IDP-Z3

Pierre Carbonnelle

Nov 19, 2020

CONTENTS:

CHAPTER

ONE

INTRODUCTION

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.

The source code of IDP-Z3 is publicly available under the GNU LGPL v3 license.

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

CHAPTER

TWO

THE IDP LANGUAGE

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 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
}

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

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

2.4 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'`.

2.4.1 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 ``.

2.4.2 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.

2.4.3 Predicates

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

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

2.4.4 Propositions and Constants

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

```
MyProp1 MyProp2.
MyConstant: int
```

2.4.5 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.

2.4.6 Include another vocabulary

A vocabulary W may include a previously defined vocabulary V:

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

2.5 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.

2.5.1 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 $F(t_1, t_2, .., t_n)$ 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.

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 $+, -, *, /, \hat{}, \%$. 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 $\#\{v_1[typeOfV_1]..v_n[typeOfV_n] : \phi\}$ 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 is sum, $v_1 v_2..v_n$ are variables, ϕ is a *sentence*, and t is a term.

The term denotes the sum of t for each distinct tuple of values for $v_1 v_2..v_n$ 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.

2.5.2 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 and $P()$ are sentences.

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: $=<, >=, \sim=$). Comparison operators can be chained as customary.

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

Logic connectives $\phi_1 \phi_2$ is a sentence when ϕ_1, ϕ_2 are two sentences and 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.

Quantified formulas *Quantified formulas* are sentences. They have one of these two forms, where $v_1, .., v_n$ are variables and ϕ is a sentence:

$$\exists v_1[typeOfV_1]..v_n[typeOfV_n] : \phi$$
$$\forall v_1[typeOfV_1]..v_n[typeOfV_n] : \phi$$

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.

"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.

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.

2.5.3 Definitions

A *definition* defines concepts, i.e. *predicate*s or *function*s, in terms of other concepts. A definition consists of a set of rules, enclosed by `{` and `}`.

Rules have one of the following forms:

$$\forall v_1[typeOfV_1]..v_n[typeOfV_n] : P(t_1, .., t_n) \leftarrow \phi.$$
$$\forall v_1[typeOfV_1]..v_n[typeOfV_n] : F(t_1, .., t_n) = t \leftarrow \phi.$$

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*. `<-` can be used instead of '←'. If the body is `true`, the left arrow and body of the rule can be omitted.

2.6 Structure

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

A *structure* specifies the interpretation of some *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 enumerations, having one of the following forms:

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

where P is a predicate of arity n, F is a function of arity n, and el_i^j are *constructor*s 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).

2.7 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 $expand(s_1, .., s_n)$ specifies that *symbols* $s_1, .., s_n$ 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 $hide(s_1, .., s_n)$ specifies that symbols $s_1, .., s_n$ 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 *expand*ed.

relevant $relevant(s_1, .., s_n)$ specifies that symbols $s_1, .., s_n$ 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.

2.8 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`.

print(...) Prints the arguments on stdout

2.8.1 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.

2.9 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)`.

CHAPTER

THREE

COMMAND LINE INTERFACE

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

```
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*

CHAPTER

FOUR

INDEX

CHAPTER

FIVE

INDICES AND TABLES

- *Index*
- search

INDEX