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 descriptive domain of any DATR theory is finite (and hence enumerable), to develop methods for representing certain forms of negation, disjunction, and constraints. This permits us to impose constraints on path values, and we are thus able to extend the usual domain of DATR descriptions of feature values to descriptions of descriptions of feature values. In this respect we are able to impose meta-level constraints.

[1] To avoid confusion, we adopt the following terminological conventions: we refer to <p_1 p_2> as a *path extension* of <p_1>. Where <p_1> is the longest defined prefix of <p_1 p_2>, we refer to p_2 as a *default extension* of <p_1>. The 'extension of a path' (its value) we refer to as its *extensional value*, or simply its *value*.

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 hierarchy itself. In particular, lexical *signs*[4] (or entries) defined with respect to a type subsumption hierarchy must satisfy type constraints corresponding to the type hierarchy. When such constraints are applied to a phrasal sign they are extra-lexical, or grammatical, constraints. A trivial type of lexical constraint would be that a lexical item must have a value for the feature `<spelling>`. This constraint might be inherited from some higher node (in the lexicon), and any lexical item can be shown to satisfy it (or not, in which case it is invalid). Other constraints are more local in the sense that they constrain only the feature-value pairs of the node at which they are defined.[5] A feature `<gender>` might be required to have a value in the set $\{male, female, neuter\}$. Such constraints might be particularly useful if a DATR lexicon was to be generated automatically from a machine readable dictionary. Assuming that such entries could be used as a source for a computational lexicon (not a bold assumption – *e.g.*, Boguraev & Briscoe (1989)), entries would be presumed to have various properties, but it is likely that a large dictionary would have exceptions, and checking the constructed nodes for constraint satisfaction would be helpful. Such constraints are *lexical* – they apply to a lexical entry in isolation and can be verified without reference to the context in which a lexical item occurs.

Any constraint which relates to context of use we call *grammatical*; constraints of this type can be represented in DATR as constants (for theories of grammar which posit such constraints as emanating from lexical entries). For example, linear precedence rules in Flickinger's theory (1987, pp. 22,85-87) derive from the lexicon, but must be evaluated by an external constraint system. Subcategorization properties of lexical entries are another grammatical constraint (in HPSG), such as a common noun subcategorizing for a determiner or a genitive noun phrase. Again this disjunction can only be evaluated in context. We have little to say about such constraints, other than they should be statable in DATR in some form which can be 'passed up' to the external user of the lexicon. Nevertheless, they may sometimes be evaluated in the lexicon when some contextual arguments are 'passed down'. For example, the external constraint system may have to evaluate whether two words can co-occur. It is conceivable that a query to the lexicon could be of the form "Does Y satisfy X's constraints?" Thus grammatical constraints which could be evaluated by the lexical component would require instantiation from elsewhere.

This work is an exploration of the ability of DATR to represent the types of constraints used in computational lexicons. The formalism itself is theory independent, and it is not our intention to produce a translation of any particular grammatical theory. We develop a classical logic (following Gazdar (1990)) for illustrating constraint combination.

Negative Path Extension

Negative path extension (NPE) is based primarily on a trivial observation: when a general rule and an exception are both available and a given pattern matches (or selects) the general case,

[2] By 'external' we mean external to the lexicon.

[3] Specifically, separate feature structures must be *unifiable*.

[4] By 'sign' we mean a feature structure corresponding to a linguistic entity.

[5] The notion of local constraints – constraints a node imposes on itself – does not seem intuitively useful, unless, as we envision, the constraints are inserted at the node by some extra-nodal entity, *e.g.*, as part of a translation into DATR from some other lexical representation.

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>` which is defined.

Using negative path extension, we can determine where an arbitrary path constituent intervenes in some sequence of enumerated constant paths. Below we have a theory which determines the location at which some arbitrary value intervenes in the sequence of four `b`'s:

```
A: <b b b b> == nowhere
   <b b b> == 4
   <b b> == 3
   <b> == 2
   <> == 1.
```

Some theorems of `A` are:

```
A: <x> = 1
   <b y x b b> = 2
   <b b x y b> = 3
   <b b b x> = 4
   <b b b b x> = nowhere.
```

We call this negative path extension since `A:<b b x y b>` has value `3` because it *does not* extend the longer prefix `<b b b>`. The prefixes defined at `A` are called *selectors*. It is important to note that the value of `A:<b b x y b>` is not dependent on the actual values `x`, and `y` merely that they differ from `b`.

Evaluable paths

Generally, the domain of path constituents (or features) which appear inside of path defining angle brackets `<` and `>` is non-overlapping with the domain of extensional value constituents (appearing on the right hand side of `=` and `==`). Both types of constituents are simply *atoms*. One would not normally expect to derive such statements as

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

However, it is not required that these domains be mutually exclusive, and DATR provides a mechanism for interpreting a value as a path constituent, or a sequence of values as a sequence of path constituents. For example, node `D` below defines its default using an evaluable path, one constituent of which (the value of `A`) happens to be a sequence.

[6] Here, and in subsequent single-node examples, there is an implicit assumption that the single node comprises the entire theory.

```
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.
```

Notice that now we have generalized the testing of equality to include testing of arguments both of which require evaluation. However, `constraint3` is not a meta-level constraint that paths `<x>` and `<y>` must share – *i.e.*, it does not enforce a requirement that `<x>` unify with `<y>` (and hence share token equality for all extensions). On the other hand for every path p, it is satisfied whenever `<x` p`>` = `<y` p`>` holds (*i.e.*, it can test type equality for any extension).

[7]It is still not quite the intuitive notion, since `Equal:<13 13 ; hi mom> = true` remains derivable. We are ignoring path extensions following the semicolon, which, at least in intention, delimits the interesting part of the query path.

[8]Selectors need not be restricted to atoms. A prefix such as `<a b c ; a b c ;>` would behave similarly.

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 choose the alternative, and say that a constraint is satisfied if it is not violated – and constraints tested against undefined values are either satisfied, if the value is optional, or violated if the value is obligatory.[10] We will show how such constraints can be represented, beginning with logical operators `And`, `Or`, and `Not`. Binary `And` could be defined by enumerating its truth table as follows:

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

With this definition we can state conjunctive constraints:

```
D1: <constraints> == And:<<c1> <c2>>
    <c1> == Equal:<13 ; <x>>
    <c2> == Equal:<14 ; <y>>
    <x> == 13
    <y> == 14.
```

[9]It would not alter the behaviour of the operators we define in this paper if they *were* included in `$terminal`. The operations would merely apply to a larger set of atoms. We define `$terminal` to exclude these other atoms merely to isolate the atoms of interest.

[10]Gazdar (1990, pp. 126-127) illustrates various non-classical logics represented in DATR. The presentation of the logic differs slightly from that given by Gazdar, where all of the predicates were defined at a single node.

The path <constraints> is defined in terms of an evaluable path, each constituent of which evaluates to a boolean value. It becomes And:<true true> when instantiated (in this particular example), hence we have the theorem:

```
D1:<constraints> = true.
```

Since And evaluates to a boolean value, it can itself appear as one of the constituents of a path on which it is invoked. Omitting for the moment the definitions of <c1>, <c2>, etc., one definition of a compound constraint with four conjuncts is:

```
D2: <constraint3> == And:<And:< <c1> <c2>>
                          And:< <c3> <c4>>>.
```

To avoid such repetitive descriptions, we refine the definition of And to be polyadic (following Gazdar (1990)) but add the delimiter ; to terminate the sequence of conjuncts:

```
And: <;> == true
     <true>   == <>
     <false>  == false.
```

This allows <constraint3> to be defined more elegantly as:

```
D3: <constraint3> == And:<<c1> <c2> <c3> <c4> ; >.
```

In a similar vein we define Or, also polyadic, and Not:

```
Or: <;> == false
    <true> == true
    <false> == <>.

Not: <true> == false
     <false> == true.
```

We can compose these logical operators to form any well-formed boolean expression as a constraint. For example:

```
D4: <c1> == And:< Or:<Equal:<<x> ; 1 ;>
                      Equal:<<x> ; 2 ;>
                      Equal:<<x> ; 3 ;>
                      ;>
                  Not:<Equal:<<y> ; 4 ;>>
                  ;
                >.
```

In particular, we have a mechanism for testing set membership and a combination of disjunction and equality, although the statement of the definition is tedious. The first conjunct of <c1> is true whenever <x> $\in \{1,2,3\}$. We will develop a concise definition of set membership presently, but first we turn our attention to the function of the delimiter of the polyadic And and Or.

The purpose of the delimiter is twofold. First, it avoids the issue of whether a zero-length sequence of conjuncts (*i.e.*, And:<>) well formed. Of more importance, it permits the logical operator to be used on arbitrary path extensions, in the same way that Equal:<<x> ; <y> ;> can be used to test equality of extensions of paths <x> and <y>. Consider, for example, evaluation of D3:<constraint3 y>. If we assume that <c1 y>, ..., <c4 y> all evaluate to true, then the evaluable path instantiates to And:<true true true true ; y>; the default extension, which is not a boolean value (or in our parlance, not a member of $terminal), becomes a suffix of the path evaluated at And. As noted by Gibbon (1990), a delimiter (such as ;) can be interpreted as a 'cut', by mapping it and the default extension which follows it, to an empty sequence, allowing for elegant function composition, which is precisely what we have done at the definition of D4:<c1>.

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.
```

This yields theorem `D6:<c5> = true.` As with implication, we can factor out a test for set membership to a single node. Our definition of `Member` requires an enumeration of all possible sequences over `$terminal` of length 2. We do this by defining a new DATR variable, `$terminal2 = $terminal`, and taking their cross-product.

```
Member:
   <$terminal ; $terminal> == true
   <$terminal ; ;> == false
   <$terminal ; > == <reduce $terminal ;>
   <reduce $terminal ; $terminal2> == <$terminal ;>.
```

With this definition,

```
Member:<<val> ; <list> ;>
```

will evaluate to `true` whenever (the value of) `<val>` is in (the value of) `<list>`, assuming that `<val>` evaluates to a suitable selector (in this case a `$terminal`). The definition of `Member` uses NPE: the first case selects those paths where the first item in the list is the value to be matched; the second where the list is exhausted; and the third is the default, where the first item in the list is not the required element, nor is the list exhausted. So we cycle through `<reduce>`, removing the first symbol of the list, back to the initial case.[11] With this definition set membership can be tested transparently – `D6`, for example, can be redefined as `D7`:

```
D7: <c5> == Member:<<person> ; 1 2 3 ;>
    <person> == 2.
```

The theorem `D7:<c5> = true` is derivable, as with the previous definition (of `D6:<c5>`).

[11] It might seem that a less complicated definition would suffice, *viz.*:

```
Member:
  <$terminal ; $terminal> == true
  <$terminal ; ;> == false
  <$terminal ; $terminal2> == <$terminal ;>.
```

This is *not* an equivalent definition; it is not even semantically well-formed, as the set `$terminal` $\cap$ `$terminal2` $\neq \emptyset$.

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 $X = x_1x_2\ldots x_n$ and $Y = y_1y_2\ldots y_m$, then we define `Subset:<` x_1 x_2 … x_n `;` y_1 y_2 … 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 $y_1y_2\ldots y_m$ `;`:

```
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 $x_0x_1\ldots x_n \subseteq y_0y_1\ldots y_m$ 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.

Path constraints versus node constraints

The constraints discussed thus far can be categorized into two classes: *path* constraints, and *node* constraints. Consider nodes `Parent3` and `Child3`:

```
Parent3: <> == undef
   <constraints> == And:<<c1> <c2> ;>
   <c1 agr> == Equal:<"<agr>" ; "<comp subject agr>" ;>
   <c1> == true
   <c2> == Is_defined:<"<oblig_path>">.

Child3: <> == undef
   <constraints> == Parent3
   <agr> == A
   <comp subject agr> == B
   <oblig_path> == 3.
```

Constraints `<c1>` and `<c2>` differ in that `<c1>` can be applied to arbitrary extensions of its defined path. For example, if `Child3:<agr` p`>` and `Child3:<comp subject agr` p`>` have the same value, then the statement `Child3:<constraints agr` p`> = true` is derivable. However, if `<`p`>` does not extend `<agr>`, evaluation of `Child3:<constraints` p`>` will cause `Parent3:<Is_defined:<"oblig_path` p`>">` to be evaluated, and it is not clear that this is a 'sensible' question. `<c2>` is a node constraint, as it does not apply to paths whose values depend on default extension (of paths defined at the node).

Just as we defined `$terminal` to enumerate all of the terminals in the theory, we can assume a variable `$feat` which enumerates all of the atoms in the theory, and we can strip off arbitrary suffixes of any path, guarding the condition `<c2>` against meaningless extensions:[15]

```
Parent2:
   <c2 $feat> == true
   <c2> == Is_defined:<"<oblig_path>">.
```

Generalizing this method of extension stripping, we define a node which strips off the suffix following the first atom in a path, and uses global inheritance to evaluate the path of length 1 at the context node. We call this node `Length_1_eval`:

```
Length_1_eval:
   <$feat $feat2> == <$feat>
   <$feat> == Is_defined:<"<$feat>">.
```

This definition is interesting because `Length_1_eval:<`$f_1 f_2 \ldots f_n$`> = "<`f_1`>"` for any atom f_1.

Summary

In this paper we have addressed the expressivity of DATR for specifying and testing constraints on the extensional values of paths. Relying on the fact that every DATR theory comprises a finite

[15] None of the machinery developed depends on `$terminal` excluding atoms which appear only in paths. Indeed, we only require that `$terminal` include the alphabet of the descriptive domain of the theory (those which appear in value sequences). `$feat` might be a larger set since there is no requirement that path constituents be introduced before they appear in a query. Thus the single statement theory

```
A:<x> == 1.
```

permits the inference

```
A:<x y> = 1.
```

although the atom `y` does not appear in the statement of the theory.

For the purposes of the present discussion we assume that `$feat` (and `$feat2`) serve as declarations of the alphabet of path descriptions, and include at least `$terminal`. In particular, we exclude queries of paths defined using 'undeclared' atoms.

number of atoms, we have expressed constraints as tests over an enumeration of them. We used DATR variables to make the definitions short, but the expansion of the statements defined in terms of these variables is finite and could be expressed without the use of variables.

The notation used to express the constraints is cumbersome, because DATR is not designed to support the expression of constraints as we have done. Nevertheless, it is interesting that they *can* be expressed in the language, because if DATR were used as a target language for a compilation of a a lexicon, such constraints could be embedded in the output to allow for verification of the constructed lexicon. In particular, any large scale automatically constructed lexicon is likely to contain exceptions to whatever expectations are made about the lexical entries, (*e.g.*, fields expected to be present on words of various types) and the mechanisms illustrated would draw attention to them – so that the expectations could be modified or the inconsistencies deleted.

References

[Boguraev and Briscoe 1989] Branimir K. Boguraev and Edward J. Briscoe, editors. *Computational Lexicography for Natural Language Processing.* Longman, Harlow, 1989.

[Carpenter 1990] Bob Carpenter. Typed feature structures: inheritance, (in)equality and extensionality. In Walter Daelemans and Gerald Gazdar, editors, *Inheritance in Natural Language Processing: Workshop Proceedings*, pages 9–18. Institute for Language Technology and AI, Tilburg University, The Netherlands, 1990.

[Evans and Gazdar 1990] Roger Evans and Gerald Gazdar, editors. *The DATR Papers, Volume I.* CSRP 139. School of Cognitive and Computing Sciences, University of Sussex, Brighton, UK, 1990.

[Evans *et al.* 1991] Roger Evans, Gerald Gazdar, and Lionel Moser. Prioritised multiple inheritance in DATR. In Ted Briscoe, Ann Copestake, and Valeria de Paiva, editors, *Proceedings of the ACQUILEX Workshop on Default Inheritance in the Lexicon.* Technical Report 238, The Computer Laboratory, Cambridge University, Cambridge, UK, 1991.

[Flickinger 1987] Dan Flickinger. *Lexical Rules in the Hierarchical Lexicon.* PhD thesis, Stanford University, 1987.

[Gazdar *et al.* 1985] Gerald Gazdar, Ewan Klein, Geoffrey K. Pullum, and Ivan Sag. *Generalized Phrase Structure Grammar.* Blackwell, Oxford, 1985.

[Gazdar 1990] Gerald Gazdar. quantifi.dtr. *The DATR Papers, Volume I*, pages 131–132, 1990. Formal DATR example file illustrating first order quantification over attributes.

[Gibbon 1990] Dafydd Gibbon. register.dtr. *The DATR Papers, Volume I*, pages 133–134, 1990. Formal DATR example file illustrating binary shift operations.

[Moser 1991] Lionel Moser. Multiple inheritance in DATR: A quick tour. In Richard Dallaway, Teresa del Soldato, and Lionel Moser, editors, *The Fourth White House Papers: Graduate Research in the Cognitive & Computing Sciences at Sussex*, Technical Report CSRP 200. School of Cognitive & Computing Sciences, University of Sussex, Brighton, UK, 1991.

[Moser 1992a] Lionel Moser. DATR paths as arguments. Technical Report CSRP 215, School of Cognitive & Computing Sciences, University of Sussex, Brighton, UK, 1992.

[Moser 1992b] Lionel Moser. More multiple inheritance in DATR. School of Cognitive & Computing Sciences, University of Sussex, Brighton, manuscript, 1992.

[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'.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                      %
% File:            ctheory.dtr                                         %
% Purpose:         Illustrate constraint logic based on the primitives %
%                  in clogic.dtr.                                      %
% Author:          Lionel Moser, September 1991                        %
% Documentation:   HELP *datr                                          %
% Related Files:   lib datr; args.dtr; clogic.dtr                      %
% Version:         6.00                                                %
%     Copyright (c) University of Sussex 1991.  All rights reserved.   %
%                                                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Negation, disjunction, equality, inequality, and obligatory and
% forbidden features as constraints are illustrated, base on the
% primitives defined in clogic.dtr, sets.dtr and arglogic.dtr.

#vars $terminal: singular plural alpha beta gamma 1 2 3 undef.

#load 'args.dtr'.
#load 'arglogic.dtr'.
#load 'clogic.dtr'.
#load 'sets.dtr'.

% Introduce constraint that something is not 3rd person singular,
% and <person> must have an appropriate value, if it's defined.
NOT_3RD_SG:
   <> == undef
   <constraints> == And:<<c0> <c1> ;>
   <c0> == Implies:<And:<Is_defined:<"<number>">
                         Is_defined:<"<person>"> ; >
                    Not:<And:<Equal:<"<person>" ; 3 ;>
                              Equal:<"<number>" ; singular ;> ; >
                        >
                   >
   <c1> == Implies:<Is_defined:<"<person>">
                    Member:<"<person>" ; 1 2 3 ;> >.

% B violates NOT_3RD_SG's constraints.
B: <> == NOT_3RD_SG
   <person> == 3
   <number> == singular.

% C satisfies NOT_3RD_SG's
% constraints.
C: <> == NOT_3RD_SG
   <person> == 1
   <number> == singular.
```

```
% -------- 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> == Safe:<>

   % This is a default for inheriting nodes.
   <local_constraints> == ()

   % Three constraints are introduced at this node.
   % In general, a node's local constraints must be ANDed in; if there aren't
   % any then make them satisfied.
   <constraints> == And:< "<local_constraints>" <c1> <c2> <c3> ;>

   % Constraint c1: disjunction
   %    <person> is in set {1,2,3}; i.e., member(<person>,{1,2,3}).
   <c1> == Member:<"<person>" ; 1 2 3 ;>

   % Constraint c2: obligatory feature
   %    <number> must be defined.
   <c2> == Is_defined:<"<number>">

   % Constraint c3: forbidden feature
   %    <alpha> must NOT be defined.
   <c3> == Not:<Is_defined:<"<alpha>">>.

% K has inherited constraints and local constraints, all of which are
% satisfied.
K: <> == ParentType
   <local_constraints> == <c0>
   <c0> == Member:<<person> ; 1 2 ;>
   <person> == 2
   <number> == singular.

% L violates its constraints - <alpha> is defined.
% satisfied.
L: <> == ParentType
   <person> == 2
   <number> == singular
   <alpha> == 3.
```

```
% 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: <constraints> = true
%     <local_constraints> = true
%     <person> = 2
%     <constraints person> = true
%     <safe person> = 2
%     <number> = singular
%     <safe number> = singular.

%  L: <constraints> = false
%     <local_constraints> = ()
%     <person> = 2
%     <constraints person> = false
%     <safe person> = constraint violation
%     <number> = singular
%     <constraints number> = false
%     <safe number> = constraint violation.

%  M: <constraints> = false
%     <local_constraints> = ()
%     <person> = 3
%     <constraints person> = false
%     <safe person> = constraint violation
%     <number> = undef
%     <constraints number> = false
%     <safe number> = constraint violation.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                      %
% File:            sets.dtr                                            %
% Purpose:         Define set operators.                               %
% Author:          Lionel Moser, December, 1991                        %
% Documentation:   HELP *datr                                          %
% Related Files:   lib datr; args.dtr                                  %
% Version:         6.00                                                %
%      Copyright (c) University of Sussex 1991.  All rights reserved.  %
%                                                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#vars $terminal:  1 2 3 4 5 6 7 8 9 0.
#vars $terminal2: 1 2 3 4 5 6 7 8 9 0.

#load 'args.dtr'.
#load 'arglogic.dtr'.  % If, Equal, Tequal
#load 'clogic.dtr'.    % And, Or, Not

% Member:<atom ; list_of_atoms ;> ==
%       true    -   if item is in list,
%       false   -   if atom is not in list.
%
% Some theorems:
%    Member: <1 ; ;> = false
%            <1 ; 1 2 3 ;> = true
%            <1 ; 2 3 ;> = false
%            <1 ; 2 1 3 ;> = true
%            <1 ; 2 3 1 ;> = true
%
% Member:<<val> ; <list> ;>
Member:
   <$terminal ; $terminal> == true
   <$terminal ; ;> == false
   <$terminal ; > == <reduce $terminal ;>
   <reduce $terminal ; $terminal2> == <$terminal ;>.
```

```
% 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)
%               <; 1 2 3 ; ;> = ()
%               <1 2 3 ; 2 3 ; ;> = (2 3)
%               <1 2 3 4 5 ; ; ;> = ()
%               <1 3 5 ; 2 4 6 ; ;> = ()
%               <1 3 5 ; 2 4 3 6 ; ;> = (3).


% Setequal:<Set1 ; Set2 ; !> ==
%     true  - if Set1 and Set2 are the same
%     false - otherwise

Setequal:
   <> == <If:<Subset>>                        % Set1 <= Set2?
   <then> == Subset:<Arg2:<> ; Arg1:<> ; !>    % Set2 <= Set1?
   <else> == false.

% Some Theorems

%  Setequal: <; ;> = true
%           <1 ; 1 ;> = true
%           <1 2 ; 1 2 ;> = true
%           <3 2 ; 2 3 ;> = true
%           <1 2 3 4 5 ; 5 4 3 2 1 ;> = true.
```