

---

**FO-dot**

**Pierre Carbonnelle, Marc Denecker**

**Sep 30, 2022**



## CONTENTS

|                        |           |
|------------------------|-----------|
| <b>1 Introduction</b>  | <b>3</b>  |
| <b>2 Rationale</b>     | <b>5</b>  |
| <b>3 Notation</b>      | <b>7</b>  |
| <b>4 FO[Core]</b>      | <b>9</b>  |
| <b>5 FO[Sugar]</b>     | <b>19</b> |
| <b>6 FO[ID]</b>        | <b>23</b> |
| <b>7 FO[Int]</b>       | <b>25</b> |
| <b>8 FO[Real]</b>      | <b>29</b> |
| <b>9 FO[Agg]</b>       | <b>31</b> |
| <b>10 FO[Infinite]</b> | <b>33</b> |
| <b>11 FO[Concept]</b>  | <b>35</b> |
| <b>12 FO[PF] ?</b>     | <b>37</b> |
| <b>13 FO[Unit] ?</b>   | <b>41</b> |
| <b>14 FO[.]</b>        | <b>43</b> |
| <b>15 Bibliography</b> | <b>47</b> |
| <b>16 Index</b>        | <b>49</b> |
| <b>Bibliography</b>    | <b>51</b> |
| <b>Index</b>           | <b>53</b> |



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](#).



## 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
- 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:

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

- PF** Partial functions
- 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.
-



## 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:

|   |
|---|
| $\forall t : \text{Open}(t) \Rightarrow \exists s : \text{Secretary}(s) \wedge \text{at}(s, \text{EntranceDesk}, t).$ |
|---|

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 its language constructs  $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow, \forall, \exists$  are basic indispensable constructs 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.

## NOTATION

The syntax of FO and its extensions is specified using the [EBNF notation](#).

The lexicon lists the tokens of the language. They are specified using [Python Regular Expressions](#). For example, an identifier satisfies `[\d\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.



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 } '}' ;
```

### 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 axioms
}

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 $\mathbb{B}$ or Bool |            |
| CONSTRUCTOR | ID                          |            |
| NAME_T      | ID                          |            |
| NAME_I      | ID or $[\text{^}]^*$        | '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;

```

### 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  $2^3 = 8$  elements.

## 4.4 Function and Predicate declaration

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

### Lexicon

| Token  | Pattern                | Example                 |
|--------|------------------------|-------------------------|
| ANNOT  | <code>\[[^\]]*\</code> | [This is an annotation] |
| NAME_S | ID                     |                         |
| TIMES  | <code>*   0</code>     |                         |
| TO     | <code>-&gt;   →</code> |                         |

### Syntax

```

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

```

### 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 : () → Bool
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 Axiom

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 ← Axiom;
Axiom ← 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};
```

(continues on next page)

(continued from previous page)

```

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.

### 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 axiom 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  $\forall x, y \in T: p(x, y) \vee f(f(x))=y.$ , 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 } ')';

```

(continues on next page)

(continued from previous page)

```

SymbolInterpretation ← '{' TupleValue { ',' TupleValue } '}' ;
SymbolInterpretation ← '{' TupleValue TO Value { ',' TupleValue TO Value } '}' ;
↪[else];
  TupleValue ← Value;
  TupleValue ← '(' Value { ',' Value } ')';
else ← 'else' 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  $T_1 \square \dots \square T_n \rightarrow T$ , where  $0 \leq n$ . (A 0-signature is noted  $() \rightarrow T$ )

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  $\Sigma$ ) given a typing function  $\tau$ , noted  $\tau(\cdot)$ , is a type-symbol partially defined by induction as follows (terminal symbols are underlined for clarity):

|   |           |   |
|---|-----------|---|
|   | $(\cdot)$ | if  |
| <u>true</u>   |           |   |
| <u>false</u>  |           |   |
| <u>x</u>  | $(x)$     | x is a variable   |
| <u><math>(t_1, \dots, t_n)</math></u>                 | $T$       | $(\tau, T_1 \times \dots \times T_n \rightarrow T) \in \Sigma$ and $\forall i \in [1, n] : \tau(t_i) = T_i$ |
| <u><math>\forall</math></u>                           |           | $(\tau) = T$ and $(\tau) = T$   |
| <u><math>\exists</math></u>                           |           | $(\tau) = T$  |
| <u><math>\exists x \in T :</math></u>                 |           | x is a variable and $(\tau, [x : T]) = T$   |
| <u><math>t_1 = t_2</math></u>                         |           | $\tau(t_1) = \tau(t_2)$   |
| <b>if then <math>t_1</math> else <math>t_2</math></b> | $T$       | $(\tau) = T$ and $(\tau, t_1) = T = (\tau, t_2)$  |
| otherwise   | undefined |   |

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

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

| Convenient notation                   | Equivalent formula                |
|---------------------------------------|-----------------------------------|
| $\Delta$                              | $\neg(\neg\forall\neg)$           |
| $\exists$                             | $\neg\forall$                     |
| $\forall$                             | $\neg\exists$                     |
| $\exists$                             | $(\wedge)\forall(\neg\wedge\neg)$ |
| <u><math>\forall x \in T :</math></u> | $\neg\exists x \in T : \neg$      |

A (total) *structure* over  $\Sigma$  consists of :

- an object domain  $D$  containing  $\perp$ , 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  $T_1 \times \dots \times T_n \rightarrow T$  to (total) functions from  $T_1 \times \dots \times T_n$  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  $\phi$  in  $(\tau)$ , denoted  $\llbracket \phi \rrbracket^{\tau} = v$ , is an element of  $D$  partially defined by induction as follows:

|  |   |   |
|--|---|---|
|  | $\llbracket \cdot \rrbracket^I$                                     | if  |
| $\underline{true}$                     | $true$  |   |
| $\underline{false}$                    | $false$   |   |
| $x$                                    | $(x)$   | $x$ is a variable in the domain of  |
| $(t_1, \dots, t_n)$                    | $(\llbracket t_1 \rrbracket^I, \dots, \llbracket t_n \rrbracket^I)$ | $(, T_1 \boxtimes \dots \boxtimes T_n \rightarrow T) \in$ and $\forall i \in [1, n] : \llbracket t_i \rrbracket^I$ is defined |
| $\underline{\forall}$                  | $false$   | $\llbracket \cdot \rrbracket^I$ is defined and false  |
| $\underline{\vee}$                     | $\llbracket \cdot \rrbracket^I \vee \llbracket \cdot \rrbracket^I$  | $\llbracket \cdot \rrbracket^I$ and $\llbracket \cdot \rrbracket^I$ are defined   |
| $\underline{\neg}$                     | $\neg \llbracket \cdot \rrbracket^I$                                | $\llbracket \cdot \rrbracket^I$ is defined  |
| $\underline{\exists x \in T} :$        | $\exists d \in T^I : \llbracket \cdot \rrbracket^I[x : d]$          | $\llbracket \cdot \rrbracket^I[x : d]$ is defined for every $d$ in $T^I$  |
| $t_1 \equiv t_2$                       | $\llbracket t_1 \rrbracket^I = \llbracket t_2 \rrbracket^I$         | $\llbracket t_1 \rrbracket^I$ and $\llbracket t_2 \rrbracket^I$ are defined   |
| <b>if then</b> $t_1$ <b>else</b> $t_2$ | $\llbracket t_1 \rrbracket^I$                                       | $\llbracket \cdot \rrbracket^I = true$ and $\llbracket t_1 \rrbracket^I$ is defined   |
| <b>if then</b> $t_1$ <b>else</b> $t_2$ | $\llbracket t_2 \rrbracket^I$                                       | $\llbracket \cdot \rrbracket^I = false$ and $\llbracket t_2 \rrbracket^I$ is defined  |
| otherwise                              | undefined   |   |

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

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



## 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 } ')';
```

### 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  $T_1 \times \dots \times T_n \rightarrow \mathbb{Q}$ .

$\exists x_1 \in P: \psi$  reduces to  $\exists x_1 \in T_1: P(x_1) \wedge \psi$  (where  $P$  is unary). It can also be written  $\exists x_1: \psi. \exists (x_1, \dots, x_n) \in P: \psi$  reduces to  $\exists x_1 \in T_1, \dots, x_n \in T_n: P(x_1, \dots, x_n) \wedge \psi$ .

Similarly,  $\forall x_1 \in P: \psi$  reduces to  $\forall x_1 \in T_1: P(x_1) \Rightarrow \psi$  (where  $P$  is unary). It can also be written  $\forall x_1: \psi. \forall (x_1, \dots, x_n) \in P: \psi$  reduces to  $\forall x_1 \in T_1, \dots, x_n \in T_n: P(x_1, \dots, x_n) \Rightarrow \psi$ .

---

**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 → ℚ, ∀ 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

$\exists x \in \{1, 2, 3\}: \varphi(x)$  reduces to  $\varphi(1) \vee \varphi(2) \vee \varphi(3)$ .  $\exists (x, y) \in \{(a, 1), (b, 2)\}: \varphi(x, y)$  reduces to  $\varphi(a, 1) \vee \varphi(b, 2)$ .

$\forall x \in \{1, 2, 3\}: \varphi(x)$  reduces to  $\varphi(1) \wedge \varphi(2) \wedge \varphi(3)$ .  $\forall (x, y) \in \{(a, 1), (b, 2)\}: \varphi(x, y)$  reduces to  $\varphi(a, 1) \wedge \varphi(b, 2)$ .

## 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 | : $\exists$   :>= |         |

### 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 := $\exists$  {A→100}. // maxTemp: Material -> Int
maxTemp(material()) is enumerated. // material: () -> Material
```

material() is thus A in every model.



## 6.1 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”.

## 6.2 Lexicon

| Token  | Pattern                      | Example |
|--------|------------------------------|---------|
| DEF    | $\leftarrow \mid \leftarrow$ |         |
| FORALL | $[\forall!]$                 |         |

## 6.3 Syntax

FO[ID] adds these production rules:

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

## 6.4 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.

## 6.5 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  $\{ p() \leftarrow \neg p() . \}$ , 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.

## 6.6 Examples

```
[Definition of convex]
{ convex() ← [All angles are below 180°]
  Vn ∈ 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.

## 6.7 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].

## FO[INT]

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

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..

### 7.1 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)

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

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

### Well-formedness

1. Division must be well-guarded to prevent division by 0 (e.g., if  $y \neq 0$  then  $x/y$  else 1).
2. Quantification over  $\mathbb{Z}$  is not allowed (relaxed in FO[Infinite]).

### Semantics

The arithmetic and comparison operators have their usual meaning.

## 7.2 Range

### Lexicon

| Token | Pattern                   | Example |
|-------|---------------------------|---------|
| POINT | <code>[a-zA-Z]*\d+</code> | R1      |

### Syntax

FO[Int] adds these production rules:

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

### Well-formedness

For TypeInterpretation

1. The alphabetic prefix in the first POINT must be the same as the one in the second POINT. It must be declared only once in a TypeInterpretation.
2. The natural number in the first POINT must be below the natural number in the second POINT.

For Quantee and Enum:

1. The alphabetic prefix in the first POINT must be the same as the one in the second POINT. It must have been declared in a TypeInterpretation.
2. The natural number in the first POINT must be below the natural number in the second POINT. They must be within the range declared by the TypeInterpretation.

For Identifier:

1. The alphabetic prefix in the POINT must have been declared in a TypeInterpretation.
2. The natural number in the POINT must be within the range declared by the TypeInterpretation.

A POINT may participate in integer arithmetic operations and comparisons as an integer (ignoring the alphabetic prefix).

### Semantics

`{R1..R8}` represents the set  $\{R1, R2, R3, R4, R5, R6, R7, R8\}$ .

## 7.3 Date

### Lexicon

| Token | Pattern                         | Example                  |
|-------|---------------------------------|--------------------------|
| DATE  | <code>#\d{4}-\d{2}-\d{2}</code> | <code>#2022-01-01</code> |

### 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.





## 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   | <code>\d</code>                             |                       |
| REAL    | <code>Real   <math>\mathbb{R}</math></code> |                       |
| REALVAL | <code>[+-]?\d+(\.\d*(e[+-]?\d+))?</code>    | <code>-0.01e-3</code> |

### 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



## 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 |
|-------|-------------|---------|
| AGG   | sum min max |         |
| CARD  | # CARD      |         |

### Syntax

FO[Agg] adds these production rules:

```
Base ← CARD '{' Quantee {' ',' Quantee} ':' Expression '}';  
Base ← AGG '(' 'lambda' Quantee {' ',' Quantee} ':' Expression ')';
```

### Well-formedness

1. in  $CARD\{x \in T : \varphi\}$ ,  $\varphi$  must be a boolean expression.
2. in  $AGG(\text{lambda } x \in T : \varphi)$ ,  $\varphi$  must be a number expression.

### Semantics

1.  $\#\{x \in T : \varphi\}$  is the cardinality of the set of elements  $x$  in  $T$  that make  $\varphi$  true.
2.  $\text{sum}(\text{lambda } x \in T : \varphi)$  is the sum of  $\varphi$  for each  $x$  in  $T$ .

### Example

```
c() = #{x ∈ T : p(x)}.
```

```
[The perimeter is the sum of the lengths of the sides]  
perimeter() = sum(lambda n ∈ Side_Nr : length(n)).
```

### FO(Agg) abstract language

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



## 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 ( $\mathbb{Z}$ ,  $\mathbb{R}$ , 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( $\mathbb{Z}$ , List) }
```

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

Here is a quantification over integers:

```
 $\forall x \in \mathbb{Z}: 0 \leq x^2.$ 
```



## 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  $\$(x)$  ( $y_1, \dots, y_n$ ) 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  $[T_1 \ \& \dots \ \& \ T_n \ \rightarrow \ T]$  is the type whose elements are the intensions of the symbol with signature  $T_1 \ \& \dots \ \& \ T_n \ \rightarrow \ T$ . 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 syptom: $(s)(p)}
```

### FO(Concept) abstract language

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





---

**Note:** This extension is work-in-progress.

---

FO[PF] extends FO[Sugar] to allow the use of partial functions (including partial predicates), i.e., n-ary functions that are not totally defined over  $T_1 \times \dots \times T_n$ .

It introduces “unary predicates in type signature”, the `partial` and `dom` keywords, and special rules for `Interpretation`.

## 12.1 Unary predicates in type signature

This section describes how unary predicates can be used to specify the domain and range of partial functions.

### Syntax

FO[PF] adds this production rule:

```
Type ← NAME_S;
```

### Well-formedness condition

1. A `NAME_S` cannot be used as a `Type` before it has been declared as a **unary predicate** symbol.
2. A `NAME_S` cannot be used as a `Type` in a `ConstructorDeclaration`.
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 interpretation of the logic connectives in FO[Core] (including `if..then..else`).

### Semantics

A unary predicate `P` with signature  $T \rightarrow \text{Bool}$  denotes a subset  $P^I$  of  $T^I$  in structure  $I$ . A unary predicate `Q` with signature  $P \rightarrow \text{Bool}$  denotes a subset  $Q^I$  of  $P^I$ . A n-ary predicate `R` with signature  $T_1 \times \dots \times T_n \rightarrow \text{Bool}$  (where any  $T_i$  can be a unary predicate) denotes a subset  $R^I$  of  $T_1^I \times \dots \times T_n^I$ .

Thus, 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 `P`.

A function `f` with signature  $P \rightarrow Q$  denotes a *total* function from the set  $P^I$  to the set  $Q^I$ . The function has no value outside of  $P^I$ . A function `f` with signature  $T_1 \times \dots \times T_n \rightarrow T$  (where `T` and any  $T_i$  can be a unary predicate) denotes a *total* function from the set  $T_1^I \times \dots \times T_n^I$  to the set  $T^I$ .

(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`.

### Example

```

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

```

## 12.2 Partial and dom keywords

This section presents an alternative way to declare partial functions, for use when the domain is not a cross-product of sets.

### Syntax

```

Declaration ← { ANNOT } 'partial' NAME_S { ',' NAME_S } ':' Signature;
Symbol ← 'dom' '(' Expression ')';

```

### Well-formedness condition

1. The argument of `dom` must be a Concept.
2. (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 interpretation of the logic connectives in FO[Core] (including `if.. then..else`).

### Semantics

For a function  $f$  declared as `partial f: T1  $\times$  ...  $\times$  Tn  $\rightarrow$  T`, formula `dom(`f)` denotes the subset of  $T1 \times \dots \times Tn$  that is the domain of  $f$ , i.e., the set of (tuple of) values for which  $f$  is defined. Thus, the signature of `dom(`f)` is  $T1 \times \dots \times Tn \rightarrow \text{Bool}$ . The signature of  $f$  may use unary predicates.

(For FO[ID]) A definition of  $f$  must define a value of the appropriate type for all tuples of arguments in the domain of  $f$ .

### Example

```

vocabulary {
  partial f: Real  $\times$  Real  $\rightarrow$  Real
}
theory {
  { !x,y in Real: dom(`f)(x,y)  $\leftarrow$  x  $\neq$  y.
    !x,y in dom(`f): f(x,y) = 1/(x-y). }
}

```

## 12.3 Interpretations

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

### Syntax

FO[PF] adds these production rules:

```

Interpretation ← 'dom' '(' '{NAME_S}' ')' ':=' SymbolInterpretation '.';

```

### Well-formedness condition

1. (Partial predicate) Unlike in FO[Core], the Interpretation of a predicate  $p$  may use the  $\rightarrow$  element, followed by true or false. When it uses the  $\rightarrow$  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  $\sigma$  must be in the domain and range of  $\sigma$ . The Interpretation of  $\sigma$  must be total over its domain. These conditions are checked at the syntax level when the Interpretation of the domain and/or range of  $\sigma$  is explicitly and totally given. These conditions also apply to the Interpretation of  $\text{dom}(\sigma)$ .

### Semantics

If the Interpretation of  $\sigma$  is total and has no else part, it also specifies the interpretation of the domain of  $\sigma$ .

### Example

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

This theory implies that Alice and Bob are the only spouse.



## 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 measurement (e.g., kg). It also checks that equations are correct in their use of units of measure.

### Lexicon

| Token | Pattern                            | Example                           |
|-------|------------------------------------|-----------------------------------|
| UNIT  | <code>[^\W\d_]+(\d+ ^2 ^3)?</code> | <code>m2, m<sup>2</sup>, Ω</code> |

### 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 / subtracted / 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/s<sup>2</sup>].



## 14.1 Lexicon

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

| Token         | Pattern                     | Example                 |
|---------------|-----------------------------|-------------------------|
| ACCESSOR      | ID                          |                         |
| AGG           | sum min max                 |                         |
| 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                    |
| MULT_DIV      | Ⓜ \* \/ %                   |                         |
| NAME_I        | ID or '[^']*'               | 'John Doe'              |
| NAME_S        | ID                          |                         |
| NAME_T        | ID                          |                         |
| NAME_TH       | ID                          |                         |
| NAME_V        | ID                          |                         |
| OR            | \  ∨                        |                         |
| POWER         | \^                          |                         |
| QUANTOR       | [∀!∃?]                      |                         |
| POINT         | [a-zA-Z]*\d+                | R1                      |
| R_IMPLICATION | <= ←                        |                         |
| REAL          | Real ℝ                      |                         |
| REALVAL       | [+-]?\d+(\.\d*(e[+-]?\d+))? | -0.01e-3                |
| SUM_MINUS     | + -                         |                         |

continues on next page

Table 1 – continued from previous page

| Token    | Pattern  | Example                       |
|----------|--|-------------------------------|
| TIMES    | * or $\boxtimes$   |                               |
| TO       | -> or $\rightarrow$  |                               |
| UNARY    | -   ~   $\neg$   |                               |
| UNIT     | $[\wedge \backslash W \backslash d _ ] + ( \backslash d +   ^ 2   ^ 3 ) ?$ | m2, m <sup>2</sup> , $\Omega$ |
| 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 } '}'
↳ ;
TypeInterpretation ← '{' POINT '..' POINT '}' ;
TypeInterpretation ← REAL '[' Units ']' ;
Units ← UNIT {UNIT} [ '/' UNIT {UNIT} ] ;
Units ← '1' '/' UNIT {UNIT} ;
ConstructorDeclaration ← NAME_I ;
ConstructorDeclaration ← CONSTRUCTOR '(' [ACCESSOR ':' ] Type
  '{',' [ACCESSOR ':' ] Type}' ')';
  Type ← NAME_T | BOOL;
  Type ← INT | Date;
  Type ← REAL;
  Type ← 'Concept' '[' Signature ']' ;
↳ *FO[Concept]
  Type ← NAME_S;

Declaration ← { ANNOT } NAME_S { ',' NAME_S } ':' Signature;
Declaration ← { ANNOT } 'partial' NAME_S { ',' NAME_S } ':' Signature;
Signature ← '(' [ Type { TIMES Type } ] ')' TO Type;
Signature ← Type { TIMES Type } TO Type;
Declaration ← VARIABLE IN ( Type | Symbol );
Declaration ← 'import' NAME_V;

TheoryBlock ← 'theory' [ NAME_TH [ : NAME_V ] ] '{' { Assertion } '}' ;
Assertion ← Interpretation;
Assertion ← { ANNOT } '{' Rule { Rule } '}' ;
Rule ← { ANNOT } { FORALL Quantee { ',' Quantee } ':' }
  Head [ DEF Expression ] '.' ;
Head ← NAME_S '(' [ Expression { ',' Expression } ] ')'
  [ '=' SumMinus ];

Assertion ← Axiom;
Axiom ← Expression '.' ;
Expression ← [ { ANNOT } QUANTOR Quantee { ',' Quantee } ':' ] RImplication;
Quantee ← VARIABLE { ',' VARIABLE } IN Type;
Quantee ← VARIABLE { ',' VARIABLE } [ IN Symbol ];
Quantee ← VarTuple { ',' VarTuple } IN Symbol;
VarTuple ← '(' VARIABLE { ',' VARIABLE } ')';
Quantee ← VARIABLE { ',' VARIABLE } IN '{' Value { ',' Value } '}' ;

```

(continues on next page)



(continued from previous page)

```

Quantee ← VarTuple {',' VarTuple} IN
    {' TupleValue {',' TupleValue } '}; *FO[Sugar]
Quantee ← VARIABLE {',' VARIABLE} IN {' POINT '..' POINT '}; *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 {' POINT '..' POINT '}; *FO[Int]

Base ← CARD {' Quantee {',' Quantee} ':' Expression '}; *FO[Agg]
Base ← AGG '(' 'lambda' 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]
Symbol ← 'dom' '(' Expression ')'; *FO[PF]

Identifier ← 'true' | 'false' | NAME_I;
Identifier ← INTEGER; *FO[Int]
Identifier ← POINT; *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 '.';
Interpretation ← NAME_S ':=' Value '.';
Interpretation ← NAME_S ':=' SymbolInterpretation '.';
Interpretation ← NAME_S INCLUDE SymbolInterpretation '.'; *FO[Sugar]
Interpretation ← 'dom' '(' NAME_S ')' ':=' SymbolInterpretation '}' '.'; *FO[PF]
Value ← Identifier;
Value ← CONSTRUCTOR '(' Value {',' Value } ')';
SymbolInterpretation ← {' TupleValue {',' TupleValue } '};
SymbolInterpretation ← {' TupleValue TO Value {',' TupleValue TO Value } '}'
↪[else];
TupleValue ← Value;
TupleValue ← '(' Value {',' Value } ')';

```

(continues on next page)

(continued from previous page)

```
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](https://arxiv.org/abs/2202.00898).
- [CVVD22] Pierre Carbonnelle, Simon Vandeveldel, Joost Vennekens, and Marc Denecker. Idp-z3: a reasoning engine for fo(.). 2022. [arXiv:2202.00343](https://arxiv.org/abs/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, Luís 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](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](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](https://arxiv.org/abs/1401.3840).





## Symbols

"in" operator, 20

## B

Binary quantification, 19

## D

Direct quantification, 20

## F

FO[Agg], 29

FO[Concept], 33

FO[Core], 7

FO[ID], 21

FO[Infinite], 31

FO[Int], 24

FO[PF], 35

FO[Real], 27

FO[Sugar], 17

FO[Unit], 39

FO[·], 3

## K

Knowledge Base, 9

## P

Partial interpretation, 20

## R

Range, 26

## S

Structure block, 10

## T

Theory block, 10

Type declaration, 11

Type inference, 19

## V

Variable declaration, 19

Vocabulary block, 10