FO-dot

Pierre Carbonnelle, Marc Denecker

Apr 05, 2024

CONTENTS

1 Introduction 3

2 Rationale 5

3 Notation 7

4 FO[Core] 9

5 FO[Sugar] 43

6 FO[PF] 49

7 FO[ID] 53

8 FO[Int] 59

9 FO[Real] 67

10 FO[Agg] 71

11 FO[Infinite] 75

12 FO[Concept] 77

13 FO[Unit] ? 79

14 FO[·] 81

15 Bibliography 85

16 Index 87

Bibliography 89

Index 91

This document describes the FO[·] (aka FO-dot) standard.

FO[·] is First Order logic with various extensions to allow complex knowledge to be expressed in a rigorous and elaboration-tolerant way. FO[·] facilitates the development of knowledge-intensive applications capable of intelligent behavior [CVVD22].

The document is meant to be a reference for Knowledge engineers and developers of reasoning engines.

Suggestions for improvements to the document can be made via issues on GitLab.

CHAPTER

ONE

INTRODUCTION

FO[·] (aka FO-dot) is First Order logic with various extensions to allow complex knowledge to be expressed in a rigorous and elaboration-tolerant way. FO[·] facilitates the development of knowledge-intensive applications capable of intelligent behavior [CVVD22]. It uses conventional logic notations as much as possible.

An FO[·] knowledge base (KB) is a set of statements in an application domain. These statements allow distinguishing the *satisfying* states of affairs (in which the statements are true), from those in which at least one of them is false.

If the statements in the KB are laws of the physical world, then they are true in any physical situations we could possibly be in. If they are a transcription of a statutory law, they are true in any legally-acceptable state of affairs. If they are rules of thumbs about what a good product configuration should be, they are true in states of affairs where the product has all the desired properties. Hence, the statements in a KB describes what is possible, acceptable or desirable, depending on the purpose of the KB.

An FO-dot knowledge base is not a program. It cannot be run or executed. It is not even the description of a computational problem. It is just a “bag of information”, describing certain properties of the application domain.

The information provided in the knowledge base can be used as input to various generic reasoning algorithms: e.g., algorithms to determine relevant questions, to compute consequences of new facts, or to explain them. These algorithms are implemented in reasoning engines, such as IDP-Z3.

These are the current extensions of FO[·]:

Core Multi-sorted logic with finite data structures and equality

Sugar Binary quantification, type inference and other convenient notations

PF Partial functions

ID (Inductive) definitions

Int Integer arithmetic, ranges and dates

Real Real arithmetic

Agg Aggregates: cardinality, sum, min, max

Infinite Quantification over infinite domains (including recursive types)

Concept Quantification over intensional objects

A reasoning engine may support all the extensions, or only a subset. The extensions are loosely coupled, so that each possible combination defines a different language. The dependencies are shown in the following diagram:

The grammar of all the extensions combined is described in the *FO[.] page*.

Other extensions are under consideration for inclusion in FO[·]:

Open Quantification over open (aka uninterpreted) types

Causal Causal processes

Cos Transcendental functions

Unit Physical quantities with Unit of Measurements

Note: We use the following nomenclature:

- FO is First Order logic;
- FO[x,y] (with square brackets) is the *concrete* language with extensions x,y; it is used to write knowledge bases;
- FO(x,y) (with parenthesis) is the *theoretical* language with extensions x,y; it is used to rigorously specify the essential semantics of the extension;
- FO[·] (resp. FO(·)) is the concrete (resp. theoretical) language with all the extensions described in this document.

CHAPTER

TWO

RATIONALE

What is the knowledge base paradigm and why do we need it ?

Declarative knowledge and its use for solving problems is studied in various fields such as knowledge representation and reasoning (KRR), computational logic and declarative problem solving paradigms. In the past half a century, an immense body of scientific knowledge about knowledge and its use for solving problems has been collected in these various domains. Yet, at this moment no coherent scientific approach to the study of knowledge and its use for problem solving has emerged. Scientific understanding is scattered over the many fields that study it.

One issue that fragments the study of knowledge and reasoning more than anything else is the type of inference (the type of reasoning with this knowledge). Historically, the study of logic was seen as the study of reasoning. This inference-centric view naturally leads to different logics specialized on specific forms of inference. To start with, classical first order logic (FO) is sometimes defined as the logic of deductive reasoning. But database languages (SQL, Datalog) are logics for query answering; constraint programming (CP) for constraint solving; answer set programming (ASP) for solving problems by computing answer sets, temporal logics for model checking, etc. All these logics differ strongly at the level of the underlying inference mechanism as well as on their syntax and on their scientific terminologies.

But knowledge is independent of the sort of reasoning task. Take the proposition:

During opening hours, at least one secretary is present at the entrance desk.

What task is this proposition to be used for? It could be used as a query to a scheduling database, or as a constraint in a scheduling problem, or as a property to be verified from a formal specification of a scheduling problem, or as a correctness property to be proven of a program that computes schedules. The proposition in itself is not bound to any specific problem or form of inference; yet, in the current state of the art, depending on the problem, the proposition needs to be rephrased in a logic that supports the required type of inference.

In fact, the proposition has a natural representation as the sentence in first order logic:

```
        ∀t : Open(t) ⇒ ∃s : Secretary(s) ∧ at(s, EntranceDesk, t).
```

Assume we want to solve a scheduling problem satisfying this formula. For a long time a "deductive" logic like FO was deemed unsuitable for such problem. Often this was blamed on the fact that deduction in FO is undecidable. However, the more essential reason is that the scheduling problem is simply not a deductive problem and FO theorem provers are useless. Instead, other declarative paradigms were developed for this such as CP.

The mix up of reasoning with the knowledge representation language is not a desirable situation; neither from a scientific point of view as it obscures the nature of knowledge and obfuscates it with the form of reasoning, nor from a pragmatical point of view as it forces the knowledge engineer to rephrase the same proposition in different logics to get different problems solved. What is needed is a knowledge-oriented scientific approach to study knowledge separated from inference and problem solving, that allows to study knowledge with scientific methods, that studies how to express knowledge as naturally and compactly as possible, and that studies the various sorts of problems that can be solved using a knowledge base. It leads to the idea to build software solutions by expressing domain knowledge in a symbolic knowledge base, and use it to solve a range of problems and tasks by applying various inference methods. Such a system is what we understand by the term ``a knowledge base system".

Why start from classical first order logic (FO)?

Classical first order logic dates from the late nineteenth century. Why use that logic? Because it's language constructs $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow, \forall, \exists$ are basic indispensable construct for expressing human knowledge. Moreover, FO's formal semantics correctly captures their meaning.

Why extend FO?

Classical language is really a very small language, with only 7 language constructs. It can be improved for knowledge representation in many different ways.

CHAPTER

THREE

NOTATION

The syntax of FO and its extensions is specified using the EBNF notation. In this notation, `[ a ]` zero or one occurrence of `a`, and `{ a }` represents zero, one, or more occurrences of `a`.

The lexicon lists the tokens of the language. They are specified using Python Regular Expressions. For example, an identifier satisfies `[^\d\W]\w*\b`. The regular expressions can be tested online, e.g., using pythex.org.

The tokens satisfying a regular expression are denoted by names in upper case, e.g., `ID`. Non-terminal symbols are specified by production rules and are denoted in CamelCase, e.g., `TypeInterpretation`. In the body of production rules, terminal symbols are denoted:

- by referring to tokens in the lexicon, e.g., `ID`;
- using string literals, e.g., `'vocabulary'`;
- using string interpolation, e.g., `'is_{CONSTRUCTOR}'`: the terminal symbol is the concatenation of `is_` and a `CONSTRUCTOR`

Some production rules in FO[Core] are modified or extended in other extensions. To allow more modularity, non-terminal strings can be defined by several production rules:

```
s ← 'A';
s ← 'B' | 'C';
```

is equivalent to:

```
s ← 'A' | 'B' | 'C';
```

The EBNF syntax is not sufficiently precise to define what is a well-formed FO[·] program. So, we complement it with an informal description of “well-formedness” conditions.

CHAPTER

FOUR

FO[CORE]

This chapter describes the syntax and semantics of the minimal language constructs supported by every member of the FO[·] language family. Some of the constraints introduced by this minimal language are removed by extensions such as FO[Infinite].

4.1 Goal

FO[Core] is a typed (aka multi-sorted) first order logic with equality, if-then-else, and special constructs to facilitate the creation of identifiers and to specify the interpretation of predicates and functions. All types have a finite interpretation, allowing grounding [WMarienD14]. (This constraint is removed in FO[Int] and FO[Real])

4.2 Knowledge Base

This section describes the high-level structure of a knowledge base.

Lexicon

A Knowledge Base is a text file encoded in UTF-8.

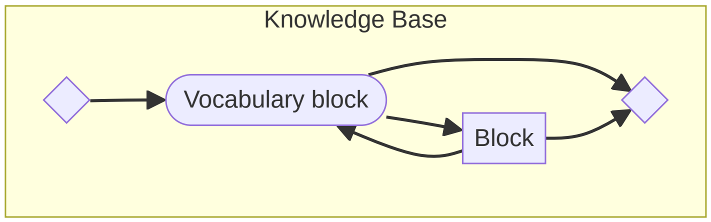
The following character(s) are "white spaces": they separate tokens, but have no meaning by themselves.

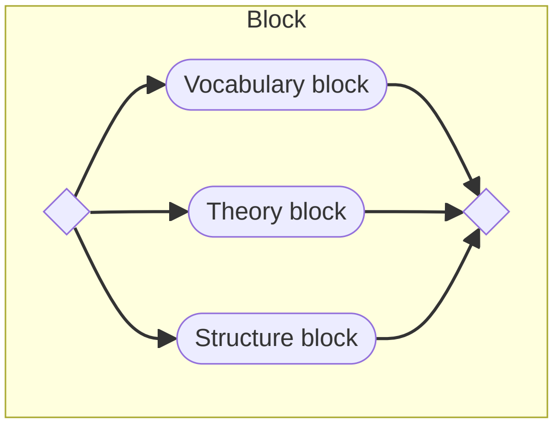
- `0x09` (tab)
- `0x10` (new line)
- `0x13` (line feed)
- `0x32` (space)
- comments: string starting from `//` till the end of the line (`0x10` or `0x13`).

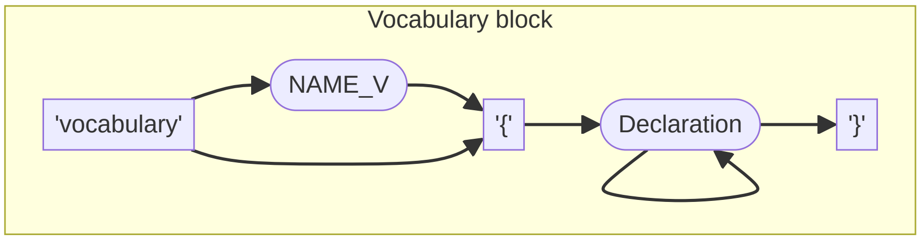
Token	Pattern	Example
ID	`[^\d\W]\w*\b`	`Color`
NAME_V	ID	`V`, the name of a vocabulary
NAME_TH	ID	`T`, the name of a theory

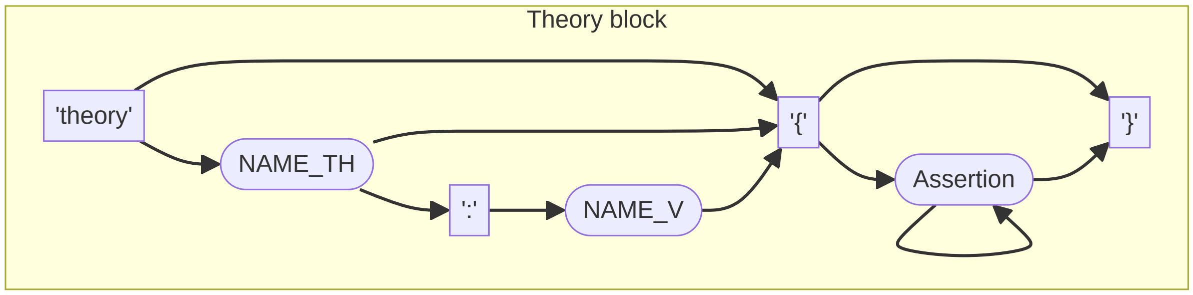
Syntax

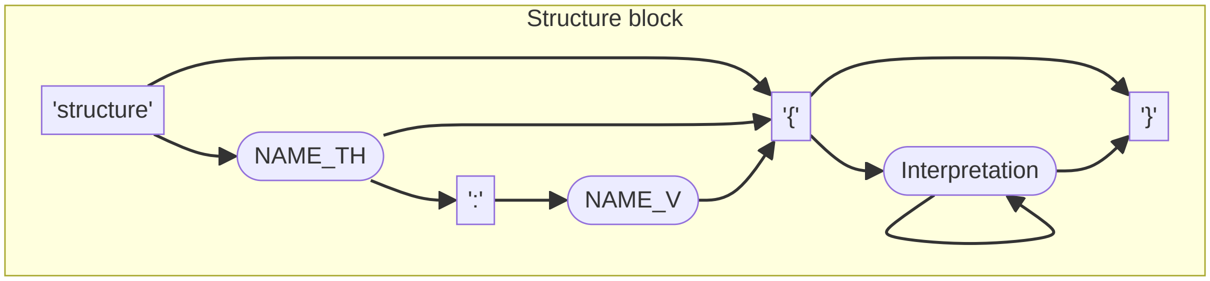
```
KnowledgeBase ← VocabularyBlock { (VocabularyBlock | TheoryBlock | StructureBlock) };
  VocabularyBlock ← 'vocabulary' [NAME_V] '{' { Declaration } '}';
  TheoryBlock ← 'theory' [ NAME_TH [: NAME_V] ] '{' { Assertion } '}';
  StructureBlock ← 'structure' [ NAME_TH [: NAME_V] ] '{' { Interpretation } '}';
```


Knowledge Base
Vocabulary block
Block


Block
Vocabulary block
Theory block
Structure block


Vocabulary block
'vocabulary'
NAME_V
'{'
Declaration
'}'


Theory block
'theory'
NAME_TH
':'
NAME_V
'{'
Assertion
'}'



Well-formedness

1. The Knowledge Base must have at least one vocabulary and one theory block.

2. The Knowledge Base may contain other types of block, e.g., procedural blocks to perform reasoning tasks. These blocks are not part of this standard.
3. If omitted, `NAME_V` (resp. `NAME_TH`) is assumed to be the string `V` (resp. `T`).
4. The `NAME_V` in a theory (or structure) block must be the same as the `NAME_V` of a previous vocabulary block.
5. When the Knowledge Base has several vocabulary (resp. theory or structure) blocks, two vocabulary (resp. theory or structure) blocks cannot have the same `NAME_V` (resp. `NAME_TH`).

Semantics

1. A state of affairs is a static "mental world" that can be communicated by a set of descriptive sentences in natural language (e.g., in English). (A state of affairs is similar to a mental model in cognitive science [JL08])
2. A *knowledge base* is a formal description of all the possible (or desirable, or acceptable) states of affairs in a problem domain.
3. A *vocabulary* declares the *symbols* denoting important *concepts* in a problem domain. Each symbol has an *informal meaning* in natural language in the problem domain. There are 3 types of symbols: type, function and predicate symbols.
4. A particular state of affairs is described by giving a total *interpretation* to every symbol. The set of these interpretations is a *total structure*. A structure is an *abstraction* of a state of affairs.
5. A *theory* (over a vocabulary) is a set of assertions that are true in every *model*, i.e., in every structure that are possible (or desirable, or acceptable). A structure in which all assertions are true is a model of the theory.
6. Many *reasoning tasks* involve only one vocabulary and theory, but some reasoning tasks, such as "determine if 2 theories are equivalent" require more.
7. A structure block is a special kind of theory blocks that only contains interpretations of types and symbols. They typically contain observations about a state of affairs.

Example

```
vocabulary {
    // here comes the declaration of types and symbols
}

theory {
    // here comes the definitions and assertions
}

structure {
    // here comes the interpretation of some symbols
}
```

4.3 Type declaration

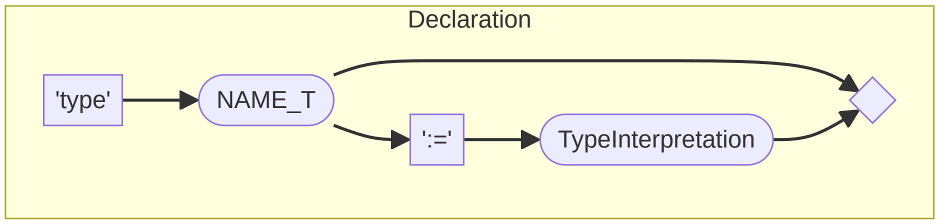
This section describes how a type is declared in the vocabulary.

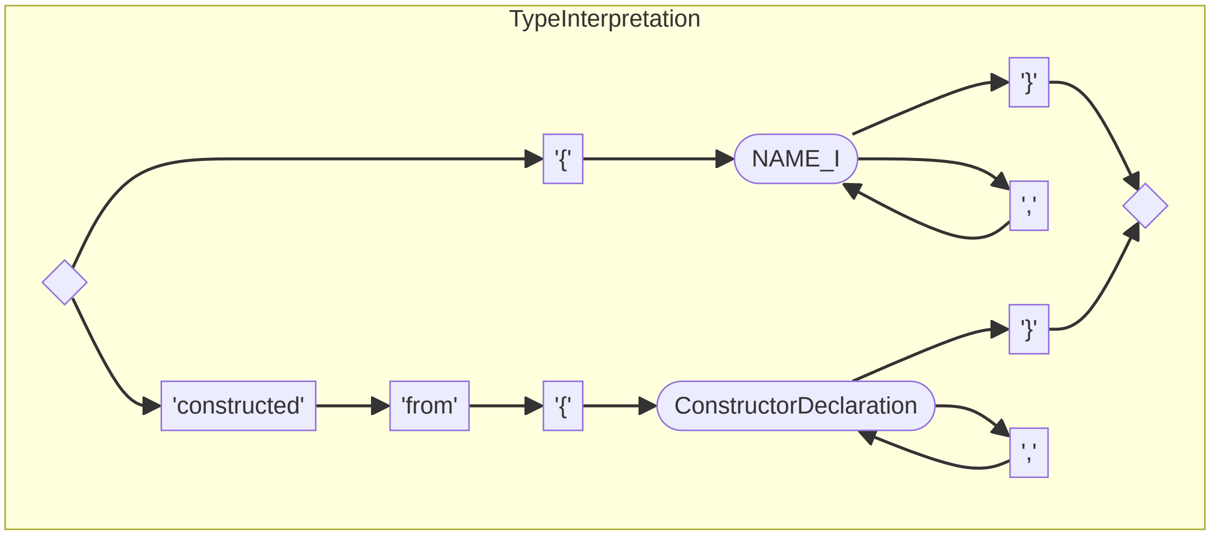
Lexicon

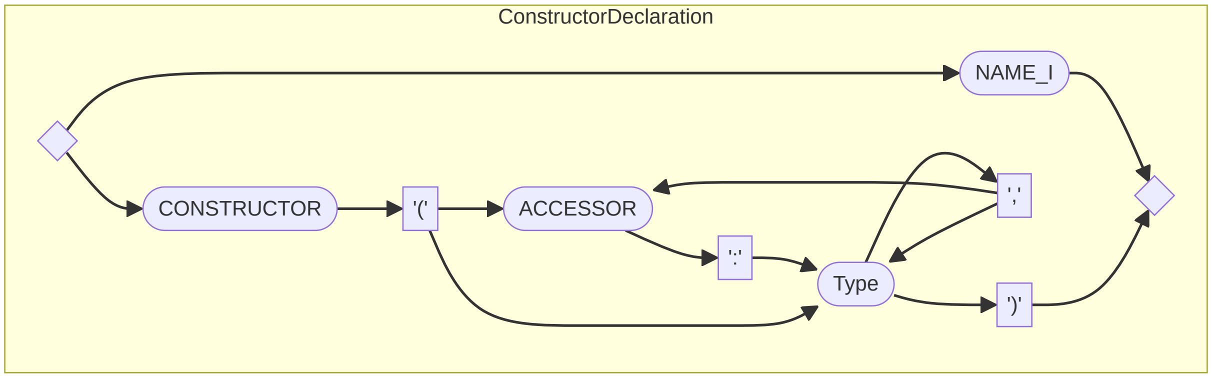
Token	Pattern	Example
ACCESSOR	ID	
BOOL	either ⍰ or `Bool`	
CONSTRUCTOR	ID	
NAME_T	ID	
NAME_I	ID or `'[^']*'`	‘John Doe’

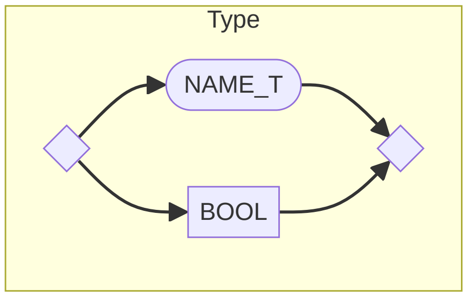
Syntax

```
Declaration ← 'type' NAME_T [':=' TypeInterpretation] ;
  TypeInterpretation ← '{' NAME_I { ',' NAME_I } '}';
  TypeInterpretation ← 'constructed' 'from'
        '{' ConstructorDeclaration { ',' ConstructorDeclaration } '}';
    ConstructorDeclaration ← NAME_I;
    ConstructorDeclaration ← CONSTRUCTOR '(' [ACCESSOR ':' ] Type
            {',' [ACCESSOR ':' ] Type} ')';
      Type ← NAME_T | BOOL;
```


Declaration
'type'
NAME_T
':='
TypeInterpretation


TypeInterpretation
'{'
NAME_I
'}'
','
'constructed'
'from'
'{'
ConstructorDeclaration
'}'
','


ConstructorDeclaration
NAME_I
CONSTRUCTOR
'('
ACCESSOR
','
':'
Type
')'



Well-formedness

1. A `NAME_T` (resp. `NAME_I`, `CONSTRUCTOR`, `ACCESSOR`) can be declared only once. A `NAME_T` cannot conflict

with another Type (e.g., `Bool`). A `NAME_I` cannot conflict with another Identifier.

2. The 4 sets of `Type`, `Identifier`, `CONSTRUCTOR` and `ACCESSOR` strings must be disjoint.
3. Constructors are not recursive: `NAME_T` occurring as argument of a `CONSTRUCTOR` must have been previously declared (relaxed in FO[Infinite]).

Semantics

1. The *domain of discourse* is a set of objects (aka *individuals*) in the state of affairs.
2. *Types* are disjoint subsets of the *domain of discourse*. The domain of discourse is the union of the types. All types are finite sets in FO[Core].
3. A type interpretation gives a unique *name* to each element of the subset (UNA: Unique Name Axiom). Every element in the subset has a *name* (DCA: Domain Closure Axiom).
4. A name can be an *identifier* or a *compound identifier*.
5. A "direct" type declaration consists of a set of identifiers.
6. A "constructed type" declaration consists of a set of *constructor declarations*. All the compound identifiers of the constructed type are obtained by applying each *constructor* in the declaration to each possible tuple of elements of their `Type` arguments (if any).

Examples

```
type Color := {red, blue, green}
type RGB := constructed from {RGB(Bool, Bool, Bool)}
```

`red`, `blue` and `green` are the 3 different `Colors`. There are only 3 `Colors`. `RGB(true, false, false)` is an element of type `RGB`. This type has 2⍰2⍰2 = 8 elements.

4.4 Function and Predicate declaration

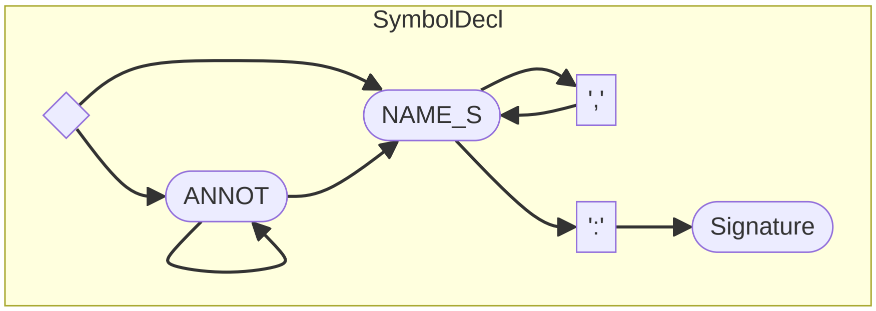
This section describes how functions and predicates are declared in the vocabulary.

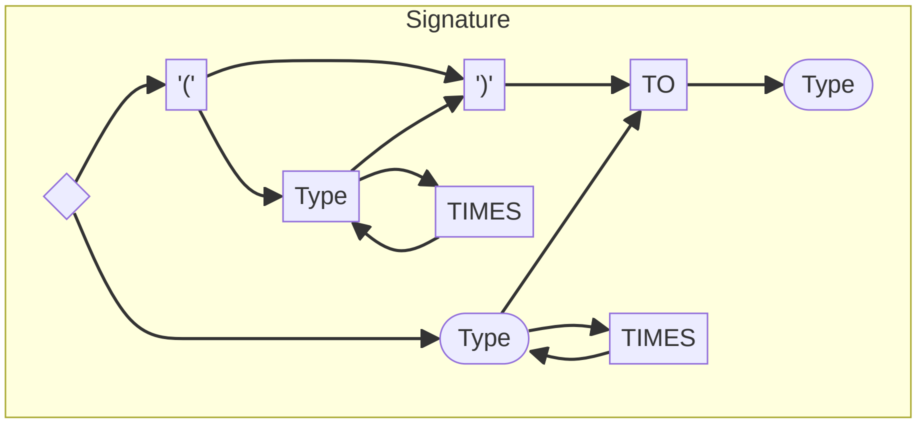
Lexicon

Token	Pattern	Example
ANNOT	`\[[^\]]*\]`	`[This is an annotation]`
NAME_S	ID	
TIMES	`*`|⍰	
TO	`->`|→	

Syntax

```
  Declaration ← SymbolDecl;
    SymbolDecl ← { ANNOT } NAME_S { ',' NAME_S } ':' Signature;
      Signature ← '(' [ Type { TIMES Type } ] ')' TO Type;
      Signature ←  Type { TIMES Type } TO Type;
```


SymbolDecl
NAME_S
','
ANNOT
':'
Signature



Well-formedness

1. A `NAME_S` can be declared only once. It cannot conflict with another (built-in) symbol (such as `abs` in FO[Int]).

2. A NAME_T cannot occur in a Signature before it has been declared.
3. The 5 sets of Type, Symbol, Identifier, CONSTRUCTOR and ACCESSOR strings must be disjoint.

Semantics

1. This Declaration declares one (or more) function symbol and its type signature. The Types before TO specify the *domain* of the function; the Type after TO specify its *range*.
2. A function whose range is BOOL is also called a *predicate*.
3. Annotations do not have logic meaning, but can be parsed by a reasoning engine. They may be used to, e.g., specify the informal semantics of a symbol, for display to a user. Their use is not specified in this standard.

Examples

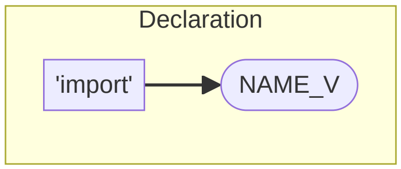
```
convex, equilateral : () → ⍰
colorOf: Country → Color

[edge of a graph]
edge : Node ⍰ Node → Bool
```

4.5 Import

This section declares how declarations can be imported from a vocabulary in another.

Syntax

```
Declaration ← 'import' NAME_V;
```



Well-formedness

1. vocabulary `NAME_V` must have been declared previously in the knowledge base.

Semantics

1. the declarations in the vocabulary named `NAME_V` are added to the vocabulary that contains the `import` declaration.

4.6 Assertions

This section describes how logic statements are added to a theory.

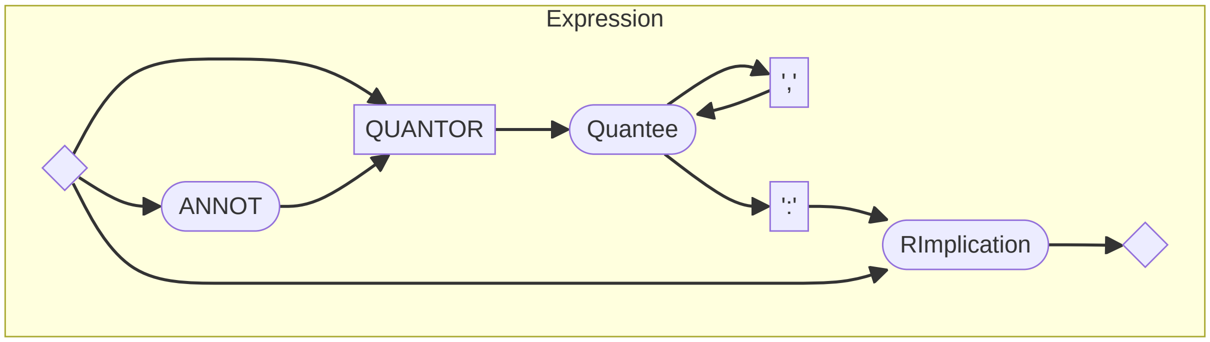
Lexicon

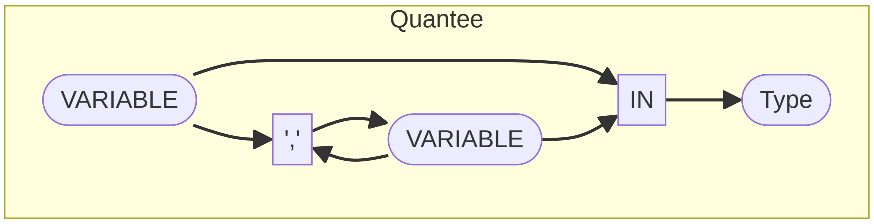
Token	Pattern	Example
AND	`&|∧`	
EQUIVALENCE	`<=>|⇔`	
IMPLICATION	`=>|⇒`	
IN	`in|∈`	
OR	`\\||∨`	
QUANTOR	`[∀!∃?]`	
R_IMPLICATION	`<=|⇐`	
UNARY	`~|¬`	
VARIABLE	ID	

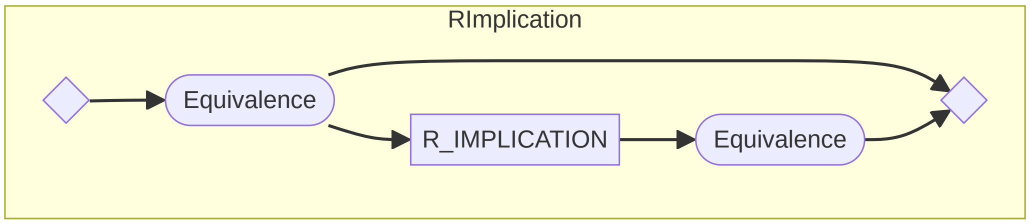
Syntax

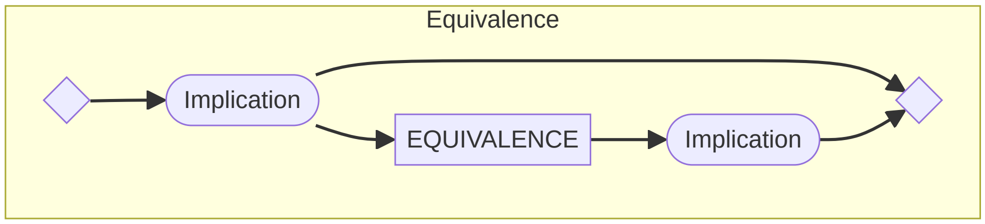
```
Assertion ← Expression '.';
Expression ← [ { ANNOT } QUANTOR Quantee {',' Quantee} ':' ] RImplication;
             Quantee ← VARIABLE {',' VARIABLE} IN Type;
    RImplication ← Equivalence [R_IMPLICATION Equivalence];
    Equivalence  ← Implication [EQUIVALENCE   Implication];
    Implication  ← Disjunction [IMPLICATION   Disjunction];
    Disjunction  ← Conjunction {OR            Conjunction};
    Conjunction  ← Unary       {AND           Unary};
    Unary        ← { UNARY }  Base;

    Base ← 'if' Expression 'then' Expression 'else' Expression;
    Base ← { ANNOT } QUANTOR Quantee {',' Quantee} ':' RImplication;
    Base ← Symbol '(' [Expression {',' Expression}] ')';
    Base ← Identifier | VARIABLE;
    Base ← { ANNOT } '(' Expression ')';

        Identifier ← 'true' | 'false' | NAME_I;
        Symbol ← NAME_S;
        Symbol ← CONSTRUCTOR | ACCESSOR | 'is_{CONSTRUCTOR}';
```

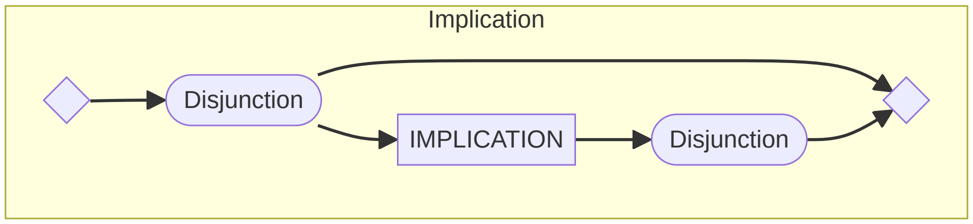
Here, `'is_{CONSTRUCTOR}'` represents any string constructed from `'is_'` and a `CONSTRUCTOR`, e.g., `is_RGB`.

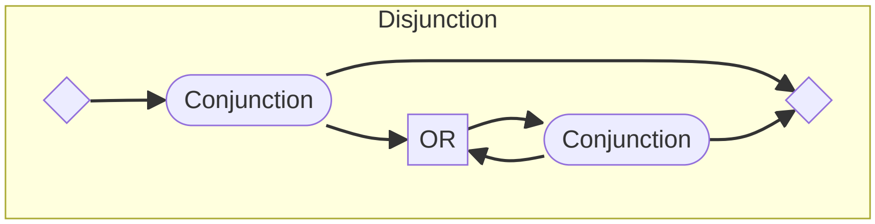

Expression
QUANTOR
Quantee
','
ANNOT
':'
RImplication

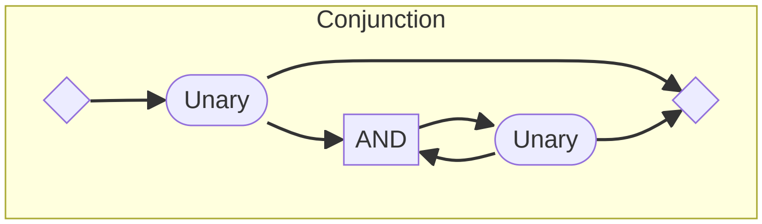

Quantee
VARIABLE
','
VARIABLE
IN
Type

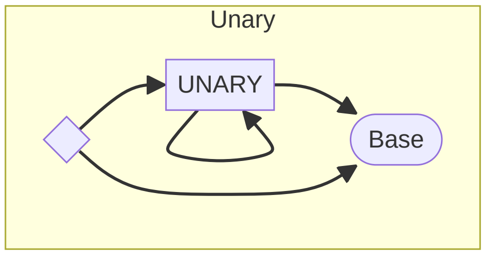

RImplication
Equivalence
R_IMPLICATION
Equivalence

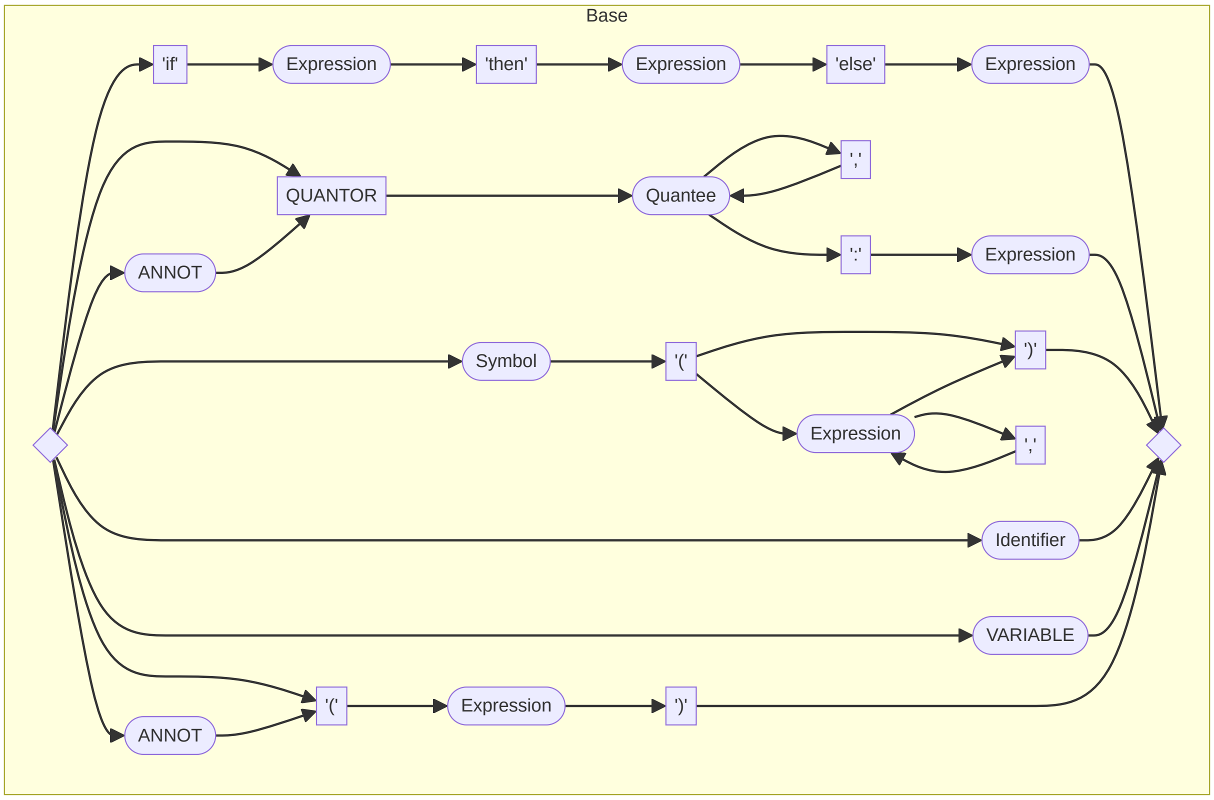

Equivalence
Implication
EQUIVALENCE
Implication


Implication
Disjunction
IMPLICATION
Disjunction


Disjunction
Conjunction
OR
Conjunction


Conjunction
Unary
AND
Unary


Unary
UNARY
Base



Well-formedness

1. Each `NAME_T`, `NAME_S` strings must have been declared in the vocabulary. Each `NAME_I`, `CONSTRUCTOR` and

ACCESSOR must have been declared before occurring in a Base.

2. Each expression has a type based on the declarations in the vocabulary. An assertion must be a boolean expression. The elements of an expression must have the appropriate types, as specified in the symbol declarations or as required by the built-in connectives.
3. Each variable must occur in the scope of a quantifier that declares it.

Semantics

1. The expressions above have their usual mathematical / logic meaning. (See also the semantics of FO(Core) below)
2. The production rules above define the precedence of the operators.
3. An is_{CONSTRUCTOR} string denotes a unary predicate that says whether the argument was constructed with CONSTRUCTOR.
4. When ACCESSOR is the name of the n-th argument of CONSTRUCTOR according to its declaration, it denotes a function that takes an object O created by CONSTRUCTOR, and returns the n-th argument that has been applied to the constructor to construct O. The occurrences of ACCESSOR in an expression must be properly guarded to ensure it is applied to appropriate arguments.

Examples

In `∀x,y ∈ T: p(x,y) ∨ f(f(x))=y.`, `x` and `y` are variables of type `T`.

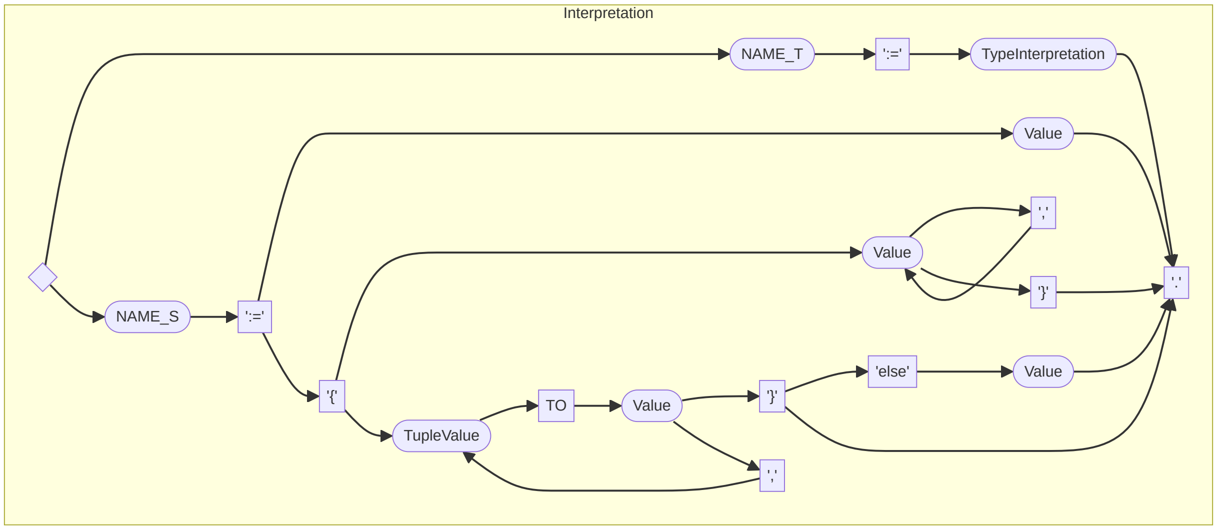
Assuming `type Color := {RGB{red: Bool, green: Bool, blue: Bool}}`, `is_RGB(RGB(true, false, false))` is `true` and `red(RGB(true, false, false))` is `true`.

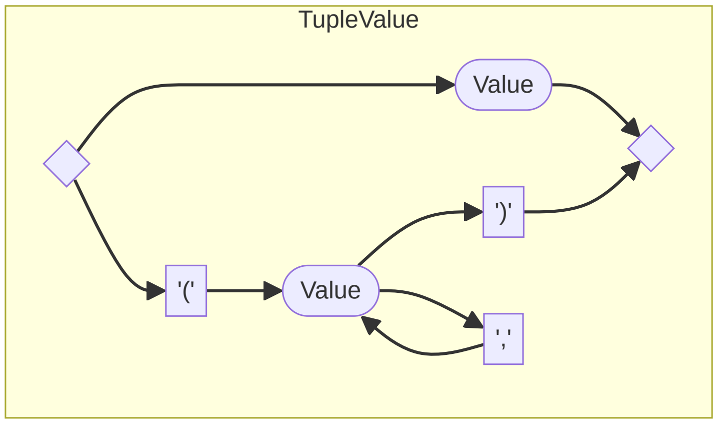
4.7 Symbol Interpretation

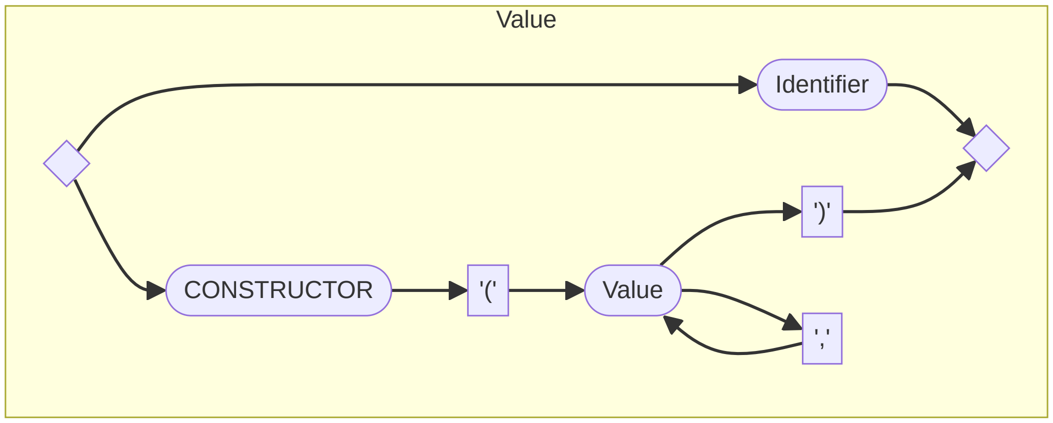
This section describes how the interpretation of a function or predicate is added to a theory or structure.

Syntax

```
Assertion ← Interpretation;
Interpretation ← NAME_T ':=' TypeInterpretation '.';
Interpretation ← NAME_S ':=' Value '.';
Interpretation ← NAME_S ':=' SymbolInterpretation '.';
  Value ← Identifier;
  Value ← CONSTRUCTOR [ '(' Value {',' Value } ')';
  SymbolInterpretation ← '{' TupleValue { ',' TupleValue } '}';
  SymbolInterpretation ← '{' TupleValue TO Value { ',' TupleValue TO Value} '}'␣
↪[else];
    TupleValue ← Value;
    TupleValue ← '(' Value { ',' Value } ')';
  else ← 'else' Value;
```


Interpretation
NAME_T
':='
TypeInterpretation
Value
','
Value
'}'
'.'
NAME_S
':='
'{'
'else'
Value
TO
Value
'}'
TupleValue
','


TupleValue
Value
')'
'('
Value
','



Well-formedness

1. A `NAME_T` or `NAME_S` can be interpreted only once.

2. All types must be interpreted, in the vocabulary or in a theory/structure.
3. Each `NAME_I`, `CONSTRUCTOR` and `ACCESSOR` must have been declared before occurring in a `Value`.
4. The interpretation can be a single value only for nullary predicates or functions (i.e., with an arity of 0).
5. The number and type of values in an interpretation must match the type signature of the `NAME_S` being interpreted.
6. The number and types of values in a compound identifier must match the type signature of the `CONSTRUCTOR`.
7. The `TO` element is mandatory in function interpretations, and prohibited in predicate interpretations (relaxed in FO[Sugar]).

Semantics

1. These assertions define the interpretation of types, predicates and functions.
2. For a type, the assertion has the same semantics as in a type declaration
3. For a predicate, the set of `TupleValue` is the set of values for which the predicate is `true`; the predicate is false for any other `TupleValue`; for nullary predicate, the unique Value can be `true` or `false`.
4. For a function, the assertion maps tuple of arguments to their value. The value after `else` is the value given to the function for tuples of arguments that are not in the enumeration. For nullary symbols, the interpretation can be a unique value.

Examples

```
Color := {red, blue, green}.
node := {A, B, C}.
colorOf := {A -> red, B -> blue} else green.
edge := {(A,B), (B,B)}.
convex := true.
```

4.8 FO(Core) abstract language

This section contains the formal theory of the FO(Core) abstract language.

The translation between the abstract language and the concrete language is not provided. However, corresponding concepts in the abstract and the concrete language are denoted by the same words: this should allow the reader to establish the correspondence by himself.

T is a set of *type-symbols*. It contains type-symbol .

A *n-signature* is a (n+1)-tuple of type-symbols, noted $T_1 ⍰ \ldots ⍰ T_n \to T$, where $0 \leq n$. (A 0-signature is noted $() \to T$)

A *vocabulary* is a set of (symbol , n-signature) pairs.

A *typing function* is a function that maps variables to type-symbols. For a given , we define $[x : T]$ to be the function $'$ identical to except that $'(x) = T$.

The *type of string* (over) given a typing function , noted $(,)$, is a type-symbol partially defined by induction as follows (terminal symbols are underlined for clarity):

	$(,)$	if
$\underline{true}$		
$\underline{false}$		
x	(x)	x is a variable
$\underline{(}t_1\underline{,}..\underline{,}t_n\underline{)}$	T	$(, T_1$⍰…⍰$T_n{\rightarrow}T) \in$ and $\forall i \in [1, n] : (t_i,) = T_i$
$\underline{\vee}$		$(,) =$ and $(,) =$
$\underline{\neg}$		$(,) =$
$\underline{\exists} x \underline{\in} T \underline{:}$		x is a variable and $(, [x : T]) =$
$t_1 \underline{=} t_2$		$(t_1,) = (t_2,)$
<u>if</u> **<u>then</u>** t_1 **<u>else</u>** t_2	T	$(,) =$ and $(t_1,) = T = (t_2,)$
otherwise	undefined	

Let $\varnothing$ be the typing function with empty domain. We say that is a *well-formed formula of type* T over if $(, \varnothing) = T$. A *sentence* is a well-formed formula of type .

This table shows how to extend the syntax with convenient notations:

Convenient notation	Equivalent formula
$\underline{\wedge}$	$\neg(\neg\vee\neg)$
$\underline{\Rightarrow}$	$\neg\vee$
$\underline{\Leftarrow}$	$\vee\neg$
$\underline{\Leftrightarrow}$	$(\wedge)\vee(\neg\wedge\neg)$
$\underline{\forall} x \underline{\in} T \underline{:}$	$\neg\exists x \in T : \neg$

A *(total) structure* over consists of :

- an object domain D containing , the set of booleans;
- a mapping of type-symbols T to disjoint subsets T of D; D is the union of all T;
- a (total) mapping from symbols with n-signature T_1⍰…⍰$T_n{\rightarrow}T$ to (total) functions from T_1⍰…⍰T_n to T.

Note: `Identifier` and `CONSTRUCTOR` strings are function symbols whose interpretation have the following properties:

1. All well-formed ground terms of type T built exclusively with them uniquely identify an element of T.
2. Every element of T is uniquely identified by a well-formed ground term of type T built exclusively with them.

We define a *variable assignment* as a mapping of variables to elements in D. A variable assignment extended so that the mapping of x is d is denoted $[x : d]$.

The *value* of formula in $(,)$, denoted $[\![t]\!]^I = v$, is an element of D partially defined by induction as follows:

	$\llbracket\rrbracket^I$	if
$\underline{true}$	$true$	
$\underline{false}$	$false$	
x	(x)	x is a variable in the domain of
$\underline{(}t_1\underline{,}..\underline{,}t_n\underline{)}$	$(\llbracket t_1\rrbracket^I, .., \llbracket t_n\rrbracket^I)$	$(, T_1⍰\ldots⍰T_n{\to}T) \in$ and $\forall i \in [1, n] : \llbracket t_i\rrbracket^I$ is defined
$\underline{\vee}$	$false$	$\llbracket\rrbracket^I$ is defined and false
$\underline{\vee}$	$\llbracket\rrbracket^I \vee \llbracket\rrbracket^I$	$\llbracket\rrbracket^I$ and $\llbracket\rrbracket^I$ are defined
$\underline{\neg}$	$\neg\llbracket\rrbracket^I$	$\llbracket\rrbracket^I$ is defined
$\underline{\exists}x\underline{\in}T\underline{:}$	$\exists d{\in}T^I : \llbracket\rrbracket^I[x : d]$	$\llbracket\rrbracket^I[x : d]$ is defined for every d in T^I
$t_1\underline{=}t_2$	$\llbracket t_1\rrbracket^I = \llbracket t_2\rrbracket^I$	$\llbracket t_1\rrbracket^I$ and $\llbracket t_2\rrbracket^I$ are defined
if **then** t_1 **else** t_2	$\llbracket t_1\rrbracket^I$	$\llbracket\rrbracket^I = true$ and $\llbracket t_1\rrbracket^I$ is defined
if **then** t_1 **else** t_2	$\llbracket t_2\rrbracket^I$	$\llbracket\rrbracket^I = false$ and $\llbracket t_2\rrbracket^I$ is defined
otherwise	undefined	

Notice that $\underline{\vee}$ and **if** **then** t_1 **else** t_2 have non-strict semantics.

We say a total structure *satisfies* sentence iff $\llbracket\rrbracket^I = \mathit{true}$ for any . This is denoted $\vDash$. Structures that satisfy are called *models* of .

CHAPTER

FIVE

FO[SUGAR]

5.1 Goal

The goal of FO[Sugar] is to add convenient notations: binary and direct quantifications, variable declaration, and the "in" operator.

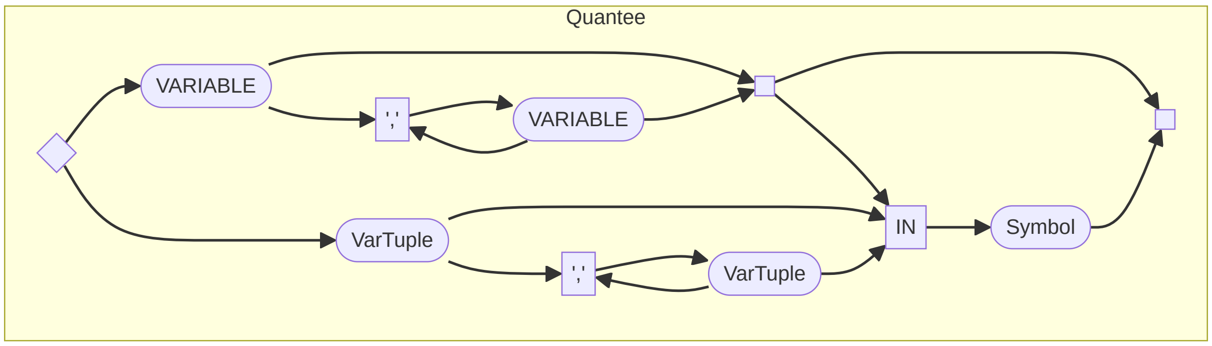
It also supports reasoning with partial interpretation of symbols.

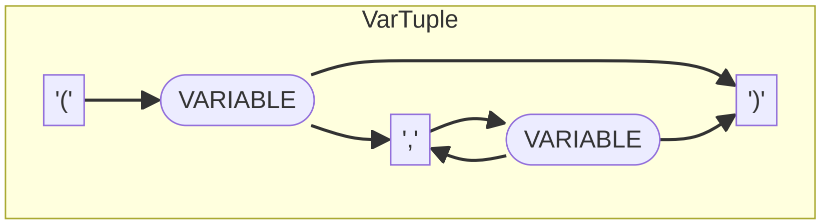
5.2 Binary Quantification and Variable Declaration

Syntax

FO[Sugar] adds these production rules:

```
Declaration ← VARIABLE IN ( Type | Symbol );
Quantee ← VARIABLE {',' VARIABLE} [ IN Symbol ];
Quantee ← VarTuple {',' VarTuple} IN Symbol;
    VarTuple ← '(' VARIABLE { ',' VARIABLE } ')';
```


Quantee
VARIABLE
','
VARIABLE
VarTuple
','
VarTuple
IN
Symbol



Well-formedness

1. `Type` must be a declared type.

2. `Symbol` must be a declared predicate.
3. `Symbol` must be a unary predicate when quantifying a `VARIABLE`, and an n-ary predicate when quantifying a `VarTuple`.
4. When `IN Symbol` is omitted, the type of the variable(s) is inferred from the variable's declaration (if given), or from the quantified formula (to make it well-formed; an error is raised otherwise).
5. When quantifying a `VarTuple`, the number of `VARIABLE` in `VarTuple` must match the arity of `Symbol`.

Semantics

Let's assume that the type signature of `P` is `T1 ⍰ ... ⍰ Tn → ⍰`.

`∃ x1 ∈ P: ψ` reduces to `∃ x1 ∈ T1: P(x1) ∧ ψ` (where `P` is unary). It can also be written `∃ x1: ψ`. `∃ (x1, ..., xn) ∈ P: ψ` reduces to `∃ x1 ∈ T1, .., xn ∈ Tn: P(x1, .., xn) ∧ ψ`.

Similarly,`∀ x1 ∈ P: ψ` reduces to `∀ x1 ∈ T1: P(x1) ⇒ ψ` (where `P` is unary). It can also be written `∀ x1: ψ`. `∀ (x1, ..., xn) ∈ P: ψ` reduces to `∀ x1 ∈ T1, .., xn ∈ Tn: P(x1, .., xn) ⇒ ψ`.

Note: When inferring `Symbol` is difficult for a human because of the complexity of the quantified formula, it is recommended to declare the variable, or explicitly state the `Symbol` in `Quantee`.

Example

Assuming that the type signature of `man` is `Person → ⍰`, `∀ x in man: mortal(x).` is equivalent to `∀ x in Person: man(x) ⇒ mortal(x).`

5.3 Direct quantification

Syntax

FO[Sugar] adds these production rules:

```
Quantee ← VARIABLE {',' VARIABLE} IN '{' Value { ',' Value } '}';
Quantee ← VarTuple {',' VarTuple} IN '{' TupleValue { ',' TupleValue } '}';
```

Well-formedness

1. The `Values` must have the same type; this type is the type of the `VARIABLEs`.
2. The `TupleValue` must be of the same type; they must not have a `TO Value`. The `VarTuple` type is the type of the `TupleValues`.

Semantics

`∃ x ∈ {1,2,3}: ψ(x)` reduces to `ψ(1) ∨ ψ(2) ∨ ψ(3)`. `∃ (x,y) ∈ {(a,1),(b,2)}: ψ(x,y)` reduces to `ψ(a,1) ∨ ψ(b,2)`.

`∀ x ∈ {1,2,3}: ψ(x)` reduces to `ψ(1) ∨ ψ(2) ∨ ψ(3)`. `∀ (x,y) ∈ {(a,1),(b,2)}: ψ(x,y)` reduces to `ψ(a,1) ∧ ψ(b,2)`.

5.4 “in” operator

Syntax

FO[Sugar] adds these production rules:

```
    Unary         ← { UNARY }  Enum;
    Enum          ← Base IN '{' Value { ',' Value } '}';
```

Notice that this extension redefines the syntax of `Unary` in FO[Core].

Well-formedness

The type of `Base` must be the same as the type of `Value`.

Semantics

`Base IN { Value1, ..., Valuen }` is the same as `Base=Value1 ∨ .. ∨ Base=Valuen`.

5.5 Partial interpretation of symbols

The goal is to support reasoning with partial interpretations of functions and predicates.

Lexicon

Token	Pattern	Example
INCLUDE	`:⊇|:>=`	

Syntax

FO[Sugar] adds these production rules:

```
   Enum ← NAME_S '(' Expression {',' Expression } ')' 'is' 'enumerated' ;
 Interpretation ← NAME_S INCLUDE SymbolInterpretation '.';
```

Well-formedness

1. `NAME_S` must be declared in the vocabulary and must be given an `Interpretation`.
2. The number and types of arguments of `NAME_S` in `Enum` must match its type signature.
3. The number and type of values `Interpretation` must match the type signature of the `NAME_S` being interpreted.
4. An `Interpretation` using `INCLUDE` must use a `TO` element after every tuple of values in its `SymbolInterpretation`. The `SymbolInterpretation` does not have to be total over its domain.

Semantics

1. A symbol interpretation given using `INCLUDE` is a **partial** interpretation of the symbol: the interpretation of the symbol is left unspecified for the tuples that are not interpreted.
2. `NAME_S(e1, .. , en) is enumerated` is `true` when the `Interpretation` of `NAME_S` is total or when the interpretation of the tuple `(e1, .. , en)` is specified in the `Interpretation` of `NAME_S`. It is `false`otherwise.

Example

The following theory requires that `material()` has a known maximum temperature.

```
    Material := {A, B, C}.
    maxTemp :⊇ {A→100}.  // maxTemp: Material -> Int
    maxTemp(material()) is enumerated.  // material: () -> Material
```

`material()` is thus `A` in every model.

CHAPTER

SIX

FO[PF]

FO[PF] extends FO[Sugar] to allow the use of partial functions, i.e., n-ary functions that are not totally defined over the cross-product of their argument types, `T1 ⨯ ... ⨯ Tn`.

It introduces syntax to declare subtypes, a subset relation between predicates, the domain and codomain of functions, and partial `Interpretations`.

6.1 Subtypes and subsets

This section describes how subtypes and subsets are declared.

Lexicon

Token	Pattern	Example
SUBSET	`<<	⊆`

Syntax

FO[PF] adds these production rules:

```
Declaration ← 'type' NAME_T [':=' TypeInterpretation] SUBSET NAME_T;

Declaration ← SymbolDecl '(' NAME_S SUBSET name_ST { TIMES name_ST } ')';
  name_ST <- type;
  name_ST <- NAME_S;
```

Well-formedness condition

1. (acyclicity) A `NAME_T` cannot be used after `SUBSET` before it has been declared as a type symbol. This prevents loops in the subtype-of relation.
2. (repetition) The `NAME_S` before `SUBSET` must be the name of the declared symbol.
3. (arity) A `NAME_ST` occuring in a cross-product must be either a `type` or a unary predicate. The number of factor in the cross-product must be the arity of the declared symbol.
4. (interpretation) A `NAME_T` must be given an interpretation in a vocabulary, theory or structure block.

Semantics

Consider these declaration

```
   type T << T1
   type T := {..} << T1

   p : T1 * .. * Tn -> Bool
   p : T1 * .. * Tn -> Bool (p << q)
   p : T1 * .. * Tn -> Bool (p << q1 * .. * qn)
```

The first two declarations say that `T` is a subtype of `T1`, i.e., that the interpretation of `T` is a subset of the interpretation of `T1`.

The next 3 declarations say that `p` has arity `n` and takes arguments of type `T1 * .. * Tn`. The fourth declaration says that `p` is a subset of `q` (where `q` also has type signature `T1 * .. * Tn -> Bool`). The fifth declaration says that `p` is a subset of the cross-product `q1 * .. * qn`, where the `qi` are previously-declared unary predicates.

Thanks to the acyclicity of the subtype-of relation, a predicate `P` always denotes a subset of a cross-product of type(s). This cross-product is determined by (recursively) looking up the signature of its superset.

Example

```
vocabulary {
   type LivingBeing := { Bob, Alice, Tom, Garfield, Kermit }
   type Person := { Bob, Alice, Tom} << LivingBeing
   married : LivingBeing → Bool  (married << Person)
}
```

6.2 Domain and codomain

Syntax

FO[PF] adds these production rules:

```
  Declaration ← SymbolDecl '(' 'total' ')';
  Declaration ← SymbolDecl '(' domain [ ',' 'codomain' ':' name_ST] ')';
  Declaration ← SymbolDecl '(' 'codomain' ':' name_ST ')';
    domain <- 'partial'
    domain <- 'domain' ':' name_ST  { TIMES name_ST }
```

Well-formedness condition

1. (acyclicity) A `name_ST` can occur in a domain or codomain only if it has been previously declared.
2. (arity) when the domain is a symbol `q`, `q` must have the same type signature as the declared symbol. When the domain is the cross-product of unary symbols `qi`, the number of `qi` must be the arity of the declared symbol, and each `qi` must have the type `Ti` of the declared symbol. The arity of the codomain must be one.
3. (Well-guardedness) Logic formulas in a theory must be *well-guarded* to ensure that the value of functions outside of their domain has no influence on the truth value of the theory. This can be achieved by using binary quantification and the non-strict (aka asymmetric) interpretation of the logic connectives in FO[Core] (including `if.. then..else`).
4. (For FO[ID]) A definition of `f` must define a value in the range of `f` for all tuples of arguments in the domain of `f`.

Semantics

Consider these declarations:

```
   f : T1 * .. * Tn -> T (total)
   f : T1 * .. * Tn -> T (domain: p1*..*pn, codomain: q)
   f : T1 * .. * Tn -> T (domain: p, codomain: q)
   f : T1 * .. * Tn -> T (partial)
```

where `T1,..,Tn` are previously declared (sub)types and `p, p1,.., pn, q` are previously-declared types or predicates.

All declarations say that `f` takes arguments of type `T1 * .. * Tn`. The `(total)` keyword is optional but recommended.

- The first declaration says that `f` is a total function over `T1 * .. * Tn`.
- The second declaration says that `f` is defined over exactly the cross-products of the sets p1*..*pn, and that its image is a subset of `q` (`T -> Bool`).
- The third declaration says that `f` is defined over exactly `p` (with type signature `T1*..*Tn -> Bool`), and that its image is a subset of `q` (`T -> Bool`).
- The fourth declaration says that `f` is defined over exactly `dom_f`, where `dom_f` is an implicitly declared predicate with type signature `T1*..*Tn -> Bool`.

Example

```
vocabulary {
   type LivingBeing := { Bob, Alice, Tom, Garfield, Kermit }
   type Person := { Bob, ALice, Tom} << LivingBeing
   married : LivingBeing → Bool      (married << Person)
   spouse: LivingBeing → LivingBeing (domain: married, codomain: married)
}
```

6.3 Interpretations

This section discusses the (possibly partial) interpretation of partial functions.

Syntax

(no change)

Well-formedness condition

1. (Partial predicate) Unlike in FO[Core], the `Interpretation` of a predicate `p` may use the → element, followed by `true` or `false`. When it uses the → element, it must use it for every `TupleValue`, and it may also use the `else` clause.
2. The `Value` in the `Interpretation` of a symbol σ must be in the domain and range of σ. The `Interpretation` of σ must be total over its domain. These conditions are checked at the syntax level when the `Interpretation` of the domain and/or range of σ is explicitly and totally given.

Semantics

If the `Interpretation` of σ is total and has no `else` part, it also specifies the interpretation of the domain of σ.

Example

```
vocabulary {
        type LivingBeing:= { Bob, Alice, Tom, Garfield, Kermit }
        person : LivingBeing → Bool
```

(continues on next page)

(continued from previous page)

```
        married : LivingBeing → Bool       (married << Person)
        spouse: LivingBeing → LivingBeing (domain: married, codomain: married)
}
theory {
        person := { Alice, Bob, Tom }
        spouse := { Alice → Bob; Bob → Alice }
}
```

This theory implies that `Alice` and `Bob` are the only `married` persons.

CHAPTER

SEVEN

FO[ID]

7.1 Goal

The goal of FO[ID] is to allow defining concepts in terms of other concepts, in a modular way. The definition can be inductive. ID stands for “Inductive Definition”.

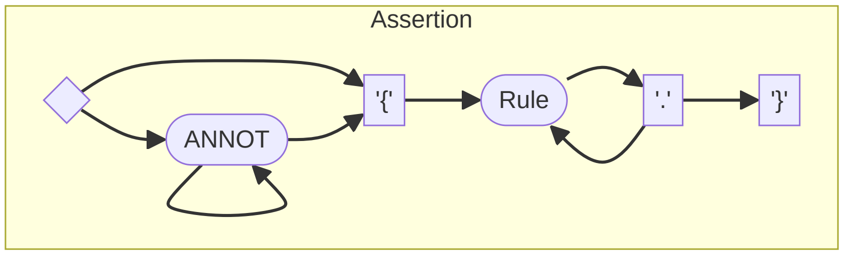
7.2 Lexicon

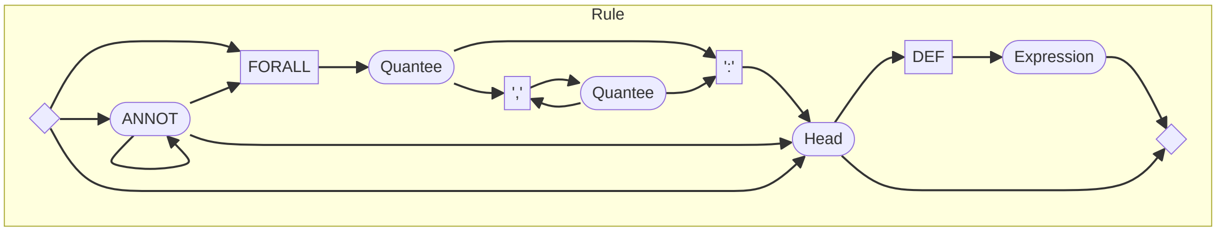
Token	Pattern	Example
DEF	`<-|←`	
FORALL	`[∀!]`	

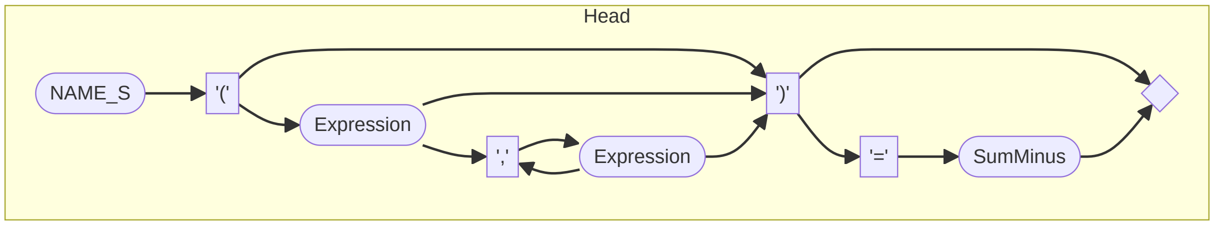
7.3 Syntax

FO[ID] adds these production rules:

```
Assertion ← { ANNOT } '{' Rule { Rule } '}';
  Rule ← { ANNOT } { FORALL Quantee {',' Quantee} ':' }
         Head [ DEF Expression ] '.';
    Head ← NAME_S '(' [ Expression {',' Expression } ] ')'
         [ '=' SumMinus ];
```


Assertion
ANNOT
'{'
Rule
'.'
'}'


Rule
ANNOT
FORALL
Quantee
','
Quantee
':'
Head
DEF
Expression


Head
NAME_S
'('
Expression
')'
','
Expression
'='
SumMinus

7.4 Well-formedness

1. The `Variable` occurring in a rule must be in the scope of a quantifier that declares it. The scope of the quantifiers in the head of a `Rule` is the whole rule.
2. `NAME_S` must have been declared in the vocabulary.
3. The number and types of `Expression` applied to `NAME_S` must match its type signature.
4. The = sign must occur in rule where `NAME_S` has been declared as a non-boolean function. The type of the `Expression` after = must be the same as the type of `NAME_S`.
5. The `Expression` after `DEF` must be boolean.

7.5 Semantics

An inductive definition specifies a unique interpretation for a predicate (or function) symbol, given the interpretation of its parameters. The *parameters* of a definition are the symbols that occur in it (by contrast, the tuple of *arguments* are the values applied to its symbol).

The semantics of a predicate definition is the well-founded semantics [Den00]. When the predicate is well-defined, i.e., the definition is not a paradox like `{ p() ← ¬p(). }`, the definition can be seen as the description of a constructive process. The construction process starts with an empty relation, i.e., the predicate is `false` for any tuple of values. Whenever the *body* of a rule (i.e. the `Expression` after `DEF`) is `true`, the tuple of values in the header is added to the relation. The process continues until a fix point is reached, i.e., no tuple of values can be added to the relation.

The construction process is similar for a function. The process starts with the function being undefined for every tuple of arguments. Whenever the body of a rule is `true`, the mapping of tuple of arguments to a value described in the header is added to the function interpretation. The process continues until a fix point is reached, i.e., no mapping can be added to the function interpretation.

The knowledge engineer should make sure that a defined function is defined over its whole domain, i.e., that the rules are such that the function is defined for every tuple of arguments in its domain, and for any interpretation of the parameters of the definition. He should also make sure that the rules of a function definition are not *conflicting*. Conflicting rules specify 2 different values for the same tuple of arguments. A reasoning engine may help him satisfy these requirements.

7.6 Examples

```
    [Definition of convex]
    { convex() ← [All angles are below 180°]
                  ∀n ∈ Side_Nr: angle(n)<180. }
```

`angle` is a parameter of the definition of `convex`. `convex` is a nullary predicate: it does not take any argument.

7.7 FO(ID) abstract language

The formal semantics of FO(ID) is described in [Den00]. FO(ID) can be reduced to FO by a process described in [PT05].

CHAPTER

EIGHT

FO[INT]

The goal of FO[Int] is to allow expressing knowledge about integer quantities using the 4 arithmetic operators and comparisons. It extends FO[PF].

It also allows creating and using types containing a range of elements, and the comparison of dates, e.g., to determine the applicability of a law..

8.1 Integer arithmetic

Lexicon

Token	Pattern	Example
COMPARISON	`=<|≤|<|~=|≠|=|>|≥|>=`	
INT	`Int|ℤ`	
INTEGER	`[+-]?\d+`	`-123`
MULT_DIV	`⍰|\*|\/|%`	
POWER	`\^`	
SUM_MINUS	`+|-`	
UNARY	`-|~|¬`	

Note that this extension extends the pattern of UNARY in FO[Core].

Syntax

FO[Int] adds these production rules:

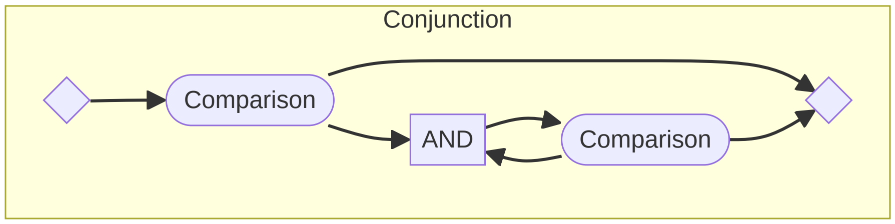
```
Type ← INT;

Conjunction  ← Comparison  {AND           Comparison};
Comparison   ← { ANNOT }
               SumMinus    {COMPARISON    SumMinus};
SumMinus     ← MultDiv     {SUM_MINUS     MultDiv};
MultDiv      ← Power       {MULT_DIV      Power};
Power        ← Unary       {POWER         Unary};
```

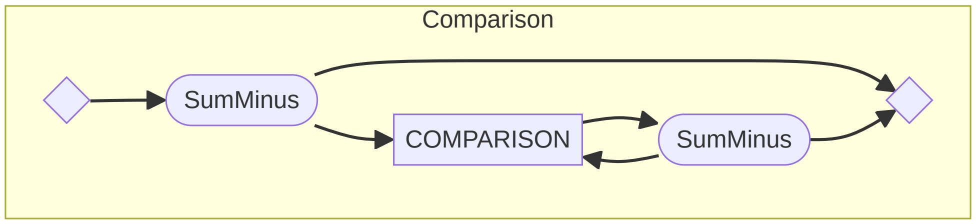
(continues on next page)

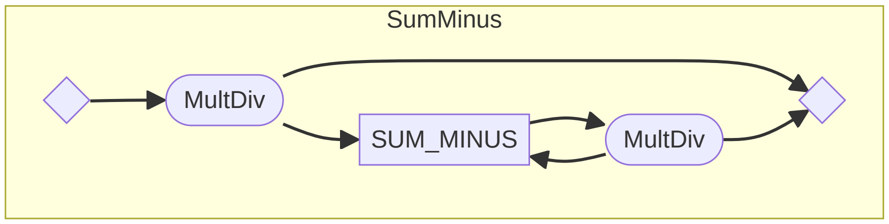
(continued from previous page)

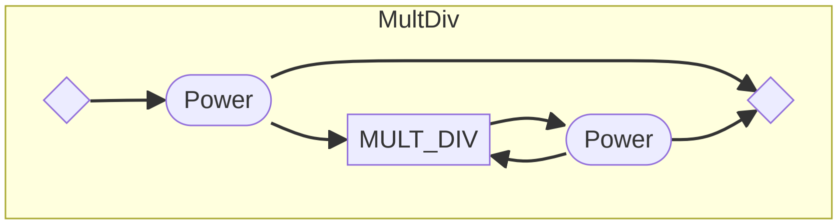
```
    Identifier ← INTEGER;
    Symbol ← 'abs';
```

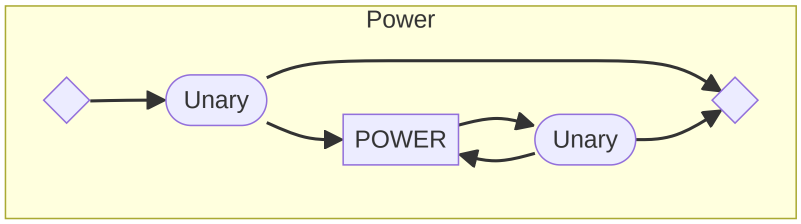
Note that this extension replaces the syntax of a Conjunction in FO[Core].


Conjunction
Comparison
AND
Comparison


Comparison
SumMinus
COMPARISON
SumMinus


SumMinus
MultDiv
SUM_MINUS
MultDiv


MultDiv
Power
MULT_DIV
Power



Well-formedness

1. Division must be well-guarded to prevent division by 0 (e.g., `if y ≠ 0 then x/y else 1`).

2. Quantification over ℤ is not allowed (relaxed in FO[Infinite]).

Semantics

The arithmetic and comparison operators have their usual meaning.

8.2 Range

Syntax

FO[Int] adds these production rules:

```
TypeInterpretation ← '{' INTEGER '..' INTEGER '}';
Quantee ← VARIABLE {',' VARIABLE} IN '{' INTEGER '..' INTEGER '}';
Enum    ← Base IN '{' INTEGER '..' INTEGER '}';
Identifier ← INTEGER;
```

Well-formedness

The first `INTEGER` must be below the second `INTEGER`.

If the interpretation of a range is not given in the vocabulary block (but in the theory or structure block), it is required to state that the declared type is a subtype of `Int`.

Semantics

`{1..8}` represents the set `{1, 2, 3, 4, 5, 6, 7, 8}`. The type that it interprets is automatically inferred to be a subtype of `Int`.

8.3 Date

Lexicon

Token	Pattern	Example
DATE	`#\d{4}-\d{2}-\d{2}`	`#2022-01-01`

Syntax

FO[Int] adds these production rules:

```
Type ← 'Date';
Identifier ← DATE | '#TODAY' ['(' INTEGER, INTEGER, INTEGER ')'];
```

Well-formedness

1. Dates can occur wherever integers can occur.
2. `DATE` must be a valid date.
3. Quantification over `Date` is not allowed (except with the FO[Infinite] extension).

Semantics

Dates are converted to proleptic Gregorian ordinals, where January 1 of year 1 has ordinal 1.

`#TODAY(y,m,d)` is today's date shifted by `y` years, `m` month and `d` days. `#TODAY(-1, 0, 0)` is today last year.

CHAPTER

NINE

FO[REAL]

Goal

The goal of FO[Real] is to allow expressing knowledge about real quantities using the 4 arithmetic operators and comparisons.

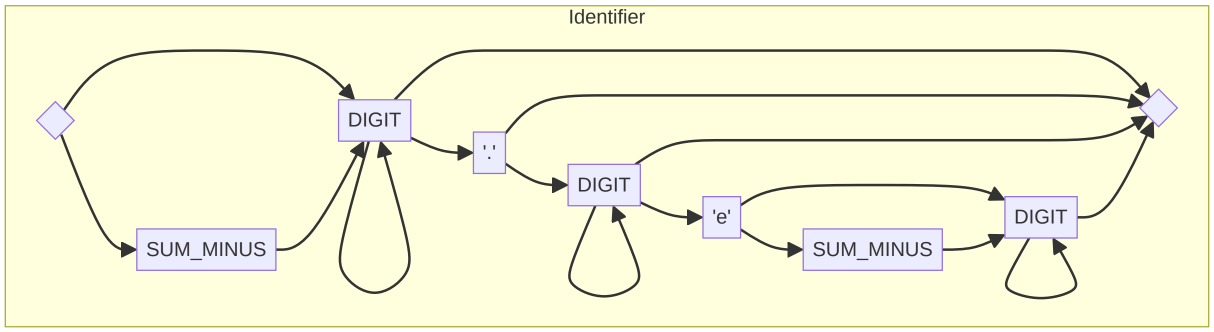
Lexicon

Token	Pattern	Example
DIGIT	`\d`	
REAL	`Real|ℝ`	
REALVAL	`[+-]?\d+(\.\d*(e[+-]?\d+))?`	`-0.01e-3`

Syntax

FO[Real] adds these production rules:

```
    Type ← REAL;
    Identifier ← REALVAL;
```



Well-formedness

1. Division must be well-guarded to prevent division by 0 (e.g., `if y ≠ 0 then x/y else 1.0`).

2. An arithmetic expression cannot mix integer and real numbers.
3. Quantification over $\mathbb{R}$ is not allowed (relaxed in FO[Infinite]).

Semantics

The arithmetic and comparison operators have their usual meaning.

TODO: discuss approximations

CHAPTER

TEN

FO[AGG]

Goal

The goal of FO[Agg] is to allow expressing knowledge about the size (aka cardinality) of a set, as well as aggregates (`sum`, `min`, and `max`). This extension extends FO[Int] and/or FO[Real].

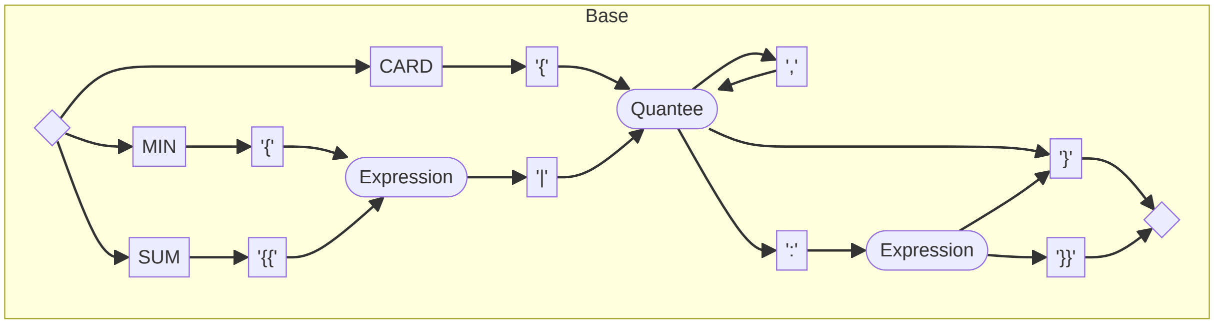
Lexicon

Token	Pattern	Example
CARD	`#|CARD`	
MIN	`min|max`	
SUM	`sum`	

Syntax

FO[Agg] adds these production rules:

```
    Base ← CARD '{' Quantee {',' Quantee}  [ ':' Expression ] '}';
    Base ← MIN '{' Expression '|' Quantee {',' Quantee}  [ ':'  Expression ] '}';
    Base ← SUM '{{' Expression '|' Quantee {',' Quantee} [ ':'  Expression ] '}}';
```



Well-formedness

1. in `CARD{ x∈T : ψ }`, ψ must be a boolean expression.

2. in `MIN{ t | x∈T: ψ }`, `t` must be a numeric term, and ψ a boolean expression.
3. in `SUM{{ t | x∈T: ψ }}`, `t` must be a numeric term, and ψ a boolean expression.
4. the variables in `t` and ψ must be `x` or declared in a outer quantification or aggregation.

Semantics

1. `#{ x∈T: ψ }` is the cardinality of the set of elements `x` in `T` that make ψ true.
2. `min{ t | x∈T: ψ }` is the minimum of `t` for each `x` in `T` such that ψ is true.
3. `sum{{ t | x∈T: ψ }}` is the sum of `t` for each `x` in `T` such that ψ is true.

Example

```
c() = #{x∈T: p(x)}.

[The perimeter is the sum of the lengths of the sides]
perimeter() = sum{{ length(n) | n ∈ Side_Nr }}.
```

FO(Agg) abstract language

The formal semantics of FO(Agg) is described in [DPB01].

CHAPTER

ELEVEN

FO[INFINITE]

Goal The goal of FO[Infinite] is to allow quantification over infinite domains. This extension extends FO[Int] and/or FO[Real].

Syntax

FO[Infinite] does not introduce new syntactic rules.

Well-formedness

1. Contrary to FO[Core], constructors can be recursive in FO[Infinite]: a `NAME_T` occurring as argument of a `CONSTRUCTOR` does not have to be *previously* declared (but it has to be declared). As a result, a constructed type may have an infinite domain.
2. Contrary to FO[Int], quantification over infinite types (ℤ, ℝ, Date) are allowed in FO[Infinite]. This also applies to binary quantifications using predicates over infinite domains (FO[Sugar].

Semantics

The formal semantics of a formula quantified over an infinite domain is a conjunction (or disjunction) of infinite length.

Example

The type for "lists of integers" can be declared as follows:

```
type List := constructed from { nil, cons(ℤ, List)}
```

`cons(1, cons(2, nil))` represents the list `[1,2]`.

Here is a quantification over integers:

```
∀ x ∈ ℤ: 0 ≤ x^2.
```

CHAPTER

TWELVE

FO[CONCEPT]

Goal

The goal of FO[Concept] is to allow quantification and aggregates over the concept of an ontology.

Syntax

FO[Concept] adds these production rules:

```
    Type ← 'Concept' '[' Signature ']';
    Identifier ← '`{NAME_S}';
    Symbol ← '$' '(' Expression ')';
```

Well-formedness

1. In `` `{NAME_S} ``, `NAME_S` must have been declared as a symbol.
2. The first argument of the `$` operator must be a Concept.
3. The number and types of the arguments applied to `$(x)` must be consistent with the signature of Concept `x`.
4. The type of `$(x)(y1, .. yn)` is the range in the signature of Concept `x`.

Semantic

FO[Concept] extends the domain of discourse with the “intension” of the symbols in the ontology. The intension of a symbol is the concepts it represents.

`Concept[T1 ⍰ ... ⍰ Tn → T]` is the type whose elements are the intensions of the symbol with signature `T1 ⍰ ... ⍰ Tn → T`. These elements are denoted by prepending the symbol with a back-tick: `` ` ``, e.g., `` `fever `` is the concept of symbol `fever`.

Example

The number of symptoms that a person `p` has can be defined as

```
    #{s in Concept[Person → ⍰]: symptom(s) ∧ $(s)(p)}
```

Or, using a binary quantification of FO[Sugar]:

```
    #{s in symtom: $(s)(p)}
```

FO(Concept) abstract language

The formal semantics of FO(Concept) is described in [CdHD22].

CHAPTER

THIRTEEN

FO[UNIT] ?

Note: This extension is work-in-progress.

Goal

This extension allows the use of synonyms for types INT and REAL, annotated with units of measurerement (e.g., `kg`). It also checks that equations are correct in their use of units of measure.

Lexicon

Token	Pattern	Example
UNIT	`[^\W\d_]+(\d+|²|³)?`	`m2, m², Ω`

Syntax

```
TypeInterpretation ← REAL '[' Units ']';
  Units ← UNIT {UNIT} ['/' UNIT {UNIT} ];
  Units ← '1' '/' UNIT {UNIT};
Identifier ← REALVAL '[' Units ']';
```

Well-formedness

1. `UNIT` are one of the 7 base units and the 22 derived units in the International System of Units (SI), with multiplying prefixes (e.g., `M`) and power suffix. Another 14 commonly-used non-SI units are also accepted.
2. Only numbers of the same type and comparable unit of measure can be added / substracted / compared. The multiplying prefixes are handled as usual.
3. The product (resp. division) of numbers of type `Real[A]` and `Real[B]` has type `Real[A B]` (resp. `Real[A/B]`).

Semantics

The interpretation of the type is the set of reals, annotated with the unit of measurement.

Example

```
vocabular {
    type Acceleration := Real[m/s²]
    a: () -> Acceleration
}
theory T {
    a() = 1[cm] / 1[s] / 1[s].
}
```

In every model of T, `a` has the value `0.01[m/s²]`.

CHAPTER

FOURTEEN

FO[·]

14.1 Lexicon

(Note: The patterns below can be tested online, e.g., using pythex.org)

Token	Pattern	Example
ACCESSOR	ID	
AND	`&|∧`	
ANNOT	`\[[^\]]*\]`	`[This is an annotation]`
BOOL	⍰ or `Bool`	
CARD	`#|CARD`	
COMPARISON	`=<|≤|<|~=|≠|=|>|≥|>=`	
CONSTRUCTOR	ID	
DATE	`#\d{4}-\d{2}-\d{2}`	`#2022-01-01`
DEF	`<-|←`	
DIGIT	`\d`	
EQUIVALENCE	`<=>|⇔`	
FORALL	`∀|!`	
ID	`[^\d\W]\w*\b`	`Color`
IMPLICATION	`=>|⇒`	
IN	`in|∈`	
INCLUDE	`⊇|>>`	
INT	`Int|ℤ`	
INTEGER	`[+-]?\d+`	`-123`
MIN	`min|max`	
MULT_DIV	`⍰|\*|\/|%`	
NAME_I	ID or `'[^']*'`	'John Doe'
NAME_S	ID	
NAME_T	ID	
NAME_TH	ID	
NAME_V	ID	
OR	`\\||∨`	
POWER	`\^`	
QUANTOR	`[∀!∃?]`	
R_IMPLICATION	`<=|⇐`	
REAL	`Real|ℝ`	
REALVAL	`[+-]?\d+(\.\d*(e[+-]?\d+))?`	`-0.01e-3`
SUBSET	`<<	⊆`
SUM	`sum`	

continues on next page

Table 1 – continued from previous page

Token	Pattern	Example
SUM_MINUS	+ | −	
TIMES	* or □	
TO	−> or →	
UNARY	− | ~ | ¬	
UNIT	`[^\W\d_]+(\d+|²|³)?`	`m2, m², Ω`
VARIABLE	ID	

14.2 Syntax

```
KnolwedgeBase ← vocabularyBlock { (VocabularyBlock | TheoryBlock | StructureBlock) };

VocabularyBlock ← 'vocabulary' [NAME_V] '{' { Declaration } '}';
  Declaration ← 'type' NAME_T [':=' TypeInterpretation] ;
    TypeInterpretation ← '{' NAME_I { ',' NAME_I } '}';
    TypeInterpretation ← 'constructed' 'from'
                         '{' ConstructorDeclaration { ',' ConstructorDeclaration } '}
↪';
      ConstructorDeclaration ← NAME_I;
      ConstructorDeclaration ← CONSTRUCTOR '(' [ACCESSOR ':' ] Type
                               {',' [ACCESSOR ':' ] Type} ')';
        Type ← NAME_T | BOOL;
        Type ← INT | Date;                                                *FO[Int]
        Type ← REAL;                                                      *FO[Real]
        Type ← 'Concept' '[' Signature ']';                              ␣
↪*FO[Concept]
    TypeInterpretation ← '{' INTEGER '..' INTEGER '}';                    *FO[Int]
    TypeInterpretation ← REAL '[' Units ']';                              *FO[Unit]
      Units ← UNIT {UNIT} ['/' UNIT {UNIT} ];                             *FO[Unit]
      Units ← '1' '/' UNIT {UNIT};                                        *FO[Unit]
  Declaration ← 'type' NAME_T [':=' TypeInterpretation] SUBSET NAME_T;    *FO[PF]

  Declaration ← SymbolDecl;
    SymbolDecl ← { ANNOT } NAME_S { ',' NAME_S } ':' Signature;
      Signature ← '(' [ Type { TIMES Type } ] ')' TO Type;
      Signature ←  [ Type { TIMES Type } ] TO Type;
  Declaration ← SymbolDecl '(' NAME_S SUBSET name_ST { TIMES name_ST } ')'; *FO[PF]
    name_ST <- type;                                                      *FO[PF]
    name_ST <- NAME_S;                                                    *FO[PF]
  Declaration ← SymbolDecl '(' 'total' ')';                               *FO[PF]
  Declaration ← SymbolDecl '(' domain [ ',' 'codomain' ':' name_ST] ')';  *FO[PF]
  Declaration ← SymbolDecl '(' 'codomain' ':' name_ST ')';                *FO[PF]
    domain <- 'partial'                                                   *FO[PF]
    domain <- 'domain' ':' name_ST  { TIMES name_ST }                     *FO[PF]

  Declaration ← VARIABLE IN ( Type | Symbol );                            *FO[Sugar]
  Declaration ← 'import' NAME_V;

TheoryBlock ← 'theory' [ NAME_TH [: NAME_V] ] '{' { Assertion } '}';
  Assertion ← Interpretation;
  Assertion ← { ANNOT } '{' Rule { Rule } '}';                            *FO[ID]
    Rule ← { ANNOT } { FORALL Quantee {',' Quantee} ':' }
           Head [ DEF Expression ] '.';                                   *FO[ID]
```

(continues on next page)

(continued from previous page)

```
      Head ← NAME_S '(' [ Expression {',' Expression } ] ')'
             [ '=' SumMinus ];                                           *FO[ID]

  Assertion ← Expression '.';
  Expression ← [ { ANNOT } QUANTOR Quantee {',' Quantee} ':' ] RImplication;
      Quantee ← VARIABLE {',' VARIABLE} IN Type;
      Quantee ← VARIABLE {',' VARIABLE} [ IN Symbol ];                   *FO[Sugar]
      Quantee ← VarTuple {',' VarTuple} IN Symbol;                       *FO[Sugar]
        VarTuple ← '(' VARIABLE { ',' VARIABLE } ')';                    *FO[Sugar]
      Quantee ← VARIABLE {',' VARIABLE} IN '{' Value { ',' Value } '}';  *FO[Sugar]
      Quantee ← VarTuple {',' VarTuple} IN
                  '{' TupleValue { ',' TupleValue } '}';                 *FO[Sugar]
      Quantee ← VARIABLE {',' VARIABLE} IN '{' INTEGER '..' INTEGER '}'; *FO[Int]
      RImplication ← Equivalence [R_IMPLICATION Equivalence];
      Equivalence  ← Implication [EQUIVALENCE   Implication];
      Implication  ← Disjunction [IMPLICATION   Disjunction];
      Disjunction  ← Conjunction {OR            Conjunction};
      Conjunction  ← Comparison  {AND           Comparison};
      Comparison   ← { ANNOT }
                     SumMinus    {COMPARISON    SumMinus};               *FO[Int]
      SumMinus     ← MultDiv     {SUM_MINUS     MultDiv};                *FO[Int]
      MultDiv      ← Power       {MULT_DIV      Power};                  *FO[Int]
      Power        ← Unary       {POWER         Unary};                  *FO[Int]
      Unary        ← { UNARY }   Enum;
      Enum         ← Base;
      Enum         ← Base IN '{' Value { ',' Value } '}';                *FO[Sugar]
      Enum         ← NAME_S '(' Expression {',' Expression } ')'
                     'is' 'enumerated' ;                                 *FO[Sugar]
      Enum         ← Base IN '{' INTEGER '..' INTEGER '}';              *FO[Int]

      Base ← CARD '{' Quantee {',' Quantee}  [ ':' Expression ] '}';    *FO[Agg]
      Base ← MIN '{' Expression '|' Quantee {',' Quantee}  [ ':'  Expression ] '}';
↪*FO[Agg]
      Base ← SUM '{{' Expression '|' Quantee {',' Quantee} [ ':'  Expression ] '}}';
↪*FO[Agg]
      Base ← 'if' Expression 'then' Expression 'else' Expression;
      Base ← { ANNOT } QUANTOR Quantee {',' Quantee} ':' RImplication;
      Base ← Symbol '(' [Expression {',' Expression}] ')';
      Base ← Identifier | VARIABLE;
      Base ← { ANNOT } '(' Expression ')';

        Symbol ← NAME_S;
        Symbol ← CONSTRUCTOR | ACCESSOR | 'is_{CONSTRUCTOR}';
        Symbol ← 'abs';                                                  *FO[Int]
        Symbol ← '$' '(' Expression ')';                                ␣
↪*FO[Concept]

        Identifier ← 'true' | 'false' | NAME_I;
        Identifier ← INTEGER;                                            *FO[Int]
        Identifier ← DATE | '#TODAY' ['(' INTEGER, INTEGER, INTEGER ')']; *FO[Int]
        Identifier ← REALVAL;                                            *FO[Real]
        Identifier ← '`{NAME_S}';                                        ␣
↪*FO[Concept]
        Identifier ← REALVAL '[' Units ']';                              *FO[Unit]

StructureBlock ← 'structure' [ NAME_TH [: NAME_V] ] '{' { Interpretation } '}';
  Interpretation ← NAME_T ':=' TypeInterpretation '.';
```

(continues on next page)

(continued from previous page)

```
Interpretation ← NAME_S ':=' Value '.';
Interpretation ← NAME_S ':=' SymbolInterpretation '.';
Interpretation ← NAME_S INCLUDE SymbolInterpretation '.';                  *FO[Sugar]
  Value ← Identifier;
  Value ← CONSTRUCTOR '(' Value {',' Value } ')';
  SymbolInterpretation ← '{' TupleValue { ',' TupleValue } '}';
  SymbolInterpretation ← '{' TupleValue TO Value { ',' TupleValue TO Value} '}'␣
↪[else];
    TupleValue ← Value;
    TupleValue ← '(' Value { ',' Value } ')';
  else ← 'else' Value;                                                     *FO[Sugar]
```

CHAPTER FIFTEEN

BIBLIOGRAPHY

CHAPTER

SIXTEEN

INDEX

BIBLIOGRAPHY

[CdHD22] Pierre Carbonnelle, Matthias Van der Hallen, and Marc Denecker. Quantification and aggregation over concepts of the ontology. 2022. arXiv:2202.00898.

[CVVD22] Pierre Carbonnelle, Simon Vandevelde, Joost Vennekens, and Marc Denecker. Idp-z3: a reasoning engine for fo(.). 2022. arXiv:2202.00343.

[Den00] Marc Denecker. Extending classical logic with inductive definitions. In John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Lu\'ıs Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic - CL 2000, First International Conference, London, UK, 24-28 July, 2000, Proceedings*, volume 1861 of Lecture Notes in Computer Science, 703–717. Springer, 2000. URL: https://doi.org/10.1007/3-540-44957-4\TU\textbackslash{}_47, doi:10.1007/3-540-44957-4_47.

[DPB01] Marc Denecker, Nikolay Pelov, and Maurice Bruynooghe. Ultimate well-founded and stable semantics for logic programs with aggregates. In Philippe Codognet, editor, *Logic Programming, 17th International Conference, ICLP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, volume 2237 of Lecture Notes in Computer Science, 212–226. Springer, 2001. URL: https://doi.org/10.1007/3-540-45635-X\TU\textbackslash{}_22, doi:10.1007/3-540-45635-X_22.

[JL08] Philip Johnson-Laird. *How we reason*. Oxford University Press, 2008.

[PT05] Nikolay Pelov and Eugenia Ternovska. Reducing inductive definitions to propositional satisfiability. In *International Conference on Logic Programming*, 221–234. Springer, 2005.

[WMarienD14] Johan Wittocx, Maarten Mariën, and Marc Denecker. Grounding FO and FO(ID) with bounds. *CoRR*, 2014. URL: http://arxiv.org/abs/1401.3840, arXiv:1401.3840.

INDEX

Symbols

"in" operator, 46

B

Binary quantification, 43

D

Direct quantification, 46

F

FO[Agg], 69
FO[Concept], 75
FO[Core], 7
FO[ID], 52
FO[Infinite], 73
FO[Int], 58
FO[PF], 48
FO[Real], 66
FO[Sugar], 42
FO[Unit], 77
FO[·], 3

K

Knowledge Base, 9

P

Partial interpretation, 47

R

Range, 66

S

Structure block, 13

T

Theory block, 12
Type declaration, 16
Type inference, 43

V

Variable declaration, 43
Vocabulary block, 11