Coercion

Release 10.6

The Sage Development Team

Jun 27, 2025

CONTENTS

CHAPTER

ONE

PRELIMINARIES

1.1 What is coercion all about?

The primary goal of coercion is to be able to transparently do arithmetic, comparisons, etc. between elements of distinct sets.

As a concrete example, when one writes $1 + 1/2$ one wants to perform arithmetic on the operands as rational numbers, despite the left being an integer. This makes sense given the obvious and natural inclusion of the integers into the rational numbers. The goal of the coercion system is to facilitate this (and more complicated arithmetic) without having to explicitly map everything over into the same domain, and at the same time being strict enough to not resolve ambiguity or accept nonsense. Here are some examples:

```
sage: 1 + 1/2
3/2
sage: R.<x,y> = ZZ[]
sage: R
Multivariate Polynomial Ring in x, y over Integer Ring
sage: parent(x)
Multivariate Polynomial Ring in x, y over Integer Ring
sage: parent(1/3)
Rational Field
sage: x+1/3
x + 1/3
sage: parent(x+1/3)
Multivariate Polynomial Ring in x, y over Rational Field

sage: GF(5)(1) + CC(I)
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +: 'Finite Field of size 5' and 'Complex␣
↪Field with 53 bits of precision'
```

```
>>> from sage.all import *
>>> Integer(1) + Integer(1)/Integer(2)
3/2
>>> R = ZZ['x, y']; (x, y,) = R._first_ngens(2)
>>> R
Multivariate Polynomial Ring in x, y over Integer Ring
>>> parent(x)
Multivariate Polynomial Ring in x, y over Integer Ring
>>> parent(Integer(1)/Integer(3))
```

(continues on next page)

(continued from previous page)

```
Rational Field
>>> x+Integer(1)/Integer(3)
x + 1/3
>>> parent(x+Integer(1)/Integer(3))
Multivariate Polynomial Ring in x, y over Rational Field

>>> GF(Integer(5))(Integer(1)) + CC(I)
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +: 'Finite Field of size 5' and 'Complex␣
↪Field with 53 bits of precision'
```

1.2 Parents and Elements

Parents are objects in concrete categories, and Elements are their members. Parents are first-class objects. Most things in Sage are either parents or have a parent. Typically whenever one sees the word *Parent* one can think *Set*. Here are some examples:

```
sage: parent(1)
Integer Ring
sage: parent(1) is ZZ
True
sage: ZZ
Integer Ring
sage: parent(1.50000000000000000000000000000000000000)
Real Field with 120 bits of precision
sage: parent(x)
Symbolic Ring
sage: x^sin(x)
x^sin(x)
sage: R.<t> = Qp(5)[]
sage: f = t^3-5; f
(1 + O(5^20))*t^3 + 4*5 + 4*5^2 + 4*5^3 + 4*5^4 + 4*5^5 + 4*5^6 + 4*5^7 + 4*5^8 + 4*5^
↪9 + 4*5^10 + 4*5^11 + 4*5^12 + 4*5^13 + 4*5^14 + 4*5^15 + 4*5^16 + 4*5^17 + 4*5^18␣
↪+ 4*5^19 + 4*5^20 + O(5^21)
sage: parent(f)
Univariate Polynomial Ring in t over 5-adic Field with capped relative precision 20
sage: f = EllipticCurve('37a').lseries().taylor_series(10); f  # abs tol 1e-14
0.997997869801216 + 0.00140712894524925*z - 0.000498127610960097*z^2 + 0.
↪000118835596665956*z^3 - 0.000021590652244270 8*z^4 + (3.20363155418421e-6)*z^5 +␣
↪O(z^6)  # 32-bit
0.997997869801216 + 0.00140712894524925*z - 0.000498127610960097*z^2 + 0.
↪000118835596665956*z^3 - 0.000021590652244270 8*z^4 + (3.20363155418427e-6)*z^5 +␣
↪O(z^6)  # 64-bit
sage: parent(f)
Power Series Ring in z over Complex Field with 53 bits of precision
```

```
>>> from sage.all import *
>>> parent(Integer(1))
Integer Ring
>>> parent(Integer(1)) is ZZ
```

(continues on next page)

(continued from previous page)

```
True
>>> ZZ
Integer Ring
>>> parent(RealNumber('1.500000000000000000000000000000000000'))
Real Field with 120 bits of precision
>>> parent(x)
Symbolic Ring
>>> x**sin(x)
x^sin(x)
>>> R = Qp(Integer(5))['t']; (t,) = R._first_ngens(1)
>>> f = t**Integer(3)-Integer(5); f
(1 + O(5^20))*t^3 + 4*5 + 4*5^2 + 4*5^3 + 4*5^4 + 4*5^5 + 4*5^6 + 4*5^7 + 4*5^8 + 4*5^
↪9 + 4*5^10 + 4*5^11 + 4*5^12 + 4*5^13 + 4*5^14 + 4*5^15 + 4*5^16 + 4*5^17 + 4*5^18␣
↪+ 4*5^19 + 4*5^20 + O(5^21)
>>> parent(f)
Univariate Polynomial Ring in t over 5-adic Field with capped relative precision 20
>>> f = EllipticCurve('37a').lseries().taylor_series(Integer(10)); f  # abs tol 1e-14
0.997997869801216 + 0.00140712894524925*z - 0.000498127610960097*z^2 + 0.
↪000118835596665956*z^3 - 0.00002159065224427 08*z^4 + (3.20363155418421e-6)*z^5 +␣
↪O(z^6)  # 32-bit
0.997997869801216 + 0.00140712894524925*z - 0.000498127610960097*z^2 + 0.
↪000118835596665956*z^3 - 0.00002159065224427 08*z^4 + (3.20363155418427e-6)*z^5 +␣
↪O(z^6)  # 64-bit
>>> parent(f)
Power Series Ring in z over Complex Field with 53 bits of precision
```

There is an important distinction between Parents and types:

```
sage: a = GF(5).random_element()
sage: b = GF(7).random_element()
sage: type(a)
<class 'sage.rings.finite_rings.integer_mod.IntegerMod_int'>
sage: type(b)
<class 'sage.rings.finite_rings.integer_mod.IntegerMod_int'>
sage: type(a) == type(b)
True
sage: parent(a)
Finite Field of size 5
sage: parent(a) == parent(b)
False
```

```
>>> from sage.all import *
>>> a = GF(Integer(5)).random_element()
>>> b = GF(Integer(7)).random_element()
>>> type(a)
<class 'sage.rings.finite_rings.integer_mod.IntegerMod_int'>
>>> type(b)
<class 'sage.rings.finite_rings.integer_mod.IntegerMod_int'>
>>> type(a) == type(b)
True
>>> parent(a)
Finite Field of size 5
```

(continues on next page)

(continued from previous page)

```
>>> parent(a) == parent(b)
False
```

However, non-Sage objects do not really have parents, but we still want to be able to reason with them, so their type is used instead:

```
sage: a = int(10)
sage: parent(a)
<... 'int'>
```

```
>>> from sage.all import *
>>> a = int(Integer(10))
>>> parent(a)
<... 'int'>
```

In fact, under the hood, a special kind of parent "The set of all Python objects of class T" is used in these cases.

Note that parents are **not** always as tight as possible.

```
sage: parent(1/2)
Rational Field
sage: parent(2/1)
Rational Field
```

```
>>> from sage.all import *
>>> parent(Integer(1)/Integer(2))
Rational Field
>>> parent(Integer(2)/Integer(1))
Rational Field
```

1.3 Maps between Parents

Many parents come with maps to and from other parents.

Sage makes a distinction between being able to **convert** between various parents, and **coerce** between them. Conversion is explicit and tries to make sense of an object in the target domain if at all possible. It is invoked by calling:

```
sage: ZZ(5)
5
sage: ZZ(10/5)
2
sage: QQ(10)
10
sage: parent(QQ(10))
Rational Field
sage: a = GF(5)(2); a
2
sage: parent(a)
Finite Field of size 5
sage: parent(ZZ(a))
Integer Ring
sage: GF(71)(1/5)
```

(continues on next page)

(continued from previous page)

```
57
sage: ZZ(1/2)
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

```
>>> from sage.all import *
>>> ZZ(Integer(5))
5
>>> ZZ(Integer(10)/Integer(5))
2
>>> QQ(Integer(10))
10
>>> parent(QQ(Integer(10)))
Rational Field
>>> a = GF(Integer(5))(Integer(2)); a
2
>>> parent(a)
Finite Field of size 5
>>> parent(ZZ(a))
Integer Ring
>>> GF(Integer(71))(Integer(1)/Integer(5))
57
>>> ZZ(Integer(1)/Integer(2))
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

Conversions need not be canonical (they may for example involve a choice of lift) or even make sense mathematically (e.g. constructions of some kind).

```
sage: ZZ("123")
123
sage: ZZ(GF(5)(14))
4
sage: ZZ['x']([4,3,2,1])
x^3 + 2*x^2 + 3*x + 4
sage: a = Qp(5, 10)(1/3); a
2 + 3*5 + 5^2 + 3*5^3 + 5^4 + 3*5^5 + 5^6 + 3*5^7 + 5^8 + 3*5^9 + O(5^10)
sage: ZZ(a)
6510417
```

```
>>> from sage.all import *
>>> ZZ("123")
123
>>> ZZ(GF(Integer(5))(Integer(14)))
4
>>> ZZ['x']([Integer(4),Integer(3),Integer(2),Integer(1)])
x^3 + 2*x^2 + 3*x + 4
>>> a = Qp(Integer(5), Integer(10))(Integer(1)/Integer(3)); a
2 + 3*5 + 5^2 + 3*5^3 + 5^4 + 3*5^5 + 5^6 + 3*5^7 + 5^8 + 3*5^9 + O(5^10)
>>> ZZ(a)
```

(continues on next page)

(continued from previous page)

```
6510417
```

On the other hand, Sage has the notion of a **coercion**, which is a canonical morphism (occasionally up to a conventional choice made by developers) between parents. A coercion from one parent to another **must** be defined on the whole domain, and always succeeds. As it may be invoked implicitly, it should be obvious and natural (in both the mathematically rigorous and colloquial sense of the word). Up to inescapable rounding issues that arise with inexact representations, these coercion morphisms should all commute. In particular, if there are coercion maps $A \to B$ and $B \to A$, then their composites must be the identity maps.

Coercions can be discovered via the `Parent.has_coerce_map_from()` method, and if needed explicitly invoked with the `Parent.coerce()` method:

```
sage: QQ.has_coerce_map_from(ZZ)
True
sage: QQ.has_coerce_map_from(RR)
False
sage: ZZ['x'].has_coerce_map_from(QQ)
False
sage: ZZ['x'].has_coerce_map_from(ZZ)
True
sage: ZZ['x'].coerce(5)
5
sage: ZZ['x'].coerce(5).parent()
Univariate Polynomial Ring in x over Integer Ring
sage: ZZ['x'].coerce(5/1)
Traceback (most recent call last):
...
TypeError: no canonical coercion from Rational Field to Univariate Polynomial Ring in␣
↪x over Integer Ring
```

```
>>> from sage.all import *
>>> QQ.has_coerce_map_from(ZZ)
True
>>> QQ.has_coerce_map_from(RR)
False
>>> ZZ['x'].has_coerce_map_from(QQ)
False
>>> ZZ['x'].has_coerce_map_from(ZZ)
True
>>> ZZ['x'].coerce(Integer(5))
5
>>> ZZ['x'].coerce(Integer(5)).parent()
Univariate Polynomial Ring in x over Integer Ring
>>> ZZ['x'].coerce(Integer(5)/Integer(1))
Traceback (most recent call last):
...
TypeError: no canonical coercion from Rational Field to Univariate Polynomial Ring in␣
↪x over Integer Ring
```

CHAPTER

TWO

BASIC ARITHMETIC RULES

Suppose we want to add two element, a and b, whose parents are A and B respectively. When we type `a+b` then

1. If A `is` B, call a._add_(b)
2. If there is a coercion $\phi : B \to A$, call a._add_(ϕ (b))
3. If there is a coercion $\phi : A \to B$, call ϕ (a)._add_(b)
4. Look for Z such that there is a coercion $\phi_A : A \to Z$ and $\phi_B : B \to Z$, call ϕ_A (a)._add_(ϕ_B (b))

These rules are evaluated in order; therefore if there are coercions in both directions, then the parent of a._add_b is A – the parent of the left-hand operand is used in such cases.

The same rules are used for subtraction, multiplication, and division. This logic is embedded in a coercion model object, which can be obtained and queried.

```
sage: parent(1 + 1/2)
Rational Field
sage: cm = coercion_model; cm
<sage.structure.coerce.CoercionModel object at ...>
sage: cm.explain(ZZ, QQ)
Coercion on left operand via
   Natural morphism:
     From: Integer Ring
     To:   Rational Field
Arithmetic performed after coercions.
Result lives in Rational Field
Rational Field

sage: cm.explain(ZZ['x','y'], QQ['x'])
Coercion on left operand via
   Coercion map:
     From: Multivariate Polynomial Ring in x, y over Integer Ring
     To:   Multivariate Polynomial Ring in x, y over Rational Field
Coercion on right operand via
   Coercion map:
     From: Univariate Polynomial Ring in x over Rational Field
     To:   Multivariate Polynomial Ring in x, y over Rational Field
Arithmetic performed after coercions.
Result lives in Multivariate Polynomial Ring in x, y over Rational Field
Multivariate Polynomial Ring in x, y over Rational Field
```

```
>>> from sage.all import *
>>> parent(Integer(1) + Integer(1)/Integer(2))
Rational Field
>>> cm = coercion_model; cm
<sage.structure.coerce.CoercionModel object at ...>
>>> cm.explain(ZZ, QQ)
Coercion on left operand via
   Natural morphism:
     From: Integer Ring
     To:   Rational Field
Arithmetic performed after coercions.
Result lives in Rational Field
Rational Field

>>> cm.explain(ZZ['x','y'], QQ['x'])
Coercion on left operand via
   Coercion map:
     From: Multivariate Polynomial Ring in x, y over Integer Ring
     To:   Multivariate Polynomial Ring in x, y over Rational Field
Coercion on right operand via
   Coercion map:
     From: Univariate Polynomial Ring in x over Rational Field
     To:   Multivariate Polynomial Ring in x, y over Rational Field
Arithmetic performed after coercions.
Result lives in Multivariate Polynomial Ring in x, y over Rational Field
Multivariate Polynomial Ring in x, y over Rational Field
```

The coercion model can be used directly for any binary operation (callable taking two arguments).

```
sage: cm.bin_op(77, 9, gcd)
1
```

```
>>> from sage.all import *
>>> cm.bin_op(Integer(77), Integer(9), gcd)
1
```

There are also **actions** in the sense that a field K acts on a module over K, or a permutation group acts on a set. These are discovered between steps 1 and 2 above.

```
sage: cm.explain(ZZ['x'], ZZ, operator.mul)
Action discovered.
   Right scalar multiplication by Integer Ring on Univariate Polynomial Ring in x␣
↪over Integer Ring
Result lives in Univariate Polynomial Ring in x over Integer Ring
Univariate Polynomial Ring in x over Integer Ring

sage: cm.explain(ZZ['x'], ZZ, operator.truediv)
Action discovered.
   Right inverse action by Rational Field on Univariate Polynomial Ring in x over␣
↪Integer Ring
   with precomposition on right by Natural morphism:
     From: Integer Ring
     To:   Rational Field
```

(continues on next page)

(continued from previous page)

```
Result lives in Univariate Polynomial Ring in x over Rational Field
Univariate Polynomial Ring in x over Rational Field

sage: f = QQ.coerce_map_from(ZZ)
sage: f(3).parent()
Rational Field
```

```
>>> from sage.all import *
>>> cm.explain(ZZ['x'], ZZ, operator.mul)
Action discovered.
   Right scalar multiplication by Integer Ring on Univariate Polynomial Ring in x␣
↪over Integer Ring
Result lives in Univariate Polynomial Ring in x over Integer Ring
Univariate Polynomial Ring in x over Integer Ring

>>> cm.explain(ZZ['x'], ZZ, operator.truediv)
Action discovered.
   Right inverse action by Rational Field on Univariate Polynomial Ring in x over␣
↪Integer Ring
   with precomposition on right by Natural morphism:
     From: Integer Ring
     To:   Rational Field
Result lives in Univariate Polynomial Ring in x over Rational Field
Univariate Polynomial Ring in x over Rational Field

>>> f = QQ.coerce_map_from(ZZ)
>>> f(Integer(3)).parent()
Rational Field
```

Note that by Issue #14711 Sage's coercion system uses maps with weak references to the domain. Such maps should only be used internally, and so a copy should be used instead (unless one knows what one is doing):

```
sage: QQ._internal_coerce_map_from(int)
(map internal to coercion system -- copy before use)
Native morphism:
  From: Set of Python objects of class 'int'
  To:   Rational Field
sage: copy(QQ._internal_coerce_map_from(int))
Native morphism:
  From: Set of Python objects of class 'int'
  To:   Rational Field
```

```
>>> from sage.all import *
>>> QQ._internal_coerce_map_from(int)
(map internal to coercion system -- copy before use)
Native morphism:
  From: Set of Python objects of class 'int'
  To:   Rational Field
>>> copy(QQ._internal_coerce_map_from(int))
Native morphism:
  From: Set of Python objects of class 'int'
  To:   Rational Field
```

Note that the user-visible method (without underscore) automates this copy:

```
sage: copy(QQ.coerce_map_from(int))
Native morphism:
  From: Set of Python objects of class 'int'
  To:   Rational Field
```

```
>>> from sage.all import *
>>> copy(QQ.coerce_map_from(int))
Native morphism:
  From: Set of Python objects of class 'int'
  To:   Rational Field
```

```
sage: QQ.has_coerce_map_from(RR)
False
sage: QQ['x'].get_action(QQ)
Right scalar multiplication by Rational Field on Univariate Polynomial Ring in x over␣
↪Rational Field
sage: QQ2 = QQ^2
sage: (QQ2).get_action(QQ)
Right scalar multiplication by Rational Field on Vector space of dimension 2 over␣
↪Rational Field
sage: QQ['x'].get_action(RR)
Right scalar multiplication by Real Field with 53 bits of precision on Univariate␣
↪Polynomial Ring in x over Rational Field
```

```
>>> from sage.all import *
>>> QQ.has_coerce_map_from(RR)
False
>>> QQ['x'].get_action(QQ)
Right scalar multiplication by Rational Field on Univariate Polynomial Ring in x over␣
↪Rational Field
>>> QQ2 = QQ**Integer(2)
>>> (QQ2).get_action(QQ)
Right scalar multiplication by Rational Field on Vector space of dimension 2 over␣
↪Rational Field
>>> QQ['x'].get_action(RR)
Right scalar multiplication by Real Field with 53 bits of precision on Univariate␣
↪Polynomial Ring in x over Rational Field
```

CHAPTER

THREE

HOW TO IMPLEMENT

3.1 Methods to implement

- Arithmetic on Elements: `_add_`, `_sub_`, `_mul_`, `_div_`

  This is where the binary arithmetic operators should be implemented. Unlike Python's `__add__`, both operands are *guaranteed* to have the same Parent at this point.

- Coercion for Parents: `_coerce_map_from_`

  Given two parents R and S, `R._coerce_map_from_(S)` is called to determine if there is a coercion $\phi : S \to R$. Note that the function is called on the potential codomain. To indicate that there is no coercion from S to R (self), return `False` or `None`. This is the default behavior. If there is a coercion, return `True` (in which case a morphism using `R._element_constructor_` will be created) or an actual `Morphism` object with S as the domain and R as the codomain.

- Actions for Parents: `_get_action_` or `_rmul_`, `_lmul_`, `_act_on_`, `_acted_upon_`

  Suppose one wants R to act on S. Some examples of this could be $R = \mathbf{Q}$, $S = \mathbf{Q}[x]$ or $R = \mathrm{Gal}(S/\mathbf{Q})$ where S is a number field. There are several ways to implement this:

  - If R is the base of S (as in the first example), simply implement `_rmul_` and/or `_lmul_` on the Elements of S. In this case `r * s` gets handled as `s._rmul_(r)` and `s * r` as `s._lmul_(r)`. The argument to `_rmul_` and `_lmul_` are *guaranteed* to be Elements of the base of S (with coercion happening beforehand if necessary).
  - If R acts on S, one can define the methods `_act_on_` on Elements of R or `_acted_upon_` on Elements of S. In this case `r * s` gets handled as `r._act_on_(s, True)` or `s._acted_upon_(r, False)` and `s * r` as `r._act_on_(s, False)` or `s._acted_upon_(r, True)`. There is no constraint on the type or parents of objects passed to these methods; raise a `TypeError` or `ValueError` if the wrong kind of object is passed in to indicate the action is not appropriate here.
  - If either R acts on S *or* S acts on R, one may implement `R._get_action_` to return an actual *`Action`* object to be used. This is how non-multiplicative actions must be implemented, and is the most powerful and complete way to do things.

  It should be noted that for the first way to work, elements of S are required to be ModuleElements. This requirement is likely to be lifted in the future.

- Element conversion/construction for Parents: use `_element_constructor_` **not** `__call__`

  The `Parent.__call__()` method dispatches to `_element_constructor_`. When someone writes `R(x, ...)`, this is the method that eventually gets called in most cases. See the documentation on the `__call__` method below.

Parents may also call the `self._populate_coercion_lists_` method in their `__init__` functions to pass any callable for use instead of `_element_constructor_`, provide a list of Parents with coercions to self (as an alternative

to implementing `_coerce_map_from_`), provide special construction methods (like `_integer_` for ZZ), etc. This also allows one to specify a single coercion embedding *out* of self (whereas the rest of the coercion functions all specify maps *into* self). There is extensive documentation in the docstring of the `_populate_coercion_lists_` method.

3.2 Example

Sometimes a simple example is worth a thousand words. Here is a minimal example of setting up a simple Ring that handles coercion. (It is easy to imagine much more sophisticated and powerful localizations, but that would obscure the main points being made here.)

```
sage: from sage.structure.richcmp import richcmp
sage: from sage.structure.element import Element

sage: class MyLocalization(Parent):
....:     def __init__(self, primes):
....:         r"""
....:         Localization of `\ZZ` away from primes.
....:         """
....:         Parent.__init__(self, base=ZZ, category=Rings().Commutative())
....:         self._primes = primes
....:         self._populate_coercion_lists_()
....:
....:     def _repr_(self) -> str:
....:         """
....:         How to print ``self``.
....:         """
....:         return "%s localized at %s" % (self.base(), self._primes)
....:
....:     def _element_constructor_(self, x):
....:         """
....:         Make sure ``x`` is a valid member of ``self``, and return the␣
↪constructed element.
....:         """
....:         if isinstance(x, MyLocalizationElement):
....:             x = x._value
....:         else:
....:             x = QQ(x)
....:         for p, e in x.denominator().factor():
....:             if p not in self._primes:
....:                 raise ValueError("not integral at %s" % p)
....:         return self.element_class(self, x)
....:
....:     def _an_element_(self):
....:         return self.element_class(self, 6)
....:
....:     def _coerce_map_from_(self, S):
....:         """
....:         The only things that coerce into this ring are:
....:
....:         - the integer ring
....:
....:         - other localizations away from fewer primes
....:         """
```

(continues on next page)

(continued from previous page)

```
....:         if S is ZZ:
....:             return True
....:         if isinstance(S, MyLocalization):
....:             return all(p in self._primes for p in S._primes)

sage: class MyLocalizationElement(Element):
....:
....:     def __init__(self, parent, x):
....:         Element.__init__(self, parent)
....:         self._value = x
....:
....:     # We are just printing out this way to make it easy to see what's going on␣
↪in the examples.
....:
....:     def _repr_(self) -> str:
....:         return f"LocalElt({self._value})"
....:
....:     # Now define addition, subtraction and multiplication of elements.
....:     # Note that self and right always have the same parent.
....:
....:     def _add_(self, right):
....:         return self.parent()(self._value + right._value)
....:
....:     def _sub_(self, right):
....:         return self.parent()(self._value - right._value)
....:
....:     def _mul_(self, right):
....:         return self.parent()(self._value * right._value)
....:
....:     # The basering was set to ZZ, so c is guaranteed to be in ZZ
....:
....:     def _rmul_(self, c):
....:         return self.parent()(c * self._value)
....:
....:     def _lmul_(self, c):
....:         return self.parent()(self._value * c)
....:
....:     def _richcmp_(self, other, op):
....:         return richcmp(self._value, other._value, op)

sage: MyLocalization.element_class = MyLocalizationElement
```

```
>>> from sage.all import *
>>> from sage.structure.richcmp import richcmp
>>> from sage.structure.element import Element

>>> class MyLocalization(Parent):
...     def __init__(self, primes):
...         r"""
...         Localization of `\ZZ` away from primes.
...         """
...         Parent.__init__(self, base=ZZ, category=Rings().Commutative())
```

(continues on next page)

(continued from previous page)

```
...         self._primes = primes
...         self._populate_coercion_lists_()
....:
>>>     def _repr_(self) -> str:
...         """
...         How to print ``self``.
...         """
...         return "%s localized at %s" % (self.base(), self._primes)
....:
>>>     def _element_constructor_(self, x):
...         """
...         Make sure ``x`` is a valid member of ``self``, and return the constructed␣
↪element.
...         """
...         if isinstance(x, MyLocalizationElement):
...             x = x._value
...         else:
...             x = QQ(x)
...         for p, e in x.denominator().factor():
...             if p not in self._primes:
...                 raise ValueError("not integral at %s" % p)
...         return self.element_class(self, x)
....:
>>>     def _an_element_(self):
...         return self.element_class(self, Integer(6))
....:
>>>     def _coerce_map_from_(self, S):
...         """
...         The only things that coerce into this ring are:
....:
>>>         - the integer ring
....:
>>>         - other localizations away from fewer primes
...         """
...         if S is ZZ:
...             return True
...         if isinstance(S, MyLocalization):
...             return all(p in self._primes for p in S._primes)

>>> class MyLocalizationElement(Element):
....:
>>>     def __init__(self, parent, x):
...         Element.__init__(self, parent)
...         self._value = x
....:
>>>     # We are just printing out this way to make it easy to see what's going on in␣
↪the examples.
....:
>>>     def _repr_(self) -> str:
...         return f"LocalElt({self._value})"
....:
>>>     # Now define addition, subtraction and multiplication of elements.
```

(continues on next page)

(continued from previous page)

```
...       # Note that self and right always have the same parent.
....:
>>>       def _add_(self, right):
...           return self.parent()(self._value + right._value)
....:
>>>       def _sub_(self, right):
...           return self.parent()(self._value - right._value)
....:
>>>       def _mul_(self, right):
...           return self.parent()(self._value * right._value)
....:
>>>       # The basering was set to ZZ, so c is guaranteed to be in ZZ
....:
>>>       def _rmul_(self, c):
...           return self.parent()(c * self._value)
....:
>>>       def _lmul_(self, c):
...           return self.parent()(self._value * c)
....:
>>>       def _richcmp_(self, other, op):
...           return richcmp(self._value, other._value, op)

>>> MyLocalization.element_class = MyLocalizationElement
```

That's all there is to it. Now we can test it out:

```
sage: TestSuite(R).run(skip="_test_pickling")
sage: R = MyLocalization([2]); R
Integer Ring localized at [2]
sage: R(1)
LocalElt(1)
sage: R(1/2)
LocalElt(1/2)
sage: R(1/3)
Traceback (most recent call last):
...
ValueError: not integral at 3

sage: R.coerce(1)
LocalElt(1)
sage: R.coerce(1/4)
Traceback (most recent call last):
...
TypeError: no canonical coercion from Rational Field to Integer Ring localized at [2]

sage: R(1/2) + R(3/4)
LocalElt(5/4)
sage: R(1/2) + 5
LocalElt(11/2)
sage: 5 + R(1/2)
LocalElt(11/2)
sage: R(1/2) + 1/7
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +: 'Integer Ring localized at [2]' and
↪'Rational Field'
sage: R(3/4) * 7
LocalElt(21/4)

sage: cm = sage.structure.element.get_coercion_model()
sage: cm.explain(R, ZZ, operator.add)
Coercion on right operand via
   Coercion map:
     From: Integer Ring
     To:   Integer Ring localized at [2]
Arithmetic performed after coercions.
Result lives in Integer Ring localized at [2]
Integer Ring localized at [2]

sage: cm.explain(R, ZZ, operator.mul)
Coercion on right operand via
    Coercion map:
      From: Integer Ring
      To:   Integer Ring localized at [2]
Arithmetic performed after coercions.
Result lives in Integer Ring localized at [2]
Integer Ring localized at [2]

sage: R6 = MyLocalization([2,3]); R6
Integer Ring localized at [2, 3]
sage: R6(1/3) - R(1/2)
LocalElt(-1/6)
sage: parent(R6(1/3) - R(1/2))
Integer Ring localized at [2, 3]

sage: R.has_coerce_map_from(ZZ)
True
sage: R.coerce_map_from(ZZ)
Coercion map:
 From: Integer Ring
 To:   Integer Ring localized at [2]

sage: R6.coerce_map_from(R)
Coercion map:
 From: Integer Ring localized at [2]
 To:   Integer Ring localized at [2, 3]

sage: R6.coerce(R(1/2))
LocalElt(1/2)

sage: cm.explain(R, R6, operator.mul)
Coercion on left operand via
   Coercion map:
     From: Integer Ring localized at [2]
```

(continues on next page)

(continued from previous page)

```
    To:   Integer Ring localized at [2, 3]
Arithmetic performed after coercions.
Result lives in Integer Ring localized at [2, 3]
Integer Ring localized at [2, 3]
```

```
>>> from sage.all import *
>>> TestSuite(R).run(skip="_test_pickling")
>>> R = MyLocalization([Integer(2)]); R
Integer Ring localized at [2]
>>> R(Integer(1))
LocalElt(1)
>>> R(Integer(1)/Integer(2))
LocalElt(1/2)
>>> R(Integer(1)/Integer(3))
Traceback (most recent call last):
...
ValueError: not integral at 3

>>> R.coerce(Integer(1))
LocalElt(1)
>>> R.coerce(Integer(1)/Integer(4))
Traceback (most recent call last):
...
TypeError: no canonical coercion from Rational Field to Integer Ring localized at [2]

>>> R(Integer(1)/Integer(2)) + R(Integer(3)/Integer(4))
LocalElt(5/4)
>>> R(Integer(1)/Integer(2)) + Integer(5)
LocalElt(11/2)
>>> Integer(5) + R(Integer(1)/Integer(2))
LocalElt(11/2)
>>> R(Integer(1)/Integer(2)) + Integer(1)/Integer(7)
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +: 'Integer Ring localized at [2]' and
↪'Rational Field'
>>> R(Integer(3)/Integer(4)) * Integer(7)
LocalElt(21/4)

>>> cm = sage.structure.element.get_coercion_model()
>>> cm.explain(R, ZZ, operator.add)
Coercion on right operand via
   Coercion map:
     From: Integer Ring
     To:   Integer Ring localized at [2]
Arithmetic performed after coercions.
Result lives in Integer Ring localized at [2]
Integer Ring localized at [2]

>>> cm.explain(R, ZZ, operator.mul)
Coercion on right operand via
    Coercion map:
```

(continues on next page)

(continued from previous page)

```
      From: Integer Ring
      To:   Integer Ring localized at [2]
Arithmetic performed after coercions.
Result lives in Integer Ring localized at [2]
Integer Ring localized at [2]

>>> R6 = MyLocalization([Integer(2),Integer(3)]); R6
Integer Ring localized at [2, 3]
>>> R6(Integer(1)/Integer(3)) - R(Integer(1)/Integer(2))
LocalElt(-1/6)
>>> parent(R6(Integer(1)/Integer(3)) - R(Integer(1)/Integer(2)))
Integer Ring localized at [2, 3]

>>> R.has_coerce_map_from(ZZ)
True
>>> R.coerce_map_from(ZZ)
Coercion map:
 From: Integer Ring
 To:   Integer Ring localized at [2]

>>> R6.coerce_map_from(R)
Coercion map:
 From: Integer Ring localized at [2]
 To:   Integer Ring localized at [2, 3]

>>> R6.coerce(R(Integer(1)/Integer(2)))
LocalElt(1/2)

>>> cm.explain(R, R6, operator.mul)
Coercion on left operand via
   Coercion map:
     From: Integer Ring localized at [2]
     To:   Integer Ring localized at [2, 3]
Arithmetic performed after coercions.
Result lives in Integer Ring localized at [2, 3]
Integer Ring localized at [2, 3]
```

3.3 Provided Methods

- `__call__`

  This provides a consistent interface for element construction. In particular, it makes sure that conversion always gives the same result as coercion, if a coercion exists. (This used to be violated for some Rings in Sage as the code for conversion and coercion got edited separately.) Let R be a Parent and assume the user types R(x), where x has parent X. Roughly speaking, the following occurs:

  1. If X `is` R, return x (*)
  2. If there is a coercion $f : X \to R$, return $f(x)$
  3. If there is a coercion $f : R \to X$, try to return $f^{-1}(x)$
  4. Return `R._element_constructor_(x)` (**)

  Keywords and extra arguments are passed on. The result of all this logic is cached.

(*) Unless there is a "copy" keyword like R(x, copy=False)

(**) Technically, a generic morphism is created from X to R, which may use magic methods like `_integer_` or other data provided by `_populate_coercion_lists_`.

- `coerce`

  Coerces elements into self, raising a type error if there is no coercion map.

- `coerce_map_from, convert_map_from`

  Returns an actual `Morphism` object to coerce/convert from another Parent to self. Barring direct construction of elements of R, `R.convert_map_from(S)` will provide a callable Python object which is the fastest way to convert elements of S to elements of R. From Cython, it can be invoked via the cdef `_call_` method.

- `has_coerce_map_from`

  Returns `True` or `False` depending on whether or not there is a coercion. `R.has_coerce_map_from(S)` is shorthand for `R.coerce_map_from(S) is not None`

- `get_action`

  This will unwind all the `_get_action_, _rmul_, _lmul_, _act_on_, _acted_upon_, ...` methods to provide an actual `Action` object, if one exists.

CHAPTER

FOUR

DISCOVERING NEW PARENTS

New parents are discovered using an algorithm in sage/category/pushout.py. The fundamental idea is that most Parents in Sage are constructed from simpler objects via various functors. These are accessed via the `construction()` method, which returns a (simpler) Parent along with a functor with which one can create self.

```
sage: CC.construction()
(AlgebraicClosureFunctor, Real Field with 53 bits of precision)
sage: RR.construction()
(Completion[+Infinity, prec=53], Rational Field)
sage: QQ.construction()
(FractionField, Integer Ring)
sage: ZZ.construction()  # None

sage: Zp(5).construction()
(Completion[5, prec=20], Integer Ring)
sage: QQ.completion(5, 100, {})
5-adic Field with capped relative precision 100
sage: c, R = RR.construction()
sage: a = CC.construction()[0]
sage: a.commutes(c)
False
sage: RR == c(QQ)
True

sage: sage.categories.pushout.construction_tower(Frac(CDF['x']))
[(None,
  Fraction Field of Univariate Polynomial Ring in x over Complex Double Field),
 (FractionField, Univariate Polynomial Ring in x over Complex Double Field),
 (Poly[x], Complex Double Field),
 (AlgebraicClosureFunctor, Real Double Field),
 (Completion[+Infinity, prec=53], Rational Field),
 (FractionField, Integer Ring)]
```

```
>>> from sage.all import *
>>> CC.construction()
(AlgebraicClosureFunctor, Real Field with 53 bits of precision)
>>> RR.construction()
(Completion[+Infinity, prec=53], Rational Field)
>>> QQ.construction()
(FractionField, Integer Ring)
>>> ZZ.construction()  # None
```

(continues on next page)

(continued from previous page)

```
>>> Zp(Integer(5)).construction()
(Completion[5, prec=20], Integer Ring)
>>> QQ.completion(Integer(5), Integer(100), {})
5-adic Field with capped relative precision 100
>>> c, R = RR.construction()
>>> a = CC.construction()[Integer(0)]
>>> a.commutes(c)
False
>>> RR == c(QQ)
True

>>> sage.categories.pushout.construction_tower(Frac(CDF['x']))
[(None,
  Fraction Field of Univariate Polynomial Ring in x over Complex Double Field),
 (FractionField, Univariate Polynomial Ring in x over Complex Double Field),
 (Poly[x], Complex Double Field),
 (AlgebraicClosureFunctor, Real Double Field),
 (Completion[+Infinity, prec=53], Rational Field),
 (FractionField, Integer Ring)]
```

Given Parents R and S, such that there is no coercion either from R to S or from S to R, one can find a common Z with coercions $R \to Z$ and $S \to Z$ by considering the sequence of construction functors to get from a common ancestor to both R and S. We then use a *heuristic* algorithm to interleave these constructors in an attempt to arrive at a suitable Z (if one exists). For example:

```
sage: ZZ['x'].construction()
(Poly[x], Integer Ring)
sage: QQ.construction()
(FractionField, Integer Ring)
sage: sage.categories.pushout.pushout(ZZ['x'], QQ)
Univariate Polynomial Ring in x over Rational Field
sage: sage.categories.pushout.pushout(ZZ['x'], QQ).construction()
(Poly[x], Rational Field)
```

```
>>> from sage.all import *
>>> ZZ['x'].construction()
(Poly[x], Integer Ring)
>>> QQ.construction()
(FractionField, Integer Ring)
>>> sage.categories.pushout.pushout(ZZ['x'], QQ)
Univariate Polynomial Ring in x over Rational Field
>>> sage.categories.pushout.pushout(ZZ['x'], QQ).construction()
(Poly[x], Rational Field)
```

The common ancestor is Z and our options for Z are Frac($\mathbf{Z}[x]$) or Frac($\mathbf{Z}$)$[x]$. In Sage we choose the later, treating the fraction field functor as binding "more tightly" than the polynomial functor, as most people agree that $\mathbf{Q}[x]$ is the more natural choice. The same procedure is applied to more complicated Parents, returning a new Parent if one can be unambiguously determined.

```
sage: sage.categories.pushout.pushout(Frac(ZZ['x,y,z']), QQ['z, t'])
Univariate Polynomial Ring in t over Fraction Field of Multivariate Polynomial Ring␣
```

(continues on next page)

(continued from previous page)

```
↪in x, y, z over Rational Field
```

```
>>> from sage.all import *
>>> sage.categories.pushout.pushout(Frac(ZZ['x,y,z']), QQ['z, t'])
Univariate Polynomial Ring in t over Fraction Field of Multivariate Polynomial Ring␣
↪in x, y, z over Rational Field
```

CHAPTER

FIVE

MODULES

5.1 The coercion model

The coercion model manages how elements of one parent get related to elements of another. For example, the integer 2 can canonically be viewed as an element of the rational numbers. (The parent of a non-element is its Python type.)

```
sage: ZZ(2).parent()
Integer Ring
sage: QQ(2).parent()
Rational Field
```

```
>>> from sage.all import *
>>> ZZ(Integer(2)).parent()
Integer Ring
>>> QQ(Integer(2)).parent()
Rational Field
```

The most prominent role of the coercion model is to make sense of binary operations between elements that have distinct parents. It does this by finding a parent where both elements make sense, and doing the operation there. For example:

```
sage: a = 1/2; a.parent()
Rational Field
sage: b = ZZ['x'].gen(); b.parent()
Univariate Polynomial Ring in x over Integer Ring
sage: a + b
x + 1/2
sage: (a + b).parent()
Univariate Polynomial Ring in x over Rational Field
```

```
>>> from sage.all import *
>>> a = Integer(1)/Integer(2); a.parent()
Rational Field
>>> b = ZZ['x'].gen(); b.parent()
Univariate Polynomial Ring in x over Integer Ring
>>> a + b
x + 1/2
>>> (a + b).parent()
Univariate Polynomial Ring in x over Rational Field
```

If there is a coercion (see below) from one of the parents to the other, the operation is always performed in the codomain of that coercion. Otherwise a reasonable attempt to create a new parent with coercion maps from both original parents is made. The results of these discoveries are cached. On failure, a `TypeError` is always raised.

Some arithmetic operations (such as multiplication) can indicate an action rather than arithmetic in a common parent. For example:

```
sage: E = EllipticCurve('37a')                                                  #␣
↪needs sage.schemes
sage: P = E(0,0)                                                                #␣
↪needs sage.schemes
sage: 5*P                                                                       #␣
↪needs sage.schemes
(1/4 : -5/8 : 1)
```

```
>>> from sage.all import *
>>> E = EllipticCurve('37a')                                                  #␣
↪needs sage.schemes
>>> P = E(Integer(0),Integer(0))                                                ␣
↪              # needs sage.schemes
>>> Integer(5)*P                                                               ␣
↪     # needs sage.schemes
(1/4 : -5/8 : 1)
```

where there is action of $\mathbf{Z}$ on the points of E given by the additive group law. Parents can specify how they act on or are acted upon by other parents.

There are two kinds of ways to get from one parent to another, coercions and conversions.

Coercions are canonical (possibly modulo a finite number of deterministic choices) morphisms, and the set of all coercions between all parents forms a commuting diagram (modulo possibly rounding issues). $\mathbf{Z} \to \mathbf{Q}$ is an example of a coercion. These are invoked implicitly by the coercion model.

Conversions try to construct an element out of their input if at all possible. Examples include sections of coercions, creating an element from a string or list, etc. and may fail on some inputs of a given type while succeeding on others (i.e. they may not be defined on the whole domain). Conversions are always explicitly invoked, and never used by the coercion model to resolve binary operations.

For more information on how to specify coercions, conversions, and actions, see the documentation for `Parent`.

class sage.structure.coerce.**CoercionModel**

Bases: `object`

See also *sage.categories.pushout*

EXAMPLES:

```
sage: f = ZZ['t', 'x'].0 + QQ['x'].0 + CyclotomicField(13).gen(); f             #␣
↪needs sage.rings.number_field
t + x + zeta13
sage: f.parent()                                                                #␣
↪needs sage.rings.number_field
Multivariate Polynomial Ring in t, x
 over Cyclotomic Field of order 13 and degree 12
sage: ZZ['x','y'].0 + ~Frac(QQ['y']).0
(x*y + 1)/y
sage: MatrixSpace(ZZ['x'], 2, 2)(2) + ~Frac(QQ['x']).0                          #␣
↪needs sage.modules
[(2*x + 1)/x           0]
[          0 (2*x + 1)/x]
sage: f = ZZ['x,y,z'].0 + QQ['w,x,z,a'].0; f
```

(continues on next page)

(continued from previous page)

```
w + x
sage: f.parent()
Multivariate Polynomial Ring in w, x, y, z, a over Rational Field
sage: ZZ['x,y,z'].0 + ZZ['w,x,z,a'].1
2*x
```

```
>>> from sage.all import *
>>> f = ZZ['t', 'x'].gen(0) + QQ['x'].gen(0) + CyclotomicField(Integer(13)).gen();
 ↪ f             # needs sage.rings.number_field
t + x + zeta13
>>> f.parent()                                                              #␣
↪needs sage.rings.number_field
Multivariate Polynomial Ring in t, x
 over Cyclotomic Field of order 13 and degree 12
>>> ZZ['x','y'].gen(0) + ~Frac(QQ['y']).gen(0)
(x*y + 1)/y
>>> MatrixSpace(ZZ['x'], Integer(2), Integer(2))(Integer(2)) + ~Frac(QQ['x']).
↪gen(0)                          # needs sage.modules
[(2*x + 1)/x           0]
[          0 (2*x + 1)/x]
>>> f = ZZ['x,y,z'].gen(0) + QQ['w,x,z,a'].gen(0); f
w + x
>>> f.parent()
Multivariate Polynomial Ring in w, x, y, z, a over Rational Field
>>> ZZ['x,y,z'].gen(0) + ZZ['w,x,z,a'].gen(1)
2*x
```

AUTHOR:

- Robert Bradshaw

analyse(*xp*, *yp*, *op='mul'*)

Emulate the process of doing arithmetic between xp and yp, returning a list of steps and the parent that the result will live in.

The *explain()* method is easier to use, but if one wants access to the actual morphism and action objects (rather than their string representations), then this is the function to use.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: GF7 = GF(7)
sage: steps, res = cm.analyse(GF7, ZZ)
sage: steps
['Coercion on right operand via',
 Natural morphism:
  From: Integer Ring
  To:   Finite Field of size 7,
 'Arithmetic performed after coercions.']
sage: res
Finite Field of size 7
sage: f = steps[1]; type(f)
<class 'sage.rings.finite_rings.integer_mod.Integer_to_IntegerMod'>
```

(continues on next page)

(continued from previous page)

```
sage: f(100)
2
```

```
>>> from sage.all import *
>>> cm = sage.structure.element.get_coercion_model()
>>> GF7 = GF(Integer(7))
>>> steps, res = cm.analyse(GF7, ZZ)
>>> steps
['Coercion on right operand via',
 Natural morphism:
  From: Integer Ring
  To:   Finite Field of size 7,
 'Arithmetic performed after coercions.']
>>> res
Finite Field of size 7
>>> f = steps[Integer(1)]; type(f)
<class 'sage.rings.finite_rings.integer_mod.Integer_to_IntegerMod'>
>>> f(Integer(100))
2
```

bin_op(*x*, *y*, *op*)

Execute the operation `op` on x and y.

It first looks for an action corresponding to `op`, and failing that, it tries to coerce x and y into a common parent and calls `op` on them.

If it cannot make sense of the operation, a `TypeError` is raised.

INPUT:

- `x` – the left operand
- `y` – the right operand
- `op` – a python function taking 2 arguments

Note

`op` is often an arithmetic operation, but need not be so.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.bin_op(1/2, 5, operator.mul)
5/2
```

```
>>> from sage.all import *
>>> cm = sage.structure.element.get_coercion_model()
>>> cm.bin_op(Integer(1)/Integer(2), Integer(5), operator.mul)
5/2
```

The operator can be any callable:

```
sage: R.<x> = ZZ['x']
sage: cm.bin_op(x^2 - 1, x + 1, gcd)
x + 1
```

```
>>> from sage.all import *
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> cm.bin_op(x**Integer(2) - Integer(1), x + Integer(1), gcd)
x + 1
```

Actions are detected and performed:

```
sage: M = matrix(ZZ, 2, 2, range(4))                                    #␣
↪needs sage.modules
sage: V = vector(ZZ, [5,7])                                             #␣
↪needs sage.modules
sage: cm.bin_op(M, V, operator.mul)                                     #␣
↪needs sage.modules
(7, 31)
```

```
>>> from sage.all import *
>>> M = matrix(ZZ, Integer(2), Integer(2), range(Integer(4)))           ␣
↪                        # needs sage.modules
>>> V = vector(ZZ, [Integer(5),Integer(7)])                            ␣
↪               # needs sage.modules
>>> cm.bin_op(M, V, operator.mul)                                      #␣
↪needs sage.modules
(7, 31)
```

canonical_coercion(*x*, *y*)

Given two elements x and y, with parents S and R respectively, find a common parent Z such that there are coercions $f : S \to Z$ and $g : R \to Z$ and return $f(x), g(y)$, which will have the same parent.

Raises a type error if no such Z can be found.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.canonical_coercion(mod(2, 10), 17)
(2, 7)

sage: # needs sage.modules
sage: x, y = cm.canonical_coercion(1/2, matrix(ZZ, 2, 2, range(4)))
sage: x
[1/2   0]
[  0 1/2]
sage: y
[0 1]
[2 3]
sage: parent(x) is parent(y)
True
```

```
>>> from sage.all import *
>>> cm = sage.structure.element.get_coercion_model()
```

(continues on next page)

(continued from previous page)

```
>>> cm.canonical_coercion(mod(Integer(2), Integer(10)), Integer(17))
(2, 7)

>>> # needs sage.modules
>>> x, y = cm.canonical_coercion(Integer(1)/Integer(2), matrix(ZZ, Integer(2),
↪ Integer(2), range(Integer(4))))
>>> x
[1/2   0]
[  0 1/2]
>>> y
[0 1]
[2 3]
>>> parent(x) is parent(y)
True
```

There is some support for non-Sage datatypes as well:

```
sage: x, y = cm.canonical_coercion(int(5), 10)
sage: type(x), type(y)
(<class 'sage.rings.integer.Integer'>, <class 'sage.rings.integer.Integer'>)

sage: x, y = cm.canonical_coercion(int(5), complex(3))
sage: type(x), type(y)
(<class 'complex'>, <class 'complex'>)

sage: class MyClass:
....:     def _sage_(self):
....:         return 13
sage: a, b = cm.canonical_coercion(MyClass(), 1/3)
sage: a, b
(13, 1/3)
sage: type(a)
<class 'sage.rings.rational.Rational'>
```

```
>>> from sage.all import *
>>> x, y = cm.canonical_coercion(int(Integer(5)), Integer(10))
>>> type(x), type(y)
(<class 'sage.rings.integer.Integer'>, <class 'sage.rings.integer.Integer'>)

>>> x, y = cm.canonical_coercion(int(Integer(5)), complex(Integer(3)))
>>> type(x), type(y)
(<class 'complex'>, <class 'complex'>)

>>> class MyClass:
...     def _sage_(self):
...         return Integer(13)
>>> a, b = cm.canonical_coercion(MyClass(), Integer(1)/Integer(3))
>>> a, b
(13, 1/3)
>>> type(a)
<class 'sage.rings.rational.Rational'>
```

We also make an exception for 0, even if **Z** does not map in:

```
sage: canonical_coercion(vector([1, 2, 3]), 0)                                  #␣
↪needs sage.modules
((1, 2, 3), (0, 0, 0))
sage: canonical_coercion(GF(5)(0), float(0))
(0, 0)
```

```
>>> from sage.all import *
>>> canonical_coercion(vector([Integer(1), Integer(2), Integer(3)]),␣
↪Integer(0))                                  # needs sage.modules
((1, 2, 3), (0, 0, 0))
>>> canonical_coercion(GF(Integer(5))(Integer(0)), float(Integer(0)))
(0, 0)
```

coercion_maps(R, S)

Give two parents R and S, return a pair of coercion maps $f : R \to Z$ and $g : S \to Z$, if such a Z can be found.

In the (common) case that $R = Z$ or $S = Z$ then `None` is returned for f or g respectively rather than constructing (and subsequently calling) the identity morphism.

If no suitable f, g can be found, a single `None` is returned. This result is cached.

Note

By Issue #14711, coerce maps should be copied when using them outside of the coercion system, because they may become defunct by garbage collection.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: f, g = cm.coercion_maps(ZZ, QQ)
sage: print(copy(f))
Natural morphism:
  From: Integer Ring
  To:   Rational Field
sage: print(g)
None

sage: ZZx = ZZ['x']
sage: f, g = cm.coercion_maps(ZZx, QQ)
sage: print(f)
(map internal to coercion system -- copy before use)
Ring morphism:
  From: Univariate Polynomial Ring in x over Integer Ring
  To:   Univariate Polynomial Ring in x over Rational Field
sage: print(g)
(map internal to coercion system -- copy before use)
Polynomial base injection morphism:
  From: Rational Field
  To:   Univariate Polynomial Ring in x over Rational Field

sage: K = GF(7)
```

(continues on next page)

(continued from previous page)

```
sage: cm.coercion_maps(QQ, K) is None
True
```

```
>>> from sage.all import *
>>> cm = sage.structure.element.get_coercion_model()
>>> f, g = cm.coercion_maps(ZZ, QQ)
>>> print(copy(f))
Natural morphism:
  From: Integer Ring
  To:   Rational Field
>>> print(g)
None

>>> ZZx = ZZ['x']
>>> f, g = cm.coercion_maps(ZZx, QQ)
>>> print(f)
(map internal to coercion system -- copy before use)
Ring morphism:
  From: Univariate Polynomial Ring in x over Integer Ring
  To:   Univariate Polynomial Ring in x over Rational Field
>>> print(g)
(map internal to coercion system -- copy before use)
Polynomial base injection morphism:
  From: Rational Field
  To:   Univariate Polynomial Ring in x over Rational Field

>>> K = GF(Integer(7))
>>> cm.coercion_maps(QQ, K) is None
True
```

Note that to break symmetry, if there is a coercion map in both directions, the parent on the left is used:

```
sage: # needs sage.modules
sage: V = QQ^3
sage: W = V.__class__(QQ, 3)
sage: V == W
True
sage: V is W
False
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.coercion_maps(V, W)
(None,
 (map internal to coercion system -- copy before use)
 Coercion map:
   From: Vector space of dimension 3 over Rational Field
   To:   Vector space of dimension 3 over Rational Field)
sage: cm.coercion_maps(W, V)
(None,
 (map internal to coercion system -- copy before use)
 Coercion map:
   From: Vector space of dimension 3 over Rational Field
   To:   Vector space of dimension 3 over Rational Field)
```

(continues on next page)

(continued from previous page)

```
sage: v = V([1,2,3])
sage: w = W([1,2,3])
sage: parent(v + w) is V
True
sage: parent(w + v) is W
True
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> V = QQ**Integer(3)
>>> W = V.__class__(QQ, Integer(3))
>>> V == W
True
>>> V is W
False
>>> cm = sage.structure.element.get_coercion_model()
>>> cm.coercion_maps(V, W)
(None,
 (map internal to coercion system -- copy before use)
 Coercion map:
   From: Vector space of dimension 3 over Rational Field
   To:   Vector space of dimension 3 over Rational Field)
>>> cm.coercion_maps(W, V)
(None,
 (map internal to coercion system -- copy before use)
 Coercion map:
   From: Vector space of dimension 3 over Rational Field
   To:   Vector space of dimension 3 over Rational Field)
>>> v = V([Integer(1),Integer(2),Integer(3)])
>>> w = W([Integer(1),Integer(2),Integer(3)])
>>> parent(v + w) is V
True
>>> parent(w + v) is W
True
```

common_parent(**args*)

Compute a common parent for all the inputs.

It's essentially an n-ary canonical coercion except it can operate on parents rather than just elements.

INPUT:

- `args` – set of elements and/or parents

OUTPUT:

A `Parent` into which each input should coerce, or raises a `TypeError` if no such `Parent` can be found.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.common_parent(ZZ, QQ)
Rational Field
sage: cm.common_parent(ZZ, QQ, RR)                                          #␣
↪needs sage.rings.real_mpfr
```

(continues on next page)

(continued from previous page)

```
Real Field with 53 bits of precision
sage: ZZT = ZZ[['T']]
sage: QQT = QQ['T']
sage: cm.common_parent(ZZT, QQT, RDF)
Power Series Ring in T over Real Double Field
sage: cm.common_parent(4r, 5r)
<class 'int'>
sage: cm.common_parent(int, float, ZZ)
<class 'float'>
sage: real_fields = [RealField(prec) for prec in [10,20..100]]              #␣
↪needs sage.rings.real_mpfr
sage: cm.common_parent(*real_fields)                                        #␣
↪needs sage.rings.real_mpfr
Real Field with 10 bits of precision
```

```
>>> from sage.all import *
>>> cm = sage.structure.element.get_coercion_model()
>>> cm.common_parent(ZZ, QQ)
Rational Field
>>> cm.common_parent(ZZ, QQ, RR)                                            #␣
↪needs sage.rings.real_mpfr
Real Field with 53 bits of precision
>>> ZZT = ZZ[['T']]
>>> QQT = QQ['T']
>>> cm.common_parent(ZZT, QQT, RDF)
Power Series Ring in T over Real Double Field
>>> cm.common_parent(4, 5)
<class 'int'>
>>> cm.common_parent(int, float, ZZ)
<class 'float'>
>>> real_fields = [RealField(prec) for prec in (ellipsis_range(Integer(10),
↪Integer(20),Ellipsis,Integer(100)))]              # needs sage.rings.real_
↪mpfr
>>> cm.common_parent(*real_fields)                                          #␣
↪needs sage.rings.real_mpfr
Real Field with 10 bits of precision
```

There are some cases where the ordering does matter, but if a parent can be found it is always the same:

```
sage: QQxy = QQ['x,y']
sage: QQyz = QQ['y,z']
sage: cm.common_parent(QQxy, QQyz) == cm.common_parent(QQyz, QQxy)
True
sage: QQzt = QQ['z,t']
sage: cm.common_parent(QQxy, QQyz, QQzt)
Multivariate Polynomial Ring in x, y, z, t over Rational Field
sage: cm.common_parent(QQxy, QQzt, QQyz)
Traceback (most recent call last):
...
TypeError: no common canonical parent for objects with parents:
'Multivariate Polynomial Ring in x, y over Rational Field' and
'Multivariate Polynomial Ring in z, t over Rational Field'
```

```
>>> from sage.all import *
>>> QQxy = QQ['x,y']
>>> QQyz = QQ['y,z']
>>> cm.common_parent(QQxy, QQyz) == cm.common_parent(QQyz, QQxy)
True
>>> QQzt = QQ['z,t']
>>> cm.common_parent(QQxy, QQyz, QQzt)
Multivariate Polynomial Ring in x, y, z, t over Rational Field
>>> cm.common_parent(QQxy, QQzt, QQyz)
Traceback (most recent call last):
...
TypeError: no common canonical parent for objects with parents:
'Multivariate Polynomial Ring in x, y over Rational Field' and
'Multivariate Polynomial Ring in z, t over Rational Field'
```

discover_action(*R*, *S*, *op*, *r=None*, *s=None*)

INPUT:

- `R` – the left `Parent` (or type)
- `S` – the right `Parent` (or type)
- `op` – the operand, typically an element of the `operator` module
- `r` – (optional) element of R
- `s` – (optional) element of S

OUTPUT: an action A such that s `op` r is given by $A(s, r)$

The steps taken are illustrated below.

EXAMPLES:

```
sage: P.<x> = ZZ['x']
sage: P.get_action(ZZ)
Right scalar multiplication by Integer Ring on
 Univariate Polynomial Ring in x over Integer Ring
sage: ZZ.get_action(P) is None
True
sage: cm = sage.structure.element.get_coercion_model()
```

```
>>> from sage.all import *
>>> P = ZZ['x']; (x,) = P._first_ngens(1)
>>> P.get_action(ZZ)
Right scalar multiplication by Integer Ring on
 Univariate Polynomial Ring in x over Integer Ring
>>> ZZ.get_action(P) is None
True
>>> cm = sage.structure.element.get_coercion_model()
```

If R or S is a `Parent`, ask it for an action by/on R:

```
sage: cm.discover_action(ZZ, P, operator.mul)
Left scalar multiplication by Integer Ring on
 Univariate Polynomial Ring in x over Integer Ring
```

```
>>> from sage.all import *
>>> cm.discover_action(ZZ, P, operator.mul)
Left scalar multiplication by Integer Ring on
 Univariate Polynomial Ring in x over Integer Ring
```

If R or S a type, recursively call *get_action()* with the Sage versions of R and/or S:

```
sage: cm.discover_action(P, int, operator.mul)
Right scalar multiplication by Integer Ring on
 Univariate Polynomial Ring in x over Integer Ring
 with precomposition on right by Native morphism:
  From: Set of Python objects of class 'int'
  To:   Integer Ring
```

```
>>> from sage.all import *
>>> cm.discover_action(P, int, operator.mul)
Right scalar multiplication by Integer Ring on
 Univariate Polynomial Ring in x over Integer Ring
 with precomposition on right by Native morphism:
  From: Set of Python objects of class 'int'
  To:   Integer Ring
```

If `op` is division, look for action on `right` by inverse:

```
sage: cm.discover_action(P, ZZ, operator.truediv)
Right inverse action by Rational Field on
 Univariate Polynomial Ring in x over Integer Ring
with precomposition on right by Natural morphism:
  From: Integer Ring
  To:   Rational Field
```

```
>>> from sage.all import *
>>> cm.discover_action(P, ZZ, operator.truediv)
Right inverse action by Rational Field on
 Univariate Polynomial Ring in x over Integer Ring
with precomposition on right by Natural morphism:
  From: Integer Ring
  To:   Rational Field
```

Check that Issue #17740 is fixed:

```
sage: R = GF(5)['x']
sage: cm.discover_action(R, ZZ, operator.truediv)
Right inverse action by Finite Field of size 5
 on Univariate Polynomial Ring in x over Finite Field of size 5
with precomposition on right by Natural morphism:
  From: Integer Ring
  To:   Finite Field of size 5
sage: cm.bin_op(R.gen(), 7, operator.truediv).parent()
Univariate Polynomial Ring in x over Finite Field of size 5
```

```
>>> from sage.all import *
>>> R = GF(Integer(5))['x']
```

(continues on next page)

(continued from previous page)

```
>>> cm.discover_action(R, ZZ, operator.truediv)
Right inverse action by Finite Field of size 5
 on Univariate Polynomial Ring in x over Finite Field of size 5
with precomposition on right by Natural morphism:
  From: Integer Ring
  To:   Finite Field of size 5
>>> cm.bin_op(R.gen(), Integer(7), operator.truediv).parent()
Univariate Polynomial Ring in x over Finite Field of size 5
```

Check that Issue #18221 is fixed:

```
sage: # needs sage.combinat sage.modules
sage: F.<x> = FreeAlgebra(QQ)
sage: x / 2
1/2*x
sage: cm.discover_action(F, ZZ, operator.truediv)
Right inverse action by Rational Field on
 Free Algebra on 1 generator (x,) over Rational Field
 with precomposition on right by Natural morphism:
  From: Integer Ring
  To:   Rational Field
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> F = FreeAlgebra(QQ, names=('x',)); (x,) = F._first_ngens(1)
>>> x / Integer(2)
1/2*x
>>> cm.discover_action(F, ZZ, operator.truediv)
Right inverse action by Rational Field on
 Free Algebra on 1 generator (x,) over Rational Field
 with precomposition on right by Natural morphism:
  From: Integer Ring
  To:   Rational Field
```

discover_coercion(R, S)

This actually implements the finding of coercion maps as described in the *coercion_maps()* method.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
```

```
>>> from sage.all import *
>>> cm = sage.structure.element.get_coercion_model()
```

If R is S, then two identity morphisms suffice:

```
sage: cm.discover_coercion(SR, SR)                                          #␣
↪needs sage.symbolic
(None, None)
```

```
>>> from sage.all import *
>>> cm.discover_coercion(SR, SR)                                            #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.symbolic
(None, None)
```

If there is a coercion map either direction, use that:

```
sage: cm.discover_coercion(ZZ, QQ)
((map internal to coercion system -- copy before use)
Natural morphism:
  From: Integer Ring
  To:   Rational Field, None)
sage: cm.discover_coercion(RR, QQ)                                          #␣
↪needs sage.rings.real_mpfr
(None, (map internal to coercion system -- copy before use)
 Generic map:
  From: Rational Field
  To:   Real Field with 53 bits of precision)
```

```
>>> from sage.all import *
>>> cm.discover_coercion(ZZ, QQ)
((map internal to coercion system -- copy before use)
Natural morphism:
  From: Integer Ring
  To:   Rational Field, None)
>>> cm.discover_coercion(RR, QQ)                                            #␣
↪needs sage.rings.real_mpfr
(None, (map internal to coercion system -- copy before use)
 Generic map:
  From: Rational Field
  To:   Real Field with 53 bits of precision)
```

Otherwise, try and compute an appropriate cover:

```
sage: ZZxy = ZZ['x,y']
sage: cm.discover_coercion(ZZxy, RDF)
((map internal to coercion system -- copy before use)
Call morphism:
  From: Multivariate Polynomial Ring in x, y over Integer Ring
  To:   Multivariate Polynomial Ring in x, y over Real Double Field,
 (map internal to coercion system -- copy before use)
 Polynomial base injection morphism:
  From: Real Double Field
  To:   Multivariate Polynomial Ring in x, y over Real Double Field)
```

```
>>> from sage.all import *
>>> ZZxy = ZZ['x,y']
>>> cm.discover_coercion(ZZxy, RDF)
((map internal to coercion system -- copy before use)
Call morphism:
  From: Multivariate Polynomial Ring in x, y over Integer Ring
  To:   Multivariate Polynomial Ring in x, y over Real Double Field,
 (map internal to coercion system -- copy before use)
 Polynomial base injection morphism:
```

(continues on next page)

(continued from previous page)

```
  From: Real Double Field
  To:   Multivariate Polynomial Ring in x, y over Real Double Field)
```

Sometimes there is a reasonable "cover," but no canonical coercion:

```
sage: sage.categories.pushout.pushout(QQ, QQ^3)                                #␣
↪needs sage.modules
Vector space of dimension 3 over Rational Field
sage: print(cm.discover_coercion(QQ, QQ^3))                                    #␣
↪needs sage.modules
None
```

```
>>> from sage.all import *
>>> sage.categories.pushout.pushout(QQ, QQ**Integer(3))                        ␣
↪       # needs sage.modules
Vector space of dimension 3 over Rational Field
>>> print(cm.discover_coercion(QQ, QQ**Integer(3)))                            ␣
↪       # needs sage.modules
None
```

division_parent(*P*)

Deduces where the result of division in `P` lies by calculating the inverse of `P.one()` or `P.an_element()`.

The result is cached.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.division_parent(ZZ)
Rational Field
sage: cm.division_parent(QQ)
Rational Field
sage: ZZx = ZZ['x']
sage: cm.division_parent(ZZx)
Fraction Field of Univariate Polynomial Ring in x over Integer Ring
sage: K = GF(41)
sage: cm.division_parent(K)
Finite Field of size 41
sage: Zmod100 = Integers(100)
sage: cm.division_parent(Zmod100)
Ring of integers modulo 100
sage: S5 = SymmetricGroup(5)                                                   #␣
↪needs sage.groups
sage: cm.division_parent(S5)                                                   #␣
↪needs sage.groups
Symmetric group of order 5! as a permutation group
```

```
>>> from sage.all import *
>>> cm = sage.structure.element.get_coercion_model()
>>> cm.division_parent(ZZ)
Rational Field
>>> cm.division_parent(QQ)
Rational Field
```

(continues on next page)

(continued from previous page)

```
>>> ZZx = ZZ['x']
>>> cm.division_parent(ZZx)
Fraction Field of Univariate Polynomial Ring in x over Integer Ring
>>> K = GF(Integer(41))
>>> cm.division_parent(K)
Finite Field of size 41
>>> Zmod100 = Integers(Integer(100))
>>> cm.division_parent(Zmod100)
Ring of integers modulo 100
>>> S5 = SymmetricGroup(Integer(5))                                    ␣
↪     # needs sage.groups
>>> cm.division_parent(S5)                                            #␣
↪needs sage.groups
Symmetric group of order 5! as a permutation group
```

exception_stack()

Return the list of exceptions that were caught in the course of executing the last binary operation. Useful for diagnosis when user-defined maps or actions raise exceptions that are caught in the course of coercion detection.

If all went well, this should be the empty list. If things aren't happening as you expect, this is a good place to check. See also coercion_traceback().

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.record_exceptions()
sage: 1/2 + 2
5/2
sage: cm.exception_stack()
[]
sage: 1/2 + GF(3)(2)
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +:
'Rational Field' and 'Finite Field of size 3'
```

```
>>> from sage.all import *
>>> cm = sage.structure.element.get_coercion_model()
>>> cm.record_exceptions()
>>> Integer(1)/Integer(2) + Integer(2)
5/2
>>> cm.exception_stack()
[]
>>> Integer(1)/Integer(2) + GF(Integer(3))(Integer(2))
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +:
'Rational Field' and 'Finite Field of size 3'
```

Now see what the actual problem was:

```
sage: import traceback
sage: cm.exception_stack()
['Traceback (most recent call last):...', 'Traceback (most recent call last):.
↪..']
sage: print(cm.exception_stack()[-1])
Traceback (most recent call last):
...
TypeError: no common canonical parent for objects with parents:
'Rational Field' and 'Finite Field of size 3'
```

```
>>> from sage.all import *
>>> import traceback
>>> cm.exception_stack()
['Traceback (most recent call last):...', 'Traceback (most recent call last):.
↪..']
>>> print(cm.exception_stack()[-Integer(1)])
Traceback (most recent call last):
...
TypeError: no common canonical parent for objects with parents:
'Rational Field' and 'Finite Field of size 3'
```

This is typically accessed via the `coercion_traceback()` function.

```
sage: coercion_traceback()
Traceback (most recent call last):
...
TypeError: no common canonical parent for objects with parents:
'Rational Field' and 'Finite Field of size 3'
```

```
>>> from sage.all import *
>>> coercion_traceback()
Traceback (most recent call last):
...
TypeError: no common canonical parent for objects with parents:
'Rational Field' and 'Finite Field of size 3'
```

explain(*xp*, *yp*, *op='mul'*, *verbosity=2*)

This function can be used to understand what coercions will happen for an arithmetic operation between xp and yp (which may be either elements or parents). If the parent of the result can be determined then it will be returned.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()

sage: cm.explain(ZZ, ZZ)
Identical parents, arithmetic performed immediately.
Result lives in Integer Ring
Integer Ring

sage: cm.explain(QQ, int)
Coercion on right operand via
    Native morphism:
```

(continues on next page)

(continued from previous page)

```
      From: Set of Python objects of class 'int'
      To:   Rational Field
Arithmetic performed after coercions.
Result lives in Rational Field
Rational Field

sage: R = ZZ['x']
sage: cm.explain(R, QQ)
Action discovered.
    Right scalar multiplication by Rational Field
     on Univariate Polynomial Ring in x over Integer Ring
Result lives in Univariate Polynomial Ring in x over Rational Field
Univariate Polynomial Ring in x over Rational Field

sage: cm.explain(ZZ['x'], QQ, operator.add)
Coercion on left operand via
    Ring morphism:
      From: Univariate Polynomial Ring in x over Integer Ring
      To:   Univariate Polynomial Ring in x over Rational Field
      Defn: Induced from base ring by
            Natural morphism:
              From: Integer Ring
              To:   Rational Field
Coercion on right operand via
    Polynomial base injection morphism:
      From: Rational Field
      To:   Univariate Polynomial Ring in x over Rational Field
Arithmetic performed after coercions.
Result lives in Univariate Polynomial Ring in x over Rational Field
Univariate Polynomial Ring in x over Rational Field
```

```
>>> from sage.all import *
>>> cm = sage.structure.element.get_coercion_model()

>>> cm.explain(ZZ, ZZ)
Identical parents, arithmetic performed immediately.
Result lives in Integer Ring
Integer Ring

>>> cm.explain(QQ, int)
Coercion on right operand via
    Native morphism:
      From: Set of Python objects of class 'int'
      To:   Rational Field
Arithmetic performed after coercions.
Result lives in Rational Field
Rational Field

>>> R = ZZ['x']
>>> cm.explain(R, QQ)
Action discovered.
    Right scalar multiplication by Rational Field
```

(continues on next page)

(continued from previous page)

```
     on Univariate Polynomial Ring in x over Integer Ring
Result lives in Univariate Polynomial Ring in x over Rational Field
Univariate Polynomial Ring in x over Rational Field

>>> cm.explain(ZZ['x'], QQ, operator.add)
Coercion on left operand via
    Ring morphism:
      From: Univariate Polynomial Ring in x over Integer Ring
      To:   Univariate Polynomial Ring in x over Rational Field
      Defn: Induced from base ring by
            Natural morphism:
              From: Integer Ring
              To:   Rational Field
Coercion on right operand via
    Polynomial base injection morphism:
      From: Rational Field
      To:   Univariate Polynomial Ring in x over Rational Field
Arithmetic performed after coercions.
Result lives in Univariate Polynomial Ring in x over Rational Field
Univariate Polynomial Ring in x over Rational Field
```

Sometimes with non-sage types there is not enough information to deduce what will actually happen:

```
sage: R100 = RealField(100)                                                    #␣
↪needs sage.rings.real_mpfr
sage: cm.explain(R100, float, operator.add)                                    #␣
↪needs sage.rings.real_mpfr
Right operand is numeric, will attempt coercion in both directions.
Unknown result parent.
sage: parent(R100(1) + float(1))                                               #␣
↪needs sage.rings.real_mpfr
<class 'float'>
sage: cm.explain(QQ, float, operator.add)
Right operand is numeric, will attempt coercion in both directions.
Unknown result parent.
sage: parent(QQ(1) + float(1))
<class 'float'>
```

```
>>> from sage.all import *
>>> R100 = RealField(Integer(100))                                             ␣
↪     # needs sage.rings.real_mpfr
>>> cm.explain(R100, float, operator.add)                                     #␣
↪needs sage.rings.real_mpfr
Right operand is numeric, will attempt coercion in both directions.
Unknown result parent.
>>> parent(R100(Integer(1)) + float(Integer(1)))                               ␣
↪              # needs sage.rings.real_mpfr
<class 'float'>
>>> cm.explain(QQ, float, operator.add)
Right operand is numeric, will attempt coercion in both directions.
Unknown result parent.
>>> parent(QQ(Integer(1)) + float(Integer(1)))
```

(continues on next page)

(continued from previous page)

```
<class 'float'>
```

Special care is taken to deal with division:

```
sage: cm.explain(ZZ, ZZ, operator.truediv)
Identical parents, arithmetic performed immediately.
Result lives in Rational Field
Rational Field

sage: ZZx = ZZ['x']
sage: QQx = QQ['x']
sage: cm.explain(ZZx, QQx, operator.truediv)
Coercion on left operand via
    Ring morphism:
      From: Univariate Polynomial Ring in x over Integer Ring
      To:   Univariate Polynomial Ring in x over Rational Field
      Defn: Induced from base ring by
            Natural morphism:
              From: Integer Ring
              To:   Rational Field
Arithmetic performed after coercions.
Result lives in Fraction Field of Univariate Polynomial Ring in x over␣
↪Rational Field
Fraction Field of Univariate Polynomial Ring in x over Rational Field

sage: cm.explain(int, ZZ, operator.truediv)
Coercion on left operand via
    Native morphism:
      From: Set of Python objects of class 'int'
      To:   Integer Ring
Arithmetic performed after coercions.
Result lives in Rational Field
Rational Field

sage: cm.explain(ZZx, ZZ, operator.truediv)
Action discovered.
    Right inverse action by Rational Field
     on Univariate Polynomial Ring in x over Integer Ring
    with precomposition on right by Natural morphism:
      From: Integer Ring
      To:   Rational Field
Result lives in Univariate Polynomial Ring in x over Rational Field
Univariate Polynomial Ring in x over Rational Field
```

```
>>> from sage.all import *
>>> cm.explain(ZZ, ZZ, operator.truediv)
Identical parents, arithmetic performed immediately.
Result lives in Rational Field
Rational Field

>>> ZZx = ZZ['x']
>>> QQx = QQ['x']
```

(continues on next page)

(continued from previous page)

```
>>> cm.explain(ZZx, QQx, operator.truediv)
Coercion on left operand via
    Ring morphism:
      From: Univariate Polynomial Ring in x over Integer Ring
      To:   Univariate Polynomial Ring in x over Rational Field
      Defn: Induced from base ring by
            Natural morphism:
              From: Integer Ring
              To:   Rational Field
Arithmetic performed after coercions.
Result lives in Fraction Field of Univariate Polynomial Ring in x over␣
↪Rational Field
Fraction Field of Univariate Polynomial Ring in x over Rational Field

>>> cm.explain(int, ZZ, operator.truediv)
Coercion on left operand via
    Native morphism:
      From: Set of Python objects of class 'int'
      To:   Integer Ring
Arithmetic performed after coercions.
Result lives in Rational Field
Rational Field

>>> cm.explain(ZZx, ZZ, operator.truediv)
Action discovered.
    Right inverse action by Rational Field
     on Univariate Polynomial Ring in x over Integer Ring
    with precomposition on right by Natural morphism:
      From: Integer Ring
      To:   Rational Field
Result lives in Univariate Polynomial Ring in x over Rational Field
Univariate Polynomial Ring in x over Rational Field
```

Note

This function is accurate only in so far as *analyse()* is kept in sync with the *bin_op()* and *canonical_coercion()* which are kept separate for maximal efficiency.

get_action(*R*, *S*, *op='mul'*, *r=None*, *s=None*)

Get the action of R on S or S on R associated to the operation op.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: ZZx = ZZ['x']
sage: cm.get_action(ZZx, ZZ, operator.mul)
Right scalar multiplication by Integer Ring
 on Univariate Polynomial Ring in x over Integer Ring
sage: cm.get_action(ZZx, QQ, operator.mul)
Right scalar multiplication by Rational Field
 on Univariate Polynomial Ring in x over Integer Ring
```

(continues on next page)

(continued from previous page)

```
sage: QQx = QQ['x']
sage: cm.get_action(QQx, int, operator.mul)
Right scalar multiplication by Integer Ring
 on Univariate Polynomial Ring in x over Rational Field
with precomposition on right by Native morphism:
  From: Set of Python objects of class 'int'
  To:   Integer Ring

sage: A = cm.get_action(QQx, ZZ, operator.truediv); A
Right inverse action by Rational Field
 on Univariate Polynomial Ring in x over Rational Field
with precomposition on right by Natural morphism:
  From: Integer Ring
  To:   Rational Field
sage: x = QQx.gen()
sage: A(x+10, 5)
1/5*x + 2
```

```
>>> from sage.all import *
>>> cm = sage.structure.element.get_coercion_model()
>>> ZZx = ZZ['x']
>>> cm.get_action(ZZx, ZZ, operator.mul)
Right scalar multiplication by Integer Ring
 on Univariate Polynomial Ring in x over Integer Ring
>>> cm.get_action(ZZx, QQ, operator.mul)
Right scalar multiplication by Rational Field
 on Univariate Polynomial Ring in x over Integer Ring
>>> QQx = QQ['x']
>>> cm.get_action(QQx, int, operator.mul)
Right scalar multiplication by Integer Ring
 on Univariate Polynomial Ring in x over Rational Field
with precomposition on right by Native morphism:
  From: Set of Python objects of class 'int'
  To:   Integer Ring

>>> A = cm.get_action(QQx, ZZ, operator.truediv); A
Right inverse action by Rational Field
 on Univariate Polynomial Ring in x over Rational Field
with precomposition on right by Natural morphism:
  From: Integer Ring
  To:   Rational Field
>>> x = QQx.gen()
>>> A(x+Integer(10), Integer(5))
1/5*x + 2
```

get_cache()

This returns the current cache of coercion maps and actions, primarily useful for debugging and introspection.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.canonical_coercion(1, 2/3)
```

(continues on next page)

(continued from previous page)

```
(1, 2/3)
sage: maps, actions = cm.get_cache()
```

```
>>> from sage.all import *
>>> cm = sage.structure.element.get_coercion_model()
>>> cm.canonical_coercion(Integer(1), Integer(2)/Integer(3))
(1, 2/3)
>>> maps, actions = cm.get_cache()
```

Now let us see what happens when we do a binary operations with an integer and a rational:

```
sage: left_morphism_ref, right_morphism_ref = maps[ZZ, QQ]
```

```
>>> from sage.all import *
>>> left_morphism_ref, right_morphism_ref = maps[ZZ, QQ]
```

Note that by Issue #14058 the coercion model only stores a weak reference to the coercion maps in this case:

```
sage: left_morphism_ref
<weakref at ...; to 'sage.rings.rational.Z_to_Q' at ...>
```

```
>>> from sage.all import *
>>> left_morphism_ref
<weakref at ...; to 'sage.rings.rational.Z_to_Q' at ...>
```

Moreover, the weakly referenced coercion map uses only a weak reference to the codomain:

```
sage: left_morphism_ref()
(map internal to coercion system -- copy before use)
Natural morphism:
  From: Integer Ring
  To:   Rational Field
```

```
>>> from sage.all import *
>>> left_morphism_ref()
(map internal to coercion system -- copy before use)
Natural morphism:
  From: Integer Ring
  To:   Rational Field
```

To get an actual valid map, we simply copy the weakly referenced coercion map:

```
sage: print(copy(left_morphism_ref()))
Natural morphism:
  From: Integer Ring
  To:   Rational Field
sage: print(right_morphism_ref)
None
```

```
>>> from sage.all import *
>>> print(copy(left_morphism_ref()))
```

(continues on next page)

(continued from previous page)

```
Natural morphism:
  From: Integer Ring
  To:   Rational Field
>>> print(right_morphism_ref)
None
```

We can see that it coerces the left operand from an integer to a rational, and doesn't do anything to the right.

Now for some actions:

```
sage: R.<x> = ZZ['x']
sage: 1/2 * x
1/2*x
sage: maps, actions = cm.get_cache()
sage: act = actions[QQ, R, operator.mul]; act
Left scalar multiplication by Rational Field
 on Univariate Polynomial Ring in x over Integer Ring
sage: act.actor()
Rational Field
sage: act.domain()
Univariate Polynomial Ring in x over Integer Ring
sage: act.codomain()
Univariate Polynomial Ring in x over Rational Field
sage: act(1/5, x+10)
1/5*x + 2
```

```
>>> from sage.all import *
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> Integer(1)/Integer(2) * x
1/2*x
>>> maps, actions = cm.get_cache()
>>> act = actions[QQ, R, operator.mul]; act
Left scalar multiplication by Rational Field
 on Univariate Polynomial Ring in x over Integer Ring
>>> act.actor()
Rational Field
>>> act.domain()
Univariate Polynomial Ring in x over Integer Ring
>>> act.codomain()
Univariate Polynomial Ring in x over Rational Field
>>> act(Integer(1)/Integer(5), x+Integer(10))
1/5*x + 2
```

record_exceptions(*value=True*)

Enables (or disables) recording of the exceptions suppressed during arithmetic.

Each time that record_exceptions is called (either enabling or disabling the record), the exception_stack is cleared.

reset_cache()

Clear the coercion cache.

This should have no impact on the result of arithmetic operations, as the exact same coercions and actions will be re-discovered when needed.

It may be useful for debugging, and may also free some memory.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: len(cm.get_cache()[0])     # random
42
sage: cm.reset_cache()
sage: cm.get_cache()
({}, {})
```

```
>>> from sage.all import *
>>> cm = sage.structure.element.get_coercion_model()
>>> len(cm.get_cache()[Integer(0)])     # random
42
>>> cm.reset_cache()
>>> cm.get_cache()
({}, {})
```

richcmp(*x*, *y*, *op*)

Given two arbitrary objects `x` and `y`, coerce them to a common parent and compare them using rich comparison operator `op`.

EXAMPLES:

```
sage: from sage.structure.element import get_coercion_model
sage: from sage.structure.richcmp import op_LT, op_LE, op_EQ, op_NE, op_GT,␣
↪op_GE
sage: richcmp = get_coercion_model().richcmp
sage: richcmp(None, None, op_EQ)
True
sage: richcmp(None, 1, op_LT)
True
sage: richcmp("hello", None, op_LE)
False
sage: richcmp(-1, 1, op_GE)
False
sage: richcmp(int(1), float(2), op_GE)
False
```

```
>>> from sage.all import *
>>> from sage.structure.element import get_coercion_model
>>> from sage.structure.richcmp import op_LT, op_LE, op_EQ, op_NE, op_GT, op_
↪GE
>>> richcmp = get_coercion_model().richcmp
>>> richcmp(None, None, op_EQ)
True
>>> richcmp(None, Integer(1), op_LT)
True
>>> richcmp("hello", None, op_LE)
False
>>> richcmp(-Integer(1), Integer(1), op_GE)
False
>>> richcmp(int(Integer(1)), float(Integer(2)), op_GE)
```

(continues on next page)

(continued from previous page)

```
False
```

If there is no coercion, we only support == and !=:

```
sage: x = QQ.one(); y = GF(2).one()
sage: richcmp(x, y, op_EQ)
False
sage: richcmp(x, y, op_NE)
True
sage: richcmp(x, y, op_GT)
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for >:
'Rational Field' and 'Finite Field of size 2'
```

```
>>> from sage.all import *
>>> x = QQ.one(); y = GF(Integer(2)).one()
>>> richcmp(x, y, op_EQ)
False
>>> richcmp(x, y, op_NE)
True
>>> richcmp(x, y, op_GT)
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for >:
'Rational Field' and 'Finite Field of size 2'
```

We support non-Sage types with the usual Python convention:

```
sage: class AlwaysEqual():
....:     def __eq__(self, other):
....:         return True
sage: x = AlwaysEqual()
sage: x == 1
True
sage: 1 == x
True
```

```
>>> from sage.all import *
>>> class AlwaysEqual():
...     def __eq__(self, other):
...         return True
>>> x = AlwaysEqual()
>>> x == Integer(1)
True
>>> Integer(1) == x
True
```

verify_action(*action*, *R*, *S*, *op*, *fix=True*)

Verify that `action` takes an element of R on the left and S on the right, raising an error if not.

This is used for consistency checking in the coercion model.

EXAMPLES:

```
sage: R.<x> = ZZ['x']
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.verify_action(R.get_action(QQ), R, QQ, operator.mul)
Right scalar multiplication by Rational Field
 on Univariate Polynomial Ring in x over Integer Ring
sage: cm.verify_action(R.get_action(QQ), RDF, R, operator.mul)
Traceback (most recent call last):
...
RuntimeError: There is a BUG in the coercion model:
    Action found for R <built-in function mul> S does not have the correct␣
↪domains
    R = Real Double Field
    S = Univariate Polynomial Ring in x over Integer Ring
    (should be Univariate Polynomial Ring in x over Integer Ring, Rational␣
↪Field)
    action = Right scalar multiplication by Rational Field
             on Univariate Polynomial Ring in x over Integer Ring
    (<class 'sage.structure.coerce_actions.RightModuleAction'>)
```

```
>>> from sage.all import *
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> cm = sage.structure.element.get_coercion_model()
>>> cm.verify_action(R.get_action(QQ), R, QQ, operator.mul)
Right scalar multiplication by Rational Field
 on Univariate Polynomial Ring in x over Integer Ring
>>> cm.verify_action(R.get_action(QQ), RDF, R, operator.mul)
Traceback (most recent call last):
...
RuntimeError: There is a BUG in the coercion model:
    Action found for R <built-in function mul> S does not have the correct␣
↪domains
    R = Real Double Field
    S = Univariate Polynomial Ring in x over Integer Ring
    (should be Univariate Polynomial Ring in x over Integer Ring, Rational␣
↪Field)
    action = Right scalar multiplication by Rational Field
             on Univariate Polynomial Ring in x over Integer Ring
    (<class 'sage.structure.coerce_actions.RightModuleAction'>)
```

verify_coercion_maps(*R*, *S*, *homs*, *fix=False*)

Make sure this is a valid pair of homomorphisms from R and S to a common parent. This function is used to protect the user against buggy parents.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: homs = QQ.coerce_map_from(ZZ), None
sage: cm.verify_coercion_maps(ZZ, QQ, homs) == homs
True
sage: homs = QQ.coerce_map_from(ZZ), RR.coerce_map_from(QQ)
sage: cm.verify_coercion_maps(ZZ, QQ, homs) == homs
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
RuntimeError: ('BUG in coercion model, codomains must be identical',
Natural morphism:
  From: Integer Ring
  To:   Rational Field,
Generic map:
  From: Rational Field
  To:   Real Field with 53 bits of precision)
```

```
>>> from sage.all import *
>>> cm = sage.structure.element.get_coercion_model()
>>> homs = QQ.coerce_map_from(ZZ), None
>>> cm.verify_coercion_maps(ZZ, QQ, homs) == homs
True
>>> homs = QQ.coerce_map_from(ZZ), RR.coerce_map_from(QQ)
>>> cm.verify_coercion_maps(ZZ, QQ, homs) == homs
Traceback (most recent call last):
...
RuntimeError: ('BUG in coercion model, codomains must be identical',
Natural morphism:
  From: Integer Ring
  To:   Rational Field,
Generic map:
  From: Rational Field
  To:   Real Field with 53 bits of precision)
```

sage.structure.coerce.**is_mpmath_type**(*t*)

Check whether the type `t` is a type whose name starts with either `mpmath.` or `sage.libs.mpmath.`.

EXAMPLES:

```
sage: # needs mpmath
sage: from sage.structure.coerce import is_mpmath_type
sage: is_mpmath_type(int)
False
sage: import mpmath
sage: is_mpmath_type(mpmath.mpc(2))
False
sage: is_mpmath_type(type(mpmath.mpc(2)))
True
sage: is_mpmath_type(type(mpmath.mpf(2)))
True
```

```
>>> from sage.all import *
>>> # needs mpmath
>>> from sage.structure.coerce import is_mpmath_type
>>> is_mpmath_type(int)
False
>>> import mpmath
>>> is_mpmath_type(mpmath.mpc(Integer(2)))
False
>>> is_mpmath_type(type(mpmath.mpc(Integer(2))))
True
```

(continues on next page)

(continued from previous page)

```
>>> is_mpmath_type(type(mpmath.mpf(Integer(2))))
True
```

sage.structure.coerce.**is_numpy_type**(*t*)

Return `True` if and only if t is a type whose name starts with `numpy`.

EXAMPLES:

```
sage: from sage.structure.coerce import is_numpy_type

sage: # needs numpy
sage: import numpy
sage: is_numpy_type(numpy.int16)
True
sage: is_numpy_type(numpy.floating)
True
sage: is_numpy_type(numpy.ndarray)
True
sage: is_numpy_type(numpy.matrix)
True

sage: is_numpy_type(int)
False
sage: is_numpy_type(Integer)
False
sage: is_numpy_type(Sudoku)                                                 #␣
↪needs sage.combinat
False
sage: is_numpy_type(None)
False
```

```
>>> from sage.all import *
>>> from sage.structure.coerce import is_numpy_type

>>> # needs numpy
>>> import numpy
>>> is_numpy_type(numpy.int16)
True
>>> is_numpy_type(numpy.floating)
True
>>> is_numpy_type(numpy.ndarray)
True
>>> is_numpy_type(numpy.matrix)
True

>>> is_numpy_type(int)
False
>>> is_numpy_type(Integer)
False
>>> is_numpy_type(Sudoku)                                                   #␣
↪needs sage.combinat
False
```

(continues on next page)

(continued from previous page)

```
>>> is_numpy_type(None)
False
```

sage.structure.coerce.**parent_is_integers**(*P*)

Check whether the type or parent represents the ring of integers.

EXAMPLES:

```
sage: from sage.structure.coerce import parent_is_integers
sage: parent_is_integers(int)
True
sage: parent_is_integers(float)
False
sage: parent_is_integers(bool)
True
sage: parent_is_integers(dict)
False

sage: import numpy                                                          #␣
↪needs numpy
sage: parent_is_integers(numpy.int16)                                       #␣
↪needs numpy
True
sage: parent_is_integers(numpy.uint64)                                      #␣
↪needs numpy
True
sage: parent_is_integers(float)
False

sage: import gmpy2
sage: parent_is_integers(gmpy2.mpz)
True
sage: parent_is_integers(gmpy2.mpq)
False
```

```
>>> from sage.all import *
>>> from sage.structure.coerce import parent_is_integers
>>> parent_is_integers(int)
True
>>> parent_is_integers(float)
False
>>> parent_is_integers(bool)
True
>>> parent_is_integers(dict)
False

>>> import numpy                                                            #␣
↪needs numpy
>>> parent_is_integers(numpy.int16)                                         #␣
↪needs numpy
True
>>> parent_is_integers(numpy.uint64)                                        #␣
```

(continues on next page)

(continued from previous page)

```
↪needs numpy
True
>>> parent_is_integers(float)
False

>>> import gmpy2
>>> parent_is_integers(gmpy2.mpz)
True
>>> parent_is_integers(gmpy2.mpq)
False
```

Ensure (Issue #27893) is fixed:

```
sage: K.<f> = QQ[]
sage: gmpy2.mpz(2) * f
2*f
```

```
>>> from sage.all import *
>>> K = QQ['f']; (f,) = K._first_ngens(1)
>>> gmpy2.mpz(Integer(2)) * f
2*f
```

sage.structure.coerce.**parent_is_numerical**(*P*)

Test if elements of the parent or type `P` can be numerically evaluated as complex numbers (in a canonical way).

EXAMPLES:

```
sage: from sage.structure.coerce import parent_is_numerical
sage: import gmpy2
sage: [parent_is_numerical(R) for R in [QQ, int, complex, gmpy2.mpc]]           #␣
↪needs sage.rings.complex_double
[True, True, True, True]
sage: [parent_is_numerical(R) for R in [RR, CC]]                                #␣
↪needs sage.rings.real_mpfr
[True, True]
sage: parent_is_numerical(QuadraticField(-1))                                   #␣
↪needs sage.rings.number_field
True
sage: import numpy; parent_is_numerical(numpy.complexfloating)                  #␣
↪needs numpy
True
sage: parent_is_numerical(SR)                                                   #␣
↪needs sage.symbolic
False
sage: [parent_is_numerical(R) for R in [QQ['x'], QQ[['x']], str]]
[False, False, False]
sage: [parent_is_numerical(R) for R in [RIF, RBF, CIF, CBF]]                    #␣
↪needs sage.libs.flint
[False, False, False, False]
```

```
>>> from sage.all import *
>>> from sage.structure.coerce import parent_is_numerical
```

(continues on next page)

(continued from previous page)

```
>>> import gmpy2
>>> [parent_is_numerical(R) for R in [QQ, int, complex, gmpy2.mpc]]           #␣
↪needs sage.rings.complex_double
[True, True, True, True]
>>> [parent_is_numerical(R) for R in [RR, CC]]                               #␣
↪needs sage.rings.real_mpfr
[True, True]
>>> parent_is_numerical(QuadraticField(-Integer(1)))                            ␣
↪     # needs sage.rings.number_field
True
>>> import numpy; parent_is_numerical(numpy.complexfloating)                 #␣
↪needs numpy
True
>>> parent_is_numerical(SR)                                                  #␣
↪needs sage.symbolic
False
>>> [parent_is_numerical(R) for R in [QQ['x'], QQ[['x']], str]]
[False, False, False]
>>> [parent_is_numerical(R) for R in [RIF, RBF, CIF, CBF]]                   #␣
↪needs sage.libs.flint
[False, False, False, False]
```

sage.structure.coerce.**parent_is_real_numerical**(*P*)

Test if elements of the parent or type `P` can be numerically evaluated as real numbers (in a canonical way).

EXAMPLES:

```
sage: from sage.structure.coerce import parent_is_real_numerical
sage: import gmpy2
sage: [parent_is_real_numerical(R) for R in [RR, QQ, ZZ, RLF, int, float, gmpy2.
↪mpq]]
[True, True, True, True, True, True, True]
sage: parent_is_real_numerical(QuadraticField(2))                            #␣
↪needs sage.rings.number_field
True
sage: import numpy; parent_is_real_numerical(numpy.integer)                  #␣
↪needs numpy
True
sage: parent_is_real_numerical(QuadraticField(-1))                           #␣
↪needs sage.rings.number_field
False
sage: [parent_is_real_numerical(R)                                           #␣
↪needs numpy
....:  for R in [CC, complex, gmpy2.mpc, numpy.complexfloating]]
[False, False, False, False]
sage: [parent_is_real_numerical(R) for R in [QQ['x'], QQ[['x']], str]]
[False, False, False]
sage: parent_is_real_numerical(SR)                                           #␣
↪needs sage.symbolic
False
sage: [parent_is_real_numerical(R) for R in [RIF, RBF, CIF, CBF]]            #␣
↪needs sage.libs.flint
[False, False, False, False]
```

```
>>> from sage.all import *
>>> from sage.structure.coerce import parent_is_real_numerical
>>> import gmpy2
>>> [parent_is_real_numerical(R) for R in [RR, QQ, ZZ, RLF, int, float, gmpy2.
↪mpq]]
[True, True, True, True, True, True, True]
>>> parent_is_real_numerical(QuadraticField(Integer(2)))                    ␣
↪     # needs sage.rings.number_field
True
>>> import numpy; parent_is_real_numerical(numpy.integer)                   #␣
↪needs numpy
True
>>> parent_is_real_numerical(QuadraticField(-Integer(1)))                   ␣
↪     # needs sage.rings.number_field
False
>>> [parent_is_real_numerical(R)                                            #␣
↪needs numpy
...  for R in [CC, complex, gmpy2.mpc, numpy.complexfloating]]
[False, False, False, False]
>>> [parent_is_real_numerical(R) for R in [QQ['x'], QQ[['x']], str]]
[False, False, False]
>>> parent_is_real_numerical(SR)                                            #␣
↪needs sage.symbolic
False
>>> [parent_is_real_numerical(R) for R in [RIF, RBF, CIF, CBF]]             #␣
↪needs sage.libs.flint
[False, False, False, False]
```

sage.structure.coerce.**py_scalar_parent**(*py_type*)

Return the Sage equivalent of the given python type, if one exists. If there is no equivalent, return `None`.

EXAMPLES:

```
sage: from sage.structure.coerce import py_scalar_parent
sage: py_scalar_parent(int)
Integer Ring
sage: py_scalar_parent(float)
Real Double Field
sage: py_scalar_parent(complex)                                             #␣
↪needs sage.rings.complex_double
Complex Double Field
sage: py_scalar_parent(bool)
Integer Ring
sage: py_scalar_parent(dict),
(None,)

sage: import fractions
sage: py_scalar_parent(fractions.Fraction)
Rational Field

sage: # needs numpy
sage: import numpy
sage: py_scalar_parent(numpy.int16)
```

(continues on next page)

(continued from previous page)

```
Integer Ring
sage: py_scalar_parent(numpy.int32)
Integer Ring
sage: py_scalar_parent(numpy.uint64)
Integer Ring
sage: py_scalar_parent(numpy.double)
Real Double Field

sage: import gmpy2
sage: py_scalar_parent(gmpy2.mpz)
Integer Ring
sage: py_scalar_parent(gmpy2.mpq)
Rational Field
sage: py_scalar_parent(gmpy2.mpfr)
Real Double Field
sage: py_scalar_parent(gmpy2.mpc)                                    #␣
↪needs sage.rings.complex_double
Complex Double Field
```

```
>>> from sage.all import *
>>> from sage.structure.coerce import py_scalar_parent
>>> py_scalar_parent(int)
Integer Ring
>>> py_scalar_parent(float)
Real Double Field
>>> py_scalar_parent(complex)                                        #␣
↪needs sage.rings.complex_double
Complex Double Field
>>> py_scalar_parent(bool)
Integer Ring
>>> py_scalar_parent(dict),
(None,)

>>> import fractions
>>> py_scalar_parent(fractions.Fraction)
Rational Field

>>> # needs numpy
>>> import numpy
>>> py_scalar_parent(numpy.int16)
Integer Ring
>>> py_scalar_parent(numpy.int32)
Integer Ring
>>> py_scalar_parent(numpy.uint64)
Integer Ring
>>> py_scalar_parent(numpy.double)
Real Double Field

>>> import gmpy2
>>> py_scalar_parent(gmpy2.mpz)
Integer Ring
>>> py_scalar_parent(gmpy2.mpq)
```

(continues on next page)

(continued from previous page)

```
Rational Field
>>> py_scalar_parent(gmpy2.mpfr)
Real Double Field
>>> py_scalar_parent(gmpy2.mpc)                                              #␣
↪needs sage.rings.complex_double
Complex Double Field
```

sage.structure.coerce.**py_scalar_to_element**(*x*)

Convert `x` to a Sage `Element` if possible.

If `x` was already an `Element` or if there is no obvious conversion possible, just return `x` itself.

EXAMPLES:

```
sage: from sage.structure.coerce import py_scalar_to_element
sage: x = py_scalar_to_element(42)
sage: x, parent(x)
(42, Integer Ring)
sage: x = py_scalar_to_element(int(42))
sage: x, parent(x)
(42, Integer Ring)
sage: x = py_scalar_to_element(float(42))
sage: x, parent(x)
(42.0, Real Double Field)
sage: x = py_scalar_to_element(complex(42))                                 #␣
↪needs sage.rings.complex_double
sage: x, parent(x)                                                          #␣
↪needs sage.rings.complex_double
(42.0, Complex Double Field)
sage: py_scalar_to_element('hello')
'hello'

sage: from fractions import Fraction
sage: f = Fraction(int(2^100), int(3^100))
sage: py_scalar_to_element(f)
1267650600228229401496703205376/515377520732011331036461129765621272702107522001
```

```
>>> from sage.all import *
>>> from sage.structure.coerce import py_scalar_to_element
>>> x = py_scalar_to_element(Integer(42))
>>> x, parent(x)
(42, Integer Ring)
>>> x = py_scalar_to_element(int(Integer(42)))
>>> x, parent(x)
(42, Integer Ring)
>>> x = py_scalar_to_element(float(Integer(42)))
>>> x, parent(x)
(42.0, Real Double Field)
>>> x = py_scalar_to_element(complex(Integer(42)))                           ␣
↪     # needs sage.rings.complex_double
>>> x, parent(x)                                                            #␣
↪needs sage.rings.complex_double
(42.0, Complex Double Field)
```

(continues on next page)

(continued from previous page)

```
>>> py_scalar_to_element('hello')
'hello'

>>> from fractions import Fraction
>>> f = Fraction(int(Integer(2)**Integer(100)), int(Integer(3)**Integer(100)))
>>> py_scalar_to_element(f)
1267650600228229401496703205376/515377520732011331036461129765621272702107522001
```

Note that bools are converted to 0 or 1:

```
sage: py_scalar_to_element(False), py_scalar_to_element(True)
(0, 1)
```

```
>>> from sage.all import *
>>> py_scalar_to_element(False), py_scalar_to_element(True)
(0, 1)
```

Test gmpy2's types:

```
sage: import gmpy2
sage: x = py_scalar_to_element(gmpy2.mpz(42))
sage: x, parent(x)
(42, Integer Ring)
sage: x = py_scalar_to_element(gmpy2.mpq('3/4'))
sage: x, parent(x)
(3/4, Rational Field)
sage: x = py_scalar_to_element(gmpy2.mpfr(42.57))
sage: x, parent(x)
(42.57, Real Double Field)
sage: x = py_scalar_to_element(gmpy2.mpc(int(42), int(42)))                    #␣
↪needs sage.rings.complex_double
sage: x, parent(x)                                                             #␣
↪needs sage.rings.complex_double
(42.0 + 42.0*I, Complex Double Field)
```

```
>>> from sage.all import *
>>> import gmpy2
>>> x = py_scalar_to_element(gmpy2.mpz(Integer(42)))
>>> x, parent(x)
(42, Integer Ring)
>>> x = py_scalar_to_element(gmpy2.mpq('3/4'))
>>> x, parent(x)
(3/4, Rational Field)
>>> x = py_scalar_to_element(gmpy2.mpfr(RealNumber('42.57')))
>>> x, parent(x)
(42.57, Real Double Field)
>>> x = py_scalar_to_element(gmpy2.mpc(int(Integer(42)), int(Integer(42))))      ␣
↪              # needs sage.rings.complex_double
>>> x, parent(x)                                                               #␣
↪needs sage.rings.complex_double
(42.0 + 42.0*I, Complex Double Field)
```

Test compatibility with *py_scalar_parent()*:

```
sage: from sage.structure.coerce import py_scalar_parent
sage: elt = [True, int(42), float(42), complex(42)]
sage: for x in elt:                                                             #␣
↪needs sage.rings.complex_double
....:     assert py_scalar_parent(type(x)) == py_scalar_to_element(x).parent()

sage: import numpy                                                              #␣
↪needs numpy
sage: elt = [numpy.int8('-12'),  numpy.uint8('143'),                            #␣
↪needs numpy
....:        numpy.int16('-33'), numpy.uint16('122'),
....:        numpy.int32('-19'), numpy.uint32('44'),
....:        numpy.int64('-3'),  numpy.uint64('552'),
....:        numpy.float16('-1.23'), numpy.float32('-2.22'),
....:        numpy.float64('-3.412'), numpy.complex64(1.2+I),
....:        numpy.complex128(-2+I)]
sage: for x in elt:                                                             #␣
↪needs numpy
....:     assert py_scalar_parent(type(x)) == py_scalar_to_element(x).parent()

sage: elt = [gmpy2.mpz(42), gmpy2.mpq('3/4'),
....:        gmpy2.mpfr(42.57), gmpy2.mpc(int(42), int(42))]
sage: for x in elt:                                                             #␣
↪needs sage.rings.complex_double
....:     assert py_scalar_parent(type(x)) == py_scalar_to_element(x).parent()
```

```
>>> from sage.all import *
>>> from sage.structure.coerce import py_scalar_parent
>>> elt = [True, int(Integer(42)), float(Integer(42)), complex(Integer(42))]
>>> for x in elt:                                                               #␣
↪needs sage.rings.complex_double
...     assert py_scalar_parent(type(x)) == py_scalar_to_element(x).parent()

>>> import numpy                                                                #␣
↪needs numpy
>>> elt = [numpy.int8('-12'),  numpy.uint8('143'),                              #␣
↪needs numpy
...        numpy.int16('-33'), numpy.uint16('122'),
...        numpy.int32('-19'), numpy.uint32('44'),
...        numpy.int64('-3'),  numpy.uint64('552'),
...        numpy.float16('-1.23'), numpy.float32('-2.22'),
...        numpy.float64('-3.412'), numpy.complex64(RealNumber('1.2')+I),
...        numpy.complex128(-Integer(2)+I)]
>>> for x in elt:                                                               #␣
↪needs numpy
...     assert py_scalar_parent(type(x)) == py_scalar_to_element(x).parent()

>>> elt = [gmpy2.mpz(Integer(42)), gmpy2.mpq('3/4'),
...        gmpy2.mpfr(RealNumber('42.57')), gmpy2.mpc(int(Integer(42)),␣
↪int(Integer(42)))]
>>> for x in elt:                                                               #␣
↪needs sage.rings.complex_double
...     assert py_scalar_parent(type(x)) == py_scalar_to_element(x).parent()
```

5.2 Coerce actions

class sage.structure.coerce_actions.**ActOnAction**

Bases: *GenericAction*

Class for actions defined via the _act_on_ method.

class sage.structure.coerce_actions.**ActedUponAction**

Bases: *GenericAction*

Class for actions defined via the _acted_upon_ method.

class sage.structure.coerce_actions.**GenericAction**

Bases: *Action*

codomain()

Return the "codomain" of this action, i.e. the Parent in which the result elements live. Typically, this should be the same as the acted upon set.

EXAMPLES:

Note that coerce actions should only be used inside of the coercion model. For this test, we need to strongly reference the domains, for otherwise they could be garbage collected, giving rise to random errors (see Issue #18157).

```
sage: M = MatrixSpace(ZZ, 2)                                                    #␣
↪needs sage.modules
sage: A = sage.structure.coerce_actions.ActedUponAction(M, Cusps, True)      #␣
↪needs sage.modular sage.modules
sage: A.codomain()                                                              #␣
↪needs sage.modular sage.modules
Set P^1(QQ) of all cusps

sage: # needs sage.groups
sage: S3 = SymmetricGroup(3)
sage: QQxyz = QQ['x,y,z']
sage: A = sage.structure.coerce_actions.ActOnAction(S3, QQxyz, False)
sage: A.codomain()
Multivariate Polynomial Ring in x, y, z over Rational Field
```

```
>>> from sage.all import *
>>> M = MatrixSpace(ZZ, Integer(2))                                             ␣
↪     # needs sage.modules
>>> A = sage.structure.coerce_actions.ActedUponAction(M, Cusps, True)      #␣
↪needs sage.modular sage.modules
>>> A.codomain()                                                              #␣
↪needs sage.modular sage.modules
Set P^1(QQ) of all cusps

>>> # needs sage.groups
>>> S3 = SymmetricGroup(Integer(3))
>>> QQxyz = QQ['x,y,z']
>>> A = sage.structure.coerce_actions.ActOnAction(S3, QQxyz, False)
>>> A.codomain()
Multivariate Polynomial Ring in x, y, z over Rational Field
```

class sage.structure.coerce_actions.**IntegerAction**

Bases: *Action*

Abstract base class representing some action by integers on something. Here, "integer" is defined loosely in the "duck typing" sense.

INPUT:

- Z – a type or parent representing integers

For the other arguments, see Action.

Note

This class is used internally in Sage's coercion model. Outside of the coercion model, special precautions are needed to prevent domains of the action from being garbage collected.

class sage.structure.coerce_actions.**IntegerMulAction**

Bases: *IntegerAction*

Implement the action $n \cdot a = a + a + ... + a$ via repeated doubling.

Both addition and negation must be defined on the set M.

INPUT:

- Z – a type or parent representing integers
- M – a ZZ-module
- m – (optional) an element of M

EXAMPLES:

```
sage: from sage.structure.coerce_actions import IntegerMulAction
sage: R.<x> = QQ['x']
sage: act = IntegerMulAction(ZZ, R)
sage: act(5, x)
5*x
sage: act(0, x)
0
sage: act(-3, x-1)
-3*x + 3
```

```
>>> from sage.all import *
>>> from sage.structure.coerce_actions import IntegerMulAction
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> act = IntegerMulAction(ZZ, R)
>>> act(Integer(5), x)
5*x
>>> act(Integer(0), x)
0
>>> act(-Integer(3), x-Integer(1))
-3*x + 3
```

class sage.structure.coerce_actions.**IntegerPowAction**

Bases: *IntegerAction*

The right action `a ^ n = a * a * ... * a` where n is an integer.

The action is implemented using the `_pow_int` method on elements.

INPUT:

- `Z` – a type or parent representing integers
- `M` – a parent whose elements implement `_pow_int`
- `m` – (optional) an element of `M`

EXAMPLES:

```
sage: from sage.structure.coerce_actions import IntegerPowAction
sage: R.<x> = LaurentSeriesRing(QQ)
sage: act = IntegerPowAction(ZZ, R)
sage: act(x, 5)
x^5
sage: act(x, -2)
x^-2
sage: act(x, int(5))
x^5
```

```
>>> from sage.all import *
>>> from sage.structure.coerce_actions import IntegerPowAction
>>> R = LaurentSeriesRing(QQ, names=('x',)); (x,) = R._first_ngens(1)
>>> act = IntegerPowAction(ZZ, R)
>>> act(x, Integer(5))
x^5
>>> act(x, -Integer(2))
x^-2
>>> act(x, int(Integer(5)))
x^5
```

class sage.structure.coerce_actions.**LeftModuleAction**

Bases: *ModuleAction*

class sage.structure.coerce_actions.**ModuleAction**

Bases: *Action*

Module action.

See also

This is an abstract class, one must actually instantiate a *LeftModuleAction* or a *RightModuleAction*.

INPUT:

- `G` – the actor, an instance of `Parent`
- `S` – the object that is acted upon
- `g` – (optional) an element of `G`
- `a` – (optional) an element of `S`
- `check` – if `True` (default), then there will be no consistency tests performed on sample elements

NOTE:

By default, the sample elements of `S` and `G` are obtained from `an_element()`, which relies on the implementation of an `_an_element_()` method. This is not always available. But usually, the action is only needed when one already *has* two elements. Hence, by Issue #14249, the coercion model will pass these two elements to the *ModuleAction* constructor.

The actual action is implemented by the `_rmul_` or `_lmul_` function on its elements. We must, however, be very particular about what we feed into these functions, because they operate under the assumption that the inputs lie exactly in the base ring and may segfault otherwise. Thus we handle all possible base extensions manually here.

codomain()

The codomain of self, which may or may not be equal to the domain.

EXAMPLES:

Note that coerce actions should only be used inside of the coercion model. For this test, we need to strongly reference the domains, for otherwise they could be garbage collected, giving rise to random errors (see Issue #18157).

```
sage: from sage.structure.coerce_actions import LeftModuleAction
sage: ZZxyz = ZZ['x,y,z']
sage: A = LeftModuleAction(QQ, ZZxyz)
sage: A.codomain()
Multivariate Polynomial Ring in x, y, z over Rational Field
```

```
>>> from sage.all import *
>>> from sage.structure.coerce_actions import LeftModuleAction
>>> ZZxyz = ZZ['x,y,z']
>>> A = LeftModuleAction(QQ, ZZxyz)
>>> A.codomain()
Multivariate Polynomial Ring in x, y, z over Rational Field
```

domain()

The domain of self, which is the module that is being acted on.

EXAMPLES:

Note that coerce actions should only be used inside of the coercion model. For this test, we need to strongly reference the domains, for otherwise they could be garbage collected, giving rise to random errors (see Issue #18157).

```
sage: from sage.structure.coerce_actions import LeftModuleAction
sage: ZZxyz = ZZ['x,y,z']
sage: A = LeftModuleAction(QQ, ZZxyz)
sage: A.domain()
Multivariate Polynomial Ring in x, y, z over Integer Ring
```

```
>>> from sage.all import *
>>> from sage.structure.coerce_actions import LeftModuleAction
>>> ZZxyz = ZZ['x,y,z']
>>> A = LeftModuleAction(QQ, ZZxyz)
>>> A.domain()
Multivariate Polynomial Ring in x, y, z over Integer Ring
```

class sage.structure.coerce_actions.**PyScalarAction**

Bases: *Action*

class sage.structure.coerce_actions.**RightModuleAction**

Bases: *ModuleAction*

sage.structure.coerce_actions.**detect_element_action**(*X*, *Y*, *X_on_left*, *X_el=None*, *Y_el=None*)

Return an action of X on Y as defined by elements of X, if any.

EXAMPLES:

Note that coerce actions should only be used inside of the coercion model. For this test, we need to strongly reference the domains, for otherwise they could be garbage collected, giving rise to random errors (see Issue #18157).

```
sage: from sage.structure.coerce_actions import detect_element_action
sage: ZZx = ZZ['x']
sage: M = MatrixSpace(ZZ, 2)                                                #␣
↪needs sage.modules
sage: detect_element_action(ZZx, ZZ, False)
Left scalar multiplication by Integer Ring
 on Univariate Polynomial Ring in x over Integer Ring
sage: detect_element_action(ZZx, QQ, True)
Right scalar multiplication by Rational Field
 on Univariate Polynomial Ring in x over Integer Ring
sage: detect_element_action(Cusps, M, False)                                #␣
↪needs sage.modular sage.modules
Left action by Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
 on Set P^1(QQ) of all cusps
sage: detect_element_action(Cusps, M, True),                                #␣
↪needs sage.modular sage.modules
(None,)
sage: detect_element_action(ZZ, QQ, True),
(None,)
```

```
>>> from sage.all import *
>>> from sage.structure.coerce_actions import detect_element_action
>>> ZZx = ZZ['x']
>>> M = MatrixSpace(ZZ, Integer(2))                                          ␣
↪     # needs sage.modules
>>> detect_element_action(ZZx, ZZ, False)
Left scalar multiplication by Integer Ring
 on Univariate Polynomial Ring in x over Integer Ring
>>> detect_element_action(ZZx, QQ, True)
Right scalar multiplication by Rational Field
 on Univariate Polynomial Ring in x over Integer Ring
>>> detect_element_action(Cusps, M, False)                                  #␣
↪needs sage.modular sage.modules
Left action by Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
 on Set P^1(QQ) of all cusps
>>> detect_element_action(Cusps, M, True),                                  #␣
↪needs sage.modular sage.modules
(None,)
>>> detect_element_action(ZZ, QQ, True),
(None,)
```

5.3 Coerce maps

class sage.structure.coerce_maps.**CCallableConvertMap_class**

Bases: Map

class sage.structure.coerce_maps.**CallableConvertMap**

Bases: Map

This lets one easily create maps from any callable object.

This is especially useful to create maps from bound methods.

EXAMPLES:

```
sage: from sage.structure.coerce_maps import CallableConvertMap
sage: def foo(P, x): return x/2
sage: f = CallableConvertMap(ZZ, QQ, foo)
sage: f(3)
3/2
sage: f
Conversion via foo map:
  From: Integer Ring
  To:   Rational Field
```

```
>>> from sage.all import *
>>> from sage.structure.coerce_maps import CallableConvertMap
>>> def foo(P, x): return x/Integer(2)
>>> f = CallableConvertMap(ZZ, QQ, foo)
>>> f(Integer(3))
3/2
>>> f
Conversion via foo map:
  From: Integer Ring
  To:   Rational Field
```

Create a homomorphism from $\mathbf{R}$ to $\mathbf{R}^+$ viewed as additive groups.

```
sage: # needs sage.symbolic
sage: f = CallableConvertMap(RR, RR, exp, parent_as_first_arg=False)
sage: f(0)
1.00000000000000
sage: f(1)
2.71828182845905
sage: f(-3)
0.0497870683678639
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> f = CallableConvertMap(RR, RR, exp, parent_as_first_arg=False)
>>> f(Integer(0))
1.00000000000000
>>> f(Integer(1))
2.71828182845905
>>> f(-Integer(3))
0.0497870683678639
```

class sage.structure.coerce_maps.**DefaultConvertMap**

Bases: Map

This morphism simply calls the codomain's element_constructor method, passing in the codomain as the first argument.

EXAMPLES:

```
sage: QQ[['x']].coerce_map_from(QQ)
Coercion map:
  From: Rational Field
  To:   Power Series Ring in x over Rational Field
```

```
>>> from sage.all import *
>>> QQ[['x']].coerce_map_from(QQ)
Coercion map:
  From: Rational Field
  To:   Power Series Ring in x over Rational Field
```

class sage.structure.coerce_maps.**DefaultConvertMap_unique**

Bases: *DefaultConvertMap*

This morphism simply defers action to the codomain's element_constructor method, WITHOUT passing in the codomain as the first argument.

This is used for creating elements that don't take a parent as the first argument to their __init__ method, for example, Integers, Rationals, Algebraic Reals… all have a unique parent. It is also used when the element_constructor is a bound method (whose self argument is assumed to be bound to the codomain).

class sage.structure.coerce_maps.**ListMorphism**

Bases: Map

class sage.structure.coerce_maps.**NamedConvertMap**

Bases: Map

This is used for creating elements via the _xxx_ methods.

For example, many elements implement an _integer_ method to convert to ZZ, or a _rational_ method to convert to QQ.

method_name

class sage.structure.coerce_maps.**TryMap**

Bases: Map

sage.structure.coerce_maps.**test_CCallableConvertMap**(*domain*, *name=None*)

For testing CCallableConvertMap_class.

5.4 Coercion via construction functors

class sage.categories.pushout.**AlgebraicClosureFunctor**

Bases: *ConstructionFunctor*

Algebraic Closure.

EXAMPLES:

```
sage: # needs sage.rings.complex_double sage.rings.number_field
sage: F = CDF.construction()[0]
sage: F(QQ)
Algebraic Field
sage: F(RR)                                                                 #␣
↪needs sage.rings.real_mpfr
Complex Field with 53 bits of precision
sage: F(F(QQ)) is F(QQ)
True
```

```
>>> from sage.all import *
>>> # needs sage.rings.complex_double sage.rings.number_field
>>> F = CDF.construction()[Integer(0)]
>>> F(QQ)
Algebraic Field
>>> F(RR)                                                                   #␣
↪needs sage.rings.real_mpfr
Complex Field with 53 bits of precision
>>> F(F(QQ)) is F(QQ)
True
```

merge(*other*)

Mathematically, Algebraic Closure subsumes Algebraic Extension. However, it seems that people do want to work with algebraic extensions of `RR`. Therefore, we do not merge with algebraic extension.

rank = 3

class sage.categories.pushout.**AlgebraicExtensionFunctor**(*polys*, *names*, *embeddings=None*, *structures=None*, *cyclotomic=None*, *precs=None*, *implementations=None*, *, *residue=None*, *latex_names=None*, ***kwds*)

Bases: *ConstructionFunctor*

Algebraic extension (univariate polynomial ring modulo principal ideal).

EXAMPLES:

```
sage: x = polygen(QQ, 'x')
sage: K.<a> = NumberField(x^3 + x^2 + 1)                                    #␣
↪needs sage.rings.number_field
sage: F = K.construction()[0]                                               #␣
↪needs sage.rings.number_field
sage: F(ZZ['t'])                                                            #␣
↪needs sage.rings.number_field
Univariate Quotient Polynomial Ring in a
 over Univariate Polynomial Ring in t over Integer Ring
 with modulus a^3 + a^2 + 1
```

```
>>> from sage.all import *
>>> x = polygen(QQ, 'x')
>>> K = NumberField(x**Integer(3) + x**Integer(2) + Integer(1), names=('a',)); (a,
↪) = K._first_ngens(1)# needs sage.rings.number_field
```

(continues on next page)

(continued from previous page)

```
>>> F = K.construction()[Integer(0)]                                    ␣
↪     # needs sage.rings.number_field
>>> F(ZZ['t'])                                                          #␣
↪needs sage.rings.number_field
Univariate Quotient Polynomial Ring in a
 over Univariate Polynomial Ring in t over Integer Ring
 with modulus a^3 + a^2 + 1
```

Note that, even if a field is algebraically closed, the algebraic extension will be constructed as the quotient of a univariate polynomial ring:

```
sage: F(CC)                                                             #␣
↪needs sage.rings.number_field
Univariate Quotient Polynomial Ring in a
 over Complex Field with 53 bits of precision
 with modulus a^3 + a^2 + 1.00000000000000
sage: F(RR)                                                             #␣
↪needs sage.rings.number_field
Univariate Quotient Polynomial Ring in a
 over Real Field with 53 bits of precision
 with modulus a^3 + a^2 + 1.00000000000000
```

```
>>> from sage.all import *
>>> F(CC)                                                               #␣
↪needs sage.rings.number_field
Univariate Quotient Polynomial Ring in a
 over Complex Field with 53 bits of precision
 with modulus a^3 + a^2 + 1.00000000000000
>>> F(RR)                                                               #␣
↪needs sage.rings.number_field
Univariate Quotient Polynomial Ring in a
 over Real Field with 53 bits of precision
 with modulus a^3 + a^2 + 1.00000000000000
```

Note that the construction functor of a number field applied to the integers returns an order (not necessarily maximal) of that field, similar to the behaviour of `ZZ.extension(...)`:

```
sage: F(ZZ)                                                             #␣
↪needs sage.rings.number_field
Order generated by a in Number Field in a with defining polynomial x^3 + x^2 + 1
```

```
>>> from sage.all import *
>>> F(ZZ)                                                               #␣
↪needs sage.rings.number_field
Order generated by a in Number Field in a with defining polynomial x^3 + x^2 + 1
```

This also holds for non-absolute number fields:

```
sage: # needs sage.rings.number_field
sage: x = polygen(QQ, 'x')
sage: K.<a,b> = NumberField([x^3 + x^2 + 1, x^2 + x + 1])
sage: F = K.construction()[0]
```

(continues on next page)

(continued from previous page)

```
sage: O = F(ZZ); O
Relative Order
 generated by [(b - 2)*a^2 + (3*b - 1)*a + 3*b + 4, a - b]
 in Number Field in a with defining polynomial x^3 + x^2 + 1
 over its base field
sage: O.ambient() is K
True
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> x = polygen(QQ, 'x')
>>> K = NumberField([x**Integer(3) + x**Integer(2) + Integer(1), x**Integer(2) +␣
↪x + Integer(1)], names=('a', 'b',)); (a, b,) = K._first_ngens(2)
>>> F = K.construction()[Integer(0)]
>>> O = F(ZZ); O
Relative Order
 generated by [(b - 2)*a^2 + (3*b - 1)*a + 3*b + 4, a - b]
 in Number Field in a with defining polynomial x^3 + x^2 + 1
 over its base field
>>> O.ambient() is K
True
```

Special cases are made for cyclotomic fields and residue fields:

```
sage: # needs sage.rings.number_field
sage: C = CyclotomicField(8)
sage: F, R = C.construction()
sage: F
AlgebraicExtensionFunctor
sage: R
Rational Field
sage: F(R)
Cyclotomic Field of order 8 and degree 4
sage: F(ZZ)
Maximal Order generated by zeta8 in Cyclotomic Field of order 8 and degree 4
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> C = CyclotomicField(Integer(8))
>>> F, R = C.construction()
>>> F
AlgebraicExtensionFunctor
>>> R
Rational Field
>>> F(R)
Cyclotomic Field of order 8 and degree 4
>>> F(ZZ)
Maximal Order generated by zeta8 in Cyclotomic Field of order 8 and degree 4
```

```
sage: # needs sage.rings.number_field
sage: K.<z> = CyclotomicField(7)
```

(continues on next page)

(continued from previous page)

```
sage: P = K.factor(17)[0][0]
sage: k = K.residue_field(P)
sage: F, R = k.construction()
sage: F
AlgebraicExtensionFunctor
sage: R
Cyclotomic Field of order 7 and degree 6
sage: F(R) is k
True
sage: F(ZZ)
Residue field of Integers modulo 17
sage: F(CyclotomicField(49))
Residue field in zbar of Fractional ideal (17)
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> K = CyclotomicField(Integer(7), names=('z',)); (z,) = K._first_ngens(1)
>>> P = K.factor(Integer(17))[Integer(0)][Integer(0)]
>>> k = K.residue_field(P)
>>> F, R = k.construction()
>>> F
AlgebraicExtensionFunctor
>>> R
Cyclotomic Field of order 7 and degree 6
>>> F(R) is k
True
>>> F(ZZ)
Residue field of Integers modulo 17
>>> F(CyclotomicField(Integer(49)))
Residue field in zbar of Fractional ideal (17)
```

expand()

Decompose the functor F into sub-functors, whose product returns F.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: P.<x> = QQ[]
sage: K.<a> = NumberField(x^3 - 5, embedding=0)
sage: L.<b> = K.extension(x^2 + a)
sage: F, R = L.construction()
sage: prod(F.expand())(R) == L
True
sage: K = NumberField([x^2 - 2, x^2 - 3],'a')
sage: F, R = K.construction()
sage: F
AlgebraicExtensionFunctor
sage: L = F.expand(); L
[AlgebraicExtensionFunctor, AlgebraicExtensionFunctor]
sage: L[-1](QQ)
Number Field in a1 with defining polynomial x^2 - 3
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> P = QQ['x']; (x,) = P._first_ngens(1)
>>> K = NumberField(x**Integer(3) - Integer(5), embedding=Integer(0), names=(
↪'a',)); (a,) = K._first_ngens(1)
>>> L = K.extension(x**Integer(2) + a, names=('b',)); (b,) = L._first_ngens(1)
>>> F, R = L.construction()
>>> prod(F.expand())(R) == L
True
>>> K = NumberField([x**Integer(2) - Integer(2), x**Integer(2) - Integer(3)],
↪'a')
>>> F, R = K.construction()
>>> F
AlgebraicExtensionFunctor
>>> L = F.expand(); L
[AlgebraicExtensionFunctor, AlgebraicExtensionFunctor]
>>> L[-Integer(1)](QQ)
Number Field in a1 with defining polynomial x^2 - 3
```

merge(*other*)

Merging with another *AlgebraicExtensionFunctor*.

INPUT:

- `other` – Construction Functor

OUTPUT:

- If `self==other`, `self` is returned.
- If `self` and `other` are simple extensions and both provide an embedding, then it is tested whether one of the number fields provided by the functors coerces into the other; the functor associated with the target of the coercion is returned. Otherwise, the construction functor associated with the pushout of the codomains of the two embeddings is returned, provided that it is a number field.
- If these two extensions are defined by Conway polynomials over finite fields, merges them into a single extension of degree the lcm of the two degrees.
- Otherwise, `None` is returned.

REMARK:

Algebraic extension with embeddings currently only works when applied to the rational field. This is why we use the admittedly strange rule above for merging.

EXAMPLES:

The following demonstrate coercions for finite fields using Conway or pseudo-Conway polynomials:

```
sage: k = GF(3^2, prefix='z'); a = k.gen()                                  #␣
↪needs sage.rings.finite_rings
sage: l = GF(3^3, prefix='z'); b = l.gen()                                  #␣
↪needs sage.rings.finite_rings
sage: a + b  # indirect doctest                                             #␣
↪needs sage.rings.finite_rings
z6^5 + 2*z6^4 + 2*z6^3 + z6^2 + 2*z6 + 1
```

```
>>> from sage.all import *
>>> k = GF(Integer(3)**Integer(2), prefix='z'); a = k.gen()
↪               # needs sage.rings.finite_rings
>>> l = GF(Integer(3)**Integer(3), prefix='z'); b = l.gen()
↪               # needs sage.rings.finite_rings
>>> a + b  # indirect doctest
↪needs sage.rings.finite_rings
z6^5 + 2*z6^4 + 2*z6^3 + z6^2 + 2*z6 + 1
```

Note that embeddings are compatible in lattices of such finite fields:

```
sage: # needs sage.rings.finite_rings
sage: m = GF(3^5, prefix='z'); c = m.gen()
sage: (a + b) + c == a + (b + c) # indirect doctest
True
sage: from sage.categories.pushout import pushout
sage: n = pushout(k, l)
sage: o = pushout(l, m)
sage: q = pushout(n, o)
sage: q(o(b)) == q(n(b)) # indirect doctest
True
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> m = GF(Integer(3)**Integer(5), prefix='z'); c = m.gen()
>>> (a + b) + c == a + (b + c) # indirect doctest
True
>>> from sage.categories.pushout import pushout
>>> n = pushout(k, l)
>>> o = pushout(l, m)
>>> q = pushout(n, o)
>>> q(o(b)) == q(n(b)) # indirect doctest
True
```

Coercion is also available for number fields:

```
sage: # needs sage.rings.number_field
sage: P.<x> = QQ[]
sage: L.<b> = NumberField(x^8 - x^4 + 1, embedding=CDF.0)
sage: M1.<c1> = NumberField(x^2 + x + 1, embedding=b^4 - 1)
sage: M2.<c2> = NumberField(x^2 + 1, embedding=-b^6)
sage: M1.coerce_map_from(M2)
sage: M2.coerce_map_from(M1)
sage: c1 + c2; parent(c1 + c2)    #indirect doctest
-b^6 + b^4 - 1
Number Field in b with defining polynomial x^8 - x^4 + 1
 with b = -0.2588190451025208? + 0.9659258262890683?*I
sage: pushout(M1['x'], M2['x'])
↪needs sage.rings.finite_rings
Univariate Polynomial Ring in x
 over Number Field in b with defining polynomial x^8 - x^4 + 1
  with b = -0.2588190451025208? + 0.9659258262890683?*I
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> P = QQ['x']; (x,) = P._first_ngens(1)
>>> L = NumberField(x**Integer(8) - x**Integer(4) + Integer(1), embedding=CDF.
↪gen(0), names=('b',)); (b,) = L._first_ngens(1)
>>> M1 = NumberField(x**Integer(2) + x + Integer(1), embedding=b**Integer(4) -
↪ Integer(1), names=('c1',)); (c1,) = M1._first_ngens(1)
>>> M2 = NumberField(x**Integer(2) + Integer(1), embedding=-b**Integer(6),␣
↪names=('c2',)); (c2,) = M2._first_ngens(1)
>>> M1.coerce_map_from(M2)
>>> M2.coerce_map_from(M1)
>>> c1 + c2; parent(c1 + c2)    #indirect doctest
-b^6 + b^4 - 1
Number Field in b with defining polynomial x^8 - x^4 + 1
 with b = -0.2588190451025208? + 0.9659258262890683?*I
>>> pushout(M1['x'], M2['x'])                                                #␣
↪needs sage.rings.finite_rings
Univariate Polynomial Ring in x
 over Number Field in b with defining polynomial x^8 - x^4 + 1
  with b = -0.2588190451025208? + 0.9659258262890683?*I
```

In the previous example, the number field `L` becomes the pushout of `M1` and `M2` since both are provided with an embedding into `L`, *and* since `L` is a number field. If two number fields are embedded into a field that is not a numberfield, no merging occurs:

```
sage: # needs sage.rings.complex_double sage.rings.number_field
sage: cbrt2 = CDF(2)^(1/3)
sage: zeta3 = CDF.zeta(3)
sage: K.<a> = NumberField(x^3 - 2, embedding=cbrt2 * zeta3)
sage: L.<b> = NumberField(x^6 - 2, embedding=1.1)
sage: L.coerce_map_from(K)
sage: K.coerce_map_from(L)
sage: pushout(K, L)                                                       #␣
↪needs sage.rings.finite_rings
Traceback (most recent call last):
...
CoercionException: ('Ambiguous Base Extension', Number Field in a with
defining polynomial x^3 - 2 with a = -0.6299605249474365? + 1.091123635971722?
↪*I,
Number Field in b with defining polynomial x^6 - 2 with b = 1.122462048309373?
↪)
```

```
>>> from sage.all import *
>>> # needs sage.rings.complex_double sage.rings.number_field
>>> cbrt2 = CDF(Integer(2))**(Integer(1)/Integer(3))
>>> zeta3 = CDF.zeta(Integer(3))
>>> K = NumberField(x**Integer(3) - Integer(2), embedding=cbrt2 * zeta3,␣
↪names=('a',)); (a,) = K._first_ngens(1)
>>> L = NumberField(x**Integer(6) - Integer(2), embedding=RealNumber('1.1'),␣
↪names=('b',)); (b,) = L._first_ngens(1)
>>> L.coerce_map_from(K)
>>> K.coerce_map_from(L)
>>> pushout(K, L)                                                           #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.rings.finite_rings
Traceback (most recent call last):
...
CoercionException: ('Ambiguous Base Extension', Number Field in a with
defining polynomial x^3 - 2 with a = -0.6299605249474365? + 1.091123635971722?
↪*I,
Number Field in b with defining polynomial x^6 - 2 with b = 1.122462048309373?
↪)
```

rank = 3

class sage.categories.pushout.**BlackBoxConstructionFunctor**(*box*)

Bases: *ConstructionFunctor*

Construction functor obtained from any callable object.

EXAMPLES:

```
sage: from sage.categories.pushout import BlackBoxConstructionFunctor

sage: # needs sage.libs.gap
sage: from sage.interfaces.gap import gap
sage: FG = BlackBoxConstructionFunctor(gap)
sage: FG
BlackBoxConstructionFunctor
sage: FG(ZZ)
Integers
sage: FG(ZZ).parent()
Gap
sage: FG == loads(dumps(FG))
True

sage: FS = BlackBoxConstructionFunctor(singular)
sage: FS(QQ['t'])                                                           #␣
↪needs sage.libs.singular
polynomial ring, over a field, global ordering
// coefficients: QQ...
// number of vars : 1
//        block   1 : ordering lp
//                  : names    t
//        block   2 : ordering C
sage: FG == FS                                                              #␣
↪needs sage.libs.gap sage.libs.singular
False
```

```
>>> from sage.all import *
>>> from sage.categories.pushout import BlackBoxConstructionFunctor

>>> # needs sage.libs.gap
>>> from sage.interfaces.gap import gap
>>> FG = BlackBoxConstructionFunctor(gap)
>>> FG
BlackBoxConstructionFunctor
```

(continues on next page)

(continued from previous page)

```
>>> FG(ZZ)
Integers
>>> FG(ZZ).parent()
Gap
>>> FG == loads(dumps(FG))
True

>>> FS = BlackBoxConstructionFunctor(singular)
>>> FS(QQ['t'])                                                    #␣
↪needs sage.libs.singular
polynomial ring, over a field, global ordering
// coefficients: QQ...
// number of vars : 1
//        block   1 : ordering lp
//                  : names    t
//        block   2 : ordering C
>>> FG == FS                                                       #␣
↪needs sage.libs.gap sage.libs.singular
False
```

rank = 100

class sage.categories.pushout.**CompletionFunctor**(*p*, *prec*, *extras=None*)

Bases: *ConstructionFunctor*

Completion of a ring with respect to a given prime (including infinity).

EXAMPLES:

```
sage: # needs sage.rings.padics
sage: R = Zp(5)
sage: R
5-adic Ring with capped relative precision 20
sage: F1 = R.construction()[0]
sage: F1
Completion[5, prec=20]
sage: F1(ZZ) is R
True
sage: F1(QQ)
5-adic Field with capped relative precision 20

sage: F2 = RR.construction()[0]
sage: F2
Completion[+Infinity, prec=53]
sage: F2(QQ) is RR
True

sage: P.<x> = ZZ[]
sage: Px = P.completion(x) # currently the only implemented completion of P
sage: Px
Power Series Ring in x over Integer Ring
sage: F3 = Px.construction()[0]
sage: F3(GF(3)['x'])
Power Series Ring in x over Finite Field of size 3
```

```
>>> from sage.all import *
>>> # needs sage.rings.padics
>>> R = Zp(Integer(5))
>>> R
5-adic Ring with capped relative precision 20
>>> F1 = R.construction()[Integer(0)]
>>> F1
Completion[5, prec=20]
>>> F1(ZZ) is R
True
>>> F1(QQ)
5-adic Field with capped relative precision 20

>>> F2 = RR.construction()[Integer(0)]
>>> F2
Completion[+Infinity, prec=53]
>>> F2(QQ) is RR
True

>>> P = ZZ['x']; (x,) = P._first_ngens(1)
>>> Px = P.completion(x) # currently the only implemented completion of P
>>> Px
Power Series Ring in x over Integer Ring
>>> F3 = Px.construction()[Integer(0)]
>>> F3(GF(Integer(3))['x'])
Power Series Ring in x over Finite Field of size 3
```

commutes(*other*)

Completion commutes with fraction fields.

EXAMPLES:

```
sage: F1 = Zp(5).construction()[0]                                              #␣
↪needs sage.rings.padics
sage: F2 = QQ.construction()[0]
sage: F1.commutes(F2)                                                           #␣
↪needs sage.rings.padics
True
```

```
>>> from sage.all import *
>>> F1 = Zp(Integer(5)).construction()[Integer(0)]                                ␣
↪              # needs sage.rings.padics
>>> F2 = QQ.construction()[Integer(0)]
>>> F1.commutes(F2)                                                            #␣
↪needs sage.rings.padics
True
```

merge(*other*)

Two Completion functors are merged, if they are equal. If the precisions of both functors coincide, then a Completion functor is returned that results from updating the `extras` dictionary of `self` by `other.extras`. Otherwise, if the completion is at infinity then merging does not increase the set precision, and if the completion is at a finite prime, merging does not decrease the capped precision.

EXAMPLES:

```
sage: # needs sage.rings.padics
sage: R1.<a> = Zp(5, prec=20)[]
sage: R2 = Qp(5, prec=40)
sage: R2(1) + a       # indirect doctest
(1 + O(5^20))*a + 1 + O(5^40)
sage: R3 = RealField(30)
sage: R4 = RealField(50)
sage: R3(1) + R4(1)   # indirect doctest
2.0000000
sage: (R3(1) + R4(1)).parent()
Real Field with 30 bits of precision
```

```
>>> from sage.all import *
>>> # needs sage.rings.padics
>>> R1 = Zp(Integer(5), prec=Integer(20))['a']; (a,) = R1._first_ngens(1)
>>> R2 = Qp(Integer(5), prec=Integer(40))
>>> R2(Integer(1)) + a       # indirect doctest
(1 + O(5^20))*a + 1 + O(5^40)
>>> R3 = RealField(Integer(30))
>>> R4 = RealField(Integer(50))
>>> R3(Integer(1)) + R4(Integer(1))    # indirect doctest
2.0000000
>>> (R3(Integer(1)) + R4(Integer(1))).parent()
Real Field with 30 bits of precision
```

rank = 4

class sage.categories.pushout.**CompositeConstructionFunctor**(**args*)

Bases: *ConstructionFunctor*

A Construction Functor composed by other Construction Functors.

INPUT:

- `F1, F2, ...` – a list of Construction Functors. The result is the composition `F1` followed by `F2` followed by …

EXAMPLES:

```
sage: from sage.categories.pushout import CompositeConstructionFunctor
sage: F = CompositeConstructionFunctor(QQ.construction()[0], ZZ['x'].
↪construction()[0],
....:                                  QQ.construction()[0], ZZ['y'].
↪construction()[0])
sage: F
Poly[y](FractionField(Poly[x](FractionField(...))))
sage: F == loads(dumps(F))
True
sage: F == CompositeConstructionFunctor(*F.all)
True
sage: F(GF(2)['t'])                                                    #␣
↪needs sage.libs.ntl
Univariate Polynomial Ring in y
 over Fraction Field of Univariate Polynomial Ring in x
```

(continues on next page)

(continued from previous page)

```
  over Fraction Field of Univariate Polynomial Ring in t
   over Finite Field of size 2 (using GF2X)
```

```
>>> from sage.all import *
>>> from sage.categories.pushout import CompositeConstructionFunctor
>>> F = CompositeConstructionFunctor(QQ.construction()[Integer(0)], ZZ['x'].
↪construction()[Integer(0)],
...                                   QQ.construction()[Integer(0)], ZZ['y'].
↪construction()[Integer(0)])
>>> F
Poly[y](FractionField(Poly[x](FractionField(...))))
>>> F == loads(dumps(F))
True
>>> F == CompositeConstructionFunctor(*F.all)
True
>>> F(GF(Integer(2))['t'])                                                    ␣
↪     # needs sage.libs.ntl
Univariate Polynomial Ring in y
 over Fraction Field of Univariate Polynomial Ring in x
  over Fraction Field of Univariate Polynomial Ring in t
   over Finite Field of size 2 (using GF2X)
```

expand()

Return expansion of a CompositeConstructionFunctor.

Note

The product over the list of components, as returned by the `expand()` method, is equal to `self`.

EXAMPLES:

```
sage: from sage.categories.pushout import CompositeConstructionFunctor
sage: F = CompositeConstructionFunctor(QQ.construction()[0],
....:                                  ZZ['x'].construction()[0],
....:                                  QQ.construction()[0],
....:                                  ZZ['y'].construction()[0])
sage: F
Poly[y](FractionField(Poly[x](FractionField(...))))
sage: prod(F.expand()) == F
True
```

```
>>> from sage.all import *
>>> from sage.categories.pushout import CompositeConstructionFunctor
>>> F = CompositeConstructionFunctor(QQ.construction()[Integer(0)],
...                                  ZZ['x'].construction()[Integer(0)],
...                                  QQ.construction()[Integer(0)],
...                                  ZZ['y'].construction()[Integer(0)])
>>> F
Poly[y](FractionField(Poly[x](FractionField(...))))
>>> prod(F.expand()) == F
True
```

class sage.categories.pushout.**ConstructionFunctor**

Bases: Functor

Base class for construction functors.

A construction functor is a functorial algebraic construction, such as the construction of a matrix ring over a given ring or the fraction field of a given ring.

In addition to the class Functor, construction functors provide rules for combining and merging constructions. This is an important part of Sage's coercion model, namely the pushout of two constructions: When a polynomial p in a variable x with integer coefficients is added to a rational number q, then Sage finds that the parents ZZ['x'] and QQ are obtained from ZZ by applying a polynomial ring construction respectively the fraction field construction. Each construction functor has an attribute rank, and the rank of the polynomial ring construction is higher than the rank of the fraction field construction. This means that the pushout of QQ and ZZ['x'], and thus a common parent in which p and q can be added, is QQ['x'], since the construction functor with a lower rank is applied first.

```
sage: F1, R = QQ.construction()
sage: F1
FractionField
sage: R
Integer Ring
sage: F2, R = (ZZ['x']).construction()
sage: F2
Poly[x]
sage: R
Integer Ring
sage: F3 = F2.pushout(F1)
sage: F3
Poly[x](FractionField(...))
sage: F3(R)
Univariate Polynomial Ring in x over Rational Field
sage: from sage.categories.pushout import pushout
sage: P.<x> = ZZ[]
sage: pushout(QQ,P)
Univariate Polynomial Ring in x over Rational Field
sage: ((x+1) + 1/2).parent()
Univariate Polynomial Ring in x over Rational Field
```

```
>>> from sage.all import *
>>> F1, R = QQ.construction()
>>> F1
FractionField
>>> R
Integer Ring
>>> F2, R = (ZZ['x']).construction()
>>> F2
Poly[x]
>>> R
Integer Ring
>>> F3 = F2.pushout(F1)
>>> F3
Poly[x](FractionField(...))
>>> F3(R)
Univariate Polynomial Ring in x over Rational Field
```

(continues on next page)

(continued from previous page)

```
>>> from sage.categories.pushout import pushout
>>> P = ZZ['x']; (x,) = P._first_ngens(1)
>>> pushout(QQ,P)
Univariate Polynomial Ring in x over Rational Field
>>> ((x+Integer(1)) + Integer(1)/Integer(2)).parent()
Univariate Polynomial Ring in x over Rational Field
```

When composing two construction functors, they are sometimes merged into one, as is the case in the Quotient construction:

```
sage: Q15, R = (ZZ.quo(15*ZZ)).construction()
sage: Q15
QuotientFunctor
sage: Q35, R = (ZZ.quo(35*ZZ)).construction()
sage: Q35
QuotientFunctor
sage: Q15.merge(Q35)
QuotientFunctor
sage: Q15.merge(Q35)(ZZ)
Ring of integers modulo 5
```

```
>>> from sage.all import *
>>> Q15, R = (ZZ.quo(Integer(15)*ZZ)).construction()
>>> Q15
QuotientFunctor
>>> Q35, R = (ZZ.quo(Integer(35)*ZZ)).construction()
>>> Q35
QuotientFunctor
>>> Q15.merge(Q35)
QuotientFunctor
>>> Q15.merge(Q35)(ZZ)
Ring of integers modulo 5
```

Functors can not only be applied to objects, but also to morphisms in the respective categories. For example:

```
sage: P.<x,y> = ZZ[]
sage: F = P.construction()[0]; F
MPoly[x,y]
sage: A.<a,b> = GF(5)[]
sage: f = A.hom([a + b, a - b], A)
sage: F(A)
Multivariate Polynomial Ring in x, y
 over Multivariate Polynomial Ring in a, b over Finite Field of size 5
sage: F(f)
Ring endomorphism of Multivariate Polynomial Ring in x, y
 over Multivariate Polynomial Ring in a, b over Finite Field of size 5
  Defn: Induced from base ring by
        Ring endomorphism of Multivariate Polynomial Ring in a, b
         over Finite Field of size 5
          Defn: a |--> a + b
                b |--> a - b
sage: F(f)(F(A)(x)*a)
```

(continues on next page)

(continued from previous page)

```
(a + b)*x
```

```
>>> from sage.all import *
>>> P = ZZ['x, y']; (x, y,) = P._first_ngens(2)
>>> F = P.construction()[Integer(0)]; F
MPoly[x,y]
>>> A = GF(Integer(5))['a, b']; (a, b,) = A._first_ngens(2)
>>> f = A.hom([a + b, a - b], A)
>>> F(A)
Multivariate Polynomial Ring in x, y
 over Multivariate Polynomial Ring in a, b over Finite Field of size 5
>>> F(f)
Ring endomorphism of Multivariate Polynomial Ring in x, y
 over Multivariate Polynomial Ring in a, b over Finite Field of size 5
  Defn: Induced from base ring by
        Ring endomorphism of Multivariate Polynomial Ring in a, b
         over Finite Field of size 5
          Defn: a |--> a + b
                b |--> a - b
>>> F(f)(F(A)(x)*a)
(a + b)*x
```

coercion_reversed = False

common_base(*other_functor*, *self_bases*, *other_bases*)

This function is called by *pushout()* when no common parent is found in the construction tower.

> **Note**
>
> The main use is for multivariate construction functors, which use this function to implement recursion for *pushout()*.

INPUT:

- `other_functor` – a construction functor
- `self_bases` – the arguments passed to this functor
- `other_bases` – the arguments passed to the functor `other_functor`

OUTPUT:

Nothing, since a *CoercionException* is raised.

> **Note**
>
> Overload this function in derived class, see e.e. *MultivariateConstructionFunctor*.

commutes(*other*)

Determine whether `self` commutes with another construction functor.

Note

By default, `False` is returned in all cases (even if the two functors are the same, since in this case *`merge()`* will apply anyway). So far there is no construction functor that overloads this method. Anyway, this method only becomes relevant if two construction functors have the same rank.

EXAMPLES:

```
sage: F = QQ.construction()[0]
sage: P = ZZ['t'].construction()[0]
sage: F.commutes(P)
False
sage: P.commutes(F)
False
sage: F.commutes(F)
False
```

```
>>> from sage.all import *
>>> F = QQ.construction()[Integer(0)]
>>> P = ZZ['t'].construction()[Integer(0)]
>>> F.commutes(P)
False
>>> P.commutes(F)
False
>>> F.commutes(F)
False
```

expand()

Decompose `self` into a list of construction functors.

Note

The default is to return the list only containing `self`.

EXAMPLES:

```
sage: F = QQ.construction()[0]
sage: F.expand()
[FractionField]
sage: Q = ZZ.quo(2).construction()[0]
sage: Q.expand()
[QuotientFunctor]
sage: P = ZZ['t'].construction()[0]
sage: FP = F*P
sage: FP.expand()
[FractionField, Poly[t]]
```

```
>>> from sage.all import *
>>> F = QQ.construction()[Integer(0)]
>>> F.expand()
[FractionField]
```

(continues on next page)

(continued from previous page)

```
>>> Q = ZZ.quo(Integer(2)).construction()[Integer(0)]
>>> Q.expand()
[QuotientFunctor]
>>> P = ZZ['t'].construction()[Integer(0)]
>>> FP = F*P
>>> FP.expand()
[FractionField, Poly[t]]
```

merge(*other*)

Merge `self` with another construction functor, or return `None`.

Note

The default is to merge only if the two functors coincide. But this may be overloaded for subclasses, such as the quotient functor.

EXAMPLES:

```
sage: F = QQ.construction()[0]
sage: P = ZZ['t'].construction()[0]
sage: F.merge(F)
FractionField
sage: F.merge(P)
sage: P.merge(F)
sage: P.merge(P)
Poly[t]
```

```
>>> from sage.all import *
>>> F = QQ.construction()[Integer(0)]
>>> P = ZZ['t'].construction()[Integer(0)]
>>> F.merge(F)
FractionField
>>> F.merge(P)
>>> P.merge(F)
>>> P.merge(P)
Poly[t]
```

pushout(*other*)

Composition of two construction functors, ordered by their ranks.

Note

- This method seems not to be used in the coercion model.
- By default, the functor with smaller rank is applied first.

class sage.categories.pushout.**EquivariantSubobjectConstructionFunctor**(*S*, *action=<built-in function mul>*, *side='left'*, *other_action=None*, *other_side='left'*)

Bases: *ConstructionFunctor*

Constructor for subobjects invariant or equivariant under given semigroup actions.

Let S be a semigroup that - acts on a parent X as $s \cdot x$ (`action`, `side='left'`) or - acts on X as $x \cdot s$ (`action`, `side='right'`), and (possibly trivially) - acts on X as $s * x$ (`other_action`, `other_side='left'`) or - acts on X as $x * s$ (`other_action`, `other_side='right'`).

The S-equivariant subobject is the subobject

$$X^S := \{x \in X : s \cdot x = s * x, \forall s \in S\}$$

when `side = other_side = 'left'` and mutatis mutandis for the other values of `side` and `other_side`.

When `other_action` is trivial, X^S is called the S-invariant subobject.

EXAMPLES:

Monoterm symmetries of a tensor, here only for matrices: row (index 0), column (index 1); the order of the extra element 2 in a permutation determines whether it is a symmetry or an antisymmetry:

```
sage: # needs sage.groups sage.modules
sage: GSym01 = PermutationGroup([[(0,1),(2,),(3,)]]); GSym01
Permutation Group with generators [(0,1)]
sage: GASym01 = PermutationGroup([[(0,1),(2,3)]]); GASym01
Permutation Group with generators [(0,1)(2,3)]
sage: from sage.categories.action import Action
sage: from sage.structure.element import Matrix
sage: class TensorIndexAction(Action):
....:     def _act_(self, g, x):
....:         if isinstance(x, Matrix):
....:             if g(0) == 1:
....:                 if g(2) == 2:
....:                     return x.transpose()
....:                 else:
....:                     return -x.transpose()
....:             else:
....:                 return x
....:         raise NotImplementedError
sage: M = matrix([[1, 2], [3, 4]]); M
[1 2]
[3 4]
sage: GSym01_action = TensorIndexAction(GSym01, M.parent())
sage: GASym01_action = TensorIndexAction(GASym01, M.parent())
sage: GSym01_action.act(GSym01.0, M)
[1 3]
[2 4]
sage: GASym01_action.act(GASym01.0, M)
[-1 -3]
[-2 -4]
sage: Sym01 = M.parent().invariant_module(GSym01, action=GSym01_action); Sym01
(Permutation Group with generators [(0,1)])-invariant submodule
 of Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
sage: list(Sym01.basis())
[B[0], B[1], B[2]]
sage: list(Sym01.basis().map(Sym01.lift))
```

(continues on next page)

(continued from previous page)

```
[
[1 0]  [0 1]  [0 0]
[0 0], [1 0], [0 1]
]
sage: ASym01 = M.parent().invariant_module(GASym01, action=GASym01_action)
sage: ASym01
(Permutation Group with generators [(0,1)(2,3)])-invariant submodule
 of Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
sage: list(ASym01.basis())
[B[0]]
sage: list(ASym01.basis().map(ASym01.lift))
[
[ 0  1]
[-1  0]
]
sage: from sage.categories.pushout import pushout
sage: pushout(Sym01, QQ)
(Permutation Group with generators [(0,1)])-invariant submodule
 of Full MatrixSpace of 2 by 2 dense matrices over Rational Field
```

```
>>> from sage.all import *
>>> # needs sage.groups sage.modules
>>> GSym01 = PermutationGroup([[(Integer(0),Integer(1)),(Integer(2),),(Integer(3),
↪)]]); GSym01
Permutation Group with generators [(0,1)]
>>> GASym01 = PermutationGroup([[(Integer(0),Integer(1)),(Integer(2),
↪Integer(3))]]); GASym01
Permutation Group with generators [(0,1)(2,3)]
>>> from sage.categories.action import Action
>>> from sage.structure.element import Matrix
>>> class TensorIndexAction(Action):
...     def _act_(self, g, x):
...         if isinstance(x, Matrix):
...             if g(Integer(0)) == Integer(1):
...                 if g(Integer(2)) == Integer(2):
...                     return x.transpose()
...                 else:
...                     return -x.transpose()
...             else:
...                 return x
...         raise NotImplementedError
>>> M = matrix([[Integer(1), Integer(2)], [Integer(3), Integer(4)]]); M
[1 2]
[3 4]
>>> GSym01_action = TensorIndexAction(GSym01, M.parent())
>>> GASym01_action = TensorIndexAction(GASym01, M.parent())
>>> GSym01_action.act(GSym01.gen(0), M)
[1 3]
[2 4]
>>> GASym01_action.act(GASym01.gen(0), M)
[-1 -3]
[-2 -4]
```

(continues on next page)

(continued from previous page)

```
>>> Sym01 = M.parent().invariant_module(GSym01, action=GSym01_action); Sym01
(Permutation Group with generators [(0,1)])-invariant submodule
 of Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
>>> list(Sym01.basis())
[B[0], B[1], B[2]]
>>> list(Sym01.basis().map(Sym01.lift))
[
[1 0]  [0 1]  [0 0]
[0 0], [1 0], [0 1]
]
>>> ASym01 = M.parent().invariant_module(GASym01, action=GASym01_action)
>>> ASym01
(Permutation Group with generators [(0,1)(2,3)])-invariant submodule
 of Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
>>> list(ASym01.basis())
[B[0]]
>>> list(ASym01.basis().map(ASym01.lift))
[
[ 0  1]
[-1  0]
]
>>> from sage.categories.pushout import pushout
>>> pushout(Sym01, QQ)
(Permutation Group with generators [(0,1)])-invariant submodule
 of Full MatrixSpace of 2 by 2 dense matrices over Rational Field
```

class sage.categories.pushout.**FractionField**

Bases: *ConstructionFunctor*

Construction functor for fraction fields.

EXAMPLES:

```
sage: F = QQ.construction()[0]
sage: F
FractionField
sage: F.domain()
Category of integral domains
sage: F.codomain()
Category of fields
sage: F(GF(5)) is GF(5)
True
sage: F(ZZ['t'])
Fraction Field of Univariate Polynomial Ring in t over Integer Ring
sage: P.<x,y> = QQ[]
sage: f = P.hom([x+2*y,3*x-y],P)
sage: F(f)
Ring endomorphism of
 Fraction Field of Multivariate Polynomial Ring in x, y over Rational Field
  Defn: x |--> x + 2*y
        y |--> 3*x - y
sage: F(f)(1/x)
1/(x + 2*y)
```

(continues on next page)

(continued from previous page)

```
sage: F == loads(dumps(F))
True
```

```
>>> from sage.all import *
>>> F = QQ.construction()[Integer(0)]
>>> F
FractionField
>>> F.domain()
Category of integral domains
>>> F.codomain()
Category of fields
>>> F(GF(Integer(5))) is GF(Integer(5))
True
>>> F(ZZ['t'])
Fraction Field of Univariate Polynomial Ring in t over Integer Ring
>>> P = QQ['x, y']; (x, y,) = P._first_ngens(2)
>>> f = P.hom([x+Integer(2)*y,Integer(3)*x-y],P)
>>> F(f)
Ring endomorphism of
 Fraction Field of Multivariate Polynomial Ring in x, y over Rational Field
  Defn: x |--> x + 2*y
        y |--> 3*x - y
>>> F(f)(Integer(1)/x)
1/(x + 2*y)
>>> F == loads(dumps(F))
True
```

rank = 5

class sage.categories.pushout.**IdentityConstructionFunctor**

Bases: *ConstructionFunctor*

A construction functor that is the identity functor.

rank = -100

class sage.categories.pushout.**InfinitePolynomialFunctor**(*gens*, *order*, *implementation*)

Bases: *ConstructionFunctor*

A Construction Functor for Infinite Polynomial Rings (see infinite_polynomial_ring).

AUTHOR:

– Simon King

This construction functor is used to provide uniqueness of infinite polynomial rings as parent structures. As usual, the construction functor allows for constructing pushouts.

Another purpose is to avoid name conflicts of variables of the to-be-constructed infinite polynomial ring with variables of the base ring, and moreover to keep the internal structure of an Infinite Polynomial Ring as simple as possible: If variables $v_1, ..., v_n$ of the given base ring generate an *ordered* sub-monoid of the monomials of the ambient Infinite Polynomial Ring, then they are removed from the base ring and merged with the generators of the ambient ring. However, if the orders don't match, an error is raised, since there was a name conflict without merging.

EXAMPLES:

```
sage: A.<a,b> = InfinitePolynomialRing(ZZ['t'])
sage: A.construction()
[InfPoly{[a,b], "lex", "dense"},
 Univariate Polynomial Ring in t over Integer Ring]
sage: type(_[0])
<class 'sage.categories.pushout.InfinitePolynomialFunctor'>
sage: B.<x,y,a_3,a_1> = PolynomialRing(QQ, order='lex')
sage: B.construction()
(MPoly[x,y,a_3,a_1], Rational Field)
sage: A.construction()[0] * B.construction()[0]
InfPoly{[a,b], "lex", "dense"}(MPoly[x,y](...))
```

```
>>> from sage.all import *
>>> A = InfinitePolynomialRing(ZZ['t'], names=('a', 'b',)); (a, b,) = A._first_
↪ngens(2)
>>> A.construction()
[InfPoly{[a,b], "lex", "dense"},
 Univariate Polynomial Ring in t over Integer Ring]
>>> type(_[Integer(0)])
<class 'sage.categories.pushout.InfinitePolynomialFunctor'>
>>> B = PolynomialRing(QQ, order='lex', names=('x', 'y', 'a_3', 'a_1',)); (x, y,␣
↪a_3, a_1,) = B._first_ngens(4)
>>> B.construction()
(MPoly[x,y,a_3,a_1], Rational Field)
>>> A.construction()[Integer(0)] * B.construction()[Integer(0)]
InfPoly{[a,b], "lex", "dense"}(MPoly[x,y](...))
```

Apparently the variables a_1, a_3 of the polynomial ring are merged with the variables $a_0, a_1, a_2, ...$ of the infinite polynomial ring; indeed, they form an ordered sub-structure. However, if the polynomial ring was given a different ordering, merging would not be allowed, resulting in a name conflict:

```
sage: R = PolynomialRing(QQ, names=['x','y','a_3','a_1'])
sage: A.construction()[0] * R.construction()[0]
Traceback (most recent call last):
...
CoercionException: Incompatible term orders lex, degrevlex
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ, names=['x','y','a_3','a_1'])
>>> A.construction()[Integer(0)] * R.construction()[Integer(0)]
Traceback (most recent call last):
...
CoercionException: Incompatible term orders lex, degrevlex
```

In an infinite polynomial ring with generator a_*, the variable a_3 will always be greater than the variable a_1. Hence, the orders are incompatible in the next example as well:

```
sage: R = PolynomialRing(QQ, names=['x','y','a_1','a_3'], order='lex')
sage: A.construction()[0] * R.construction()[0]
Traceback (most recent call last):
...
CoercionException: Overlapping variables (('a', 'b'),['a_1', 'a_3'])
are incompatible
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ, names=['x','y','a_1','a_3'], order='lex')
>>> A.construction()[Integer(0)] * R.construction()[Integer(0)]
Traceback (most recent call last):
...
CoercionException: Overlapping variables (('a', 'b'),['a_1', 'a_3'])
are incompatible
```

Another requirement is that after merging the order of the remaining variables must be unique. This is not the case in the following example, since it is not clear whether the variables x, y should be greater or smaller than the variables b_*:

```
sage: R = PolynomialRing(QQ, names=['a_3','a_1','x','y'], order='lex')
sage: A.construction()[0] * R.construction()[0]
Traceback (most recent call last):
...
CoercionException: Overlapping variables (('a', 'b'),['a_3', 'a_1'])
are incompatible
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ, names=['a_3','a_1','x','y'], order='lex')
>>> A.construction()[Integer(0)] * R.construction()[Integer(0)]
Traceback (most recent call last):
...
CoercionException: Overlapping variables (('a', 'b'),['a_3', 'a_1'])
are incompatible
```

Since the construction functors are actually used to construct infinite polynomial rings, the following result is no surprise:

```
sage: C.<a,b> = InfinitePolynomialRing(B); C
Infinite polynomial ring in a, b
 over Multivariate Polynomial Ring in x, y over Rational Field
```

```
>>> from sage.all import *
>>> C = InfinitePolynomialRing(B, names=('a', 'b',)); (a, b,) = C._first_ngens(2);
↪ C
Infinite polynomial ring in a, b
 over Multivariate Polynomial Ring in x, y over Rational Field
```

There is also an overlap in the next example:

```
sage: X.<w,x,y> = InfinitePolynomialRing(ZZ)
sage: Y.<x,y,z> = InfinitePolynomialRing(QQ)
```

```
>>> from sage.all import *
>>> X = InfinitePolynomialRing(ZZ, names=('w', 'x', 'y',)); (w, x, y,) = X._first_
↪ngens(3)
>>> Y = InfinitePolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = Y._first_
↪ngens(3)
```

X and Y have an overlapping generators x_*, y_*. Since the default lexicographic order is used in both rings, it gives rise to isomorphic sub-monoids in both X and Y. They are merged in the pushout, which also yields a common

parent for doing arithmetic:

```
sage: P = sage.categories.pushout.pushout(Y,X); P
Infinite polynomial ring in w, x, y, z over Rational Field
sage: w[2]+z[3]
w_2 + z_3
sage: _.parent() is P
True
```

```
>>> from sage.all import *
>>> P = sage.categories.pushout.pushout(Y,X); P
Infinite polynomial ring in w, x, y, z over Rational Field
>>> w[Integer(2)]+z[Integer(3)]
w_2 + z_3
>>> _.parent() is P
True
```

expand()

Decompose the functor F into sub-functors, whose product returns F.

EXAMPLES:

```
sage: A = InfinitePolynomialRing(QQ, ['x','y'], order='degrevlex')
sage: F = A.construction()[0]; F
InfPoly{[x,y], "degrevlex", "dense"}
sage: F.expand()
[InfPoly{[y], "degrevlex", "dense"}, InfPoly{[x], "degrevlex", "dense"}]
sage: A = InfinitePolynomialRing(QQ, ['x','y','z'], order='degrevlex')
sage: F = A.construction()[0]; F
InfPoly{[x,y,z], "degrevlex", "dense"}
sage: F.expand()
[InfPoly{[z], "degrevlex", "dense"},
 InfPoly{[y], "degrevlex", "dense"},
 InfPoly{[x], "degrevlex", "dense"}]
sage: prod(F.expand())==F
True
```

```
>>> from sage.all import *
>>> A = InfinitePolynomialRing(QQ, ['x','y'], order='degrevlex')
>>> F = A.construction()[Integer(0)]; F
InfPoly{[x,y], "degrevlex", "dense"}
>>> F.expand()
[InfPoly{[y], "degrevlex", "dense"}, InfPoly{[x], "degrevlex", "dense"}]
>>> A = InfinitePolynomialRing(QQ, ['x','y','z'], order='degrevlex')
>>> F = A.construction()[Integer(0)]; F
InfPoly{[x,y,z], "degrevlex", "dense"}
>>> F.expand()
[InfPoly{[z], "degrevlex", "dense"},
 InfPoly{[y], "degrevlex", "dense"},
 InfPoly{[x], "degrevlex", "dense"}]
>>> prod(F.expand())==F
True
```

merge(*other*)

Merge two construction functors of infinite polynomial rings, regardless of monomial order and implementation.

The purpose is to have a pushout (and thus, arithmetic) even in cases when the parents are isomorphic as rings, but not as ordered rings.

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ, implementation='sparse')
sage: Y.<x,y> = InfinitePolynomialRing(QQ, order='degrevlex')
sage: X.construction()
[InfPoly{[x,y], "lex", "sparse"}, Rational Field]
sage: Y.construction()
[InfPoly{[x,y], "degrevlex", "dense"}, Rational Field]
sage: Y.construction()[0].merge(Y.construction()[0])
InfPoly{[x,y], "degrevlex", "dense"}
sage: y[3] + X(x[2])
x_2 + y_3
sage: _.parent().construction()
[InfPoly{[x,y], "degrevlex", "dense"}, Rational Field]
```

```
>>> from sage.all import *
>>> X = InfinitePolynomialRing(QQ, implementation='sparse', names=('x', 'y',
↪)); (x, y,) = X._first_ngens(2)
>>> Y = InfinitePolynomialRing(QQ, order='degrevlex', names=('x', 'y',)); (x,␣
↪y,) = Y._first_ngens(2)
>>> X.construction()
[InfPoly{[x,y], "lex", "sparse"}, Rational Field]
>>> Y.construction()
[InfPoly{[x,y], "degrevlex", "dense"}, Rational Field]
>>> Y.construction()[Integer(0)].merge(Y.construction()[Integer(0)])
InfPoly{[x,y], "degrevlex", "dense"}
>>> y[Integer(3)] + X(x[Integer(2)])
x_2 + y_3
>>> _.parent().construction()
[InfPoly{[x,y], "degrevlex", "dense"}, Rational Field]
```

rank = 9.5

class sage.categories.pushout.**LaurentPolynomialFunctor**(*var*, *multi_variate=False*)

Bases: *ConstructionFunctor*

Construction functor for Laurent polynomial rings.

EXAMPLES:

```
sage: L.<t> = LaurentPolynomialRing(ZZ)
sage: F = L.construction()[0]
sage: F
LaurentPolynomialFunctor
sage: F(QQ)
Univariate Laurent Polynomial Ring in t over Rational Field
sage: K.<x> = LaurentPolynomialRing(ZZ)
sage: F(K)
```

(continues on next page)

(continued from previous page)

```
Univariate Laurent Polynomial Ring in t
 over Univariate Laurent Polynomial Ring in x over Integer Ring
sage: P.<x,y> = ZZ[]
sage: f = P.hom([x + 2*y, 3*x - y],P)
sage: F(f)
Ring endomorphism of Univariate Laurent Polynomial Ring in t
 over Multivariate Polynomial Ring in x, y over Integer Ring
  Defn: Induced from base ring by
        Ring endomorphism of Multivariate Polynomial Ring in x, y over Integer␣
↪Ring
          Defn: x |--> x + 2*y
                y |--> 3*x - y
sage: F(f)(x*F(P).gen()^-2 + y*F(P).gen()^3)
(x + 2*y)*t^-2 + (3*x - y)*t^3
```

```
>>> from sage.all import *
>>> L = LaurentPolynomialRing(ZZ, names=('t',)); (t,) = L._first_ngens(1)
>>> F = L.construction()[Integer(0)]
>>> F
LaurentPolynomialFunctor
>>> F(QQ)
Univariate Laurent Polynomial Ring in t over Rational Field
>>> K = LaurentPolynomialRing(ZZ, names=('x',)); (x,) = K._first_ngens(1)
>>> F(K)
Univariate Laurent Polynomial Ring in t
 over Univariate Laurent Polynomial Ring in x over Integer Ring
>>> P = ZZ['x, y']; (x, y,) = P._first_ngens(2)
>>> f = P.hom([x + Integer(2)*y, Integer(3)*x - y],P)
>>> F(f)
Ring endomorphism of Univariate Laurent Polynomial Ring in t
 over Multivariate Polynomial Ring in x, y over Integer Ring
  Defn: Induced from base ring by
        Ring endomorphism of Multivariate Polynomial Ring in x, y over Integer␣
↪Ring
          Defn: x |--> x + 2*y
                y |--> 3*x - y
>>> F(f)(x*F(P).gen()**-Integer(2) + y*F(P).gen()**Integer(3))
(x + 2*y)*t^-2 + (3*x - y)*t^3
```

merge(*other*)

Two Laurent polynomial construction functors merge if the variable names coincide.

The result is multivariate if one of the arguments is multivariate.

EXAMPLES:

```
sage: from sage.categories.pushout import LaurentPolynomialFunctor
sage: F1 = LaurentPolynomialFunctor('t')
sage: F2 = LaurentPolynomialFunctor('t', multi_variate=True)
sage: F1.merge(F2)
LaurentPolynomialFunctor
sage: F1.merge(F2)(LaurentPolynomialRing(GF(2), 'a'))                          #␣
 ↪needs sage.modules
```

(continues on next page)

(continued from previous page)

```
Multivariate Laurent Polynomial Ring in a, t over Finite Field of size 2
sage: F1.merge(F1)(LaurentPolynomialRing(GF(2), 'a'))
Univariate Laurent Polynomial Ring in t over
 Univariate Laurent Polynomial Ring in a over Finite Field of size 2
```

```
>>> from sage.all import *
>>> from sage.categories.pushout import LaurentPolynomialFunctor
>>> F1 = LaurentPolynomialFunctor('t')
>>> F2 = LaurentPolynomialFunctor('t', multi_variate=True)
>>> F1.merge(F2)
LaurentPolynomialFunctor
>>> F1.merge(F2)(LaurentPolynomialRing(GF(Integer(2)), 'a'))
↪     # needs sage.modules
Multivariate Laurent Polynomial Ring in a, t over Finite Field of size 2
>>> F1.merge(F1)(LaurentPolynomialRing(GF(Integer(2)), 'a'))
Univariate Laurent Polynomial Ring in t over
 Univariate Laurent Polynomial Ring in a over Finite Field of size 2
```

rank = 9

class sage.categories.pushout.**MatrixFunctor**(*nrows*, *ncols*, *is_sparse=False*)

Bases: *ConstructionFunctor*

A construction functor for matrices over rings.

EXAMPLES:

```
sage: # needs sage.modules
sage: MS = MatrixSpace(ZZ, 2, 3)
sage: F = MS.construction()[0]; F
MatrixFunctor
sage: MS = MatrixSpace(ZZ, 2)
sage: F = MS.construction()[0]; F
MatrixFunctor
sage: P.<x,y> = QQ[]
sage: R = F(P); R
Full MatrixSpace of 2 by 2 dense matrices
 over Multivariate Polynomial Ring in x, y over Rational Field
sage: f = P.hom([x+y, x-y], P); F(f)
Ring endomorphism
 of Full MatrixSpace of 2 by 2 dense matrices
  over Multivariate Polynomial Ring in x, y over Rational Field
  Defn: Induced from base ring by
        Ring endomorphism
         of Multivariate Polynomial Ring in x, y over Rational Field
          Defn: x |--> x + y
                y |--> x - y
sage: M = R([x, y, x*y, x + y])
sage: F(f)(M)
[    x + y     x - y]
[x^2 - y^2       2*x]
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> MS = MatrixSpace(ZZ, Integer(2), Integer(3))
>>> F = MS.construction()[Integer(0)]; F
MatrixFunctor
>>> MS = MatrixSpace(ZZ, Integer(2))
>>> F = MS.construction()[Integer(0)]; F
MatrixFunctor
>>> P = QQ['x, y']; (x, y,) = P._first_ngens(2)
>>> R = F(P); R
Full MatrixSpace of 2 by 2 dense matrices
 over Multivariate Polynomial Ring in x, y over Rational Field
>>> f = P.hom([x+y, x-y], P); F(f)
Ring endomorphism
 of Full MatrixSpace of 2 by 2 dense matrices
  over Multivariate Polynomial Ring in x, y over Rational Field
  Defn: Induced from base ring by
        Ring endomorphism
         of Multivariate Polynomial Ring in x, y over Rational Field
          Defn: x |--> x + y
                y |--> x - y
>>> M = R([x, y, x*y, x + y])
>>> F(f)(M)
[    x + y     x - y]
[x^2 - y^2       2*x]
```

merge(*other*)

Merging is only happening if both functors are matrix functors of the same dimension.

The result is sparse if and only if both given functors are sparse.

EXAMPLES:

```
sage: # needs sage.modules
sage: F1 = MatrixSpace(ZZ, 2, 2).construction()[0]
sage: F2 = MatrixSpace(ZZ, 2, 3).construction()[0]
sage: F3 = MatrixSpace(ZZ, 2, 2, sparse=True).construction()[0]
sage: F1.merge(F2)
sage: F1.merge(F3)
MatrixFunctor
sage: F13 = F1.merge(F3)
sage: F13.is_sparse
False
sage: F1.is_sparse
False
sage: F3.is_sparse
True
sage: F3.merge(F3).is_sparse
True
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> F1 = MatrixSpace(ZZ, Integer(2), Integer(2)).construction()[Integer(0)]
```

(continues on next page)

(continued from previous page)

```
>>> F2 = MatrixSpace(ZZ, Integer(2), Integer(3)).construction()[Integer(0)]
>>> F3 = MatrixSpace(ZZ, Integer(2), Integer(2), sparse=True).
↪construction()[Integer(0)]
>>> F1.merge(F2)
>>> F1.merge(F3)
MatrixFunctor
>>> F13 = F1.merge(F3)
>>> F13.is_sparse
False
>>> F1.is_sparse
False
>>> F3.is_sparse
True
>>> F3.merge(F3).is_sparse
True
```

rank = 10

class sage.categories.pushout.**MultiPolynomialFunctor**(*vars*, *term_order*)

Bases: *ConstructionFunctor*

A constructor for multivariate polynomial rings.

EXAMPLES:

```
sage: P.<x,y> = ZZ[]
sage: F = P.construction()[0]; F
MPoly[x,y]
sage: A.<a,b> = GF(5)[]
sage: F(A)
Multivariate Polynomial Ring in x, y
 over Multivariate Polynomial Ring in a, b over Finite Field of size 5
sage: f = A.hom([a+b, a-b], A)
sage: F(f)
Ring endomorphism of Multivariate Polynomial Ring in x, y
 over Multivariate Polynomial Ring in a, b over Finite Field of size 5
  Defn: Induced from base ring by
        Ring endomorphism of Multivariate Polynomial Ring in a, b over Finite␣
↪Field of size 5
          Defn: a |--> a + b
                b |--> a - b
sage: F(f)(F(A)(x)*a)
(a + b)*x
```

```
>>> from sage.all import *
>>> P = ZZ['x, y']; (x, y,) = P._first_ngens(2)
>>> F = P.construction()[Integer(0)]; F
MPoly[x,y]
>>> A = GF(Integer(5))['a, b']; (a, b,) = A._first_ngens(2)
>>> F(A)
Multivariate Polynomial Ring in x, y
 over Multivariate Polynomial Ring in a, b over Finite Field of size 5
>>> f = A.hom([a+b, a-b], A)
```

(continues on next page)

(continued from previous page)

```
>>> F(f)
Ring endomorphism of Multivariate Polynomial Ring in x, y
 over Multivariate Polynomial Ring in a, b over Finite Field of size 5
  Defn: Induced from base ring by
        Ring endomorphism of Multivariate Polynomial Ring in a, b over Finite␣
↪Field of size 5
          Defn: a |--> a + b
                b |--> a - b
>>> F(f)(F(A)(x)*a)
(a + b)*x
```

expand()

Decompose `self` into a list of construction functors.

EXAMPLES:

```
sage: F = QQ['x,y,z,t'].construction()[0]; F
MPoly[x,y,z,t]
sage: F.expand()
[MPoly[t], MPoly[z], MPoly[y], MPoly[x]]
```

```
>>> from sage.all import *
>>> F = QQ['x,y,z,t'].construction()[Integer(0)]; F
MPoly[x,y,z,t]
>>> F.expand()
[MPoly[t], MPoly[z], MPoly[y], MPoly[x]]
```

Now an actual use case:

```
sage: R.<x,y,z> = ZZ[]
sage: S.<z,t> = QQ[]
sage: x+t
x + t
sage: parent(x+t)
Multivariate Polynomial Ring in x, y, z, t over Rational Field
sage: T.<y,s> = QQ[]
sage: x + s
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +:
'Multivariate Polynomial Ring in x, y, z over Integer Ring' and
'Multivariate Polynomial Ring in y, s over Rational Field'
sage: R = PolynomialRing(ZZ, 'x', 50)
sage: S = PolynomialRing(GF(5), 'x', 20)
sage: R.gen(0) + S.gen(0)
2*x0
```

```
>>> from sage.all import *
>>> R = ZZ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> S = QQ['z, t']; (z, t,) = S._first_ngens(2)
>>> x+t
x + t
```

(continues on next page)

(continued from previous page)

```
>>> parent(x+t)
Multivariate Polynomial Ring in x, y, z, t over Rational Field
>>> T = QQ['y, s']; (y, s,) = T._first_ngens(2)
>>> x + s
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +:
'Multivariate Polynomial Ring in x, y, z over Integer Ring' and
'Multivariate Polynomial Ring in y, s over Rational Field'
>>> R = PolynomialRing(ZZ, 'x', Integer(50))
>>> S = PolynomialRing(GF(Integer(5)), 'x', Integer(20))
>>> R.gen(Integer(0)) + S.gen(Integer(0))
2*x0
```

merge(*other*)

Merge `self` with another construction functor, or return `None`.

EXAMPLES:

```
sage: F = sage.categories.pushout.MultiPolynomialFunctor(['x','y'], None)
sage: G = sage.categories.pushout.MultiPolynomialFunctor(['t'], None)
sage: F.merge(G) is None
True
sage: F.merge(F)
MPoly[x,y]
```

```
>>> from sage.all import *
>>> F = sage.categories.pushout.MultiPolynomialFunctor(['x','y'], None)
>>> G = sage.categories.pushout.MultiPolynomialFunctor(['t'], None)
>>> F.merge(G) is None
True
>>> F.merge(F)
MPoly[x,y]
```

rank = 9

class sage.categories.pushout.**MultivariateConstructionFunctor**

Bases: *ConstructionFunctor*

An abstract base class for functors that take multiple inputs (e.g. Cartesian products).

common_base(*other_functor*, *self_bases*, *other_bases*)

This function is called by *pushout()* when no common parent is found in the construction tower.

INPUT:

- `other_functor` – a construction functor
- `self_bases` – the arguments passed to this functor
- `other_bases` – the arguments passed to the functor `other_functor`

OUTPUT: a parent

If no common base is found a *sage.structure.coerce_exceptions.CoercionException* is raised.

Note

Overload this function in derived class, see e.g. *MultivariateConstructionFunctor*.

class sage.categories.pushout.**PermutationGroupFunctor**(*gens*, *domain*)

Bases: *ConstructionFunctor*

EXAMPLES:

```
sage: from sage.categories.pushout import PermutationGroupFunctor
sage: PF = PermutationGroupFunctor([PermutationGroupElement([(1,2)])],      #␣
↪needs sage.groups
....:                               [1,2]); PF
PermutationGroupFunctor[(1,2)]
```

```
>>> from sage.all import *
>>> from sage.categories.pushout import PermutationGroupFunctor
>>> PF = PermutationGroupFunctor([PermutationGroupElement([(Integer(1),
↪Integer(2))])],      # needs sage.groups
...                               [Integer(1),Integer(2)]); PF
PermutationGroupFunctor[(1,2)]
```

gens()

EXAMPLES:

```
sage: P1 = PermutationGroup([[(1,2)]])                                      #␣
↪needs sage.groups
sage: PF, P = P1.construction()                                             #␣
↪needs sage.groups
sage: PF.gens()                                                             #␣
↪needs sage.groups
((1,2),)
```

```
>>> from sage.all import *
>>> P1 = PermutationGroup([[(Integer(1),Integer(2))]])                       ␣
↪              # needs sage.groups
>>> PF, P = P1.construction()                                              #␣
↪needs sage.groups
>>> PF.gens()                                                              #␣
↪needs sage.groups
((1,2),)
```

merge(*other*)

Merge `self` with another construction functor, or return `None`.

EXAMPLES:

```
sage: # needs sage.groups
sage: P1 = PermutationGroup([[(1,2)]])
sage: PF1, P = P1.construction()
sage: P2 = PermutationGroup([[(1,3)]])
sage: PF2, P = P2.construction()
```

(continues on next page)

(continued from previous page)

```
sage: PF1.merge(PF2)
PermutationGroupFunctor[(1,2), (1,3)]
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> P1 = PermutationGroup([[(Integer(1),Integer(2))]])
>>> PF1, P = P1.construction()
>>> P2 = PermutationGroup([[(Integer(1),Integer(3))]])
>>> PF2, P = P2.construction()
>>> PF1.merge(PF2)
PermutationGroupFunctor[(1,2), (1,3)]
```

rank = 10

class sage.categories.pushout.**PolynomialFunctor**(*var*, *multi_variate=False*, *sparse=False*, *implementation=None*)

Bases: *ConstructionFunctor*

Construction functor for univariate polynomial rings.

EXAMPLES:

```
sage: P = ZZ['t'].construction()[0]
sage: P(GF(3))
Univariate Polynomial Ring in t over Finite Field of size 3
sage: P == loads(dumps(P))
True
sage: R.<x,y> = GF(5)[]
sage: f = R.hom([x + 2*y, 3*x - y], R)
sage: P(f)((x+y) * P(R).0)
(-x + y)*t
```

```
>>> from sage.all import *
>>> P = ZZ['t'].construction()[Integer(0)]
>>> P(GF(Integer(3)))
Univariate Polynomial Ring in t over Finite Field of size 3
>>> P == loads(dumps(P))
True
>>> R = GF(Integer(5))['x, y']; (x, y,) = R._first_ngens(2)
>>> f = R.hom([x + Integer(2)*y, Integer(3)*x - y], R)
>>> P(f)((x+y) * P(R).gen(0))
(-x + y)*t
```

By Issue #9944, the construction functor distinguishes sparse and dense polynomial rings. Before, the following example failed:

```
sage: R.<x> = PolynomialRing(GF(5), sparse=True)
sage: F, B = R.construction()
sage: F(B) is R
True
sage: S.<x> = PolynomialRing(ZZ)
sage: R.has_coerce_map_from(S)
False
```

(continues on next page)

(continued from previous page)

```
sage: S.has_coerce_map_from(R)
False
sage: S.0 + R.0
2*x
sage: (S.0 + R.0).parent()
Univariate Polynomial Ring in x over Finite Field of size 5
sage: (S.0 + R.0).parent().is_sparse()
False
```

```
>>> from sage.all import *
>>> R = PolynomialRing(GF(Integer(5)), sparse=True, names=('x',)); (x,) = R._
↪first_ngens(1)
>>> F, B = R.construction()
>>> F(B) is R
True
>>> S = PolynomialRing(ZZ, names=('x',)); (x,) = S._first_ngens(1)
>>> R.has_coerce_map_from(S)
False
>>> S.has_coerce_map_from(R)
False
>>> S.gen(0) + R.gen(0)
2*x
>>> (S.gen(0) + R.gen(0)).parent()
Univariate Polynomial Ring in x over Finite Field of size 5
>>> (S.gen(0) + R.gen(0)).parent().is_sparse()
False
```

merge(*other*)

Merge `self` with another construction functor, or return `None`.

Note

Internally, the merging is delegated to the merging of multipolynomial construction functors. But in effect, this does the same as the default implementation, that returns `None` unless the to-be-merged functors coincide.

EXAMPLES:

```
sage: P = ZZ['x'].construction()[0]
sage: Q = ZZ['y','x'].construction()[0]
sage: P.merge(Q)
sage: P.merge(P) is P
True
```

```
>>> from sage.all import *
>>> P = ZZ['x'].construction()[Integer(0)]
>>> Q = ZZ['y','x'].construction()[Integer(0)]
>>> P.merge(Q)
>>> P.merge(P) is P
True
```

rank = 9

class sage.categories.pushout.**QuotientFunctor**(*I*, *names=None*, *as_field=False*, *domain=None*, *codomain=None*, ***kwds*)

Bases: *ConstructionFunctor*

Construction functor for quotient rings.

Note

The functor keeps track of variable names. Optionally, it may keep track of additional properties of the quotient, such as its category or its implementation.

EXAMPLES:

```
sage: P.<x,y> = ZZ[]
sage: Q = P.quo([x^2 + y^2] * P)
sage: F = Q.construction()[0]
sage: F(QQ['x','y'])
Quotient of Multivariate Polynomial Ring in x, y over Rational Field
 by the ideal (x^2 + y^2)
sage: F(QQ['x','y']) == QQ['x','y'].quo([x^2 + y^2] * QQ['x','y'])
True
sage: F(QQ['x','y','z'])
Traceback (most recent call last):
...
CoercionException: Cannot apply this quotient functor to
 Multivariate Polynomial Ring in x, y, z over Rational Field
sage: F(QQ['y','z'])                                                    #␣
↪needs sage.rings.finite_rings
Traceback (most recent call last):
...
TypeError: Could not find a mapping of the passed element to this ring.
```

```
>>> from sage.all import *
>>> P = ZZ['x, y']; (x, y,) = P._first_ngens(2)
>>> Q = P.quo([x**Integer(2) + y**Integer(2)] * P)
>>> F = Q.construction()[Integer(0)]
>>> F(QQ['x','y'])
Quotient of Multivariate Polynomial Ring in x, y over Rational Field
 by the ideal (x^2 + y^2)
>>> F(QQ['x','y']) == QQ['x','y'].quo([x**Integer(2) + y**Integer(2)] * QQ['x','y
↪'])
True
>>> F(QQ['x','y','z'])
Traceback (most recent call last):
...
CoercionException: Cannot apply this quotient functor to
 Multivariate Polynomial Ring in x, y, z over Rational Field
>>> F(QQ['y','z'])                                                      #␣
↪needs sage.rings.finite_rings
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
TypeError: Could not find a mapping of the passed element to this ring.
```

merge(*other*)

Two quotient functors with coinciding names are merged by taking the gcd of their moduli, the meet of their domains, and the join of their codomains.

In particular, if one of the functors being merged knows that the quotient is going to be a field, then the merged functor will return fields as well.

EXAMPLES:

```
sage: # needs sage.libs.pari
sage: P.<x> = QQ[]
sage: Q1 = P.quo([(x^2+1)^2*(x^2-3)])
sage: Q2 = P.quo([(x^2+1)^2*(x^5+3)])
sage: from sage.categories.pushout import pushout
sage: pushout(Q1,Q2)     # indirect doctest
Univariate Quotient Polynomial Ring in xbar over Rational Field
 with modulus x^4 + 2*x^2 + 1
```

```
>>> from sage.all import *
>>> # needs sage.libs.pari
>>> P = QQ['x']; (x,) = P._first_ngens(1)
>>> Q1 = P.quo([(x**Integer(2)+Integer(1))**Integer(2)*(x**Integer(2)-
↪Integer(3))])
>>> Q2 = P.
↪quo([(x**Integer(2)+Integer(1))**Integer(2)*(x**Integer(5)+Integer(3))])
>>> from sage.categories.pushout import pushout
>>> pushout(Q1,Q2)     # indirect doctest
Univariate Quotient Polynomial Ring in xbar over Rational Field
 with modulus x^4 + 2*x^2 + 1
```

The following was fixed in Issue #8800:

```
sage: pushout(GF(5), Integers(5))                                              #␣
↪needs sage.libs.pari
Finite Field of size 5
```

```
>>> from sage.all import *
>>> pushout(GF(Integer(5)), Integers(Integer(5)))                               ␣
↪              # needs sage.libs.pari
Finite Field of size 5
```

rank = 4.5

class sage.categories.pushout.**SubspaceFunctor**(*basis*)

Bases: *ConstructionFunctor*

Constructing a subspace of an ambient free module, given by a basis.

Note

This construction functor keeps track of the basis. It can only be applied to free modules into which this basis coerces.

EXAMPLES:

```
sage: # needs sage.modules
sage: M = ZZ^3
sage: S = M.submodule([(1,2,3), (4,5,6)]); S
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[1 2 3]
[0 3 6]
sage: F = S.construction()[0]
sage: F(GF(2)^3)
Vector space of degree 3 and dimension 2 over Finite Field of size 2
User basis matrix:
[1 0 1]
[0 1 0]
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> M = ZZ**Integer(3)
>>> S = M.submodule([(Integer(1),Integer(2),Integer(3)), (Integer(4),Integer(5),
↪Integer(6))]); S
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[1 2 3]
[0 3 6]
>>> F = S.construction()[Integer(0)]
>>> F(GF(Integer(2))**Integer(3))
Vector space of degree 3 and dimension 2 over Finite Field of size 2
User basis matrix:
[1 0 1]
[0 1 0]
```

coercion_reversed = True

merge(*other*)

Two Subspace Functors are merged into a construction functor of the sum of two subspaces.

EXAMPLES:

```
sage: # needs sage.modules
sage: M = GF(5)^3
sage: S1 = M.submodule([(1,2,3),(4,5,6)])
sage: S2 = M.submodule([(2,2,3)])
sage: F1 = S1.construction()[0]
sage: F2 = S2.construction()[0]
sage: F1.merge(F2)
SubspaceFunctor
sage: F1.merge(F2)(GF(5)^3) == S1 + S2
True
sage: F1.merge(F2)(GF(5)['t']^3)
```

(continues on next page)

(continued from previous page)

```
Free module of degree 3 and rank 3
 over Univariate Polynomial Ring in t over Finite Field of size 5
User basis matrix:
[1 0 0]
[0 1 0]
[0 0 1]
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> M = GF(Integer(5))**Integer(3)
>>> S1 = M.submodule([(Integer(1),Integer(2),Integer(3)),(Integer(4),
↪Integer(5),Integer(6))])
>>> S2 = M.submodule([(Integer(2),Integer(2),Integer(3))])
>>> F1 = S1.construction()[Integer(0)]
>>> F2 = S2.construction()[Integer(0)]
>>> F1.merge(F2)
SubspaceFunctor
>>> F1.merge(F2)(GF(Integer(5))**Integer(3)) == S1 + S2
True
>>> F1.merge(F2)(GF(Integer(5))['t']**Integer(3))
Free module of degree 3 and rank 3
 over Univariate Polynomial Ring in t over Finite Field of size 5
User basis matrix:
[1 0 0]
[0 1 0]
[0 0 1]
```

rank = 11

class sage.categories.pushout.**VectorFunctor**(*n=None*, *is_sparse=False*, *inner_product_matrix=None*, ***, *with_basis='standard'*, *basis_keys=None*, *name_mapping=None*, *latex_name_mapping=None*)

Bases: *ConstructionFunctor*

A construction functor for free modules over commutative rings.

EXAMPLES:

```
sage: # needs sage.modules
sage: F = (ZZ^3).construction()[0]
sage: F
VectorFunctor
sage: F(GF(2)['t'])                                                    #␣
↪needs sage.libs.ntl
Ambient free module of rank 3
 over the principal ideal domain Univariate Polynomial Ring in t
  over Finite Field of size 2 (using GF2X)
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> F = (ZZ**Integer(3)).construction()[Integer(0)]
>>> F
VectorFunctor
```

(continues on next page)

(continued from previous page)

```
>>> F(GF(Integer(2))['t'])                                                    ␣
↪     # needs sage.libs.ntl
Ambient free module of rank 3
 over the principal ideal domain Univariate Polynomial Ring in t
  over Finite Field of size 2 (using GF2X)
```

merge(*other*)

Two constructors of free modules merge, if the module ranks and the inner products coincide. If both have explicitly given inner product matrices, they must coincide as well.

EXAMPLES:

Two modules without explicitly given inner product allow coercion:

```
sage: M1 = QQ^3                                                             #␣
↪needs sage.modules
sage: P.<t> = ZZ[]
sage: M2 = FreeModule(P, 3)                                                 #␣
↪needs sage.modules
sage: M1([1,1/2,1/3]) + M2([t,t^2+t,3])      # indirect doctest             #␣
↪needs sage.modules
(t + 1, t^2 + t + 1/2, 10/3)
```

```
>>> from sage.all import *
>>> M1 = QQ**Integer(3)                                                     ␣
↪      # needs sage.modules
>>> P = ZZ['t']; (t,) = P._first_ngens(1)
>>> M2 = FreeModule(P, Integer(3))                                          ␣
↪     # needs sage.modules
>>> M1([Integer(1),Integer(1)/Integer(2),Integer(1)/Integer(3)]) + M2([t,
↪t**Integer(2)+t,Integer(3)])      # indirect doctest                # needs␣
↪sage.modules
(t + 1, t^2 + t + 1/2, 10/3)
```

If only one summand has an explicit inner product, the result will be provided with it:

```
sage: M3 = FreeModule(P, 3, inner_product_matrix=Matrix(3, 3, range(9)))    #␣
↪needs sage.modules
sage: M1([1,1/2,1/3]) + M3([t,t^2+t,3])                                     #␣
↪needs sage.modules
(t + 1, t^2 + t + 1/2, 10/3)
sage: (M1([1,1/2,1/3]) + M3([t,t^2+t,3])).parent().inner_product_matrix()   #␣
↪needs sage.modules
[0 1 2]
[3 4 5]
[6 7 8]
```

```
>>> from sage.all import *
>>> M3 = FreeModule(P, Integer(3), inner_product_matrix=Matrix(Integer(3),␣
↪Integer(3), range(Integer(9))))     # needs sage.modules
>>> M1([Integer(1),Integer(1)/Integer(2),Integer(1)/Integer(3)]) + M3([t,
↪t**Integer(2)+t,Integer(3)])                                       # needs␣
↪sage.modules
```

(continues on next page)

(continued from previous page)

```
(t + 1, t^2 + t + 1/2, 10/3)
>>> (M1([Integer(1),Integer(1)/Integer(2),Integer(1)/Integer(3)]) + M3([t,
↪t**Integer(2)+t,Integer(3)])).parent().inner_product_matrix()    # needs␣
↪sage.modules
[0 1 2]
[3 4 5]
[6 7 8]
```

If both summands have an explicit inner product (even if it is the standard inner product), then the products must coincide. The only difference between `M1` and `M4` in the following example is the fact that the default inner product was *explicitly* requested for `M4`. It is therefore not possible to coerce with a different inner product:

```
sage: # needs sage.modules
sage: M4 = FreeModule(QQ, 3, inner_product_matrix=Matrix(3, 3, 1))
sage: M4 == M1
True
sage: M4.inner_product_matrix() == M1.inner_product_matrix()
True
sage: M4([1,1/2,1/3]) + M3([t,t^2+t,3])      # indirect doctest
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +:
'Ambient quadratic space of dimension 3 over Rational Field
Inner product matrix:
[1 0 0]
[0 1 0]
[0 0 1]' and
'Ambient free quadratic module of rank 3 over the integral domain
Univariate Polynomial Ring in t over Integer Ring
Inner product matrix:
[0 1 2]
[3 4 5]
[6 7 8]'
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> M4 = FreeModule(QQ, Integer(3), inner_product_matrix=Matrix(Integer(3),␣
↪Integer(3), Integer(1)))
>>> M4 == M1
True
>>> M4.inner_product_matrix() == M1.inner_product_matrix()
True
>>> M4([Integer(1),Integer(1)/Integer(2),Integer(1)/Integer(3)]) + M3([t,
↪t**Integer(2)+t,Integer(3)])      # indirect doctest
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +:
'Ambient quadratic space of dimension 3 over Rational Field
Inner product matrix:
[1 0 0]
[0 1 0]
```

(continues on next page)

(continued from previous page)

```
[0 0 1]' and
'Ambient free quadratic module of rank 3 over the integral domain
Univariate Polynomial Ring in t over Integer Ring
Inner product matrix:
[0 1 2]
[3 4 5]
[6 7 8]'
```

Names are removed when they conflict:

```
sage: # needs sage.modules
sage: from sage.categories.pushout import VectorFunctor, pushout
sage: M_ZZx = FreeModule(ZZ['x'], 4, with_basis=None, name='M_ZZx')
sage: N_ZZx = FreeModule(ZZ['x'], 4, with_basis=None, name='N_ZZx')
sage: pushout(M_ZZx, QQ)
Rank-4 free module M_ZZx_base_ext
 over the Univariate Polynomial Ring in x over Rational Field
sage: pushout(M_ZZx, N_ZZx)
Rank-4 free module
 over the Univariate Polynomial Ring in x over Integer Ring
sage: pushout(pushout(M_ZZx, N_ZZx), QQ)
Rank-4 free module
 over the Univariate Polynomial Ring in x over Rational Field
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> from sage.categories.pushout import VectorFunctor, pushout
>>> M_ZZx = FreeModule(ZZ['x'], Integer(4), with_basis=None, name='M_ZZx')
>>> N_ZZx = FreeModule(ZZ['x'], Integer(4), with_basis=None, name='N_ZZx')
>>> pushout(M_ZZx, QQ)
Rank-4 free module M_ZZx_base_ext
 over the Univariate Polynomial Ring in x over Rational Field
>>> pushout(M_ZZx, N_ZZx)
Rank-4 free module
 over the Univariate Polynomial Ring in x over Integer Ring
>>> pushout(pushout(M_ZZx, N_ZZx), QQ)
Rank-4 free module
 over the Univariate Polynomial Ring in x over Rational Field
```

rank = 10

sage.categories.pushout.**construction_tower**(*R*)

An auxiliary function that is used in *pushout()* and *pushout_lattice()*.

INPUT:

- R – an object

OUTPUT:

A constructive description of the object from scratch, by a list of pairs of a construction functor and an object to which the construction functor is to be applied. The first pair is formed by None and the given object.

EXAMPLES:

```
sage: from sage.categories.pushout import construction_tower
sage: construction_tower(MatrixSpace(FractionField(QQ['t']), 2))                #␣
↪needs sage.modules
[(None, Full MatrixSpace of 2 by 2 dense matrices over Fraction Field
         of Univariate Polynomial Ring in t over Rational Field),
 (MatrixFunctor, Fraction Field
                 of Univariate Polynomial Ring in t over Rational Field),
 (FractionField, Univariate Polynomial Ring in t over Rational Field),
 (Poly[t], Rational Field), (FractionField, Integer Ring)]
```

```
>>> from sage.all import *
>>> from sage.categories.pushout import construction_tower
>>> construction_tower(MatrixSpace(FractionField(QQ['t']), Integer(2)))          ␣
↪     # needs sage.modules
[(None, Full MatrixSpace of 2 by 2 dense matrices over Fraction Field
         of Univariate Polynomial Ring in t over Rational Field),
 (MatrixFunctor, Fraction Field
                 of Univariate Polynomial Ring in t over Rational Field),
 (FractionField, Univariate Polynomial Ring in t over Rational Field),
 (Poly[t], Rational Field), (FractionField, Integer Ring)]
```

sage.categories.pushout.**expand_tower**(*tower*)

An auxiliary function that is used in *pushout()*.

INPUT:

- `tower` – a construction tower as returned by *construction_tower()*

OUTPUT: a new construction tower with all the construction functors expanded

EXAMPLES:

```
sage: from sage.categories.pushout import construction_tower, expand_tower
sage: construction_tower(QQ['x,y,z'])
[(None, Multivariate Polynomial Ring in x, y, z over Rational Field),
 (MPoly[x,y,z], Rational Field),
 (FractionField, Integer Ring)]
sage: expand_tower(construction_tower(QQ['x,y,z']))
[(None, Multivariate Polynomial Ring in x, y, z over Rational Field),
 (MPoly[z], Univariate Polynomial Ring in y
             over Univariate Polynomial Ring in x over Rational Field),
 (MPoly[y], Univariate Polynomial Ring in x over Rational Field),
 (MPoly[x], Rational Field),
 (FractionField, Integer Ring)]
```

```
>>> from sage.all import *
>>> from sage.categories.pushout import construction_tower, expand_tower
>>> construction_tower(QQ['x,y,z'])
[(None, Multivariate Polynomial Ring in x, y, z over Rational Field),
 (MPoly[x,y,z], Rational Field),
 (FractionField, Integer Ring)]
>>> expand_tower(construction_tower(QQ['x,y,z']))
[(None, Multivariate Polynomial Ring in x, y, z over Rational Field),
 (MPoly[z], Univariate Polynomial Ring in y
```

(continues on next page)

(continued from previous page)

```
            over Univariate Polynomial Ring in x over Rational Field),
 (MPoly[y], Univariate Polynomial Ring in x over Rational Field),
 (MPoly[x], Rational Field),
 (FractionField, Integer Ring)]
```

sage.categories.pushout.**pushout**(*R*, *S*)

Given a pair of objects R and S, try to construct a reasonable object Y and return maps such that canonically $R \leftarrow Y \rightarrow S$.

ALGORITHM:

This incorporates the idea of functors discussed at Sage Days 4. Every object R can be viewed as an initial object and a series of functors (e.g. polynomial, quotient, extension, completion, vector/matrix, etc.). Call the series of increasingly simple objects (with the associated functors) the "tower" of R. The construction method is used to create the tower.

Given two objects R and S, try to find a common initial object Z. If the towers of R and S meet, let Z be their join. Otherwise, see if the top of one coerces naturally into the other.

Now we have an initial object and two ordered lists of functors to apply. We wish to merge these in an unambiguous order, popping elements off the top of one or the other tower as we apply them to Z.

- If the functors are of distinct types, there is an absolute ordering given by the rank attribute. Use this.
- Otherwise:
  - If the tops are equal, we (try to) merge them.
  - If exactly one occurs lower in the other tower, we may unambiguously apply the other (hoping for a later merge).
  - If the tops commute, we can apply either first.
  - Otherwise fail due to ambiguity.

The algorithm assumes by default that when a construction F is applied to an object X, the object $F(X)$ admits a coercion map from X. However, the algorithm can also handle the case where $F(X)$ has a coercion map *to* X instead. In this case, the attribute `coercion_reversed` of the class implementing F should be set to `True`.

EXAMPLES:

Here our "towers" are $R = Complete_7(Frac(\mathbf{Z}))$ and $Frac(Poly_x(\mathbf{Z}))$, which give us $Frac(Poly_x(Complete_7(Frac(\mathbf{Z}))))$:

```
sage: from sage.categories.pushout import pushout
sage: pushout(Qp(7), Frac(ZZ['x']))                                             #␣
↪needs sage.rings.padics
Fraction Field of Univariate Polynomial Ring in x
 over 7-adic Field with capped relative precision 20
```

```
>>> from sage.all import *
>>> from sage.categories.pushout import pushout
>>> pushout(Qp(Integer(7)), Frac(ZZ['x']))                                      ␣
↪     # needs sage.rings.padics
Fraction Field of Univariate Polynomial Ring in x
 over 7-adic Field with capped relative precision 20
```

Note we get the same thing with

```
sage: pushout(Zp(7), Frac(QQ['x']))                                             #␣
↪needs sage.rings.padics
Fraction Field of Univariate Polynomial Ring in x
 over 7-adic Field with capped relative precision 20
sage: pushout(Zp(7)['x'], Frac(QQ['x']))                                        #␣
↪needs sage.rings.padics
Fraction Field of Univariate Polynomial Ring in x
 over 7-adic Field with capped relative precision 20
```

```
>>> from sage.all import *
>>> pushout(Zp(Integer(7)), Frac(QQ['x']))                                      ␣
↪     # needs sage.rings.padics
Fraction Field of Univariate Polynomial Ring in x
 over 7-adic Field with capped relative precision 20
>>> pushout(Zp(Integer(7))['x'], Frac(QQ['x']))                                 ␣
↪     # needs sage.rings.padics
Fraction Field of Univariate Polynomial Ring in x
 over 7-adic Field with capped relative precision 20
```

Note that polynomial variable ordering must be unambiguously determined.

```
sage: pushout(ZZ['x,y,z'], QQ['w,z,t'])
Traceback (most recent call last):
...
CoercionException: ('Ambiguous Base Extension',
Multivariate Polynomial Ring in x, y, z over Integer Ring,
Multivariate Polynomial Ring in w, z, t over Rational Field)
sage: pushout(ZZ['x,y,z'], QQ['w,x,z,t'])
Multivariate Polynomial Ring in w, x, y, z, t over Rational Field
```

```
>>> from sage.all import *
>>> pushout(ZZ['x,y,z'], QQ['w,z,t'])
Traceback (most recent call last):
...
CoercionException: ('Ambiguous Base Extension',
Multivariate Polynomial Ring in x, y, z over Integer Ring,
Multivariate Polynomial Ring in w, z, t over Rational Field)
>>> pushout(ZZ['x,y,z'], QQ['w,x,z,t'])
Multivariate Polynomial Ring in w, x, y, z, t over Rational Field
```

Some other examples:

```
sage: pushout(Zp(7)['y'], Frac(QQ['t'])['x,y,z'])                               #␣
↪needs sage.rings.padics
Multivariate Polynomial Ring in x, y, z
 over Fraction Field of Univariate Polynomial Ring in t
  over 7-adic Field with capped relative precision 20
sage: pushout(ZZ['x,y,z'], Frac(ZZ['x'])['y'])
Multivariate Polynomial Ring in y, z
 over Fraction Field of Univariate Polynomial Ring in x over Integer Ring
sage: pushout(MatrixSpace(RDF, 2, 2), Frac(ZZ['x']))                           #␣
↪needs sage.modules
Full MatrixSpace of 2 by 2 dense matrices
```

(continues on next page)

(continued from previous page)

```
 over Fraction Field of Univariate Polynomial Ring in x over Real Double Field
sage: pushout(ZZ, MatrixSpace(ZZ[['x']], 3, 3))                                #␣
↪needs sage.modules
Full MatrixSpace of 3 by 3 dense matrices
 over Power Series Ring in x over Integer Ring
sage: pushout(QQ['x,y'], ZZ[['x']])
Univariate Polynomial Ring in y
 over Power Series Ring in x over Rational Field
sage: pushout(Frac(ZZ['x']), QQ[['x']])
Laurent Series Ring in x over Rational Field
```

```
>>> from sage.all import *
>>> pushout(Zp(Integer(7))['y'], Frac(QQ['t'])['x,y,z'])                       ␣
↪     # needs sage.rings.padics
Multivariate Polynomial Ring in x, y, z
 over Fraction Field of Univariate Polynomial Ring in t
  over 7-adic Field with capped relative precision 20
>>> pushout(ZZ['x,y,z'], Frac(ZZ['x'])['y'])
Multivariate Polynomial Ring in y, z
 over Fraction Field of Univariate Polynomial Ring in x over Integer Ring
>>> pushout(MatrixSpace(RDF, Integer(2), Integer(2)), Frac(ZZ['x']))           ␣
↪              # needs sage.modules
Full MatrixSpace of 2 by 2 dense matrices
 over Fraction Field of Univariate Polynomial Ring in x over Real Double Field
>>> pushout(ZZ, MatrixSpace(ZZ[['x']], Integer(3), Integer(3)))               ␣
↪              # needs sage.modules
Full MatrixSpace of 3 by 3 dense matrices
 over Power Series Ring in x over Integer Ring
>>> pushout(QQ['x,y'], ZZ[['x']])
Univariate Polynomial Ring in y
 over Power Series Ring in x over Rational Field
>>> pushout(Frac(ZZ['x']), QQ[['x']])
Laurent Series Ring in x over Rational Field
```

A construction with `coercion_reversed=True` (currently only the *SubspaceFunctor* construction) is only applied if it leads to a valid coercion:

```
sage: # needs sage.modules
sage: A = ZZ^2
sage: V = span([[1, 2]], QQ)
sage: P = sage.categories.pushout.pushout(A, V)
sage: P
Vector space of dimension 2 over Rational Field
sage: P.has_coerce_map_from(A)
True

sage: # needs sage.modules
sage: V = (QQ^3).span([[1, 2, 3/4]])
sage: A = ZZ^3
sage: pushout(A, V)
Vector space of dimension 3 over Rational Field
sage: B = A.span([[0, 0, 2/3]])
```

(continues on next page)

(continued from previous page)

```
sage: pushout(B, V)
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[1 2 0]
[0 0 1]
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> A = ZZ**Integer(2)
>>> V = span([[Integer(1), Integer(2)]], QQ)
>>> P = sage.categories.pushout.pushout(A, V)
>>> P
Vector space of dimension 2 over Rational Field
>>> P.has_coerce_map_from(A)
True

>>> # needs sage.modules
>>> V = (QQ**Integer(3)).span([[Integer(1), Integer(2), Integer(3)/Integer(4)]])
>>> A = ZZ**Integer(3)
>>> pushout(A, V)
Vector space of dimension 3 over Rational Field
>>> B = A.span([[Integer(0), Integer(0), Integer(2)/Integer(3)]])
>>> pushout(B, V)
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[1 2 0]
[0 0 1]
```

Some more tests with `coercion_reversed=True`:

```
sage: from sage.categories.pushout import ConstructionFunctor
sage: class EvenPolynomialRing(type(QQ['x'])):
....:     def __init__(self, base, var):
....:         super().__init__(base, var)
....:         self.register_embedding(base[var])
....:     def __repr__(self):
....:         return "Even Power " + super().__repr__()
....:     def construction(self):
....:         return EvenPolynomialFunctor(), self.base()[self.variable_name()]
....:     def _coerce_map_from_(self, R):
....:         return self.base().has_coerce_map_from(R)
sage: class EvenPolynomialFunctor(ConstructionFunctor):
....:     rank = 10
....:     coercion_reversed = True
....:     def __init__(self):
....:         ConstructionFunctor.__init__(self, Rings(), Rings())
....:     def _apply_functor(self, R):
....:         return EvenPolynomialRing(R.base(), R.variable_name())
sage: pushout(EvenPolynomialRing(QQ, 'x'), ZZ)
Even Power Univariate Polynomial Ring in x over Rational Field
sage: pushout(EvenPolynomialRing(QQ, 'x'), QQ)
Even Power Univariate Polynomial Ring in x over Rational Field
```

(continues on next page)

(continued from previous page)

```
sage: pushout(EvenPolynomialRing(QQ, 'x'), RR)                                  #␣
↪needs sage.rings.real_mpfr
Even Power Univariate Polynomial Ring in x over Real Field with 53 bits of␣
↪precision

sage: pushout(EvenPolynomialRing(QQ, 'x'), ZZ['x'])
Univariate Polynomial Ring in x over Rational Field
sage: pushout(EvenPolynomialRing(QQ, 'x'), QQ['x'])
Univariate Polynomial Ring in x over Rational Field
sage: pushout(EvenPolynomialRing(QQ, 'x'), RR['x'])                             #␣
↪needs sage.rings.real_mpfr
Univariate Polynomial Ring in x over Real Field with 53 bits of precision

sage: pushout(EvenPolynomialRing(QQ, 'x'), EvenPolynomialRing(QQ, 'x'))
Even Power Univariate Polynomial Ring in x over Rational Field
sage: pushout(EvenPolynomialRing(QQ, 'x'), EvenPolynomialRing(RR, 'x'))         #␣
↪needs sage.rings.real_mpfr
Even Power Univariate Polynomial Ring in x over Real Field with 53 bits of␣
↪precision

sage: pushout(EvenPolynomialRing(QQ, 'x')^2, RR^2)                              #␣
↪needs sage.modules sage.rings.real_mpfr
Ambient free module of rank 2
 over the principal ideal domain Even Power Univariate Polynomial Ring in x
  over Real Field with 53 bits of precision
sage: pushout(EvenPolynomialRing(QQ, 'x')^2, RR['x']^2)                         #␣
↪needs sage.modules sage.rings.real_mpfr
Ambient free module of rank 2
 over the principal ideal domain Univariate Polynomial Ring in x
 over Real Field with 53 bits of precision
```

```
>>> from sage.all import *
>>> from sage.categories.pushout import ConstructionFunctor
>>> class EvenPolynomialRing(type(QQ['x'])):
...     def __init__(self, base, var):
...         super().__init__(base, var)
...         self.register_embedding(base[var])
...     def __repr__(self):
...         return "Even Power " + super().__repr__()
...     def construction(self):
...         return EvenPolynomialFunctor(), self.base()[self.variable_name()]
...     def _coerce_map_from_(self, R):
...         return self.base().has_coerce_map_from(R)
>>> class EvenPolynomialFunctor(ConstructionFunctor):
...     rank = Integer(10)
...     coercion_reversed = True
...     def __init__(self):
...         ConstructionFunctor.__init__(self, Rings(), Rings())
...     def _apply_functor(self, R):
...         return EvenPolynomialRing(R.base(), R.variable_name())
>>> pushout(EvenPolynomialRing(QQ, 'x'), ZZ)
Even Power Univariate Polynomial Ring in x over Rational Field
```

(continues on next page)

(continued from previous page)

```
>>> pushout(EvenPolynomialRing(QQ, 'x'), QQ)
Even Power Univariate Polynomial Ring in x over Rational Field
>>> pushout(EvenPolynomialRing(QQ, 'x'), RR)                                    #␣
↪needs sage.rings.real_mpfr
Even Power Univariate Polynomial Ring in x over Real Field with 53 bits of␣
↪precision

>>> pushout(EvenPolynomialRing(QQ, 'x'), ZZ['x'])
Univariate Polynomial Ring in x over Rational Field
>>> pushout(EvenPolynomialRing(QQ, 'x'), QQ['x'])
Univariate Polynomial Ring in x over Rational Field
>>> pushout(EvenPolynomialRing(QQ, 'x'), RR['x'])                               #␣
↪needs sage.rings.real_mpfr
Univariate Polynomial Ring in x over Real Field with 53 bits of precision

>>> pushout(EvenPolynomialRing(QQ, 'x'), EvenPolynomialRing(QQ, 'x'))
Even Power Univariate Polynomial Ring in x over Rational Field
>>> pushout(EvenPolynomialRing(QQ, 'x'), EvenPolynomialRing(RR, 'x'))           #␣
↪needs sage.rings.real_mpfr
Even Power Univariate Polynomial Ring in x over Real Field with 53 bits of␣
↪precision

>>> pushout(EvenPolynomialRing(QQ, 'x')**Integer(2), RR**Integer(2))            ␣
↪                # needs sage.modules sage.rings.real_mpfr
Ambient free module of rank 2
 over the principal ideal domain Even Power Univariate Polynomial Ring in x
  over Real Field with 53 bits of precision
>>> pushout(EvenPolynomialRing(QQ, 'x')**Integer(2), RR['x']**Integer(2))       ␣
↪                # needs sage.modules sage.rings.real_mpfr
Ambient free module of rank 2
 over the principal ideal domain Univariate Polynomial Ring in x
 over Real Field with 53 bits of precision
```

Some more tests related to univariate/multivariate constructions. We consider a generalization of polynomial rings, where in addition to the coefficient ring C we also specify an additive monoid E for the exponents of the indeterminate. In particular, the elements of such a parent are given by

$$\sum_{i=0}^{I} c_i X^{e_i}$$

with $c_i \in C$ and $e_i \in E$. We define

```
sage: class GPolynomialRing(Parent):
....:     def __init__(self, coefficients, var, exponents):
....:         self.coefficients = coefficients
....:         self.var = var
....:         self.exponents = exponents
....:         super().__init__(category=Rings())
....:     def _repr_(self):
....:         return 'Generalized Polynomial Ring in %s^(%s) over %s' % (
....:                self.var, self.exponents, self.coefficients)
....:     def construction(self):
```

(continues on next page)

(continued from previous page)

```
....:         return GPolynomialFunctor(self.var, self.exponents), self.
↪coefficients
....:     def _coerce_map_from_(self, R):
....:         return self.coefficients.has_coerce_map_from(R)
```

```
>>> from sage.all import *
>>> class GPolynomialRing(Parent):
...     def __init__(self, coefficients, var, exponents):
...         self.coefficients = coefficients
...         self.var = var
...         self.exponents = exponents
...         super().__init__(category=Rings())
...     def _repr_(self):
...         return 'Generalized Polynomial Ring in %s^(%s) over %s' % (
...                self.var, self.exponents, self.coefficients)
...     def construction(self):
...         return GPolynomialFunctor(self.var, self.exponents), self.coefficients
...     def _coerce_map_from_(self, R):
...         return self.coefficients.has_coerce_map_from(R)
```

and

```
sage: class GPolynomialFunctor(ConstructionFunctor):
....:     rank = 10
....:     def __init__(self, var, exponents):
....:         self.var = var
....:         self.exponents = exponents
....:         ConstructionFunctor.__init__(self, Rings(), Rings())
....:     def _repr_(self):
....:         return 'GPoly[%s^(%s)]' % (self.var, self.exponents)
....:     def _apply_functor(self, coefficients):
....:         return GPolynomialRing(coefficients, self.var, self.exponents)
....:     def merge(self, other):
....:         if isinstance(other, GPolynomialFunctor) and self.var == other.var:
....:             exponents = pushout(self.exponents, other.exponents)
....:             return GPolynomialFunctor(self.var, exponents)
```

```
>>> from sage.all import *
>>> class GPolynomialFunctor(ConstructionFunctor):
...     rank = Integer(10)
...     def __init__(self, var, exponents):
...         self.var = var
...         self.exponents = exponents
...         ConstructionFunctor.__init__(self, Rings(), Rings())
...     def _repr_(self):
...         return 'GPoly[%s^(%s)]' % (self.var, self.exponents)
...     def _apply_functor(self, coefficients):
...         return GPolynomialRing(coefficients, self.var, self.exponents)
...     def merge(self, other):
...         if isinstance(other, GPolynomialFunctor) and self.var == other.var:
...             exponents = pushout(self.exponents, other.exponents)
...             return GPolynomialFunctor(self.var, exponents)
```

We can construct a parent now in two different ways:

```
sage: GPolynomialRing(QQ, 'X', ZZ)
Generalized Polynomial Ring in X^(Integer Ring) over Rational Field
sage: GP_ZZ = GPolynomialFunctor('X', ZZ); GP_ZZ
GPoly[X^(Integer Ring)]
sage: GP_ZZ(QQ)
Generalized Polynomial Ring in X^(Integer Ring) over Rational Field
```

```
>>> from sage.all import *
>>> GPolynomialRing(QQ, 'X', ZZ)
Generalized Polynomial Ring in X^(Integer Ring) over Rational Field
>>> GP_ZZ = GPolynomialFunctor('X', ZZ); GP_ZZ
GPoly[X^(Integer Ring)]
>>> GP_ZZ(QQ)
Generalized Polynomial Ring in X^(Integer Ring) over Rational Field
```

Since the construction

```
sage: GP_ZZ(QQ).construction()
(GPoly[X^(Integer Ring)], Rational Field)
```

```
>>> from sage.all import *
>>> GP_ZZ(QQ).construction()
(GPoly[X^(Integer Ring)], Rational Field)
```

uses the coefficient ring, we have the usual coercion with respect to this parameter:

```
sage: pushout(GP_ZZ(ZZ), GP_ZZ(QQ))
Generalized Polynomial Ring in X^(Integer Ring) over Rational Field
sage: pushout(GP_ZZ(ZZ['t']), GP_ZZ(QQ))
Generalized Polynomial Ring in X^(Integer Ring)
  over Univariate Polynomial Ring in t over Rational Field
sage: pushout(GP_ZZ(ZZ['a,b']), GP_ZZ(ZZ['b,c']))
Generalized Polynomial Ring in X^(Integer Ring)
  over Multivariate Polynomial Ring in a, b, c over Integer Ring
sage: pushout(GP_ZZ(ZZ['a,b']), GP_ZZ(QQ['b,c']))
Generalized Polynomial Ring in X^(Integer Ring)
  over Multivariate Polynomial Ring in a, b, c over Rational Field
sage: pushout(GP_ZZ(ZZ['a,b']), GP_ZZ(ZZ['c,d']))
Traceback (most recent call last):
...
CoercionException: ('Ambiguous Base Extension', ...)
```

```
>>> from sage.all import *
>>> pushout(GP_ZZ(ZZ), GP_ZZ(QQ))
Generalized Polynomial Ring in X^(Integer Ring) over Rational Field
>>> pushout(GP_ZZ(ZZ['t']), GP_ZZ(QQ))
Generalized Polynomial Ring in X^(Integer Ring)
  over Univariate Polynomial Ring in t over Rational Field
>>> pushout(GP_ZZ(ZZ['a,b']), GP_ZZ(ZZ['b,c']))
Generalized Polynomial Ring in X^(Integer Ring)
  over Multivariate Polynomial Ring in a, b, c over Integer Ring
```

(continues on next page)

(continued from previous page)

```
>>> pushout(GP_ZZ(ZZ['a,b']), GP_ZZ(QQ['b,c']))
Generalized Polynomial Ring in X^(Integer Ring)
  over Multivariate Polynomial Ring in a, b, c over Rational Field
>>> pushout(GP_ZZ(ZZ['a,b']), GP_ZZ(ZZ['c,d']))
Traceback (most recent call last):
...
CoercionException: ('Ambiguous Base Extension', ...)
```

```
sage: GP_QQ = GPolynomialFunctor('X', QQ)
sage: pushout(GP_ZZ(ZZ), GP_QQ(ZZ))
Generalized Polynomial Ring in X^(Rational Field) over Integer Ring
sage: pushout(GP_QQ(ZZ), GP_ZZ(ZZ))
Generalized Polynomial Ring in X^(Rational Field) over Integer Ring
```

```
>>> from sage.all import *
>>> GP_QQ = GPolynomialFunctor('X', QQ)
>>> pushout(GP_ZZ(ZZ), GP_QQ(ZZ))
Generalized Polynomial Ring in X^(Rational Field) over Integer Ring
>>> pushout(GP_QQ(ZZ), GP_ZZ(ZZ))
Generalized Polynomial Ring in X^(Rational Field) over Integer Ring
```

```
sage: GP_ZZt = GPolynomialFunctor('X', ZZ['t'])
sage: pushout(GP_ZZt(ZZ), GP_QQ(ZZ))
Generalized Polynomial Ring in X^(Univariate Polynomial Ring in t
  over Rational Field) over Integer Ring
```

```
>>> from sage.all import *
>>> GP_ZZt = GPolynomialFunctor('X', ZZ['t'])
>>> pushout(GP_ZZt(ZZ), GP_QQ(ZZ))
Generalized Polynomial Ring in X^(Univariate Polynomial Ring in t
  over Rational Field) over Integer Ring
```

```
sage: pushout(GP_ZZ(ZZ), GP_QQ(QQ))
Generalized Polynomial Ring in X^(Rational Field) over Rational Field
sage: pushout(GP_ZZ(QQ), GP_QQ(ZZ))
Generalized Polynomial Ring in X^(Rational Field) over Rational Field
sage: pushout(GP_ZZt(QQ), GP_QQ(ZZ))
Generalized Polynomial Ring in X^(Univariate Polynomial Ring in t
  over Rational Field) over Rational Field
sage: pushout(GP_ZZt(ZZ), GP_QQ(QQ))
Generalized Polynomial Ring in X^(Univariate Polynomial Ring in t
  over Rational Field) over Rational Field
sage: pushout(GP_ZZt(ZZ['a,b']), GP_QQ(ZZ['c,d']))
Traceback (most recent call last):
...
CoercionException: ('Ambiguous Base Extension', ...)
sage: pushout(GP_ZZt(ZZ['a,b']), GP_QQ(ZZ['b,c']))
Generalized Polynomial Ring
 in X^(Univariate Polynomial Ring in t over Rational Field)
  over Multivariate Polynomial Ring in a, b, c over Integer Ring
```

```
>>> from sage.all import *
>>> pushout(GP_ZZ(ZZ), GP_QQ(QQ))
Generalized Polynomial Ring in X^(Rational Field) over Rational Field
>>> pushout(GP_ZZ(QQ), GP_QQ(ZZ))
Generalized Polynomial Ring in X^(Rational Field) over Rational Field
>>> pushout(GP_ZZt(QQ), GP_QQ(ZZ))
Generalized Polynomial Ring in X^(Univariate Polynomial Ring in t
  over Rational Field) over Rational Field
>>> pushout(GP_ZZt(ZZ), GP_QQ(QQ))
Generalized Polynomial Ring in X^(Univariate Polynomial Ring in t
  over Rational Field) over Rational Field
>>> pushout(GP_ZZt(ZZ['a,b']), GP_QQ(ZZ['c,d']))
Traceback (most recent call last):
...
CoercionException: ('Ambiguous Base Extension', ...)
>>> pushout(GP_ZZt(ZZ['a,b']), GP_QQ(ZZ['b,c']))
Generalized Polynomial Ring
 in X^(Univariate Polynomial Ring in t over Rational Field)
  over Multivariate Polynomial Ring in a, b, c over Integer Ring
```

Some tests with Cartesian products:

```
sage: from sage.sets.cartesian_product import CartesianProduct
sage: A = CartesianProduct((ZZ['x'], QQ['y'], QQ['z']),
....:                      Sets().CartesianProducts())
sage: B = CartesianProduct((ZZ['x'], ZZ['y'], ZZ['t']['z']),
....:                      Sets().CartesianProducts())
sage: A.construction()
(The cartesian_product functorial construction,
 (Univariate Polynomial Ring in x over Integer Ring,
  Univariate Polynomial Ring in y over Rational Field,
  Univariate Polynomial Ring in z over Rational Field))
sage: pushout(A, B)
The Cartesian product of
 (Univariate Polynomial Ring in x over Integer Ring,
  Univariate Polynomial Ring in y over Rational Field,
  Univariate Polynomial Ring in z over
   Univariate Polynomial Ring in t over Rational Field)
sage: pushout(ZZ, cartesian_product([ZZ, QQ]))
Traceback (most recent call last):
...
CoercionException: 'NoneType' object is not iterable
```

```
>>> from sage.all import *
>>> from sage.sets.cartesian_product import CartesianProduct
>>> A = CartesianProduct((ZZ['x'], QQ['y'], QQ['z']),
...                      Sets().CartesianProducts())
>>> B = CartesianProduct((ZZ['x'], ZZ['y'], ZZ['t']['z']),
...                      Sets().CartesianProducts())
>>> A.construction()
(The cartesian_product functorial construction,
 (Univariate Polynomial Ring in x over Integer Ring,
  Univariate Polynomial Ring in y over Rational Field,
```

(continues on next page)

(continued from previous page)

```
  Univariate Polynomial Ring in z over Rational Field))
>>> pushout(A, B)
The Cartesian product of
 (Univariate Polynomial Ring in x over Integer Ring,
  Univariate Polynomial Ring in y over Rational Field,
  Univariate Polynomial Ring in z over
   Univariate Polynomial Ring in t over Rational Field)
>>> pushout(ZZ, cartesian_product([ZZ, QQ]))
Traceback (most recent call last):
...
CoercionException: 'NoneType' object is not iterable
```

```
sage: from sage.categories.pushout import PolynomialFunctor
sage: from sage.sets.cartesian_product import CartesianProduct
sage: class CartesianProductPoly(CartesianProduct):
....:     def __init__(self, polynomial_rings):
....:         sort = sorted(polynomial_rings,
....:                       key=lambda P: P.variable_name())
....:         super().__init__(sort, Sets().CartesianProducts())
....:     def vars(self):
....:         return tuple(P.variable_name()
....:                      for P in self.cartesian_factors())
....:     def _pushout_(self, other):
....:         if isinstance(other, CartesianProductPoly):
....:             s_vars = self.vars()
....:             o_vars = other.vars()
....:             if s_vars == o_vars:
....:                 return
....:             return pushout(CartesianProductPoly(
....:                     self.cartesian_factors() +
....:                     tuple(f for f in other.cartesian_factors()
....:                           if f.variable_name() not in s_vars)),
....:                 CartesianProductPoly(
....:                     other.cartesian_factors() +
....:                     tuple(f for f in self.cartesian_factors()
....:                           if f.variable_name() not in o_vars)))
....:         C = other.construction()
....:         if C is None:
....:             return
....:         elif isinstance(C[0], PolynomialFunctor):
....:             return pushout(self, CartesianProductPoly((other,)))
```

```
>>> from sage.all import *
>>> from sage.categories.pushout import PolynomialFunctor
>>> from sage.sets.cartesian_product import CartesianProduct
>>> class CartesianProductPoly(CartesianProduct):
...     def __init__(self, polynomial_rings):
...         sort = sorted(polynomial_rings,
...                       key=lambda P: P.variable_name())
...         super().__init__(sort, Sets().CartesianProducts())
...     def vars(self):
```

(continues on next page)

(continued from previous page)

```
...         return tuple(P.variable_name()
...                      for P in self.cartesian_factors())
...     def _pushout_(self, other):
...         if isinstance(other, CartesianProductPoly):
...             s_vars = self.vars()
...             o_vars = other.vars()
...             if s_vars == o_vars:
...                 return
...             return pushout(CartesianProductPoly(
...                     self.cartesian_factors() +
...                     tuple(f for f in other.cartesian_factors()
...                           if f.variable_name() not in s_vars)),
...                 CartesianProductPoly(
...                     other.cartesian_factors() +
...                     tuple(f for f in self.cartesian_factors()
...                           if f.variable_name() not in o_vars)))
...         C = other.construction()
...         if C is None:
...             return
...         elif isinstance(C[Integer(0)], PolynomialFunctor):
...             return pushout(self, CartesianProductPoly((other,)))
```

```
sage: pushout(CartesianProductPoly((ZZ['x'],)),
....:         CartesianProductPoly((ZZ['y'],)))
The Cartesian product of
 (Univariate Polynomial Ring in x over Integer Ring,
  Univariate Polynomial Ring in y over Integer Ring)
sage: pushout(CartesianProductPoly((ZZ['x'], ZZ['y'])),
....:         CartesianProductPoly((ZZ['x'], ZZ['z'])))
The Cartesian product of
 (Univariate Polynomial Ring in x over Integer Ring,
  Univariate Polynomial Ring in y over Integer Ring,
  Univariate Polynomial Ring in z over Integer Ring)
sage: pushout(CartesianProductPoly((QQ['a,b']['x'], QQ['y'])),                #␣
↪needs sage.symbolic
....:         CartesianProductPoly((ZZ['b,c']['x'], SR['z'])))
The Cartesian product of
 (Univariate Polynomial Ring in x over
    Multivariate Polynomial Ring in a, b, c over Rational Field,
  Univariate Polynomial Ring in y over Rational Field,
  Univariate Polynomial Ring in z over Symbolic Ring)
```

```
>>> from sage.all import *
>>> pushout(CartesianProductPoly((ZZ['x'],)),
...         CartesianProductPoly((ZZ['y'],)))
The Cartesian product of
 (Univariate Polynomial Ring in x over Integer Ring,
  Univariate Polynomial Ring in y over Integer Ring)
>>> pushout(CartesianProductPoly((ZZ['x'], ZZ['y'])),
...         CartesianProductPoly((ZZ['x'], ZZ['z'])))
The Cartesian product of
```

(continues on next page)

(continued from previous page)

```
 (Univariate Polynomial Ring in x over Integer Ring,
  Univariate Polynomial Ring in y over Integer Ring,
  Univariate Polynomial Ring in z over Integer Ring)
>>> pushout(CartesianProductPoly((QQ['a,b']['x'], QQ['y'])),                    #␣
↪needs sage.symbolic
...         CartesianProductPoly((ZZ['b,c']['x'], SR['z'])))
The Cartesian product of
 (Univariate Polynomial Ring in x over
    Multivariate Polynomial Ring in a, b, c over Rational Field,
  Univariate Polynomial Ring in y over Rational Field,
  Univariate Polynomial Ring in z over Symbolic Ring)
```

```
sage: pushout(CartesianProductPoly((ZZ['x'],)), ZZ['y'])
The Cartesian product of
 (Univariate Polynomial Ring in x over Integer Ring,
  Univariate Polynomial Ring in y over Integer Ring)
sage: pushout(QQ['b,c']['y'], CartesianProductPoly((ZZ['a,b']['x'],)))
The Cartesian product of
 (Univariate Polynomial Ring in x over
    Multivariate Polynomial Ring in a, b over Integer Ring,
  Univariate Polynomial Ring in y over
    Multivariate Polynomial Ring in b, c over Rational Field)
```

```
>>> from sage.all import *
>>> pushout(CartesianProductPoly((ZZ['x'],)), ZZ['y'])
The Cartesian product of
 (Univariate Polynomial Ring in x over Integer Ring,
  Univariate Polynomial Ring in y over Integer Ring)
>>> pushout(QQ['b,c']['y'], CartesianProductPoly((ZZ['a,b']['x'],)))
The Cartesian product of
 (Univariate Polynomial Ring in x over
    Multivariate Polynomial Ring in a, b over Integer Ring,
  Univariate Polynomial Ring in y over
    Multivariate Polynomial Ring in b, c over Rational Field)
```

```
sage: pushout(CartesianProductPoly((ZZ['x'],)), ZZ)
Traceback (most recent call last):
...
CoercionException: No common base ("join") found for
The cartesian_product functorial construction(...) and None(Integer Ring):
(Multivariate) functors are incompatible.
```

```
>>> from sage.all import *
>>> pushout(CartesianProductPoly((ZZ['x'],)), ZZ)
Traceback (most recent call last):
...
CoercionException: No common base ("join") found for
The cartesian_product functorial construction(...) and None(Integer Ring):
(Multivariate) functors are incompatible.
```

AUTHORS:

- Robert Bradshaw
- Peter Bruin
- Simon King
- Daniel Krenn
- David Roe

sage.categories.pushout.**pushout_lattice**(*R*, *S*)

Given a pair of objects R and S, try to construct a reasonable object Y and return maps such that canonically $R \leftarrow Y \rightarrow S$.

ALGORITHM:

This is based on the model that arose from much discussion at Sage Days 4. Going up the tower of constructions of R and S (e.g. the reals come from the rationals come from the integers), try to find a common parent, and then try to fill in a lattice with these two towers as sides with the top as the common ancestor and the bottom will be the desired ring.

See the code for a specific worked-out example.

EXAMPLES:

```
sage: from sage.categories.pushout import pushout_lattice
sage: A, B = pushout_lattice(Qp(7), Frac(ZZ['x']))                          #␣
↪needs sage.rings.padics
sage: A.codomain()                                                          #␣
↪needs sage.rings.padics
Fraction Field of Univariate Polynomial Ring in x
 over 7-adic Field with capped relative precision 20
sage: A.codomain() is B.codomain()                                          #␣
↪needs sage.rings.padics
True
sage: A, B = pushout_lattice(ZZ, MatrixSpace(ZZ[['x']], 3, 3))             #␣
↪needs sage.modules
sage: B                                                                     #␣
↪needs sage.modules
Identity endomorphism of Full MatrixSpace of 3 by 3 dense matrices
 over Power Series Ring in x over Integer Ring
```

```
>>> from sage.all import *
>>> from sage.categories.pushout import pushout_lattice
>>> A, B = pushout_lattice(Qp(Integer(7)), Frac(ZZ['x']))                    ␣
↪     # needs sage.rings.padics
>>> A.codomain()                                                           #␣
↪needs sage.rings.padics
Fraction Field of Univariate Polynomial Ring in x
 over 7-adic Field with capped relative precision 20
>>> A.codomain() is B.codomain()                                           #␣
↪needs sage.rings.padics
True
>>> A, B = pushout_lattice(ZZ, MatrixSpace(ZZ[['x']], Integer(3), Integer(3)))  ␣
↪              # needs sage.modules
>>> B                                                                      #␣
↪needs sage.modules
```

(continues on next page)

(continued from previous page)

```
Identity endomorphism of Full MatrixSpace of 3 by 3 dense matrices
 over Power Series Ring in x over Integer Ring
```

AUTHOR:

- Robert Bradshaw

sage.categories.pushout.**type_to_parent**(*P*)

An auxiliary function that is used in *pushout()*.

INPUT:

- P – a type

OUTPUT: a Sage parent structure corresponding to the given type

5.5 Group, ring, etc. actions on objects

The terminology and notation used is suggestive of groups acting on sets, but this framework can be used for modules, algebras, etc.

A group action $G \times S \to S$ is a functor from G to Sets.

Warning

An *Action* object only keeps a weak reference to the underlying set which is acted upon. This decision was made in Issue #715 in order to allow garbage collection within the coercion framework (this is where actions are mainly used) and avoid memory leaks.

```
sage: from sage.categories.action import Action
sage: class P: pass
sage: A = Action(P(),P())
sage: import gc
sage: _ = gc.collect()
sage: A
<repr(<sage.categories.action.Action at 0x...>) failed:
 RuntimeError: This action acted on a set that became garbage collected>
```

```
>>> from sage.all import *
>>> from sage.categories.action import Action
>>> class P: pass
>>> A = Action(P(),P())
>>> import gc
>>> _ = gc.collect()
>>> A
<repr(<sage.categories.action.Action at 0x...>) failed:
 RuntimeError: This action acted on a set that became garbage collected>
```

To avoid garbage collection of the underlying set, it is sufficient to create a strong reference to it before the action is created.

```
sage: _ = gc.collect()
sage: from sage.categories.action import Action
sage: class P: pass
sage: q = P()
sage: A = Action(P(),q)
sage: gc.collect()
0
sage: A
Left action by <__main__.P ... at ...> on <__main__.P ... at ...>
```

```
>>> from sage.all import *
>>> _ = gc.collect()
>>> from sage.categories.action import Action
>>> class P: pass
>>> q = P()
>>> A = Action(P(),q)
>>> gc.collect()
0
>>> A
Left action by <__main__.P ... at ...> on <__main__.P ... at ...>
```

AUTHOR:

- Robert Bradshaw: initial version

class sage.categories.action.**Action**

Bases: Functor

The action of G on S.

INPUT:

- G – a parent or Python type
- S – a parent or Python type
- is_left – boolean (default: True); whether elements of G are on the left
- op – (default: None) operation. This is not used by *Action* itself, but other classes may use it

G

act(*g*, *x*)

This is a consistent interface for acting on x by g, regardless of whether it's a left or right action.

If needed, g and x are converted to the correct parent.

EXAMPLES:

```
sage: R.<x> = ZZ []
sage: from sage.structure.coerce_actions import IntegerMulAction
sage: A = IntegerMulAction(ZZ, R, True)   # Left action
sage: A.act(5, x)
5*x
sage: A.act(int(5), x)
5*x
sage: A = IntegerMulAction(ZZ, R, False)  # Right action
sage: A.act(5, x)
5*x
sage: A.act(int(5), x)
5*x
```

```
>>> from sage.all import *
>>> R = ZZ ['x']; (x,) = R._first_ngens(1)
>>> from sage.structure.coerce_actions import IntegerMulAction
>>> A = IntegerMulAction(ZZ, R, True)   # Left action
```

(continues on next page)

(continued from previous page)

```
>>> A.act(Integer(5), x)
5*x
>>> A.act(int(Integer(5)), x)
5*x
>>> A = IntegerMulAction(ZZ, R, False)  # Right action
>>> A.act(Integer(5), x)
5*x
>>> A.act(int(Integer(5)), x)
5*x
```

actor()

codomain()

domain()

is_left()

left_domain()

op

operation()

right_domain()

class sage.categories.action.**ActionEndomorphism**

Bases: Morphism

The endomorphism defined by the action of one element.

EXAMPLES:

```
sage: A = ZZ['x'].get_action(QQ, self_on_left=False, op=operator.mul)
sage: A
Left scalar multiplication by Rational Field
 on Univariate Polynomial Ring in x over Integer Ring
sage: A(1/2)
Action of 1/2 on Univariate Polynomial Ring in x over Integer Ring
under Left scalar multiplication by Rational Field on Univariate
Polynomial Ring in x over Integer Ring.
```

```
>>> from sage.all import *
>>> A = ZZ['x'].get_action(QQ, self_on_left=False, op=operator.mul)
>>> A
Left scalar multiplication by Rational Field
 on Univariate Polynomial Ring in x over Integer Ring
>>> A(Integer(1)/Integer(2))
Action of 1/2 on Univariate Polynomial Ring in x over Integer Ring
under Left scalar multiplication by Rational Field on Univariate
Polynomial Ring in x over Integer Ring.
```

class sage.categories.action.**InverseAction**

Bases: *Action*

An action that acts as the inverse of the given action.

EXAMPLES:

```
sage: V = QQ^3                                                                  #␣
↪needs sage.modules
sage: v = V((1, 2, 3))                                                          #␣
↪needs sage.modules
sage: cm = get_coercion_model()

sage: # needs sage.modules
sage: a = cm.get_action(V, QQ, operator.mul)
sage: a
Right scalar multiplication by Rational Field
 on Vector space of dimension 3 over Rational Field
sage: ~a
Right inverse action by Rational Field
 on Vector space of dimension 3 over Rational Field
sage: (~a)(v, 1/3)
(3, 6, 9)

sage: # needs sage.modules
sage: b = cm.get_action(QQ, V, operator.mul)
sage: b
Left scalar multiplication by Rational Field
 on Vector space of dimension 3 over Rational Field
sage: ~b
Left inverse action by Rational Field
 on Vector space of dimension 3 over Rational Field
sage: (~b)(1/3, v)
(3, 6, 9)

sage: c = cm.get_action(ZZ, list, operator.mul)
sage: c
Left action by Integer Ring on <... 'list'>
sage: ~c
Traceback (most recent call last):
...
TypeError: no inverse defined for Left action by Integer Ring on <... 'list'>
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3)                                                          ␣
↪      # needs sage.modules
>>> v = V((Integer(1), Integer(2), Integer(3)))                                 ␣
↪                        # needs sage.modules
>>> cm = get_coercion_model()

>>> # needs sage.modules
>>> a = cm.get_action(V, QQ, operator.mul)
>>> a
Right scalar multiplication by Rational Field
 on Vector space of dimension 3 over Rational Field
>>> ~a
Right inverse action by Rational Field
 on Vector space of dimension 3 over Rational Field
```

(continues on next page)

(continued from previous page)

```
>>> (~a)(v, Integer(1)/Integer(3))
(3, 6, 9)

>>> # needs sage.modules
>>> b = cm.get_action(QQ, V, operator.mul)
>>> b
Left scalar multiplication by Rational Field
 on Vector space of dimension 3 over Rational Field
>>> ~b
Left inverse action by Rational Field
 on Vector space of dimension 3 over Rational Field
>>> (~b)(Integer(1)/Integer(3), v)
(3, 6, 9)

>>> c = cm.get_action(ZZ, list, operator.mul)
>>> c
Left action by Integer Ring on <... 'list'>
>>> ~c
Traceback (most recent call last):
...
TypeError: no inverse defined for Left action by Integer Ring on <... 'list'>
```

codomain()

class sage.categories.action.**PrecomposedAction**

Bases: *Action*

A precomposed action first applies given maps, and then applying an action to the return values of the maps.

EXAMPLES:

We demonstrate that an example discussed on Issue #14711 did not become a problem:

```
sage: # needs sage.libs.flint sage.modular
sage: E = ModularSymbols(11).2
sage: s = E.modular_symbol_rep()
sage: del E,s
sage: import gc
sage: _ = gc.collect()
sage: E = ModularSymbols(11).2
sage: v = E.manin_symbol_rep()
sage: c,x = v[0]
sage: y = x.modular_symbol_rep()
sage: coercion_model.get_action(QQ, parent(y), op=operator.mul)
Left scalar multiplication by Rational Field
 on Abelian Group of all Formal Finite Sums over Rational Field
 with precomposition on right by Coercion map:
  From: Abelian Group of all Formal Finite Sums over Integer Ring
  To:   Abelian Group of all Formal Finite Sums over Rational Field
```

```
>>> from sage.all import *
>>> # needs sage.libs.flint sage.modular
>>> E = ModularSymbols(Integer(11)).gen(2)
```

(continues on next page)

(continued from previous page)

```
>>> s = E.modular_symbol_rep()
>>> del E,s
>>> import gc
>>> _ = gc.collect()
>>> E = ModularSymbols(Integer(11)).gen(2)
>>> v = E.manin_symbol_rep()
>>> c,x = v[Integer(0)]
>>> y = x.modular_symbol_rep()
>>> coercion_model.get_action(QQ, parent(y), op=operator.mul)
Left scalar multiplication by Rational Field
 on Abelian Group of all Formal Finite Sums over Rational Field
 with precomposition on right by Coercion map:
  From: Abelian Group of all Formal Finite Sums over Integer Ring
  To:   Abelian Group of all Formal Finite Sums over Rational Field
```

codomain()

domain()

left_precomposition

The left map to precompose with, or `None` if there is no left precomposition map.

right_precomposition

The right map to precompose with, or `None` if there is no right precomposition map.

5.6 Containers for storing coercion data

This module provides *TripleDict* and *MonoDict*. These are structures similar to `WeakKeyDictionary` in Python's weakref module, and are optimized for lookup speed. The keys for *TripleDict* consist of triples (k1,k2,k3) and are looked up by identity rather than equality. The keys are stored by weakrefs if possible. If any one of the components k1, k2, k3 gets garbage collected, then the entry is removed from the *TripleDict*.

Key components that do not allow for weakrefs are stored via a normal refcounted reference. That means that any entry stored using a triple (k1,k2,k3) so that none of the k1,k2,k3 allows a weak reference behaves as an entry in a normal dictionary: Its existence in *TripleDict* prevents it from being garbage collected.

That container currently is used to store coercion and conversion maps between two parents (Issue #715) and to store homsets of pairs of objects of a category (Issue #11521). In both cases, it is essential that the parent structures remain garbage collectable, it is essential that the data access is faster than with a usual `WeakKeyDictionary`, and we enforce the "unique parent condition" in Sage (parent structures should be identical if they are equal).

MonoDict behaves similarly, but it takes a single item as a key. It is used for caching the parents which allow a coercion map into a fixed other parent (Issue #12313).

By Issue #14159, *MonoDict* and *TripleDict* can be optionally used with weak references on the values.

Note that this kind of dictionary is also used for caching actions and coerce maps. In previous versions of Sage, the cache was by strong references and resulted in a memory leak in the following example. However, this leak was fixed by Issue #715, using weak references:

```
sage: # needs sage.combinat sage.modules sage.rings.finite_rings
sage: K.<t> = GF(2^55)
sage: for i in range(20):
....:     a = K.random_element()
```

(continues on next page)

(continued from previous page)

```
....:     E = EllipticCurve(j=a)
....:     P = E.random_point()
....:     Q = 2*P
sage: L = [Partitions(n) for n in range(200)]  # purge strong cache in␣
↪CachedRepresentation
sage: import gc
sage: n = gc.collect()
sage: from sage.schemes.elliptic_curves.ell_finite_field import EllipticCurve_finite_
↪field
sage: LE = [x for x in gc.get_objects() if isinstance(x, EllipticCurve_finite_field)]
sage: len(LE)
1
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.modules sage.rings.finite_rings
>>> K = GF(Integer(2)**Integer(55), names=('t',)); (t,) = K._first_ngens(1)
>>> for i in range(Integer(20)):
...     a = K.random_element()
...     E = EllipticCurve(j=a)
...     P = E.random_point()
...     Q = Integer(2)*P
>>> L = [Partitions(n) for n in range(Integer(200))]  # purge strong cache in␣
↪CachedRepresentation
>>> import gc
>>> n = gc.collect()
>>> from sage.schemes.elliptic_curves.ell_finite_field import EllipticCurve_finite_
↪field
>>> LE = [x for x in gc.get_objects() if isinstance(x, EllipticCurve_finite_field)]
>>> len(LE)
1
```

class sage.structure.coerce_dict.**MonoDict**

Bases: `object`

This is a hashtable specifically designed for (read) speed in the coercion model.

It differs from a python WeakKeyDictionary in the following important ways:

- Comparison is done using the 'is' rather than '==' operator.
- Only weak references to the keys are stored if at all possible. Keys that do not allow for weak references are stored with a normal refcounted reference.
- The callback of the weak references is safe against recursion, see below.

There are special cdef set/get methods for faster access. It is bare-bones in the sense that not all dictionary methods are implemented.

IMPLEMENTATION:

It is implemented as a hash table with open addressing, similar to python's dict.

INPUT:

- `data` – (optional) iterable defining initial data, as dict or iterable of (key, value) pairs
- `weak_values` – boolean (default: `False`); if it is `True`, weak references to the values in this dictionary will be used, when possible

EXAMPLES:

```
sage: from sage.structure.coerce_dict import MonoDict
sage: L = MonoDict()
sage: a = 'a'; b = 'ab'; c = '-15'
sage: L[a] = 1
sage: L[b] = 2
sage: L[c] = 3
```

```
>>> from sage.all import *
>>> from sage.structure.coerce_dict import MonoDict
>>> L = MonoDict()
>>> a = 'a'; b = 'ab'; c = '-15'
>>> L[a] = Integer(1)
>>> L[b] = Integer(2)
>>> L[c] = Integer(3)
```

The key is expected to be a unique object. Hence, the item stored for `c` cannot be obtained by providing another equal string:

```
sage: L[a]
1
sage: L[b]
2
sage: L[c]
3
sage: L['-15']
Traceback (most recent call last):
...
KeyError: '-15'
```

```
>>> from sage.all import *
>>> L[a]
1
>>> L[b]
2
>>> L[c]
3
>>> L['-15']
Traceback (most recent call last):
...
KeyError: '-15'
```

Not all features of Python dictionaries are available, but iteration over the dictionary items is possible:

```
sage: sorted(L.items())
[('-15', 3), ('a', 1), ('ab', 2)]
sage: del L[c]
sage: sorted(L.items())
[('a', 1), ('ab', 2)]
sage: len(L)
2
sage: for i in range(1000):
```

(continues on next page)

(continued from previous page)

```
....:     L[i] = i
sage: len(L)
1002
sage: L['a']
1
sage: L['c']
Traceback (most recent call last):
...
KeyError: 'c'
```

```
>>> from sage.all import *
>>> sorted(L.items())
[('-15', 3), ('a', 1), ('ab', 2)]
>>> del L[c]
>>> sorted(L.items())
[('a', 1), ('ab', 2)]
>>> len(L)
2
>>> for i in range(Integer(1000)):
...     L[i] = i
>>> len(L)
1002
>>> L['a']
1
>>> L['c']
Traceback (most recent call last):
...
KeyError: 'c'
```

Note that `MW` also accepts values that do not allow for weak references:

```
    sage: MW[k] = int(5)
    sage: MW[k]
    5

The following demonstrates that :class:`MonoDict` is safer than
:class:`~weakref.WeakKeyDictionary` against recursions created by nested
callbacks; compare :issue:`15069` (the mechanism used now is different, though)::

    sage: M = MonoDict()
    sage: class A: pass
    sage: a = A()
    sage: prev = a
    sage: for i in range(1000):
    ....:     newA = A()
    ....:     M[prev] = newA
    ....:     prev = newA
    sage: len(M)
    1000
    sage: del a
    sage: len(M)
    0
```

(continues on next page)

(continued from previous page)

```
The corresponding example with a Python :class:`weakref.WeakKeyDictionary`
would result in a too deep recursion during deletion of the dictionary
items::

    sage: import weakref
    sage: M = weakref.WeakKeyDictionary()
    sage: a = A()
    sage: prev = a
    sage: for i in range(1000):
    ....:     newA = A()
    ....:     M[prev] = newA
    ....:     prev = newA
    sage: len(M)
    1000

Check that also in the presence of circular references, :class:`MonoDict`
gets properly collected::

    sage: import gc
    sage: def count_type(T):
    ....:     return len([c for c in gc.get_objects() if isinstance(c,T)])
    sage: gc.freeze()  # so that gc.collect() only deals with our trash
    sage: N = count_type(MonoDict)
    sage: for i in range(100):
    ....:     V = [MonoDict({"id":j+100*i}) for j in range(100)]
    ....:     n = len(V)
    ....:     for i in range(n): V[i][V[(i+1)%n]] = (i+1)%n
    ....:     del V
    ....:     _ = gc.collect()
    ....:     assert count_type(MonoDict) == N
    sage: count_type(MonoDict) == N
    True
    sage: gc.unfreeze()

AUTHORS:

- Simon King (2012-01)
- Nils Bruin (2012-08)
- Simon King (2013-02)
- Nils Bruin (2013-11)
```

copy()

Return a copy of this *MonoDict* as Python dict.

EXAMPLES:

```
sage: from sage.structure.coerce_dict import MonoDict
sage: L = MonoDict()
sage: L[1] = 42
sage: L.copy()
{1: 42}
```

```
>>> from sage.all import *
>>> from sage.structure.coerce_dict import MonoDict
>>> L = MonoDict()
>>> L[Integer(1)] = Integer(42)
>>> L.copy()
{1: 42}
```

items()

Iterate over the `(key, value)` pairs of this *MonoDict*.

EXAMPLES:

```
sage: from sage.structure.coerce_dict import MonoDict
sage: L = MonoDict()
sage: L[1] = None
sage: L[2] = True
sage: L.items()
<...generator object at ...>
sage: sorted(L.items())
[(1, None), (2, True)]
```

```
>>> from sage.all import *
>>> from sage.structure.coerce_dict import MonoDict
>>> L = MonoDict()
>>> L[Integer(1)] = None
>>> L[Integer(2)] = True
>>> L.items()
<...generator object at ...>
>>> sorted(L.items())
[(1, None), (2, True)]
```

class sage.structure.coerce_dict.**MonoDictEraser**

Bases: `object`

Erase items from a *MonoDict* when a weak reference becomes invalid.

This is of internal use only. Instances of this class will be passed as a callback function when creating a weak reference.

EXAMPLES:

```
sage: from sage.structure.coerce_dict import MonoDict
sage: class A: pass
sage: a = A()
sage: M = MonoDict()
sage: M[a] = 1
sage: len(M)
1
sage: del a
sage: import gc
sage: n = gc.collect()
sage: len(M)     # indirect doctest
0
```

```
>>> from sage.all import *
>>> from sage.structure.coerce_dict import MonoDict
>>> class A: pass
>>> a = A()
>>> M = MonoDict()
>>> M[a] = Integer(1)
>>> len(M)
1
>>> del a
>>> import gc
>>> n = gc.collect()
>>> len(M)     # indirect doctest
0
```

AUTHOR:

- Simon King (2012-01)
- Nils Bruin (2013-11)

class sage.structure.coerce_dict.**ObjectWrapper**

Bases: object

A simple fast wrapper around a Python object. This is like a 1-element tuple except that it does not keep a reference to the wrapped object.

class sage.structure.coerce_dict.**TripleDict**

Bases: object

This is a hashtable specifically designed for (read) speed in the coercion model.

It differs from a python dict in the following important ways:

- All keys must be sequence of exactly three elements. All sequence types (tuple, list, etc.) map to the same item.
- Any of the three key components that support weak-refs are stored via a weakref. If any of these components gets garbage collected then the entire entry is removed. In that sense, this structure behaves like a nested WeakKeyDictionary.
- Comparison is done using the 'is' rather than '==' operator.

There are special cdef set/get methods for faster access. It is bare-bones in the sense that not all dictionary methods are implemented.

INPUT:

- data – (optional) iterable defining initial data, as dict or iterable of (key, value) pairs
- weak_values – boolean (default: False); if it is True, weak references to the values in this dictionary will be used, when possible

IMPLEMENTATION:

It is implemented as a hash table with open addressing, similar to python's dict.

EXAMPLES:

```
sage: from sage.structure.coerce_dict import TripleDict
sage: L = TripleDict()
sage: a = 'a'; b = 'b'; c = 'c'
```

(continues on next page)

(continued from previous page)

```
sage: L[a,b,c] = 1
sage: L[a,b,c]
1
sage: L[c,b,a] = -1
sage: sorted(L.items())
[(('a', 'b', 'c'), 1), (('c', 'b', 'a'), -1)]
sage: del L[a,b,c]
sage: list(L.items())
[(('c', 'b', 'a'), -1)]
sage: len(L)
1
sage: for i in range(1000):
....:     L[i,i,i] = i
sage: len(L)
1001
sage: L = TripleDict(L)
sage: L[c,b,a]
-1
sage: L[a,b,c]
Traceback (most recent call last):
...
KeyError: ('a', 'b', 'c')
sage: L[a]
Traceback (most recent call last):
...
KeyError: 'a'
sage: L[a] = 1
Traceback (most recent call last):
...
KeyError: 'a'
```

```
>>> from sage.all import *
>>> from sage.structure.coerce_dict import TripleDict
>>> L = TripleDict()
>>> a = 'a'; b = 'b'; c = 'c'
>>> L[a,b,c] = Integer(1)
>>> L[a,b,c]
1
>>> L[c,b,a] = -Integer(1)
>>> sorted(L.items())
[(('a', 'b', 'c'), 1), (('c', 'b', 'a'), -1)]
>>> del L[a,b,c]
>>> list(L.items())
[(('c', 'b', 'a'), -1)]
>>> len(L)
1
>>> for i in range(Integer(1000)):
...     L[i,i,i] = i
>>> len(L)
1001
>>> L = TripleDict(L)
>>> L[c,b,a]
```

(continues on next page)

(continued from previous page)

```
-1
>>> L[a,b,c]
Traceback (most recent call last):
...
KeyError: ('a', 'b', 'c')
>>> L[a]
Traceback (most recent call last):
...
KeyError: 'a'
>>> L[a] = Integer(1)
Traceback (most recent call last):
...
KeyError: 'a'
```

AUTHORS:

- Robert Bradshaw, 2007-08
- Simon King, 2012-01
- Nils Bruin, 2012-08
- Simon King, 2013-02
- Nils Bruin, 2013-11

copy()

Return a copy of this *TripleDict* as Python dict.

EXAMPLES:

```
sage: from sage.structure.coerce_dict import TripleDict
sage: L = TripleDict()
sage: L[1,2,3] = 42
sage: L.copy()
{(1, 2, 3): 42}
```

```
>>> from sage.all import *
>>> from sage.structure.coerce_dict import TripleDict
>>> L = TripleDict()
>>> L[Integer(1),Integer(2),Integer(3)] = Integer(42)
>>> L.copy()
{(1, 2, 3): 42}
```

items()

Iterate over the `(key, value)` pairs of this *TripleDict*.

EXAMPLES:

```
sage: from sage.structure.coerce_dict import TripleDict
sage: L = TripleDict()
sage: L[1,2,3] = None
sage: L.items()
<...generator object at ...>
sage: list(L.items())
[((1, 2, 3), None)]
```

```
>>> from sage.all import *
>>> from sage.structure.coerce_dict import TripleDict
>>> L = TripleDict()
>>> L[Integer(1),Integer(2),Integer(3)] = None
>>> L.items()
<...generator object at ...>
>>> list(L.items())
[((1, 2, 3), None)]
```

class sage.structure.coerce_dict.**TripleDictEraser**

Bases: `object`

Erases items from a *TripleDict* when a weak reference becomes invalid.

This is of internal use only. Instances of this class will be passed as a callback function when creating a weak reference.

EXAMPLES:

```
sage: from sage.structure.coerce_dict import TripleDict
sage: class A: pass
sage: a = A()
sage: T = TripleDict()
sage: T[a,ZZ,None] = 1
sage: T[ZZ,a,1] = 2
sage: T[a,a,ZZ] = 3
sage: len(T)
3
sage: del a
sage: import gc
sage: n = gc.collect()
sage: len(T) # indirect doctest
0
```

```
>>> from sage.all import *
>>> from sage.structure.coerce_dict import TripleDict
>>> class A: pass
>>> a = A()
>>> T = TripleDict()
>>> T[a,ZZ,None] = Integer(1)
>>> T[ZZ,a,Integer(1)] = Integer(2)
>>> T[a,a,ZZ] = Integer(3)
>>> len(T)
3
>>> del a
>>> import gc
>>> n = gc.collect()
>>> len(T) # indirect doctest
0
```

AUTHOR:

- Simon King (2012-01)
- Nils Bruin (2013-11)

5.7 Exceptions raised by the coercion model

exception sage.structure.coerce_exceptions.**CoercionException**

Bases: TypeError

This is the baseclass of exceptions that the coercion model raises when trying to discover coercions. We do not use standard Python exceptions to avoid inadvertently catching and suppressing real errors.

Usually one raises this to indicate the attempted action is not implemented/appropriate, but if there are other things to try not to immediately abort to the user.

CHAPTER

SIX

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

INDEX

N

O

P

Q

R

S

T

V