SAARLAND UNIVERSITY ARTIFICIAL INTELLIGENCE
Prof. Jana Koehler, Dr. Álvaro Torralba

Practical Exercise Sheet 2.
Solutions due Tuesday, **May 28, 23:59** uploaded in the AI CMS. See submission instructions below.

This exercise sheet is accompanied with several source files, which we provide through a separate `zip` archive. You can download this archive from from the AI CMS under the Materials category.

Exercise 3: PL1 Modeling in Z3. (10 Points)

In this exercise, your task is to model different problems in predicate logic, and to solve those models using Z3[1] [2]. To do so, you have to create a text file in Z3's input language, which you can then pass as command line argument to the Z3 executable. The Z3 executable is available in the VM through typing `z3` in a terminal. Printing additional information about the solving process can be done through the `-st` option, e.g., `z3 -st path/to/model.z3`. The result of Z3 (`sat` or `unsat`) will be printed to the console. Table 1 shows an overview of the Z3 language fragments that are relevant for this sheet. **You must not use any statement that is not included in this table**.

(i) Use Z3 to show that under assumptions (1), (2), and (3):

(1) $\forall x \in \mathbb{Z} : \exists y \in \mathbb{Z} : P(x, y)$

(2) $\forall x \in \mathbb{Z} : \exists y \in \mathbb{Z} : \neg R(x, y)$

(3) $\forall x \in \mathbb{Z} : \forall y \in \mathbb{Z} : [P(x, y) \vee \neg R(x, y)] \implies [\forall z \in \mathbb{Z} : Q(x, z) \vee (\neg P(y, z) \wedge R(y, z))]$

the formula $\forall x \in \mathbb{Z} : \exists y \in \mathbb{Z} : Q(x, y)$ is valid. Start with the provided template file `practex3i-model.z3`.

(ii) Consider a chain of 4 *elements*. Encode the following model in Z3, and show that it is unsatisfiable. `practex3ii-model.z3` defines 4 different predicates which you must use in order to do so:[3] `IsZero`, `Sum`, `Successor`, `Predicate`.

(1) Every element has exactly one successor.

(2) The successor relation is asymmetric.

[1] `https://github.com/Z3Prover/z3`
[2] An introduction can be found in `http://rise4fun.com/z3/tutorial/guide`.
[3] If needed, you are however free to define additional predicates.

(3) Exactly one of the elements is the *zero element*.

(4) Adding to any element the zero element gives the same element again.

(5) Summing is commutative.

(6) The sum of two elements is uniquely defined, i.e., the sum of two elements has at most one result.

(7) Summing is invariant under the successor relation, i.e., if z is the result of the sum of two elements x and y and x' is the successor of x, then the successor of z is the result of the sum of x' and y.

(8) `Predicate` holds for an element if and only if it does not hold for its successor.

(9) The zero element satisfies `Predicate`.

(10) There is an element whose sum with itself does not satisfy `Predicate`.

`practex3ii-model.z3` already contains an encoding of formula (10). Your task is to model the remaining ones, and to use `Z3` in order to proof that this model is indeed unsatisfiable.

Hint: Your model should become satisfiable when commenting out any one of the formulas above. To debug your model, you may want to use the `(get-model)` directive.

Exercise 4: Unification Algorithm. (10 Points)

Implement the unification algorithm presented in the lecture. To do so, we provide the following two files:

- `pl1_formula.py` (DO NOT MODIFY THIS FILE!)
  This file provides the base classes to represent PL1 clauses, along with some utility functions that may become useful for the implementation of the unification algorithm. Make sure to read the comments at the beginning of the file.

  PL1 clauses are represented as lists of **`Atom`** objects. An **`Atom`** object is an instantiation of an n-ary **`PredicateSymbol`** with a list of n **`Term`** objects. There are three types of **`Term`** objects: **`Variable`**, **`Constant`**, and **`Function`**. Similar to **`Atom`**s, a **`Function`** gives an instantiation of an n-ary **`FunctionSymbol`** with a list of n **`Term`** objects. Consider the PL1 clause $\{P(x, c, f(y)), P(x, z, z)\}$ from slide 25, chapter 8. The representation in Python looks as follows:

```
1 P = PredicateSymbol("P", 3)
2 f = FunctionSymbol("f", 1)
```

```
x = Variable("x")
y = Variable("y")
z = Variable("z")
c = Constant("c")
atom1 = P(x, c, f(y))
atom2 = P(x, z, z)
clause = [atom1, atom2]
```

All PL1 expression objects can be compared via the == and != comparison operators. Additionally, they provide two functions:

- `contains_variable`
  In the example above, both `x.contains_variable(x)` and `atom1.contains_variable(y)` return true, while `c.contains_variable(x)` and `atom2.contains_variable(y)` return false.
- `apply_substitution`
  For the substitution $s = \{\frac{x}{f(y)}\}$, `atom1.apply_substitution(s)` returns an **Atom** object representing $P(f(y), c, f(y))$. Note that the application of `apply_substitution` creates a new **Atom** object, in particular leaving `atom1` untouched.

The predicate symbol of an **Atom** object `a` can be accessed via `a.symbol`; the terms via `a.terms`. Similarly for **Function** objects. For example, `atom1.terms[0]` above gives `x`, `atoms1.terms[2]` gives the **Function** object representing $f(y)$. Function and predicate symbols provide information about their arity via the `arity` field.

- `unification.py`
  The implementation of the unification algorithm goes in here. Missing parts are indicated with `TODO` comments. The bottom of the file lists some example calls. Substitutions are represented as objects of the **Substitution** class. The file provides detailed class information in the comments.

The implementation of the unification algorithm is split into three parts:

(i) Composition of two substitution objects (slide 21 of chapter 8). Your task is to fill in the content of the **Substitution**`.Composition(s1, s2)` method. This function should return a new **Substitution** object representing the composed substitution $s_1 s_2$.

(ii) The computation of the disagreement set (slide 25 of chapter 8). Fill in the missing content of the `compute_disagreement_set(atoms)` function. You can assume that

`atoms` always gives a list of **`Atom`** objects, all using the same predicate symbol. The result should be a list of **`Term`** objects, the disagreement set, possibly empty if all atoms in `atoms` are the same.

(iii) The actual unification algorithm (slide 26 of chapter 8). Use your implementations of (i) and (ii) to implement the `unification(atoms)` function. As for the disagreement set computation, you can assume that `atoms` always gives a list of **`Atom`** objects, all using the same predicate symbol. If the expressions given by `atoms` are indeed unifiable, `unification` should return a **`Substitution`** object that stores an idempotent mgu. Otherwise the function should return `None`.

Submission Instructions

Solutions need to be packaged into a `.zip` file and uploaded in the AI CMS. The `.zip` file has to contain a single folder with name:

`AI2019_PE2_mat1_mat2_mat3`

where `mat1`, ..., `mat3` are the the matriculation numbers of the students who submit together. This folder must contain the following files:

- `authors.txt` listing the names and matriculation numbers of all students who submit together. Use one line per student.
- The `.z3` model files containing your solutions to Exercise 3.
- Your modified `unification.py` script, with the implementation of Exercise 4.

Do not add any other folder or subfolder, i.e., place all files directly into `AI2019_PE2_mat1_mat2_mat3`. Do not place any file outside of `AI2019_PE2_mat1_mat2_mat3`. Only one student of each group needs to do the submission. **We will deduct 3 points if you do not adhere to these rules!**

Statement	Description
General	
`(check-sat)`	Checks whether the predicate logic problem defined up to this point is satisfiable.
`(assert E)`	Adds the boolean / predicate logic expression `E` to the list of formulas to be verified in `check-sat`.
`(get-value (E))`	Prints the value of `E`, where `E` can be an arbitrary expression such as constant, propositions, or boolean combination thereof (must occur after `(check-sat)`).
`(get-model)`	Prints all variable assignments (must occur after `(check-sat)`).
`(echo "message")`	Prints `message` to the console.
`; This is a comment`	Commenting.
Mathematical Expressions	
`(declare-const var Int)`	Defines integer variable `var`.
`var`	Evaluates to the value of variable `var`.
Boolean Expressions	
`true`	Constant for true.
`false`	Constant for false.
`(declare-const p Bool)`	Declares a new proposition with name `p`.
`p`	Evaluates to the truth assignment of `p`
`(not E)`	Evaluates to the negation of `E`.
`(and` E_1 `...` E_n`)`	Conjunction over the expressions E_1 to E_n.
`(or` E_1 `...` E_n`)`	Disjunction over the expressions E_1 to E_n.
`(=` E_1 `...` E_n`)`	Is true if and only if the expressions E_1 to E_n evaluate to the same value. Can be used e.g. to compare variables, or to compute the equivalence over boolean expresions.
`(=>` E E'`)`	Implication, is true if E implies E'.
`(distinct` E_1 `...` E_n`)`	Is true iff every expresion E_1 to E_n evaluates to a different value.
Predicate Logic	
`(declare-fun P (`$T_1 \dots T_n$`) Bool)`	Declares an n-ary predicate with name `P` and parameter types T_1 to T_n.
`(P` t_1 `...` t_n`)`	Evaluates to the value of the n-ary predicate `P` with arguments t_1 to t_n.
`(forall ((`x_1 T_1`) ...`	All quantification over n variables: x_1 with type T_1, ..., x_n with type T_n. $x_1, \dots, x_n$ can be used in the expression `E`.
`(exists ((`x_1 T_1`) ...`	Existential quantification over n variables: x_1 with type T_1, ..., x_n with type T_n. $x_1, \dots, x_n$ can be used in the expression `E`.

Table 1: Z3 input language fragment you may use for the predicate logic modeling exercises. You may use the data types `Int`, `Bool`, or the ones specified in the template files we provide.