
IDP-Z3

Jan 04, 2021

Contents:

1	Introduction	1
1.1	Installation using poetry	1
1.2	Installation using pip	2
1.3	Installation of idp_solver module	3
2	The IDP Language	5
2.1	Overview	5
2.2	Shebang	6
2.3	Vocabulary	6
2.4	Theory	7
2.5	Structure	9
2.6	Main block	9
2.7	Differences with IDP3	11
2.8	Syntax summary	11
3	Command Line Interface	13
4	Interactive Consultant	15
4.1	Display	15
4.2	Vocabulary annotations	16
4.3	Environment	16
4.4	Default Structure	17
5	Appendix: IDP-Z3 developer reference	19
5.1	Architecture	19
5.2	idp_solver module	22
5.3	idp_server module	33
6	Index	37
7	Indices and tables	39
	Python Module Index	41
	Index	43

IDP-Z3 is a collection of software components implementing the Knowledge Base paradigm using the IDP language and a Z3 SMT solver.

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 components together enable 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_solver module

The `idp_solver` module is available for installation through the official Python package repository. This comes with a command line program, `idp_solver` 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_solver
poetry install
```

Installing the module via pip can be done as such:

```
pip3 install idp_solver
```


2.1 Overview

The IDP language is used to create knowledge bases. An IDP program 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 knowledge base 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 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 an alphabetic character or `_`, followed by alphanumeric characters or `_`. Symbols can also be string literals delimited by `'`, e.g., `'blue planet'`.

2.3.1 Types

IDP-Z3 has the following built-in types: `bool`, `int`, `real`, ``Symbols`.

Custom types can be defined by specifying a range of numeric literals, or a list of constructors (of arity 0).

```
type side = {1..4}  
type color constructed from {red, blue, green}
```

The type ``Symbols` 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 ```. For the above example, the constructors of ``Symbols` are: ``red`, ``blue`, ``green`.

2.3.2 Functions

A function with name `MyFunc`, input types `T1`, `T2`, `T3` and output type `T`, is declared by:

```
MyFunc(T1, T2, T3) : T
```

IDP-Z3 does not support partial functions.

2.3.3 Predicates

A predicate with name `MyPred` and argument types `T1`, `T2`, `T3` is declared by:

```
MyPred(T1, T2, T3)
```

2.3.4 Propositions and Constants

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

```
MyProposition
MyConstant: int
```

2.3.5 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.

Constant a *constant* is a term whose *type* is derived from its declaration in the *vocabulary*.

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

Function application $F(t_1, t_2, \dots, t_n)$ is a term, when F is a *function* symbol of arity n , and t_1, t_2, \dots, 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.

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 is one of the following math operators $+, -, *, /, ^, \%$. 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 $\#\{v_1[\text{typeOf}V_1]..v_n[\text{typeOf}V_n] : \phi\}$ is a numerical term when $v_1 v_2 .. v_n$ are variables, and ϕ 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 ϕ true.

Arithmetic aggregate $\{v_1[typeOfV_1]..v_n[typeOfV_n] : \phi : t\}$ is a numerical term when ϕ is a *sentence*, and t is a term.

The term denotes the sum of t for each distinct tuple of values for $v_1v_2..v_n$ which make ϕ true.

(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 $P(t_1, t_2, \dots, t_n)$ is a sentence, when P is a *predicate* symbol of arity n , and t_1, t_2, \dots, 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 and $P()$ are sentences.

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

Negation $\neg \phi$ is a sentence (or, using ascii characters: `~ \phi`) when ϕ is a sentence.

Logic connectives $\phi_1 \phi_2$ is a sentence when ϕ_1, ϕ_2 are two sentences and ϕ is one of the following logic connectives `\vee, \wedge, \Rightarrow, \Leftarrow, \Leftrightarrow` (or using ascii characters: `|, &, ==>, <==, <==>` respectively). Logic connectives can be chained as customary.

Parenthesis (ϕ) is a sentence when ϕ 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 these two forms, where v_1, \dots, v_n are variables and ϕ is a sentence:

$$\begin{aligned} \exists v_1[typeOfV_1]..v_n[typeOfV_n] : \phi \\ \forall v_1[typeOfV_1]..v_n[typeOfV_n] : \phi \end{aligned}$$

Alternatively, ascii characters can be used: `?, !`, respectively. For example, `!x[int] y[int]: f(x, y)=f(y, x)`. A variable may only occur in the ϕ sentence of a quantifier declaring that variable.

When quantifying a formula of type ``Symbols`, the expression must contain a “guard” to prevent arity or type error. A guard is a predicate over ``Symbols` that is defined by an enumeration in the same theory block. In the following example, `symmetric` must be defined by enumeration.

```
!`p[`Symbols]: symmetric(`p) => (!x y : `p(x, y) => `p(y, 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.

“in {1,2,3,4}” $f(args)$ in *enumeration* is a sentence, where f is a function applied to arguments $args$ and *enumeration* is an enumeration.

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. *predicates* or *functions*, in terms of other concepts. A definition consists of a set of rules, enclosed by `{` and `}`.

Rules have one of the following forms:

$$\begin{aligned} \forall v_1[typeOfV_1]..v_n[typeOfV_n] : P(t_1, .., t_n) &\leftarrow \phi. \\ \forall v_1[typeOfV_1]..v_n[typeOfV_n] : F(t_1, .., t_n) &= t \leftarrow \phi. \end{aligned}$$

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 ϕ is a formula that may contain these variables. $P(t_1, t_2, .., t_n)$ is called the *head* of the rule and ϕ the *body*. \leftarrow can be used instead of ' \leftarrow '. 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 *predicates* and *functions*, by enumeration. If the names are omitted, the structure is named `S`, for vocabulary `V`.

A structure is a set of enumerations, having one of the following forms:

$$\begin{aligned} P &= \{ el_1^1, el_1^2, .. el_1^n; \\ &\quad el_2^1, el_2^2, .. el_2^n; \\ &\quad .. \\ &\quad \} \\ F &= \{ el_1^1, el_1^2, .. el_1^n, el_1; \\ &\quad el_2^1, el_2^2, .. el_2^n, el_2; \\ &\quad .. \\ &\quad \} \text{ else } el \\ Z &= el. \end{aligned}$$

where P is a predicate of arity n , F is a function of arity n , and el_i^j are *constructors* or numeric literals.

The first statement enumerates the tuples of terms that make the predicate P true.

The second statement specifies the value el_i^n for the function F applied to the tuple of el_i^j arguments. The element after *else* specifies the function value for the non-enumerated tuples of arguments. This default value is optional; when omitted, the value of the function for the non-enumerated tuples, if any, is unspecified.

The third statement assigns the value el to the symbol Z (of arity 0).

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
}
```

The vocabularies, theories and structures defined in other blocks of the IDP program are available through variables of the same name.

The following functions are 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, complete=False) Returns a list of models of the theory that are expansion of the structure. `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 models can be asked to be complete or partial (i.e., in which “don’t care” terms are not specified).

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.

print(...) Prints the arguments on stdout

2.6.1 Problem class

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

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

add(self, block) Adds a theory or structure block to the problem.

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

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

expand(self, max=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 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, complete=False) Returns the problem with its `assignments` property updated with values such that the term is minimized (or maximized if `minimize` is `False`) `term` is a string (e.g. `"Length(1)"`). The models can be asked to be complete or partial (i.e., in which “don’t care” terms are not specified).

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

propagate(self) Returns the problem 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 problem). Terms and propositions starting with ‘_’ are ignored.

simplify(self) Returns the problem with a simplified formula of the problem, by substituting terms and atoms by their values specified in a structure or obtained by propagation.

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.

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](#)

Inductive definitions IDP-Z3 does not support inductive definitions.

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.

N-ary constructors IDP-Z3 does not support n-ary constructors, e.g., `RGB(int, int, int)`. See [IEP 06](#)

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 Type enumerations must be done in the vocabulary block (not in the structure block). 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. `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 constructed from {c1, c2}
  type T = {1;2;3}
  type T = {1..3}
  // built-in types: bool, int, real, `Symbols

  p
  p(T)
  f(T):T

  [this is the intended meaning of p]
  p

  extern vocabulary W
```

(continues on next page)

(continued from previous page)

```

}

theory T:V {
  (¬p1p2 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[T]: p(x).                          !x[T]: p(x).
  x: p(x).                             ?x: p(x).

  f() in {1;2;3}.
  f() = #{x[T]: p(x)}.
  f() = sum{x[T]: p(x): f(x)}.
  if p1 then p2 else p3.
  f1() = if p then f2() else f3().

  p = {1;2;3}
  p(1) is enumerated.
  p(5) is not enumerated.

  { p(1).
    x: p1(x) <- p2(x).                !x: 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 = {1;2;3}
  f = {1,1; 2,2}
  f = {1,1} else 2
  f=1.
}

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

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

```

Command Line Interface

IDP-Z3 can be run through a Command Line Interface, using poetry (see [Installation](#)):

```
poetry run python3 IDP-Z3.py path/to/file.idp
```

where `path/to/file.idp` is a relative path to the file containing the IDP program to be run. This file must contain a [main block](#).

Alternatively, you can run it using pip-installed packages.

```
python3 IDP-Z3.py path/to/file.idp
```

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.

4.1 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*(s_1, \dots, s_n) specifies that *symbols* s_1, \dots, s_n 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*(s_1, \dots, s_n) specifies that symbols s_1, \dots, s_n 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*(s_1, \dots, s_n) specifies that symbols s_1, \dots, s_n 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 symbols 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 will be available in the interface which allows toggling immediate propagation on and off.

By default, this button is not present.

4.2 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.

4.3 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 IDP 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 knowledge base 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
}
```

(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  
}
```

4.4 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 solver: performs inferences 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 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](#).

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_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.

5.1.4 IDP-Z3 solver

The code for the IDP-Z3 solver and IDP-Z3-CLI is in the `/idp_solver` folder. The IDP-Z3 solver 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. (`expand_quantifiers()`)
3. when a structure is given:
 1. expand quantifiers based on the structure (grounding); perform type inference as necessary (`expand_quantifiers()`)
 2. simplify the theory using the data in the structure and the laws of logic (by `interpret()` and `update_exprs()`)
 3. instantiate the definitions for every calls of the defined symbols (recursively) (`interpret()`)
 4. convert to Z3, adding the type constraints not enforced by Z3 (`.translate()`)

The code is organised by steps, not by classes: for example, all methods to substitute an expression by another are grouped in `Substitute.py`. We use [monkey-patching](#) to attach methods to the classes declared in another module.

Important classes of the IDP-Z3 solver are: `Expression`, `Assignment`, `Problem`.

`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()`).

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_solver module

5.2.1 idp_solver.Assignments

Classes to store assignments of values to questions

class idp_solver.Assignments.Status
Describes how the value of a question was obtained

class idp_solver.Assignments.Assignment (*sentence: idp_solver.Expression.Expression, value: Optional[idp_solver.Expression.Expression], status: Optional[idp_solver.Assignments.Status], relevant: Optional[bool] = False*)

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.

same_as (*other*: *idp_solver.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]]

class *idp_solver.Assignments.Assignments* (*arg, **kw)

Contains a set of Assignment

copy () → a shallow copy of D

5.2.2 idp_solver.Expression

(They are monkey-patched by other modules)

class *idp_solver.Expression.Expression*

Bases: object

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 source code.

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

Type Dict

original

The original expression, before transformation.

Type *Expression*

fresh_vars

The set of names of the variables in the expression.

Type Set(string)

copy()

create a deep copy (except for Constructor and NumberConstant)

annotate (*voc*, *q_vars*)

annotate tree after parsing

annotate1 ()

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

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 Constructor, IfExpr, AUnary, Fresh_Variable, Number_constant, Brackets

unknown_symbols (*co_constraints=True*)
 returns the list of symbol declarations in self, ignoring type constraints
 returns Dict[name, Declaration]

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

as_rigid ()
 returns a NumberConstant or Constructor, or None

substitute (*e0*, *e1*, *assignments*, *todo=None*)
 recursively substitute *e0* by *e1* in self (*e0* is not a Fresh_Variable)
 implementation for everything but AppliedSymbol, Variable and Fresh_variable

instantiate (*e0*, *e1*, *theory*)
 recursively substitute Fresh_Variable *e0* by *e1* in self
 instantiating *e0*=*x* by *e1*=*f* in self=*x*(*y*) returns *f*(*y*) (or any instance of *f* if arities don't match)

interpret (*theory*)
 for every defined term in self, add the instantiated definition as co-constraint
 implementation for everything but AppliedSymbol, Variable and Fresh_variable

expand_quantifiers (*theory*)
 replaces quantified formula by its expansion
 implementation for everything but AQuantification and AAgregate

symbolic_propagate (*assignments*: *idp_solver.Assignments.Assignments*,
truth: *Optional[idp_solver.Expression.Constructor]* =
true) \rightarrow *List[Tuple[idp_solver.Expression.Expression,*
idp_solver.Expression.Constructor]]
 returns the consequences of *self=truth* that are in assignments.
 The consequences are obtained by symbolic processing (no calls to Z3).

Parameters

- **assignments** (*Assignments*) – The set of questions to chose from. Their value is ignored.
- **truth** (*Constructor*, *optional*) – The truth value of the expression *self*. Defaults to TRUE.

Returns A list of pairs (Expression, bool), descing the literals that are implicant

propagate1 (*assignments*, *truth*)
 returns the list of symbolic_propagate of self (default implementation)

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]]

class `idp_solver.Expression.Constructor` (**kwargs)

Bases: `idp_solver.Expression.Expression`

as_rigid()

returns a NumberConstant or Constructor, or None

update_exprs (*new_exprs*)

change sub_exprs and simplify, while keeping relevant info.

class `idp_solver.Expression.IfExpr` (**kwargs)

Bases: `idp_solver.Expression.Expression`

annotatel()

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

class `idp_solver.Expression.AQuantification` (**kwargs)

Bases: `idp_solver.Expression.Expression`

classmethod **make** (*q*, *q_vars*, *f*)

make and annotate a quantified formula

annotate (*voc*, *q_vars*)

annotate tree after parsing

annotatel()

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

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 Constructor, IfExpr, AUnary, Fresh_Variable, Number_constant, Brackets

class `idp_solver.Expression.BinaryOperator` (**kwargs)

Bases: `idp_solver.Expression.Expression`

classmethod **make** (*ops*, *operands*)

creates a BinaryOp beware: cls must be specific for ops !

annotatel()

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

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 Constructor, IfExpr, AUnary, Fresh_Variable, Number_constant, Brackets

```

class idp_solver.Expression.AImplication(**kwargs)
    Bases: idp_solver.Expression.BinaryOperator

class idp_solver.Expression.AEquivalence(**kwargs)
    Bases: idp_solver.Expression.BinaryOperator

class idp_solver.Expression.ARImplication(**kwargs)
    Bases: idp_solver.Expression.BinaryOperator

    annotate(voc, q_vars)
        annotate tree after parsing

class idp_solver.Expression.ADisjunction(**kwargs)
    Bases: idp_solver.Expression.BinaryOperator

class idp_solver.Expression.AConjunction(**kwargs)
    Bases: idp_solver.Expression.BinaryOperator

class idp_solver.Expression.AComparison(**kwargs)
    Bases: idp_solver.Expression.BinaryOperator

    annotate(voc, q_vars)
        annotate tree after parsing

    annotatel1()
        annotations that are common to __init__ and make()

class idp_solver.Expression.ASumMinus(**kwargs)
    Bases: idp_solver.Expression.BinaryOperator

class idp_solver.Expression.AMultDiv(**kwargs)
    Bases: idp_solver.Expression.BinaryOperator

class idp_solver.Expression.APower(**kwargs)
    Bases: idp_solver.Expression.BinaryOperator

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

    annotatel1()
        annotations that are common to __init__ and make()

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

    annotate(voc, q_vars)
        annotate tree after parsing

    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 Constructor, IfExpr, AUnary, Fresh_Variable, Number_constant, Brackets

class idp_solver.Expression.AppliedSymbol(**kwargs)
    Bases: idp_solver.Expression.Expression

    annotate(voc, q_vars)
        annotate tree after parsing

```

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

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 Constructor, IfExpr, AUnary, Fresh_Variable, Number_constant, Brackets

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

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

class idp_solver.Expression.Variable (**kwargs)
    Bases: idp_solver.Expression.AppliedSymbol

annotate (voc, q_vars)
    annotate tree after parsing

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 Constructor, IfExpr, AUnary, Fresh_Variable, Number_constant, Brackets

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

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

class idp_solver.Expression.Fresh_Variable (name, sort)
    Bases: idp_solver.Expression.Expression

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

class idp_solver.Expression.NumberConstant (**kwargs)
    Bases: idp_solver.Expression.Expression

as_rigid ()
    returns a NumberConstant or Constructor, or None

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

class idp_solver.Expression.Brackets (**kwargs)
    Bases: idp_solver.Expression.Expression

as_rigid ()
    returns a NumberConstant or Constructor, or None
```


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

5.2.3 `idp_solver.idp_to_Z3`

Translates AST tree to Z3

TODO: vocabulary

5.2.4 `idp_solver.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)

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

5.2.5 `idp_solver.Parse`

Classes to parse and annotate an IDP-Z3 theory.

class `idp_solver.Parse.Idp` (***kwargs*)
 Bases: `object`

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

execute()
 Execute the IDP program

class `idp_solver.Parse.Vocabulary` (***kwargs*)
 Bases: `object`

The class of AST nodes representing a vocabulary block.

class `idp_solver.Parse.SymbolDeclaration` (***kwargs*)
 Bases: `object`

The class of AST nodes representing an entry in the vocabulary, declaring a symbol.

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

name
 the identifier of the symbol

Type string

sorts
 the types of the arguments

Type List[Sort]

out
 the type of the symbol

type
 the name of the type of the symbol

Type string

arity

the number of arguments

Type int

function

True if the symbol is a function

Type bool

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*]

typeConstraints

the type constraint on the ranges of the symbol applied to each possible tuple of arguments

Type List[*Expression*]

class idp_solver.Parse.Theory (**kwargs)

Bases: object

The class of AST nodes representing a theory block.

class idp_solver.Parse.Structure (**kwargs)

Bases: object

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

5.2.6 idp_solver.Problem

Class to represent a collection of theory and structure blocks.

class idp_solver.Problem.Problem (*blocks)

A collection of theory and structure blocks.

constraints

a set of assertions.

Type *OrderedSet*

assignments

the set of assignments. The assignments are updated by the different steps of the problem resolution.

Type *Assignment*

clark

A mapping of defined symbol to the rule that defines it.

Type dict[*SymbolDeclaration*, Rule]

def_constraints

A mapping of defined symbol to the whole-domain constraint equivalent to its definition.

Type dict[*SymbolDeclaration*], *Expression*

interpretations

A mapping of enumerated symbols to their interpretation.

Type dict[string, SymbolInterpretation]

_formula

the logic formula that represents the problem.

Type *Expression*, optional

questions

the set of questions in the problem. Questions include predicates and functions applied to arguments, comparisons, and variable-free quantified expressions.

Type *OrderedSet*

co_constraints

the set of co_constraints in the problem.

Type *OrderedSet*

classmethod make (*theories*, *structures*)

polymorphic creation

formula ()

the formula encoding the knowledge base

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

output: a list of Assignments, ending with a string

symbolic_propagate (*tag=<Status.UNIVERSAL: 4>*)

determine the immediate consequences of the constraints

propagate (*tag=<Status.CONSEQUENCE: 6>*, *extended=False*)

determine all the consequences of the constraints

simplify ()

simplify constraints using known assignments

decision_table (*goal_string=""*, *timeout=20*, *max_rows=50*, *first_hit=True*)

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.

Returns the non-empty cells of the decision table

Return type list(list(*Assignment*))

5.2.7 idp_solver.Run

Classes to execute the main block of an IDP program

`idp_solver.Run.model_check` (*theories*, *structures=None*)
output: “sat”, “unsat” or “unknown”

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

`idp_solver.Run.model_propagate` (*theories*, *structures=None*)
output: a list of Assignment

`idp_solver.Run.decision_table` (*theories*, *structures=None*, *goal_string=""*, *timeout=20*, *max_rows=50*, *first_hit=True*)
output: a list of rows for a decision table

`idp_solver.Run.execute` (*self*)
Execute the IDP program

5.2.8 idp_solver.Simplify

Methods to simplify a logic expression.

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

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

5.2.9 idp_solver.Substitute

Methods to

- substitute a constant by its value in an expression
- replace symbols interpreted in a structure by their interpretation
- instantiate an expresion, i.e. replace a variable by a value
- expand quantifiers

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

(see docs/zettlr/Substitute.md)

5.2.10 idp_solver.utils

Various utilities (in particular, OrderedSet)

```
idp_solver.utils.indent = '\n '
```

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.

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

5.3 idp_server module

5.3.1 idp_server.Inferences

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

```
idp_server.Inferences.get_relevant_subtences (self)
    sets 'relevant' in self.assignments sets rank of symbols in self.relevant_symbols removes irrelevant constraints in self.constraints
```

5.3.2 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.decode_UTF (json_str: str) → str
    Convert all Python unicode to actual unicode characters.
```

Parameters `json_str` – the string to convert

Returns the converted string

Return type str

```
idp_server.IO.json_to_literals (state, jsonstr: str)
```

Parse a json string and create assignments in a state accordingly. This function can also overwrite assignments that have already been set as a default assignment, effectively overriding the default.

Parameters

- **state** – a State object containing the concepts that appear in the json
- **jsonstr** – the user's assignments in json

Returns the assignments

Return type idp_solver.Assignments

5.3.3 idp_server.rest

This module implements the IDP-Z3 web server

```
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
```

5.3.4 idp_server.State

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

```
class idp_server.State.State (idp: idp_solver.Parse.Idp)
```

Contains a state of problem solving

```
add_given (jsonstr: str)
```

Add the assignments that the user gave through the interface. These are in the form of a json string. This method also sets the values of the default structure.

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

Returns the state with the jsonstr added

Return type *State*

```
idp_server.State.make_state (idp: idp_solver.Parse.Idp, jsonstr: str) → idp_server.State.State
```

Manages the cache of States.

Parameters

- **idp** – IDP code parsed into Idp object
- **jsonstr** – the user's assignments in json

Returns the complete state of the system

Return type *State*

CHAPTER 6

Index

CHAPTER 7

Indices and tables

- *Index*
- search

i

- `idp_server.Inferences`, 33
- `idp_server.IO`, 33
- `idp_server.rest`, 33
- `idp_server.State`, 34
- `idp_solver.Assignments`, 22
- `idp_solver.Expression`, 23
- `idp_solver.Idp_to_Z3`, 29
- `idp_solver.Parse`, 29
- `idp_solver.Problem`, 30
- `idp_solver.Propagate`, 29
- `idp_solver.Run`, 32
- `idp_solver.Simplify`, 32
- `idp_solver.Substitute`, 32
- `idp_solver.utils`, 32

Symbols

`_formula` (*idp_solver.Problem.Problem* attribute), 31

A

AAggregate (class in *idp_solver.Expression*), 27

AComparison (class in *idp_solver.Expression*), 27

AConjunction (class in *idp_solver.Expression*), 27

`add_given()` (*idp_solver.State.State* method), 34

ADisjunction (class in *idp_solver.Expression*), 27

AEquivalence (class in *idp_solver.Expression*), 27

AImplication (class in *idp_solver.Expression*), 26

AMultDiv (class in *idp_solver.Expression*), 27

`annotate()` (*idp_solver.Expression.AAggregate* method), 27

`annotate()` (*idp_solver.Expression.AComparison* method), 27

`annotate()` (*idp_solver.Expression.AppliedSymbol* method), 27

`annotate()` (*idp_solver.Expression.AQuantification* method), 26

`annotate()` (*idp_solver.Expression.ARImplication* method), 27

`annotate()` (*idp_solver.Expression.Expression* method), 24

`annotate()` (*idp_solver.Expression.Variable* method), 28

`annotate()` (*idp_solver.Parse.Structure* method), 30

`annotatel()` (*idp_solver.Expression.AComparison* method), 27

`annotatel()` (*idp_solver.Expression.AppliedSymbol* method), 27

`annotatel()` (*idp_solver.Expression.AQuantification* method), 26

`annotatel()` (*idp_solver.Expression.AUnary* method), 27

`annotatel()` (*idp_solver.Expression.BinaryOperator* method), 26

`annotatel()` (*idp_solver.Expression.Brackets* method), 28

`annotatel()` (*idp_solver.Expression.Expression* method), 24

`annotatel()` (*idp_solver.Expression.IfExpr* method), 26

`annotation` (vocabulary), 16

`annotations` (*idp_solver.Expression.Expression* attribute), 24

`annotations` (*idp_solver.Parse.SymbolDeclaration* attribute), 29

APower (class in *idp_solver.Expression*), 27

AppliedSymbol (class in *idp_solver.Expression*), 27

AQuantification (class in *idp_solver.Expression*), 26

ARImplication (class in *idp_solver.Expression*), 27

`arity` (*idp_solver.Parse.SymbolDeclaration* attribute), 29

`as_rigid()` (*idp_solver.Expression.Brackets* method), 28

`as_rigid()` (*idp_solver.Expression.Constructor* method), 26

`as_rigid()` (*idp_solver.Expression.Expression* method), 25

`as_rigid()` (*idp_solver.Expression.NumberConstant* method), 28

`as_set_condition()` (*idp_solver.Assignments.Assignment* method), 23

`as_set_condition()` (*idp_solver.Expression.Expression* method), 25

Assignment (class in *idp_solver.Assignments*), 22

Assignments (class in *idp_solver.Assignments*), 23

`assignments` (*idp_solver.Problem.Problem* attribute), 30

ASumMinus (class in *idp_solver.Expression*), 27

AUnary (class in *idp_solver.Expression*), 27

B

BinaryOperator (class in *idp_solver.Expression*), 26

Brackets (class in *idp_solver.Expression*), 28

C

clark (*idp_solver.Problem.Problem attribute*), 30
 co_constraint (*idp_solver.Expression.Expression attribute*), 24
 co_constraints (*idp_solver.Problem.Problem attribute*), 31
 co_constraints() (*idp_solver.Expression.Expression method*), 25
 code (*idp_solver.Expression.Expression attribute*), 23
 collect() (*idp_solver.Expression.AAggregate method*), 27
 collect() (*idp_solver.Expression.AppliedSymbol method*), 28
 collect() (*idp_solver.Expression.AQuantification method*), 26
 collect() (*idp_solver.Expression.BinaryOperator method*), 26
 collect() (*idp_solver.Expression.Expression method*), 24
 collect() (*idp_solver.Expression.Variable method*), 28
 constant, 6
 constraint, 8
 constraints (*idp_solver.Problem.Problem attribute*), 30
 constructor, 6
 Constructor (*class in idp_solver.Expression*), 26
 copy() (*idp_solver.Assignments.Assignments method*), 23
 copy() (*idp_solver.Expression.Expression method*), 24

D

decision_table() (*idp_solver.Problem.Problem method*), 31
 decision_table() (*in module idp_solver.Run*), 32
 decode_UTF() (*in module idp_server.IO*), 33
 def_constraints (*idp_solver.Problem.Problem attribute*), 31
 default structure, 17
 definition, 8
 display block, 15
 domain (*idp_solver.Parse.SymbolDeclaration attribute*), 30

E

environment, 16
 eval (*class in idp_server.rest*), 34
 evalWithGraph (*class in idp_server.rest*), 34
 execute() (*idp_solver.Parse.Idp method*), 29
 execute() (*in module idp_solver.Run*), 32
 expand() (*idp_solver.Problem.Problem method*), 31
 expand_quantifiers() (*idp_solver.Expression.Expression method*), 25
 expanded view, 15

Expression (*class in idp_solver.Expression*), 23

F

formula() (*idp_solver.Problem.Problem method*), 31
 Fresh_Variable (*class in idp_solver.Expression*), 28
 fresh_vars (*idp_solver.Expression.Expression attribute*), 24
 function, 6
 function (*idp_solver.Parse.SymbolDeclaration attribute*), 30

G

get_relevant_subtences() (*in module idp_server.Inferences*), 33

H

HelloWorld (*class in idp_server.rest*), 33

I

Idp (*class in idp_solver.Parse*), 29
 IDP3, 11
 idp_server.Inferences (*module*), 33
 idp_server.IO (*module*), 33
 idp_server.rest (*module*), 33
 idp_server.State (*module*), 34
 idp_solver.Assignments (*module*), 22
 idp_solver.Expression (*module*), 23
 idp_solver.Idp_to_Z3 (*module*), 29
 idp_solver.Parse (*module*), 29
 idp_solver.Problem (*module*), 30
 idp_solver.Propagate (*module*), 29
 idp_solver.Run (*module*), 32
 idp_solver.Simplify (*module*), 32
 idp_solver.Substitute (*module*), 32
 idp_solver.utils (*module*), 32
 idpOf() (*in module idp_server.rest*), 33
 IfExpr (*class in idp_solver.Expression*), 26
 include vocabulary, 7
 indented (*in module idp_solver.utils*), 32
 Installation, 1
 instances (*idp_solver.Parse.SymbolDeclaration attribute*), 30
 instantiate() (*idp_solver.Expression.Expression method*), 25
 intended meaning, 16
 Interactive Consultant, 1
 interpret() (*idp_solver.Expression.Expression method*), 25
 interpretations (*idp_solver.Problem.Problem attribute*), 31

J

join_set_conditions() (*in module idp_solver.Simplify*), 32

`json_to_literals()` (in module `idp_server.IO`), 33

M

`main_block`, 9

`make()` (`idp_solver.Expression.AQuantification` class method), 26

`make()` (`idp_solver.Expression.BinaryOperator` class method), 26

`make()` (`idp_solver.Problem.Problem` class method), 31

`make_state()` (in module `idp_server.State`), 34

`meta` (class in `idp_server.rest`), 34

`metaJSON()` (in module `idp_server.IO`), 33

`metaWithGraph` (class in `idp_server.rest`), 34

`model_check()` (in module `idp_solver.Run`), 32

`model_expand()` (in module `idp_solver.Run`), 32

`model_propagate()` (in module `idp_solver.Run`), 32

N

`name` (`idp_solver.Parse.SymbolDeclaration` attribute), 29

`negate()` (`idp_solver.Assignments.Assignment` method), 23

`NumberConstant` (class in `idp_solver.Expression`), 28

O

`OrderedSet` (class in `idp_solver.utils`), 33

`original` (`idp_solver.Expression.Expression` attribute), 24

`out` (`idp_solver.Parse.SymbolDeclaration` attribute), 29

P

`post()` (`idp_server.rest.meta` method), 34

`post()` (`idp_server.rest.metaWithGraph` method), 34

`post()` (`idp_server.rest.run` method), 34

`predicate`, 6

`Problem` (class in `idp_solver.Problem`), 30

`propagate()` (`idp_solver.Problem.Problem` method), 31

`propagate1()` (`idp_solver.Expression.Expression` method), 25

`proposition`, 6

Q

`quantifier expression`, 8

`questions` (`idp_solver.Problem.Problem` attribute), 31

R

`range` (`idp_solver.Parse.SymbolDeclaration` attribute), 30

`relevant` (`idp_solver.Assignments.Assignment` attribute), 23

`rule`, 8

`run` (class in `idp_server.rest`), 34

S

`same_as()` (`idp_solver.Assignments.Assignment` method), 23

`sentence`, 8

`sentence` (`idp_solver.Assignments.Assignment` attribute), 22

`Shebang`, 5

`simpler` (`idp_solver.Expression.Expression` attribute), 24

`simplify()` (`idp_solver.Problem.Problem` method), 31

`sorts` (`idp_solver.Parse.SymbolDeclaration` attribute), 29

`State` (class in `idp_server.State`), 34

`Status` (class in `idp_solver.Assignments`), 22

`status` (`idp_solver.Assignments.Assignment` attribute), 23

`structure`, 9

`Structure` (class in `idp_solver.Parse`), 30

`sub_exprs` (`idp_solver.Expression.Expression` attribute), 24

`substitute()` (`idp_solver.Expression.AppliedSymbol` method), 28

`substitute()` (`idp_solver.Expression.Expression` method), 25

`substitute()` (`idp_solver.Expression.Variable` method), 28

`symbol`, 6

`symbol_decl` (`idp_solver.Assignments.Assignment` attribute), 23

`SymbolDeclaration` (class in `idp_solver.Parse`), 29

`symbolic_propagate()` (`idp_solver.Expression.Expression` method), 25

`symbolic_propagate()` (`idp_solver.Problem.Problem` method), 31

T

`term`, 7

`theory`, 7

`Theory` (class in `idp_solver.Parse`), 30

`type`, 6

`type` (`idp_solver.Expression.Expression` attribute), 24

`type` (`idp_solver.Parse.SymbolDeclaration` attribute), 29

`typeConstraints` (`idp_solver.Parse.SymbolDeclaration` attribute), 30

U

`unknown_symbols()` (`idp_solver.Expression.Expression` method), 25

`update_exprs()` (`idp_solver.Expression.AppliedSymbol` method), 28

`update_exprs()` (`idp_solver.Expression.Constructor` method), 26

`update_exprs()` (`idp_solver.Expression.Fresh_Variable` method), 28

`update_exprs()` (*idp_solver.Expression.NumberConstant*
 method), 28
`update_exprs()` (*idp_solver.Expression.Variable*
 method), 28

V

`value` (*idp_solver.Assignments.Assignment attribute*),
 22
`value` (*idp_solver.Expression.Expression attribute*), 24
`Variable` (*class in idp_solver.Expression*), 28
`vocabulary`, 6
`Vocabulary` (*class in idp_solver.Parse*), 29