args.dtr'.
%#load 'arglogic.dtr'.

% Polyadic AND
% And:<bool bool ... bool ;> ==
%                   true   -  if all bools are true;
%                   false  -  if any bools are false.
And: <> == '**** ERROR: (And) Invalid argument'
     <;> == true
     <true>   == <>
     <false>  == false.

% Polyadic OR
% Or:<bool bool ... bool ;> ==
%        true  -   if any bool is true;
%        false -   if no bool is true.
Or: <> == '**** ERROR: (Or) Invalid argument'
   <;> == '**** ERROR: (Or) Nil bool list'
   <true ;> == true
   <false ;> == false
   <true>   == true.

% Not.
Not: <> == '**** ERROR: (Not) Invalid argument'
   <true> == false
   <false> == true.

% Implication
Implies:
   <> == true
   <true false> == false.
```

```
% We assume a default value of indef for any path lacking a value
% within the descriptive domain of the theory.
% A path is defined if it does not extend 'undef'.
Is_defined:
   <undef> == false
   <> == true.

% Safe:<path>
% A path is 'safe' if it satisfies its constraints when evaluated with
% respect to the global context. If the path is safe, then return this
% value; if it is not safe, return a message.
Safe: <> == <If:< "<constraints>" > >
   <then> == "<>"
   <else> == 'constraint violation'.
```

% %%

```
% -------- Some theorems --------
%   B: <person> = 3
%      <constraints> = false.

%   C: <person> = 1
%      <constraints> = true.


% ParentType introduces compound constraints, and contains all of the
% top-level machinery.
ParentType:
   <> == undef
   <safe> ==
```

```
            % M violates its constraints - <number> is not defined.
            % satisfied.
            M: <> == ParentType
               <person> == 3.

            % --- Some theorems ----------------
            %
            %  ParentType: <local_constraints> = ()
            %              <constraints> = false
            %              <safe> = constraint violation
            %              <c1> = false
            %              <c2> = false
            %              <c3> = true.

            %  K:=  = false
%              <c3> = true.

%  K:=  = false
 <local_constraints> = €)false
```

```
% Subset takes a cross-product of $terminal, so requires $terminal2
% to be defined identically to $terminal.
%
% Subset:< X0 X1 X2 ... Xn ; Y0 Y1 Y2 ... Ym ;> ==
%          true    - if every Xi is in Y
%          false    - otherwise
Subset:
   <;> == true
   <$terminal ;> == Member
   <$terminal $terminal2> == And:<Member:<$terminal ; Arg2 ;>
                                 Subset:<$terminal2> ;>.

% --- Some theorems ----
%  Subset: <; ;> = true
%          <; 1 2 3 2 ;> = true
%          <1 ; 1 2 3 2 ;> = true
%          <1 ; 2 3 2 2 ;> = false
%          <2 1 ; 2 3 2 2 ;> = false
%          <2 1 3 ; 2 3 2 2 1 ;> = true.


% Union:<X0 X1 ... Xn ; Y0 Y1 ... Ym ;> == X U Y
% We define X U Y as Y + (Y - X)
X2: <> == Arg1.
Y2: <> == Arg2.

Union:
   <;> == Y2
   <$terminal> == <If:<Member:<$terminal ; Y2 ; !> > $terminal>
   <then $terminal> == Union:<>
   <else $terminal> == Union:<X2:<> ; Y2:<> $terminal ; !>.

% --- Some theorems -----
%
% Union: <1 2 3 ; 1 2 3 ;> = (1 2 3)
%        <; 1 2 3 ;> = (1 2 3)
%        <1 2 3 ; 2 3 ;> = (2 3 1)
%        <1 2 3 4 5 ; ;> = (1 2 3 4 5)
%        <1 3 5 ; 2 4 6 ;> = (2 4 6 1 3 5)
%        <; ;> = ().
```

```
% Intersection:<X0 X1 ... Xn ; Y0 Y1 ... Ym ; ;> == XIY  % initial call
% Intersection:<X0 X1 ... Xn ; Y0 Y1 ... Ym ; XIY ;>     % recursive call
%
% arg names
X1: <> == Arg1.
Y1: <> == Arg2.
XIY:<> == Arg3.   % X intersection Y

Intersection:
   <;> == XIY
   <$terminal> == <If:<Member:<$terminal ; Y1 ; !> > $terminal>
   <then $terminal> == Intersection:< X1:<> ; Y1:<> ; XIY:<> $terminal ; !>
   <else $terminal> == Intersection:<>.

% --- Some theorems -----
%
% Intersection: <1 2 3 ; 1 2 3 ; ;> = (1 2 3)
%
```