The Meta-ACG Preprocessor

Valentin D. Richard

7 March 2022

My situation

My PhD goal : **ACG grammar** of French interrogatives in discourse and dialogue

- Using ACGtk

My situation

My PhD goal : **ACG grammar** of French interrogatives in discourse and dialogue

- Using ACGtk
- Using **feature structures** (FS) :
  - agreement
  - anaphora resolution

My situation

My PhD goal : **ACG grammar** of French interrogatives in discourse and dialogue

- Using ACGtk
- Using **feature structures** (FS) :
  - agreement
  - anaphora resolution
- Deep syntax inspired by HPSG [**?**]
  - SLASH feature for wh-phrase extraction

Why a preprocessor ?

Problem : Feature structures not (yet) implemented in ACGtk

Why a preprocessor ?

Problem : Feature structures not (yet) implemented in ACGtk

Solution : Simulate features as multiple atomic types, like in [**?**]
e.g. `np[3,sg]` as `np_3_sg`

Why a preprocessor ?

Problem : Feature structures not (yet) implemented in ACGtk

Solution : Simulate features as multiple atomic types, like in [**?**]
e.g. `np[3,sg]` as `np_3_sg`

Other problem : Combinatorial explosion

- too many possibilities to keep track of by hand
- many risks of dumb mistakes

Why a preprocessor ?

Problem : Feature structures not (yet) implemented in ACGtk

Solution : Simulate features as multiple atomic types, like in [**?**]
e.g. `np[3,sg]` as `np_3_sg`

Other problem : Combinatorial explosion

- too many possibilities to keep track of by hand
- many risks of dumb mistakes

Proposal : **Meta-ACG preprocessor** to automate this process

Building a preprocessor

Building a preprocessor

Usefulness :

1. **Reduce redundancy**
2. **Simulate feature structures**

ACGtk bureaucracy

Standard ACGtk grammar :

deep syntax

```
signature Deep =
  n : type ;
  PARK : n ;
end
```

ACGtk bureaucracy

Standard ACGtk grammar :

deep syntax

```
signature Deep =
  n : type ;
  PARK : n ;
end
```

surface signature

```
signature Surface =
  o : type ;
  string = o -> o : type ;
  .
  :
  park : string ;
end
```

ACGtk bureaucracy

Standard ACGtk grammar :

deep syntax

```
signature Deep =
  n : type ;
  PARK : n ;
end
```

surface signature

```
signature Surface =
  o : type ;
  string = o -> o : type ;
  .
  :
  park : string ;
end
```

surface syntax

```
lexicon syntax (Deep) : Surface =
  n := string ;
  PARK := park ;
end
```

(and similar for semantics)

ACGtk bureaucracy

Standard ACGtk grammar :

deep syntax

```
signature Deep =
  n : type ;
  PARK : n ;
end
```

surface signature

```
signature Surface =
  o : type ;
  string = o -> o : type ;
  .
  :
  park : string ;
end
```

surface syntax

```
lexicon syntax (Deep) : Surface =
  n := string ;
  PARK := park ;
end
```

(and similar for semantics)

- Expected parts (conventional)

ACGtk bureaucracy

Standard ACGtk grammar :

deep syntax

```
signature Deep =
  n : type ;
  PARK : n ;
end
```

surface signature

```
signature Surface =
  o : type ;
  string = o -> o : type ;
  .
  :
  park : string ;
end
```

surface syntax

```
lexicon syntax (Deep) : Surface =
  n := string ;
  PARK := park ;
end
```

(and similar for semantics)

- Expected parts (conventional)
- Predictable parts

ACGtk bureaucracy

Standard ACGtk grammar :

deep syntax

```
signature Deep =
  n : type ;
  PARK : n ;
end
```

surface signature

```
signature Surface =
  o : type ;
  string = o -> o : type ;
  .
  :
  park : string ;
end
```

surface syntax

```
lexicon syntax (Deep) : Surface =
  n := string ;
  PARK := park ;
end
```

(and similar for semantics)

- Expected parts (conventional)
- Predictable parts
- **What really matters**

Syntax of input file

MACG syntax :

```
Command: param
  declaration
  declaration
Command: param
  declaration
⋮
```

Available commands :

- **Type** : mandatory `param` (the type name)
- **Constant**
- **Rule** : CFG rule

Syntax of input file

MACG syntax :

```
Command: param
  declaration
  declaration
Command: param
  declaration
⋮
```

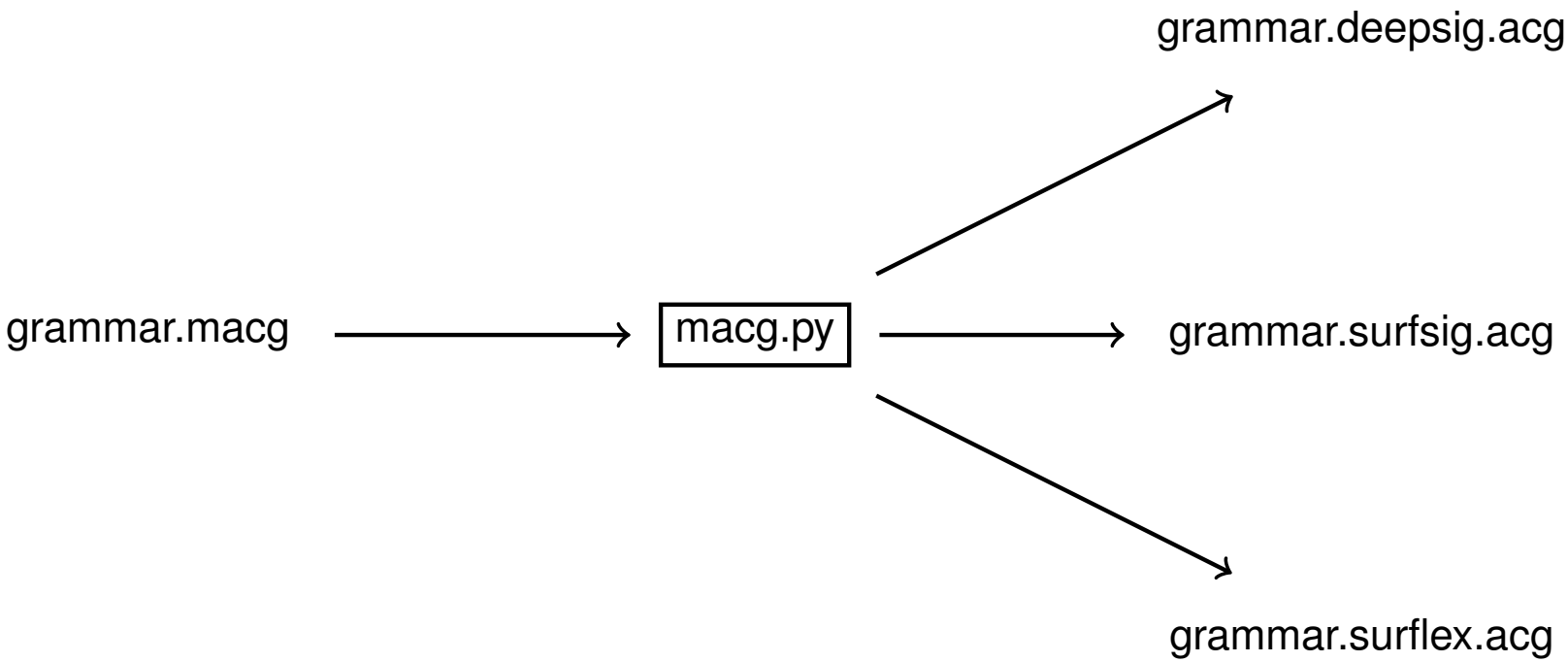
Example :

```
Type: np
Type: vp
Type: s


Constant:
  John, Mary : np
  slept : vp


Rule:
  np -> vp -> s
```

Available commands :

- **Type** : mandatory `param` (the type name)
- **Constant**
- **Rule** : CFG rule

Preprocessing



Python program

Limitations

How much should be able to be customized ?

- ☑ Common surface word, e.g.

```
THAT_det (that) : det ;
THAT_comp (that) : comp
```

Limitations

How much should be able to be customized ?

- ☑ Common surface word, e.g.
  ```
  THAT_det (that) : det ;
  THAT_comp (that) : comp
  ```
- ☐ Special type map, e.g.
  ```
  n := (e -> t) ;
  ```
- ☐ Special constant map, e.g. function word semantics
  ```
  SOME := lambda p, q. (Ex x. (p x) & (q x)) ;
  ```

Limitations

How much should be able to be customized ?

- ☑ Common surface word, e.g.
  ```
  THAT_det (that) : det ;
  THAT_comp (that) : comp
  ```
- ☐ Special type map, e.g.
  ```
  n := (e -> t) ;
  ```
- ☐ Special constant map, e.g. function word semantics
  ```
  SOME := lambda p, q. (Ex x. (p x) & (q x)) ;
  ```
- ? Customizable signature / lexicon names

Typed feature structures

Feature structure (FS) = set of pairs **attribute-value**

$$\text{sleeps} : \begin{bmatrix} \textit{verb} & \\ \text{AGR} & \begin{bmatrix} \textit{agreement} & \\ \text{PERSON} & 3 \\ \text{NUMBER} & \textit{singular} \end{bmatrix} \\ \text{TENSE} & \textit{present} \end{bmatrix} \tag{1}$$

Typed feature structures

Feature structure (FS) = set of pairs **attribute-value**

$$\text{sleeps}: \begin{bmatrix} \textit{verb} & \\ \text{AGR} & \begin{bmatrix} \textit{agreement} & \\ \text{PERSON} & 3 \\ \text{NUMBER} & \textit{singular} \end{bmatrix} \\ \text{TENSE} & \textit{present} \end{bmatrix} \quad (1)$$

- Values can be **atomic** or **sub-FS**

Typed feature structures

Feature structure (FS) = set of pairs **attribute-value**

$$\text{sleeps} : \begin{bmatrix} \textit{verb} & \\ \text{AGR} & \begin{bmatrix} \textit{agreement} & \\ \text{PERSON} & 3 \\ \text{NUMBER} & \textit{singular} \end{bmatrix} \\ \text{TENSE} & \textit{present} \end{bmatrix} \quad (1)$$

- Values can be **atomic** or **sub-FS**
- **Typed** : each FS has predefined set of possible attributes and values

FS class :

$$\begin{bmatrix} \textit{agreement} & \\ \text{PERSON} & \{1, 2, 3\} \\ \text{NUMBER} & \{\textit{singular}, \textit{plural}\} \end{bmatrix} \quad (2)$$

FS instance :

$$\begin{bmatrix} \textit{agreement} & \\ \text{PERSON} & 3 \end{bmatrix} \quad (3)$$

Typed feature structures

Feature structure (FS) = set of pairs **attribute-value**

$$\text{sleeps} : \begin{bmatrix} \textit{verb} & \\ \text{AGR} & \begin{bmatrix} \textit{agreement} & \\ \text{PERSON} & 3 \\ \text{NUMBER} & \textit{singular} \end{bmatrix} \\ \text{TENSE} & \textit{present} \end{bmatrix} \quad (1)$$

- Values can be **atomic** or **sub-FS**
- **Typed** : each FS has predefined set of possible attributes and values
- **Underspecification** : some attributes may lack

FS class :

$$\begin{bmatrix} \textit{agreement} & \\ \text{PERSON} & \{1, 2, 3\} \\ \text{NUMBER} & \{\textit{singular}, \textit{plural}\} \end{bmatrix} \quad (2)$$

FS instance :

$$\begin{bmatrix} \textit{agreement} & \\ \text{PERSON} & 3 \end{bmatrix} \quad (3)$$

Subsumption and unification

Operations on FSs f, f'

f **subsumes** f' is they have the same type and f' specifies f more

e.g.

$$\begin{bmatrix} \textit{agreement} \\ \text{PERSON} \quad 1 \end{bmatrix} \sqsubseteq \begin{bmatrix} \textit{agreement} \\ \text{PERSON} \quad 1 \\ \text{NUMBER} \quad \textit{singular} \end{bmatrix},$$

Subsumption and unification

Operations on FSs f, f'

f **subsumes** f' is they have the same type and f' specifies f more

e.g.

$$\begin{bmatrix} \textit{agreement} \\ \text{PERSON} \quad 1 \end{bmatrix} \sqsubseteq \begin{bmatrix} \textit{agreement} \\ \text{PERSON} \quad 1 \\ \text{NUMBER} \quad \textit{singular} \end{bmatrix}, \begin{bmatrix} \textit{agreement} \\ \text{PERSON} \quad 2 \end{bmatrix} \not\sqsubseteq \begin{bmatrix} \textit{agreement} \\ \text{PERSON} \quad 1 \\ \text{NUMBER} \quad \textit{singular} \end{bmatrix}$$

Subsumption and unification

Operations on FSs f, f'

f **subsumes** f' is they have the same type and f' specifies f more

e.g.

$$\begin{bmatrix} \textit{agreement} \\ \text{PERSON} \quad 1 \end{bmatrix} \sqsubseteq \begin{bmatrix} \textit{agreement} \\ \text{PERSON} \quad 1 \\ \text{NUMBER} \quad \textit{singular} \end{bmatrix}, \begin{bmatrix} \textit{agreement} \\ \text{PERSON} \quad 2 \end{bmatrix} \not\sqsubseteq \begin{bmatrix} \textit{agreement} \\ \text{PERSON} \quad 1 \\ \text{NUMBER} \quad \textit{singular} \end{bmatrix}$$

The **unification** of f and f' is the less specific FS g such that $f \sqsubseteq g$ and $f' \sqsubseteq g$ (if it exists)

e.g.

$$\begin{bmatrix} \textit{agreement} \\ \text{PERSON} \quad 1 \end{bmatrix} \sqcup \begin{bmatrix} \textit{agreement} \\ \text{NUMBER} \quad \textit{singular} \end{bmatrix} = \begin{bmatrix} \textit{agreement} \\ \text{PERSON} \quad \textit{1} \\ \text{NUMBER} \quad \textit{singular} \end{bmatrix}$$

Subsumption and unification

Operations on FSs f, f'

f **subsumes** f' is they have the same type and f' specifies f more

e.g.

$$\begin{bmatrix} \textit{agreement} \\ \text{PERSON} \quad 1 \end{bmatrix} \sqsubseteq \begin{bmatrix} \textit{agreement} \\ \text{PERSON} \quad 1 \\ \text{NUMBER} \quad \textit{singular} \end{bmatrix}, \begin{bmatrix} \textit{agreement} \\ \text{PERSON} \quad 2 \end{bmatrix} \not\sqsubseteq \begin{bmatrix} \textit{agreement} \\ \text{PERSON} \quad 1 \\ \text{NUMBER} \quad \textit{singular} \end{bmatrix}$$

The **unification** of f and f' is the less specific FS g such that $f \sqsubseteq g$ and $f' \sqsubseteq g$ (if it exists)

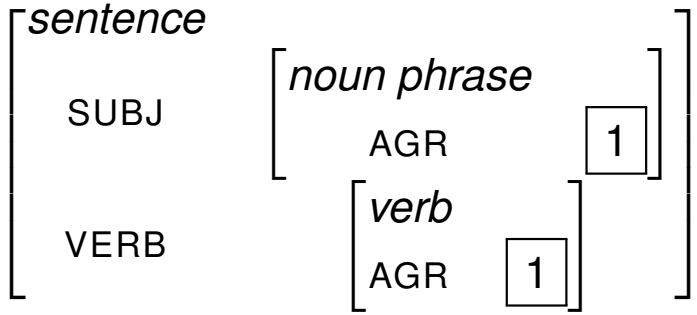
e.g.

$$\begin{bmatrix} \textit{agreement} \\ \text{PERSON} \quad 1 \end{bmatrix} \sqcup \begin{bmatrix} \textit{agreement} \\ \text{NUMBER} \quad \textit{singular} \end{bmatrix} = \begin{bmatrix} \textit{agreement} \\ \text{PERSON} \quad \textit{1} \\ \text{NUMBER} \quad \textit{singular} \end{bmatrix}$$

but $\begin{bmatrix} \textit{agreement} \\ \text{PERSON} \quad 1 \end{bmatrix}$ and $\begin{bmatrix} \textit{agreement} \\ \text{PERSON} \quad 2 \end{bmatrix}$ can't unify

Unification in rules

E.g. The AGR values of the subject and the verb have to unify



Unification in rules

E.g. The AGR values of the subject and the verb have to unify

$$\begin{bmatrix} \textit{sentence} & & \\ \text{SUBJ} & \begin{bmatrix} \textit{noun phrase} & \\ \text{AGR} & \boxed{1} \end{bmatrix} \\ \text{VERB} & \begin{bmatrix} \textit{verb} & \\ \text{AGR} & \boxed{1} \end{bmatrix} \end{bmatrix}$$

$$\begin{bmatrix} \textit{noun phrase} & & \\ \text{AGR} & \boxed{2} & \begin{bmatrix} \textit{agreement} & \\ \text{PERSON} & 2 \end{bmatrix} \end{bmatrix}$$

you

$$\begin{bmatrix} \textit{verb} & & \\ \text{AGR} & \boxed{3} & \begin{bmatrix} \textit{agreement} \\ \textit{past} \end{bmatrix} \\ \text{TENSE} & & \end{bmatrix}$$

slept

Unification in rules

E.g. The AGR values of the subject and the verb have to unify

$$\begin{bmatrix} \textit{sentence} & & \\ \text{SUBJ} & \begin{bmatrix} \textit{noun phrase} & \\ \text{AGR} & \boxed{1} \end{bmatrix} \\ \text{VERB} & \begin{bmatrix} \textit{verb} & \\ \text{AGR} & \boxed{1} \end{bmatrix} \end{bmatrix}$$

Parsing is **findind** $\boxed{1}$ such that $\boxed{1} = \boxed{2} \sqcup \boxed{3}$

$$\begin{bmatrix} \textit{noun phrase} & & \\ \text{AGR} & \boxed{2} & \begin{bmatrix} \textit{agreement} & \\ \text{PERSON} & 2 \end{bmatrix} \end{bmatrix}$$

you

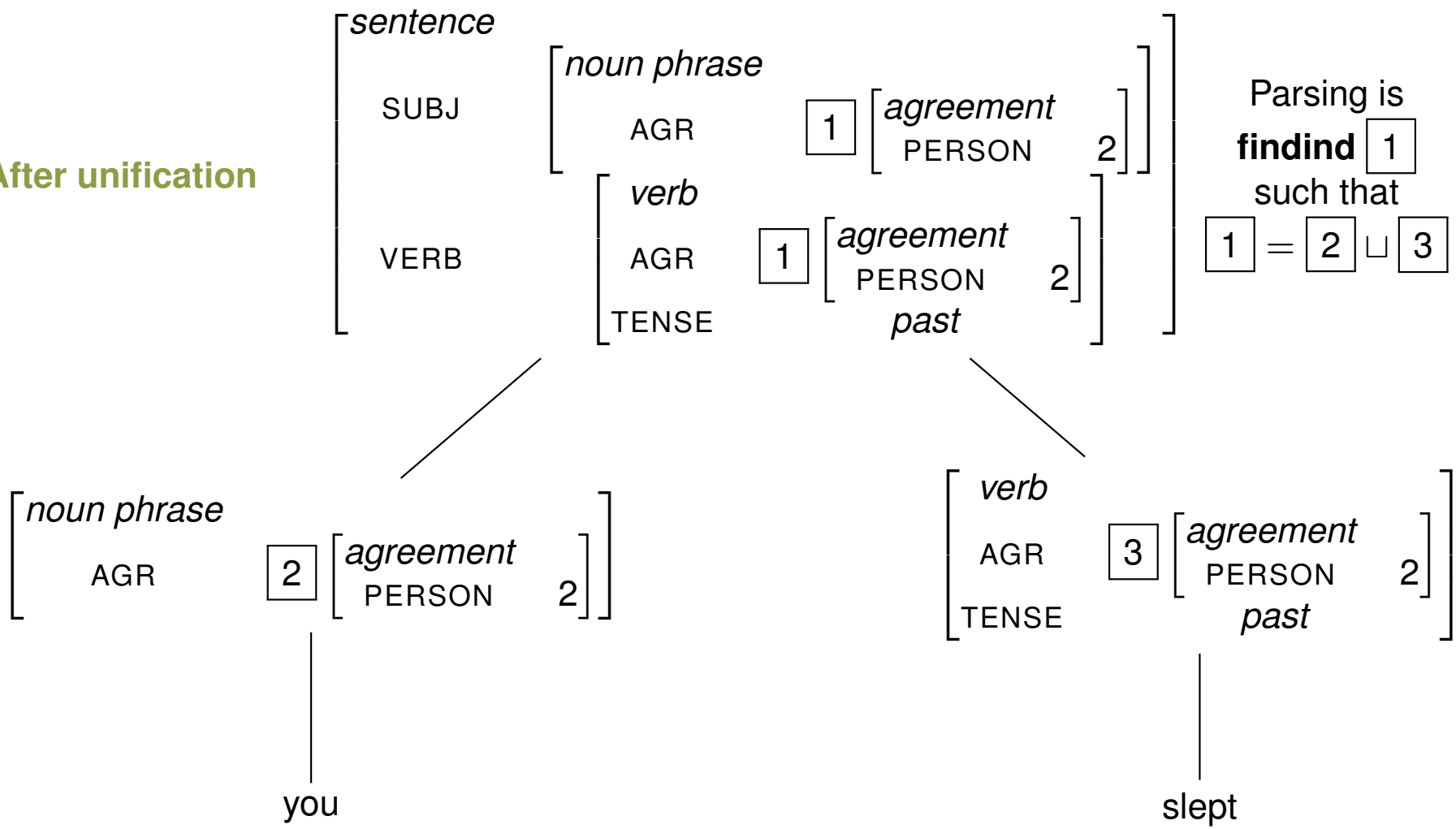
$$\begin{bmatrix} \textit{verb} & & \\ \text{AGR} & \boxed{3} & \begin{bmatrix} \textit{agreement} \\ \textit{past} \end{bmatrix} \\ \text{TENSE} & & \end{bmatrix}$$

slept

Unification in rules

E.g. The AGR values of the subject and the verb have to unify

After unification



Simulating FSs with simple types

Simulating FSs with simple types

As many atomic types as possible FSs : **full instances**

FS class

$$\begin{bmatrix} v & \\ \text{AGR} & \begin{bmatrix} agr & \\ \text{P} & \{1,2,3\} \\ \text{N} & \{sg, pl\} \end{bmatrix} \\ \text{T} & \{prst, past\} \end{bmatrix}$$

Simulating FSs with simple types

As many atomic types as possible FSs : **full instances**

FS class

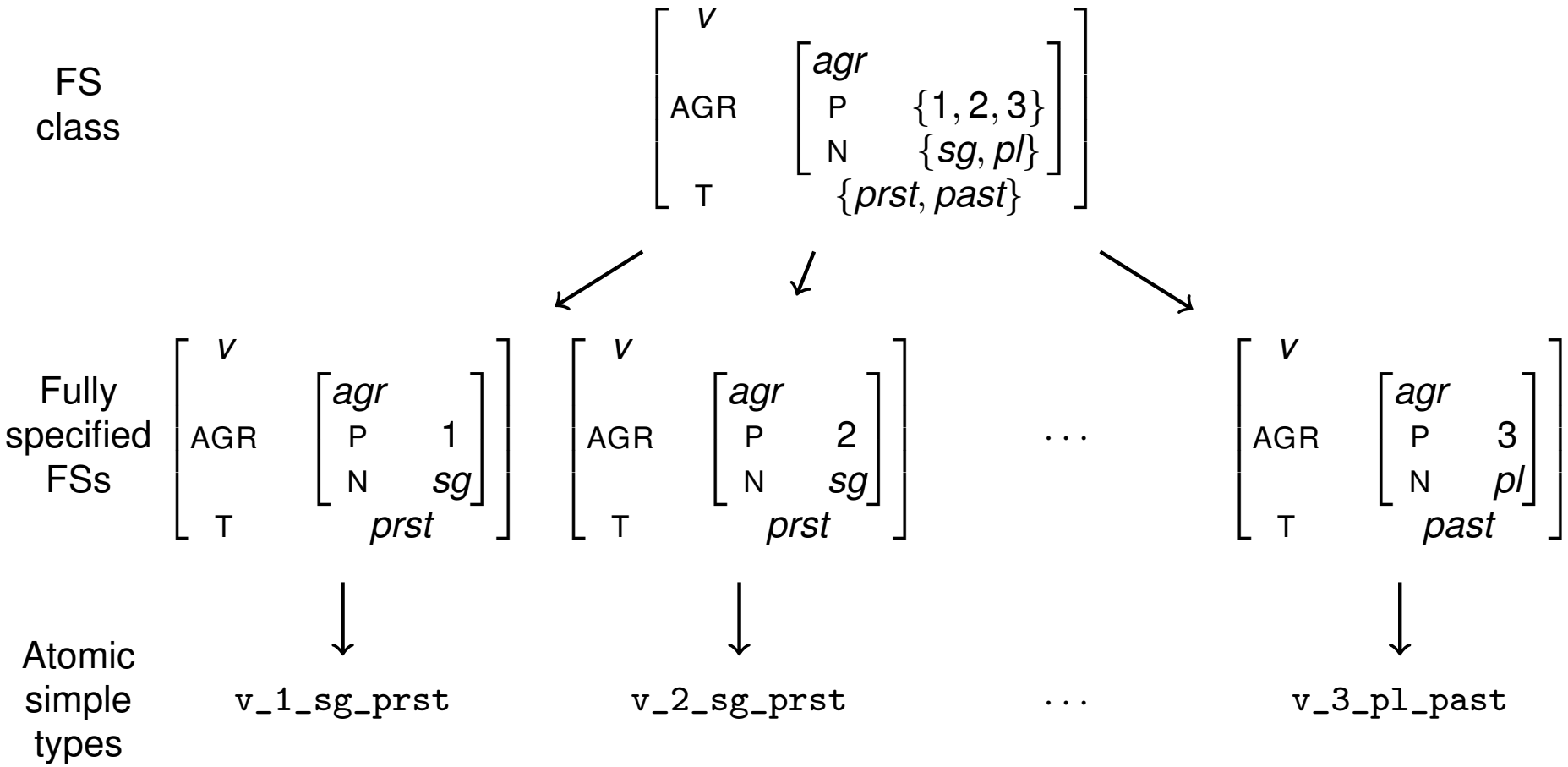
$$\begin{bmatrix} v & \\ \text{AGR} & \begin{bmatrix} agr & \\ \text{P} & \{1,2,3\} \\ \text{N} & \{sg, pl\} \end{bmatrix} \\ \text{T} & \{prst, past\} \end{bmatrix}$$

Fully specified FSs

$$\begin{bmatrix} v & \\ \text{AGR} & \begin{bmatrix} agr & \\ \text{P} & 1 \\ \text{N} & sg \end{bmatrix} \\ \text{T} & prst \end{bmatrix} \quad \begin{bmatrix} v & \\ \text{AGR} & \begin{bmatrix} agr & \\ \text{P} & 2 \\ \text{N} & sg \end{bmatrix} \\ \text{T} & prst \end{bmatrix} \quad \cdots \quad \begin{bmatrix} v & \\ \text{AGR} & \begin{bmatrix} agr & \\ \text{P} & 3 \\ \text{N} & pl \end{bmatrix} \\ \text{T} & past \end{bmatrix}$$

Simulating FSs with simple types

As many atomic types as possible FSs : **full instances**



Implementation

MACG syntax like NLTK (python)

```
sleeps : v[AGR = agr[P=3, N=sg], T = prst]
```

Implementation

MACG syntax like NLTK (python)

```
sleeps : v[AGR = agr[P=3, N=sg], T = prst]
```

Rules with variables over atomic subFSs :

```
np[AGR = @a] -> v[AGR = @a, T = @t] -> s[T = @t]
```

As many rule instances as they are possible values of `@a` and `@t`

Implementation

MACG syntax like NLTK (python)

```
sleeps : v[AGR = agr[P=3, N=sg], T = prst]
```

Rules with variables over atomic subFSs :

```
np[AGR = @a] -> v[AGR = @a, T = @t] -> s[T = @t]
```

As many rule instances as they are possible values of `@a` and `@t`

Type classes must be defined before !

```
Type: agr
  P : 1, 2, 3
  N : sg, pl
Type: vp
  AGR : agr
  T : prst, past
Type: s
  T : prst, past
```

Implementation

MACG syntax like NLTK (python)

```
sleeps : v[AGR = agr[P=3, N=sg], T = prst]
```

Rules with variables over atomic subFSs :

```
np[AGR = @a] -> v[AGR = @a, T = @t] -> s[T = @t]
```

As many rule instances as they are possible values of `@a` and `@t`

Type classes must be defined before !

```
Type: agr
  P : 1, 2, 3
  N : sg, pl
Type: vp
  AGR : agr
  T : prst, past
Type: s
  T : prst, past
```

Additional functionalities :

- boolean features, e.g. `is : v[+AUX]`
- Absence of a feature, e.g. no agreement `be : v[-AGR, T=inf]`

Formal proof of work

Given (underspecified) FSs f, f' :

$$\mathrm{FI}(f) = \{g \mid f \sqsubseteq g \text{ and } g \text{ is fully specified}\} \tag{4}$$

Formal proof of work

Given (underspecified) FSs f, f' :

$$\mathrm{FI}(f) = \{g \mid f \sqsubseteq g \text{ and } g \text{ is fully specified}\} \tag{4}$$

Property

$$f \text{ and } f' \text{ unify} \quad \text{iff} \quad \mathrm{FI}(f) \cap \mathrm{FI}(f') \neq \emptyset$$

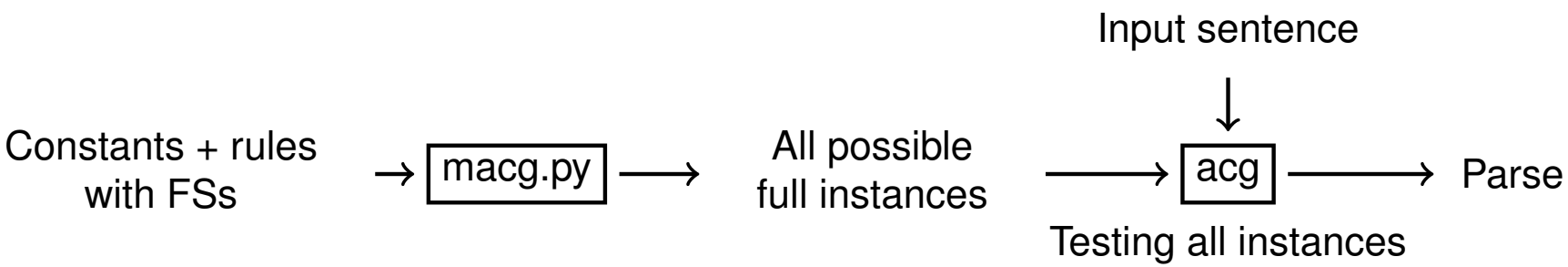
Formal proof of work

Given (underspecified) FSs f, f' :

$$\mathrm{FI}(f) = \{g \mid f \sqsubseteq g \text{ and } g \text{ is fully specified}\} \tag{4}$$

Property

f and f' unify iff $\mathrm{FI}(f) \cap \mathrm{FI}(f') \neq \emptyset$



Conclusion

MACG preprocessor :

- Reduces redondancy
- Simulates feature structures
- Easy way to create a CFG in ACGtk

Conclusion

MACG preprocessor :

- Reduces redondancy
- Simulates feature structures
- Easy way to create a CFG in ACGtk

Limitations :

- Combinatorial explosion : reduces performances (e.g. coordination)
- Limited use of variables
- Limited “customization”

Conclusion

MACG preprocessor :

- Reduces redondancy
- Simulates feature structures
- Easy way to create a CFG in ACGtk

Limitations :

- Combinatorial explosion : reduces performances (e.g. coordination)
- Limited use of variables
- Limited “customization”

Future prospects :

- □ Higher-order types
- □ (Concatenative) morphological rules
- ? Type hierarchy and inheritance
- ? Reentrancy