py-typedlogic

None

Table of contents

1. py	r-typedlogic: Bridging Formal Logic and Typed Python	4
1.1	Key Features	4
1.2	Why TypedLogic?	5
1.3	Installation	5
1.4	Specify logical axioms directly in Python	5
1.5	Performing reasoning from within Python	6
2. Co	oncepts	7
2.1	typed-logic: Bridging Formal Logic and Typed Python	7
2.2	Core Concepts	7
2.3	Data Model	10
2.4	Decorators	0
2.5	Generators	0
2.6	Solvers	0
2.7	Models	0
3. Sc	olvers	0
3.1	Solvers	0
3.2	Clingo Integrations	0
3.3	Z3 Integrations	0
3.4	Souffle Integrations	0
3.5	Prover9 Integrations	0
3.6	SnakeLog Integrations	0
3.7	LLM Integrations	0
4. Co	ompilers	0
4.1	Compilers	0
4.2	YAML Compiler	0
4.3	Prolog Compiler	0
4.4	TPTP Compiler	0
4.5	FOR Compiler	0
4.6	S-Expression Compiler	0
5. Pa	arsers	0
5.1	Parsers	0
5.2	Python Parser	0
5.3	YAML Parser	0
5.4	OWLPython Parser	0
5.5	RDF Parser	0

6. Frameworks	0
6.1 OWL-DL	0
7. CLI Reference	0
8. typedlogic	0
8.1 convert	0
8.2 solve	0
9. Troubleshooting	0
9.1 Type Checking Errors	0
9.2 Solver Not Finding Solutions	0
9.3 Performance Issues	0
9.4 Integration Issues	0
9.5 Unexpected Logical Results	0
10. Roadmap	0
10.1 Solvers	0
10.2 Framework Integrations	0
10.3 Transformations	0
10.4 Documentation	0
10.5 Current Limitations	0

1. py-typedlogic: Bridging Formal Logic and Typed Python

TypedLogic is a powerful Python package that bridges the gap between formal logic and strongly typed Python code. It allows you to leverage fast logic programming engines like Souffle while specifying your logic in mypy-validated Python code.

```
links.py
                               stdout
                                             Semantics
                 run.py
# links.py
from pydantic import BaseModel
from typedlogic import FactMixin, gen2
from typedlogic.decorators import axiom
class Link(BaseModel, FactMixin):
      ""A link between two entities"""
    source: ID
class Path(BaseModel, FactMixin):
     ""An N-hop path between two entities"""
    source: ID
    target: ID
    hops: int
{\tt assert\ Link(source=x,\ target=y)\ >>\ Path(source=x,\ target=y,\ hops=1)}
def transitivity(x: ID, y: ID, z: ID, d1: int, d2: int):
    """Transitivity of paths, plus hop counting"""
    assert ((Path(source=x, target=y, hops=d1) & Path(source=y, target=z, hops=d2)) >>
             {\tt Path(source=x,\ target=z,\ hops=d1+d2))}
@axiom
def reflexivity():
     """No paths back to self"""
    assert not any(Path(source=x, target=x, hops=d) for x, d in gen2(ID, int))
from typedlogic.integrations.souffle_solver import SouffleSolver
import links as links
solver = SouffleSolver()
solver.load(links) ## source for definitions and axioms
  Add data
links = [Link(source='CA', target='OR'), Link(source='OR', target='WA')] for link in links:
    solver.add(link)
model = solver.model()
for fact in model.iter_retrieve("Path"):
    print(fact)
```

Output:

```
Path(source='CA', target='0R', hops=1)
Path(source='CA', target='WA', hops=1)
Path(source='CA', target='WA', hops=2)
```

To convert the Python code to first-order logic, use the CLI:

```
{\it typedlogic \ convert \ links.py \ -t \ fol}
```

Output:

```
\forall [x:ID \ y:ID]. \ Link(x, y) \rightarrow Path(x, y, 1)

\forall [x:ID \ y:ID \ z:ID \ d:int]. \ Path(x, y, d1) \land Path(y, z, d2) \rightarrow Path(x, z, d1+d2)

\neg \exists [x:ID \ d:int]. \ Path(x, x, d)
```

1.1 Key Features

• Write logical axioms and rules using familiar Python syntax

- · Benefit from strong typing and mypy validation
- Seamless integration with logic programming engines
- Support for various solvers, including Z3 and Souffle
- Compatible with popular Python libraries like Pydantic

1.2 Why TypedLogic?

TypedLogic combines the best of both worlds: the expressiveness and familiarity of Python with the power of formal logic and fast logic programming engines. This unique approach allows developers to:

- 1. Write more maintainable and less error-prone logical rules
- 2. Catch type-related errors early in the development process
- 3. Seamlessly integrate logical reasoning into existing Python projects
- 4. Leverage the performance of specialized logic engines without sacrificing the Python ecosystem

Get started with TypedLogic and experience a new way of combining logic programming with strongly typed Python!

1.3 Installation

Install TypedLogic using pip:

```
pip install typedlogic
```

With all extras pre-installed:

```
pip install typedlogic[all]
```

You can also use pipx to run the ${\hbox{\scriptsize CLI}}$ without installing the package globally:

```
pipx run "typedlogic[all]" --help

## Define predicates using Pythonic idioms

""python
from typedlogic.integrations.frameworks.pydantic import FactBaseModel

ID = str

class Link(FactBaseModel):
    source: ID
    target: ID

class Path(FactBaseModel):
    source: ID
    target: ID
```

These can be used in the standard way in Python:

```
links = [Link(source='CA', target='OR'), Link(source='OR', target='WA')]
```

1.4 Specify logical axioms directly in Python

```
@axiom
def link_implies_path(x: ID, y: ID):
    """"For all x, y, if there is a link from x to y, then there is a path from x to y"""
    if Link(source=x, target=y):
        assert Path(source=x, target=y)

@axiom
def transitivity(x: ID, y: ID, z: ID):
    """For all x, y, z, if there is a path from x to y and a path from y to z,
        then there is a path from x to z"""
```

```
if Path(source=x, target=y) and Path(source=y, target=z):
    assert Path(source=x, target=z)
```

1.5 Performing reasoning from within Python

```
from typedlogic.integrations.snakelogic import SnakeSolver

solver = SnakeSolver()
solver.load("links.py") ## source for definitions and axioms
for link in links:
    solver.add(link)
model = solver.model()
for fact in model.iter_retrieve("Path"):
    print(fact)
```

prints:

```
Path(source='CA', target='OR')
Path(source='CA', target='WA')
Path(source='CA', target='WA')
```

2. Concepts

2.1 typed-logic: Bridging Formal Logic and Typed Python

typed-logic is a Python package that allows Python data models to be augmented using formal logical statements, which are then interpreted by *solvers* which can reason over combinations of programs and data allowing for *satisfiability checking*, and the generation of new data.

Currently the solvers supported are:

- Z3
- Souffle
- Clingo
- Prover9
- Snakelog

With support for more solvers (Vampire, OWL reasoners, etc.) planned.

typed-logic is aimed primarily at software developers and data modelers who are logic-curious, but don't necessarily have a background in formal logic. It is especially aimed at Python developers who like to use lightweight ways of ensuring program and data correctness, such as Pydantic for data and mypy for type checking of programs.

2.1.1 Key Features

- · Write logical axioms and rules using familiar Python syntax
- Benefit from strong typing and mypy validation
- Seamless integration with logic programming engines
- \bullet Support for various solvers, including Z3 and Souffle
- Compatible with popular Python libraries like Pydantic

2.1.2 Why TypedLogic?

TypedLogic combines the best of both worlds: the expressiveness and familiarity of Python with the power of formal logic and fast logic programming engines. This unique approach allows developers to:

- 1. Write more maintainable and less error-prone logical rules
- 2. Catch type-related errors early in the development process
- 3. Seamlessly integrate logical reasoning into existing Python projects
- 4. Leverage the performance of specialized logic engines without sacrificing the Python ecosystem

Get started with TypedLogic and experience a new way of combining logic programming with strongly typed Python!

2.2 Core Concepts

TypedLogic is built around several core concepts that blend logical programming with typed Python. Understanding these concepts is crucial for effectively using the library.

2.2.1 Use Python idioms to define your data structures

Facts are the basic units of information in TypedLogic. They are represented as Python classes that inherit from both pydantic.BaseModel and FactMixin. This approach allows you to define strongly-typed facts with automatic validation.

Example:

```
from pydantic import BaseModel
from typedlogic import FactMixin

PersonID = str
PetID = str

class Person(BaseModel, FactMixin):
    name: PersonID

class PersonAge(BaseModel, FactMixin):
    name: PersonID
    age: int

class Pet(BaseModel, FactMixin):
    name: PetID

class PetSpecies(BaseModel, FactMixin):
    name: PetID

class PetSpecies(BaseModel, FactMixin):
    name: PetID

class PetSpecies(BaseModel, FactMixin):
    person: PetSonID
    pet: PetID
```

Note in many logic frameworks these definitions don't have a direct translation, but in sorted logics, these may correspond to predicate definitions.

2.2.2 Axioms

Axioms are logical rules or statements that define relationships between facts. In TypedLogic, axioms are defined using Python functions $\frac{1}{2}$ decorated with @axiom.

Example:

```
from typedlogic.decorators import axiom

@axiom
def constraints(person: PersonID, pet: PetID):
    if OwnsPet(person=person, pet=pet):
        assert Person(name=person) and Pet(name=pet)
```

You can also derive new facts from axioms:

2.2.3 Generators

Generators like gen1, gen2, etc., are used within axioms to create typed placeholders for variables. They help maintain type safety while defining logical rules.

You can use generators in combination with Python all and any functions to express quantified sentences:

```
@axiom
def entail_same_owner():
    assert all(SameOwner(pet1=pet1, pet2=pet2)
        for person, pet1, pet2 in gen3(PersonID, PetID, PetID)
        if OwnsPet(person=person, pet=pet1) and OwnsPet(person=person, pet=pet2))
```

See Generators for more information.

2.2.4 Solvers

TypedLogic supports multiple solvers, including Z3 and Souffle. Solvers are responsible for reasoning over the facts and axioms to derive new information or check for consistency.

Example (using Z3 solver):

```
from typedlogic.integrations.solvers.z3 import Z3Solver
solver = Z3Solver()
solver.add(theory)
result = solver.check()
```

2.2.5 Theories

A Theory in TypedLogic is a collection of predicate definitions, facts, and axioms. It represents a complete knowledge base that can be reasoned over.

Example:

```
from typedlogic import Theory

theory = Theory(
    name="family_relationships",
    predicate_definitions=[...],
    sentence_groups=[...],
)
```

2.3 Data Model

Data model for the typed-logic framework.

This module defines the core classes and structures used to represent logical constructs such as sentences, terms, predicates, and theories. It is based on the Common Logic Interchange Format (CLIF) and the Common Logic Standard (CL), with additions to make working with simple type systems easier.

Logical axioms are called sentences which organized into theories., which can be loaded into a solver.

While one of the goals of typed-logic is to be able to write logic intuitively in Python, this data model is independent of the mapping from the Python language to the logic language; it can be used independently of the python syntax.

Here is an example:

2.3.1 PredicateDefinition dataclass

Defines the name and arguments of a predicate.

```
>>> pdef = PredicateDefinition(predicate='FriendOf',
... arguments={'x': 'str', 'y': 'str'})
```

The arguments are mappings between variable names and types. You can use either base types (e.g. 'str', 'int', 'float') or custom types.

Custom types should be defined in the theory's $type_definitions$ attribute.

```
classDiagram
class PredicateDefinition {
    +String predicate
    +Dict arguments
    +String description
    +Dict metadata
}
PredicateDefinition --> "*" PredicateDefinition : parents
```

Source code in src/typedlogic/datamodel.py @dataclass class PredicateDefinition: Defines the name and arguments of a predicate. >>> pdef = PredicateDefinition(predicate='FriendOf', ... arguments={'x': 'str', 'y': 'str'}) The arguments are mappings between variable names and types. You can use either base types (e.g. 'str', 'int', 'float') or custom types. Custom types should be defined in the theory's `type_definitions` attribute. >>> pdef = PredicateDefinition(predicate='FriendOf', arguments={'x': 'Person', 'y': 'Person'}) >>> theory = Theory(... name="My theory", ... name="My theory", ... type_definitions={'Person': 'str'}, ... predicate_definitions=[pdef], ``mermaid classDiagram class PredicateDefinition { +String predicate +Dict arguments +String description +Dict metadata PredicateDefinition --> "*" PredicateDefinition : parents predicate: str arguments: Dict[str, str] description: Optional[str] = None metadata: Optional[Dict[str, Any]] = None parents: Optional[List[str]] = None python_class: Optional[Type] = None def argument_base_type(self, arg: str) -> str: typ = self.arguments[arg] try: import pydantic if isinstance(typ, pydantic.fields.FieldInfo): tvp = tvp.annotation except ImportError: pass return str(typ) @classmethod def from_class(cls, python_class: Type) -> "PredicateDefinition": Create a predicate definition from a python class :param predicate_class: :return: $\label{lem:class} return \ PredicateDefinition(predicate=python_class.__name__, \\ arguments=\{k: \ v \ for \ k, \ v \ in \ python_class.__annotations__.items()\},$

from_class(python_class) classmethod

Create a predicate definition from a python class

Parameters:

Name	Туре	Description	Default
predicate_class			required

Returns:



```
Source code in src/typedlogic/datamodel.py
 95
     @classmethod
def from_class(cls, python_class: Type) -> "PredicateDefinition":
96
97
          Create a predicate definition from a python class
 98
 99
          :param predicate_class:
100
          :return:
101
102
         return PredicateDefinition(predicate=python_class.__name__, arguments={k: v for k, v in python_class.__annotations__.items()},
103
104
105
```

2.3.2 Variable dataclass

A variable in a logical sentence.

Variables can have domains (types) specified:

Source code in src/typedlogic/datamodel.py @dataclass class Variable: A variable in a logical sentence. >>> x = Variable('x') >>> y = Variable('y') >>> s = Forall([x, y], Implies(Term('friend_of', x, y), Term('friend_of', y, x))) Variables can have domains (types) specified: >>> x = Variable('x', domain='str') >>> y = Variable('y', domain='str') >>> z = Variable('y', domain='int') >>> xa = Variable('xa', domain='int') >>> s = Forall([x, y, z]) The domains should be either base types or defined types in the theory's `type_definitions` attribute. domain: Optional[str] = None constraints: Optional[List[str]] = None def __eq__(self, other): return isinstance(other, Variable) and self.name == other.name def __str__(self): return "?" + self.name def __hash__(self): return hash(self.name) def as_sexpr(self) -> SExpression: sexpr = [type(self).__name__, self.name] if self.domain: return sexpr + [self.domain] else: return sexpr

2.3.3 Sentence

Bases: ABC

Base class for logical sentences.

Do not use this class directly; use one of the subclasses instead.

Model:

```
classDiagram

Sentence <|-- Term

Sentence <|-- BooleanSentence

Sentence <|-- QuantifiedSentence

Sentence <|-- Extension
```

Source code in src/typedlogic/datamodel.py ~	

```
157
       class Sentence(ABC):
158
159
            Base class for logical sentences.
160
            Do not use this class directly; use one of the subclasses instead.
161
162
163
            ```mermaid
164
165
 classDiagram
 ClassUlagram
Sentence <|-- Term
Sentence <|-- BooleanSentence
Sentence <|-- QuantifiedSentence
Sentence <|-- Extension
166
167
168
169
170
171
172
173
 def __init__(self):
174
 self._annotations = {}
175
 def __and__(self, other):
 return And(self, other)
176
177
178
 def __or__(self, other):
 return Or(self, other)
179
180
181
 def __invert__(self):
 return Not(self)
182
183
 def __sub__(self):
 return NegationAsFailure(self)
184
185
186
 def __rshift__(self, other):
187
 return Implies(self, other)
188
 def __lshift__(self, other):
 return Implied(self, other)
189
190
191
 def __xor__(self, other):
192
 return Xor(self, other)
193
 def iff(self, other):
 return Iff(self, other)
194
195
196
 def __lt__(self, other):
197
 return Term(operator.lt.__name__, self, other)
198
199
 def __le__(self, other):
200
 return Term(operator.le.__name__, self, other)
201
 def __gt__(self, other):
 return Term(operator.gt.__name__, self, other)
202
203
204
 def __ge__(self, other):
205
 return Term(operator.ge.__name__, self, other)
206
 def __add__(self, other):
 return Term(operator.add.__name__, self, other)
207
208
209
 def annotations(self) -> Dict[str, Any]:
210
211
212
 Annotations for the sentence.
213
 Annotations are always logically silent, but can be used to store metadata or other information.
214
215
 :return:
216
 return self._annotations or {}
217
218
 def add_annotation(self, key: str, value: Any):
219
220
 Add an annotation to the sentence
221
222
 :param key:
223
 :param value:
:return:
224
225
 if not self._annotations:
226
 self._annotations = {}
self._annotations[key] = value
227
228
 def as_sexpr(self) -> SExpression:
 raise NotImplementedError(f"type = {type(self)} // {self}")
229
230
231
 @property
232
 def arguments(self) -> List[Any]:
 raise NotImplementedError(f"type = {type(self)} // {self}")
233
234
235
236
237
238
239
240
241
```

```
242
243
244
245
246
```

```
annotations: Dict[str, Any] property
```

Annotations for the sentence.

Annotations are always logically silent, but can be used to store metadata or other information.

#### **Returns:**

Туре	Description		
Dict[str, Any]			

add\_annotation(key, value)

Add an annotation to the sentence

#### **Parameters:**

Name	Туре	Description	Default
key	str		required
value	Any		required

#### **Returns:**

Туре	Description

```
def add_annotation(self, key: str, value: Any):

"""

Add an annotation to the sentence

:param key:
:param value:
:return:
:return:
:""

if not self._annotations = {}
self._annotations[key] = value
```

# 2.3.4 Term

Bases: Sentence

An atomic part of a sentence.

A ground term is a term with no variables:

```
>>> t = Term('FriendOf', 'Alice', 'Bob')
>>> t
FriendOf(Alice, Bob)
>>> t.values
('Alice', 'Bob')
>>> t.is_ground
True
```

# Keyword argument based initialization is also supported:

```
>>> t = Term('FriendOf', dict(about='Alice', friend='Bob'))
>>> t.values
('Alice', 'Bob')
>>> t.positional
False
```

# Mappings:

• Corresponds to AtomicSentence in Common Logic

Sour	ce code in src/typedlogic/dat	camodel.py 💙		

```
270
 class Term(Sentence):
271
272
 An atomic part of a sentence.
273
 A ground term is a term with no variables:
274
275
 >>> t = Term('FriendOf', 'Alice', 'Bob')
276
277
 FriendOf(Alice, Bob)
278
 >>> t.values
279
 ('Alice', 'Bob')
 >>> t.is_ground
280
 True
281
282
 Keyword argument based initialization is also supported:
283
 >>> t = Term('FriendOf', dict(about='Alice', friend='Bob'))
>>> t.values
284
285
 ('Alice', 'Bob')
>>> t.positional
286
287
 False
288
289
 Mappings:
290
 - Corresponds to AtomicSentence in Common Logic
291
292
293
 def __init__(self, predicate: str, *args, **kwargs):
294
 self.predicate = predicate
295
 if not args:
 self.positional = None
296
 bindings = {}
elif len(args) == 1 and isinstance(args[0], dict):
bindings = args[0]
self.positional = False
297
298
299
300
 else:
301
 bindings = \{f'arg\{i\}': arg for i, arg in enumerate(args)\}
 self.positional = True
self.bindings = bindings
302
303
 self._annotations = kwargs
304
305
 @property
 def is_constant(self):
306
307
 :return: True if the term is a constant (zero arguments)
308
309
 return not self.bindings
310
311
 @property
312
 def is_ground(self):
313
314
 :return: True if none of the arguments are variables
315
 return not any(isinstance(v, Variable) for v in self.bindings.values())
316
317
 @property
318
 def values(self) -> Tuple[Any, ...]:
319
 Representation of the arguments of the term as a fixed-position tuples
320
321
322
 return tuple([v for v in self.bindings.values()])
323
324
325
 def variables(self) -> List[Variable]:
326
 :return: All of the arguments that are variables
327
328
 return [v for v in self.bindings.values() if isinstance(v, Variable)]
329
 @property
330
 def variable_names(self) -> List[str]:
 return [v.name for v in self.bindings.values() if isinstance(v, Variable)]
331
332
333
 def make_keyword_indexed(self, keywords: List[str]):
334
335
 Convert positional arguments to keyword arguments
336
 if self.positional:
337
 self.bindings = \{k: \ v \ for \ k, \ v \ in \ zip(keywords, \ self.bindings.values(), \ strict=False)\} \\ self.positional = False
338
339
340
 def __repr__(self):
 if not self.bindings:
341
 return f'{self.predicate}'
342
 elif self.positional:
343
 \label{eq:continuous} return \ f'\{self.predicate\}(\{",\ ".join(f"\{v\}"\ for\ v\ in\ self.bindings.values())\})'
344
345
 return \ f'\{self.predicate\}(\{",\ ".join(f"\{v\}" \ for \ k,\ v\ in \ self.bindings.items())\})'
346
347
 def __eq__(self, other):
 # return isinstance(other, Term) and self.predicate == other.predicate and self.bindings == other.bindings
348
 return isinstance(other, Term) and self.predicate == other.predicate and self.values == other.values
349
350
 def __hash__(self):
 return hash((self.predicate, tuple(self.values)))
351
352
 def as_sexpr(self) -> SExpression:
353
 return \ [self.predicate] \ + \ [as_sexpr(v) \ for \ v \ in \ self.bindings.values()]
354
```