py-typedlogic

None

None

None

Table of contents

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 run.py stdout Semantics

```python
# links.py
from pydantic import BaseModel
from typedlogic import FactMixin, gen2
from typedlogic.decorators import axiom

ID = str

class Link(BaseModel, FactMixin):
    """A link between two entities"""
    source: ID
    target: ID

class Path(BaseModel, FactMixin):
    """An N-hop path between two entities"""
    source: ID
    target: ID
    hops: int

@axiom
def path_from_link(x: ID, y: ID):
    """If there is a link from x to y, there is a path from x to y"""
    assert Link(source=x, target=y) >> Path(source=x, target=y, hops=1)

@axiom
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)) >>
            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
from links import Link
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='OR', hops=1)
Path(source='OR', target='WA', hops=1)
Path(source='CA', target='WA', hops=2)
```

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

`typedlogic convert links.py -t fol`

Output:

```
∀[x:ID y:ID]. Link(x, y) → Path(x, y, 1)
∀[x:ID y:ID z:ID d1:int d2:int]. Path(x, y, d1) ∧ Path(y, z, d2) → Path(x, z, d1+d2)
¬∃[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
- Integration with multiple solvers and logic engines, including Z3 and Souffle
- Compatible with popular Python validation libraries like Pydantic

1.2 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 CLI without installing the package globally:

```
pipx run typedlogic --help
```

1.3 Define predicates using Pythonic idioms

Inherit from one of the TypedLogic base models to add semantics to your data model. For Pydantic:

```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:

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

1.4 Specify logical axioms directly in Python

Once you have defined your data model predicates, you can specify logical axioms directly in Python:

```python
from typedlogic.decorators import axiom

@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

Use any of the existing solvers to perform reasoning:

```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)
```

outputs:

```
Path(source='CA', target='OR')
Path(source='OR', 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
- ProbLog

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
- 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
    species: str

class OwnsPet(BaseModel, FactMixin):
    person: PersonID
    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 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:

```
class SameOwner(BaseModel, FactMixin):
    pet1: PetID
    pet2: PetID

from typedlogic.decorators import axiom

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

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.

Overview

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:

```
    >>> from typedlogic import Term, Forall, Implies
    >>> x = Variable('x')
    >>> y = Variable('y')
    >>> pdef = PredicateDefinition(predicate='FriendOf',
    ...                            arguments={'x': 'str', 'y': 'str'}),
    >>> theory = Theory(
    ...     name="My theory",
    ...     predicate_definitions=[pdef],
    ... )
    >>> s = Forall([x, y],
    ...            Implies(Term('friend_of', x, y),
    ...                    Term('friend_of', y, x)))
    >>> theory.add(s)
```

Classes

2.3.1 `PredicateDefinition` `dataclass`

Defines the name and arguments of a predicate.

Example:

```
>>> 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",
...     type_definitions={'Person': 'str'},
...     predicate_definitions=[pdef],
... )
```

Model:

```
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.

    Example:

        >>> 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",
        ...     type_definitions={'Person': 'str'},
        ...     predicate_definitions=[pdef],
        ... )

    Model:

    ```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):
                typ = typ.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:
        """
        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	Type	Description	Default
`predicate_class`			*required*

Returns:

Type	Description
`PredicateDefinition`	

Source code in `src/typedlogic/datamodel.py`

```
@classmethod
def from_class(cls, python_class: Type) -> "PredicateDefinition":
    """
    Create a predicate definition from a python class

    :param predicate_class:
    :return:
    """
    return PredicateDefinition(
        predicate=python_class.__name__,
        arguments={k: v for k, v in python_class.__annotations__.items()},
    )
```

2.3.2 `Variable` `dataclass`

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')
>>> ya = Variable('ya', domain='int')
>>> s = Forall([x, y, z],
...            Implies(And(Term('ParentOf', x, y),
...                        Term('Age', x, xa),
...                        Term('Age', y, ya)),
...                    Term('OlderThan', x, y)))

The domains should be either base types or defined types in the theory's `type_definitions` attribute.
```

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')
        >>> ya = Variable('ya', domain='int')
        >>> s = Forall([x, y, z],
        ...            Implies(And(Term('ParentOf', x, y),
        ...                        Term('Age', x, xa),
        ...                        Term('Age', y, ya)),
        ...                    Term('OlderThan', x, y)))

        The domains should be either base types or defined types in the theory's `type_definitions` attribute.

    """

    name: str
    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`

```python
class Sentence(ABC):
    """
    Base class for logical sentences.

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

    Model:

    ```mermaid
    classDiagram
    Sentence <|-- Term
    Sentence <|-- BooleanSentence
    Sentence <|-- QuantifiedSentence
    Sentence <|-- Extension
    ```


    """

    def __init__(self):
        self._annotations = {}

    def __and__(self, other):
        return And(self, other)

    def __or__(self, other):
        return Or(self, other)

    def __invert__(self):
        return Not(self)

    def __sub__(self):
        return NegationAsFailure(self)

    def __rshift__(self, other):
        return Implies(self, other)

    def __lshift__(self, other):
        return Implied(self, other)

    def __xor__(self, other):
        return Xor(self, other)

    def iff(self, other):
        return Iff(self, other)

    def __lt__(self, other):
        return Term(operator.lt.__name__, self, other)

    def __le__(self, other):
        return Term(operator.le.__name__, self, other)

    def __gt__(self, other):
        return Term(operator.gt.__name__, self, other)

    def __ge__(self, other):
        return Term(operator.ge.__name__, self, other)

    def __add__(self, other):
        return Term(operator.add.__name__, self, other)

    @property
    def annotations(self) -> Dict[str, Any]:
        """
        Annotations for the sentence.

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

        :return:
        """
        return self._annotations or {}

    def add_annotation(self, key: str, value: Any):
        """
        Add an annotation to the sentence

        :param key:
        :param value:
        :return:
        """
        if not self._annotations:
            self._annotations = {}
        self._annotations[key] = value

    def as_sexpr(self) -> SExpression:
        raise NotImplementedError(f"type = {type(self)} // {self}")

    @property
    def arguments(self) -> List[Any]:
        raise NotImplementedError(f"type = {type(self)} // {self}")
```

`annotations` `property`

Annotations for the sentence.

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

Returns:

Type	Description
`Dict[str, Any]`	

`add_annotation(key, value)`

Add an annotation to the sentence

Parameters:

Name	Type	Description	Default
`key`	`str`		*required*
`value`	`Any`		*required*

Returns:

Type	Description

Source code in `src/typedlogic/datamodel.py`

```
def add_annotation(self, key: str, value: Any):
    """
    Add an annotation to the sentence

    :param key:
    :param value:
    :return:
    """
    if not self._annotations:
        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

Source code in `src/typedlogic/datamodel.py`

```python
class Term(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
    """

    def __init__(self, predicate: str, *args, **kwargs):
        self.predicate = predicate
        if not args:
            self.positional = None
            bindings = {}
        elif len(args) == 1 and isinstance(args[0], dict):
            bindings = args[0]
            self.positional = False
        else:
            bindings = {f"arg{i}": arg for i, arg in enumerate(args)}
            self.positional = True
        self.bindings = bindings
        self._annotations = kwargs

    @property
    def is_constant(self):
        """
        :return: True if the term is a constant (zero arguments)
        """
        return not self.bindings

    @property
    def is_ground(self):
        """
        :return: True if none of the arguments are variables
        """
        return not any(isinstance(v, Variable) for v in self.bindings.values())

    @property
    def values(self) -> Tuple[Any, ...]:
        """
        Representation of the arguments of the term as a fixed-position tuples
        :return:
        """
        return tuple([v for v in self.bindings.values()])

    @property
    def variables(self) -> List[Variable]:
        """
        :return: All of the arguments that are variables
        """
        return [v for v in self.bindings.values() if isinstance(v, Variable)]

    @property
    def variable_names(self) -> List[str]:
        return [v.name for v in self.bindings.values() if isinstance(v, Variable)]

    def make_keyword_indexed(self, keywords: List[str]):
        """
        Convert positional arguments to keyword arguments
        """
        if self.positional:
            self.bindings = {k: v for k, v in zip(keywords, self.bindings.values(), strict=False)}
            self.positional = False

    def __repr__(self):
        if not self.bindings:
            return f"{self.predicate}"
        elif self.positional:
            return f'{self.predicate}({", ".join(f"{v}" for v in self.bindings.values())})'
        else:
            return f'{self.predicate}({", ".join(f"{v}" for k, v in self.bindings.items())})'

    def __eq__(self, other):
        # return isinstance(other, Term) and self.predicate == other.predicate and self.bindings == other.bindings
        return isinstance(other, Term) and self.predicate == other.predicate and self.values == other.values

    def __hash__(self):
        return hash((self.predicate, tuple(self.values)))

    def as_sexpr(self) -> SExpression:
        return [self.predicate] + [as_sexpr(v) for v in self.bindings.values()]
```