Content Planning for CCG

A Graph Transformation Toolkit

Bernd Kiefer
Bernd.Kiefer@dfki.de

Deutsches Forschungszentrum für künstliche Intelligenz

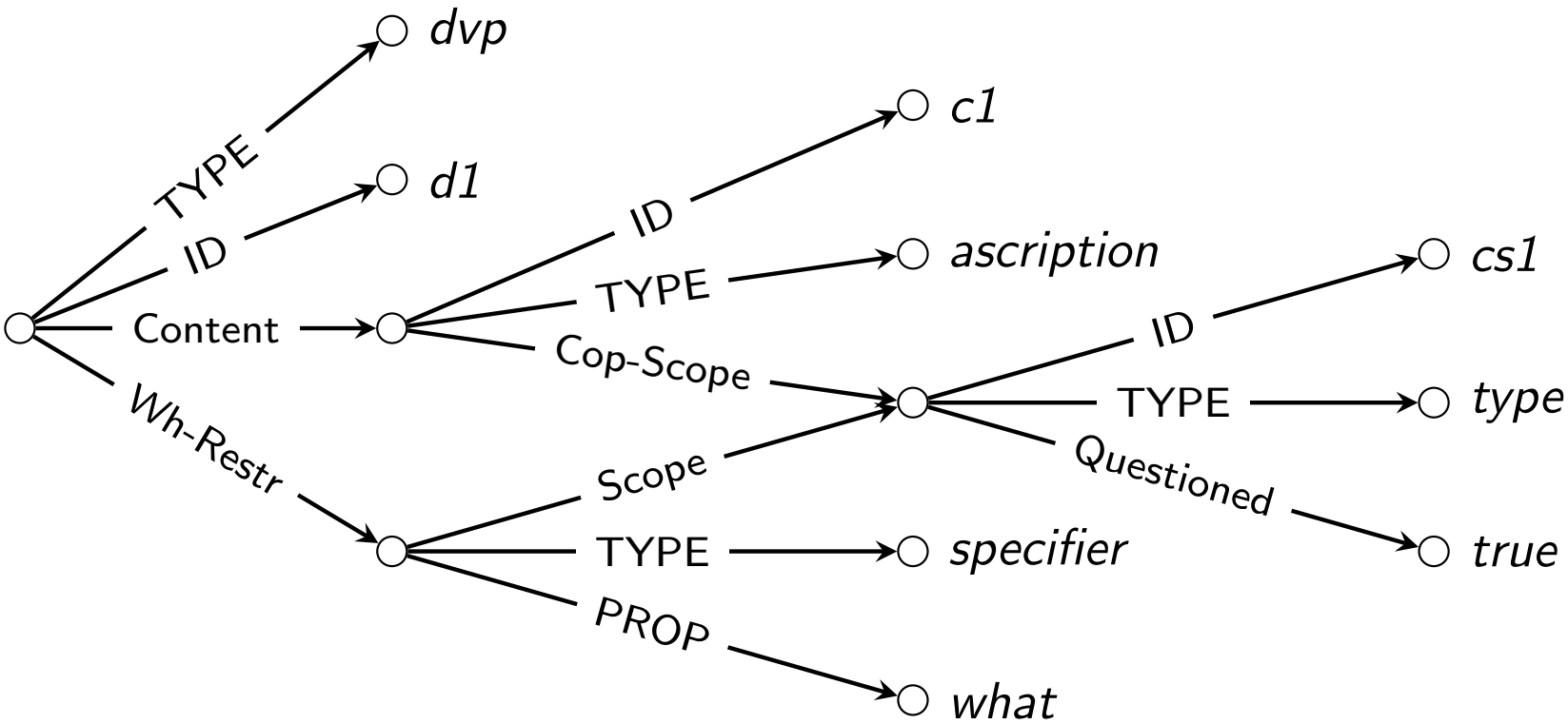
November 8, 2010

Overview

Logical Forms as Graphs

A CCG logical form can be interpreted as an acyclic graph:

```
@d1:dvp(<Content>(c1:ascription ^
                  <Cop-Scope>(cs1:type ^ <Questioned>true))
        <Wh-Restr>(:specifier ^ what ^ <Scope> (cs1:type)))
```



Content Planning as Graph Transformation

The Content Planner as a Rule System

The Planner consists of

- A set of transformation rules that specify the local modifications
- A processing engine that applies the rules to all sub-parts of the logical form with a specific strategy

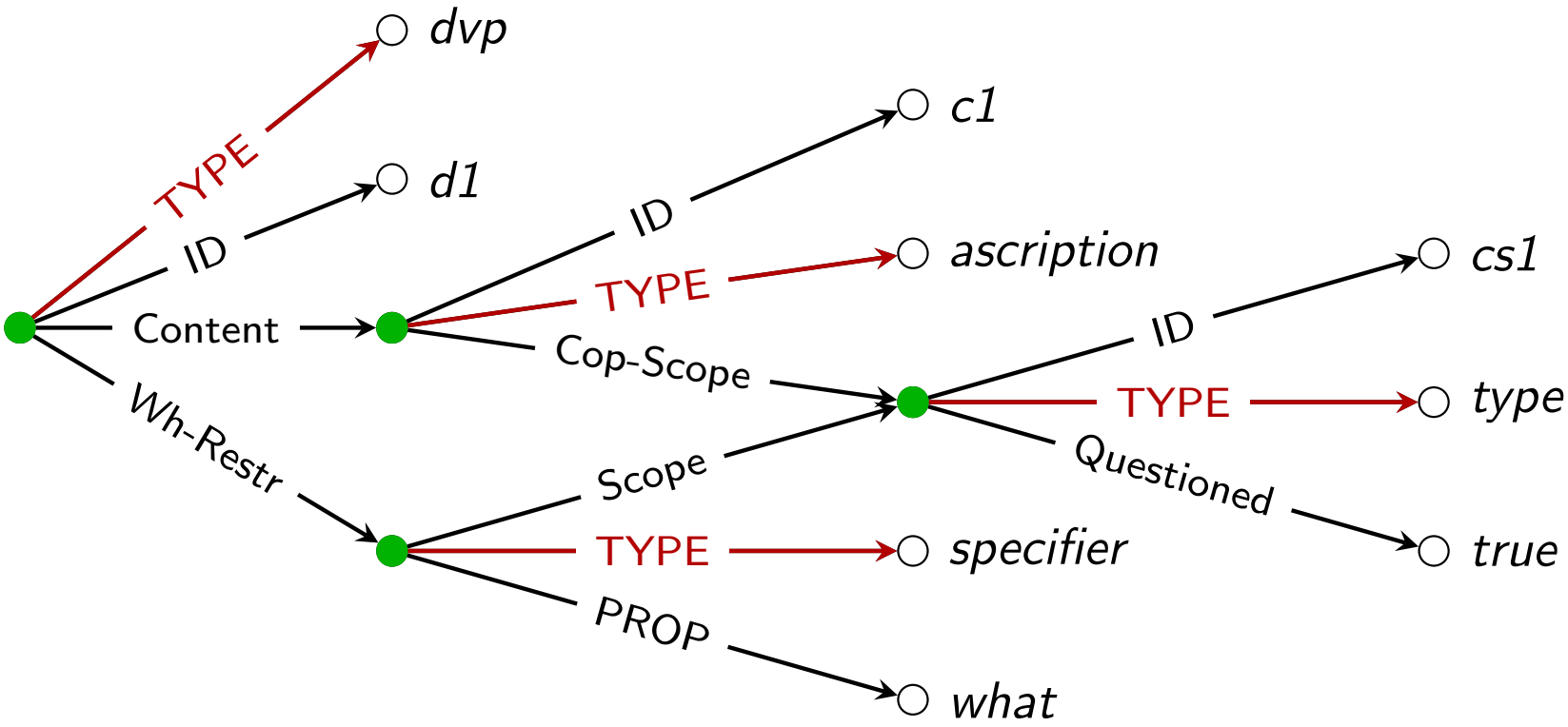
Transformation rules: Antecedent and Consequent

- *Match Part* specify what shape some substructure of the graph must exhibit to make it applicable to a rule
- *Action Part* specify the modifications to parts of the substructure

Step 1: Match Patterns

Specify patterns where certain actions should be applied

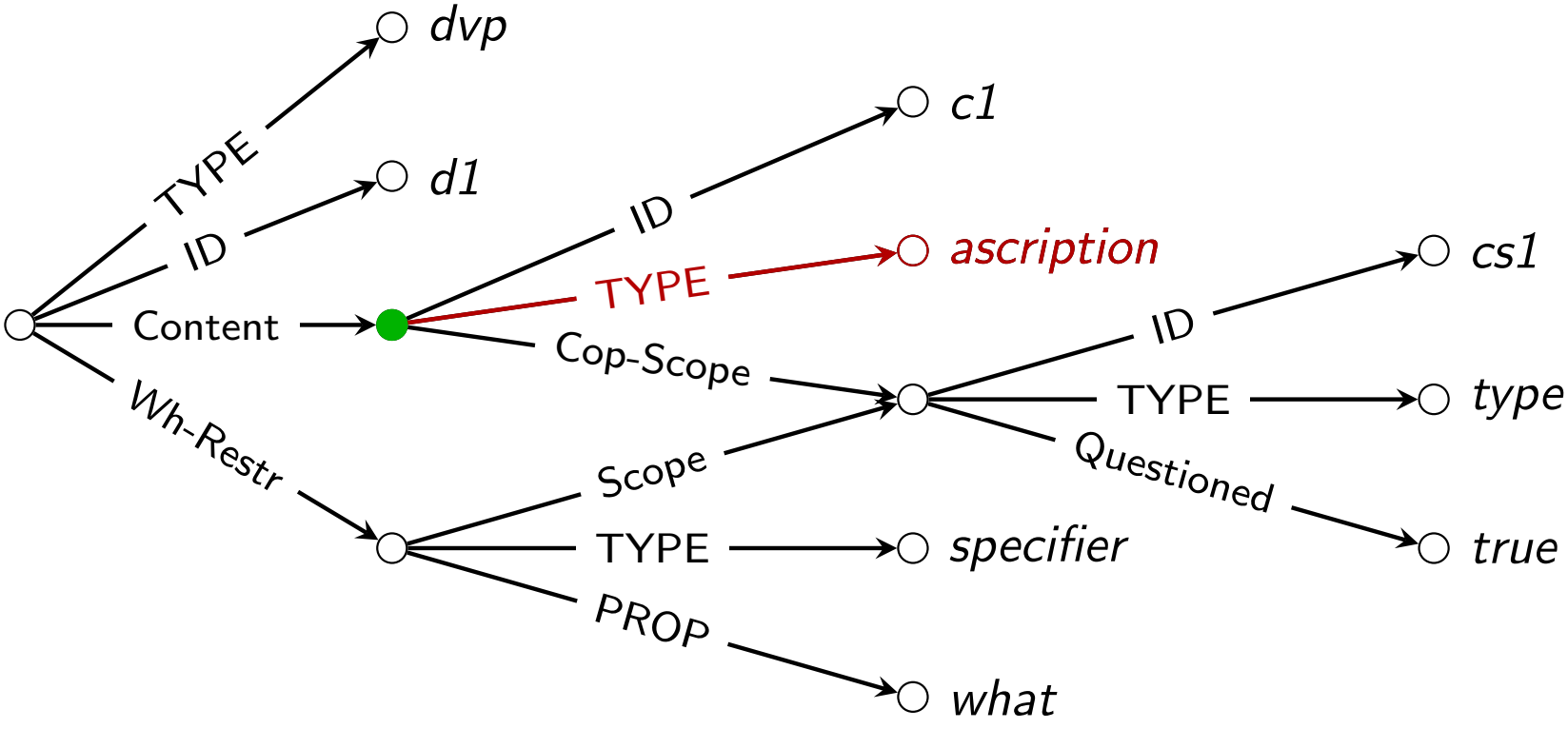
<TYPE> (activate all nodes that have an outgoing TYPE edge)



Step 1: Match Patterns

Specify patterns where certain actions should be applied

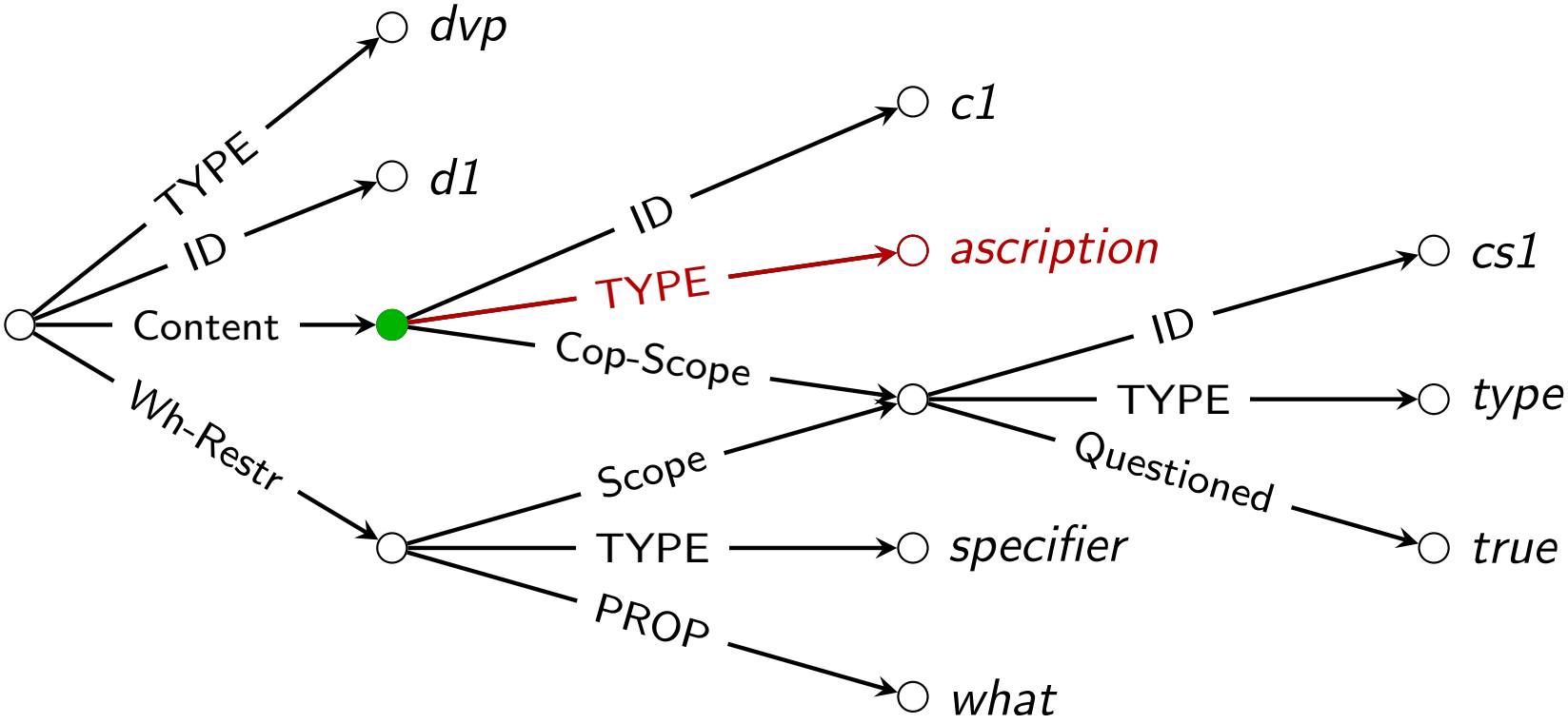
`<TYPE> ascription` (alternatively: `:ascription`)



Step 2: Actions

How should the active node be modified

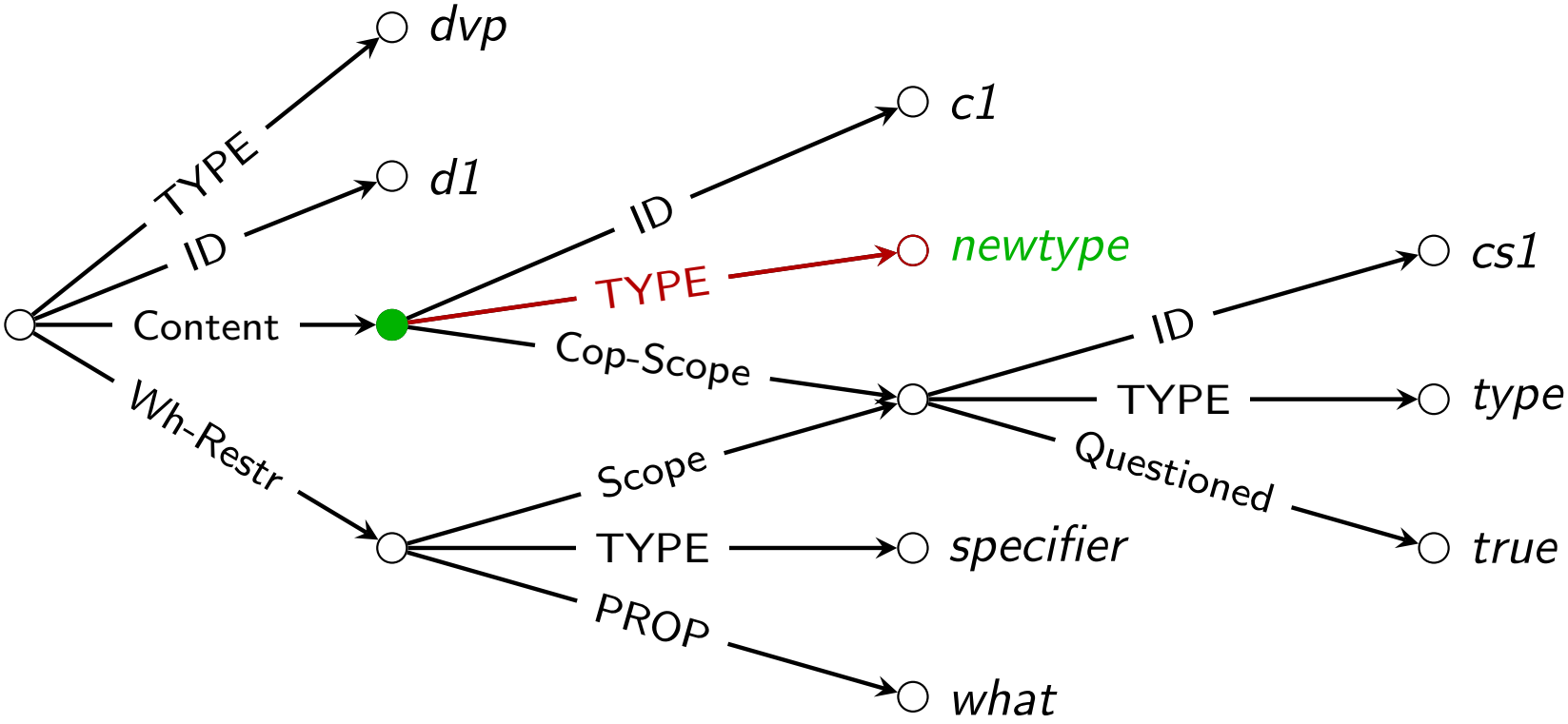
`# ^ :newtype` overwrites the type (shorthand for `# ^ <TYPE>newtype`)



Step 2: Actions

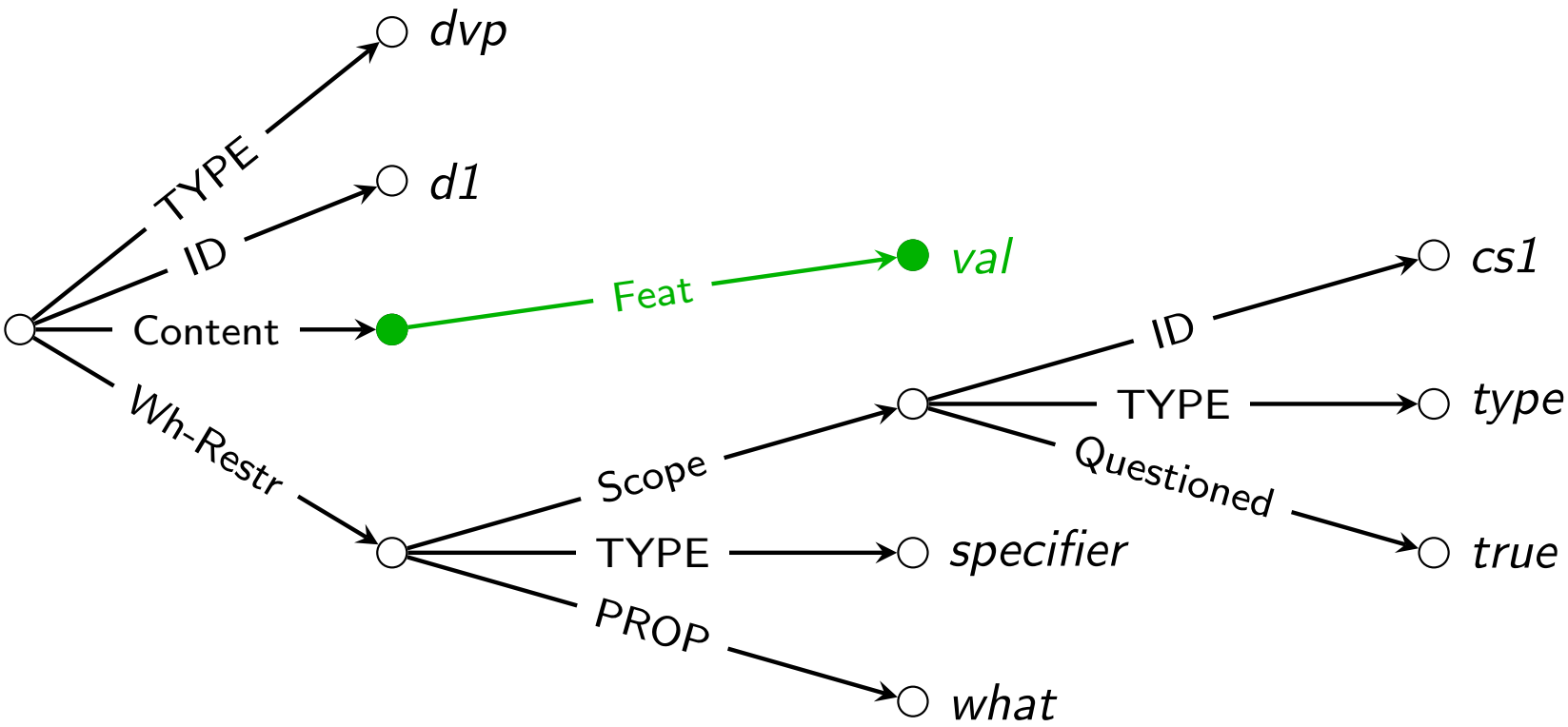
How should the active node be modified

`# ^ :newtype` overwrites the type (shorthand for `# ^ <TYPE>newtype`)



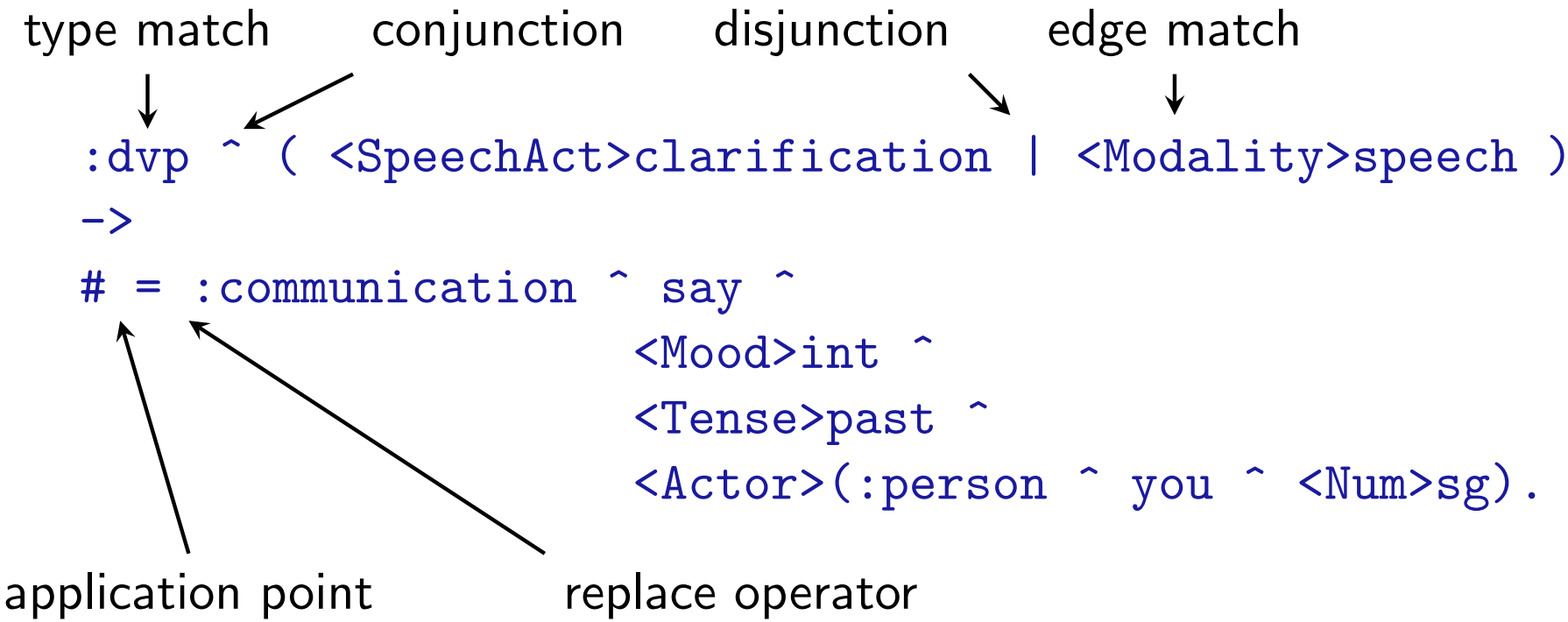
Step 2: Actions

`#` = `<Feat>` `val` replaces active node with a new node having only a type attribute



Simple Rule Example

Match (‘,’ Match) ‘->’ Action (‘,’ Action)* ‘.’*



Rule Structure

Match Part

- *Required:* a match specification for the current node
- Optional: a comma-separated list of match specifications against global variables or function calls.
  - Starts with the global variable or function call and an ‘ ^ ’ to indicate matching
  - followed by the (possibly complex) match expression in the same way as for the current node

Action Part: List of comma-separated action specifications

- application point: a local variable or # for the current node, or a global variable, optionally followed by a feature path (*global var map*)
- operator: addition ‘ ^ ’, replacement ‘=’, or deletion ‘ ! ’
- the addition or replacement, or a single feature for deletion

Rule Details – Atomic Matches

Value Matches

- `<Feat>` exists an outgoing edge with name `Feat`
- `prop` is the proposition or feature value equal to `prop`
- `:type` is the type value equal to `type`
- `id:` is the name of the nominal equal to `id`

Rule Details – Matches with Variables

Binding Local Variables

- `#var:` bind the *whole node* to the local variable `var`
- `:#var` bind the *type* to the local variable `var`
- `#var` bind the *feature or proposition value* to the local variable `var`

Tests with bound Variables

- If a local variable is already bound, the match succeeds if what has been bound is *structurally equal* to the match node
- `= #var:` check if the matched node is coreferent to what is bound to `#var`
- *Bound* global variables can be used for tests in the same way (if unbound $\Rightarrow$ failure)

Rule Details – Complex Matches

- Operators:
  - conjunction ‘ ^ ’
  - disjunction ‘ | ’
  - negation ‘ ! ’
- Grouping with parentheses
- short forms for some matches:

  `id:type` $\equiv$ `id:^:type`

  `#idvar:#typevar` $\equiv$ `#idvar:^:#typevar`

  `<Feat>val` tests for existence of `Feat` *and* value `val`

Rule Details – Variables

Local Variables

- have local scope in one rule
- can only be bound *once* on the left hand side during the matching phase
- if mentioned in the actions, the variable is replaced by its bound value

Global Variables

- can only be set or modified as separate action on the right hand side of a rule
- lifetime ends only when processing of the input structure is finished
- test special conditions on the left hand side
- use as replacement or addition value on the right hand side

Rule Details – Variables II

Right Hand Side Local Variables

- have local scope in one rule
- can only occur on the right hand side
- can be bound *once* on the right hand side either as application point, or to establish coreferences in an replacement
- if mentioned in the actions, the variable is replaced by its bound value
- are distinguished from local variables to detect binding errors

Restrictions on matching

- Note the difference of matching a whole node vs. type or preposition
- Currently: No matching of nodes or sub-nodes against variables or function calls

Rule Details – Actions

- First Element of an action: the point in the graph the action is applied to
  - # to specify the current match node
  - A local, global or right-local variable
  - global variables can be followed by a feature path to get a map-like structure
- Second Element: modification operation: addition/overwriting ‘ ^ ’, replacement ‘=’, or deletion ‘ ! ’
- Third Element: A partial logical form, possibly containing variables and function calls whose content will be evaluated before the change is applied
- More than one action is possible, separate by comma

Some Examples

Sub-node is replaced with fresh content. Notice the match condition for the absence of a feature

```
:dvp ^ ! <AcknoModality> ^ <SpeechAct> assertion
     ^ <Relation> accept ^ <Content> ( #c1: )
->
#c1 = (:marker ^ ok).
```

Selecting a specific relation, and adding to it

```
<C>(<Mod>(#m:g ^ ! <Cont>)) -> #m ^ <Cont>(:new ^ clean).
```

Some Examples

Add a default in case of feature absence

```
:ascription ^ !<Tense> -> # ^ <Tense>pres.
```

Set a global variable as marker

```
:dvp ^ <SpeechAct>#v -> ##speechact = #v.
```

test for the global variable (multiple matches)

```
:ascription ^ <Tense>, ##speechact ^ assertion
->
# ^ <Mood>ind.
```

Some Examples

type disjunction, variable t matching the whole node
add two relations, delete Target feature (multiple actions)

```
:ascription ^ <Target> #t:(entity | thing)
->
# ^ <Cop-Restr>(#t:) ^ <Subject>(#t:),
# ! <Target>.
```

Less preferable, the same variable name has to be used twice!
Works only because of disjunction.

```
:ascription ^ (<Target> (#t:entity) | <Target> (#t:thing))
->
# ^ <Cop-Restr>(#t:), # ^ <Subject>(#t:),
# ! <Target>.
```

Some Examples

Randomizing with complex values, using right-local variables

```
:dvp ^ <SpeechAct>opening ^ <Content> (#c1:top)
->
###opening1 = :opening ^ hi,
###opening2 = :opening ^ hello,
# ! <SpeechAct>,
#c1 = random(###opening1, ###opening2): .
```

Note the colon after the function call! It means that the whole node is the value, not just the proposition.

Some Examples

Alternative randomization

```
:dvp ^ <SpeechAct>closing ^ <Content> (#c1:top)
->
##randomclosing = random(1,2).

:dvp ^ <SpeechAct>closing ^ <Content> (#c1:top),
##randomclosing ^ 1
->
#c1 = :closing ^ bye.

:dvp ^ <SpeechAct>closing ^ <Content> (#c1:top),
##randomclosing ^ 2
->
#c1 = :closing ^ see_you.
```

Some Examples

Using global variable as node store
Again, note the colon after the global variable in the addition

```
:ascription ^ <Target> (#t:testtype)
->
##fromStore = #t:.

:ascription ^ !<PointToTarget>
->
# ^ <PointToTarget> ##fromStore:.
```

After application of these rules `Target` and `PointToTarget` point to the same node.

Some Examples

Adding to relations the wrong way

```
:dvp ^ <SpeechAct>question
     ^ <Content>(#cont:ascription ^
                 <Subject>(:entity ^
                           <Delimitation>unique ^
                           <Quantification>specific) ^
                 <Cop-Scope>(#cop-scope: ^ <Questioned>true))
->
#cont ^ <Wh-Restr>(:specifier ^ what ^ <Scope> #cop-scope:)
      ^ <Subject>( context ^ <Proximity> proximal )
      ^ <Cop-Scope>(<Delimitation>unique ^
                    <Quantification>specific ^ <Num> sg),
#cop-scope ! <Questioned>.
```

This will not add to the existing Subject and Cop-Scope, but introduce new ones.

Some Examples

Adding to relations: the correct alternative

```
:dvp ^ <SpeechAct>question
     ^ <Content>(#cont:ascription ^
                 <Subject>(#subj:entity ^
                           <Delimitation>unique ^
                           <Quantification>specific) ^
                 <Cop-Scope>(#cop-scope: ^ <Questioned>true))
->
#cont ^ <Wh-Restr>(:specifier ^ what ^ <Scope> #cop-scope:),
#subj ^ context ^ <Proximity> proximal,
#cop-scope ! <Questioned>,
#cop-scope ^ <Delimitation>unique ^
             <Quantification>specific ^ <Num> sg.
```

This adds the new information to the previously matched nodes.

Some Examples

Global Variable Maps

```
:dvp ^ <Content> #c: -> ##gvar<Content> = #c:.

:dvp ^ !<Cont2>, ##gvar ^ <Content> #v:
->
# ^ <Cont2> ( #v: ).
```

Or, alternatively for the second rule:

```
:dvp ^ !<Cont2>, ##gvar ^ <Content>
->
# ^ <Cont2> ( ##gvar<Content>: ).
```

This is syntactic sugar which is only allowed with global variables in the current version. The previous version allowed paths for all application points, which is dangerous and should never have been in the syntax.

Some Examples

All you can do with local variables

```
:a ^ <F> (#i:#t ^ #p)
->
##partial = :#t ^ #p,
##whole = #i:.

:a ^ !<W>
->
# ^ <W> ##whole: ^ <P> ##partial:.
```

Be careful that you use bound variables correctly! If you use them as simple (type or proposition) values on the right hand side, you must have bound them to simple values, or complex values containing the appropriate edge!

Some Examples

All you can do with variables II

```
:a ^ ! <Subject>
->
# ^ <Subject> ###s: ^ <WhRestr> ###s:.
```

Right-local variables can also be used to establish coreferences in the replacement part.

All you can do with variables: equality checks

Check for structural equality (will also succeed if coreferent)

```
:dvp ^ <Content>#c: ^ <Wh-Restr>#c: -> # ^ :equals.
```

Check for node identity (coreference)

```
:dvp ^ <Content>#c: ^ <Wh-Restr> = #c: -> # ^ identical.
```

Processing Strategy

- Outer loop: fix point computation, i.e., retry until no more changes occur.
- One step of the loop:
  - Match the rules with all subnodes of the dag, and store all successful matches together with the established local variable bindings
  - The nodes are traversed in post-order, the rules in the order in which they were loaded
  - Apply the actions in the same order in which the matches were found

Consequences of processing strategy

- Values may be changed/overwritten multiple times in one iteration, i.e., former effects can be removed again
  - On the same level
  - On a lower level
- Actions may be applied to already deleted material
- Since relations may occur several times, addition of relations must be done very carefully to avoid infinite loops

The New Content Planner

Goal: Self-Explanatory Rules

The specification of the rules should be such that is it (quite) obvious

- what they are applied to and under which conditions
- what the effect will be if the rules are applied
- all rules work *locally* (well, almost)
- all that without having to resort to idiosyncratic names

The New Content Planner

Problems

- Actions are not always adding information monotonically $\Rightarrow$ order-dependence of rule application
- Formalism of CCG LFs allows multiple relations with the same name leaving one node:
  - Selection of matched edge becomes ambiguous
  - No proper notion of unification
  - Equality and Subsumption hard to implement
  - Danger of infinite additions
- distinction in LF formalism for types and propositions propagated to the rule formalism, not sure if it adds to clarity
- Because of nonmonotonicity, cumulated effects are not that obvious and depend on the processing strategy