lambda_calculus

Eric Niklas Wolf

Apr 10, 2024

CONTENTS

Welcome to lambda_calculus`s documentation.

This project implements basic operations of the Lambda calculus as a python package and contains helpers to define custom ones.

It is intended to be used for educational purposes and is not optimized for speed. Furthermore, it expects all terms to be finite, which means the absence of cycles. `RecursionError` may be raised when using an infinite term or the evaluation is too complex.

CHAPTER

ONE

INSTALLATION

Overview over the most common ways of installing this project.

1.1 Installation from PyPI

If you just want to get started or receive updates you can use the package available on PyPI and install it with the following command:

```
python3 -m pip install lambda-calculus
```

Sematic Versioning is is attempted to be adhered to.

1.2 Installation from source

This project adheres to PEP 517 and can be build using build:

```
python3 -m build
```

The resulting wheel should be platform and machine independent because this is a pure python project.

CHAPTER

TWO

PACKAGE API

2.1 Package terms

2.1.1 Module abc

Predefined Variables for all ASCII letters

2.1.2 Module logic

Implementations of boolean values and logical operators

lambda_calculus.terms.logic.**TRUE**: Final = **Abstraction(bound='x', body=Abstraction(bound='y', body=Variable(name='x')))**

Term representing True.

lambda_calculus.terms.logic.**FALSE**: Final = **Abstraction(bound='x', body=Abstraction(bound='y', body=Variable(name='y')))**

Term representing False

lambda_calculus.terms.logic.**AND**: Final = **Abstraction(bound='p', body=Abstraction(bound='q', body=Application(abstraction=Application(abstraction=Variable(name='p'), argument=Variable(name='q')), argument=Variable(name='p'))))**

Term implementing logical conjunction between its two arguments.

lambda_calculus.terms.logic.**OR**: Final = **Abstraction(bound='p', body=Abstraction(bound='q', body=Application(abstraction=Application(abstraction=Variable(name='p'), argument=Variable(name='p')), argument=Variable(name='q'))))**

Term implementing logical disjunction between its two arguments.

lambda_calculus.terms.logic.**NOT**: Final = **Abstraction(bound='p', body=Application(abstraction=Application(abstraction=Variable(name='p'), argument=Abstraction(bound='x', body=Abstraction(bound='y', body=Variable(name='y')))), argument=Abstraction(bound='x', body=Abstraction(bound='y', body=Variable(name='x')))))**

Term performing logical negation of its argument.

lambda_calculus.terms.logic.**IF_THEN_ELSE**: Final = **Abstraction(bound='p', body=Abstraction(bound='a', body=Abstraction(bound='b', body=Application(abstraction=Application(abstraction=Variable(name='p'), argument=Variable(name='a')), argument=Variable(name='b')))))**

Term evaluating to its second argument if its first argument is TRUE or its third argument otherwise.

2.1.3 Module arithmetic

Implementations of natural numbers and arithmetic operators

lambda_calculus.terms.arithmetic.**ISZERO**: Final = Abstraction(bound='n', body=Application(abstraction=Application(abstraction=Variable(name='n'), argument=Abstraction(bound='x', body=Abstraction(bound='x', body=Abstraction(bound='y', body=Variable(name='y'))))), argument=Abstraction(bound='x', body=Abstraction(bound='y', body=Variable(name='x')))))

Term which evaluates to *lambda_calculus.terms.logic.TRUE* if its argument is zero, *lambda_calculus.terms.logic.FALSE* otherwise

lambda_calculus.terms.arithmetic.**SUCCESSOR**: Final = Abstraction(bound='n', body=Abstraction(bound='f', body=Abstraction(bound='x', body=Application(abstraction=Variable(name='f'), argument=Application(abstraction=Application(abstraction=Variable(name='n'), argument=Variable(name='f')), argument=Variable(name='x'))))))

Term evaluating to its argument incremented by one.

lambda_calculus.terms.arithmetic.**PREDECESSOR**: Final = Abstraction(bound='n', body=Abstraction(bound='f', body=Abstraction(bound='x', body=Application(abstraction=Application(abstraction=Application(abstraction=Variable(name='n'), argument=Abstraction(bound='g', body=Abstraction(bound='h', body=Application(abstraction=Variable(name='h'), argument=Application(abstraction=Variable(name='g'), argument=Variable(name='f'))))))), argument=Abstraction(bound='u', body=Variable(name='x'))), argument=Abstraction(bound='u', body=Variable(name='u'))))))

Term ealuating to its argument decremented by one.

lambda_calculus.terms.arithmetic.**ADD**: Final = Abstraction(bound='m', body=Abstraction(bound='n', body=Abstraction(bound='f', body=Abstraction(bound='x', body=Application(abstraction=Application(abstraction=Variable(name='m'), argument=Variable(name='f')), argument=Application(abstraction=Application(abstraction=Variable(name='n'), argument=Variable(name='f')), argument=Variable(name='x')))))))

Term evaluating to the sum of its two arguments.

lambda_calculus.terms.arithmetic.**SUBTRACT**: Final = Abstraction(bound='m', body=Abstraction(bound='n', body=Application(abstraction=Application(abstraction=Variable(name='n'), argument=Abstraction(bound='n', body=Abstraction(bound='f', body=Abstraction(bound='x', body=Application(abstraction=Application(abstraction=Application(abstraction=Variable(name='n'), argument=Abstraction(bound='g', body=Abstraction(bound='h', body=Application(abstraction=Variable(name='h'), argument=Application(abstraction=Variable(name='g'), argument=Variable(name='f'))))))), argument=Abstraction(bound='u', body=Variable(name='x'))), argument=Abstraction(bound='u', body=Variable(name='u')))))))), argument=Variable(name='m'))))

Term evaluating to the difference of its two arguments.

lambda_calculus.terms.arithmetic.**MULTIPLY**: Final = Abstraction(bound='m', body=Abstraction(bound='n', body=Abstraction(bound='f', body=Application(abstraction=Variable(name='m'), argument=Application(abstraction=Variable(name='n'), argument=Variable(name='f'))))))

Term evaluating to the product of its two arguments.

lambda_calculus.terms.arithmetic.**POWER**: Final = Abstraction(bound='b', body=Abstraction(bound='e', body=Application(abstraction=Variable(name='e'), argument=Variable(name='b'))))

Term evaluating to its first argument to the power of its second argument.

lambda_calculus.terms.arithmetic.**number**(*n: int*) → *Abstraction*[str]

Encode a number as a lambda term.

Parameters
n – number to encode

Raises
ValueError – If the number is negative

Returns
requested term

2.1.4 Module pairs

Implementation of pairs

lambda_calculus.terms.pairs.**PAIR**: Final = Abstraction(bound='x', body=Abstraction(bound='y', body=Abstraction(bound='f', body=Application(abstraction=Application(abstraction=Variable(name='f'), argument=Variable(name='x')), argument=Variable(name='y')))))

Term evaluating to a ordered pair of its two arguments.

lambda_calculus.terms.pairs.**FIRST**: Final = Abstraction(bound='p', body=Application(abstraction=Variable(name='p'), argument=Abstraction(bound='x', body=Abstraction(bound='y', body=Variable(name='x')))))

Term evaluating to the first value in its argument.

lambda_calculus.terms.pairs.**SECOND**: Final = Abstraction(bound='p', body=Application(abstraction=Variable(name='p'), argument=Abstraction(bound='x', body=Abstraction(bound='y', body=Variable(name='y')))))

Term evaluating to the second value in its argument.

lambda_calculus.terms.pairs.**NIL**: Final = Abstraction(bound='x', body=Abstraction(bound='x', body=Abstraction(bound='y', body=Variable(name='x'))))

Special Term encoding an empty pair.

lambda_calculus.terms.pairs.**NULL**: Final = Abstraction(bound='p', body=Application(abstraction=Variable(name='p'), argument=Abstraction(bound='x', body=Abstraction(bound='y', body=Abstraction(bound='x', body=Abstraction(bound='y', body=Variable(name='y')))))))

Term evaluating to logic.TRUE if its argument is NIL, logic.FALSE otherwise.

2.1.5 Module combinators

Common combinators

lambda_calculus.terms.combinators.**Y**: Final = **Abstraction(bound='g', body=Application(abstraction=Abstraction(bound='x', body=Application(abstraction=Variable(name='g'), argument=Application(abstraction=Variable(name='x'), argument=Variable(name='x')))), argument=Abstraction(bound='x', body=Application(abstraction=Variable(name='g'), argument=Application(abstraction=Variable(name='x'), argument=Variable(name='x'))))))**

Y combinator used to define recursive terms.

lambda_calculus.terms.combinators.**S**: Final = **Abstraction(bound='x', body=Abstraction(bound='y', body=Abstraction(bound='z', body=Application(abstraction=Application(abstraction=Variable(name='x'), argument=Variable(name='z')), argument=Application(abstraction=Variable(name='y'), argument=Variable(name='z'))))))**

S combinator of the SKI combinator calculus.

lambda_calculus.terms.combinators.**K**: Final = **Abstraction(bound='x', body=Abstraction(bound='y', body=Variable(name='x')))**

K combinator of the SKI combinator calculus.

lambda_calculus.terms.combinators.**I**: Final = **Abstraction(bound='x', body=Variable(name='x'))**

I combinator of the SKI combinator calculus.

lambda_calculus.terms.combinators.**B**: Final = **Abstraction(bound='x', body=Abstraction(bound='y', body=Abstraction(bound='z', body=Application(abstraction=Variable(name='x'), argument=Application(abstraction=Variable(name='y'), argument=Variable(name='z'))))))**

B combinator of the BCKW combinator calculus.

lambda_calculus.terms.combinators.**C**: Final = **Abstraction(bound='x', body=Abstraction(bound='y', body=Abstraction(bound='z', body=Application(abstraction=Application(abstraction=Variable(name='x'), argument=Variable(name='z')), argument=Variable(name='y')))))**

C combinator of the BCKW combinator calculus.

lambda_calculus.terms.combinators.**W**: Final = **Abstraction(bound='x', body=Abstraction(bound='y', body=Application(abstraction=Application(abstraction=Variable(name='x'), argument=Variable(name='y')), argument=Variable(name='y'))))**

W combinator of the BCKW combinator calculus.

lambda_calculus.terms.combinators.**DELTA**: Final = **Abstraction(bound='x', body=Application(abstraction=Variable(name='x'), argument=Variable(name='x')))**

Term applying its argument to itself.

lambda_calculus.terms.combinators.**OMEGA**: Final = **Application(abstraction=Abstraction(bound='x', body=Application(abstraction=Variable(name='x'), argument=Variable(name='x'))), argument=Abstraction(bound='x', body=Application(abstraction=Variable(name='x'), argument=Variable(name='x'))))**

Smallest term with no beta normal form.

Lambda Terms

class lambda_calculus.terms.**Term**

Bases: Iterable[Term[V]]

ABC for Lambda terms.

Type Variables:

V: represents the type of variables used in terms

__iter__() → Iterator[*Term*[V]]

Returns
Iterator over all subterms

abstract __str__() → str

Create a string representation.

Returns
lambda term string

abstract free_variables() → Set[V]

Calculate the free variables of this Term.

Returns
variables not bound by an abstraction

abstract bound_variables() → Set[V]

Calculate the bound variables of this Term.

Returns
variables bound by an abstraction

abstract is_beta_normal_form() → bool

Check if this Term is in beta-normal form.

Returns
if no beta reductions can be performed

abstract accept(*visitor:* Visitor*[T, V]*) → T

Accept a visitor by calling his corresponding method.

Parameters
visitor – Visitor to accept

Returns
value returned by the visitors corresponding method

abstract(**variables: V*) → *Abstraction*[V]

Create an Abstraction binding multiple variables.

Parameters
variables – Variables to bind, from first to last

Returns
requested Abstraction term

apply_to(**arguments:* Term*[V]*) → *Application*[V]

Create an Application applying self to multiple arguments.

Parameters
arguments – arguments to apply to, from first to last

Returns
requested Application term

substitute(*variable: V*, *value:* Term*[V]*) → *Term*[V]

Substitute a free variable with a Term.

Parameters

- **variable** – Variable to substitute
- **value** – Value to be substituted

Raises
errors.CollisionError – If substitution would bind free variables

Returns
new term

is_combinator() → bool

Check if this Term has no free variables.

Returns
If there are no free variables

final class lambda_calculus.terms.**Variable**(*name: V*)

Bases: *Term*[V]

Term consisting of a Variable

Parameters
name – Name of the Variable

classmethod with_valid_name(*name: V*) → *Variable*[V]

Create an instance with a valid name.

Parameters
name – Name of the Variable

Raises
ValueError – If the name would conflict with string representations

Returns
requested Variable term

__str__() → str

Create a string representation.

Returns
variable name

free_variables() → Set[V]

Calculate the free variables of this Term.

Returns
variables not bound by an abstraction

bound_variables() → Set[V]

Calculate the bound variables of this Term.

Returns
variables bound by an abstraction

`is_beta_normal_form`() → bool

Check if this Term is in beta-normal form.

Returns
if no beta reductions can be performed

`accept`(*visitor:* Visitor*[T, V]*) → T

Accept a visitor by calling visitors.Visitor.visit_variable.

Parameters
visitor – Visitor to accept

Returns
value returned by visitors.Visitor.visit_variable

`final class` lambda_calculus.terms.**Abstraction**(*bound: V*, *body:* Term*[V]*)

Bases: *Term*[V]

Term consisting of a lambda abstraction.

Parameters

- **bound** – variable to be bound by this abstraction
- **body** – term to be abstracted

`classmethod curried`(*variables: Sequence[V]*, *body:* Term*[V]*) → *Abstraction*[V]

Create an Abstraction binding multiple variables.

Parameters

- **variables** – variables to be bound, from first to last
- **body** – term to be abstracted

Raises
ValueError – If no variables are passed

Returns
requested Abstraction term

`__str__`() → str

Create a string representation.

Returns
({bound}.{body})

`free_variables`() → Set[V]

Calculate the free variables of this Term.

Returns
variables not bound by an abstraction

`bound_variables`() → Set[V]

Calculate the free variables of this Term.

Returns
variables not bound by an abstraction

`is_beta_normal_form`() → bool

Check if this Term is in beta-normal form.

Returns
if no beta reductions can be performed

alpha_conversion(*new: V*) → *Abstraction*[V]

Rename the bound variable

Parameters
new – new variable to bind

Raises
errors.CollisionError – If the new variable is a free variable

Returns
new term

eta_reduction() → *Term*[V]

Remove a useless abstraction.

Raises
ValueError – If abstraction is not useless

Returns
new term

accept(*visitor:* Visitor*[T, V]*) → T

Accept a visitor by calling visitors.Visitor.visit_abstraction.

Parameters
visitor – Visitor to accept

Returns
value returned by visitors.Visitor.visit_abstraction

replace(*, *bound: Optional[V] = None*, *body: Optional[*Term*[V]] = None*) → *Abstraction*[V]

Return a copy with specific attributes replaced.

Parameters

- **bound** – new value for bound variable, defaults to current
- **body** – new value for body, defaults to current

Returns
new term

final class lambda_calculus.terms.**Application**(*abstraction:* Term*[V]*, *argument:* Term*[V]*)

Bases: *Term*[V]

Term consisting of an application.

Parameters

- **abstraction** – abstraction to be applied
- **argument** – argument which to apply the abstraction to

classmethod with_arguments(*abstraction:* Term*[V]*, *arguments: Sequence[*Term*[V]]*) → *Application*[V]

Create an Application applying the abstraction to multiple arguments.

Parameters

- **abstraction** – abstraction to be applied
- **arguments** – arguments which to apply the abstraction to, from first to last

Raises
ValueError – If no arguments are passed

Returns
requested Application term

__str__() → str

Create a string representation.

Returns
({abstraction} {argument})

free_variables() → Set[V]

Calculate the free variables of this Term.

Returns
variables not bound by an abstraction

bound_variables() → Set[V]

Calculate the free variables of this Term.

Returns
variables not bound by an abstraction

is_redex() → bool

Check if this term can be reduced.

Returns
If a beta reduction can be performed

is_beta_normal_form() → bool

Check if this Term is in beta-normal form.

Returns
if no beta reductions can be performed

beta_reduction() → *Term*[V]

Remove the abstraction.

Raises
ValueError – If this term can not be reduced

Returns
new term

accept(*visitor:* Visitor*[T, V]*) → T

Accept a visitor by calling visitors.Visitor.visit_application.

Parameters
visitor – Visitor to accept

Returns
value returned by visitors.Visitor.visit_application

replace(*, *abstraction: Optional*[Term*[V]] = None*, *argument: Optional*[Term*[V]] = None*) → *Application*[V]

Return a copy with specific attributes replaced.

Parameters

- **abstraction** – abstraction to be applied, defaults to current
- **argument** – argument which to apply the abstraction to, defaults to current

Returns
new term

2.2 Package visitors

2.2.1 Package substitution

Module checked

Substitutions checking if the substitutions are valid

final class lambda_calculus.visitors.substitution.checked.**CheckedSubstitution**(*variable: V*, *value:* Term*[V]*, *free_variables:* *Set[V]*)

Bases: *Substitution*[V]

Substitution which checks if a free variable gets bound.

Parameters

- **variable** – variable to substitute
- **value** – value which should be substituted
- **free_variables** – free variables which should not be bound

Raises
errors.CollisionError – If a free variable gets bound

classmethod from_substitution(*variable: V*, *value:* Term*[V]*) → *CheckedSubstitution*[V]

Create an instance from the substitution it should perform

Parameters

- **variable** – variable to substitute
- **value** – value which should be substituted

Returns
new instance with free_variables set to the free variables of value

bind_variable(*name: V*) → None

Mark a variable as bound.

Bound variables are not automatically unbound and can be bound multiple times.

Parameters
name – name of the variable

unbind_variable(*name: V*) → None

Mark a variable as not bound.

A variable needs to be unbound multiple times if it was bound multiple times.

Parameters
name – name of the variable

Raises
KeyError – If the variable is not bound

visit_variable(*variable:* Variable*[V]*) → *Term*[V]

Visit a Variable term.

Parameters
variable – variable term to visit

Raises
errors.CollisionError – If the substitution whould bind free variables

Returns
variable term or value which should be substituted

visit_abstraction(*abstraction:* Abstraction*[V]*) → *Abstraction*[V]

Visit an Abstraction term.

Parameters
abstraction – abstraction term to visit

Raises
errors.CollisionError – If a substitution in the body whould bind free variables

Returns
abstraction term or new term with substitutions performed

visit_application(*application:* Application*[V]*) → *Application*[V]

Visit an Application term.

Parameters
appliation – application term to visit

Raises
errors.CollisionError – If a substitution in the abstraction or argument whould bind free variables

Returns
new term with substitutions performed

Module renaming

Substitutions performing automatic alpha conversion

class lambda_calculus.visitors.substitution.renaming.**RenamingSubstitution**

Bases: *DeferrableSubstitution*[V]

ABC for Substitutions which rename bound variables if a free variable gets bound.

abstract prevent_collision(*abstraction:* Abstraction*[V]*) → *Abstraction*[V]

Prevent collisions by renaming bound variables.

Parameters
abstraction – abstraction term which could bind free variables

Returns
abstraction term which does not bind free variables

final trace() → *TracingDecorator*[V]

Create a new visitor which yields when an alpha conversion occurs.

Returns
new visitor wrapping this instance

final visit_variable(*variable:* Variable*[V]*) → *Term*[V]

Visit a Variable term.

Parameters
variable – variable term to visit

Returns
variable term or value which should be substituted

final defer_abstraction(*abstraction:* Abstraction*[V]*) → tuple[*lambda_calculus.terms.Abstraction*[V], Optional[*lambda_calculus.visitors.substitution.renaming.RenamingSubstitution*[V]]]

Visit an Abstraction term.

Parameters
abstraction – abstraction term to visit

Returns
tuple containing an abstraction term not binding free variables and this visitor to be used for visiting its body if variable is not bound

final defer_application(*application:* Application*[V]*) → tuple[*lambda_calculus.terms.Application*[V], *lambda_calculus.visitors.substitution.renaming.RenamingSubstitution*[V], *lambda_calculus.visitors.substitution.renaming.RenamingSubstitution*[V]]

Visit an Application term.

Parameters
application – application term to visit

Returns
tuple containing the application term and this visitor to be used for visiting its abstraction and argument

final class lambda_calculus.visitors.substitution.renaming.**TracingDecorator**(*substitution:* RenamingSubstitution*[V]*)

Bases: *Visitor*[Generator[terms.Term[V], None, terms.Term[V]], V]

Visitor which transforms a *RenamingSubstitution* into an Generator which yields after performing an alpha conversion and returns the term with substitutions.

Parameters
substitution – instance to wrap

visit_variable(*variable:* Variable*[V]*) → Generator[*Variable*[V], None, *Term*[V]]

Visit a Variable term.

Parameters
variable – variable term to visit

Returns
empty Generator returning the result of *RenamingSubstitution.visit_variable()*

visit_abstraction(*abstraction:* Abstraction*[V]*) → Generator[*Abstraction*[V], None, *Abstraction*[V]]

Visit an Abstraction term

Parameters
abstraction – abstraction term to visit

Returns
Generator yielding alpha conversions and returning the term with substitutions

visit_application(*application:* Application*[V]*) → Generator[*Application*[V], None, *Application*[V]]

Visit an Application term

Parameters
appliation – application term to visit

Returns
Generator yielding alpha conversions and returning the term with substitutions

final class lambda_calculus.visitors.substitution.renaming.**CountingSubstitution**(*variable: str*, *value:* Term*[str]*, *free_variables: Set[str]*)

Bases: *RenamingSubstitution*[str]

Substitution which renames bound variables if a free variable gets bound by appending a number.

Parameters

- **variable** – variable to substitute
- **value** – value which should be substituted
- **free_variables** – free variables which should not be bound

classmethod from_substitution(*variable: str*, *value:* Term*[str]*) → *CountingSubstitution*

Create an instance from the substitution it should perform

Parameters

- **variable** – variable to substitute
- **value** – value which should be substituted

Returns
new instance with free_variables set to the free variables of value

prevent_collision(*abstraction:* Abstraction*[str]*) → *Abstraction*[str]

Prevent collisions by appending a number.

Parameters
abstraction – abstraction term which could bind free variables

Returns
abstraction term which does not bind free variables

Module unsafe

Substitutions which dont check if the substitutions are valid

final class lambda_calculus.visitors.substitution.unsafe.**UnsafeSubstitution**(*variable: V*, *value:* Term*[V]*)

Bases: *DeferrableSubstitution*[V]

Substitution which does not check if a free variable gets bound.

Parameters

- **variable** – variable to substitute
- **value** – value which should be substituted

classmethod from_substitution(*variable: V*, *value:* Term*[V]*) → *UnsafeSubstitution*[V]

Create an instance from the substitution it should perform

Parameters

- **variable** – variable to substitute
- **value** – value which should be substituted

Returns
new instance

visit_variable(*variable:* Variable*[V]*) → *Term*[V]

Visit a Variable term.

Parameters
variable – variable term to visit

Returns
variable term or value which should be substituted

defer_abstraction(*abstraction:* Abstraction*[V]*) → tuple[*lambda_calculus.terms.Abstraction*[V], Optional[*lambda_calculus.visitors.substitution.unsafe.UnsafeSubstitution*[V]]]

Visit an Abstraction term.

Parameters
abstraction – abstraction term to visit

Returns
tuple containing the abstraction term and this visitor to be used for visiting its body if variable is not bound

defer_application(*application:* Application*[V]*) → tuple[*lambda_calculus.terms.Application*[V], *lambda_calculus.visitors.substitution.unsafe.UnsafeSubstitution*[V], *lambda_calculus.visitors.substitution.unsafe.UnsafeSubstitution*[V]]

Visit an Application term.

Parameters
application – application term to visit

Returns
tuple containing the application term and this visitor to be used for visiting its abstraction and argument

Visitors for variable substitution

class lambda_calculus.visitors.substitution.**Substitution**

Bases: *Visitor*[terms.Term[V], V]

ABC for Visitors which replace a free Variable with another term.

Type Variables:

V: represents the type of variables used in terms

abstract visit_abstraction(*abstraction:* Abstraction*[V]*) → *Abstraction*[V]

Visit an Abstraction term

The body is not automatically visited.

Parameters
abstraction – abstraction term to visit

Returns
new term with substitutions performed

abstract visit_application(*application:* Application*[V]*) → *Application*[V]

Visit an Application term

The abstraction and argument are not automatically visited.

Parameters
appliation – application term to visit

Returns
new term with substitutions performed

abstract classmethod from_substitution(*variable: V*, *value:* Term*[V]*) → T

Create an instance from the substitution it should perform

Parameters

- **variable** – variable to substitute
- **value** – value which should be substituted

Returns
new instance

class lambda_calculus.visitors.substitution.**DeferrableSubstitution**

Bases: *DeferrableVisitor*[terms.Term[V], V], *Substitution*[V]

ABC for Substitutions which can be performed lazyly.

abstract defer_abstraction(*abstraction:* Abstraction*[V]*) →
tuple[*lambda_calculus.terms.Abstraction*[V],
Optional[*lambda_calculus.visitors.substitution.DeferrableSubstitution*[V]]]

Visit an Abstraction term.

Parameters
abstraction – abstraction term to visit

Returns
tuple containing a new term instance with substitutions performed and a visitor to be used for visiting its body

abstract defer_application(*application:* Application*[V]*) →
tuple[*lambda_calculus.terms.Application*[V],
Optional[*lambda_calculus.visitors.substitution.DeferrableSubstitution*[V]],
Optional[*lambda_calculus.visitors.substitution.DeferrableSubstitution*[V]]]

Visit an Application term.

Parameters
application – application term to visit

Returns
tuple containing a new term instance with substitutions performed and visitors to be used for visiting its abstraction and argument

final visit_abstraction(*abstraction:* Abstraction*[V]*) → *Abstraction*[V]

Visit an Abstraction term

The body is visited after performing substitution.

Parameters
abstraction – abstraction term to visit

Returns
new term instance with substitutions performed

final visit_application(*application:* Application*[V]*) → *Application*[V]

Visit an Application term

The abstraction and argument are visited after performing substitution.

Parameters
appliation – application term to visit

Returns
new term instance with substitutions performed

2.2.2 Module normalisation

Visitor for term normalisation

class lambda_calculus.visitors.normalisation.**Conversion**(*value*)

Bases: Enum

Conversion performed by normalisation

final class lambda_calculus.visitors.normalisation.**BetaNormalisingVisitor**

Bases: *Visitor*[Iterator[tuple[Conversion, *Term*[str]]], str]

Visitor which transforms a term into its beta normal form, yielding intermediate steps until it is reached

No steps are yielded if the term is already in its beta normal form.

Remember that some terms dont thave a beta normal form and can cause infinite recursion.

skip_intermediate(*term:* Term*[str]*) → *Term*[str]

Calculate the beta normal form directly.

Parameters
term – term which should be transformed into ist beta normal form

Returns
new term representing the beta normal form if it exists

visit_variable(*variable:* Variable*[str]*) → Iterator[tuple['Conversion', *lambda_calculus.terms.Term*[str]]]

Visit a Variable term.

Parameters
variable – variable term to visit

Returns
empty Iterator, variables are already in beta normal form

visit_abstraction(*abstraction:* Abstraction*[str]*) → Iterator[tuple['Conversion', *lambda_calculus.terms.Term*[str]]]

Visit an Abstraction term.

Parameters
abstraction – abstraction term to visit

Returns
Iterator yielding steps performed on its body

beta_reduction(*abstraction:* Abstraction*[str]*, *argument:* Term*[str]*) → Generator[tuple['Conversion', *lambda_calculus.terms.Term*[str]], None, *Term*[str]]

Perform beta reduction of an application.

Parameters

- **abstraction** – abstraction of the application
- **argument** – argument of the application

Returns
Generator yielding steps and returning the reduced term

visit_application(*application:* Application*[str]*) → Iterator[tuple['Conversion', *lambda_calculus.terms.Term*[str]]]

Visit an Application term

The abstraction and argument are not automatically visited.

Parameters
application – application term to visit

Returns
steps for performing beta reduction if possible and performed on its result or abstraction and argument

2.2.3 Module walking

Visitor for walking terms

final class lambda_calculus.visitors.walking.**DepthFirstVisitor**

Bases: *BottomUpVisitor*[Iterator[terms.Term[V]], V]

Visitor yielding subterms depth first

Type Variables:

V: represents the type of variables used in terms

visit_variable(*variable:* Variable*[V]*) → Iterator[*Term*[V]]

Visit a Variable term.

Parameters
variable – variable term to visit

Returns
Iterator yielding the term

ascend_abstraction(*abstraction:* Abstraction*[V]*, *body:* *Iterator*[Term*[V]*]) → Iterator[*Term*[V]]

Visit an Abstraction term after visiting its body.

Parameters

- **abstraction** – abstraction term to visit
- **body** – Iterator produced by visiting its body

Returns
term appended to its body Iterator

ascend_application(*application:* Application*[V]*, *abstraction:* *Iterator[*Term*[V]]*, *argument:* *Iterator[*Term*[V]]*) → Iterator[*Term*[V]]

Visit an Application term after visiting its abstraction and argument.

Parameters

- **application** – application term to visit
- **abstraction** – Iterator produced by visiting its abstraction
- **argument** – Iterator produced by visiting its argument

Returns
term appended to its abstraction and argument Iterators

Visitors for performing operations on Terms

class lambda_calculus.visitors.**Visitor**

Bases: ABC, Generic[T, V]

ABC for Visitors visiting Terms.

The visitor is responsible for visiting child terms.

Type Variables:

T: represents the type of the result produced by visiting terms V: represents the type of variables used in terms

final visit(*term:* Term*[V]*) → T

Visit a term

Parameters
term – term to visit

Returns
Result of calling *terms.Term.accept()* with self as argument

abstract visit_variable(*variable:* Variable*[V]*) → T

Visit a Variable term.

Parameters
variable – variable term to visit

Returns
value as required by its type variable

abstract visit_abstraction(*abstraction:* Abstraction*[V]*) → T

Visit an Abstraction term

The body is not automatically visited.

Parameters
abstraction – abstraction term to visit

Returns
value as required by its type variable

abstract visit_application(*application:* Application*[V]*) → T

Visit an Application term

The abstraction and argument are not automatically visited.

Parameters
appliation – application term to visit

Returns
value as required by its type variable

class lambda_calculus.visitors.**BottomUpVisitor**

Bases: *Visitor*[T, V]

ABC for visitors which visit child terms first

Child terms are automatically visited.

final visit_abstraction(*abstraction:* Abstraction*[V]*) → T

Visit an Abstraction term

The body is visited before calling *ascend_abstraction()*.

Parameters
abstraction – abstraction term to visit

Returns
value returned by *ascend_abstraction()*

final visit_application(*application:* Application*[V]*) → T

Visit an Application term

The abstraction and argument are visited before calling *ascend_application()*.

Parameters
application – application term to visit

Returns
value returned by *ascend_application()*

abstract ascend_abstraction(*abstraction:* Abstraction*[V]*, *body: T*) → T

Visit an Abstraction term after visiting its body.

Parameters

- **abstraction** – abstraction term to visit
- **body** – value produced by visiting its body

Returns
value as required by its type variable

abstract ascend_application(*application:* Application*[V]*, *abstraction: T*, *argument: T*) → T

Visit an Application term after visiting its abstraction and argument.

Parameters

- **application** – application term to visit
- **abstraction** – value produced by visiting its abstraction
- **argument** – value produced by visiting its argument

Returns
value as required by its type variable

class lambda_calculus.visitors.**DeferrableVisitor**

Bases: *Visitor*[T, V]

ABC for visitors which can visit terms top down lazyly.

abstract defer_abstraction(*abstraction:* Abstraction*[V]*) → tuple[T, Optional[*lambda_calculus.visitors.DeferrableVisitor*[T, V]]]

Visit an Abstraction term.

Parameters
abstraction – abstraction term to visit

Returns
tuple containing a value as required by its type variable and a visitor to be used for visiting its body

abstract defer_application(*application:* Application*[V]*) → tuple[T, Optional[*lambda_calculus.visitors.DeferrableVisitor*[T, V]], Optional[*lambda_calculus.visitors.DeferrableVisitor*[T, V]]]

Visit an Application term.

Parameters
application – application term to visit

Returns
tuple containing a value as required by its type variable and visitors to be used for visiting its abstraction and argument

2.3 Module errors

Errors raised by Term operations

exception lambda_calculus.errors.**CollisionError**(*message: str*, *collisions: Collection[V]*)

Bases: ValueError, Generic[V]

Exception thrown when a variable already exists, for example as a free variable.

Type Variables:

V: represents the type of variables

Parameters

- **message** – message to be displayed
- **collisions** – variables which already exist

Implementation of the Lambda calculus

final class lambda_calculus.**Variable**(*name: V*)

Bases: *Term*[V]

Term consisting of a Variable

Parameters
name – Name of the Variable

Reference to *terms.Variable* for convenience

final class lambda_calculus.**Abstraction**(*bound: V*, *body:* Term*[V]*)

Bases: *Term*[V]

Term consisting of a lambda abstraction.

Parameters

- **bound** – variable to be bound by this abstraction
- **body** – term to be abstracted

Reference to *terms.Abstraction* for convenience

final class lambda_calculus.**Application**(*abstraction:* Term*[V]*, *argument:* Term*[V]*)

Bases: *Term*[V]

Term consisting of an application.

Parameters

- **abstraction** – abstraction to be applied
- **argument** – argument which to apply the abstraction to

Reference to *terms.Application* for convenience

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

l

INDEX

Symbols

A

B

C

D

E

F

I

K

L

W

Y