IDP-Z3

Pierre Carbonnelle

Dec 16, 2021

CONTENTS:

INTRODUCTION

IDP-Z3 is a software collection implementing the Knowledge Base paradigm using the FO() language. FO() 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`

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 web IDE 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 web IDE 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
```

THE FO() LANGUAGE

2.1 Overview

The FO() language is used to create knowledge bases. An IDP-Z3 program contains an FO() knowledge base and instructions to perform tasks on it. It is made of the following blocks of code:

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

theory specify the definitions and constraints satisfied by any solutions.

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

display (optional) configure the user interface of the *Interactive Consultant*.

main (optional) executable procedure in the context of the knowledge base

The basic skeleton of an IDP-Z3 program for the Interactive Consultant is as follows:

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

theory {
    // here comes the definitions and constraints
}

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

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

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 Web IDE 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`. The equivalent ASCII symbols are `Bool`, `Int`, and `Real`.

Boolean literals are `true` and `false`. Number literals follow Python's conventions. Date literals follow ISO 8601 conventions, prefixed with # (`#yyyy-mm-dd`). `#TODAY` is also a Date literal.

The type `Concept` has one constructor for each symbol (i.e., function, predicate or constant) declared in the vocabulary. 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 types `T1`, `T2`, `T3` and output type `T`, are declared by:

```
myFunc1, myFunc2 : T1  T2  T3 → T
```

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

The input and output types of myFunc1 can be sets of concepts with a specific type signature, indicated in brackets. For example, T1 can be *Concept[TT1 -> TT2]* to denote the concepts with type signature *TT1 -> TT2*.

IDP-Z3 does not support partial functions.

2.3.3 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;
- `arity:  Concept` → `Concept` yields the arity of a symbol;
- `input_domain :  Concept   ` → `Concept` yields the n-th input-domain of a symbol;
- `output_domain:  Concept` → `Concept` yields the output domain of a symbol.

2.3.4 Predicates

The predicates with name `myPred1`, `myPred2` and argument types `T1`, `T2`, `T3` are declared by:

```
myPred1, myPred2 : T1  T2  T3 →
```

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

The input and output types of myPred1 can be sets of concepts with a specific type signature, indicated in brackets. For example, T1 can be *Concept[TT1 -> TT2]* to denote the concepts with type signature *TT1 -> TT2*.

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

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

2.4 Theory

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

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

Before explaining their syntax, we need to introduce the concept of term.

2.4.1 Mathematical expressions and Terms

A *term* is inductively defined as follows:

Numeric literal Numeric literals that follow the Python conventions are numerical terms of type `Int` or `Real`.

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*, as defined in the function declaration in the vocabulary. The resulting type 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.

> `$(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 constraints

A *constraint* 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 .

When quantifying a formula of type Concept, the expression must contain a "guard" to prevent arity or type error. A guard is a condition that can be resolved using the available enumerations. In the following example, `symmetric` must be defined by enumeration.

```
symmetric := {`edge}
 s  Concept: symmetric(s) => (x, y : $(s)(x,y)  $(s)(y,x)).
```

An alternative is to use the introspection functions `arity`, `input_domain`, `output_domain`:

```
s   Concept: arity(s)=2   input_domain(s,1)=input_domain(s,2)
      (x   $(input_domain(s,1)), y   $(input_domain(s,2)) : $(s)(x,y)   $(s)(y,x)).
```

Another alternative is to add the signature of the Concept in the quantification:

```
x   Concept[()→B]: $(x)().
```

"is (not) enumerated" `f(a,b) is enumerated` and `f(a,b) is not enumerated` are sentences, 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.

"(not) in {1,2,3,4}" `f(args) in enum` and `f(args) not in enum` are sentences, 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}` or `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.

2.5 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, by enumeration. If the names are omitted, the structure is named S, for vocabulary V.

A structure is a set of statement of the form `<symbol> := <enumeration>`, e.g., `P := {1..9}`, where the enumeration can be:

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 }`

Additional notes:

- the enumeration for a predicate specifies the tuples that make the predicate true; any other tuple make it false.

- the enumeration for a function 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.

- parenthesis around a tuple can be omitted when the arity is 1, e.g., `{1-2, 3->4}`

- a predicate may be enumerated using a CSV format, with one tuple per line, e.g., :

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

2.6 Main block

The *main block* consists of python-like statements to be executed by the *IDP-Z3 executable* or the Web IDE, in the context of the knowledge base. It takes the following form:

```
procedure main() {
    // here comes the python-like code to be executed
}
```

Within that block, the vocabularies, theories and structures defined in other blocks of the IDP-Z3 program are available through variables of the same name.

The following functions are also available:

model_check(theory, structure=None) Returns string `sat`, `unsat` or `unknown`, depending on whether the theory has a model expanding the structure. `theory` and `structure` can be lists, in which case their elements are merged. The structure is optional.

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

model_expand(theory, structure=None, max=10, timeout=10, complete=False) Returns a (possibly empty) list of models of the theory that are expansion of the structure, possibly followed by a string message. `theory` and `structure` can be lists, in which case their elements are merged. The structure is optional. 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` (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 argument to see them.

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

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

model_propagate(theory, structure=None) Returns a list of assignments that are true in any expansion of the structure consistent with the theory. `theory` and `structure` can be lists, in which case their elements are merged. The structure is optional. Terms and symbols starting with '_' are ignored.

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

decision_table(theories, structures=None, goal_string="", timeout=20, max_rows=50, first_hit=True) Experimental. Returns the rows for a decision table that defines `goal_string`. `goal_string` must be a predicate application defined in the theory.

pretty_print(. . .) Prints its argument on stdout, in a readable form.

2.6.1 Theory class

The main block can also use instances of the `Theory` class. This is beneficial when several inferences must be made in a row (e.g., `Theory(T,S).propagate().simplify().formula()`). Instances of the `Theory` class represent a collection of theory and structure blocks. The class has the following methods:

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

Use *extended=True* when the truth value of inequalities and quantified formula is of interest (e.g. for the Interactive Consultant).

add(self, *blocks) Adds a list of theory or structure blocks to the theory.

assert_(self, code: str, value: Any) Asserts that an expression has a value, e.g. *theory.assert_("p()", True)*.

copy(self) Returns an independent copy of a theory.

formula(self) Returns a Z3 object representing the logic formula equivalent to the theory. This object can be converted to a string using `str()`.

expand(self, max=10, timeout=10, complete=False) Returns 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` (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).

optimize(self, term, minimize=True) Updates the Theory so that the value of term in the `assignments` property is the minimal (or maximal if `minimize` is `False`) value that is compatible with the Theory. `term` is a string (e.g. `"Length(1)"`).

symbolic_propagate(self) Returns the theory with its `assignments` property updated with direct consequences of the constraints of the theory. This propagation is less complete than `propagate()`.

propagate(self) Returns the theory with its `assignments` property updated with values for all terms and atoms that have the same value in every model (i.e., satisfying structure of the theory). Terms and propositions starting with '_' are ignored.

get_range(self, term:str) Returns a list of the possible values of the term (as strings).

explain(self, consequence=None) Returns the facts and laws to explain a consequence in the Theory.

> The string `consequence` must be a key in the `assignments` property of the Theory, or `None`. If it is `None`, and the theory is unsatisfiable, the method returns a minimal set of facts and laws that make it so. The facts are a list of Assignment, and the laws are a list of `Expression`.

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

decision_table(self, goal_string="", timeout=20, max_rows=50, first_hit=True) 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`.

duration(msg: str) Returns the processing time since the last call to *duration()*, or since the begining of execution, followed by the msg (possibly an f-string)

2.7 Differences with IDP3

Here are the main differences with IDP3, listed for migration purposes:

min/max aggregates IDP-Z3 does not support these aggregates (yet). See IEP 05

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

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

LTC IDP-Z3 does not support LTC vocabularies.

Namespaces IDP-Z3 does not support namespaces.

Partial functions IDP-Z3 does not support partial functions. The handling of division by 0 may differ. See IEP 07

Programming API IDP3 procedures are written in Lua, IDP-Z3 procedures are written in Python-like language.

Qualified quantifications IDP-Z3 does not support qualified quantifications, e.g. `!2 x[color]: p(x)`.. (p. 11 of the IDP3 manual).

Structure IDP-Z3 does not support `u` uncertain interpretations (p.17 of IDP3 manual). Function enumerations must have an `else` part. (see also IEP 04)

Type IDP-Z3 does not support type hierarchies.

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

2.8 Syntax summary

The following code illustrates the syntax of 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 : () →

    extern vocabulary 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  Concept: arity(x)=0  $(x)().         ?x in Concept: arity(x)=0 & $(x)().
    x  $(input_domain(`p,1)): p(x).          ?x in $(input_domain(`p,1)): p(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().

    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
}

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

procedure main() {
    pretty_print(model_check    (T,S))
    pretty_print(model_expand   (T,S))
    pretty_print(model_propagate(T,S))
}
```

See also the *Built-in functions*.

PYTHON API

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

```python
from idp_engine import IDP, model_expand
kb = IDP.from_file("path/to/file.idp")
T, S = kb.get_blocks("T, S")
for model in model_expand(T,S):
    print(model)
```

Besides the methods and class available in the *main block*, idp_engine exposes the IDP class, described below.

3.1 IDP class

The IDP class exposes the following methods:

from_file(file_path: str) This class method parses the *IDP-Z3 program* in the file located at file_path.

from_str(code: str) This class method parses the *IDP-Z3 program* in the code string.

parse(file_or_string) DEPRECATED: This class method parses the *IDP-Z3 program* in the file or string.

get_blocks(names) This instance method returns the list of blocks whose names are given in a comma-separated string.

execute() This instance methods executes the *main()* procedure block in the IDP-Z3 program.

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 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] [-o OUTPUT] [--full-formula] [--no-timing] FILE

IDP-Z3

positional arguments:
  FILE                  path to the .idp file

  optional arguments:
    -h, --help          show this help message and exit
    -o OUTPUT, --output OUTPUT
    name of the output file
    --full-formula      show the full formula
    --no-timing         don't display timing information
```

FIVE

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. However, there are some specific changes and additions when using IDP-Z3 in the Interactive Consultant, which are explained further in this chapter.

5.1 Display

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.

relevant `relevant(`s1, .., `sn)` specifies that symbols `s1, .., sn` are relevant, i.e. that they should never be greyed out.

> Irrelevant symbols and sub-sentences, i.e. symbols whose interpretation do not constrain the interpretation of the relevant symbols, are greyed out by the Interactive Consultant.

goal `goal(`s)` specifies that symbol `s` is a goal, i.e. that it is relevant and shown expanded.

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*.

5.2 Vocabulary annotations

To improve the display of functions and predicates in the *Interactive Consultant*, they 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.

5.3 Environment

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 {
```

```
    extern vocabulary environment
    // here comes the specification of the vocabulary to describe the decisions and␣
↪their consequences
}

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

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

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

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

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

5.4 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
}
```

APPENDIX: IDP-Z3 DEVELOPER REFERENCE

Note: The contents of this reference are intended for people who want to further develop IDP-Z3.

Note: Despite our best efforts, this documentation may not be complete and up-to-date.

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.

6.1 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.

6.1.1 Web clients

The repository for the web clients is in a separate GitLab repository.

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_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.

6.1.2 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.

6.1.3 IDP-Z3 server

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

The IDP-Z3 server is written in python 3.8, using the Flask framework. Pages are served by `/idp_server/rest.py`. Static files are served from the `/idp_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($\cdot$) 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($\cdot$) 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 same as the above, sans the last added assignment.
- **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.

6.1.4 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()`)

The graph of calls is outlined in `/docs/zettlr/Call graph.md`.

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.

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()`).

6.1.5 Z3

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

You may also want to refer to the Z3py reference.

6.1.6 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

6.2 `idp_engine` **module**

6.2.1 **idp_engine.Parse**

Classes to parse an IDP-Z3 theory.

class idp_engine.Parse.**IDP**(***kwargs*)
    Bases: *[idp_engine.Expression.ASTNode](#)*

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

> **Parameters**
>
> - **code** (*str*) – source code of the IDP program
>
> - **vocabularies** (*dict[str,* `Vocabulary`*]*) – list of vocabulary blocks, by name
>
> - **theories** (*dict[str,* `TheoryBlock`*]*) – list of theory blocks, by name
>
> - **structures** (*dict[str,* `Structure`*]*) – list of structure blocks, by name
>
> - **procedures** (*dict[str,* `Procedure`*]*) – list of procedure blocks, by name
>
> - **display** (`Display`*, Optional*) – display block, if any

    classmethod from_file(*file: str*) → *[idp_engine.Parse.IDP](#)*
        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: str*) → *[idp_engine.Parse.IDP](#)*
        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: str*) → *[idp_engine.Parse.IDP](#)*
        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: List[str]*)
        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 IDP program

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

>   The class of AST nodes representing a vocabulary block.

>   **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 idp_engine.Parse.**Annotations**(***kwargs*)
>   Bases: *idp_engine.Expression.ASTNode*

class idp_engine.Parse.**Extern**(***kwargs*)
>   Bases: *idp_engine.Expression.ASTNode*

class idp_engine.Parse.**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*
>   - **constructors** (*[Constructor]*) – list of constructors in the enumeration
>   - **range** (*[Expression]*) – list of expressions of that type
>   - **interpretation** (*SymbolInterpretation*) – the symbol interpretation
>   - **map** (*Dict[string, Expression]*) – a mapping from code to Expression in range

class idp_engine.Parse.**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

in_domain
: the list of possible tuples of arguments

    Type List

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

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

Represents a Symbol. Handles synonyms.

name
: name of the symbol

    Type string

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

The class of AST nodes representing a theory block.

class idp_engine.Parse.**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

set_level_symbols()
    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(*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

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

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 idp_engine.Parse.**Structure**(***kwargs*)

Bases: *idp_engine.Expression.ASTNode*

The class of AST nodes representing an structure block.

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 idp_engine.Parse.**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

contains(*args*, *function*, *arity=None*, *rank=0*, *tuples=None*)

returns an Expression that says whether Tuple args is in the enumeration

class idp_engine.Parse.**Tuple**(***kwargs*)

Bases: *idp_engine.Expression.ASTNode*

class idp_engine.Parse.**Display**(***kwargs*)

Bases: *idp_engine.Expression.ASTNode*

class idp_engine.Parse.**Procedure**(***kwargs*)

Bases: *idp_engine.Expression.ASTNode*

6.2.2 **idp_engine.Expression**

(They are monkey-patched by other modules)

class idp_engine.Expression.**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 idp_engine.Expression.**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

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, IfExpr, 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: dict*)
>   fills the list *constructors* with all constructors belonging to open types.

co_constraints(*co_constraints*)
>   collects the constraints attached to AST nodes, e.g. instantiated definitions
>
>   *co_constraints* is an OrderedSet of Expression

is_assignment() → bool

>   **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:* [idp_engine.Assignments.Assignments](#)) → [*idp_engine.Expression.Expression*](#)
>   simplify the expression using the assignments

symbolic_propagate(*assignments: Assignments*, *tag: Status*, *truth: Optional[*[idp_engine.Expression.Expression](#)*] = 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.

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

translate(*problem: Theory*, *vars={}*) → z3.z3.ExprRef
>   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() → Tuple[Optional[*AppliedSymbol*], Optional[bool], Optional[Enumeration]]
  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*) → *idp_engine.Expression.Expression*
  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 idp_engine.Expression.**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*

class idp_engine.Expression.**IfExpr**(***kwargs*)
>   Bases: *idp_engine.Expression.Expression*

>   **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}

>   **translate1**(*problem: Theory*, *vars={}*) → z3.z3.ExprRef
>   >   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

class idp_engine.Expression.**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*]

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

> > **Type** Domain, 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`

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

> **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, IfExpr, 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]

> **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*

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

    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, IfExpr, 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}

class idp_engine.Expression.**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 recur-
               sive symbols

            • **head** (–) – head of the rule we are adding level mapping symbols to.

            • **pos_justification** (–) – whether we are adding symbols to the direct positive jus-
               tification (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

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

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

        Returns Expression

class idp_engine.Expression.**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

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

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

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

> **is_assignment**()
> > Returns: bool: True if *self* assigns a rigid term to a rigid function application

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

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

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

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

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

> **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, IfExpr, 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}

class idp_engine.Expression.**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

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, IfExpr, 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: dict*)
> fills the list *constructors* with all constructors belonging to open types.

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

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

class idp_engine.Expression.**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().

> **classmethod construct**(*constructor:* idp_engine.Expression.Constructor)
>> Create an UnappliedSymbol from a constructor

class idp_engine.Expression.**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

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

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

class idp_engine.Expression.**Number**(***kwargs*)
> Bases: *idp_engine.Expression.Expression*

> **real**()
>> converts the INT number to REAL

class idp_engine.Expression.**Brackets**(***kwargs*)
> Bases: *idp_engine.Expression.Expression*

6.2.3 idp_engine.Annotate

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

idp_engine.Annotate.**get_instantiables**(*self*, *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

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

6.2.4 idp_engine.Interpret

Methods to interpret a theory in a data structure

- substitute a constant by its value in an expression
- replace symbols interpreted in a structure by their interpretation
- expand quantifiers

This module also includes methods to:

- substitute an 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)

idp_engine.Interpret.**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

6.2.5 idp_engine.Simplify

Methods to simplify a logic expression.

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

idp_engine.Simplify.**join_set_conditions**(*assignments: List[*idp_engine.Assignments.Assignment*]*)
→ List[*idp_engine.Assignments.Assignment*]
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*]

6.2.6 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.

idp_engine.Propagate.**simplify_with**(*self:* idp_engine.Expression.Expression, *assignments:* idp_engine.Assignments.Assignments) →
idp_engine.Expression.Expression
simplify the expression using the assignments

6.2.7 idp_engine.idp_to_Z3

Translates AST tree to Z3

TODO: vocabulary

6.2.8 idp_engine.Problem

Class to represent a collection of theory and structure blocks.

class idp_engine.Problem.**Propagation**(*value*)
Describe propagation method

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

> **extended**
> True when the truth value of inequalities and quantified formula is of interest (e.g. in the Interactive Consultant)
>
> > **Type** Bool
>
> **declarations**
> the list of type and symbol declarations
>
> > **Type** dict[str, Type]

constraints
> a set of assertions.
>
>> **Type** *OrderedSet*

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
>
>> **Type** *Assignment*

definitions
> a list of definitions in this problem
>
>> **Type** [*Definition*]

def_constraints
> A mapping of defined symbol to the whole-domain constraints equivalent to its definition.
>
>> **Type** dict[*SymbolDeclaration*, *Definition*], list[*Expression*]

interpretations
> A mapping of enumerated symbols to their interpretation.
>
>> **Type** dict[string, SymbolInterpretation]

goals
> A set of goal symbols
>
>> **Type** dict[string, *SymbolDeclaration*]

_constraintz
> a list of assertions, co_constraints and definitions in Z3 form
>
>> **Type** List(ExprRef), Optional

_formula
> the Z3 formula that represents the problem (assertions, co_constraints, definitions and assignments).
>
>> **Type** ExprRef, optional

co_constraints
> the set of co_constraints in the problem.
>
>> **Type** *OrderedSet*

propagated
> true if a propagation has been done
>
>> **Type** Bool

assigned
> set of questions asserted since last propagate
>
>> **Type** *OrderedSet*

cleared
> set of questions unassigned since last propagate
>
>> **Type** *OrderedSet*

propagate_success
> whether the last propagate call failed or not
>
>> **Type** Bool

z3
    mapping from string of the code to Z3 expression, to avoid recomputing it

        Type   dict[str, ExprRef]

ctx
    Z3 context

classmethod make(*theories*, *structures*, *extended=False*)
    polymorphic creation

assert_(*code: str*, *value: Any*, *status: idp_engine.Assignments.Status = <Status.GIVEN: 8>*)
    asserts that an expression has a value (or not)

        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

constraintz()
    list of constraints, co_constraints and definitions in Z3 form

formula()
    the formula encoding the knowledge base

expand(*max=10*, *timeout=10*, *complete=False*)
    output: a list of Assignments, ending with a string

symbolic_propagate(*tag=<Status.UNIVERSAL: 3>*)
    determine the immediate consequences of the constraints

propagate(*tag=<Status.CONSEQUENCE: 4>*, *method=<Propagation.DEFAULT: 1>*)
    determine all the consequences of the constraints

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

explain(*consequence=None*)
    Pre: the problem is UNSAT (under the negation of the consequence if not None)

    Returns the facts and laws that make the problem UNSAT.

        Parameters

            • **self** (*Theory*) – the problem state

            • **consequence** (*string | None*) – the code of the sentence to be explained. Must be
              a key in self.assignments

        Returns   list of facts and laws that explain the consequence

        Return type   (facts, laws) (List[*Assignment*], List[*Expression*])]

simplify()
    returns a simpler copy of the Theory, using known assignments

    Assignments obtained by propagation become fixed constraints.

decision_table(*goal_string=''*, *timeout=20*, *max_rows=50*, *first_hit=True*, *verify=False*)
    returns a decision table for *goal_string*, given *self*.

        Parameters

            • **goal_string** (*str, optional*) – the last column of the table.

- **timeout** (*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

> **Return type** list(list(*Assignment*))

6.2.9 idp_engine.Assignments

Classes to store assignments of values to questions

class idp_engine.Assignments.**Status**(*value*)
> Describes how the value of a question was obtained

class idp_engine.Assignments.**Assignment**(*sentence:* idp_engine.Expression.Expression,
> *value: Optional[*idp_engine.Expression.Expression*],*
> *status: Optional[*idp_engine.Assignments.Status*],*
> *relevant: Optional[bool] = 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.

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

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.

same_as (*other:* idp_engine.Assignments.Assignment) → bool
> 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

class idp_engine.Assignments.**Assignments**(**arg*, ***kw*)
> Contains a set of Assignment
>
> **copy**() → a shallow copy of D

6.2.10 idp_engine.Run

Classes to execute the main block of an IDP program

idp_engine.Run.**model_check**(*theories*, *structures=None*)
> output: "sat", "unsat" or "unknown"

idp_engine.Run.**model_expand**(*theories*, *structures=None*, *max=10*, *timeout=10*, *complete=False*, *extended=False*, *sort=False*)
> output: a list of Assignments, ending with a string

idp_engine.Run.**model_propagate**(*theories*, *structures=None*, *sort=False*)
> output: a list of Assignment

idp_engine.Run.**decision_table**(*theories*, *structures=None*, *goal_string=''*, *timeout=20*, *max_rows=50*, *first_hit=True*, *verify=False*)
> returns a decision table for *goal_string*, given *theories* and *structures*.
>
> > **Parameters**
> >
> > - **goal_string** (*str, optional*) – the last column of the table.
> >
> > - **timeout** (*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** *str* – a textual representation of each rule

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

idp_engine.Run.**execute**(*self*)
>   Execute the IDP program

6.2.11 **idp_engine.utils**

Various utilities (in particular, OrderedSet)

class idp_engine.utils.**Semantics**(*value*)
>   Semantics for inductive definitions

idp_engine.utils.**DEF_SEMANTICS = <Semantics.WELLFOUNDED: 3>**
>   String constants

idp_engine.utils.**DEFAULT = 'default'**
>   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 idp_engine.utils.**IDPZ3Error**
>   raised whenever an error occurs in the conversion from AST to Z3

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

6.3 **idp_server module**

6.3.1 **idp_server.Inferences**

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

idp_server.Inferences.**split_constraints**(*constraints:* idp_engine.utils.OrderedSet) → *idp_engine.utils.OrderedSet*
>   replace [.., a b, ..] by [.., a, b, ..]
>
>   This is to avoid dependencies between a and b (see issue #95).
>
>   >   **Parameters constraints** (OrderedSet) – set of constraints that may contain conjunctions
>   >
>   >   **Returns** set of constraints without top-level conjunctions
>   >
>   >   **Return type** *OrderedSet*

idp_server.Inferences.**get_relevant_questions**(*self: State*)
>   sets 'relevant in self.assignments sets rank of symbols in self.relevant_symbols removes irrelevant constraints in self.constraints

6.3.2 idp_server.IO

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

`idp_server.IO.`**`metaJSON`**(*state*)

> Format a response to meta request.

> > **Parameters idp** – the response

> > **Returns out** a meta request

`idp_server.IO.`**`load_json`**(*state:* idp_engine.Problem.Theory, *jsonstr: str*)

> Parse a json string and update assignments in a state accordingly.

> > **Parameters**

> > > • **state** – a Theory object containing the concepts that appear in the json

> > > • **jsonstr** – the user's assignments in json

> > **Returns** the assignments

> > **Return type** idp_engine.Assignments

6.3.3 idp_server.rest

This module implements the IDP-Z3 web server

To profile it, set with_profiling to True

class `idp_server.rest.`**`HelloWorld`**

`idp_server.rest.`**`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 `idp_server.rest.`**`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 `idp_server.rest.`**`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 `idp_server.rest.`**`metaWithGraph`**

> **`post`**()

> > Method to export the metaJSON from the resource.

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

class idp_server.rest.**eval**

class idp_server.rest.**evalWithGraph**

6.3.4 idp_server.State

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

class idp_server.State.**State**(*idp:* idp_engine.Parse.IDP)

> Contains a state of problem solving

> **classmethod make**(*idp:* idp_engine.Parse.IDP, *previous_active:* *str*, *jsonstr:* *str*) → *idp_server.State.State*

>> Manage the cache of State

>> **Parameters**

>>> • **idp** (IDP) – idp source code

>>> • **previous_active** (*str*) – previous input from client

>>> • **jsonstr** (*str*) – input from client

>> **Returns** a State

>> **Return type** *State*

> **add_given**(*jsonstr: str*)

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

>> **Parameters** **jsonstr** – the user's assignment in json

>> **Returns** the state with the jsonstr added

>> **Return type** *State*

SEVEN

INDEX

EIGHT

INDICES AND TABLES

- *Index*

- search

PYTHON MODULE INDEX