IDP-Z3

Pierre Carbonnelle

May 27, 2022

CONTENTS:

CHAPTER

ONE

INTRODUCTION

IDP-Z3 is a software collection implementing the Knowledge Base paradigm using the FO() language. FO() (aka FO-dot) is First Order logic, extended with definitions, types, arithmetic, aggregates and intensional objects. In the Knowledge Base paradigm, the knowledge about a particular problem domain is encoded using a declarative language, and later used to solve particular problems by applying the appropriate type of reasoning, or "inference". The inferences include:

- model checking: does a particular solution satisfy the laws in the knowledge base ?
- model search: extend a partial solution into a full solution
- model propagation: find the facts that are common to all solutions that extend a partial one

The *IDP-Z3 engine* enables the creation of these solutions:

- the Interactive Consultant, which allow a knowledge expert to enter knowledge about a particular problem domain, and an end user to interactively find solutions for particular problem instances;
- *a program* with a command line interface to compute inferences on a knowledge base;
- a web-based Interactive Development Environment (IDE) to create Knowledge bases.

Warning: You may want to verify that you are seeing the documentation relevant for the version of IDP-Z3 you are using. On readthedocs, you can see the version under the title (top left corner), and you can change it using the listbox at the bottom left corner.

1.1 Installation using poetry

Poetry is a package manager for python.

- Install python3 on your machine
- Install poetry
  - after that, logout and login if requested, to update `$PATH`
- Use git to clone https://gitlab.com/krr/IDP-Z3 to a directory on your machine
- Open a terminal in that directory
- If you have several versions of python3, and want to run on a particular one, e.g., 3.9:
  - run `poetry env use 3.9`
  - replace `python3` by `python3.9` in the commands below
- **Run `poetry install`**

  - Run `poetry install --no-dev` if you do not plan to contribute to IDP-Z3 development

To launch the Interactive Consultant web server:

- open a terminal in that directory and run `poetry run python3 main.py`

After that, you can open

- the Interactive Consultant at http://127.0.0.1:5000
- the webIDE at http://127.0.0.1:5000/IDE

1.2 Installation using pip

IDP-Z3 can be installed using the python package ecosystem.

- install python 3, with pip3, making sure that python3 is in the PATH.
- use git to clone https://gitlab.com/krr/IDP-Z3 to a directory on your machine
- (For Linux and MacOS) open a terminal in that directory and run the following commands.

```
python3 -m venv .
source bin/activate
python3 -m pip install -r requirements.txt
```

- (For Windows) open a terminal in that directory and run the following commands.

```
python3 -m venv .
.\Scripts\activate
python3 -m pip install -r requirements.txt
```

To launch the web server on Linux/MacOS, run

```
source bin/activate
python3 main.py
```

On Windows, the commands are:

```
.\Scripts\activate
python3 main.py
```

After that, you can open

- the Interactive Consultant at http://127.0.0.1:5000
- the webIDE at http://127.0.0.1:5000/IDE

1.3 Installation of idp_engine module

The idp_engine module is available for installation through the official Python package repository. This comes with a command line program, `idp_engine` that functions as described in *Command Line Interface*.

To install the module via poetry, the following commands can be used to add the module, and then install it.

```
poetry add idp_engine
poetry install
```

Installing the module via pip can be done as such:

```
pip3 install idp_engine
```

CHAPTER

TWO

THE FO() LANGUAGE

2.1 Overview

The FO() (aka FO-dot) language is used to create knowledge bases. An FO-dot knowledge base is a text file containing the following blocks of code:

vocabulary specify the types, predicates, functions and constants used to describe the problem domain.

theory specify the definitions and axioms satisfied by any solutions.

structure (optional) specify the interpretation of some predicates, functions and constants.

The basic skeleton of an FO-dot knowledge base is as follows:

```
vocabulary {
    // here comes the specification of the vocabulary
}

theory {
    // here comes the definitions and axioms
}

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

Everything between `//` and the end of the line is a comment.

2.2 Shebang

New in version 0.5.5

The first line of an IDP-Z3 program may be a shebang line, specifying the version of IDP-Z3 to be used. When a version is specified, the Interactive Consultant and webIDE will be redirected to a server on the web running that version. The list of versions is available here. (The IDP-Z3 executable ignores the shebang.)

Example: `#! IDP-Z3 0.5.4`

2.3 Vocabulary

```
vocabulary V {
    // here comes the vocabulary named V
}
```

The *vocabulary* block specifies the types, predicates, functions and constants used to describe the problem domain. If the name is omitted, the vocabulary is named V.

Each declaration goes on a new line (or are space separated). Symbols begins with a word character excluding digits, followed by word characters. Word characters include alphabetic characters, digits, _, and unicode characters that can occur in words. Symbols can also be string literals delimited by `'`, e.g., `'blue planet'`.

2.3.1 Types

IDP-Z3 supports built-in and custom types.

The built-in types are: , , , `Date`, and `Concept[signature]` (where `signature` is any type signature, e.g. `()->Bool`). The equivalent ASCII symbols are `Bool`, `Int`, and `Real`.

The type `Concept[signature]` has one constructor for each symbol (i.e., function, predicate or constant) declared in the vocabulary with that signature. The constructors are the names of the symbol, prefixed with `` ` ``.

Custom types are declared using the keyword `type`, e.g., `type color`. Their name should be singular and capitalized, by convention.

Their extension can be defined in a *structure*, or directly in the vocabulary, by specifying:

- a list of (ranges of) numeric literals, e.g., `type someNumbers := {0,1,2}` or `type byte := {0..255}`
- a list of (ranges of) dates, e.g., `type dates := {#2021-01-01, #2022-01-01}` or `type dates := {#2021-01-01 .. #2022-01-01}`
- a list of nullary constructors, e.g., `type Color := {Red, Blue, Green}`
- a list of n-ary constructors; in that case, the enumeration must be preceded by `constructed from`, e.g., `type Color2 := constructed from {Red, Blue, Green, RGB(R: Byte, G: Byte, B: Byte)}`

In the above example, the constructors of `` `Color `` are : `Red`, `Blue`, `Green`.

The constructors of `` `Color2 `` are : `Red`, `Blue`, `Green` and `RGB`. Each constructor have an associated function (e.g., `is_Red`, or `is_RGB`) to test if a Color2 term was created with that constructor. The `RGB` constructor takes 3 arguments of type `Byte`. `R`, `G` and `B` are accessor functions: when given a Color2 term constructed with RGB, they return the associated Byte. (When given a Color2 not constructed with RGB, they may raise an error)

2.3.2 Functions

The functions with name `myFunc1`, `myFunc2`, input domains `p1`, `p2`, `p3` and output range `p`, are declared by:

```
myFunc1, myFunc2 : p1  p2  p3 → p
```

Their name should not start with a capital letter, by convention. The ASCII equivalent of is *, and of → is ->.

The domains and ranges must be one of the following:

- a previously-declared type
- a previously-declared unary predicate

- `Concept[<signature>]` to denote the set of concepts with a particular signature, e.g. `Concept[Person -> Bool]`.

The functions must be total over their domain.

The type of each argument can be directly read in the signature when it is a type or set of concepts, or obtained by looking up the type of the argument of the unary predicate otherwise.

2.3.3 Predicates

A predicate is a function whose range is `Bool`.

There is a built-in predicate `T: T →` for each type T (`T(x)` is `true` for any `x` in `T`).

A unary predicate is always interpreted as a subset of a type.

2.3.4 Built-in functions

The following functions are built-in:

- `abs: Int → Int` (or `abs: Float → Float`) yields the absolute value of an integer (or float) expression;

2.3.5 Propositions and Constants

A proposition is a predicate of arity 0; a constant is a function of arity 0.

```
MyProposition : () →
MyConstant: () → Int
```

2.3.6 Include another vocabulary

A vocabulary W may include a previously defined vocabulary V:

```
vocabulary W {
    import V
    // here comes the vocabulary named W
}
```

2.3.7 Symbol annotations

To improve the display of functions and predicates in the *Interactive Consultant*, their declaration in the vocabulary can be annotated with their intended meaning, a short comment, or a long comment. These annotations are enclosed in `[` and `]`, and come before the symbol declaration.

Intended meaning `[this is a text]` specifies the intended meaning of the symbol. This text is shown in the header of the symbol's box.

Short info `[short:this is a short comment]` specifies the short comment of the symbol. This comment is shown when the mouse is over the info icon in the header of the symbol's box.

Long info `[long:this is a long comment]` specifies the long comment of the symbol. This comment is shown when the user clicks the info icon in the header of the symbol's box.

2.4 Theory

```
theory T:V {
    // here comes the theory named T, on vocabulary named V
}
```

A *theory* is a set of axioms and definitions to be satisfied, and of symbol interpretations. If the names are omitted, the theory is named T, for vocabulary V.

Symbol interpretations are described in the Section on *Structure*. Before explaining the syntax of axioms and definitions, we need to introduce the concept of term.

2.4.1 Mathematical expressions and Terms

A *term* is inductively defined as follows:

Boolean, Numeric and Date literals

> `true` and `false` are boolean terms.
>
> Numeric literals that follow the Python conventions are numerical terms of type `Int` or `Real`.
>
> Date literals are terms. They follow ISO 8601 conventions, prefixed with `#` (i.e., `#yyyy-mm-dd`). `#TODAY` is also a Date literal representing today's date. `#TODAY(y, m, d)` is a Date literal representing today shifted by `y` years, `m` months and `d` days, where `y`, `m` and `d` are integer literals (e.g., `#TODAY(-18, 0, 0)` is today 18 years ago).

Constructor Each constructor of a *type* is a term having that type.

Variable A variable is a term. Its *type* is derived from the *quantifier expression* that declares it (see below).

Function application `F(t_1, t_2,.., t_n)` is a term, when `F` is a *function* symbol of arity `n`, and `t_1, t_2,.., t_n` are terms. Each term must be of the appropriate *type*, and in the domain of the function, as defined in the function declaration in the vocabulary. The resulting type and range of the function application is also defined in the function declaration. If the arity of `F` is 0, i.e., if `F` is a *constant*, then `F()` is a term.

> **Warning:** The knowledge engineer must use appropriate if-then-else guards to ensure that the value of a function outside of its domain has no influence on the truth value of a statement.
>
> For example: `if y = 0 then 0 else x/y`.

> `$(s)(t_1, t_2,.., t_n)` is a term, when `s` is an expression of type Concept that denotes a function of arity `n`, and `t_1, t_2,.., t_n` are terms.
>
> Please note that there are built-in *function*s (see *Built-in functions*).

Negation `-t` is a numerical term, when `t` is a numerical term.

Arithmetic `t_1  t_2` is a numerical term, when `t_1`, `t_2` are two numerical terms, and is one of the following math operators `+, -, * (or ), /, ^, %`. Mathematical operators can be chained as customary (e.g. `x+y+z`). The usual order of binding is used.

Parenthesis `(t)` is a term, when `t` is a term

Cardinality aggregate `#{v_1 in typeOfV_1, .., v_n in typeOfV_n :  }` is a numerical term when `v_1 v_2 .. v_n` are variables, and is a *sentence* containing these variables.

> The term denotes the number of tuples of distinct values for `v_1 v_2 .. v_n` which make true.

Aggregate over anonymous function `agg(lambda v_1 in typeOfV_1, .., v_n in typeOfV_n : t)` is a numerical term where `agg` can be any of (`sum`, `min`, `max`), `v_1 v_2 .. v_n` are variables and `t` is a term.

The term `sum(lambda v in T : t(v))` denotes the sum of `t(v)` for each `v` in `T`. Similarly, `min` (resp. `max`) can be used to compute the minimum (resp. maximum) of `t(v)` for each `v` in `T`. `t(v)` can use the construct `(if .. then .. else ..)` to filter out unwanted `v` values.

(if .. then .. else ..) `(if t_1 then t_2 else t_3)` is a term when `t_1` is a sentence, `t_2` and `t_3` are terms of the same type.

2.4.2 Sentences and axioms

An *axiom* is a sentence followed by `.`. A *sentence* is inductively defined as follows:

true and false `true` and `false` are sentences.

Predicate application `P(t_1, t_2,.., t_n)` is a sentence, when `P` is a *predicate* symbol of arity `n`, and `t_1, t_2,.., t_n` are terms. Each term must be of the appropriate *type*, as defined in the predicate declaration. If the arity of `P` is 0, i.e., if `P` is a proposition, then `P()` is a sentence.

`$(s)(t_1, t_2,.., t_n)` is a sentence, when `s` is an expression of type Concept that denotes a predicate of arity `n`, and `t_1, t_2,.., t_n` are terms.

Comparison `t_1 t_2` is a sentence, when `t_1`, `t_2` are two numerical terms and is one of the following comparison operators `<, , =, , >,` (or, using ascii characters: `=<, >=, ~=`). Comparison operators can be chained as customary.

Negation `¬` is a sentence (or, using ascii characters: `~`) when is a sentence.

Logic connectives `_1 _2` is a sentence when `_1, _2` are two sentences and is one of the following logic connectives `,,,,` (or using ascii characters: `|, \&, =>, <=, <=>` respectively). Logic connectives can be chained as customary.

Parenthesis () is a sentence when is a sentence.

Enumeration An enumeration (e.g. `p := {1;2;3}`) is a sentence. Enumerations follow the syntax described in *structure*.

Quantified formulas *Quantified formulas* are sentences. They have one of the following forms, where `v_1, .., v_n` are variables, `p, p_1, .., p_n` are types or predicates, and is a sentence involving those variables:

```
v_1, v_n: (v_1, v_n).

v_1, v_n  p: (v_1, v_n).

(v_1, v_n)  p: (v_1, v_n).

v_1  p_1, v_n  p_n: (v_1, v_n).
```

Alternatively, the existential quantifier, , can be used. Ascii characters can also be used: `?`, `!`, respectively. For example, `! x, y in Int:  f(x,y)=f(y,x).`

A variable may only occur in the sentence of a quantifier declaring that variable. In the first form above, the type of each variable is inferred from their use in .

"is enumerated" `f(a,b) is enumerated` is a sentence, where `f` is a function defined by an enumeration and applied to arguments `a` and `b`. Its truth value reflects whether `(a,b)` is enumerated in `f`'s enumeration. If the enumeration has a default value, every tuple of arguments is enumerated.

"in {1,2,3,4}" `f(args) in enum` is a sentence, where `f` is a function applied to arguments `args` and `enum` is an enumeration. This can also be written using Unicode: `f() ∈ {1,2,3}`.

if .. then .. else .. `if t_1 then t_2 else t_3` is a sentence when `t_1`, `t_2` and `t_3` are sentences.

2.4.3 Definitions

A *definition* defines concepts, i.e. *predicate*s or *function*s, in terms of other concepts. If a predicate is inductively defined in terms of itself, the definition employs the *well-founded* semantics. A definition consists of a set of rules, enclosed by `{` and `}`.

Rules have one of the following forms:

```
v_1 ∈ T_1, v_n ∈ T_n: P(t_1, .., t_n) ← |phi|.

v_1 ∈ T_1, v_n ∈ T_n: F(t_1, .., t_n) = t ← |phi|.
```

where P is a *predicate* symbol, F is a *function* symbol, `t`, `t_1, t_2,.., t_n` are terms that may contain the variables `v_1 v_2 .. v_n` and is a formula that may contain these variables. `P(t_1, t_2,.., t_n)` is called the *head* of the rule and the *body*. `<-` can be used instead of `←`. If the body is `true`, the left arrow and body of the rule can be omitted.

2.4.4 Annotations

Some expressions can be annotated with their informal meaning, between brackets. For example, `[age is a positive number] 0 =< age()`. Such annotations are used in the Interactive Consultant.

The following expressions can be annotated:

- Definitions
- Rules
- Constraints
- Quantified formula
- Comparisons
- Membership in an enumeration
- Brackets

When necessary, use parenthesis to avoid ambiguity, e.g. `[Positive or p] ( [Positive] x()<0 ) | p()`..

2.5 Structure

```
structure S:V {
    // here comes the structure named S, for vocabulary named V
}
```

A *structure* specifies the interpretation of some *type*, *predicate*s and *function*s. If the names are omitted, the structure is named S, for vocabulary V.

A structure is a set of symbol interpretations of the form :

- `<symbol> := <total interpretation>.`, e.g., `P := {1..9}`,

- `<symbol> : <partial interpretation>.`, e.g., `P : {A->1}`.

A total interpretation fully specifies the interpretation of the symbol; a partial interpretation does not.

An error occurs when the domain of the symbol is already known by enumeration, and its total interpretation does not cover the domain. If the domain of the symbol is not known, it is inferred from the total interpretation.

A total interpretation takes one of these forms:

for nullary predicates (propositions) `true` or `false`

for non-numeric types and unary predicates: a set of rigid terms (numbers, dates, identifiers, or constructors applied to rigid terms), e.g., `{red, blue, green}`.

for numeric types and unary predicates: a set of numeric literals and ranges, e.g., `{0,1,2}`, `{0..255}` or `{0..9, 90..99}`

for date types and unary predicates: a set of date literals and ranges, e.g., `{#2021-01-01, #2022-01-01}` or `{#2021-01-01 .. #2022-01-01}`

for types: a set of n-ary constructors, preceded by `constructed from`, e.g., `constructed from {Red, Blue, Green, RGB(R: Byte, G: Byte, B: Byte)}` (see more details in *type*s)

for n-ary predicates: a set of tuples of rigid terms, e.g., `{(a,b), (a,c)}`.

for nullary functions: a rigid term, e.g. `5` or `#2021-01-01`, or `red` or `rgb(0,0,0)`

for n-ary functions: a set of tuples and their associated values, e.g., `{ (1,2)->3, (4, 5)->6 }`. The interpretation may be followed by `else <default>`, where `<default>` is a default value (a rigid term), i.e., a value for the non-enumerated tuples, if any.

A partial interpretation takes one of these forms:

for n-ary functions: a set of tuples and their associated values, e.g., `{ (1,2)->3, (4, 5)->6 }`

Additional notes:

- the set of tuples in the interpretation of a predicate is exactly the set of tuples that make the predicate true; any other tuple makes it false.
- parenthesis around a tuple can be omitted when the arity is 1, e.g., `{1-2, 3->4}`
- the interpretation of a predicate may be specified using the CSV format, with one tuple per line, e.g., :

```
P := {
1 2
3 4
5 6
}
```

- The interpretation of `goal_string` is used to compute relevance relative to goals (see the `determine_relevance` method in the *Theory class*).
- The tuples of an interpretation can be given in any order.

2.6 Differences with IDP3

Here are the main differences with IDP3 (the previous version of IDP-Z3), listed for migration purposes:

Infinite domains IDP-Z3 supports infinite domains: `Int, Real`. However, quantifications over infinite domains is discouraged.

Type IDP-Z3 supports type hierarchies differently from IDP3: subtypes are now represented by unary predicates; unary predicates can be used wherever types can be used, i.e., in type signatures and in quantifiers.

LTC IDP-Z3 does not support LTC vocabularies.

Namespaces IDP-Z3 does not support namespaces.

Partial functions In IDP-Z3, a function must be total over a cross-product of (sub-)types. The handling of division by 0 may differ. See IEP 07

Syntax changes The syntax of quantifications and aggregates has slightly change. IDP-Z3 does not support qualified quantifications, e.g. `!2 x[color]: p(x).`. (p. 11 of the IDP3 manual). Such statements can be implemented in IDP-Z3 using cardinality constraints instead.

if .. then .. else .. IDP-Z3 supports *if .. then .. else ..* terms and sentences.

Structure IDP-Z3 supports partial interpretations of functions using the `:>=` sign. (It currently does not support partial interpretations of predicates)

To improve performance, do not quantify over the value of a function. Use `p(f(x))` instead of `?y: f(x)=y & p(y)`.

CHAPTER

THREE

IDP-Z3

IDP-Z3 is used to perform reasoning on FO() knowledge bases. It can be invoked in 3 ways:

- via a web interface, called webIDE.
- in a shell, using the Command Line Interface of IDP-Z3.
- in a Python program: by using classes and functions imported from the `idp_engine` package available on Pypi.

These methods are further described below.

Warning: An *FO-dot* program is a text file containing only *vocabulary, theory* and, *structure* blocks, as described in *FO-dot*. An *IDP* program may additionally contain a *main()* procedure block, with instructions to process the FO-dot program. This procedure block is described later in this chapter.

3.1 webIDE

The webIDE of IDP-Z3 is accessible online, and can be *run locally*.

The webIDE allows you to enter an IDP-Z3 program, with *FO-dot vocabulary, theory, structure* blocks and a *main block*, and to run it.

3.1.1 Main block

The *main block* consists of python-like statements to be executed by the *IDP-Z3 executable* or the *webIDE*, in the context of the knowledge base. Below is an example of a main block.

```
procedure main() {
    pretty_print(Theory(T, S).propagate())
    duration("End")
}
```

Within that block, the following variables, classes and functions are available:

- variables containing the vocabularies, theories and structures specified in the same IDP-Z3 program. The variables have the name of the block.
- the functions exposed by the `idp_engine`, described *here*;
- the `Theory` class, described *here*.

3.1.2 idp_engine functions

The following Python functions can be used to perform computations using FO-dot knowledge bases:

model_check(**theories*)
Returns a string stating whether the combination of theories is satisfiable.

For example, `print(model_check(T, S))` will print `sat` if theory named `T` has a model expanding structure named `S`.

Parameters **theories** (*Union[TheoryBlock, Structure, Theory]*) – 1 or more (data) theories.

Returns `sat`, `unsat` or `unknown`

Return type str

model_expand(**theories*, *max=10*, *timeout_seconds=10*, *complete=False*, *extended=False*, *sort=False*)
Returns a (possibly empty) list of models of the combination of theories, followed by a string message.

For example, `print(model_expand(T, S))` will return (up to) 10 string representations of models of theory named `T` expanding structure named `S`.

The string message can be one of the following:

- `No models.`
- `More models may be available. Change the max argument to see them.`
- `More models may be available. Change the timeout_seconds argument to see them.`
- `More models may be available. Change the max and timeout_seconds arguments to see them.`

Parameters

- **theories** (*Union[TheoryBlock, Structure, Theory]*) – 1 or more (data) theories.
- **max** (*int, optional*) – max number of models. Defaults to 10.
- **timeout_seconds** (*int, optional*) – timeout_seconds in seconds. Defaults to 10.
- **complete** (*bool, optional*) – True to obtain complete structures. Defaults to False.
- **extended** (*bool, optional*) – use *True* when the truth value of inequalities and quantified formula is of interest (e.g. for the Interactive Consultant). Defaults to False.
- **sort** (*bool, optional*) – True if the models should be in alphabetical order. Defaults to False.

Yields str

Return type Iterator[str]

model_propagate(**theories*, *sort=False*)
Returns a list of assignments that are true in any model of the combination of theories.

Terms and symbols starting with '_' are ignored.

For example, `print(model_propagate(T, S))` will return the assignments that are true in any expansion of the structure named `S` consistent with the theory named `T`.

Parameters

- **theories** (*Union[TheoryBlock, Structure, Theory]*) – 1 or more (data) theories.

- **sort** (*bool, optional*) – True if the assignments should be in alphabetical order. Defaults to False.

Yields str

Return type Iterator[str]

maximize(**theories*, *term*)
Returns the list of assignments that are true in any model that maximizes *term*.

Parameters

- **theories** (*Union[TheoryBlock, Structure, Theory]*) – 1 or more (data) theories.
- **term** (*str*) – a string representing a term

Yields str

Return type Iterator[str]

minimize(**theories*, *term*)
Returns the list of assignments that are true in any model that minimizes *term*.

Parameters

- **theories** (*Union[TheoryBlock, Structure, Theory]*) – 1 or more (data) theories.
- **term** (*str*) – a string representing a term

Yields str

Return type Iterator[str]

decision_table(**theories*, *goal_string=''*, *timeout_seconds=20*, *max_rows=50*, *first_hit=True*, *verify=False*)
Experimental. Returns a decision table for *goal_string*, given the combination of theories.

Parameters

- **theories** (*Union[TheoryBlock, Structure, Theory]*) – 1 or more (data) theories.
- **goal_string** (*str, optional*) – the last column of the table. Must be a predicate application defined in the theory, e.g. `eligible()`.
- **timeout_seconds** (*int, optional*) – maximum duration in seconds. Defaults to 20.
- **max_rows** (*int, optional*) – maximum number of rows. Defaults to 50.
- **first_hit** (*bool, optional*) – requested hit-policy. Defaults to True.
- **verify** (*bool, optional*) – request verification of table completeness. Defaults to False

Yields a textual representation of each rule

Return type Iterator[str]

determine_relevance(**theories*)
Generates a list of questions that are relevant, or that can appear in a justification of a `goal_symbol`.

The questions are preceded with `` ? `` when their answer is unknown.

When an *irrelevant* value is changed in a model M of the theories, the resulting M' structure is still a model. Relevant questions are those that are not irrelevant.

If `goal_symbol` has an enumeration in the theory (e.g., `goal_symbol := {`tax_amount}.`), relevance is computed relative to those goals.

Definitions in the theory are ignored, unless they influence axioms in the theory or goals in `goal_symbol`.

Yields relevant questions

Parameters **theories** (*Union[idp_engine.Parse.TheoryBlock, idp_engine.Parse.Structure, idp_engine.Theory.Theory]*) –

Return type Iterator[str]

pretty_print(*x=''*)
Prints its argument on stdout, in a readable form.

Parameters **x** (*Any, optional*) – the result of an API call. Defaults to "".

Return type None

duration(*msg=''*)
Returns the processing time since the last call to *duration()*, or since the begining of execution

Parameters **msg** (*str*) –

Return type str

3.1.3 Theory class

Instances of the `Theory` class represent a collection of theory and structure blocks.

Many operations on Theory instances can be chained, e.g., `Theory(T,S).propagate().simplify().formula()`.

The class has the following methods:

class Theory(**theories*, *extended=False*)
A collection of theory and structure blocks.

assignments (Assignments): the set of assignments. The assignments are updated by the different steps of the problem resolution. Assignments include inequalities and quantified formula when the problem is extended

Parameters

- **theories** (*Union[idp_engine.Parse.TheoryBlock, idp_engine.Parse.Structure, Theory]*) –
- **extended** (*bool*) –

Return type None

__init__(**theories*, *extended=False*)
Creates an instance of `Theory` for the list of theories, e.g., `Theory(T,S)`.

Parameters

- **theories** (*Union[TheoryBlock, Structure, Theory]*) – 1 or more (data) theories.
- **extended** (*bool, optional*) – use *True* when the truth value of inequalities and quantified formula is of interest (e.g. for the Interactive Consultant). Defaults to False.

Return type None

add(**theories*)
Adds a list of theories to the theory.

Parameters **theories** (*Union[TheoryBlock, Structure, Theory]*) – 1 or more (data) theories.

Return type *idp_engine.Theory.Theory*

assert_(*code*, *value*, *status=Status.GIVEN*)
asserts that an expression has a value (or not), e.g. `theory.assert_("p()", True)`

Parameters

- **code** (*str*) – the code of the expression, e.g., "p()"
- **value** (*Any*) – a Python value, e.g., True
- **status** (*Status, Optional*) – how the value was obtained. Default: S.GIVEN

Return type *idp_engine.Theory.Theory*

copy()
Returns an independent copy of a theory.

Return type *idp_engine.Theory.Theory*

decision_table(*goal_string=''*, *timeout_seconds=20*, *max_rows=50*, *first_hit=True*, *verify=False*)
Experimental. Returns the rows for a decision table that defines goal_string.

goal_string must be a predicate application defined in the theory. The theory must be created with extended=True.

Parameters

- **goal_string** (*str, optional*) – the last column of the table.
- **timeout_seconds** (*int, optional*) – maximum duration in seconds. Defaults to 20.
- **max_rows** (*int, optional*) – maximum number of rows. Defaults to 50.
- **first_hit** (*bool, optional*) – requested hit-policy. Defaults to True.
- **verify** (*bool, optional*) – request verification of table completeness. Defaults to False

Returns the non-empty cells of the decision table for goal_string, given self. bool: whether or not the timeout limit was reached.

Return type list(list(Assignment))

determine_relevance()
Determines the questions that are relevant in a model, or that can appear in a justification of a goal_symbol.

When an *irrelevant* value is changed in a model M of the theory, the resulting M' structure is still a model. Relevant questions are those that are not irrelevant.

Call must be made after a propagation, on a Theory created with extended=True. The result is found in the relevant attribute of the assignments in self.assignments.

If goal_symbol has an enumeration in the theory (e.g., goal_symbol := {`tax_amount}.), relevance is computed relative to those goals.

Definitions in the theory are ignored, unless they influence axioms in the theory or goals in goal_symbol.

Returns the Theory with relevant information in self.assignments.

Return type *Theory*

Parameters **self** (idp_engine.Theory.Theory) –

disable_law(*code*)
Disables a law, represented as a code string taken from the output of explain(...).

The law should not result from a structure (e.g., from p:=true.) or from a types (e.g., from T:={1..10} and c: () -> T).

Parameters **code** (*str*) – the code of the law to be disabled

Raises **AssertionError** – if code is unknown

Return type *idp_engine.Theory.Theory*

enable_law(*code*)

Enables a law, represented as a code string taken from the output of explain(...).

The law should not result from a structure (e.g., from `p:=true.`) or from a types (e.g., from `T:={1..10}` and `c: () -> T`).

Parameters **code** (*str*) – the code of the law to be enabled

Raises **AssertionError** – if code is unknown

Return type *idp_engine.Theory.Theory*

expand(*max=10*, *timeout_seconds=10*, *complete=False*)

Generates a list of models of the theory that are expansion of the known assignments.

The result is limited to `max` models (10 by default), or unlimited if `max` is 0. The search for new models is stopped when processing exceeds `timeout_seconds` (in seconds) (unless it is 0). The models can be asked to be complete or partial (i.e., in which "don't care" terms are not specified).

The string message can be one of the following:

- `No models.`
- `More models may be available. Change the max argument to see them.`
- `More models may be available. Change the timeout_seconds argument to see them.`
- `More models may be available. Change the max and timeout_seconds arguments to see them.`

Parameters

- **max** (*int, optional*) – maximum number of models. Defaults to 10.
- **timeout_seconds** (*int, optional*) – timeout_seconds in seconds. Defaults to 10.
- **complete** (*bool, optional*) – `True` for complete models. Defaults to False.

Yields the models, followed by a string message

Return type Iterator[Union[idp_engine.Assignments.Assignments, str]]

explain(*consequence=None*)

Returns the facts and laws that make the Theory unsatisfiable, or that explains a consequence.

Parameters

- **self** (`Theory`) – the problem state
- **consequence** (*string, optional*) – the code of the consequence to be explained. Must be a key in `self.assignments`

Returns list of facts and laws that explain the consequence

Return type (List[Assignment], List[Expression])]

formula()

Returns a Z3 object representing the logic formula equivalent to the theory.

This object can be converted to a string using `str()`.

Return type z3.z3.BoolRef

`get_range`(*term*)
Returns a list of the possible values of the term.

Parameters `term` (*str*) – terms whose possible values are requested, e.g. `subtype()`. Must be a key in `self.assignments`

Returns e.g., `['right triangle', 'regular triangle']`

Return type List[str]

`optimize`(*term*, *minimize=True*)
Updates the Theory so that the value of term in the `assignments` property is the optimal value that is compatible with the Theory.

Chain it with a call to *expand* to obtain a model, or to *propagate* to propagate the optimal value.

Parameters

- `term` (*str*) – e.g., `"Length(1)"`
- `minimize` (*bool*) – `True` to minimize `term`, `False` to maximize it

Return type *idp_engine.Theory.Theory*

`propagate`(*tag=Status.CONSEQUENCE*, *method=Propagation.DEFAULT*)

Returns the theory with its `assignments` property updated with values for all terms and atoms that have the same value in every model of the theory.

`self.satisfied` is also updated to reflect satisfiability.

Terms and propositions starting with _ are ignored.

Args: tag (S): the status of propagated assignments method (Propagation): the particular propagation to use

Parameters

- `tag` (*idp_engine.Assignments.Status*) –
- `method` (*idp_engine.Theory.Propagation*) –

Return type *idp_engine.Theory.Theory*

`simplify`()
Returns a simpler copy of the theory, with a simplified formula obtained by substituting terms and atoms by their known values.

Assignments obtained by propagation become UNIVERSAL constraints.

Return type *idp_engine.Theory.Theory*

`symbolic_propagate`(*tag=Status.UNIVERSAL*)
Returns the theory with its `assignments` property updated with direct consequences of the constraints of the theory.

This propagation is less complete than `propagate()`.

Parameters `tag` (*S*) – the status of propagated assignments

Return type *idp_engine.Theory.Theory*

`to_smt_lib`()
Returns an SMT-LIB version of the theory

Return type str

3.2 Command Line Interface

IDP-Z3 can be run through a Command Line Interface.

If you have downloaded IDP-Z3 from the GitLab repo, you may run the CLI using poetry (see *Installation*):

```
poetry run python3 idp-engine.py path/to/file.idp
```

where *path/to/file.idp* is the path to the file containing the IDP-Z3 program to be run. This file must contain an *FO-dot* knowledge base (vocabulary, theory and structure blocks), and a *main block*.

Alternatively, if you installed it via pip, you can run it with the following command:

```
idp-engine path/to/file.idp
```

The usage of the CLI is as follows:

```
usage: idp-engine.py [-h] [--version] [-o OUTPUT] [--full-formula] [--no-timing] FILE

IDP-Z3

positional arguments:
  FILE                  path to the .idp file

options:
  -h, --help            show this help message and exit
  --version, -v         show program's version number and exit
  -o OUTPUT, --output OUTPUT
                        name of the output file
  --full-formula        show the full formula
  --no-timing           don't display timing information
```

3.3 Python API

The core of the IDP-Z3 software is a Python component available on Pypi. The following code illustrates how to invoke it.

```
from idp_engine import IDP, Theory, duration
kb = IDP.from_file("path/to/file.idp")
T, S = kb.get_blocks("T, S")
theory = Theory(T, S)
for model in theory.expand():
    print(model)
duration("End")
```

The file `path/to/file.idp` must contain an *FO-dot* knowledge base (with vocabulary, theory and, optionally, structure blocks).

`idp_engine` exposes *useful functions*, as well as the `Theory` (described *here*) and `IDP` classes.

3.3.1 IDP class

The `IDP` class exposes the following methods:

class IDP(***kwargs*)

The class of AST nodes representing an IDP-Z3 program.

__init__(***kwargs*)

classmethod from_file(*file*)

parse an IDP program from file

Parameters **file** (*str*) – path to the source file

Returns the result of parsing the IDP program

Return type *IDP*

classmethod from_str(*code*)

parse an IDP program

Parameters **code** (*str*) – source code to be parsed

Returns the result of parsing the IDP program

Return type *IDP*

classmethod parse(*file_or_string*)

DEPRECATED: parse an IDP program

Parameters **file_or_string** (*str*) – path to the source file, or the source code itself

Returns the result of parsing the IDP program

Return type *IDP*

get_blocks(*blocks*)

returns the AST nodes for the blocks whose names are given

Parameters **blocks** (*List[str]*) – list of names of the blocks to retrieve

Returns list of AST nodes

Return type List[Union[Vocabulary, TheoryBlock, Structure, Procedure, Display]]

execute()

Execute the `main()` procedure block in the IDP program

Parameters **self** (idp_engine.Parse.IDP) –

Return type None

CHAPTER

FOUR

INTERACTIVE CONSULTANT

The Interactive Consultant tool enables experts to digitize their knowledge of a specific problem domain. With the resulting knowledge base, an online interface is automatically created that serves as a web tool supporting end users to find solutions for specific problems within that knowledge domain.

The tool uses source code in the IDP-Z3 language as input. It recognizes the *annotations in vocabulary* and *in expressions*. However, there are some specific changes and additions when using IDP-Z3 in the Interactive Consultant, which are explained further in this chapter.

4.1 Display block

The *display block* configures the user interface of the *Interactive Consultant*. It consists of a set of *display facts*, i.e., *predicate* and *function applications* terminated by `..`

The following predicates and functions are available:

expand `expand := {`s1, .., `sn}.` specifies that *symbols* `s1, .., sn` are shown expanded, i.e., that all sub-sentences of the theory where they occur are shown on the screen.

For example, `expand := {`length}.` will force the Interactive Consultant to show all sub-sentences containing *length*.

hide `hide(`s1, .., `sn)` specifies that symbols `s1, .., sn` are not shown on the screen.

For example, `hide(`length).` will force the Interactive Consultant to not display the box containing *length* information.

view() `view() = normal.` (default) specifies that symbols are displayed in normal mode.

`view() = expanded.` specifies that symbols are displayed expanded.

goal_symbol `goal_symbol := {`s1, .., `sn}.` specifies that symbols `s1, .., sn` are always relevant, i.e. that they should never be greyed out. `goal_symbol` can only be used in an enumeration.

Irrelevant symbols and questions, i.e. expressions whose interpretation do not constrain the interpretation of the relevant symbols, are greyed out by the Interactive Consultant.

moveSymbols When the *display block* contains `moveSymbols()`, the Interactive Consultant is allowed to change the layout of symbols on the screen, so that relevant symbols come first.

By default, the symbols do not move.

optionalPropagation When the *display block* contains `optionalPropagation()`, a toggle button is shown next to the menu to allow toggling immediate propagation on and off.

By default, this button is not present.

manualPropagation When `manualPropagation()` is present in the *display block*, automatic propagation is disabled in the interface. Instead, a button is added to the menu that computes propagation when clicked.

optionalRelevance When the *display block* contains `optionalRelevance()`, a toggle button is shown next to the menu to allow toggling immediate computation of relevance on and off.

By default, this button is not present.

manualRelevance When `manualRelevance()` is present in the *display block*, automatic computation of relevant questions is disabled in the interface. Instead, a menu option is available in the "Reasoning" menu that computes relevance when selected.

unit `unit('unitstr', `s1, ..., `sn)` specifies the unit of one or more symbols. This unit will then show up in the symbol's header in the Interactive Consultant. `unitstr` may not be a symbol declared in the vocabulary.

For example: `unit('m', `length, `perimeter).`

heading Experimental: this feature is likely to change in the future.

`heading('label', `p1, ..., `pn)` will force the display of the `p1, ..., pn` symbols under a heading called `label`. `label` may not be a symbol declared in the vocabulary.

noOptimization `noOptimization(`s1, .., `sn)` specifies that no optimization buttons appear in the Interactive Consultant for symbols `s1, .., sn`.

For example, `noOptimization(`angle).` will hide the arrow up and arrow down buttons next to the input fields for *angle*.

4.2 Environment & Decision

Often, some elements of a problem instance are under the control of the user (possibly indirectly), while others are not.

To capture this difference, the FO() language allows the creation of 2 vocabularies and 2 theories. The first one is called 'environment', the second 'decision'. Hence, a more advanced skeleton of an IDP-Z3 program is:

```
vocabulary environment {
    // here comes the specification of the vocabulary to describe the environment
}

vocabulary decision {
    import environment
    // here comes the specification of the vocabulary to describe the decisions and␣
↪their consequences
}

theory environment:environment {
    // here comes the definitions and axioms satisfied by any environment possibly faced␣
↪by the user
}

theory decision:decision {
    // here comes the definitions and axioms to be satisfied by any solution
}

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

(continues on next page)

(continued from previous page)

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

display {
    // here comes the configuration of the user interface
}
```

4.3 Default Structure

The *default structure* functions similarly to a normal *Structure*, in the sense that it can be used to set values of symbols. However, these values are set as if they were given by the user: they are shown in the interface as selected values. The symbols can still be assigned different values, or they can be unset.

In this way, this type of structure is used to form a *default* set of values for symbols. Such a structure is given the name 'default', to denote that it specifies default values. The syntax of the block remains the same.

```
structure default {
    // here comes the structure
}
```

CHAPTER

FIVE

APPENDIX: IDP-Z3 INTERNAL REFERENCE

Warning: This reference is only intended for the **core IDP-Z3 development team**. If you do not work on the IDP-Z3 engine itself, but just want to use it in your applications, please use our *Python API* instead.

The components of IDP-Z3 are shown below.

- webIDE client: browser-based application to edit and run IDP-Z3 programs
- Interactive Consultant client: browser-based user-friendly decision support application
- Read_the_docs : online documentation
- Homepage
- IDP-Z3 server: web server for both web applications
- IDP-Z3 command line interface
- IDP-Z3 engine: performs reasoning on IDP-Z3 theories
- Z3: SMT solver developed by Microsoft

The source code of IDP-Z3 is publicly available under the GNU LGPL v3 license. You may want to check the Development and deployment guide.

5.1 Architecture

This document presents the technical architecture of IDP-Z3.

Essentially, the IDP-Z3 components translate the requested inferences on the knowledge base into satisfiability problems that Z3 can solve.

5.1.1 Web clients

The source code for the web clients is in the IDP_Z3_web_client folder.

The clients are written in Typescript, using the Angular framework (version 7.1), and the primeNG library of widgets. It uses the Monaco editor. The interactions with the server are controlled by idp.service.ts. The AppSettings file contains important settings, such as the address of the IDP-Z3 sample theories.

The web clients are sent to the browser by the IDP-Z3 server as static files. The static files are generated by the `/IDP-Z3/deploy.py` script as part of the deployment, and saved in the `/IDP-Z3/idp_web_server/static` folder.

See the Appendix of Development and deployment guide on the wiki for a discussion on how to set-up your environment to develop web clients.

The `/docs/zettlr/REST.md` file describes the format of the data exchanged between the web client and the server. The exchange of data while using web clients can be visualised in the developer mode of most browsers (Chrome, Mozilla, ...).

The web clients could be packaged into an executable using nativefier.

5.1.2 Read The Docs, Homepage

The online documentation and Homepage are written in ReStructuredText, generated using sphinx and hosted on readthedocs.org and GitLab Pages respectively. The contents is in the `/docs` and `/homepage` folders of IDP-Z3.

We use the following sphinx extensions: Mermaid (diagrams), and Markdown.

5.1.3 IDP-Z3 server

The code for the IDP-Z3 server is in the `/idp_web_server` folder.

The IDP-Z3 server is written in python 3.8, using the Flask framework. Pages are served by `/idp_web_server/rest.py`. Static files are served from the `/idp_web_server/static` directory, including the compiled version of the client software.

At start-up, and every time the idp code is changed on the client, the idp code is sent to the `/meta` URL by the client. The server responds with the list of symbols to be displayed. A subsequent call (`/eval`) returns the questions to be displayed. After that, when the user clicks on a GUI element, information is sent to the `/eval` URL, and the server responds as necessary.

The information given by the user is combined with the idp code (in *State.py*), and, using adequate inferences, the questions are put in these categories with their associated value (if any):

- given: given by the user
- universal: always true (or false), per idp code
- consequences: consequences of user's input according to theory
- irrelevant: made irrelevant by user's input
- unknown

The IDP-Z3 server implements custom inferences such as the computation of relevance (*Inferences.py*), and the handling of environmental vs. decision variables.

API endpoints

The IDP-Z3 server exposes multiple API endpoints, which are used to communicate information between the interface and server.

/run

POST: Runs an IDP program containing a main block. The program is be executed by the IDP-Z3 directly, and the output is returned. This endpoint is e.g. used to execute the code in the IDP webIDE.

Arguments:

- **code**: IDP code, containing a main block.

Returns:

- A string, containing the output of the IDP-Z3 engine after executing the program.

/meta

POST: generate the metaJSON for an IDP program. In the IC, this metaJSON is among others used to correctly lay out the different symbol tiles and to generate extra expanded symbols.

Arguments:

- **code**: IDP code, with or without main block.

Returns:

- **symbols**: contains information on each symbol used in the FO(·) specification. This information includes symbol name, type, view, ...
- **optionalPropagation**: a bool representing if a propagation toggle should be shown in the interface.
- **manualPropagation**: a bool representing if propagation should be manual via a button.
- **optionalRelevance**: a bool representing if a relevance toggle should be shown in the interface.
- **manualRelevance**: a bool representing if relevance computation should be manual via a button.
- **valueinfo**: contains information on the values for each symbol used in the FO(·) specification.

/eval

POST: execute one of IDP-Z3's inference methods.

Arguments:

- **method**: string containing the method to execute. Supported methods are: *checkCode*, *propagate*, *get_range*, *modelexpand*, *explain*, *minimize*, and *abstract*.
- **code**: the IDP code.
- **active**: the active assignments, already input in the interface.
- **previous_active**: the assignments after the last full propagation.
- **ignore**: user-disabled laws to ignore.
- **symbol**: the name of a symbol, only used for *minimize*, *explain* and *checkCode*.
- **value**: a value, only used for *explain*.
- **field**: the applied symbol for which a range must be determined, only for *get_range*.
- **minimize**: *true* for minimization, *false* for maximization.

Returns:

- **Global**: the global information of the current state of the IC.
- A field for every symbol that appears in the IDP program, containing all its information.

5.1.4 IDP-Z3 engine

The code for the IDP-Z3 engine and IDP-Z3-CLI is in the `/idp_engine` folder. The IDP-Z3 engine exposes an API implemented by *Run.py* and *Problem.py*.

Translating knowledge inferences into satisfiability problems that Z3 can solve involves these steps:

1. parsing the idp code and the info entered by the user,
2. converting it to the Z3 format,
3. calling the appropriate method,
4. formatting the response.

The IDP-Z3 code is parsed into an abstract syntax tree (AST) using the textx package, according to this grammar. There is one python class per type of AST nodes (see *Parse.py* and *Expression.py*

The conversion to the Z3 format is performed by the following passes over the AST generated by the parser:

1. annotate the nodes by resolving names, and computing some derived information (e.g. type) (`annotate()`)
2. expand quantifiers in the theory, as far as possible. (`interpret()`)
3. when a structure is given, use the interpretation (`interpret()`), i.e.:

   a) expand quantifiers based on the structure (grounding); perform type inference as necessary;

   b) simplify the theory using the data in the structure and the laws of logic;

   c) instantiate the definitions for every calls of the defined symbols (recursively)
4. convert to Z3, adding the type constraints not enforced by Z3 (`.translate()`)

Substitute() modifies the AST "in place". Because the results of step 1-2 are cached, steps 4-7 are done after copying the AST (custom `copy()`).

The graph of calls is outlined below:

The code is organised by steps, not by classes: for example, all methods to annotate an expression by another are grouped in *Annotate.py*. We use monkey-patching to attach methods to the classes declared in another module.

Important classes of the IDP-Z3 engine are: Expression, Assignment, Theory.

Below is a simplified class diagram for the classes of the Abstract Syntax tree.

And a simplified class diagram for the `Theory` class:

5.1.5 Z3

See this tutorial for an introduction to Z3 (or this guide).

You may also want to refer to the Z3py reference.

5.1.6 Appendix: Dependencies and Licences

The IDP-Z3 tools are published under the GNU LGPL v3 license.

The server software uses the following components (see requirements.txt):

- Z3: MIT license
- Z3-solver: MIT license
- Flask: BSD License (BSD-3-Clause)
- flask_restful : BSD license
- flask_cors : MIT license
- pycallgraph2 : GNU GPLv2
- gunicorn : MIT license
- textx: MIT license

The client-side software uses the following components:

- Angular: MIT-style license
- PrimeNg: MIT license
- ngx-monaco-editor: MIT license
- packery: GPL-3.0
- primeicons: MIT
- isotope-layout: GNU GPL-3.0
- isotope-packery: MIT
- core-js: MIT
- dev: None
- git-describe: MIT
- rxjs: Apache 2.0
- tslib: Apache 2.0
- zone.js: MIT

5.2 `idp_engine` module

5.2.1 idp_engine.Parse

Classes to parse an IDP-Z3 theory.

class `IDP`(***kwargs*)

Bases: `idp_engine.Expression.ASTNode`

The class of AST nodes representing an IDP-Z3 program.

`__init__`(***kwargs*)

classmethod `from_file`(*file*)

parse an IDP program from file

Parameters **file** (*str*) – path to the source file

Returns the result of parsing the IDP program

Return type *IDP*

classmethod from_str(*code*)
parse an IDP program

Parameters **code** (*str*) – source code to be parsed

Returns the result of parsing the IDP program

Return type *IDP*

classmethod parse(*file_or_string*)
DEPRECATED: parse an IDP program

Parameters **file_or_string** (*str*) – path to the source file, or the source code itself

Returns the result of parsing the IDP program

Return type *IDP*

get_blocks(*blocks*)
returns the AST nodes for the blocks whose names are given

Parameters **blocks** (*List[str]*) – list of names of the blocks to retrieve

Returns list of AST nodes

Return type List[Union[Vocabulary, TheoryBlock, Structure, Procedure, Display]]

execute()
Execute the `main()` procedure block in the IDP program

Parameters **self** (idp_engine.Parse.IDP) –

Return type None

class Vocabulary(***kwargs*)
Bases: idp_engine.Expression.ASTNode

The class of AST nodes representing a vocabulary block.

__init__(***kwargs*)

add_voc_to_block(*block*)
adds the enumerations in a vocabulary to a theory or structure block

Parameters **block** (Theory) – the block to be updated

class Annotations(***kwargs*)
Bases: idp_engine.Expression.ASTNode

__init__(***kwargs*)

class Import(***kwargs*)
Bases: idp_engine.Expression.ASTNode

__init__(***kwargs*)

class TypeDeclaration(***kwargs*)
Bases: idp_engine.Expression.ASTNode

AST node to represent *type <symbol> := <enumeration>*

Parameters

- **name** (*string*) – name of the type
- **arity** (*int*) – the number of arguments
- **sorts** (*List[Symbol]*) – the types of the arguments
- **out** (*Symbol*) – the Boolean Symbol
- **type** (*string*) – Z3 type of an element of the type; same as *name*
- **base_type** – self
- **constructors** (*[Constructor]*) – list of constructors in the enumeration
- **interpretation** (*SymbolInterpretation*) – the symbol interpretation
- **map** (*Dict[string, Expression]*) – a mapping from code to Expression in range

__init__(***kwargs*)

contains_element(*term*, *interpretations*, *extensions*)
returns an Expression that is TRUE when *term* is in the type

Parameters

- **term** (*idp_engine.Expression.Expression*) –
- **interpretations** (*Dict[str, idp_engine.Parse.SymbolInterpretation]*) –
- **extensions** (*Dict[str, Tuple[Optional[List[List[idp_engine.Expression.Expression]]], Optional[Callable]]]*) –

Return type idp_engine.Expression.Expression

class SymbolDeclaration(***kwargs*)
Bases: idp_engine.Expression.ASTNode

The class of AST nodes representing an entry in the vocabulary, declaring one or more symbols. Multi-symbols declaration are replaced by single-symbol declarations before the annotate() stage.

annotations
the annotations given by the expert.

annotations['reading'] is the annotation giving the intended meaning of the expression (in English).

symbols
the symbols being defined, before expansion

Type [Symbol]

name
the identifier of the symbol, after expansion of the node

Type string

arity
the number of arguments

Type int

sorts
the types of the arguments

Type List[Symbol]

out
the type of the symbol

Type Symbol

type
name of the Z3 type of an instance of the symbol

Type string

base_type
base type of the unary predicate (None otherwise)

Type TypeDeclaration

instances
a mapping from the code of a symbol applied to a tuple of arguments to its parsed AST

Type Dict[string, Expression]

range
the list of possible values

Type List[Expression]

private
True if the symbol name starts with '_' (for use in IC)

Type Bool

unit
the unit of the symbol, such as m (meters)

Type str

heading
the heading that the symbol should belong to

Type str

optimizable
whether this symbol should get optimize buttons in the IC

Type bool

__init__(***kwargs*)

has_in_domain(*args*, *interpretations*, *extensions*)
Returns an expression that is TRUE when *args* are in the domain of the symbol.

Parameters

- **args** (*List[Expression]*) – the list of arguments to be checked, e.g. *[1, 2]*
- **interpretations** (*Dict[str, idp_engine.Parse.SymbolInterpretation]*) –
- **extensions** (*Dict[str, Tuple[Optional[List[List[idp_engine.Expression.Expression]]], Optional[Callable]]]*) –

Returns whether *(1,2)* is in the domain of the symbol

Return type Expression

has_in_range(*value*, *interpretations*, *extensions*)
Returns an expression that says whether *value* is in the range of the symbol.

Parameters

- **value** (*idp_engine.Expression.Expression*) –
- **interpretations** (*Dict[str, idp_engine.Parse.SymbolInterpretation]*) –

- **extensions** (*Dict[str, Tuple[Optional[List[List[idp_engine.Expression.Expression]]], Optional[Callable]]]*) –

Return type idp_engine.Expression.Expression

contains_element(*term*, *interpretations*, *extensions*)
returns an Expression that is TRUE when *term* satisfies the predicate

Parameters

- **term** (*idp_engine.Expression.Expression*) –
- **interpretations** (*Dict[str, idp_engine.Parse.SymbolInterpretation]*) –
- **extensions** (*Dict[str, Tuple[Optional[List[List[idp_engine.Expression.Expression]]], Optional[Callable]]]*) –

Return type idp_engine.Expression.Expression

class Symbol(***kwargs*)
Bases: idp_engine.Expression.Expression

Represents a Symbol. Handles synonyms.

name
name of the symbol

Type string

__init__(***kwargs*)

has_element(*term*, *interpretations*, *extensions*)
Returns an expression that says whether *term* is in the type/predicate denoted by *self*.

Parameters

- **term** (*Expression*) – the argument to be checked
- **interpretations** (*Dict[str, SymbolInterpretation]*) –
- **extensions** (*Dict[str, Tuple[Optional[List[List[idp_engine.Expression.Expression]]], Optional[Callable]]]*) –

Returns whether *term* is in the type denoted by *self*.

Return type Expression

annotate(*voc*, *q_vars*)
annotate tree after parsing

Resolve names and determine type as well as variables in the expression

Parameters

- **voc** (*Vocabulary*) – the vocabulary
- **q_vars** (*Dict[str, Variable]*) – the quantifier variables that may appear in the expression

Returns an equivalent AST node, with updated type, .variables

Return type Expression

instantiate(*e0*, *e1*, *problem=None*)
Recursively substitute Variable in e0 by e1 in a copy of self.

Interpret appliedSymbols immediately if grounded (and not occurring in head of definition). Update .variables.

`translate`(*problem*, *vars={}*)
Converts the syntax tree to a Z3 expression, using .value and .simpler if present

Parameters

- **problem** (Theory) – holds the context for the translation (e.g. a cache of translations).
- **vars** (*Dict[id, ExprRef], optional*) – mapping from Variable's id to Z3 translation. Filled in by AQuantifier. Defaults to {}.

Returns Z3 expression

Return type ExprRef

`class TheoryBlock`(***kwargs*)
Bases: idp_engine.Expression.ASTNode

The class of AST nodes representing a theory block.

`__init__`(***kwargs*)

`class Definition`(***kwargs*)
Bases: idp_engine.Expression.ASTNode

The class of AST nodes representing an inductive definition. id (num): unique identifier for each definition

rules ([Rule]): set of rules for the definition, e.g., *!x: p(x) <- q(x)*

canonicals (Dict[Declaration, List[Rule]]): normalized rule for each defined symbol, e.g., *!$p!1$: p($p!1$) <- q($p!1$)*

instantiables (Dict[Declaration], List[Expression]): list of instantiable expressions for each symbol, e.g., *p($p!1$) <=> q($p!1$)*

clarks (Dict[Declaration, Transformed Rule]): normalized rule for each defined symbol (used to be Clark completion) e.g., *!$p!1$: p($p!1$) <=> q($p!1$)*

def_vars (Dict[String, Dict[String, Variable]]): Fresh variables for arguments and result

level_symbols (Dict[SymbolDeclaration, Symbol]): map of recursively defined symbols to level mapping symbols

cache (Dict[SymbolDeclaration, str, Expression]): cache of instantiation of the definition

inst_def_level (int): depth of recursion during instantiation

`__init__`(***kwargs*)

`set_level_symbols`(*voc*)
Calculates which symbols in the definition are recursively defined, creates a corresponding level mapping symbol, and stores these in self.level_symbols.

`add_def_constraints`(*instantiables*, *problem*, *result*)
result is updated with the constraints for this definition.

The *instantiables* (of the definition) are expanded in *problem*.

Parameters

- **instantiables** (*Dict[SymbolDeclaration, List[Expression]]*) – the constraints without the quantification
- **problem** (Theory) – contains the structure for the expansion/interpretation of the constraints
- **result** (*Dict[SymbolDeclaration, Definition, List[Expression]]*) – a mapping from (Symbol, Definition) to the list of constraints

get_instantiables(*interpretations*, *extensions*, *for_explain=False*)
compute Definition.instantiables, with level-mapping if definition is inductive

Uses implications instead of equivalence if *for_explain* is True

Example: *{ p() <- q(). p() <- r().}* Result when not for_explain: *p() <=> q() | r()* Result when for_explain : *p() <= q(). p() <= r(). p() => (q() | r()).*

Parameters

- **for_explain** (*Bool*) – Use implications instead of equivalence, for rule-specific explanations
- **interpretations** (*Dict[str, idp_engine.Parse.SymbolInterpretation]*) –
- **extensions** (*Dict[str, Tuple[Optional[List[List[idp_engine.Expression.Expression]]], Optional[Callable]]]*) –

interpret(*problem*)
updates problem.def_constraints, by expanding the definitions

Parameters **problem** (Theory) – containts the enumerations for the expansion; is updated with the expanded definitions

class Rule(***kwargs*)
Bases: idp_engine.Expression.ASTNode

__init__(***kwargs*)

instantiate_definition(*new_args*, *theory*)
Create an instance of the definition for new_args, and interpret it for theory.

Parameters

- **new_args** (*[Expression]*) – tuple of arguments to be applied to the defined symbol
- **theory** (Theory) – the context for the interpretation

Returns a boolean expression

Return type Expression

rename_args(*new_vars*)
for Clark's completion input : '!v: f(args) <- body(args)' output: '!nv: f(nv) <- nv=args & body(args)'

class Structure(***kwargs*)
Bases: idp_engine.Expression.ASTNode

The class of AST nodes representing an structure block.

__init__(***kwargs*)
The textx parser creates the Structure object. All information used in this method directly comes from text.

annotate(*idp*)
Annotates the structure with the enumerations found in it. Every enumeration is converted into an assignment, which is added to *self.assignments*.

Parameters **idp** – a *Parse.IDP* object.

Returns None

class Enumeration(***kwargs*)
Bases: idp_engine.Expression.ASTNode

Represents an enumeration of tuples of expressions. Used for predicates, or types without n-ary constructors.

tuples
OrderedSet of Tuple of Expression

Type OrderedSet[Tuple]

constructors
List of Constructor

Type List[Constructor], optional

__init__(***kwargs*)

contains(*args*, *function*, *arity=None*, *rank=0*, *tuples=None*, *interpretations=None*, *extensions=None*)
returns an Expression that says whether Tuple args is in the enumeration

Parameters

- **interpretations** (*Optional[Dict[str, idp_engine.Parse.SymbolInterpretation]]*) –
- **extensions** (*Optional[Dict[str, Tuple[Optional[List[List[idp_engine.Expression.Expression]]], Optional[Callable]]]]*) –

Return type idp_engine.Expression.Expression

extensionE(*interpretations=None*, *extensions=None*)
computes the extension of an enumeration, i.e., a set of tuples and a filter

Parameters

- **interpretations** (*Dict[str, "SymbolInterpretation"], optional*) – _description_. Defaults to None.
- **extensions** (*Dict[str, Extension], optional*) – _description_. Defaults to None.

Returns _description_

Return type Extension

class Tuple(***kwargs*)
Bases: idp_engine.Expression.ASTNode

__init__(***kwargs*)

class Display(***kwargs*)
Bases: idp_engine.Expression.ASTNode

__init__(***kwargs*)

run(*idp*)
apply the display block to the idp theory

class Procedure(***kwargs*)
Bases: idp_engine.Expression.ASTNode

__init__(***kwargs*)

5.2.2 idp_engine.Expression

(They are monkey-patched by other modules)

class ASTNode

Bases: `object`

superclass of all AST nodes

check(*condition*, *msg*)

raises an exception if *condition* is not True

Parameters

- **condition** (*Bool*) – condition to be satisfied
- **msg** (*str*) – error message

Raises **IDPZ3Error** – when *condition* is not met

dedup_nodes(*kwargs*, *arg_name*)

pops *arg_name* from kwargs as a list of named items and returns a mapping from name to items

Parameters

- **kwargs** (*Dict[str, ASTNode]*) –
- **arg_name** (*str*) – name of the kwargs argument, e.g. "interpretations"

Returns mapping from *name* to AST nodes

Return type Dict[str, ASTNode]

Raises **AssertionError** – in case of duplicate name

class Expression

Bases: `idp_engine.Expression.ASTNode`

The abstract class of AST nodes representing (sub-)expressions.

code

Textual representation of the expression. Often used as a key.

It is generated from the sub-tree. Some tree transformations change it (e.g., instantiate), others don't.

Type string

sub_exprs

The children of the AST node.

The list may be reduced by simplification.

Type List[Expression]

type

The name of the type of the expression, e.g., `bool`.

Type string

co_constraint

A constraint attached to the node.

For example, the co_constraint of `square(length(top()))` is `square(length(top())) = length(top())*length(top()).`, assuming `square` is appropriately defined.

The co_constraint of a defined symbol applied to arguments is the instantiation of the definition for those arguments. This is useful for definitions over infinite domains, as well as to compute relevant questions.

Type Expression, optional

simpler
A simpler, equivalent expression.

Equivalence is computed in the context of the theory and structure. Simplifying an expression is useful for efficiency and to compute relevant questions.

Type Expression, optional

value
A rigid term equivalent to the expression, obtained by transformation.

Equivalence is computed in the context of the theory and structure.

Type Optional[Expression]

annotations
The set of annotations given by the expert in the IDP-Z3 program.

`annotations['reading']` is the annotation giving the intended meaning of the expression (in English).

Type Dict[str, str]

original
The original expression, before propagation and simplification.

Type Expression

variables
The set of names of the variables in the expression.

Type Set(string)

is_type_constraint_for
name of the symbol for which the expression is a type constraint

Type string

__init__()

copy()
create a deep copy (except for rigid terms and variables)

collect(*questions*, *all_=True*, *co_constraints=True*)
collects the questions in self.

questions is an OrderedSet of Expression Questions are the terms and the simplest sub-formula that can be evaluated. *collect* uses the simplified version of the expression.

all_=False : ignore expanded formulas and AppliedSymbol interpreted in a structure co_constraints=False : ignore co_constraints

default implementation for UnappliedSymbol, AIfExpr, AUnary, Variable, Number_constant, Brackets

collect_symbols(*symbols=None*, *co_constraints=True*)
returns the list of symbol declarations in self, ignoring type constraints

returns Dict[name, Declaration]

collect_nested_symbols(*symbols*, *is_nested*)
returns the set of symbol declarations that occur (in)directly under an aggregate or some nested term, where is_nested is flipped to True the moment we reach such an expression

returns {SymbolDeclaration}

generate_constructors(*constructors*)
fills the list *constructors* with all constructors belonging to open types.

Parameters **constructors** (*dict*) –

co_constraints(*co_constraints*)
collects the constraints attached to AST nodes, e.g. instantiated definitions

co_constraints is an OrderedSet of Expression

is_assignment()

Returns True if *self* assigns a rigid term to a rigid function application

Return type bool

update_exprs(*new_exprs*)
change sub_exprs and simplify, while keeping relevant info.

substitute(*e0*, *e1*, *assignments*, *tag=None*)
recursively substitute e0 by e1 in self (e0 is not a Variable)

if tag is present, updates assignments with symbolic propagation of co-constraints.

implementation for everything but AppliedSymbol, UnappliedSymbol and Fresh_variable

instantiate(*e0*, *e1*, *problem=None*)
Recursively substitute Variable in e0 by e1 in a copy of self.

Interpret appliedSymbols immediately if grounded (and not occurring in head of definition). Update .variables.

instantiate1(*e0*, *e1*, *problem=None*)
Recursively substitute Variable in e0 by e1 in self.

Interpret appliedSymbols immediately if grounded (and not occurring in head of definition). Update .variables.

simplify_with(*assignments*)
simplify the expression using the assignments

Parameters

- **self** (*idp_engine.Expression.Expression*) –
- **assignments** (*idp_engine.Assignments.Assignments*) –

Return type idp_engine.Expression.Expression

symbolic_propagate(*assignments*, *tag*, *truth=true*)
updates assignments with the consequences of *self=truth*.

The consequences are obtained by symbolic processing (no calls to Z3).

Parameters

- **assignments** (*Assignments*) – The set of assignments to update.
- **truth** (*Expression, optional*) – The truth value of the expression *self*. Defaults to TRUE.
- **tag** (*Status*) –

propagate1(*assignments*, *tag*, *truth*)
returns the list of symbolic_propagate of self, ignoring value and simpler

`translate`(*problem*, *vars={}*)
Converts the syntax tree to a Z3 expression, using .value and .simpler if present

Parameters

- **problem** (Theory) – holds the context for the translation (e.g. a cache of translations).
- **vars** (*Dict[id, ExprRef], optional*) – mapping from Variable's id to Z3 translation. Filled in by AQuantifier. Defaults to {}.

Returns Z3 expression

Return type ExprRef

`as_set_condition`()
Returns an equivalent expression of the type "x in y", or None

Returns meaning "expr is (not) in enumeration"

Return type Tuple[Optional[AppliedSymbol], Optional[bool], Optional[Enumeration]]

`split_equivalences`()
Returns an equivalent expression where equivalences are replaced by implications

Returns Expression

`add_level_mapping`(*level_symbols*, *head*, *pos_justification*, *polarity*)

Returns an expression where level mapping atoms (e.g., lvl_p > lvl_q) are added to atoms containing recursive symbols.

Parameters

- **level_symbols** (-) – the level mapping symbols as well as their corresponding recursive symbols
- **head** (-) – head of the rule we are adding level mapping symbols to.
- **pos_justification** (-) – whether we are adding symbols to the direct positive justification (e.g., head => body) or direct negative justification (e.g., body => head) part of the rule.
- **polarity** (-) – whether the current expression occurs under negation.

Returns Expression

`annotate`(*voc*, *q_vars*)
annotate tree after parsing

Resolve names and determine type as well as variables in the expression

Parameters

- **voc** (*Vocabulary*) – the vocabulary
- **q_vars** (*Dict[str, Variable]*) – the quantifier variables that may appear in the expression

Returns an equivalent AST node, with updated type, .variables

Return type Expression

`annotate1`()
annotations that are common to __init__ and make()

interpret(*problem*)
uses information in the problem and its vocabulary to: - expand quantifiers in the expression - simplify the expression using known assignments and enumerations - instantiate definitions

Parameters **problem** (Theory) – the Theory to apply

Returns the resulting expression

Return type Expression

class Constructor(***kwargs*)
Bases: idp_engine.Expression.ASTNode

Constructor declaration

name
name of the constructor

Type string

sorts
types of the arguments of the constructor

Type List[Symbol]

type
name of the type that contains this constructor

Type string

arity
number of arguments of the constructor

Type Int

tester
function to test if the constructor

Type SymbolDeclaration

has been applied to some arguments

Type e.g., is_rgb

symbol
only for Symbol constructors

Type Symbol

__init__(***kwargs*)

class AIfExpr(***kwargs*)
Bases: idp_engine.Expression.Expression

__init__(***kwargs*)

collect_nested_symbols(*symbols*, *is_nested*)
returns the set of symbol declarations that occur (in)directly under an aggregate or some nested term, where is_nested is flipped to True the moment we reach such an expression

returns {SymbolDeclaration}

annotate1()
annotations that are common to __init__ and make()

translate1(*problem*, *vars={}*)
Converts the syntax tree to a Z3 expression, ignoring .value and .simpler

Parameters

- **problem** (Theory) – holds the context for the translation (e.g. a cache of translations).
- **vars** (*Dict[id, ExprRef], optional*) – mapping from Variable's id to Z3 translation. Filled in by AQuantifier. Defaults to {}.

Returns Z3 expression

Return type ExprRef

update_exprs(*new_exprs*)
change sub_exprs and simplify, while keeping relevant info.

class Quantee(***kwargs*)
Bases: idp_engine.Expression.Expression

represents the description of quantification, e.g., *x in T* or *(x,y) in P* The *Concept* type may be qualified, e.g. *Concept[Color->Bool]*

vars
the (tuples of) variables being quantified

Type List[List[Variable]]

subtype
a literal Type to quantify over, e.g., *Color* or *Concept[Color->Bool]*.

Type Type, Optional

sort
a dereferencing expression, e.g.,. *$(i)*.

Type SymbolExpr, Optional

sub_exprs
the (unqualified) type or predicate to quantify over,

Type List[SymbolExpr], Optional

e.g., `[Color], [Concept] or [$

Type i

arity
the length of the tuple of variables

Type int

decl
the (unqualified) Declaration to quantify over, after resolution of *$(i)*.

Type SymbolDeclaration, Optional

e.g., the declaration of `Color`

__init__(***kwargs*)

update_exprs(*new_exprs*)
change sub_exprs and simplify, while keeping relevant info.

class AQuantification(***kwargs*)
Bases: idp_engine.Expression.Expression

ASTNode representing a quantified formula

Parameters

- **annotations** (*Dict[str, str]*) – The set of annotations given by the expert in the IDP-Z3 program.

  annotations['reading'] is the annotation giving the intended meaning of the expression (in English).

- **q** (*str*) – either ‘’ or ‘’
- **quantees** (*List[Quantee]*) – list of variable declarations
- **f** (*Expression*) – the formula being quantified

__init__(***kwargs*)

classmethod make(*q*, *quantees*, *f*, *annotations=None*)
make and annotate a quantified formula

copy()
create a deep copy (except for rigid terms and variables)

collect(*questions*, *all_=True*, *co_constraints=True*)
collects the questions in self.

questions is an OrderedSet of Expression Questions are the terms and the simplest sub-formula that can be evaluated. *collect* uses the simplified version of the expression.

all_=False : ignore expanded formulas and AppliedSymbol interpreted in a structure co_constraints=False : ignore co_constraints

default implementation for UnappliedSymbol, AIfExpr, AUnary, Variable, Number_constant, Brackets

collect_symbols(*symbols=None*, *co_constraints=True*)
returns the list of symbol declarations in self, ignoring type constraints

returns Dict[name, Declaration]

annotate(*voc*, *q_vars*)
annotate tree after parsing

Resolve names and determine type as well as variables in the expression

Parameters

- **voc** (*Vocabulary*) – the vocabulary
- **q_vars** (*Dict[str, Variable]*) – the quantifier variables that may appear in the expression

Returns an equivalent AST node, with updated type, .variables

Return type Expression

annotate1()
annotations that are common to __init__ and make()

instantiate1(*e0*, *e1*, *problem=None*)
Recursively substitute Variable in e0 by e1 in self.

Interpret appliedSymbols immediately if grounded (and not occurring in head of definition). Update .variables.

interpret(*problem*)
apply information in the problem and its vocabulary

Parameters **problem** (Theory) – the problem to be applied

Returns the expanded quantifier expression

Return type Expression

symbolic_propagate(*assignments*, *tag*, *truth=true*)
updates assignments with the consequences of *self=truth*.

The consequences are obtained by symbolic processing (no calls to Z3).

Parameters

- **assignments** (*Assignments*) – The set of assignments to update.
- **truth** (*Expression, optional*) – The truth value of the expression *self*. Defaults to TRUE.

update_exprs(*new_exprs*)
change sub_exprs and simplify, while keeping relevant info.

class Operator(***kwargs*)
Bases: idp_engine.Expression.Expression

__init__(***kwargs*)

classmethod make(*ops*, *operands*, *annotations=None*)
creates a BinaryOp beware: cls must be specific for ops !

collect(*questions*, *all_=True*, *co_constraints=True*)
collects the questions in self.

questions is an OrderedSet of Expression Questions are the terms and the simplest sub-formula that can be evaluated. *collect* uses the simplified version of the expression.

all_=False : ignore expanded formulas and AppliedSymbol interpreted in a structure co_constraints=False : ignore co_constraints

default implementation for UnappliedSymbol, AIfExpr, AUnary, Variable, Number_constant, Brackets

collect_nested_symbols(*symbols*, *is_nested*)
returns the set of symbol declarations that occur (in)directly under an aggregate or some nested term, where is_nested is flipped to True the moment we reach such an expression

returns {SymbolDeclaration}

annotate1()
annotations that are common to __init__ and make()

class AImplication(***kwargs*)
Bases: idp_engine.Expression.Operator

add_level_mapping(*level_symbols*, *head*, *pos_justification*, *polarity*)

Returns an expression where level mapping atoms (e.g., lvl_p > lvl_q) are added to atoms containing recursive symbols.

Parameters

- **level_symbols** (-) – the level mapping symbols as well as their corresponding recursive symbols
- **head** (-) – head of the rule we are adding level mapping symbols to.
- **pos_justification** (-) – whether we are adding symbols to the direct positive justification (e.g., head => body) or direct negative justification (e.g., body => head) part of the rule.

- **polarity** (-) – whether the current expression occurs under negation.

Returns Expression

annotate1()
annotations that are common to __init__ and make()

update_exprs(*new_exprs*)
change sub_exprs and simplify, while keeping relevant info.

class AEquivalence(***kwargs*)
Bases: idp_engine.Expression.Operator

split_equivalences()
Returns an equivalent expression where equivalences are replaced by implications

Returns Expression

annotate1()
annotations that are common to __init__ and make()

update_exprs(*new_exprs*)
change sub_exprs and simplify, while keeping relevant info.

class ARImplication(***kwargs*)
Bases: idp_engine.Expression.Operator

add_level_mapping(*level_symbols*, *head*, *pos_justification*, *polarity*)

Returns an expression where level mapping atoms (e.g., lvl_p > lvl_q) are added to atoms containing recursive symbols.

Parameters

- **level_symbols** (-) – the level mapping symbols as well as their corresponding recursive symbols
- **head** (-) – head of the rule we are adding level mapping symbols to.
- **pos_justification** (-) – whether we are adding symbols to the direct positive justification (e.g., head => body) or direct negative justification (e.g., body => head) part of the rule.
- **polarity** (-) – whether the current expression occurs under negation.

Returns Expression

annotate(*voc*, *q_vars*)
annotate tree after parsing

Resolve names and determine type as well as variables in the expression

Parameters

- **voc** (*Vocabulary*) – the vocabulary
- **q_vars** (*Dict[str, Variable]*) – the quantifier variables that may appear in the expression

Returns an equivalent AST node, with updated type, .variables

Return type Expression

class ADisjunction(***kwargs*)
Bases: idp_engine.Expression.Operator

propagate1(*assignments*, *tag*, *truth=true*)
returns the list of symbolic_propagate of self, ignoring value and simpler

update_exprs(*new_exprs*)
change sub_exprs and simplify, while keeping relevant info.

class AConjunction(***kwargs*)
Bases: idp_engine.Expression.Operator

propagate1(*assignments*, *tag*, *truth=true*)
returns the list of symbolic_propagate of self, ignoring value and simpler

update_exprs(*new_exprs*)
change sub_exprs and simplify, while keeping relevant info.

class AComparison(***kwargs*)
Bases: idp_engine.Expression.Operator

__init__(***kwargs*)

is_assignment()

Returns True if *self* assigns a rigid term to a rigid function application

Return type bool

annotate(*voc*, *q_vars*)
annotate tree after parsing

Resolve names and determine type as well as variables in the expression

Parameters

- **voc** (*Vocabulary*) – the vocabulary
- **q_vars** (*Dict[str, Variable]*) – the quantifier variables that may appear in the expression

Returns an equivalent AST node, with updated type, .variables

Return type Expression

as_set_condition()
Returns an equivalent expression of the type "x in y", or None

Returns meaning "expr is (not) in enumeration"

Return type Tuple[Optional[AppliedSymbol], Optional[bool], Optional[Enumeration]]

propagate1(*assignments*, *tag*, *truth=true*)
returns the list of symbolic_propagate of self, ignoring value and simpler

update_exprs(*new_exprs*)
change sub_exprs and simplify, while keeping relevant info.

class ASumMinus(***kwargs*)
Bases: idp_engine.Expression.Operator

update_exprs(*new_exprs*)
change sub_exprs and simplify, while keeping relevant info.

class AMultDiv(***kwargs*)
Bases: idp_engine.Expression.Operator

update_exprs(*new_exprs*)
change sub_exprs and simplify, while keeping relevant info.

class APower(***kwargs*)
Bases: idp_engine.Expression.Operator

update_exprs(*new_exprs*)
change sub_exprs and simplify, while keeping relevant info.

class AUnary(***kwargs*)
Bases: idp_engine.Expression.Expression

__init__(***kwargs*)

add_level_mapping(*level_symbols*, *head*, *pos_justification*, *polarity*)

Returns an expression where level mapping atoms (e.g., lvl_p > lvl_q) are added to atoms containing recursive symbols.

Parameters

- **level_symbols** (-) – the level mapping symbols as well as their corresponding recursive symbols
- **head** (-) – head of the rule we are adding level mapping symbols to.
- **pos_justification** (-) – whether we are adding symbols to the direct positive justification (e.g., head => body) or direct negative justification (e.g., body => head) part of the rule.
- **polarity** (-) – whether the current expression occurs under negation.

Returns Expression

annotate1()
annotations that are common to __init__ and make()

as_set_condition()
Returns an equivalent expression of the type "x in y", or None

Returns meaning "expr is (not) in enumeration"

Return type Tuple[Optional[AppliedSymbol], Optional[bool], Optional[Enumeration]]

propagate1(*assignments*, *tag*, *truth=true*)
returns the list of symbolic_propagate of self, ignoring value and simpler

update_exprs(*new_exprs*)
change sub_exprs and simplify, while keeping relevant info.

class AAggregate(***kwargs*)
Bases: idp_engine.Expression.Expression

__init__(***kwargs*)

copy()
create a deep copy (except for rigid terms and variables)

collect(*questions*, *all_=True*, *co_constraints=True*)
collects the questions in self.

questions is an OrderedSet of Expression Questions are the terms and the simplest sub-formula that can be evaluated. *collect* uses the simplified version of the expression.

all_=False : ignore expanded formulas and AppliedSymbol interpreted in a structure co_constraints=False : ignore co_constraints

default implementation for UnappliedSymbol, AIfExpr, AUnary, Variable, Number_constant, Brackets

`collect_symbols`(*symbols=None*, *co_constraints=True*)
returns the list of symbol declarations in self, ignoring type constraints

returns Dict[name, Declaration]

`collect_nested_symbols`(*symbols*, *is_nested*)
returns the set of symbol declarations that occur (in)directly under an aggregate or some nested term, where is_nested is flipped to True the moment we reach such an expression

returns {SymbolDeclaration}

`annotate`(*voc*, *q_vars*)
annotate tree after parsing

Resolve names and determine type as well as variables in the expression

Parameters

- **voc** (*Vocabulary*) – the vocabulary
- **q_vars** (*Dict[str, Variable]*) – the quantifier variables that may appear in the expression

Returns an equivalent AST node, with updated type, .variables

Return type Expression

`annotate1`()
annotations that are common to __init__ and make()

`instantiate1`(*e0*, *e1*, *problem=None*)
Recursively substitute Variable in e0 by e1 in self.

Interpret appliedSymbols immediately if grounded (and not occurring in head of definition). Update .variables.

`interpret`(*problem*)
uses information in the problem and its vocabulary to: - expand quantifiers in the expression - simplify the expression using known assignments and enumerations - instantiate definitions

Parameters **problem** (Theory) – the Theory to apply

Returns the resulting expression

Return type Expression

`update_exprs`(*new_exprs*)
change sub_exprs and simplify, while keeping relevant info.

`class AppliedSymbol`(***kwargs*)
Bases: `idp_engine.Expression.Expression`

Represents a symbol applied to arguments

Parameters

- **symbol** (*Expression*) – the symbol to be applied to arguments
- **is_enumerated** (*string*) – '' or 'is enumerated' or 'is not enumerated'
- **is_enumeration** (*string*) – '' or 'in' or 'not in'

- **in_enumeration** (*Enumeration*) – the enumeration following 'in'
- **decl** (*Declaration*) – the declaration of the symbol, if known
- **in_head** (*Bool*) – True if the AppliedSymbol occurs in the head of a rule

__init__(***kwargs*)

copy()
create a deep copy (except for rigid terms and variables)

collect(*questions*, *all_=True*, *co_constraints=True*)
collects the questions in self.

questions is an OrderedSet of Expression Questions are the terms and the simplest sub-formula that can be evaluated. *collect* uses the simplified version of the expression.

all_=False : ignore expanded formulas and AppliedSymbol interpreted in a structure co_constraints=False : ignore co_constraints

default implementation for UnappliedSymbol, AIfExpr, AUnary, Variable, Number_constant, Brackets

collect_symbols(*symbols=None*, *co_constraints=True*)
returns the list of symbol declarations in self, ignoring type constraints

returns Dict[name, Declaration]

collect_nested_symbols(*symbols*, *is_nested*)
returns the set of symbol declarations that occur (in)directly under an aggregate or some nested term, where is_nested is flipped to True the moment we reach such an expression

returns {SymbolDeclaration}

generate_constructors(*constructors*)
fills the list *constructors* with all constructors belonging to open types.

Parameters **constructors** (*dict*) –

add_level_mapping(*level_symbols*, *head*, *pos_justification*, *polarity*)

Returns an expression where level mapping atoms (e.g., lvl_p > lvl_q) are added to atoms containing recursive symbols.

Parameters

- **level_symbols** (-) – the level mapping symbols as well as their corresponding recursive symbols
- **head** (-) – head of the rule we are adding level mapping symbols to.
- **pos_justification** (-) – whether we are adding symbols to the direct positive justification (e.g., head => body) or direct negative justification (e.g., body => head) part of the rule.
- **polarity** (-) – whether the current expression occurs under negation.

Returns Expression

annotate(*voc*, *q_vars*)
annotate tree after parsing

Resolve names and determine type as well as variables in the expression

Parameters

- **voc** (*Vocabulary*) – the vocabulary
- **q_vars** (*Dict[str, Variable]*) – the quantifier variables that may appear in the expression

Returns an equivalent AST node, with updated type, .variables

Return type Expression

annotate1()
annotations that are common to __init__ and make()

as_set_condition()
Returns an equivalent expression of the type "x in y", or None

Returns meaning "expr is (not) in enumeration"

Return type Tuple[Optional[AppliedSymbol], Optional[bool], Optional[Enumeration]]

instantiate1(*e0*, *e1*, *problem=None*)
Recursively substitute Variable in e0 by e1 in self.

Interpret appliedSymbols immediately if grounded (and not occurring in head of definition). Update .variables.

interpret(*problem*)
uses information in the problem and its vocabulary to: - expand quantifiers in the expression - simplify the expression using known assignments and enumerations - instantiate definitions

Parameters **problem** (Theory) – the Theory to apply

Returns the resulting expression

Return type Expression

substitute(*e0*, *e1*, *assignments*, *tag=None*)
recursively substitute e0 by e1 in self

update_exprs(*new_exprs*)
change sub_exprs and simplify, while keeping relevant info.

class UnappliedSymbol(***kwargs*)
Bases: idp_engine.Expression.Expression

The result of parsing a symbol not applied to arguments. Can be a constructor or a quantified variable.

Variables are converted to Variable() by annotate().

__init__(***kwargs*)

classmethod construct(*constructor*)
Create an UnappliedSymbol from a constructor

Parameters **constructor** (*idp_engine.Expression.Constructor*) –

annotate(*voc*, *q_vars*)
annotate tree after parsing

Resolve names and determine type as well as variables in the expression

Parameters

- **voc** (*Vocabulary*) – the vocabulary
- **q_vars** (*Dict[str, Variable]*) – the quantifier variables that may appear in the expression

Returns an equivalent AST node, with updated type, .variables

Return type Expression

class Variable(***kwargs*)

Bases: idp_engine.Expression.Expression

AST node for a variable in a quantification or aggregate

Parameters

- **name** (*str*) – name of the variable
- **sort** (*Optional[Symbol]*) – sort of the variable, if known

__init__(***kwargs*)

copy()

create a deep copy (except for rigid terms and variables)

annotate1()

annotations that are common to __init__ and make()

annotate(*voc*, *q_vars*)

annotate tree after parsing

Resolve names and determine type as well as variables in the expression

Parameters

- **voc** (*Vocabulary*) – the vocabulary
- **q_vars** (*Dict[str, Variable]*) – the quantifier variables that may appear in the expression

Returns an equivalent AST node, with updated type, .variables

Return type Expression

instantiate1(*e0*, *e1*, *problem=None*)

Recursively substitute Variable in e0 by e1 in self.

Interpret appliedSymbols immediately if grounded (and not occurring in head of definition). Update .variables.

interpret(*problem*)

uses information in the problem and its vocabulary to: - expand quantifiers in the expression - simplify the expression using known assignments and enumerations - instantiate definitions

Parameters **problem** (Theory) – the Theory to apply

Returns the resulting expression

Return type Expression

substitute(*e0*, *e1*, *assignments*, *tag=None*)

recursively substitute e0 by e1 in self (e0 is not a Variable)

if tag is present, updates assignments with symbolic propagation of co-constraints.

implementation for everything but AppliedSymbol, UnappliedSymbol and Fresh_variable

translate(*problem*, *vars={}*)

Converts the syntax tree to a Z3 expression, using .value and .simpler if present

Parameters

- **problem** (Theory) – holds the context for the translation (e.g. a cache of translations).

- **vars** (*Dict[id, ExprRef], optional*) – mapping from Variable's id to Z3 translation. Filled in by AQuantifier. Defaults to {}.

Returns Z3 expression

Return type ExprRef

class Number(***kwargs*)

Bases: idp_engine.Expression.Expression

__init__(***kwargs*)

real()

converts the INT number to REAL

annotate(*voc*, *q_vars*)

annotate tree after parsing

Resolve names and determine type as well as variables in the expression

Parameters

- **voc** (*Vocabulary*) – the vocabulary
- **q_vars** (*Dict[str, Variable]*) – the quantifier variables that may appear in the expression

Returns an equivalent AST node, with updated type, .variables

Return type Expression

translate(*problem*, *vars={}*)

Converts the syntax tree to a Z3 expression, using .value and .simpler if present

Parameters

- **problem** (Theory) – holds the context for the translation (e.g. a cache of translations).
- **vars** (*Dict[id, ExprRef], optional*) – mapping from Variable's id to Z3 translation. Filled in by AQuantifier. Defaults to {}.

Returns Z3 expression

Return type ExprRef

class Brackets(***kwargs*)

Bases: idp_engine.Expression.Expression

annotate1()

annotations that are common to __init__ and make()

symbolic_propagate(*assignments*, *tag*, *truth=true*)

updates assignments with the consequences of *self=truth*.

The consequences are obtained by symbolic processing (no calls to Z3).

Parameters

- **assignments** (*Assignments*) – The set of assignments to update.
- **truth** (*Expression, optional*) – The truth value of the expression *self*. Defaults to TRUE.

update_exprs(*new_exprs*)

change sub_exprs and simplify, while keeping relevant info.

__init__(***kwargs*)

5.2.3 idp_engine.Annotate

Methods to annotate the Abstract Syntax Tree (AST) of an IDP-Z3 program.

get_instantiables(*self*, *interpretations*, *extensions*, *for_explain=False*)
compute Definition.instantiables, with level-mapping if definition is inductive

Uses implications instead of equivalence if *for_explain* is True

Example: *{ p() <- q(). p() <- r().}* Result when not for_explain: *p() <=> q() | r()* Result when for_explain : *p() <= q(). p() <= r(). p() => (q() | r()).*

Parameters

- **for_explain** (*Bool*) – Use implications instead of equivalence, for rule-specific explanations
- **interpretations** (*Dict[str, idp_engine.Parse.SymbolInterpretation]*) –
- **extensions** (*Dict[str, Tuple[Optional[List[List[idp_engine.Expression.Expression]]], Optional[Callable]]]*) –

rename_args(*self*, *new_vars*)
for Clark's completion input : '!v: f(args) <- body(args)' output: '!nv: f(nv) <- nv=args & body(args)'

5.2.4 idp_engine.Interpret

Methods to ground / interpret a theory in a data structure

- expand quantifiers
- replace symbols interpreted in the structure by their interpretation

This module also includes methods to:

- substitute a node by another in an AST tree
- instantiate an expresion, i.e. replace a variable by a value

This module monkey-patches the ASTNode class and sub-classes.

(see docs/zettlr/Substitute.md)

add_def_constraints(*self*, *instantiables*, *problem*, *result*)
result is updated with the constraints for this definition.

The *instantiables* (of the definition) are expanded in *problem*.

Parameters

- **instantiables** (*Dict[SymbolDeclaration, List[Expression]]*) – the constraints without the quantification
- **problem** (Theory) – contains the structure for the expansion/interpretation of the constraints
- **result** (*Dict[SymbolDeclaration, Definition, List[Expression]]*) – a mapping from (Symbol, Definition) to the list of constraints

extension(*self*, *interpretations*, *extensions*)
returns the extension of a Type, given some interpretations.

Normally, the extension is already in *extensions*. However, for Concept[T->T], an additional filtering is applied.

Parameters

- **interpretations** (*Dict[str, SymbolInterpretation]*) –
- **symbols** (*the known interpretations of types and*) –
- **extensions** (*Dict[str, Tuple[Optional[List[List[idp_engine.Expression.Expression]]], Optional[Callable]]]*) –

Returns a superset of the extension of self, and a function that, given arguments, returns an Expression that says whether the arguments are in the extension of self

Return type Extension

5.2.5 idp_engine.Simplify

Methods to simplify a logic expression.

This module monkey-patches the Expression class and sub-classes.

join_set_conditions(*assignments*)

In a list of assignments, merge assignments that are set-conditions on the same term.

An equality and a membership predicate (*in* operator) are both set-conditions.

Parameters **assignments** (*List[Assignment]*) – the list of assignments to make more compact

Returns the compacted list of assignments

Return type List[Assignment]

5.2.6 idp_engine.Propagate

Computes the consequences of an expression, i.e., the sub-expressions that are necessarily true (or false) if the expression is true (or false)

It has 2 parts: * symbolic propagation * Z3 propagation

This module monkey-patches the Expression and Theory classes and sub-classes.

simplify_with(*self*, *assignments*)

simplify the expression using the assignments

Parameters

- **self** (*idp_engine.Expression.Expression*) –
- **assignments** (*idp_engine.Assignments.Assignments*) –

Return type idp_engine.Expression.Expression

5.2.7 idp_engine.idp_to_Z3

Translates AST tree to Z3

TODO: vocabulary

get_symbols_z(*zexpr*, *symbols*)

adds the symbols in zexpr to symbols

Parameters

- **zexpr** (*ExprRef*) – a Z3 expression

- **symbols** (*set(str)*) – set of symbol name

5.2.8 idp_engine.Theory

Class to represent a collection of theory and structure blocks.

class Propagation(*value*)
Describe propagation method

class Theory(**theories*, *extended=False*)
A collection of theory and structure blocks.

assignments (Assignments): the set of assignments. The assignments are updated by the different steps of the problem resolution. Assignments include inequalities and quantified formula when the problem is extended

Parameters

- **theories** (*Union[idp_engine.Parse.TheoryBlock, idp_engine.Parse.Structure, Theory]*) –
- **extended** (*bool*) –

Return type None

__init__(**theories*, *extended=False*)
Creates an instance of `Theory` for the list of theories, e.g., `Theory(T,S)`.

Parameters

- **theories** (*Union[TheoryBlock, Structure, Theory]*) – 1 or more (data) theories.
- **extended** (*bool, optional*) – use *True* when the truth value of inequalities and quantified formula is of interest (e.g. for the Interactive Consultant). Defaults to False.

Return type None

add(**theories*)
Adds a list of theories to the theory.

Parameters **theories** (*Union[TheoryBlock, Structure, Theory]*) – 1 or more (data) theories.

Return type *idp_engine.Theory.Theory*

assert_(*code*, *value*, *status=Status.GIVEN*)
asserts that an expression has a value (or not), e.g. `theory.assert_("p()", True)`

Parameters

- **code** (*str*) – the code of the expression, e.g., `"p()"`
- **value** (*Any*) – a Python value, e.g., `True`
- **status** (*Status, Optional*) – how the value was obtained. Default: S.GIVEN

Return type *idp_engine.Theory.Theory*

copy()
Returns an independent copy of a theory.

Return type *idp_engine.Theory.Theory*

decision_table(*goal_string=''*, *timeout_seconds=20*, *max_rows=50*, *first_hit=True*, *verify=False*)
Experimental. Returns the rows for a decision table that defines `goal_string`.

`goal_string` must be a predicate application defined in the theory. The theory must be created with `extended=True`.

Parameters

- **goal_string** (*str, optional*) – the last column of the table.
- **timeout_seconds** (*int, optional*) – maximum duration in seconds. Defaults to 20.
- **max_rows** (*int, optional*) – maximum number of rows. Defaults to 50.
- **first_hit** (*bool, optional*) – requested hit-policy. Defaults to True.
- **verify** (*bool, optional*) – request verification of table completeness. Defaults to False

Returns the non-empty cells of the decision table for `goal_string`, given `self`. bool: whether or not the timeout limit was reached.

Return type list(list(Assignment))

determine_relevance()
Determines the questions that are relevant in a model, or that can appear in a justification of a `goal_symbol`.

When an *irrelevant* value is changed in a model M of the theory, the resulting M' structure is still a model. Relevant questions are those that are not irrelevant.

Call must be made after a propagation, on a Theory created with `extended=True`. The result is found in the `relevant` attribute of the assignments in `self.assignments`.

If `goal_symbol` has an enumeration in the theory (e.g., `goal_symbol := {`tax_amount}.`), relevance is computed relative to those goals.

Definitions in the theory are ignored, unless they influence axioms in the theory or goals in `goal_symbol`.

Returns the Theory with relevant information in `self.assignments`.

Return type *Theory*

Parameters **self** (idp_engine.Theory.Theory) –

disable_law(*code*)
Disables a law, represented as a code string taken from the output of explain(...).

The law should not result from a structure (e.g., from `p:=true.`) or from a types (e.g., from `T:={1..10}` and `c: () -> T`).

Parameters **code** (*str*) – the code of the law to be disabled

Raises **AssertionError** – if code is unknown

Return type *idp_engine.Theory.Theory*

enable_law(*code*)
Enables a law, represented as a code string taken from the output of explain(...).

The law should not result from a structure (e.g., from `p:=true.`) or from a types (e.g., from `T:={1..10}` and `c: () -> T`).

Parameters **code** (*str*) – the code of the law to be enabled

Raises **AssertionError** – if code is unknown

Return type *idp_engine.Theory.Theory*

expand(*max=10*, *timeout_seconds=10*, *complete=False*)

Generates a list of models of the theory that are expansion of the known assignments.

The result is limited to `max` models (10 by default), or unlimited if `max` is 0. The search for new models is stopped when processing exceeds `timeout_seconds` (in seconds) (unless it is 0). The models can be asked to be complete or partial (i.e., in which "don't care" terms are not specified).

The string message can be one of the following:

- `No models.`
- `More models may be available. Change the max argument to see them.`
- `More models may be available. Change the timeout_seconds argument to see them.`
- `More models may be available. Change the max and timeout_seconds arguments to see them.`

Parameters

- **max** (*int, optional*) – maximum number of models. Defaults to 10.
- **timeout_seconds** (*int, optional*) – timeout_seconds in seconds. Defaults to 10.
- **complete** (*bool, optional*) – `True` for complete models. Defaults to False.

Yields the models, followed by a string message

Return type Iterator[Union[idp_engine.Assignments.Assignments, str]]

explain(*consequence=None*)

Returns the facts and laws that make the Theory unsatisfiable, or that explains a consequence.

Parameters

- **self** (Theory) – the problem state
- **consequence** (*string, optional*) – the code of the consequence to be explained. Must be a key in `self.assignments`

Returns list of facts and laws that explain the consequence

Return type (List[Assignment], List[Expression])]

formula()

Returns a Z3 object representing the logic formula equivalent to the theory.

This object can be converted to a string using `str()`.

Return type z3.z3.BoolRef

get_range(*term*)

Returns a list of the possible values of the term.

Parameters **term** (*str*) – terms whose possible values are requested, e.g. `subtype()`. Must be a key in `self.assignments`

Returns e.g., `['right triangle', 'regular triangle']`

Return type List[str]

optimize(*term*, *minimize=True*)

Updates the Theory so that the value of term in the `assignments` property is the optimal value that is compatible with the Theory.

Chain it with a call to *expand* to obtain a model, or to *propagate* to propagate the optimal value.

Parameters

- **term** (*str*) – e.g., `"Length(1)"`
- **minimize** (*bool*) – `True` to minimize `term`, `False` to maximize it

Return type *idp_engine.Theory.Theory*

propagate(*tag=Status.CONSEQUENCE*, *method=Propagation.DEFAULT*)

Returns the theory with its `assignments` property updated with values for all terms and atoms that have the same value in every model of the theory.

`self.satisfied` is also updated to reflect satisfiability.

Terms and propositions starting with _ are ignored.

Args: tag (S): the status of propagated assignments method (Propagation): the particular propagation to use

Parameters

- **tag** (*idp_engine.Assignments.Status*) –
- **method** (*idp_engine.Theory.Propagation*) –

Return type *idp_engine.Theory.Theory*

simplify()

Returns a simpler copy of the theory, with a simplified formula obtained by substituting terms and atoms by their known values.

Assignments obtained by propagation become UNIVERSAL constraints.

Return type *idp_engine.Theory.Theory*

symbolic_propagate(*tag=Status.UNIVERSAL*)

Returns the theory with its `assignments` property updated with direct consequences of the constraints of the theory.

This propagation is less complete than `propagate()`.

Parameters **tag** (*S*) – the status of propagated assignments

Return type *idp_engine.Theory.Theory*

to_smt_lib()

Returns an SMT-LIB version of the theory

Return type str

5.2.9 idp_engine.Assignments

Classes to store assignments of values to questions

class Status(*value*)

Describes how the value of a question was obtained

class Assignment(*sentence*, *value*, *status*, *relevant=True*)

Represent the assignment of a value to a question. Questions can be:

- predicates and functions applied to arguments,
- comparisons,

- outermost quantified expressions

A value is a rigid term.

An assignment also has a reference to the symbol under which it should be displayed.

Parameters

- **sentence** (*idp_engine.Expression.Expression*) –
- **value** (*Optional[idp_engine.Expression.Expression]*) –
- **status** (*Optional[idp_engine.Assignments.Status]*) –
- **relevant** (*Optional[bool]*) –

sentence
the question to be assigned a value

Type Expression

value
a rigid term

Type Expression, optional

status
qualifies how the value was obtained

Type Status, optional

is_certainly_undefined
True for functions applied to arguments certainly outside of its domain

Type bool

relevant
states whether the sentence is relevant

Type bool, optional

symbol_decl
declaration of the symbol under which

Type SymbolDeclaration

it should be displayed in the IC.

__init__(*sentence*, *value*, *status*, *relevant=True*)

Parameters

- **sentence** (*idp_engine.Expression.Expression*) –
- **value** (*Optional[idp_engine.Expression.Expression]*) –
- **status** (*Optional[idp_engine.Assignments.Status]*) –
- **relevant** (*Optional[bool]*) –

same_as(*other*)
returns True if self has the same sentence and truth value as other.

Parameters **other** (*Assignment*) – an assignment

Returns True if self has the same sentence and truth value as other.

Return type bool

negate()
returns an Assignment for the same sentence, but an opposite truth value.

Raises **AssertionError** – Cannot negate a non-boolean assignment

Returns returns an Assignment for the same sentence, but an opposite truth value.

Return type [type]

as_set_condition()
returns an equivalent set condition, or None

Returns meaning "appSymb is (not) in enumeration"

Return type Tuple[Optional[AppliedSymbol], Optional[bool], Optional[Enumeration]]

unset()
Unsets the value of an assignment.

Returns None

Return type None

class Assignments(**arg*, ***kw*)
Contains a set of Assignment

__init__(**arg*, ***kw*)

copy() → a shallow copy of D

Parameters **shallow** (*bool*) –

Return type idp_engine.Assignments.Assignments

5.2.10 idp_engine.Run

The following Python functions can be used to perform computations using FO-dot knowledge bases:

model_check(**theories*)
Returns a string stating whether the combination of theories is satisfiable.

For example, `print(model_check(T, S))` will print `sat` if theory named `T` has a model expanding structure named `S`.

Parameters **theories** (*Union[TheoryBlock, Structure, Theory]*) – 1 or more (data) theories.

Returns `sat`, `unsat` or `unknown`

Return type str

model_expand(**theories*, *max=10*, *timeout_seconds=10*, *complete=False*, *extended=False*, *sort=False*)
Returns a (possibly empty) list of models of the combination of theories, followed by a string message.

For example, `print(model_expand(T, S))` will return (up to) 10 string representations of models of theory named `T` expanding structure named `S`.

The string message can be one of the following:

- `No models.`
- `More models may be available. Change the max argument to see them.`
- `More models may be available. Change the timeout_seconds argument to see them.`

- More models may be available. Change the max and timeout_seconds arguments to see them.

Parameters

- **theories** (*Union[TheoryBlock, Structure, Theory]*) – 1 or more (data) theories.
- **max** (*int, optional*) – max number of models. Defaults to 10.
- **timeout_seconds** (*int, optional*) – timeout_seconds in seconds. Defaults to 10.
- **complete** (*bool, optional*) – True to obtain complete structures. Defaults to False.
- **extended** (*bool, optional*) – use *True* when the truth value of inequalities and quantified formula is of interest (e.g. for the Interactive Consultant). Defaults to False.
- **sort** (*bool, optional*) – True if the models should be in alphabetical order. Defaults to False.

Yields str

Return type Iterator[str]

model_propagate(**theories*, *sort=False*)

Returns a list of assignments that are true in any model of the combination of theories.

Terms and symbols starting with '_' are ignored.

For example, `print(model_propagate(T, S))` will return the assignments that are true in any expansion of the structure named `S` consistent with the theory named `T`.

Parameters

- **theories** (*Union[TheoryBlock, Structure, Theory]*) – 1 or more (data) theories.
- **sort** (*bool, optional*) – True if the assignments should be in alphabetical order. Defaults to False.

Yields str

Return type Iterator[str]

maximize(**theories*, *term*)

Returns the list of assignments that are true in any model that maximizes *term*.

Parameters

- **theories** (*Union[TheoryBlock, Structure, Theory]*) – 1 or more (data) theories.
- **term** (*str*) – a string representing a term

Yields str

Return type Iterator[str]

minimize(**theories*, *term*)

Returns the list of assignments that are true in any model that minimizes *term*.

Parameters

- **theories** (*Union[TheoryBlock, Structure, Theory]*) – 1 or more (data) theories.
- **term** (*str*) – a string representing a term

Yields str

Return type Iterator[str]

decision_table(**theories*, *goal_string=''*, *timeout_seconds=20*, *max_rows=50*, *first_hit=True*, *verify=False*)
Experimental. Returns a decision table for *goal_string*, given the combination of theories.

Parameters

- **theories** (*Union[TheoryBlock, Structure, Theory]*) – 1 or more (data) theories.
- **goal_string** (*str, optional*) – the last column of the table. Must be a predicate application defined in the theory, e.g. `eligible()`.
- **timeout_seconds** (*int, optional*) – maximum duration in seconds. Defaults to 20.
- **max_rows** (*int, optional*) – maximum number of rows. Defaults to 50.
- **first_hit** (*bool, optional*) – requested hit-policy. Defaults to True.
- **verify** (*bool, optional*) – request verification of table completeness. Defaults to False

Yields a textual representation of each rule

Return type Iterator[str]

determine_relevance(**theories*)
Generates a list of questions that are relevant, or that can appear in a justification of a `goal_symbol`.

The questions are preceded with `` ? `` when their answer is unknown.

When an *irrelevant* value is changed in a model M of the theories, the resulting M' structure is still a model. Relevant questions are those that are not irrelevant.

If `goal_symbol` has an enumeration in the theory (e.g., `goal_symbol := {`tax_amount}.`), relevance is computed relative to those goals.

Definitions in the theory are ignored, unless they influence axioms in the theory or goals in `goal_symbol`.

Yields relevant questions

Parameters **theories** (*Union[idp_engine.Parse.TheoryBlock, idp_engine.Parse.Structure, idp_engine.Theory.Theory]*) –

Return type Iterator[str]

pretty_print(*x=''*)
Prints its argument on stdout, in a readable form.

Parameters **x** (*Any, optional*) – the result of an API call. Defaults to "".

Return type None

duration(*msg=''*)
Returns the processing time since the last call to *duration()*, or since the begining of execution

Parameters **msg** (*str*) –

Return type str

execute(*self*)
Execute the `main()` procedure block in the IDP program

Parameters **self** (idp_engine.Parse.IDP) –

Return type None

5.2.11 idp_engine.utils

Various utilities (in particular, OrderedSet)

class Semantics(*value*)
Semantics for inductive definitions

DEF_SEMANTICS = Semantics.WELLFOUNDED
String constants

NOT_SATISFIABLE = 'Not satisfiable.'
Module that monkey-patches json module when it's imported so JSONEncoder.default() automatically checks for a special "to_json()" method and uses it to encode the object if found.

exception IDPZ3Error
raised whenever an error occurs in the conversion from AST to Z3

class OrderedSet(*els=[]*)
a list of expressions without duplicates (first-in is selected)

__init__(*els=[]*)

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

5.3 idp_web_server module

5.3.1 idp_web_server.Inferences

This module contains the logic for inferences that are specific for the Interactive Consultant.

5.3.2 idp_web_server.IO

This module contains code to create and analyze messages to/from the web client.

metaJSON(*state*)
Format a response to meta request.

Parameters **idp** – the response

Returns out a meta request

load_json(*assignments*, *jsonstr*, *keep_defaults*)
Parse a json string and update assignments in a state accordingly.

Parameters

- **assignments** – an assignments containing the concepts that appear in the json
- **jsonstr** (*str*) – assignments in json
- **keep_defaults** (*bool*) – whether to not delete the default assignments
- **jsonstr** –
- **keep_defaults** –

Post assignments is updated with information in json

5.3.3 idp_web_server.rest

This module implements the IDP-Z3 web server

To profile it, set with_profiling to True

class HelloWorld

idpOf(*code*)
Function to retrieve an IDP object for IDP code. If the object doesn't exist yet, we create it. *idps* is a dict which contains an IDP object for each IDP code. This way, easy caching can be achieved.

Parameters **code** – the IDP code.

Returns **IDP** the IDP object.

class run
Class which handles the run. <<Explanation of what the run is here.>>

Parameters **Resource** – <<explanation of resource>>

post()
Method to run an IDP program with a procedure block.

:returns stdout.

class meta
Class which handles the meta. <<Explanation of what the meta is here.>>

Parameters **Resource** – <<explanation of resource>>

post()
Method to export the metaJSON from the resource.

Returns **metaJSON** a json string containing the meta.

class metaWithGraph

post()
Method to export the metaJSON from the resource.

Returns **metaJSON** a json string containing the meta.

class eval

class evalWithGraph

5.3.4 idp_web_server.State

Management of the State of problem solving with the Interactive Consultant.

class State(*idp*)
Contains a state of problem solving

Parameters **idp** (idp_engine.Parse.IDP) –

classmethod make(*idp*, *previous_active*, *active*, *ignore=None*)
Manage the cache of State

Parameters

- **idp** (IDP) – idp source code
- **previous_active** (*str*) – assignments due to previous full propagation

- **active** (*str*) – assignment choices from client
- **ignore** (*str*) – user-disabled laws to ignore

Returns a State

Return type State

__init__(*idp*)

Creates an instance of `Theory` for the list of theories, e.g., `Theory(T,S)`.

Parameters

- **theories** (*Union[TheoryBlock, Structure, Theory]*) – 1 or more (data) theories.
- **extended** (*bool, optional*) – use *True* when the truth value of inequalities and quantified formula is of interest (e.g. for the Interactive Consultant). Defaults to False.
- **idp** (idp_engine.Parse.IDP) –

add_given(*jsonstr*, *previous*, *keep_defaults=False*)

Add the assignments that the user gave through the interface. These are in the form of a json string.

Parameters

- **jsonstr** (*str*) – the user's assignment in json
- **previous** (*str*) – the assignments from the last propagation
- **keep_default** – whether default assignments should not be reset
- **jsonstr** –
- **previous** –
- **keep_defaults** (*bool*) –

Post the state has the jsonstr and previous added

CHAPTER

SIX

APPENDIX: SYNTAX SUMMARY

The following code illustrates the syntax of the various blocks used in IDP-Z3.

T denotes a type, c a constructor, p a proposition or predicate, f a constant or function. The equivalent ASCII-only encoding is shown on the right.

```
vocabulary V {
    type T
    type T := {c1, c2, c3}
    type T := constructed from {c1, c2(T1, f:T2)}
    type T := {1,2,3}
    type T := {1..3}
    // built-in types: , , , Date, Concept Bool, Int, Real, Date, Concept

    p : () →                                    p: () -> Bool
    p1, p2 : T1  T2 →                          p1, p2: T1*T2 -> Bool
    f: T → T                                     f: T -> T
    f1, f2: Concept[T1->T2] → T                  f1, f2: Concept[T1->T2] -> T

    [this is the intended meaning of p]
    p : () →

    import W
}

theory T:V {
    (¬p1()p2()  p3()  p4()  p5())  p6(). (~p1()&p2() | p3() => p4() <=> p5()) <= p6().
    p(f1(f2())).
    f1() < f2()  f3() = f4()  f5() > f6().  f1() < f2() =< f3() = f4() >= f5() > f6().
    f()  c.                                  f() ~= c.
    x,y  T: p(x,y).                         !x,y in T: p(x,y).
    x  p, (y,z)  q: q(x,x)  p(y)  p(z).  !x in p, (y,z) in q: q(x,x) | p(y) | p(z).
    x  Concept[()→B]: $(x)().                ?x in Concept[()->B]: $(x)().
    x: p(x).                                 ?x: p(x).


    f() in {1,2,3}.
    f() = #{xT: p(x)}.                       f() = #{x in T: p(x)}.
    f() = sum(lambda xT: f(x)).              f() = sum(lambda x in T: f(x)).
    if p1() then p2() else p3().
    f1() = if p() then f2() else f3().
```

(continues on next page)

(continued from previous page)

```
    p := {1,2,3}.
    p(#2020-01-01) is enumerated.
    p(#TODAY) is not enumerated.

    { p(1).
      xT: p1(x) ← p2(x).                    !x in T: p1(x) <- p2(x).
      f(1)=1.
      x: f(x)=1 ← p(x).                      !x: f(x)=1 <- p(x).
    }

    [this is the intended meaning of the rule]
    p().
}

structure S:V {
    p := false.
    p := {1,2,3}.
    p := {0..9, 100}.
    p := {#2021-01-01}.
    p := {(1,2), (3,4)}.
    p := {
    1 2
    3 4
    }.

    f := 1.
    f := {→1} .                                 f := {-> 1}.
    f := {1→1, 2→2}.                            f := {1->1, 2->2}.
    f := {(1,2)→3} else 2.                      f := {(1,2)->3} else 2.
    f : {(1,2)→3}.                             f :>= {(1,2)->3}.
}

display {
    goal_symbol := {`p1, `p2}.
    hide(`p).
    expand := {`p}.
    view() = expanded.
    optionalPropagation().
}

procedure main() {
    pretty_print(model_check    (T,S))
    pretty_print(model_expand   (T,S))
    pretty_print(model_propagate(T,S))
    pretty_print(minimize(T,S, term="cost()"))
}
```

See also the *Built-in functions*.

CHAPTER

SEVEN

INDEX

CHAPTER

EIGHT

INDICES AND TABLES

- *Index*
- search

PYTHON MODULE INDEX

i

INDEX

T

V