May 16, 2025 – Typing Summit @ PyCon US 2025

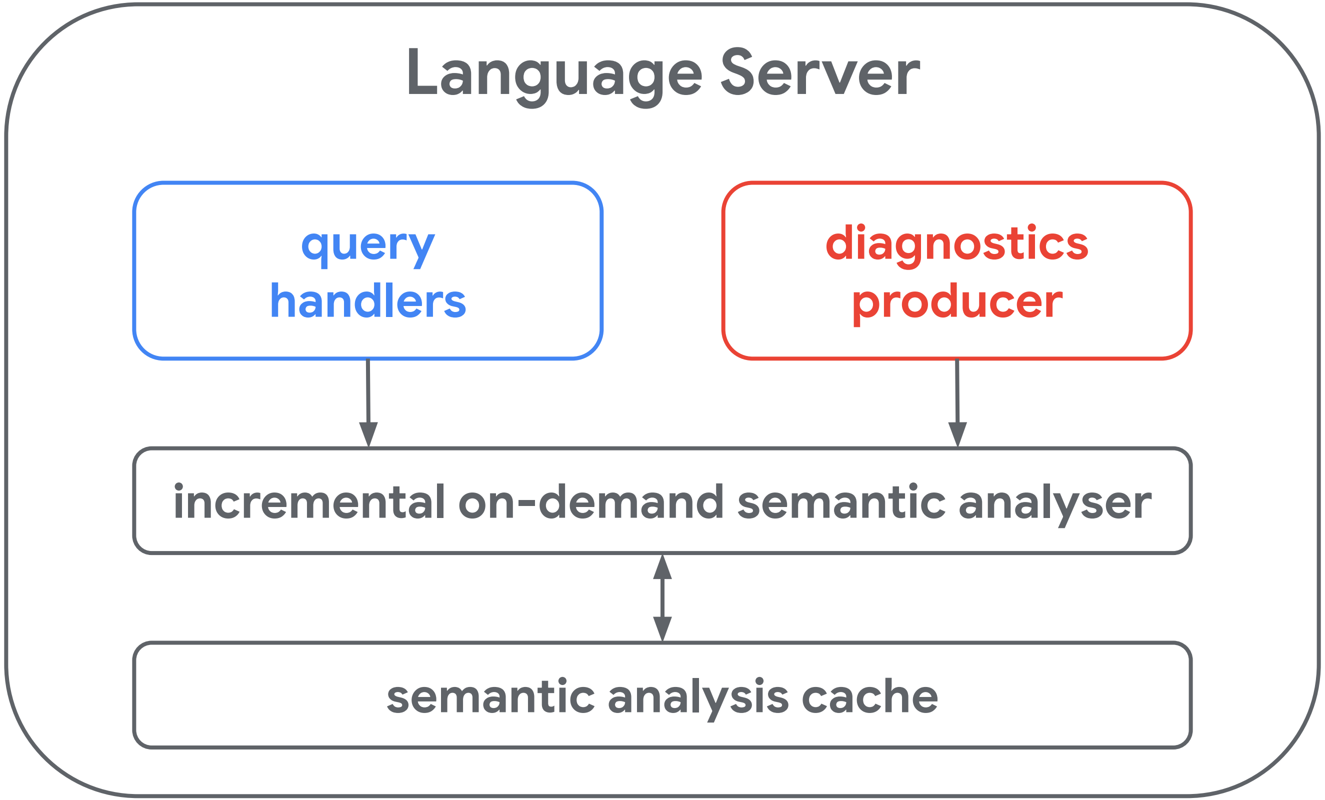
Linear Time Variance Inference for PEP 695

Martin Huschenbett

Python Team @ Google

drmh@google.com

Context



Language Server
query handlers
diagnostics producer
incremental on-demand semantic analyser
semantic analysis cache

Warm-up

Question: What was variance again?

Assumption:

`Snake <: Animal`

Invariance:

`list[Snake] ⊥ list[Animal]`

Covariance:

`Sequence[Snake] <: Sequence[Animal]`

Contravariance:

`Callable[[Animal], bool] <: Callable[[Snake], bool]`

PEP 695 and generic classes

```python
# Python 3.11 and before

X = TypeVar("X", bound=Animal,
        covariant=True)

class Cage(Generic[X]):
    animal: Final[X]

    def __init__(self, animal: X):
        self.animal = animal

    def open(self) -> X:
        return self.animal
```

```python
# Python 3.12 and after




# How to make Cage covariant?
class Cage[X: Animal]:
    animal: Final[X]

    def __init__(self, animal: X):
        self.animal = animal

    def open(self) -> X:
        return self.animal
```

PEP 695 and generic classes

```python
# Python 3.11 and before

X = TypeVar("X", bound=Animal,
        covariant=True)

class Cage(Generic[X]):
    animal: Final[X]

    def __init__(self, animal: X):
        self.animal = animal

    def open(self) -> X:
        return self.animal
```

```python
# Python 3.12 and after



# How to make Cage covariant?
class Cage[X: Animal]:
    animal: Final[X]

    def __init__(self, animal: X):
        self.animal = animal

    def open(self) -> X:
        return self.animal
```

VARIANCE INFERENCE!

Variance inference à la PEP 695

Definition: Let C be a generic class.

1. C is <u>covariant</u> if `S <: T` implies `C[S] <: C[T]` for all types `S` and `T`.
2. C is <u>contravariant</u> if `S <: T` implies `C[T] <: C[S]` for all types `S` and `T`.

Algorithm: Reduce to subtyping using parametricity and `X <: object`.

1. Return that C is <u>covariant</u> if *structurally* `C[X] <: C[object]`.
2. Return that C is <u>contravariant</u> if *structurally* `C[object] <: C[X]`.

```
class Cage[X](Protocol):
    animal: Final[X]
    def __init__(self, animal: X): ...
    def open(self) -> X: ...
```

WHAT ABOUT (MUTUALLY) RECURSIVE CLASSES?

Mutually recursive classes

```
class C[X]:
    def f(self) -> D[X]:
        ...
    def g(self, x: X) -> None:
        ...


class D[Y]:
    def h(self) -> C[Y]:
        ...


def cast(c: C[bool]) -> C[int]:
    return c  # Check this!
```

```
IsSubtype(C[bool], C[int])
├─IsSubtype(bool, int)──return True
├─IsCovariant(C)
│ ├─IsStructuralSubtype(C[X], C[object])
│ │ ├─IsSubtype(D[X], D[object])
│ │ │ ├─IsSubtype(X, object)──return True
│ │ │ ├─IsCovariant(D)
│ │ │ │ ├─IsStructuralSubtype(D[Y], D[object])
│ │ │ │ │ ├─IsSubtype(C[Y], C[object])
│ │ │ │ │ │ ├─IsSubtype(Y, object)──return True
│ │ │ │ │ │ ├─IsCovariant(C)
│ │ │ │ │ │ │ ├─LOOP DETECTED: USE COINDUCTION!
│ │ │ │ │ │ │ └─return True
│ │ │ │ │ │ └─return True
│ │ │ │ │ └─return True
│ │ │ │ └─return True
│ │ │ └─return True
│ │ ├─IsSubtype(object, X)──return False
│ │ └─return False
│ └─return False
├─IsSubtype(int, bool)──return False
└─return False
```

Mutually recursive classes

```
class C[X]:
    def f(self) -> D[X]:
        ...
    def g(self, x: X) -> None:
        ...


class D[Y]:
    def h(self) -> C[Y]:
        ...


def cast(c: C[bool]) -> C[int]:
    return c  # Check this!
```

```
IsSubtype(C[bool], C[int])
├─IsSubtype(bool, int)──return True
├─IsCovariant(C)
│ ├─IsStructuralSubtype(C[X], C[object])
│ │ ├─IsSubtype(D[X], D[object])
│ │ │ ├─IsSubtype(X, object)──return True
│ │ │ ├─IsCovariant(D)
│ │ │ │ ├─IsStructuralSubtype(D[Y], D[object])
│ │ │ │ │ ├─IsSubtype(C[Y], C[object])
│ │ │ │ │ │ ├─IsSubtype(Y, object)──return True
│ │ │ │ │ │ ├─IsCovariant(C)
│ │ │ │ │ │ │ ├─LOOP DETECTED: USE COINDUCTION!
│ │ │ │ │ │ │ └─return True
│ │ │ │ │ │ └─return True
│ │ │ │ │ └─return True
│ │ │ │ └─return True
│ │ │ └─return True
│ │ ├─IsSubtype(object, X)──return False
│ │ └─return False
│ └─return False
├─IsSubtype(int, bool)──return False
└─return False
```

Mutually recursive classes

```
class C[X]:
    def f(self) -> D[X]:
        ...
    def g(self, x: X) -> None:
        ...


class D[Y]:
    def h(self) -> C[Y]:
        ...


def cast(c: C[bool]) -> C[int]:
    return c  # Check this!
```

```
IsSubtype(C[bool], C[int])
├─IsSubtype(bool, int)──return True
├─IsCovariant(C)
│ ├─IsStructuralSubtype(C[X], C[object])
│ │ ├─IsSubtype(D[X], D[object])
│ │ │ ├─IsSubtype(X, object)──return True
│ │ │ ├─IsCovariant(D)
│ │ │ │ ├─IsStructuralSubtype(D[Y], D[object])
│ │ │ │ │ ├─IsSubtype(C[Y], C[object])
│ │ │ │ │ │ ├─IsSubtype(Y, object)──return True
│ │ │ │ │ │ ├─IsCovariant(C)
│ │ │ │ │ │ │ ├─LOOP DETECTED: USE COINDUCTION!
│ │ │ │ │ │ │ └─return True
│ │ │ │ │ │ └─return True
│ │ │ │ │ └─return True
│ │ │ │ └─return True
│ │ │ └─return True
│ │ ├─IsSubtype(object, X)──return False
│ │ └─return False
│ └─return False
├─IsSubtype(int, bool)──return False
└─return False
```

Mutually recursive classes

```python
class C[X]:
    def f(self) -> D[X]:
        ...
    def g(self, x: X) -> None:
        ...


class D[Y]:
    def h(self) -> C[Y]:
        ...


def cast(c: C[bool]) -> C[int]:
    return c  # Check this!
```

```
IsSubtype(C[bool], C[int])
├─IsSubtype(bool, int)──return True
├─IsCovariant(C)
│ ├─IsStructuralSubtype(C[X], C[object])
│ │ ├─IsSubtype(D[X], D[object])
│ │ │ ├─IsSubtype(X, object)──return True
│ │ │ ├─IsCovariant(D)
│ │ │ │ ├─IsStructuralSubtype(D[Y], D[object])
│ │ │ │ │ ├─IsSubtype(C[Y], C[object])
│ │ │ │ │ │ ├─IsSubtype(Y, object)──return True
│ │ │ │ │ │ ├─IsCovariant(C)
│ │ │ │ │ │ │ ├─LOOP DETECTED: USE COINDUCTION!
│ │ │ │ │ │ │ └─return True
│ │ │ │ │ │ └─return True
│ │ │ │ │ └─return True
│ │ │ │ └─return True
│ │ │ └─return True
│ │ ├─IsSubtype(object, X)──return False
│ │ └─return False
│ └─return False
├─IsSubtype(int, bool)──return False
└─return False
```

Mutually recursive classes

```python
class C[X]:
    def f(self) -> D[X]:
        ...
    def g(self, x: X) -> None:
        ...


class D[Y]:
    def h(self) -> C[Y]:
        ...


def cast(c: C[bool]) -> C[int]:
    return c  # Check this!
```

```
IsSubtype(C[bool], C[int])
├─IsSubtype(bool, int)──return True
├─IsCovariant(C)
│ ├─IsStructuralSubtype(C[X], C[object])
│ │ ├─IsSubtype(D[X], D[object])
│ │ │ ├─IsSubtype(X, object)──return True
│ │ │ ├─IsCovariant(D)
│ │ │ │ ├─IsStructuralSubtype(D[Y], D[object])
│ │ │ │ │ ├─IsSubtype(C[Y], C[object])
│ │ │ │ │ │ ├─IsSubtype(Y, object)──return True
│ │ │ │ │ │ ├─IsCovariant(C)
│ │ │ │ │ │ │ ├─LOOP DETECTED: USE COINDUCTION!
│ │ │ │ │ │ │ └─return True
│ │ │ │ │ │ └─return True
│ │ │ │ │ └─return True
│ │ │ │ └─return True
│ │ │ └─return True
│ │ ├─IsSubtype(object, X)──return False
│ │ └─return False
│ └─return False
├─IsSubtype(int, bool)──return False
└─return False
```

Mutually recursive classes

```
class C[X]:
    def f(self) -> D[X]:
        ...
    def g(self, x: X) -> None:
        ...


class D[Y]:
    def h(self) -> C[Y]:
        ...


def cast(c: C[bool]) -> C[int]:
    return c  # Check this!
    ~~~~~~~~~
      Not C[bool] <: C[int].
```

```
IsSubtype(C[bool], C[int])
├─IsSubtype(bool, int)──return True
├─IsCovariant(C)
│ ├─IsStructuralSubtype(C[X], C[object])
│ │ ├─IsSubtype(D[X], D[object])
│ │ │ ├─IsSubtype(X, object)──return True
│ │ │ ├─IsCovariant(D)
│ │ │ │ ├─IsStructuralSubtype(D[Y], D[object])
│ │ │ │ │ ├─IsSubtype(C[Y], C[object])
│ │ │ │ │ │ ├─IsSubtype(Y, object)──return True
│ │ │ │ │ │ ├─IsCovariant(C)
│ │ │ │ │ │ │ ├─LOOP DETECTED: USE COINDUCTION!
│ │ │ │ │ │ │ └─return True
│ │ │ │ │ │ └─return True
│ │ │ │ │ └─return True
│ │ │ │ └─return True
│ │ │ └─return True
│ │ ├─IsSubtype(object, X)──return False
│ │ └─return False
│ └─return False
├─IsSubtype(int, bool)──return False
└─return False
```

Mutually recursive classes

```
class C[X]:
    def f(self) -> D[X]:
        ...
    def g(self, x: X) -> None:
        ...


class D[Y]:
    def h(self) -> C[Y]:
        ...


def cast(c: C[bool]) -> C[int]:
    return c  # Check this!
    ~~~~~~~~
      Not C[bool] <: C[int].
```

```
IsSubtype(C[bool], C[int])
├─IsSubtype(bool, int)──return True
├─IsCovariant(C)
│ ├─IsStructuralSubtype(C[X], C[object])
│ │ ├─IsSubtype(D[X], D[object])
│ │ │ ├─IsSubtype(X, object)──return True
│ │ │ ├─IsCovariant(D)
│ │ │ │ ├─IsStructuralSubtype(D[Y], D[object])
│ │ │ │ │ ├─IsSubtype(C[Y], C[object])
│ │ │ │ │ │ ├─IsSubtype(Y, object)──return True
│ │ │ │ │ │ ├─IsCovariant(C)
│ │ │ │ │ │ │ ├─LOOP DETECTED: USE COINDUCTION!
│ │ │ │ │ │ │ └─return True
│ │ │ │ │ │ └─return True
│ │ │ │ │ └─return True
│ │ │ │ └─return True
│ │ │ └─return True
│ │ ├─IsSubtype(object, X)──return False
│ │ └─return False
│ └─return False
├─IsSubtype(int, bool)──return False
└─return False
```

QUADRATIC RUNTIME!

Linear time algorithm

Input: Signatures of classes `C1, ..., Cn` that are closed under dependencies.

1. Traverse the signatures of `C1, ..., Cn` and emit boolean equations.
2. Solve the system of boolean equations.
3. Read variances off assignments to certain boolean variables.

Output: *Optimal* variances for classes `C1, ..., Cn`.

Complexity:

1. Time for generating all boolean equations = O(input size)
2. Time for solving equations = O(size of equation system) = **O(input size)**

Generalised variance

Definition: Let T be a type and X a type variable.

1. T is <u>covariant in X</u> if $U <: V$ implies $T[X \mapsto U] <: T[X \mapsto V]$ for all U and V.
2. T is <u>contravariant in X</u> if $U <: V$ implies $T[X \mapsto V] <: T[X \mapsto U]$ for all U and V.

Observations:

1. If X does not appear in T, T is covariant and contravariant in X.
2. Let C be a generic class. If C is covariant/contravariant, C[X] is covariant/contravariant in X.
3. X is covariant in X but not contravariant in X.

Mutually recursive classes revisited

```
class C[X]:
    def f(self) -> D[X]: ...
    def g(self, x: X) -> None: ...

class D[Y]:
    def h(self) -> C[Y]: ...
```

Equations:

$C^{+} = F^{+X} \wedge G^{+X}$

$C^{-} = F^{-X} \wedge G^{-X}$

$D^{+} = H^{+Y}$

$D^{-} = H^{-Y}$

$F^{+X} = D^{+}$

$F^{-X} = D^{-}$

$G^{+X} = X^{-X} \wedge N^{+X}$

$G^{-X} = X^{+X} \wedge N^{-X}$

$H^{+Y} = C^{+}$

$H^{-Y} = C^{-}$

$X^{+X} = \mathtt{true}$

$X^{-X} = \mathtt{false}$

$N^{+X} = \mathtt{true}$

$N^{-X} = \mathtt{true}$

Boolean variables:

C^{+}/C^{-} ~ C is covariant/contravariant

D^{+}/D^{-} ~ D is covariant/contravariant

F^{+X}/F^{-X} ~ f's signature is covariant/contravariant in X

G^{+X}/G^{-X} ~ g's signature is covariant/contravariant in X

H^{+Y}/H^{-Y} ~ h's signature is covariant/contravariant in Y

X^{+X}/X^{-X} ~ X is covariant/contravariant in X

N^{+X}/N^{-X} ~ None is covariant/contravariant in X

Mutually recursive classes revisited

```
class C[X]:
    def f(self) -> D[X]: ...
    def g(self, x: X) -> None: ...

class D[Y]:
    def h(self) -> C[Y]: ...
```

Boolean variables:

C^+ / C^- ~ C is covariant/contravariant

D^+ / D^- ~ D is covariant/contravariant

F^{+X} / F^{-X} ~ f's signature is covariant/contravariant in X

G^{+X} / G^{-X} ~ g's signature is covariant/contravariant in X

H^{+Y} / H^{-Y} ~ h's signature is covariant/contravariant in Y

X^{+X} / X^{-X} ~ X is covariant/contravariant in X

N^{+X} / N^{-X} ~ None is covariant/contravariant in X

Equations:

$C^+ = F^{+X} \wedge G^{+X}$

$C^- = F^{-X} \wedge G^{-X}$

$D^+ = H^{+Y}$

$D^- = H^{-Y}$

$F^{+X} = D^+$

$F^{-X} = D^-$

$G^{+X} = X^{-X} \wedge N^{+X}$

$G^{-X} = X^{+X} \wedge N^{-X}$

$H^{+Y} = C^+$

$H^{-Y} = C^-$

$X^{+X} = \mathtt{true}$

$X^{-X} = \mathtt{false}$

$N^{+X} = \mathtt{true}$

$N^{-X} = \mathtt{true}$

Mutually recursive classes revisited

```
class C[X]:
    def f(self) -> D[X]: ...
    def g(self, x: X) -> None: ...

class D[Y]:
    def h(self) -> C[Y]: ...
```

Boolean variables:

C^+/C^- ~ C is covariant/contravariant

D^+/D^- ~ D is covariant/contravariant

F^{+X}/F^{-X} ~ f's signature is covariant/contravariant in X

G^{+X}/G^{-X} ~ g's signature is covariant/contravariant in X

H^{+Y}/H^{-Y} ~ h's signature is covariant/contravariant in Y

X^{+X}/X^{-X} ~ X is covariant/contravariant in X

N^{+X}/N^{-X} ~ None is covariant/contravariant in X

Equations:

$C^+ = F^{+X} \wedge G^{+X}$

$C^- = F^{-X} \wedge G^{-X}$

$D^+ = H^{+Y}$

$D^- = H^{-Y}$

$F^{+X} = D^+$

$F^{-X} = D^-$

$G^{+X} = X^{-X} \wedge N^{+X}$

$G^{-X} = X^{+X} \wedge N^{-X}$

$H^{+Y} = C^+$

$H^{-Y} = C^-$

$X^{+X} = \mathtt{true}$

$X^{-X} = \mathtt{false}$

$N^{+X} = \mathtt{true}$

$N^{-X} = \mathtt{true}$

Mutually recursive classes revisited

```
class C[X]:
    def f(self) -> D[X]: ...
    def g(self, x: X) -> None: ...

class D[Y]:
    def h(self) -> C[Y]: ...
```

Boolean variables:

C^{+}/C^{-} ~ C is covariant/contravariant

D^{+}/D^{-} ~ D is covariant/contravariant

F^{+X}/F^{-X} ~ f's signature is covariant/contravariant in X

G^{+X}/G^{-X} ~ g's signature is covariant/contravariant in X

H^{+Y}/H^{-Y} ~ h's signature is covariant/contravariant in Y

X^{+X}/X^{-X} ~ X is covariant/contravariant in X

N^{+X}/N^{-X} ~ None is covariant/contravariant in X

Equations:

$C^{+} = F^{+X} \wedge G^{+X}$

$C^{-} = F^{-X} \wedge G^{-X}$

$D^{+} = H^{+Y}$

$D^{-} = H^{-Y}$

$F^{+X} = D^{+}$

$F^{-X} = D^{-}$

$G^{+X} = X^{-X} \wedge N^{+X} = \text{false}$

$G^{-X} = X^{+X} \wedge N^{-X} = \text{true}$

$H^{+Y} = C^{+}$

$H^{-Y} = C^{-}$

$X^{+X} = \text{true}$

$X^{-X} = \text{false}$

$N^{+X} = \text{true}$

$N^{-X} = \text{true}$

Mutually recursive classes revisited

```
class C[X]:
    def f(self) -> D[X]: ...
    def g(self, x: X) -> None: ...

class D[Y]:
    def h(self) -> C[Y]: ...
```

Boolean variables:

C^{+}/C^{-} ~ C is covariant/contravariant

D^{+}/D^{-} ~ D is covariant/contravariant

F^{+X}/F^{-X} ~ f's signature is covariant/contravariant in X

G^{+X}/G^{-X} ~ g's signature is covariant/contravariant in X

H^{+Y}/H^{-Y} ~ h's signature is covariant/contravariant in Y

X^{+X}/X^{-X} ~ X is covariant/contravariant in X

N^{+X}/N^{-X} ~ None is covariant/contravariant in X

Equations:

$C^{+} = F^{+X} \wedge G^{+X} = \texttt{false}$

$C^{-} = F^{-X} \wedge G^{-X} = F^{-X}$

$D^{+} = H^{+Y}$

$D^{-} = H^{-Y}$

$F^{+X} = D^{+}$

$F^{-X} = D^{-}$

$G^{+X} = X^{-X} \wedge N^{+X} = \texttt{false}$

$G^{-X} = X^{+X} \wedge N^{-X} = \texttt{true}$

$H^{+Y} = C^{+}$

$H^{-Y} = C^{-}$

$X^{+X} = \texttt{true}$

$X^{-X} = \texttt{false}$

$N^{+X} = \texttt{true}$

$N^{-X} = \texttt{true}$

Mutually recursive classes revisited

```
class C[X]:
    def f(self) -> D[X]: ...
    def g(self, x: X) -> None: ...

class D[Y]:
    def h(self) -> C[Y]: ...
```

Boolean variables:

C^+ / C^- ~ C is covariant/contravariant

D^+ / D^- ~ D is covariant/contravariant

F^{+X} / F^{-X} ~ f's signature is covariant/contravariant in X

G^{+X} / G^{-X} ~ g's signature is covariant/contravariant in X

H^{+Y} / H^{-Y} ~ h's signature is covariant/contravariant in Y

X^{+X} / X^{-X} ~ X is covariant/contravariant in X

N^{+X} / N^{-X} ~ None is covariant/contravariant in X

Equations:

$C^+ = F^{+X} \wedge G^{+X} = \mathtt{false}$

$C^- = F^{-X} \wedge G^{-X} = F^{-X}$

$D^+ = H^{+Y} = \mathtt{false}$

$D^- = H^{-Y}$

$F^{+X} = D^+ = \mathtt{false}$

$F^{-X} = D^-$

$G^{+X} = X^{-X} \wedge N^{+X} = \mathtt{false}$

$G^{-X} = X^{+X} \wedge N^{-X} = \mathtt{true}$

$H^{+Y} = C^+ = \mathtt{false}$

$H^{-Y} = C^-$

$X^{+X} = \mathtt{true}$

$X^{-X} = \mathtt{false}$

$N^{+X} = \mathtt{true}$

$N^{-X} = \mathtt{true}$

Mutually recursive classes revisited

```
class C[X]:
    def f(self) -> D[X]: ...
    def g(self, x: X) -> None: ...

class D[Y]:
    def h(self) -> C[Y]: ...
```

Boolean variables:

C^{+}/C^{-} ~ C is covariant/contravariant

D^{+}/D^{-} ~ D is covariant/contravariant

F^{+X}/F^{-X} ~ f's signature is covariant/contravariant in X

G^{+X}/G^{-X} ~ g's signature is covariant/contravariant in X

H^{+Y}/H^{-Y} ~ h's signature is covariant/contravariant in Y

X^{+X}/X^{-X} ~ X is covariant/contravariant in X

N^{+X}/N^{-X} ~ None is covariant/contravariant in X

Equations:

$C^{+} = F^{+X} \wedge G^{+X} = \text{false}$

$C^{-} = F^{-X} \wedge G^{-X} = \text{true}$

$D^{+} = H^{+Y} = \text{false}$

$D^{-} = H^{-Y} = \text{true}$

$F^{+X} = D^{+} = \text{false}$

$F^{-X} = D^{-} = \text{true}$

$G^{+X} = X^{-X} \wedge N^{+X} = \text{false}$

$G^{-X} = X^{+X} \wedge N^{-X} = \text{true}$

$H^{+Y} = C^{+} = \text{false}$

$H^{-Y} = C^{-} = \text{true}$

$X^{+X} = \text{true}$

$X^{-X} = \text{false}$

$N^{+X} = \text{true}$

$N^{-X} = \text{true}$

Mutually recursive classes revisited

```
class C[X]:
    def f(self) -> D[X]: ...
    def g(self, x: X) -> None: ...
```

CONTRAVARIANT!

```
class D[Y]:
    def h(self) -> C[Y]: ...
```

CONTRAVARIANT!

Boolean variables:

C^{+}/C^{-} ~ C is covariant/contravariant

D^{+}/D^{-} ~ D is covariant/contravariant

F^{+X}/F^{-X} ~ f's signature is covariant/contravariant in X

G^{+X}/G^{-X} ~ g's signature is covariant/contravariant in X

H^{+Y}/H^{-Y} ~ h's signature is covariant/contravariant in Y

X^{+X}/X^{-X} ~ X is covariant/contravariant in X

N^{+X}/N^{-X} ~ None is covariant/contravariant in X

Equations:

$C^{+} = F^{+X} \wedge G^{+X} = \text{false}$

$C^{-} = F^{-X} \wedge G^{-X} = \text{true}$

$D^{+} = H^{+Y} = \text{false}$

$D^{-} = H^{-Y} = \text{true}$

$F^{+X} = D^{+} = \text{false}$

$F^{-X} = D^{-} = \text{true}$

$G^{+X} = X^{-X} \wedge N^{+X} = \text{false}$

$G^{-X} = X^{+X} \wedge N^{-X} = \text{true}$

$H^{+Y} = C^{+} = \text{false}$

$H^{-Y} = C^{-} = \text{true}$

$X^{+X} = \text{true}$

$X^{-X} = \text{false}$

$N^{+X} = \text{true}$

$N^{-X} = \text{true}$

Equations for classes

```
class C[X](B):
    a: Final[S]
    b: T
    def f(self, u: U, v: V) -> W: ...
```

Equations:

$$C^{+} = B^{+X} \wedge S^{+X} \wedge T^{+X} \wedge T^{-X} \wedge F^{+X}$$

$$C^{-} = B^{-X} \wedge S^{-X} \wedge T^{+X} \wedge T^{-X} \wedge F^{-X}$$

$$F^{+X} = U^{-X} \wedge V^{-X} \wedge W^{+X}$$

$$F^{-X} = U^{+X} \wedge V^{+X} \wedge W^{-X}$$

Intuition for fields:

- `a: Final[S]` ≈ `def get_a(self) -> S`
- `b: T` ≈ `def get_b(self) -> T` + `def set_b(self, b: T) -> None`

Equations for builtins

$$(S \mid T)^{+X} = S^{+X} \wedge T^{+X}$$

$$(S \mid T)^{-X} = S^{-X} \wedge T^{-X}$$

$$\texttt{tuple}[S,\ T]^{+X} = S^{+X} \wedge T^{+X}$$

$$\texttt{tuple}[S,\ T]^{-X} = S^{-X} \wedge T^{-X}$$

$$\texttt{tuple}[T,\ \ldots]^{+X} = T^{+X}$$

$$\texttt{tuple}[T,\ \ldots]^{-X} = T^{-X}$$

$$\vdots$$

Equations for specialisations

Theorem: Let C be a generic class, T a type, and X a type variable. The type `C[T]` is covariant/contravariant in X if one of the following conditions is met:

1. C is covariant and T is covariant/contravariant in X,
2. C is contravariant and T is contravariant/covariant in X,
3. C is covariant and contravariant,
4. T is covariant and contravariant in X.

Equations:

$$C[T]^{+X} = (C^{+} \wedge T^{+X}) \vee (C^{-} \wedge T^{-X}) \vee (C^{+} \wedge C^{-}) \vee (T^{+X} \wedge T^{-X})$$

$$C[T]^{-X} = (C^{+} \wedge T^{-X}) \vee (C^{-} \wedge T^{+X}) \vee (C^{+} \wedge C^{-}) \vee (T^{+X} \wedge T^{-X})$$

Linear time algorithm

Input: Signatures of classes `C1, ..., Cn` that are closed under dependencies.

1. Traverse the signatures of `C1, ..., Cn` and emit boolean equations:
   a. **Incremental:** If `Ci`'s variances already known, emit $\mathtt{Ci}^{\pm}$ `= true/false`.
2. Solve the system of boolean equations:
   a. Propagate constants and simplify as long as possible.
   b. Set all unconstrained boolean variables to `true`: sound and optimal.
   c. Yields the *greatest fixed point* of the equation system (Knaster–Tarski).
3. Read variances off assignments to boolean variables $\mathtt{C1}^{\pm}$`, ..., `$\mathtt{Cn}^{\pm}$.

Output: *Optimal* variances for classes `C1, ..., Cn`.

Linear time algorithm

Input: Signatures of classes `C1, ..., Cn` that are closed under dependencies.

1. Traverse the signatures of `C1, ..., Cn` and emit boolean equations:
   a. **Incremental:** If `Ci`'s variances already known, emit $Ci^{\pm}$ `= true/false`.
2. Solve the system of boolean equations:
   a. Propagate constants and simplify as long as possible.
   b. Set all unconstrained boolean variables to `true`: sound and optimal.
   c. Yields the *greatest fixed point* of the equation system (Knaster–Tarski).
3. Read variances off assignments to boolean variables $C1^{\pm}, ..., Cn^{\pm}$.

Output: *Optimal* variances for classes `C1, ..., Cn`.

Linear time algorithm

Input: Signatures of classes `C1, ..., Cn` that are closed under dependencies.

1. Traverse the signatures of `C1, ..., Cn` and emit boolean equations:
   a. **Incremental:** If `Ci`'s variances already known, emit `Ci`$^{\pm}$ `= true/false`.
2. Solve the system of boolean equations:
   a. Propagate constants and simplify as long as possible.
   b. Set all unconstrained boolean variables to `true`: sound and optimal.
   c. Yields the *greatest fixed point* of the equation system (Knaster–Tarski).
3. Read variances off assignments to boolean variables `C1`$^{\pm}$`, ..., Cn`$^{\pm}$.

Output: *Optimal* variances for classes `C1, ..., Cn`.

Linear time algorithm

Input: Signatures of classes `C1, ..., Cn` that are closed under dependencies.

1. Traverse the signatures of `C1, ..., Cn` and emit boolean equations:
   a. **Incremental:** If `Ci`'s variances already known, emit $\texttt{Ci}^{\pm}$ `= true/false`.
2. Solve the system of boolean equations:
   a. Propagate constants and simplify as long as possible.
   b. Set all unconstrained boolean variables to `true`: sound and optimal.
   c. Yields the *greatest fixed point* of the equation system (Knaster–Tarski).
3. Read variances off assignments to boolean variables $\texttt{C1}^{\pm}$, ..., $\texttt{Cn}^{\pm}$.

Output: *Optimal* variances for classes `C1, ..., Cn`.

Open questions

1. How do we best integrate short circuiting techniques into this approach?
2. Can we use this approach to improve incrementality/caching for subtyping?
3. Can we support immutable data structures better?

```
class Pair[X]:
    fst: Final[X]
    snd: Final[X]
    def __init__(self, fst: X, snd: X):
        self.fst = fst; self.snd = snd
    def replace_fst(self, fst: X) -> Pair[X]:
        return Pair(fst, self.snd)
```

`Pair` is inferred as invariant but covariant would be safe too.

Open questions

1. How do we best integrate short circuiting techniques into this approach?
2. Can we use this approach to improve incrementality/caching for subtyping?
3. Can we support immutable data structures better?

```
class Pair[X]:
    fst: Final[X]
    snd: Final[X]
    def __init__(self, fst: X, snd: X):
        self.fst = fst; self.snd = snd
    def replace_fst(self, fst: X) -> Pair[X]:
        return Pair(fst, self.snd)
```

THANK YOU!

`Pair` is inferred as invariant but covariant would be safe too.

Appendix

Interesting read

[Taming the Wildcards: Combining Definition- and Use-Site Variance]

Example for quadratic runtime

```
class A₀[X]:
    def g(self, x: X) -> None: ...

# for i = 1, ..., n:
class Aᵢ[X]:
    def f(self) -> Aᵢ₊₁[X]: ...
    def g(self) -> Aᵢ₋₁[X]: ...

class Aₙ₊₁[X]:
    def f(self) -> X: ...
```

Queries:

$\mathrm{IsCovariant}(A_1), \ldots, \mathrm{IsCovariant}(A_n)$

Equations:

$$
\begin{aligned}
A_0 &= \text{false} \\
A_1 &= A_2 \land A_0 \\
A_2 &= A_3 \land A_1 \\
A_3 &= A_4 \land A_2 \\
&\vdots \\
A_i &= A_{i+1} \land A_{i-1} \\
&\vdots \\
A_n &= A_{n+1} \land A_{n-1} \\
A_{n+1} &= \text{true}
\end{aligned}
$$