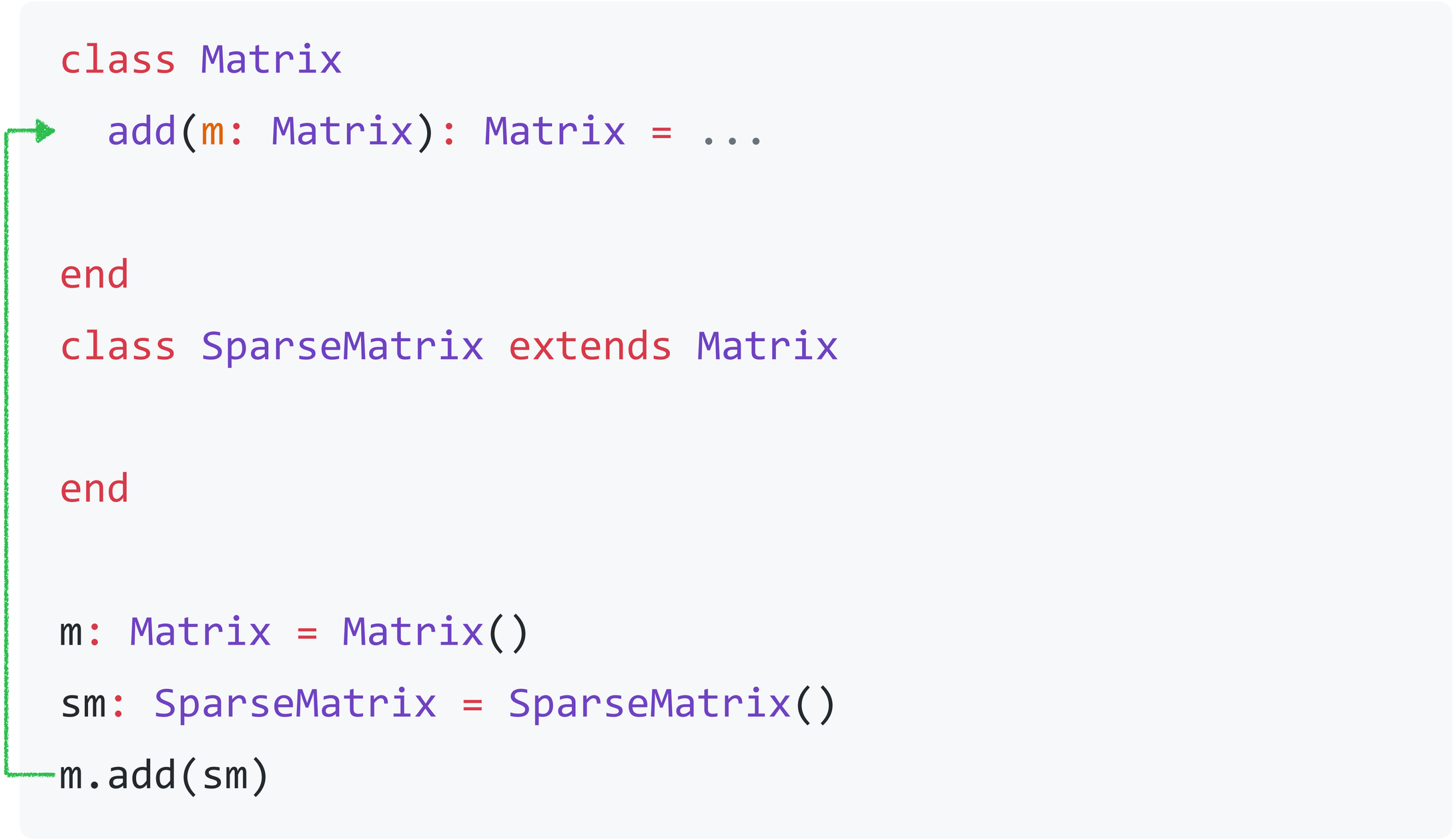
Polymorphic Symmetric Multiple Dispatch with Variance*¶

Gyunghee Park[§‡]
Jaemin Hong[§]
Guy L. Steele Jr.[‡]
Sukyoung Ryu[‡]

[§]KAIST, Republic of Korea
[‡]Oracle Labs, USA
*Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '19)
¶This work has received funding from National Research Foundation of Korea (NRF) (Grants NRF-2017R1A2B3012020 and 2017M3C4A7068177)

PLRG


class Matrix
add(m: Matrix): Matrix = ...
end
class SparseMatrix extends Matrix
end
m: Matrix = Matrix()
sm: SparseMatrix = SparseMatrix()
m.add(sm)

```
class Matrix
  add(m: Matrix): Matrix = ...
  add(m: SparseMatrix): Matrix = ...
end
class SparseMatrix extends Matrix


end

m: Matrix = Matrix()
sm: SparseMatrix = SparseMatrix()
m.add(sm)
```

Method overloading

Method dispatch

```
class Matrix
  add(m: Matrix): Matrix = ...
  add(m: SparseMatrix): Matrix = ...
end
class SparseMatrix extends Matrix
  add(m: Matrix): Matrix = ...
end

m: Matrix = Matrix()
sm: SparseMatrix = SparseMatrix()
sm.add(m)
```

Method overloading

Method dispatch

```
class Matrix
  add(m: Matrix): Matrix = ...
  add(m: SparseMatrix): Matrix = ...
end
class SparseMatrix extends Matrix
  add(m: Matrix): Matrix = ...
end

m: Matrix = Matrix()
sm: Matrix = SparseMatrix()
sm.add(m)
```

Method overloading

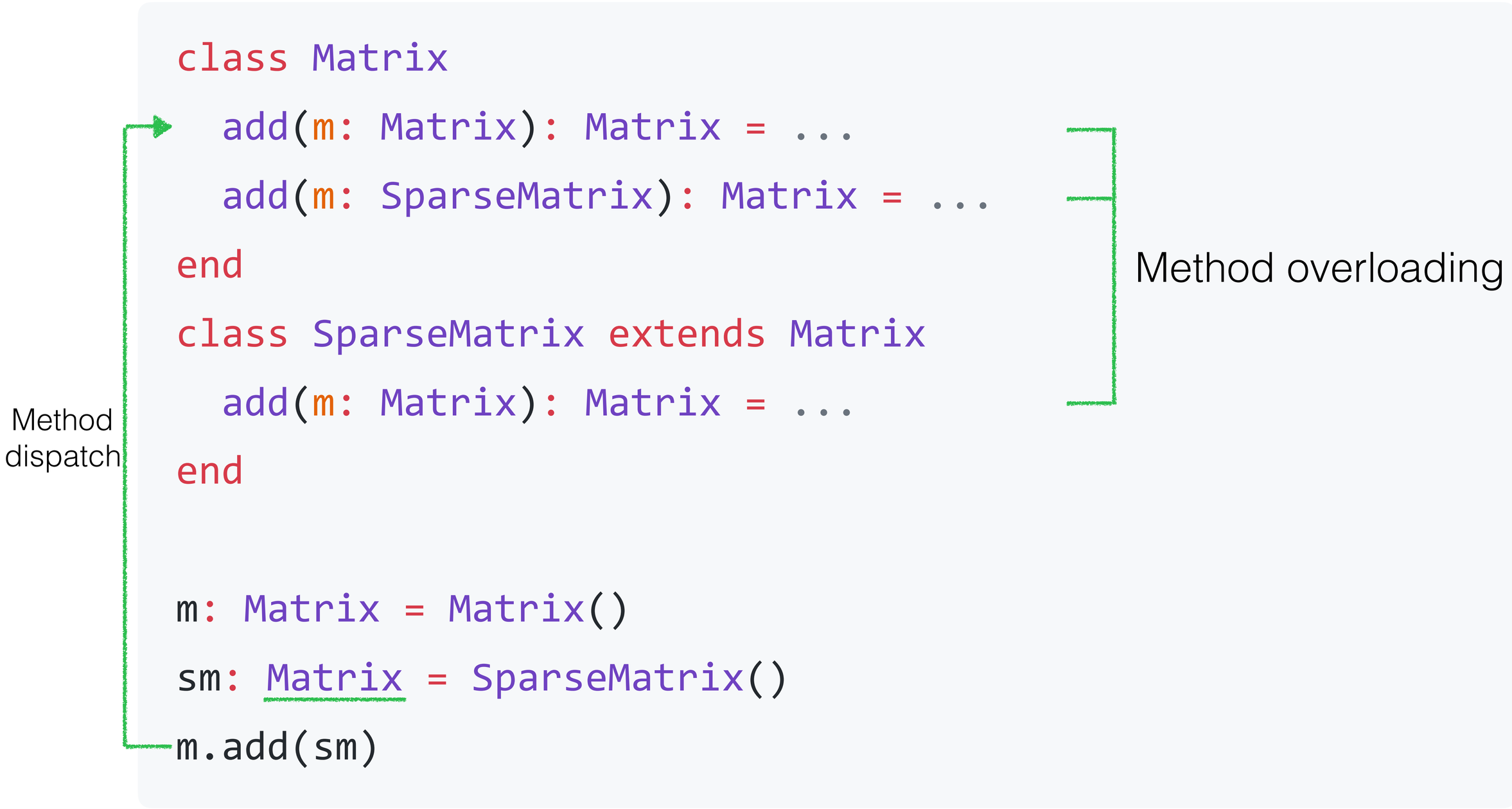
Method dispatch

```
class Matrix
  add(m: Matrix): Matrix = ...
  add(m: SparseMatrix): Matrix = ...
end
class SparseMatrix extends Matrix
  add(m: Matrix): Matrix = ...
d


m: Matrix = Matrix()
sm: Matrix = SparseMatrix()
sm.add(m)
```

Method overloading

Dynamic

Method dispatch


class Matrix
add(m: Matrix): Matrix = ...
add(m: SparseMatrix): Matrix = ...
end
class SparseMatrix extends Matrix
add(m: Matrix): Matrix = ...
end
m: Matrix = Matrix()
sm: Matrix = SparseMatrix()
m.add(sm)
Method dispatch
Method overloading

PLRG

```
class Matrix
  add(m: Matrix): Matrix = ...
  add(m: SparseMatrix): Matrix = ...
end
class SparseMatrix extends Matrix
  add(m: Matrix): Matrix = ...
end

m: Matrix = Matrix()
sm: Matrix = SparseMatrix()
m.add(sm)
```

Single

Method dispatch

Method overloading

```
class Matrix
  add(m: Matrix): Matrix = ...
  add(m: SparseMatrix): Matrix = ...
end
class SparseMatrix
  add(m: Matrix): M
end

m: Matrix = Matrix()
sm: Matrix = SparseMatrix()
m.add(sm)
```

Single

Method dispatch

Method overloading

python™

PLRG

```
class Matrix
  add(m: Matrix): Matrix = ...
  add(m: SparseMatrix): Matrix = ...
end
class SparseMatrix
  add(m: Matrix): M
end

m: Matrix = Matrix()
sm: Matrix = SparseMatrix()
m.add(sm)
```

Single

Method dispatch

Method overloading

python™

The binary method problem

PLRG

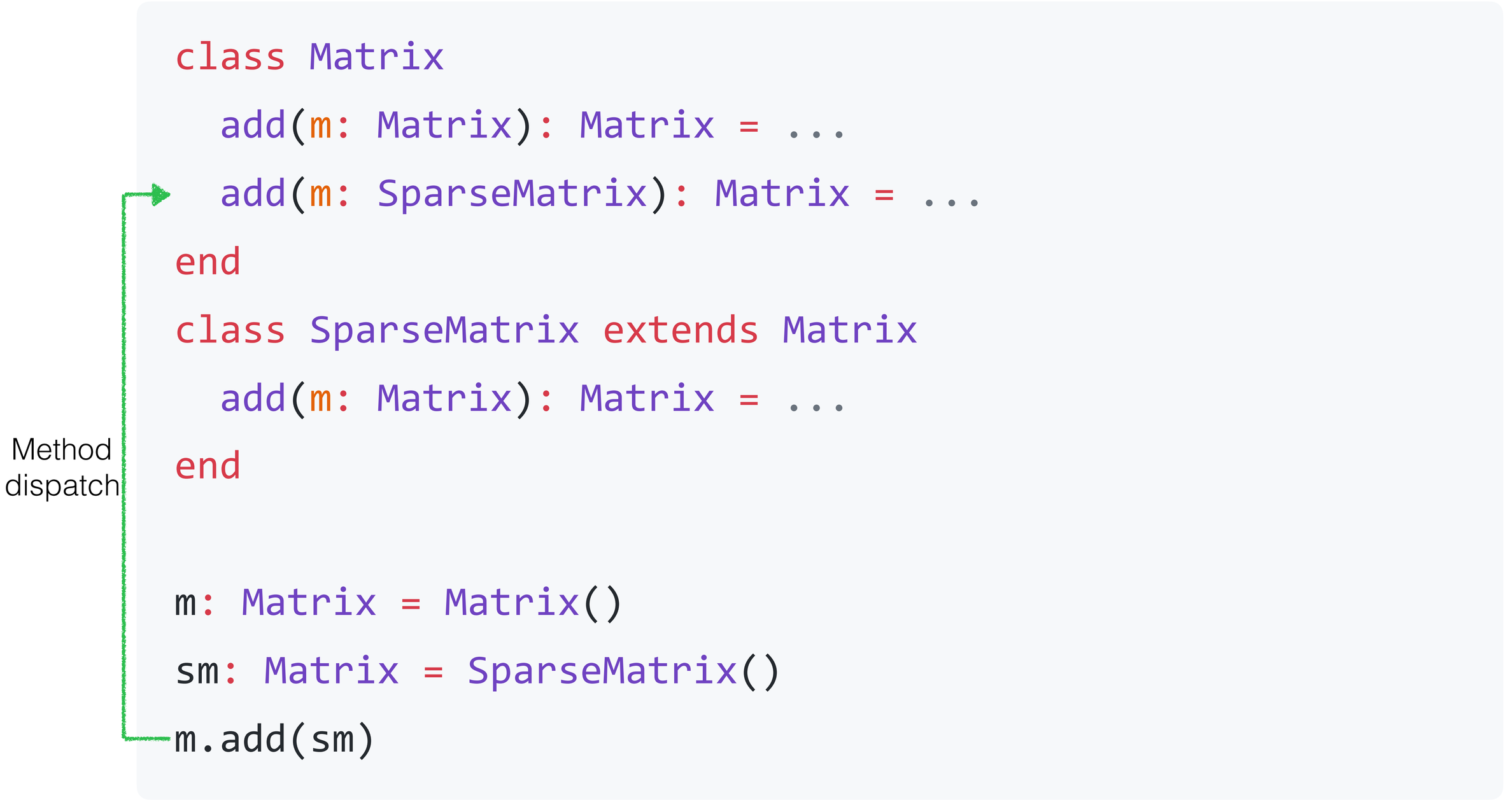
Chapter 30

Object Equality

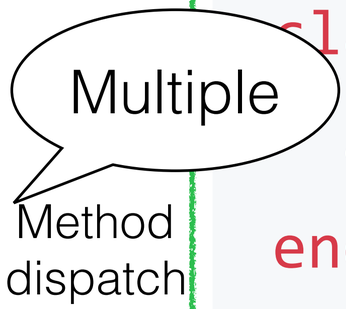
Comparing two values for equality is ubiquitous in programming. It is also more tricky than it looks at first glance. This chapter looks at object equality in detail and gives some recommendations to consider when you design your own equality tests.

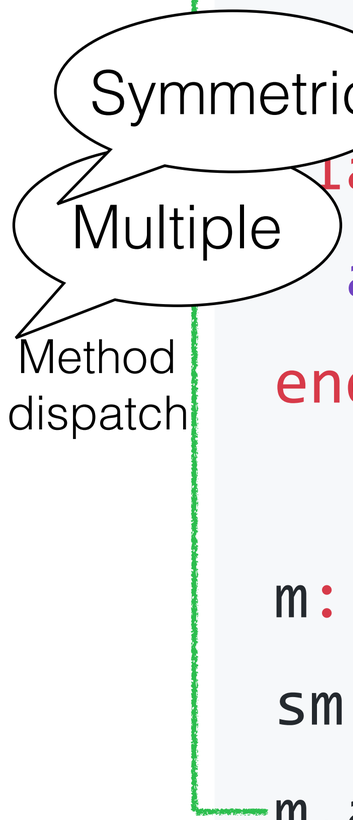
```
class Point(val x: Int, val y: Int) {
  override def equals(other: Any) = other match {




  }

}
class ColoredPoint(x: Int, y: Int, val color: Color.Value) extends Point(x, y) {
  override def equals(other: Any) = other match {




  }

}
```

```scala
class Point(val x: Int, val y: Int) {
  override def equals(other: Any) = other match {
    case that: Point =>
      (that canEqual this) && (this.x == that.x) && (this.y == that.y)
    case _ =>
      false
  }
  def canEqual(other: Any) = other.isInstanceOf[Point]
}
class ColoredPoint(x: Int, y: Int, val color: Color.Value) extends Point(x, y) {
  override def equals(other: Any) = other match {
    case that: ColoredPoint =>
      (that canEqual this) && super.equals(that) && this.color == that.color
    case _ =>
      false
  }
  override def canEqual(other: Any) = other.isInstanceOf[ColoredPoint]
}
```


class Matrix
add(m: Matrix): Matrix = ...
add(m: SparseMatrix): Matrix = ...
end
class SparseMatrix extends Matrix
add(m: Matrix): Matrix = ...
end
m: Matrix = Matrix()
sm: Matrix = SparseMatrix()
m.add(sm)
Method dispatch

```
class Matrix
  add(m: Matrix): Matrix = ...
  add(m: SparseMatrix): Matrix = ...
end
class SparseMatrix extends Matrix
  add(m: Matrix): Matrix = ...
end

m: Matrix = Matrix()
sm: Matrix = SparseMatrix()
m.add(sm)
```






```
class Matrix
  add(m: Matrix): Matrix = ...
  add(m: SparseMatrix): Matrix = ...

class SparseMatrix extends Matrix
  add(m: Matrix): Matrix = ...
end


m: Matrix = Matrix()
sm: Matrix = SparseMatrix()
m.add(sm)
```

```
class Matrix
  add(m: Matrix): Matrix = ...
  add(m: SparseMatrix): Matrix = ...
class SparseMatrix extends Matrix
  add(m: Matrix): Matrix = ...
end

m: Matrix = Matrix()
sm: Matrix = SparseMatrix()
m.add(sm)
```

Symmetric

Multiple

Method dispatch

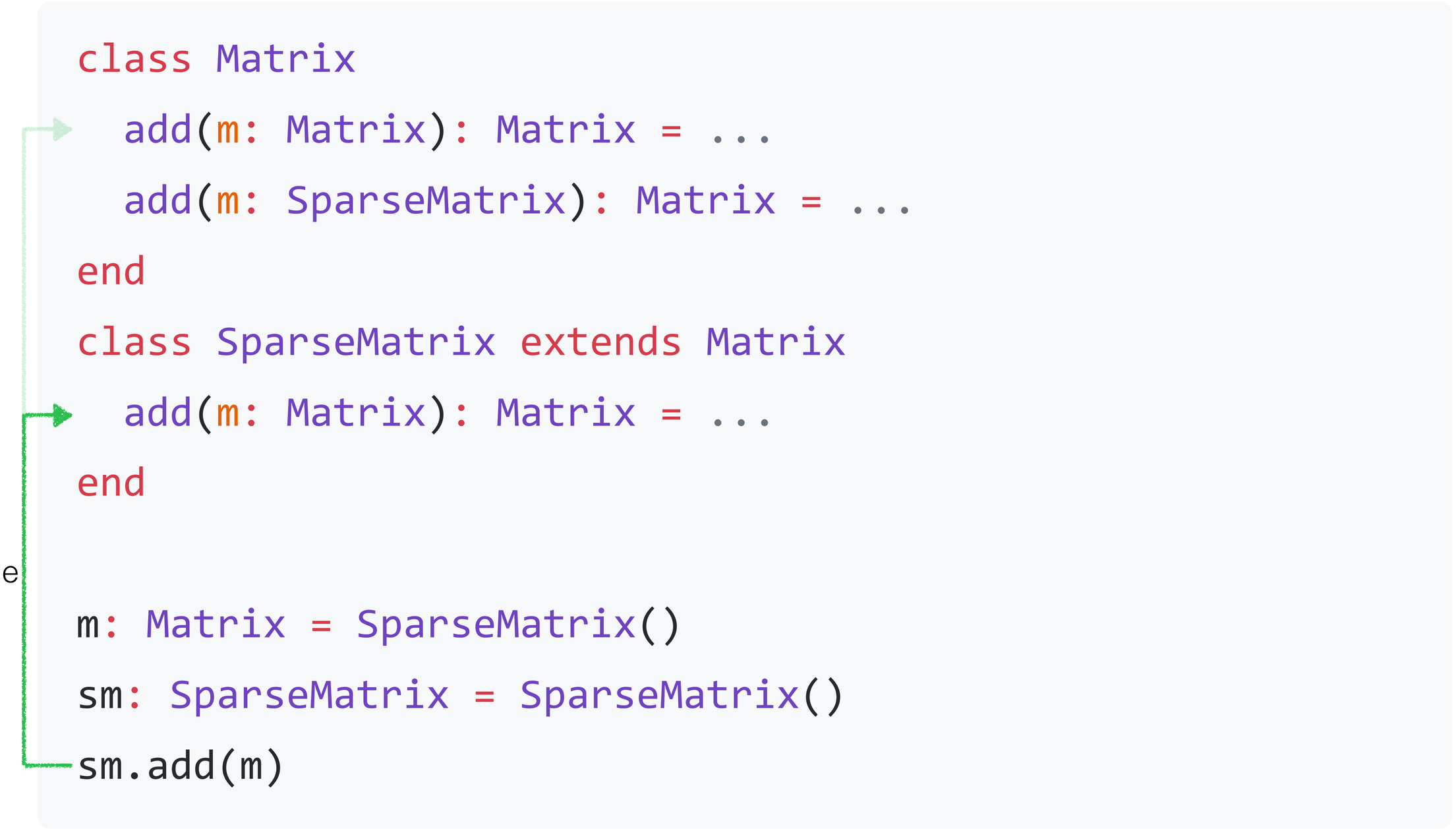
```
class Matrix
  add(m: Matrix): Matrix = ...
  add(m: SparseMatrix): Matrix = ...
class SparseMatrix extends Matrix
  add(m: Matrix): Matrix = ...
end

m: Matrix = Matrix()
sm: Matrix = SparseMatrix()
m.add(sm)
```

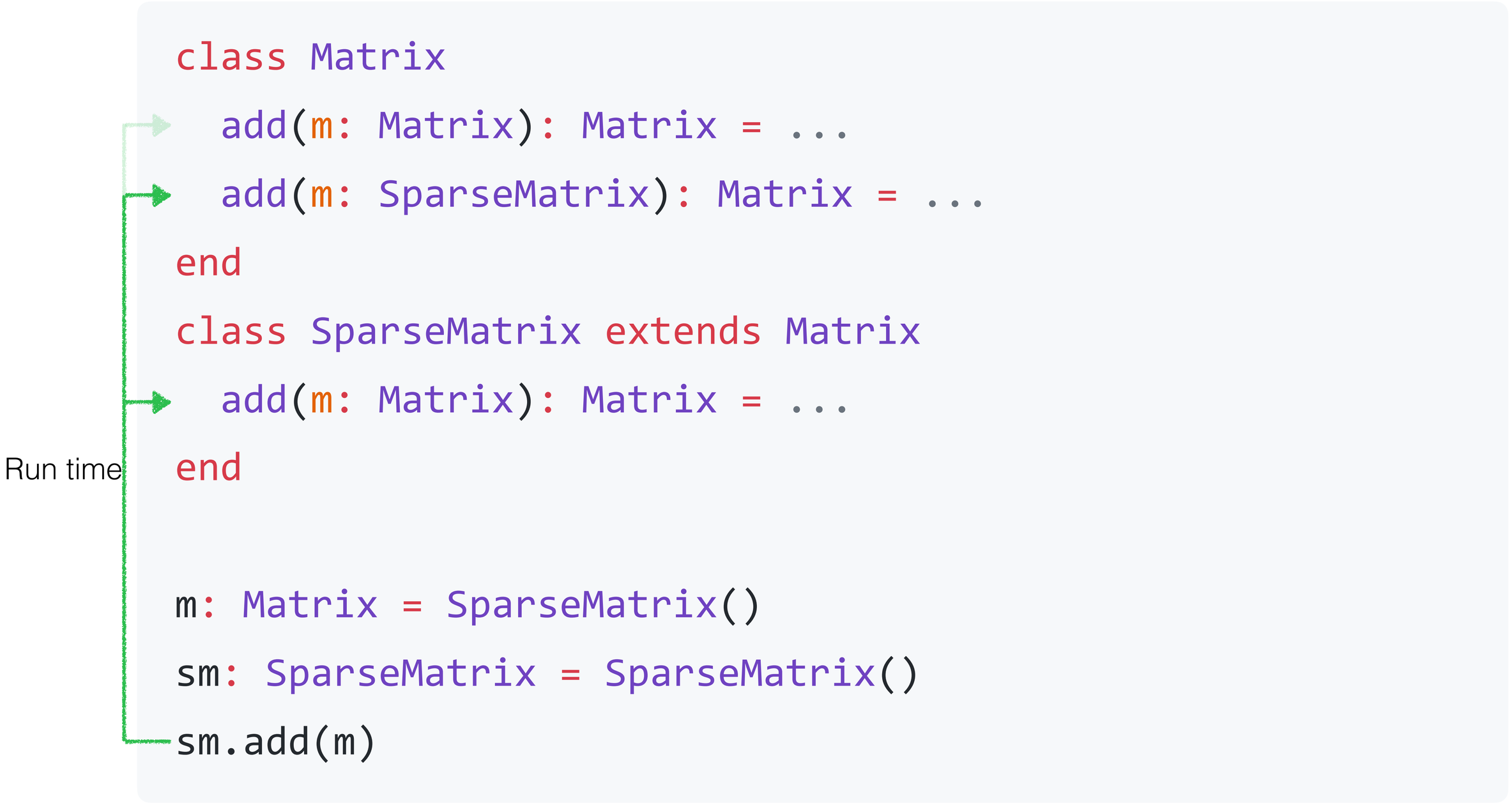
Symmetric

Multiple

Method dispatch

Dynamic languages


class Matrix
add(m: Matrix): Matrix = ...
add(m: SparseMatrix): Matrix = ...
end
class SparseMatrix extends Matrix
add(m: Matrix): Matrix = ...
end
Compile time
m: Matrix = SparseMatrix()
sm: SparseMatrix = SparseMatrix()
sm.add(m)


class Matrix
add(m: Matrix): Matrix = ...
add(m: SparseMatrix): Matrix = ...
end
class SparseMatrix extends Matrix
add(m: Matrix): Matrix = ...
end
m: Matrix = SparseMatrix()
sm: SparseMatrix = SparseMatrix()
sm.add(m)
Run time

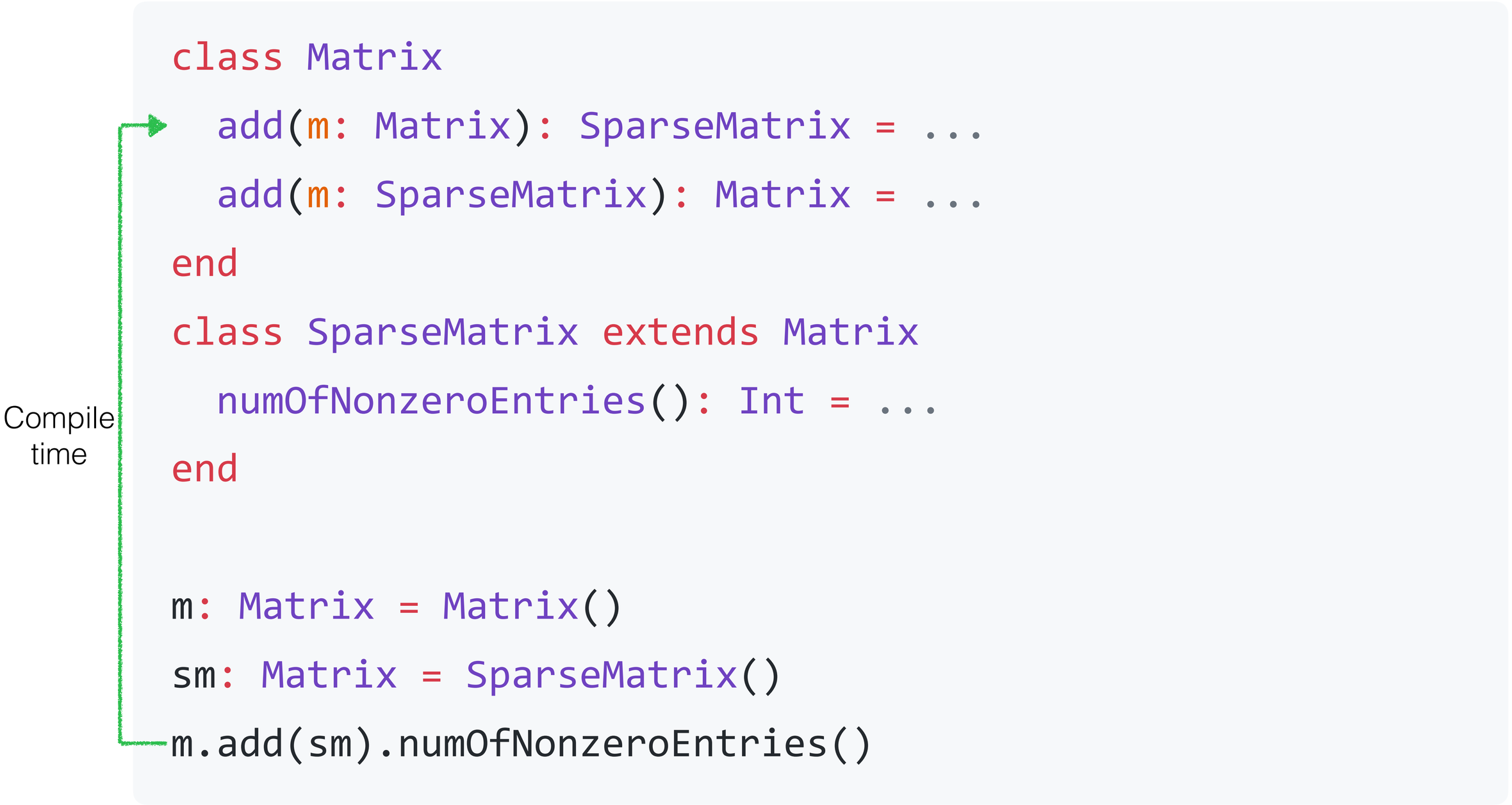
```
class Matrix
  add(m: Matrix): Matrix = ...
  (Matrix, SparseMatrix)
end
class SparseMatrix extends Matrix
  (SparseMatrix, Matrix)
end

m: Matrix = SparseMatrix()
sm: SparseMatrix = SparseMatrix()
sm.add(m)
```

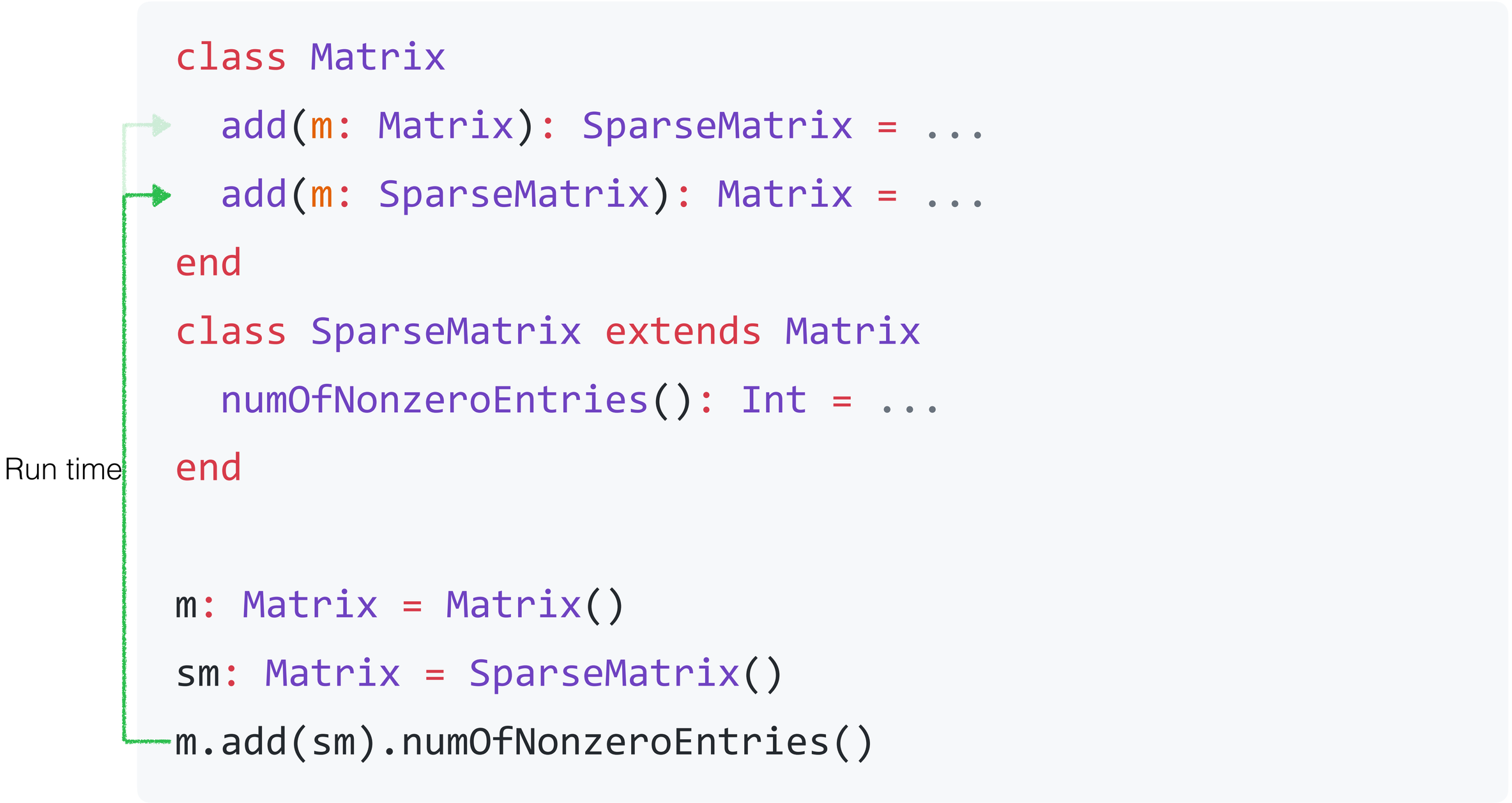
Run time

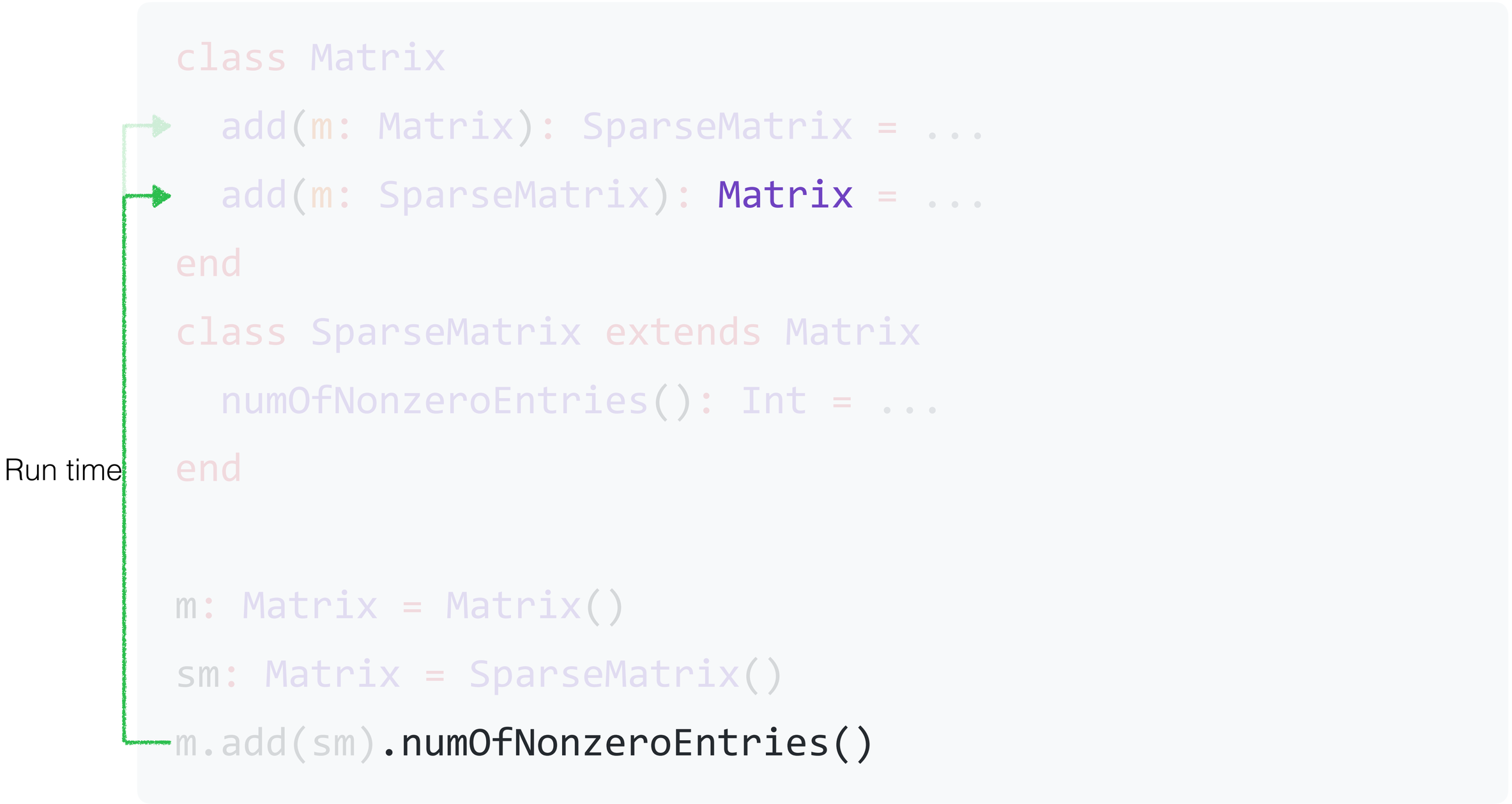
```
class Matrix
  add(m: Matrix): Matrix = ...
  (Matrix, SparseMatrix)
end
class SparseMatrix extends Matrix
  (SparseMatrix, Matrix)
end

m: Matrix = SparseMatrix()
sm: SparseMatrix = SparseMatrix()
sm.add(m)
```

Run time

Ambiguous method calls


Compile time
class Matrix
add(m: Matrix): SparseMatrix = ...
add(m: SparseMatrix): Matrix = ...
end
class SparseMatrix extends Matrix
numOfNonzeroEntries(): Int = ...
end
m: Matrix = Matrix()
sm: Matrix = SparseMatrix()
m.add(sm).numOfNonzeroEntries()


class Matrix
add(m: Matrix): SparseMatrix = ...
add(m: SparseMatrix): Matrix = ...
end
class SparseMatrix extends Matrix
numOfNonzeroEntries(): Int = ...
end
m: Matrix = Matrix()
sm: Matrix = SparseMatrix()
m.add(sm).numOfNonzeroEntries()
Run time

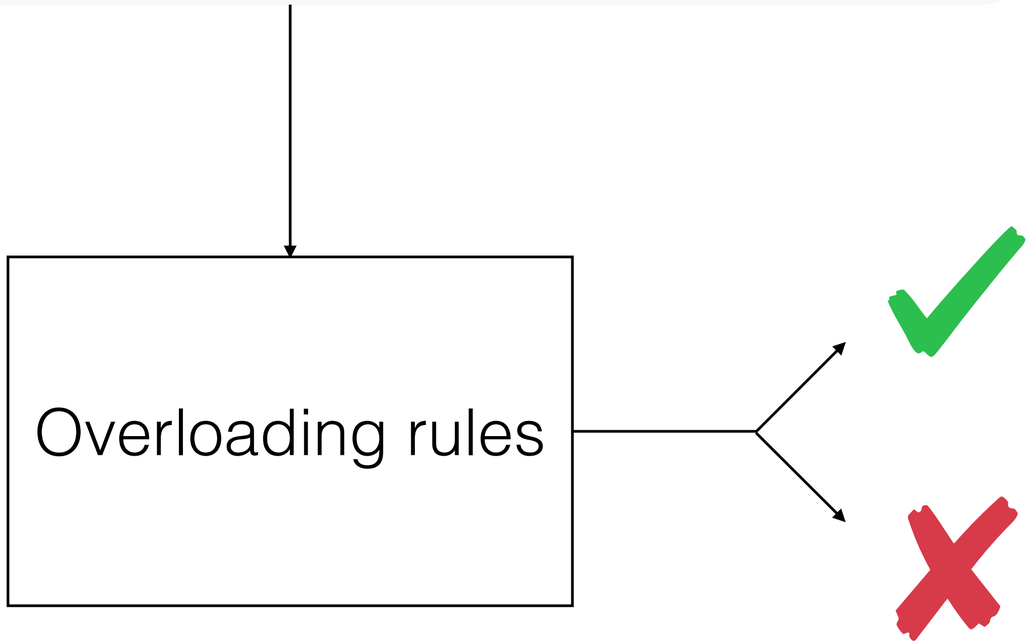

class Matrix
add(m: Matrix): SparseMatrix = ...
add(m: SparseMatrix): Matrix = ...
end
class SparseMatrix extends Matrix
numOfNonzeroEntries(): Int = ...
end
m: Matrix = Matrix()
sm: Matrix = SparseMatrix()
m.add(sm).numOfNonzeroEntries()
Run time

```
class Matrix
  add(m: Matrix): SparseMatrix = ...
  add(m: SparseMatrix): Matrix = ...
end
class SparseMatrix extends Matrix
  numOfNonzeroEntries(): Int = ...
end

m: Matrix = Matrix()
sm: Matrix = SparseMatrix()
m.add(sm).numOfNonzeroEntries()
```

Run time

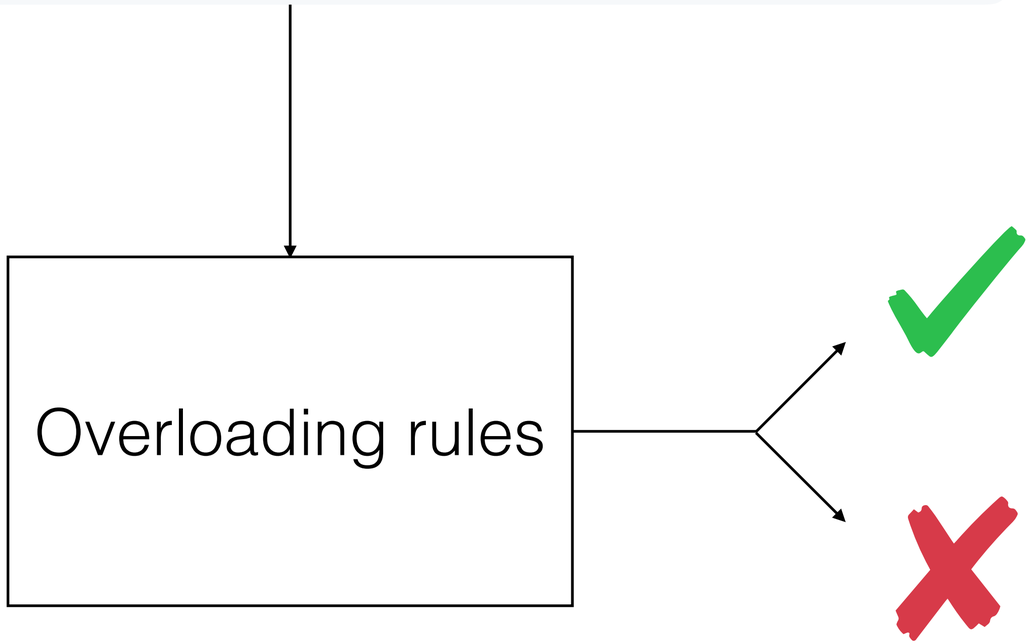
Broken type preservation

```
add(m: Matrix, m: Matrix): Matrix = ...
add(m: Matrix, m: SparseMatrix): Matrix = ...
add(m: SparseMatrix, m: SparseMatrix): SparseMatrix = ...
...
```



Compile time

```
add(m: Matrix, m: Matrix): Matrix = ...
add(m: Matrix, m: SparseMatrix): Matrix = ...
add(m: SparseMatrix, m: SparseMatrix): SparseMatrix = ...
...
```



add_S

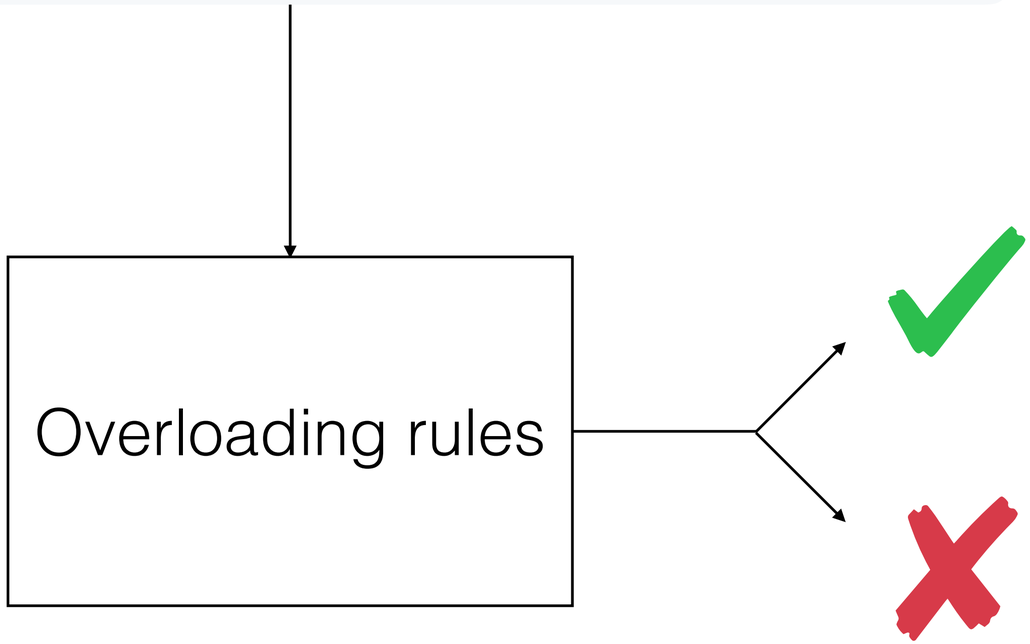
add_1 add_2 add_3 add_4

Compile time

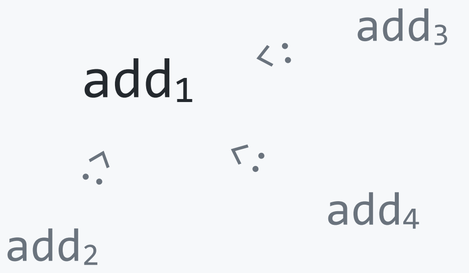
Run time

PLRG

```
add(m: Matrix, m: Matrix): Matrix = ...
add(m: Matrix, m: SparseMatrix): Matrix = ...
add(m: SparseMatrix, m: SparseMatrix): SparseMatrix = ...
...
```



add_S



(parameter type)

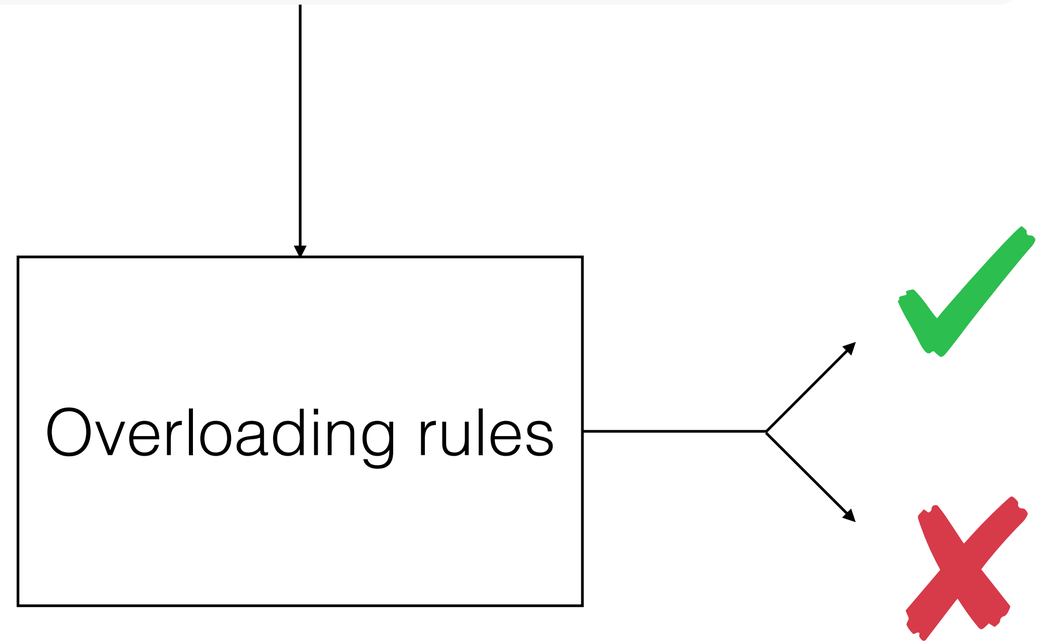
Unambiguity

Compile time

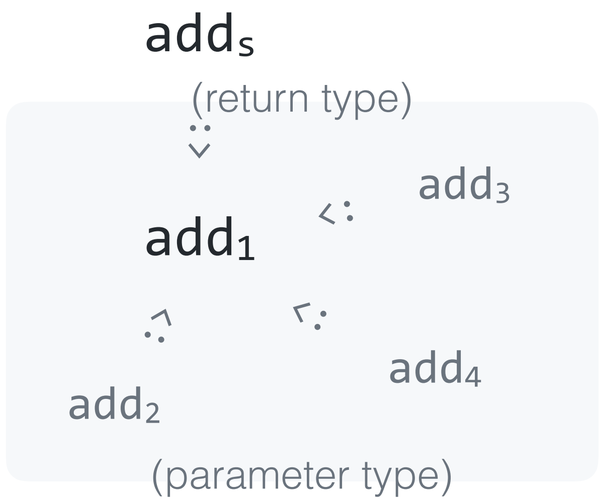
Run time

PLRG

```
add(m: Matrix, m: Matrix): Matrix = ...
add(m: Matrix, m: SparseMatrix): Matrix = ...
add(m: SparseMatrix, m: SparseMatrix): SparseMatrix = ...
...
```



Type Preservation



Unambiguity

Compile time

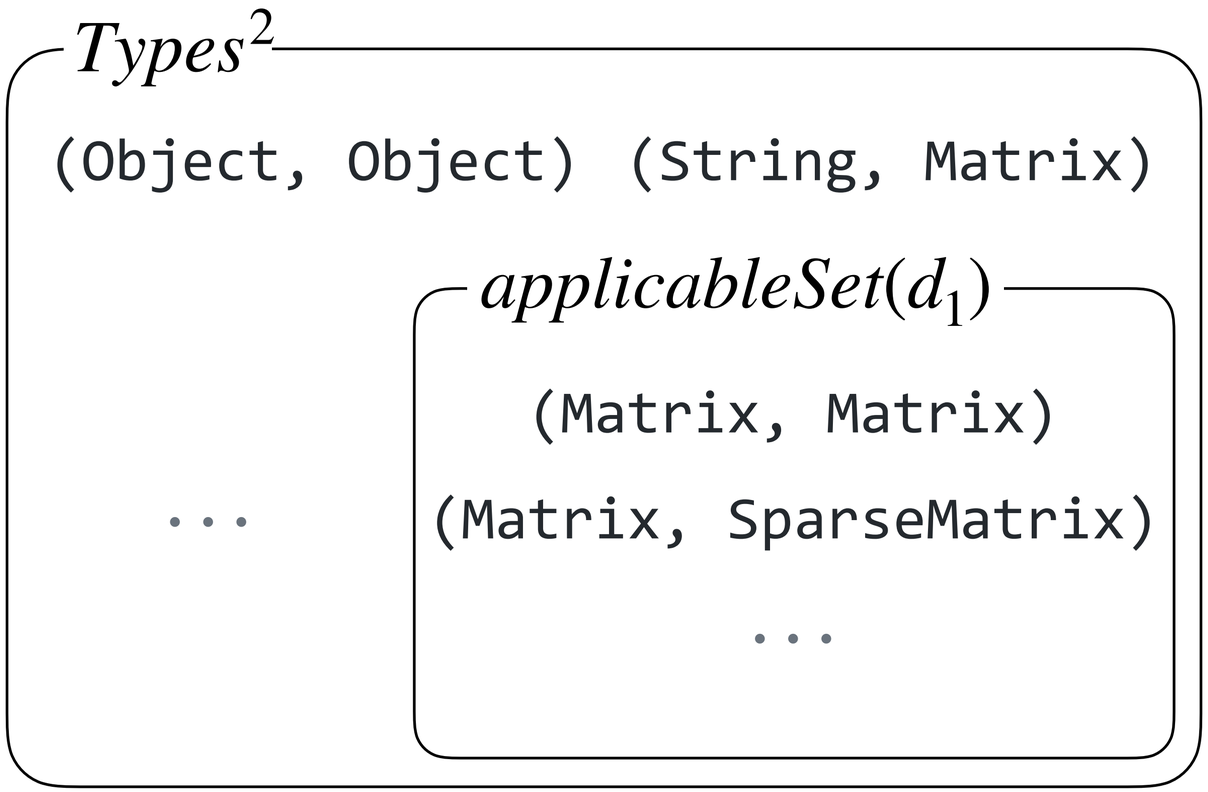
Run time

PLRG

$$applicableSet(d) = \{T : d \textbf{ is applicable to } T\}$$

d_1 :

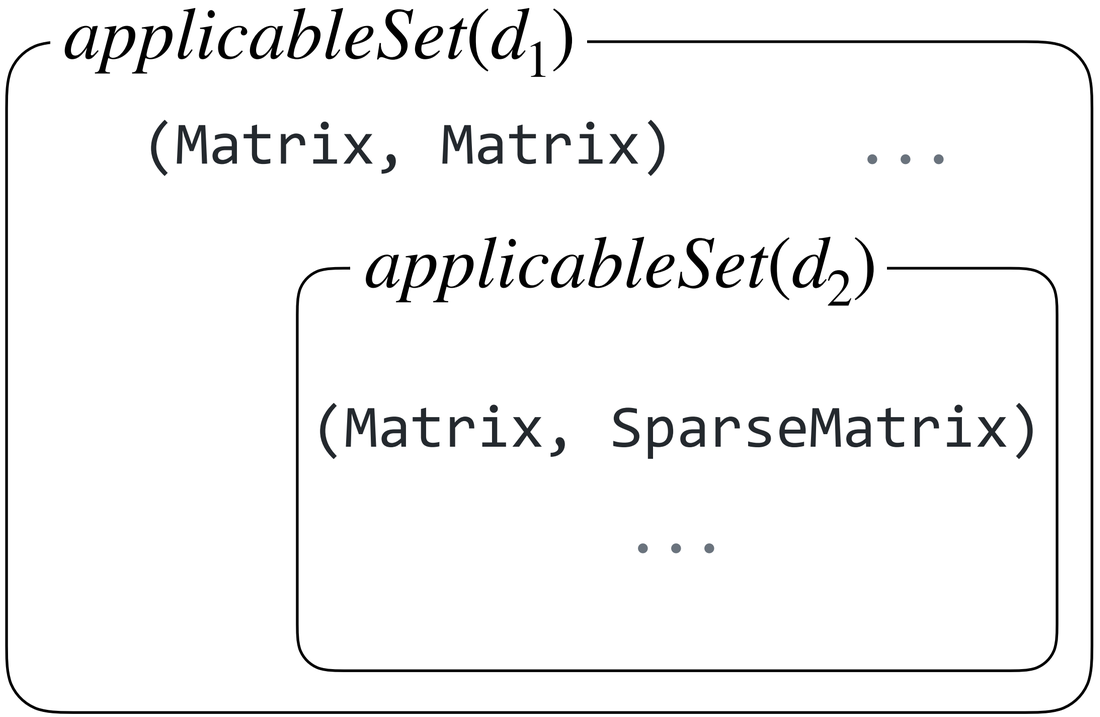
```
add(m: Matrix, m: Matrix): Matrix = ...
```



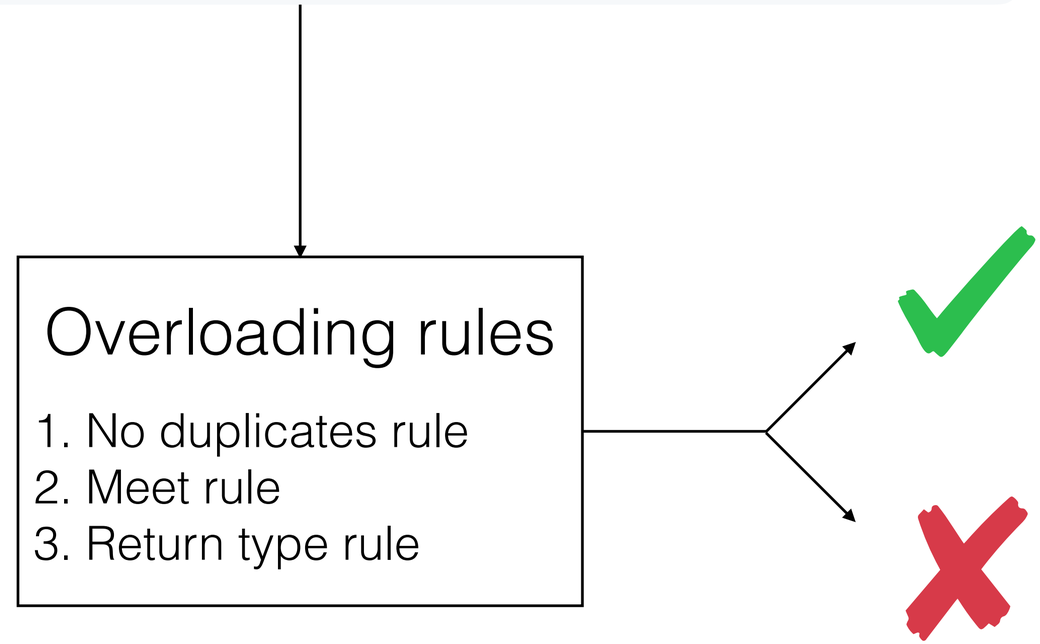
$$d_2 \sqsubseteq d_1 \textit{ iff } applicableSet(d_2) \subseteq applicableSet(d_1)$$

is more specific than

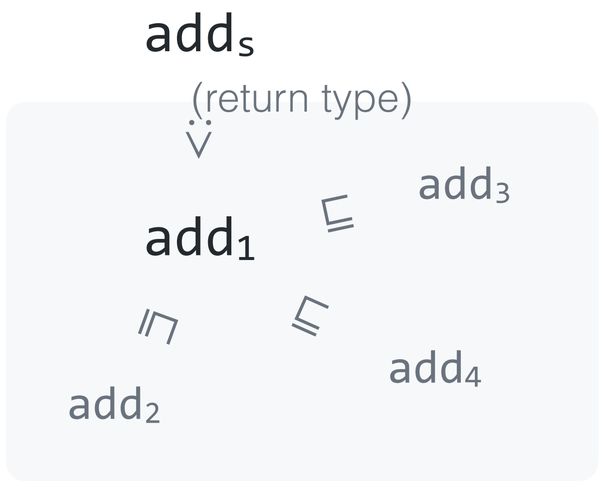
d_1 :
```
add(m: Matrix, m: Matrix): Matrix = ...
```
d_2 :
```
add(m: Matrix, m: SparseMatrix): Matrix = ...
```



```
add(m: Matrix, m: Matrix): Matrix = ...
add(m: Matrix, m: SparseMatrix): Matrix = ...
add(m: SparseMatrix, m: SparseMatrix): SparseMatrix = ...
...
```



Type Preservation

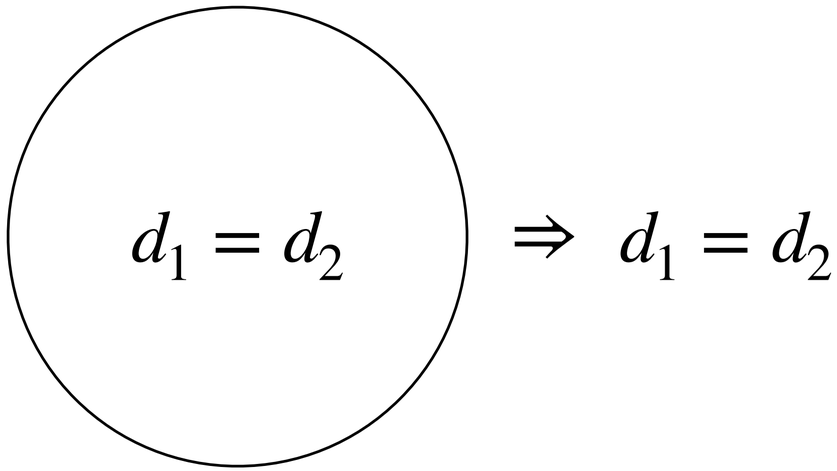


Unambiguity

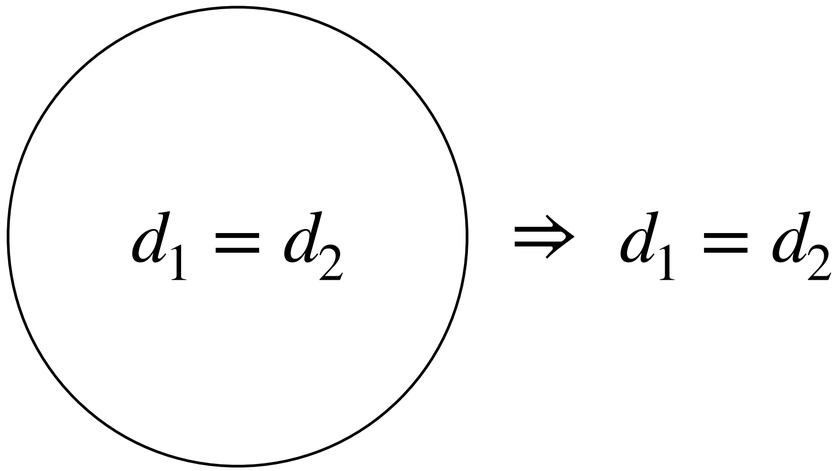
Compile time

Run time

PLRG

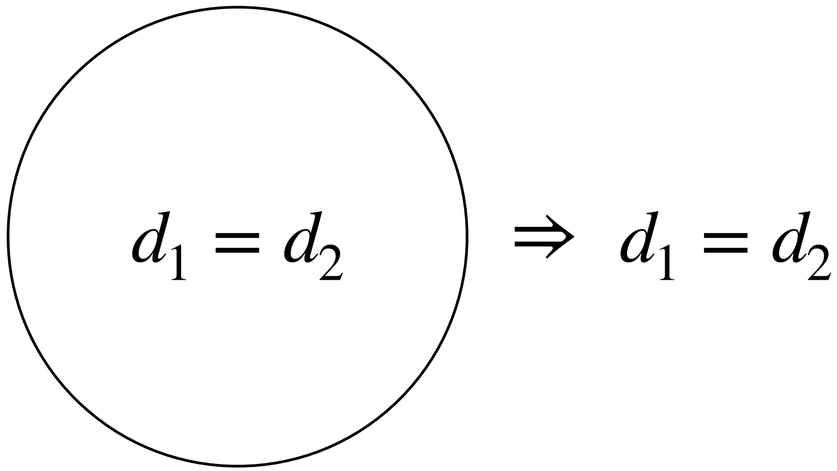


No duplicates rule



No duplicates rule

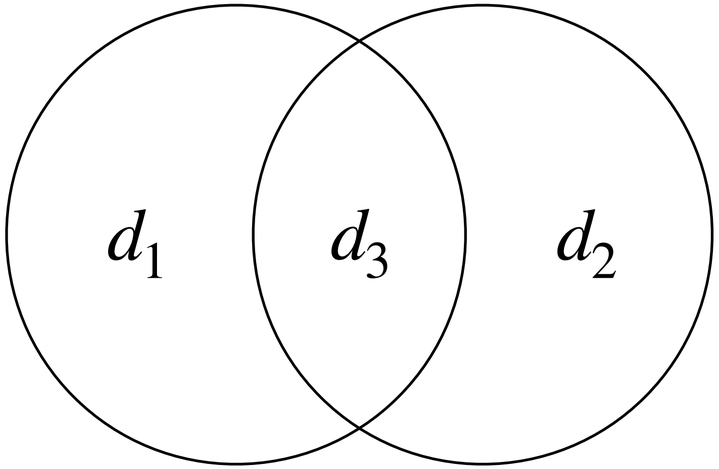
```
class Matrix
  add(m: Matrix): Matrix = ...
  add(m: Matrix): Matrix = ...
end
```



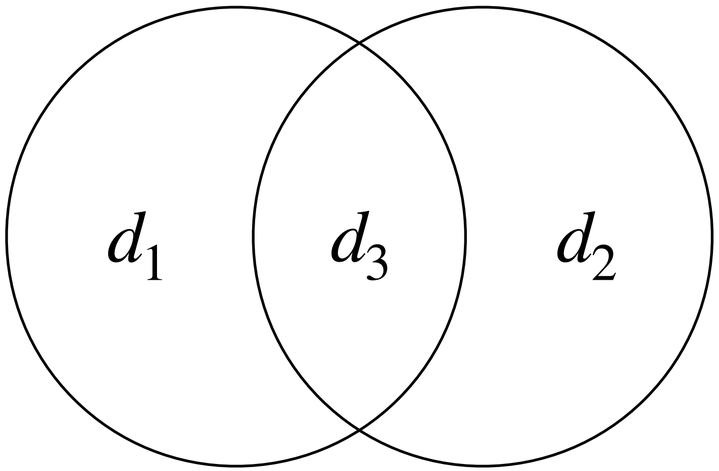
No duplicates rule

```
class Matrix
  add(m: Matrix): Matrix = ...


end
```


PLRG

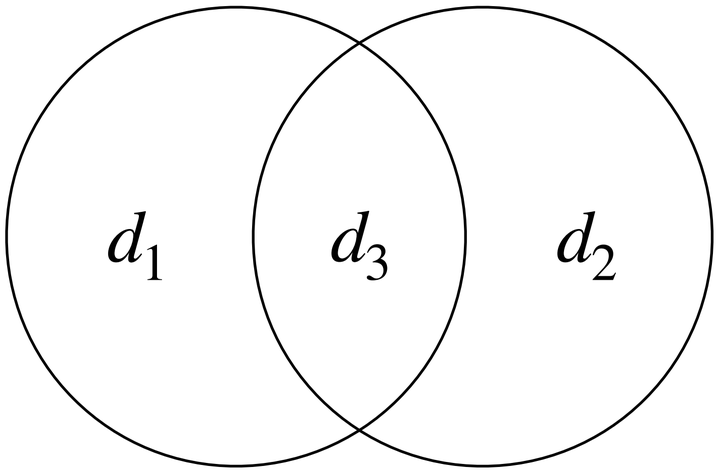


Meet rule



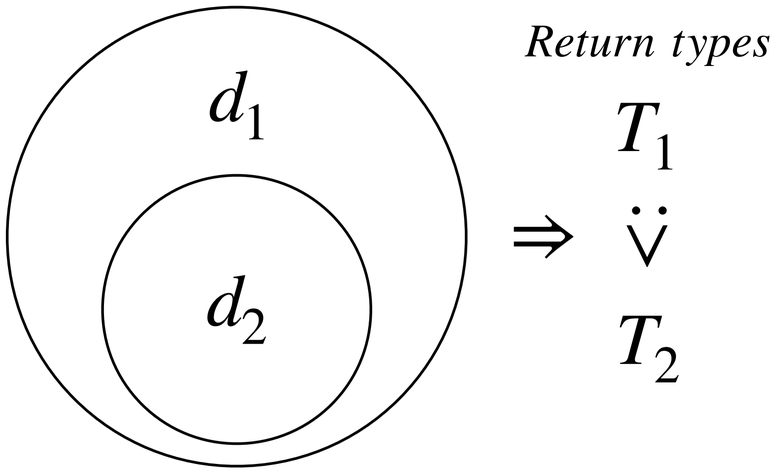
Meet rule

```
class Matrix
  add(m: SparseMatrix): Matrix = ...
end
class SparseMatrix extends Matrix
  add(m: Matrix): Matrix = ...


end
```

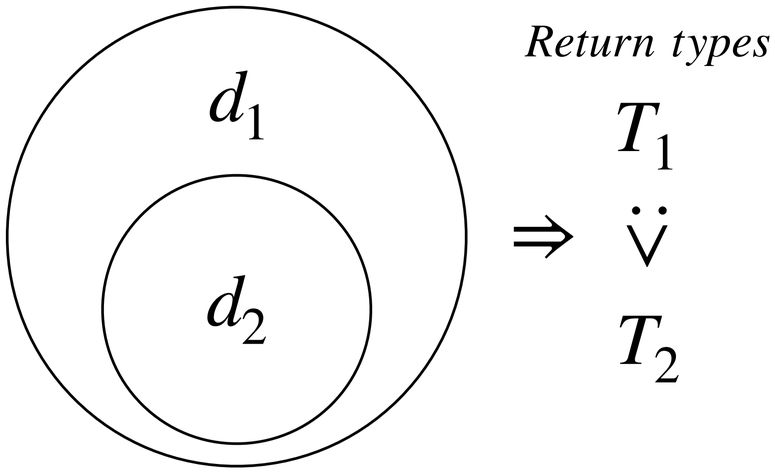


Meet rule

```
class Matrix
  add(m: SparseMatrix): Matrix = ...
end
class SparseMatrix extends Matrix
  add(m: Matrix): Matrix = ...
  add(m: SparseMatrix): Matrix = ...
end
```

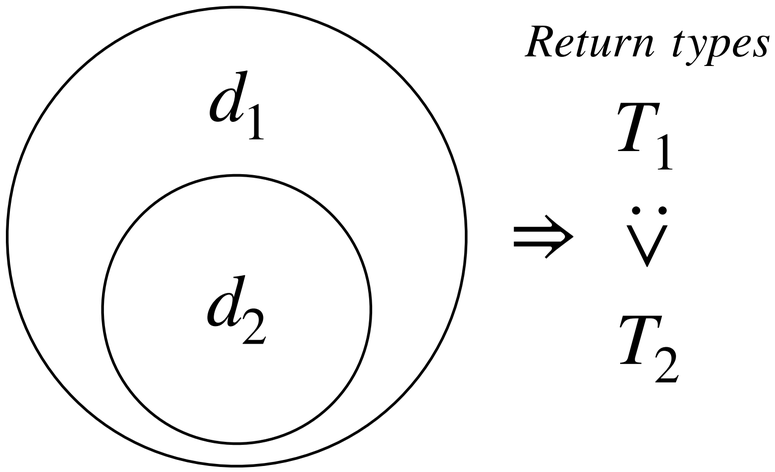


Return type rule



Return type rule

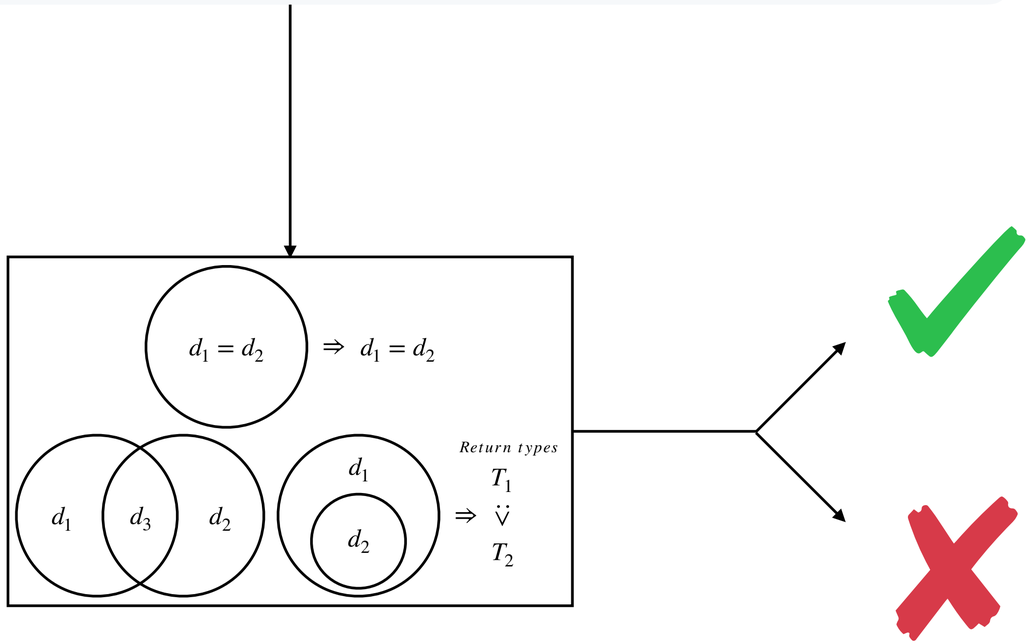
```
class Matrix
  add(m: Matrix): SparseMatrix = ...
  add(m: SparseMatrix): Matrix = ...
end
class SparseMatrix extends Matrix
end
```



Return type rule

```
class Matrix
  add(m: Matrix): Matrix = ...
  add(m: SparseMatrix): Matrix = ...
end
class SparseMatrix extends Matrix
end
```

```
add(m: Matrix, m: Matrix): Matrix = ...
add(m: Matrix, m: SparseMatrix): Matrix = ...
add(m: SparseMatrix, m: SparseMatrix): SparseMatrix = ...
...
```



Type Preservation

add_S

(return type)

⋮

add_1 add_2 add_3 add_4

Unambiguity

Compile time

Run time

PLRG

```
class Array[T] end

makeArray[T](t: T): Array[T] = ...

i: Int = 1
makeArray(i)
```

Parametric polymorphism

d_1 :

```
makeArray[T](t: T): Array[T] = ...
```

$instances(d_1)$:

```
makeArray(t: Object): Array[Object] = ...
makeArray(t: String): Array[String] = ...
makeArray(t: Number): Array[Number] = ...
makeArray(t: Int): Array[Int] = ...
...
```

$$applicableSet(d) = \{T : \exists D \in instances(d) \,.\, D \textbf{ is applicable to } T\}$$

d_1 :

```
makeArray[T](t: T): Array[T] = ...
```

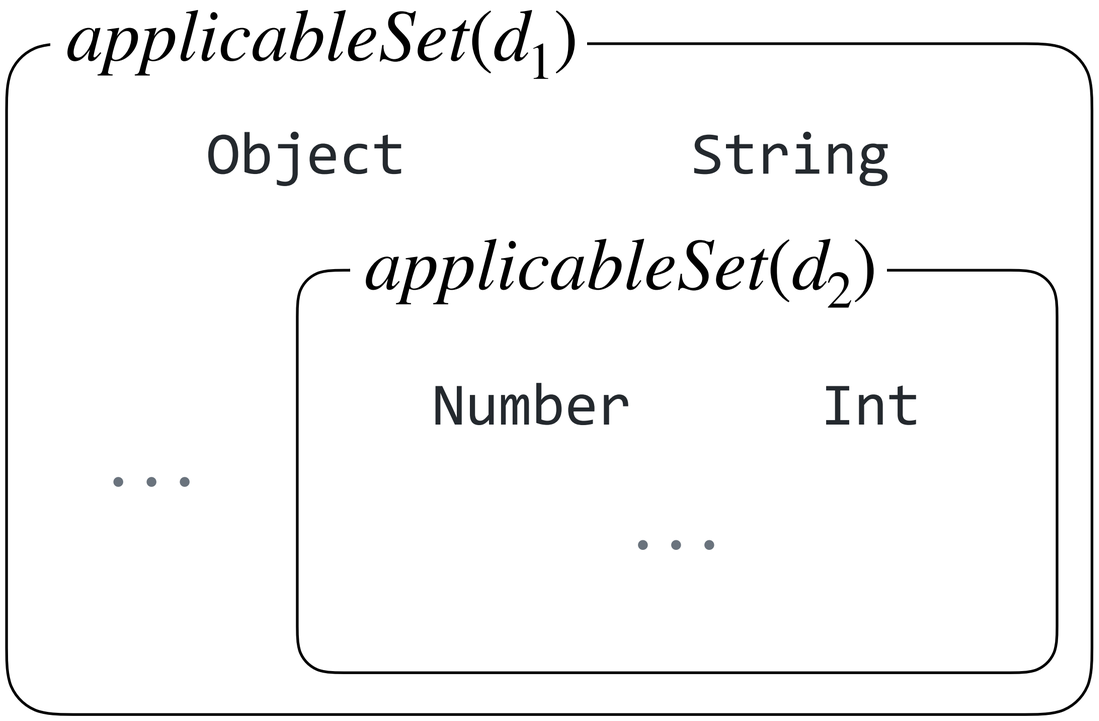
$Types = applicableSet(d_1)$

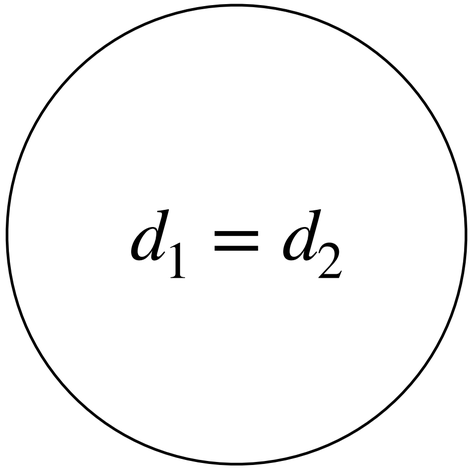
```
Object          String

Number          Int

...
```

$$d_2 \sqsubseteq d_1 \textit{ iff } applicableSet(d_2) \subseteq applicableSet(d_1)$$

is more specific than

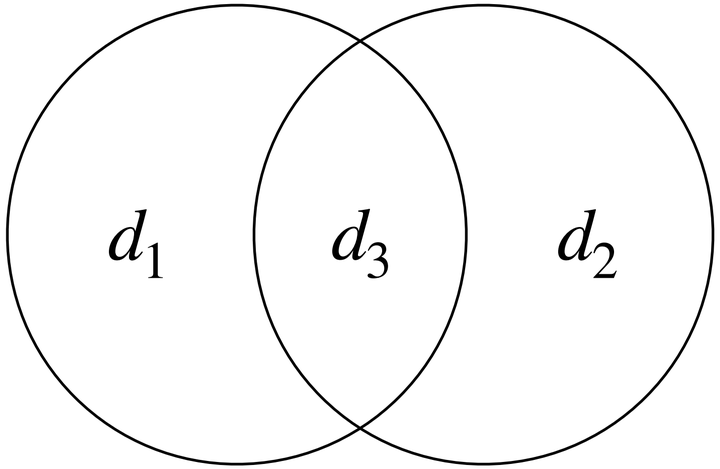
d_1 :
```
makeArray[T](t: T): Array[T] = ...
```
d_2 :
```
makeArray(i: Number): Array[Number] = ...
```



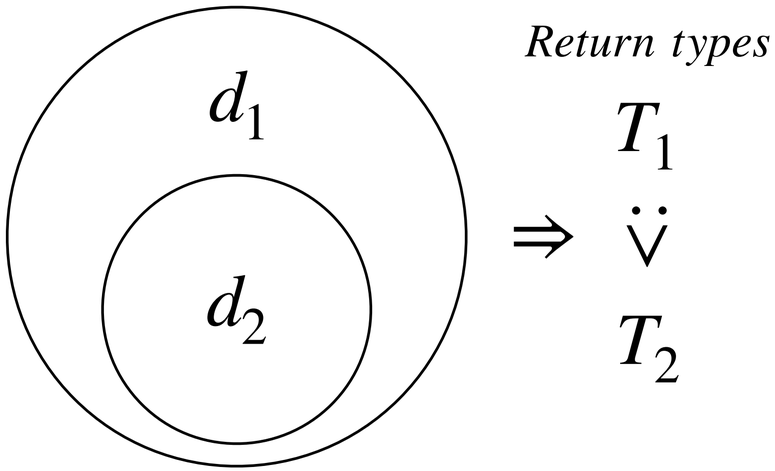


$\Rightarrow \; d_1 = d_2$

No duplicates rule



Meet rule



Return type rule

```
class Array[T] end


makeArray[T](t: T): Array[T] = ...
makeArray(i: Number): Array[Number] = ...
```

Return types

T_1

$\Rightarrow$ $\vdots$

T_2

Return type rule

Compile time

```
class Array[T] end

makeArray[T](t: T): Array[T] = ...
makeArray(i: Number): Array[Number] = ...

i: Object = 1
a: Array[Object] = makeArray(i)
a[0] = "1"
```


PLRG

Return types

T_1

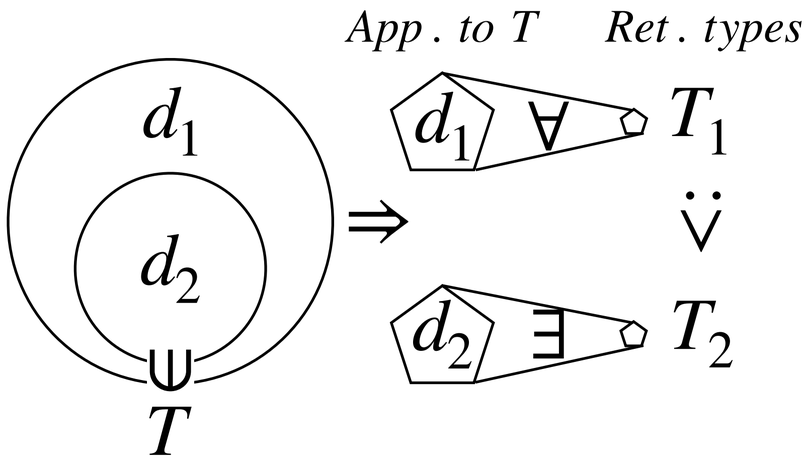
$\Rightarrow$ $\vdots$

T_2

Return type rule

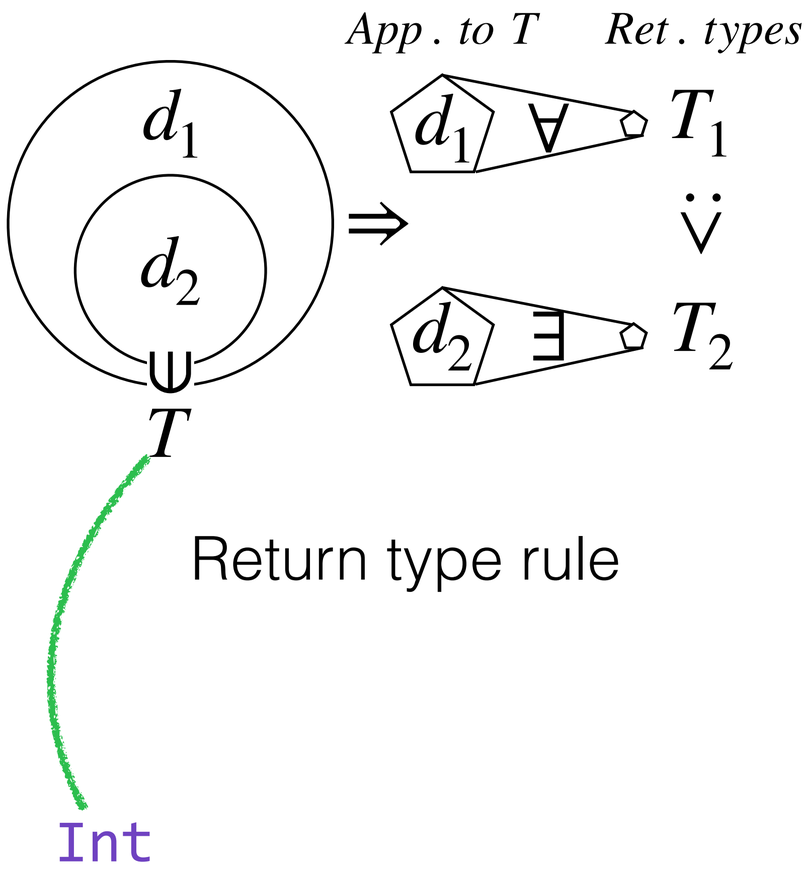
Run time

```
class Array[T] end

makeArray[T](t: T): Array[T] = ...
makeArray(i: Number): Array[Number] = ...

i: Object = 1
a: Array[Object] = makeArray(i)
a[0] = "1"
```



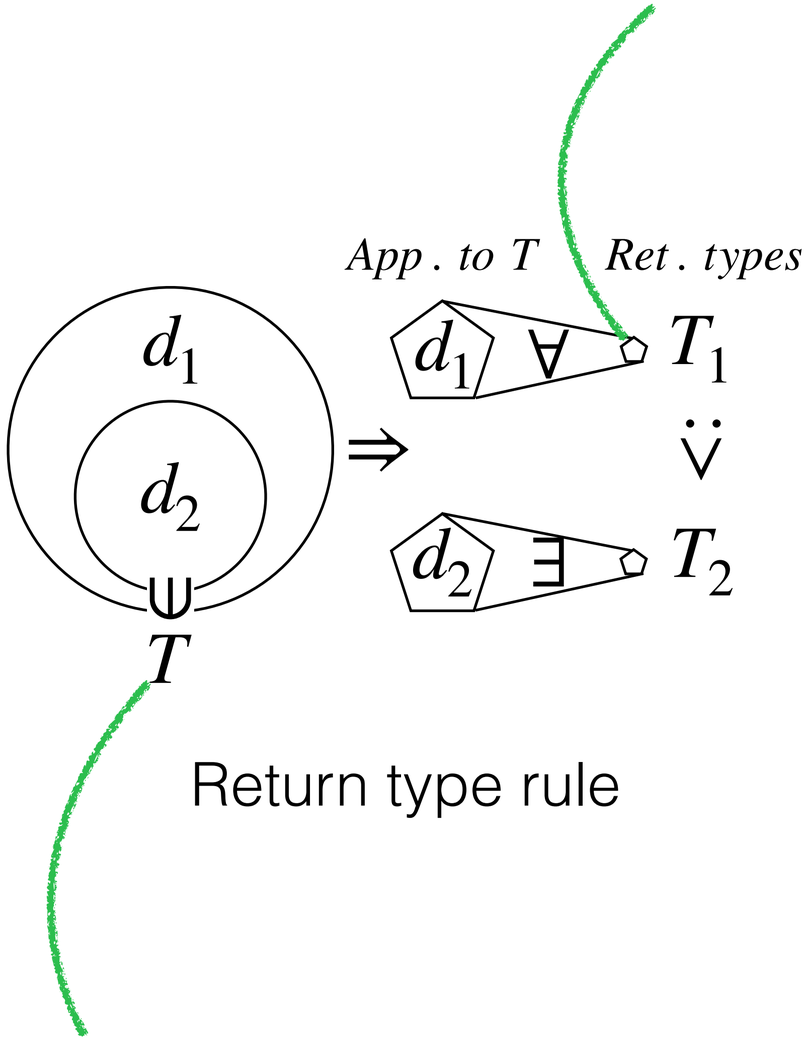
Return type rule

```
class Array[T] end


makeArray[T](t: T): Array[T] = ...
makeArray(i: Number): Array[Number] = ...
```




```
class Array[T] end


makeArray[T](t: T): Array[T] = ...
makeArray(i: Number): Array[Number] = ...
```

```
makeArray(i: Object): Array[Object] = ...
```

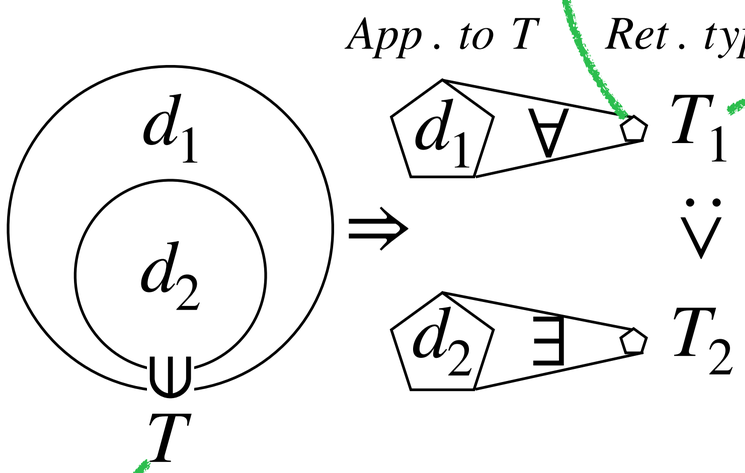



```
class Array[T] end


makeArray[T](t: T): Array[T] = ...
makeArray(i: Number): Array[Number] = ...
```

PLRG

```
makeArray(i: Object): Array[Object] = ...
```

Array[Object]



Return type rule

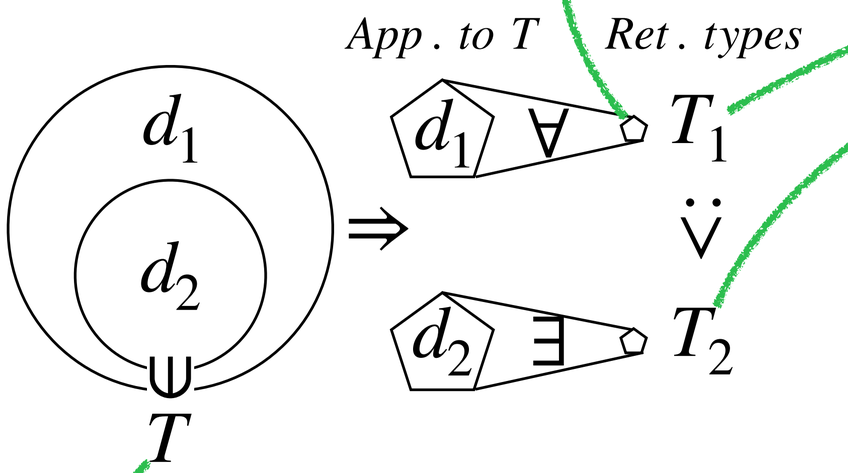
```
class Array[T] end

makeArray[T](t: T): Array[T] = ...
makeArray(i: Number): Array[Number] = ...
```

Int

PLRG

```
makeArray(i: Object): Array[Object] = ...
```

Array[Object]



Return type rule

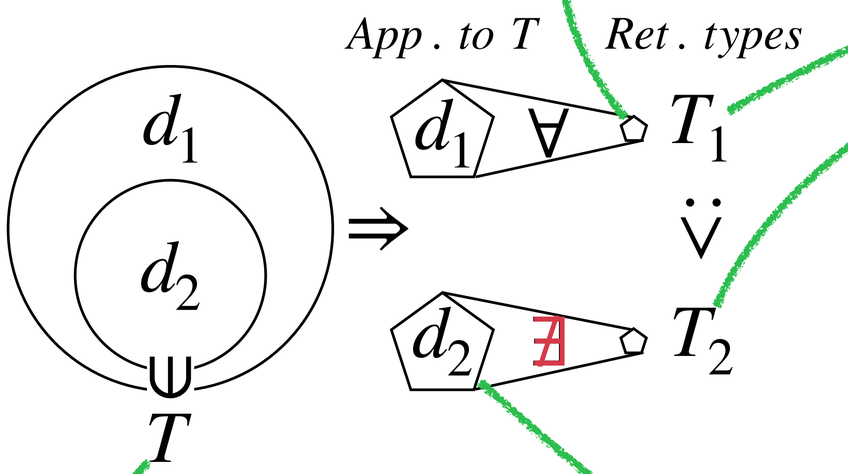
```
class Array[T] end


makeArray[T](t: T): Array[T] = ...
makeArray(i: Number): Array[Number] = ...
```

Int

PLRG

```
makeArray(i: Object): Array[Object] = ...
```

Array[Object]



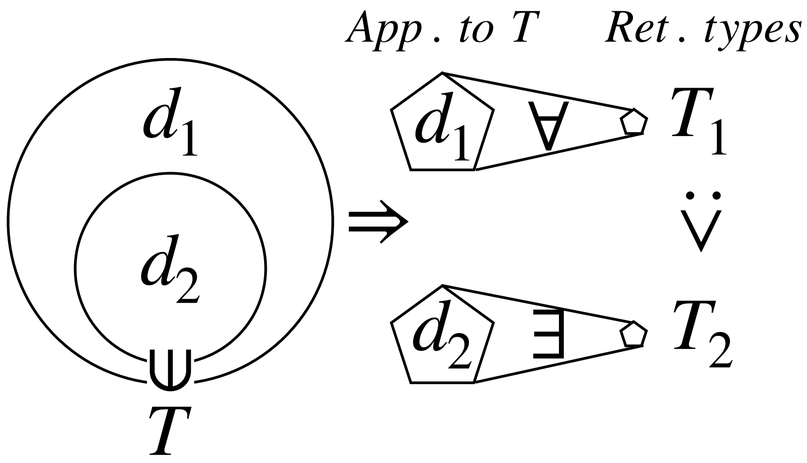
Return type rule

```
class Array[T] end

makeArray[T](t: T): Array[T] = ...
makeArray(i: Number): Array[Number] = ...
```

```
makeArray(i: Number): Array[Number] = ...
```

Int

PLRG

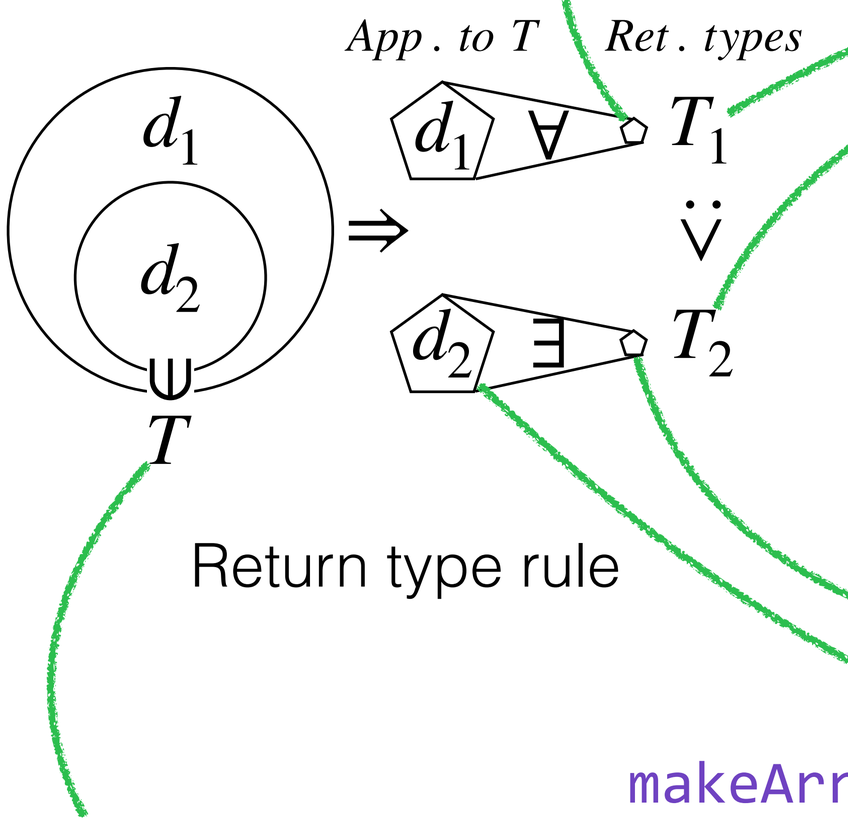


Return type rule

```
class Array[T] end


makeArray[T](t: T): Array[T] = ...
makeArray[T](i: Number): Array[T] = ...
```


PLRG

```
makeArray(i: Object): Array[Object] = ...
```



Return type rule

```
Int
```

```
Array[Object]
```

```
class Array[T] end


makeArray[T](t: T): Array[T] = ...
makeArray[T](i: Number): Array[T] = ...
```

✓

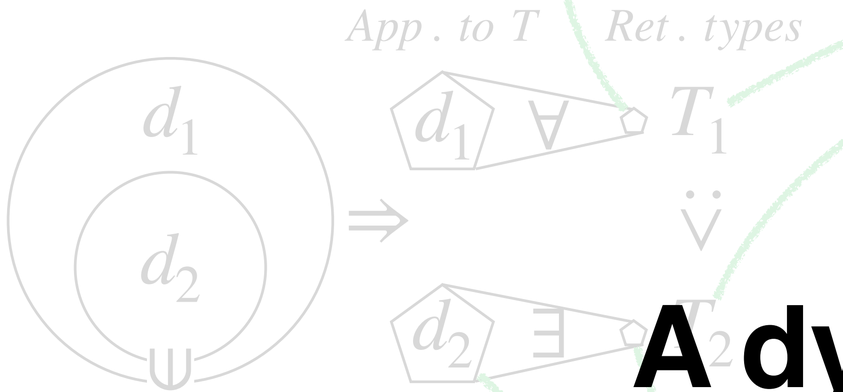
```
makeArray(i: Number): Array[Object] = ...
```

```
makeArray[T](i: Number): Array[T] = ...
```


PLRG

```
makeArray(i: Object): Array[Object] = ...
```

Array[Object]




```
class Array[T] end
```

Return type rule

A dynamic dispatch algorithm needs statically determined return types

```
makeArray[T](i: Number): Array[T] = ...
```

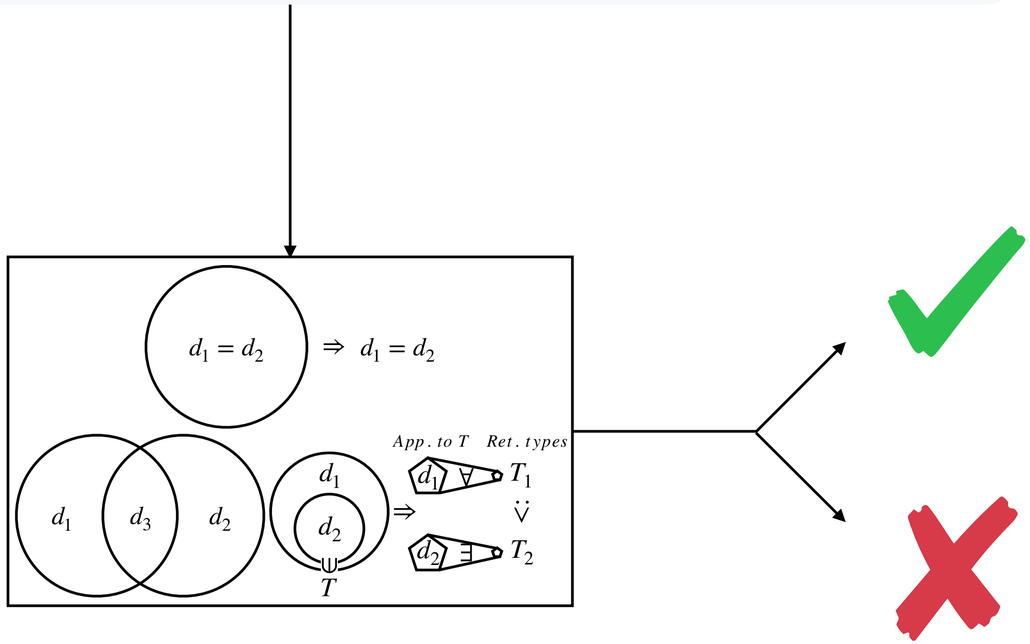
```
makeArray(i: Number): Array[Object] = ...
```

```
makeArray[T](i: Number): Array[T] = ...
```

Int

PLRG

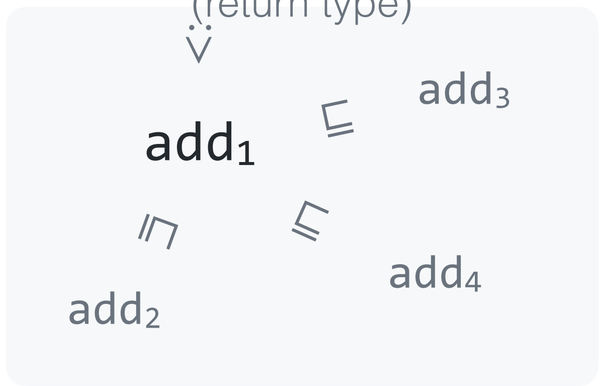
```
makeArray[T](t: T): Array[T] = ...
makeArray(t: Number): Array[Number] = ...
makeArray[T](i: Number): Array[T] = ...
...
```



Type Preservation

$add_S : T$

(return type)



Unambiguity

Compile time

Run time

PLRG

FGFV Calculus

Program	Π	::=	$\overline{\psi}, e$
Class declaration	ψ	::=	$\texttt{trait}\ T[\![\overline{V\ \beta}]\!] <: \{\overline{t}\}\ \overline{\mu}\ \texttt{end} \mid \texttt{object}\ O[\![\overline{\beta}]\!](\overline{x{:}\,\tau}) <: \{\overline{t}\}\ \overline{\mu}\ \texttt{end}$
Method definition	μ	::=	$m[\![\overline{\kappa}]\!](\overline{x{:}\,\tau}){:}\,\tau = e$
Class type parameter binding	β	::=	$P <: \{\overline{\tau}\}$
Method type parameter binding	κ	::=	$\{\overline{\tau}\} <: P <: \{\overline{\tau}\}$
Variance mark	V	::=	$\texttt{+} \mid \texttt{-} \mid \texttt{=}$
Expression	e	::=	$z \mid ((\overline{x{:}\,\tau}){:}\,\tau \Rightarrow e) \mid e@(\overline{e}) \mid O[\![\overline{\tau}]\!](\overline{e}) \mid e.m(\overline{e})$
Bindable variable	z	::=	$x \mid \texttt{self}$
Type	τ	::=	$P \mid c \mid (\overline{\tau}) \mid (\tau \to \tau) \mid \texttt{Any}$
Constructed type	c	::=	$t \mid O[\![\overline{\tau}]\!]$
Trait type	t	::=	$T[\![\overline{\tau}]\!]$

FGFV Calculus

Multiple inheritance

Program	Π	::=	$\overline{\psi}, e$
Class declaration	ψ	::=	$\texttt{trait}\ T[\![\overline{V\ \beta}]\!] <: \{\overline{t}\}\ \overline{\mu}\ \texttt{end} \mid \texttt{object}\ O[\![\overline{\beta}]\!](\overline{x{:}\,\tau}) <: \{\overline{t}\}\ \overline{\mu}\ \texttt{end}$
Method definition	μ	::=	$m[\![\overline{\kappa}]\!](\overline{x{:}\,\tau}){:}\,\tau = e$
Class type parameter binding	β	::=	$P <: \{\overline{\tau}\}$
Method type parameter binding	κ	::=	$\{\overline{\tau}\} <: P <: \{\overline{\tau}\}$
Variance mark	V	::=	$+ \mid - \mid =$
Expression	e	::=	$z \mid ((\overline{x{:}\,\tau}){:}\,\tau \Rightarrow e) \mid e@(\overline{e}) \mid O[\![\overline{\tau}]\!](\overline{e}) \mid e.m(\overline{e})$
Bindable variable	z	::=	$x \mid \texttt{self}$
Type	τ	::=	$P \mid c \mid (\overline{\tau}) \mid (\tau \to \tau) \mid \texttt{Any}$
Constructed type	c	::=	$t \mid O[\![\overline{\tau}]\!]$
Trait type	t	::=	$T[\![\overline{\tau}]\!]$

FGFV Calculus

Multiple inheritance

Parametric polymorphism

Program	Π	::=	$\overline{\psi}, e$
Class declaration	ψ	::=	$\texttt{trait}\ T[\![\overline{V\ \beta}]\!] <: \{\overline{t}\}\ \overline{\mu}\ \texttt{end} \mid \texttt{object}\ O[\![\overline{\beta}]\!](\overline{x{:}\,\tau}) <: \{\overline{t}\}\ \overline{\mu}\ \texttt{end}$
Method definition	μ	::=	$m[\![\overline{\kappa}]\!](\overline{x{:}\,\tau}){:}\,\tau = e$
Class type parameter binding	β	::=	$P <: \{\overline{\tau}\}$
Method type parameter binding	κ	::=	$\{\overline{\tau}\} <: P <: \{\overline{\tau}\}$
Variance mark	V	::=	$+ \mid - \mid =$
Expression	e	::=	$z \mid ((\overline{x{:}\,\tau}){:}\,\tau \Rightarrow e) \mid e@(\overline{e}) \mid O[\![\overline{\tau}]\!](\overline{e}) \mid e.m(\overline{e})$
Bindable variable	z	::=	$x \mid \texttt{self}$
Type	τ	::=	$P \mid c \mid (\overline{\tau}) \mid (\tau \to \tau) \mid \texttt{Any}$
Constructed type	c	::=	$t \mid O[\![\overline{\tau}]\!]$
Trait type	t	::=	$T[\![\overline{\tau}]\!]$

FGFV Calculus

Program	Π	::=	$\overline{\psi}, e$
Class declaration	ψ	::=	$\texttt{trait}\ T[\![\overline{V}\ \overline{\beta}]\!] <: \{\overline{t}\}\ \overline{\mu}\ \texttt{end} \mid \texttt{object}\ O[\![\overline{\beta}]\!](\overline{x{:}\,\tau}) <: \{\overline{t}\}\ \overline{\mu}\ \texttt{end}$
Method definition	μ	::=	$m[\![\overline{\kappa}]\!](\overline{x{:}\,\tau}){:}\,\tau = e$
Class type parameter binding	β	::=	$P <: \{\overline{\tau}\}$
Method type parameter binding	κ	::=	$\{\overline{\tau}\} <: P <: \{\overline{\tau}\}$
Variance mark	V	::=	$\texttt{+} \mid \texttt{-} \mid \texttt{=}$
Expression	e	::=	$z \mid ((\overline{x{:}\,\tau}){:}\,\tau \Rightarrow e) \mid e@(\overline{e}) \mid O[\![\overline{\tau}]\!](\overline{e}) \mid e.m(\overline{e})$
Bindable variable	z	::=	$x \mid \texttt{self}$
Type	τ	::=	$P \mid c \mid (\overline{\tau}) \mid (\tau \to \tau) \mid \texttt{Any}$
Constructed type	c	::=	$t \mid O[\![\overline{\tau}]\!]$
Trait type	t	::=	$T[\![\overline{\tau}]\!]$

Multiple inheritance

Parametric polymorphism

Variance

FGFV Calculus

Multiple inheritance

Program	Π	::=	$\overline{\psi}, e$
Class declaration	ψ	::=	trait $T[\![\overline{\alpha}]\!] <: \{\overline{t}\}\ \overline{\mu}$ end $\mid$ object $O[\![\overline{\beta}]\!](\overline{x{:}\tau}) <: \{\overline{t}\}\ \overline{\mu}$ end

```
class A[+T] end
class A[=T] end
class A[-T] end
```

$A[T] <: A[S]$ *iff* $T <: S$

$A[T] <: A[S]$ *iff* $T \equiv S$

$A[T] <: A[S]$ *iff* $T :> S$

ametric polymorphism

Variance

@$(\overline{e})$ $\mid$ $O[\![\overline{\tau}]\!](\overline{e})$ $\mid$ $e.m(\overline{e})$

Type	τ	::=	$P \mid c \mid (\overline{\tau}) \mid (\tau \to \tau) \mid$ Any
Constructed type	c	::=	$t \mid O[\![\overline{\tau}]\!]$
Trait type	t	::=	$T[\![\overline{\tau}]\!]$

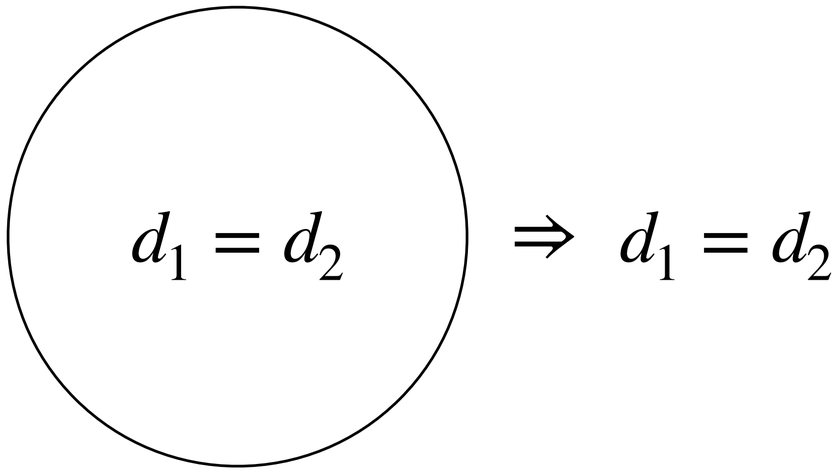
PLRG

d_1 :
d_2 :

```
add(m: Matrix, m: Matrix): Matrix = ...
makeArray[T](t: T): Array[T] = ...
```

$$dom(d_1) = \exists[](Matrix, Matrix)$$
$$dom(d_2) = \exists[T <: Object](T)$$

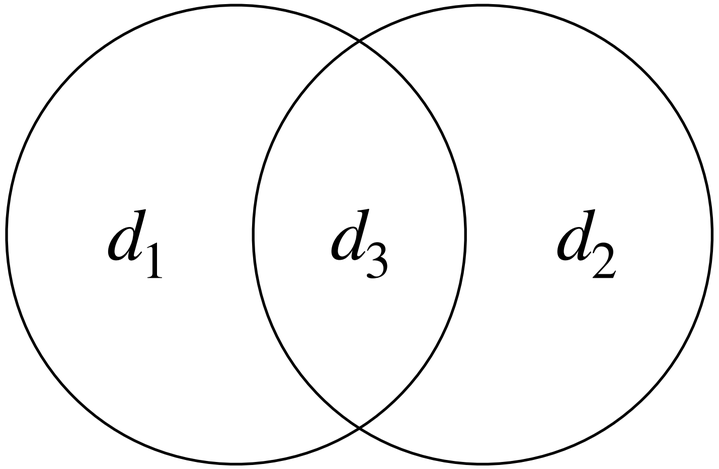
$$arrow(d_1) = \forall[](Matrix, Matrix) \rightarrow Matrix$$
$$arrow(d_2) = \forall[T <: Object](T) \rightarrow Array[T]$$



No duplicates rule

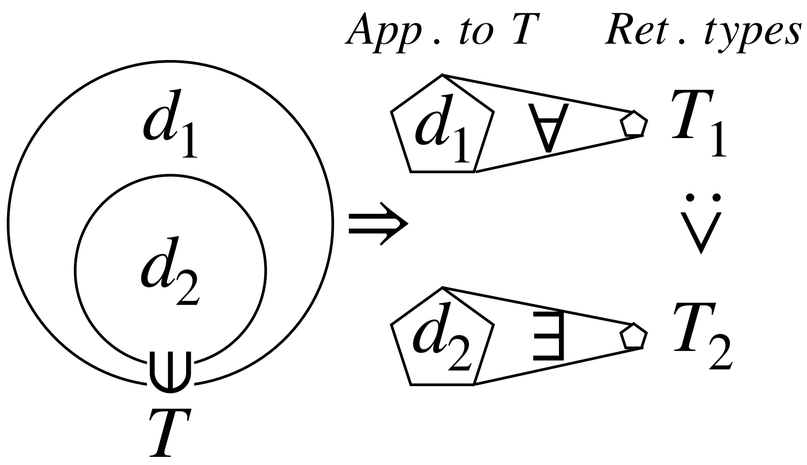
$$\frac{\neg\left(\Delta \vdash dom(d) \sqsubseteq dom(d')\right)}{\Delta \vdash d \text{ not duplicate of } d'}$$

$$\frac{\neg\left(\Delta \vdash dom(d') \sqsubseteq dom(d)\right)}{\Delta \vdash d \text{ not duplicate of } d'}$$



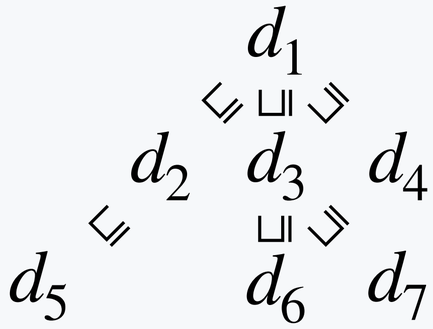
Meet rule

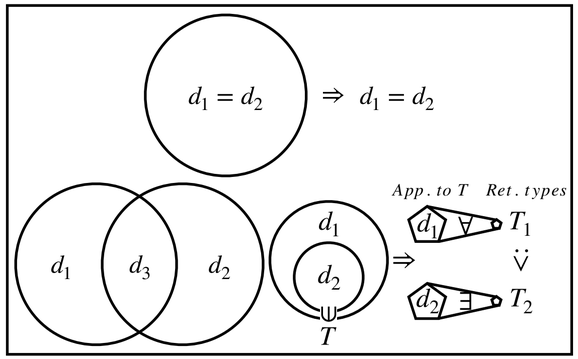
$$\frac{d_3 \in \{\overline{d}\} \quad name(d_1) = name(d_2) = name(d_3) \qquad \Delta \vdash dom(d_3) \equiv \left(dom(d_1) \sqcap dom(d_2)\right)}{\Delta \vdash d_1 \text{ meet } d_2 \text{ wrt } \{\overline{d}\} \text{ ok}}$$



$$\frac{arrow(d) = \forall[\![\overline{\kappa}]\!](\alpha \to \rho) \qquad \overline{\kappa = \{\overline{\chi}\} <: P <: \{\overline{\eta}\}} \qquad arrow(d') = \forall[\![\overline{\kappa'}]\!](\alpha' \to \rho') \qquad \overline{\kappa' = \{\overline{\chi'}\} <: Q <: \{\overline{\eta'}\}} \qquad distinct(\overline{P}, \overline{Q}) \qquad \Delta \vdash dom(d) \sqsubseteq dom(d') \qquad \Delta \vdash \forall[\![\overline{\kappa}]\!](\alpha \to \rho) \sqsubseteq \forall[\![\overline{\kappa}, \overline{\kappa'}]\!]((\alpha \sqcap \alpha') \to \rho')}{\Delta \vdash d \text{ return type wrt } d' \text{ ok}}$$

Return type rule



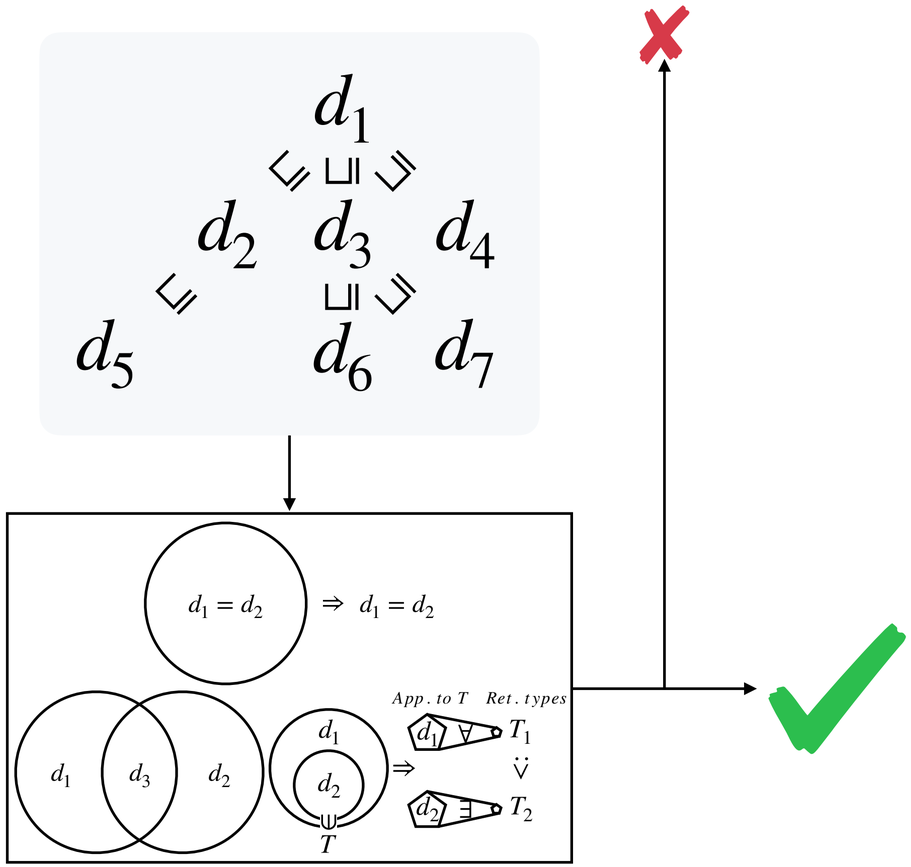



```
m(e)
```

Compile time

Run time

PLRG

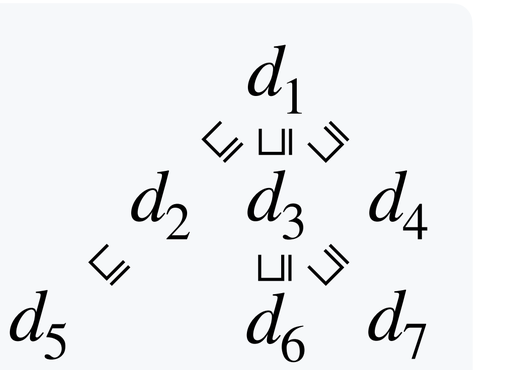


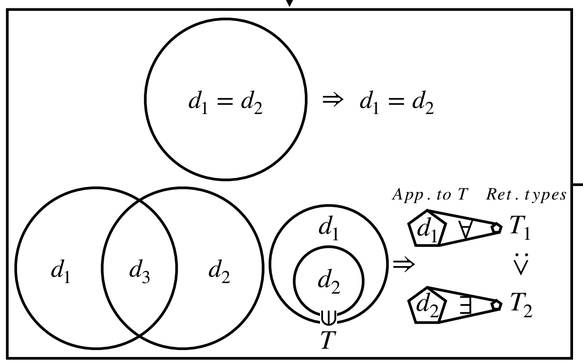

```
m(e)
```

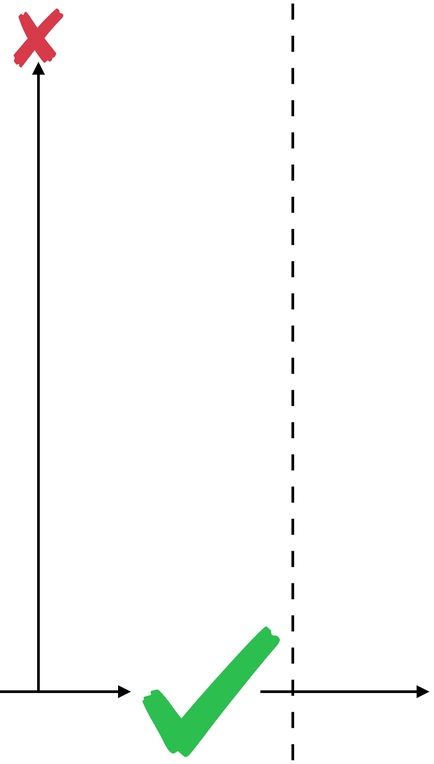
Compile time

Run time

PLRG





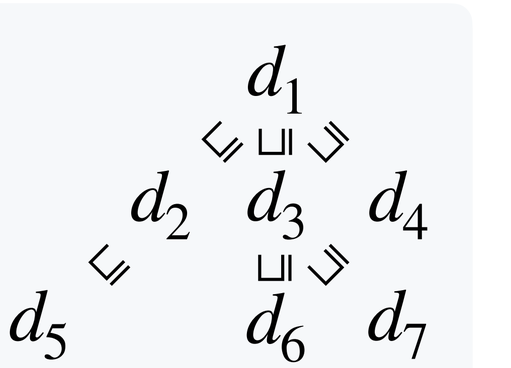


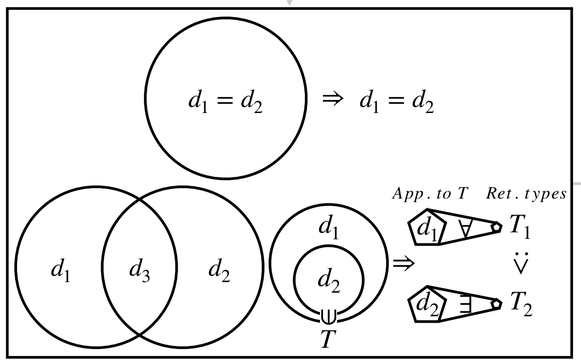

```
m(e)
```

Compile time

Run time

PLRG





d_1 d_2 d_3 d_4 d_5 d_6 d_7

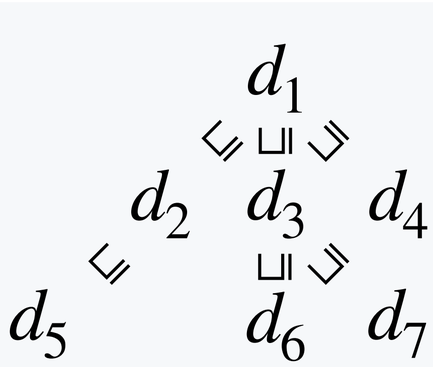
```
m(e)
```

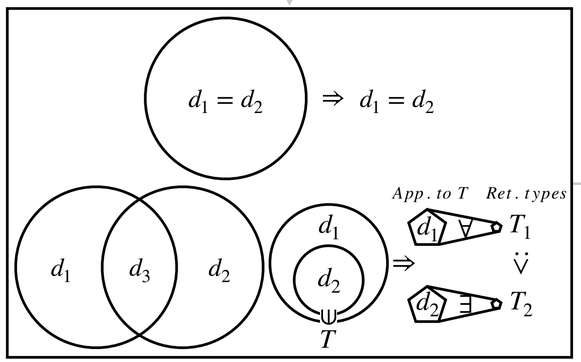
```
m(a): Tr
```

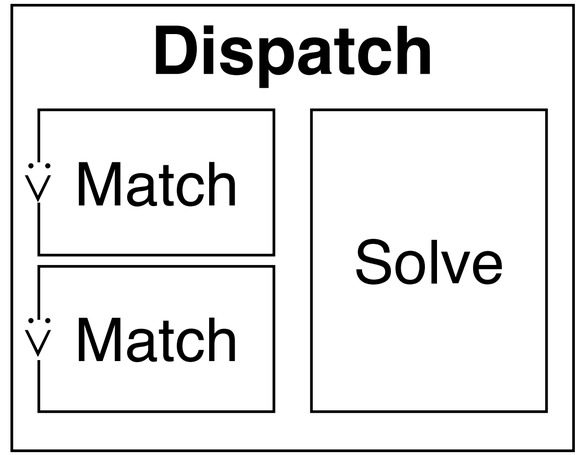
Compile time

Run time

PLRG







d_1 d_2 d_3 d_4 d_5 d_6 d_7

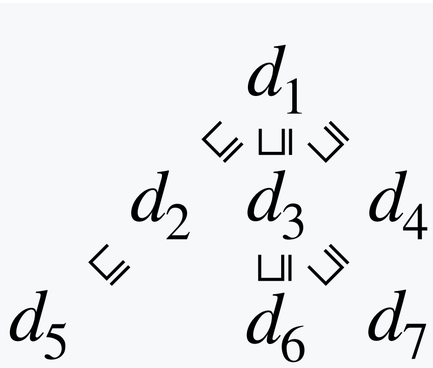
```
m(e)
```

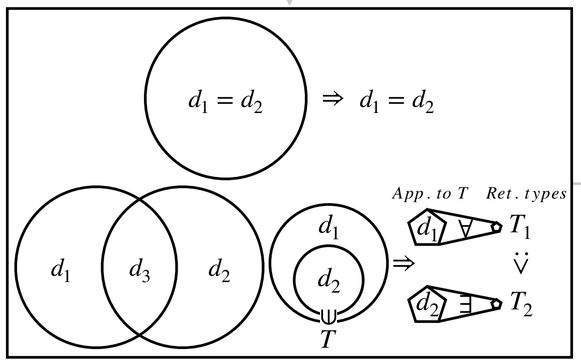
```
m(v): Tr
  ..
Ta
```

Compile time

Run time

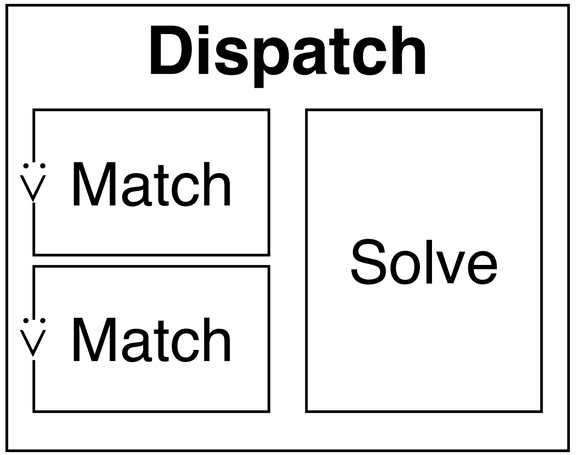
PLRG





`m(e)`

`m[P](Sa): Sr`



d_1 d_2 d_3 d_4 d_5 d_6 d_7

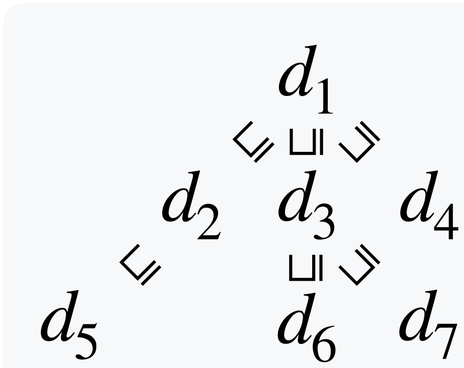
`m(v): Tr`

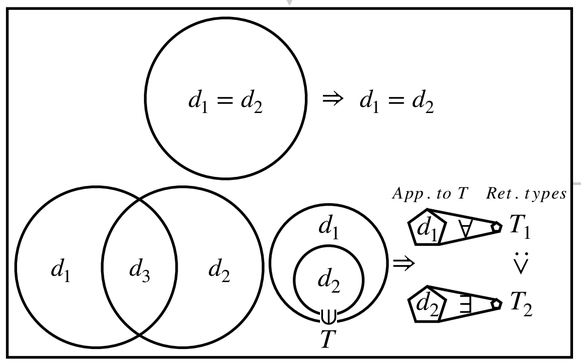
`Ta`

Compile time

Run time

PLRG

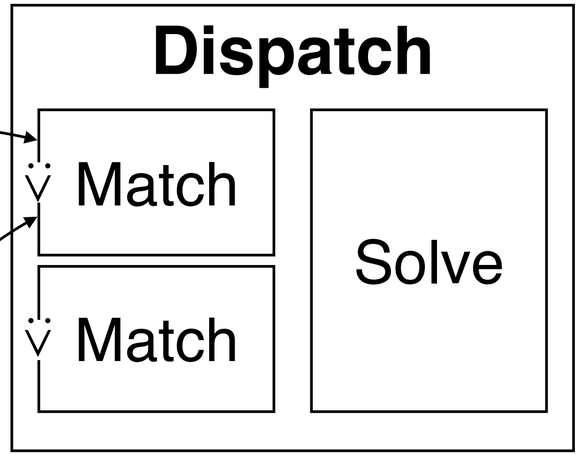




`m(e)`

Compile time

`m[P](Sa): Sr`

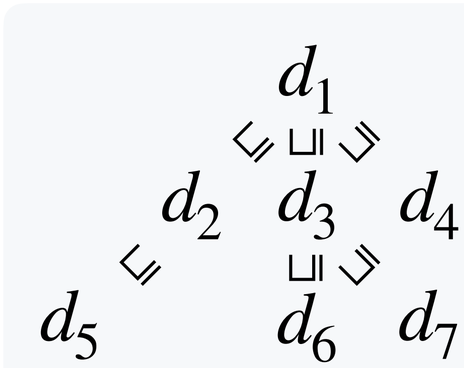


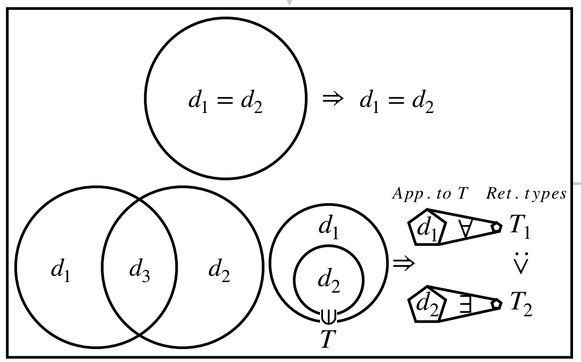
d_1 d_2 d_3 d_4 d_5 d_6 d_7

```
m(v): Tr
  ..
Ta
```

Run time

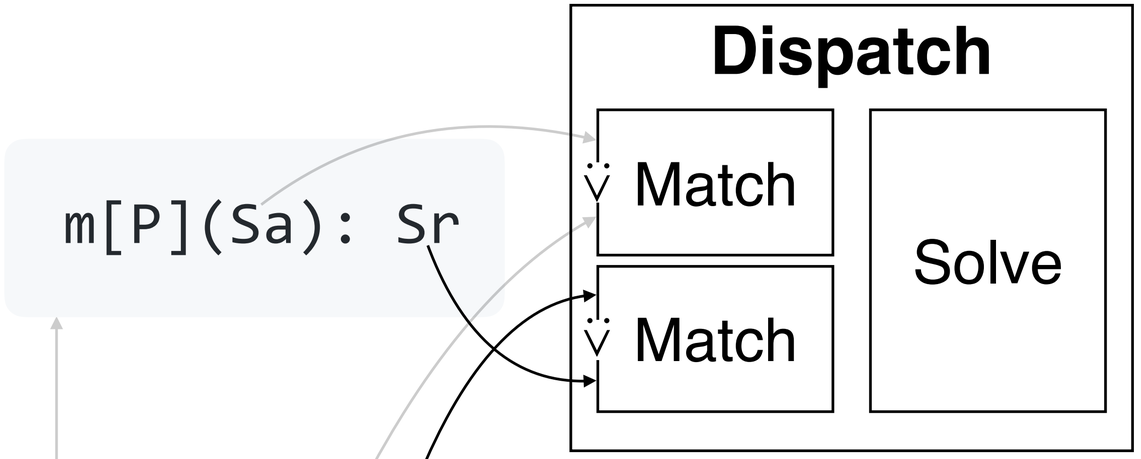
PLRG





`m(e)`

Compile time



`m[P](Sa): Sr`

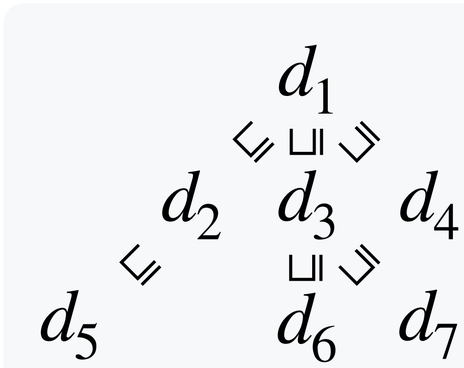
d_1 d_2 d_3 d_4 d_5 d_6 d_7

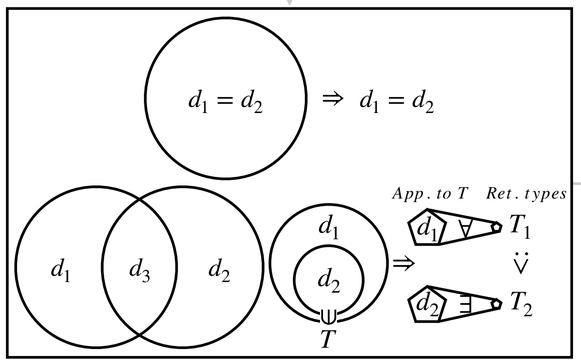
`m(v): Tr`

`Ta`

Run time

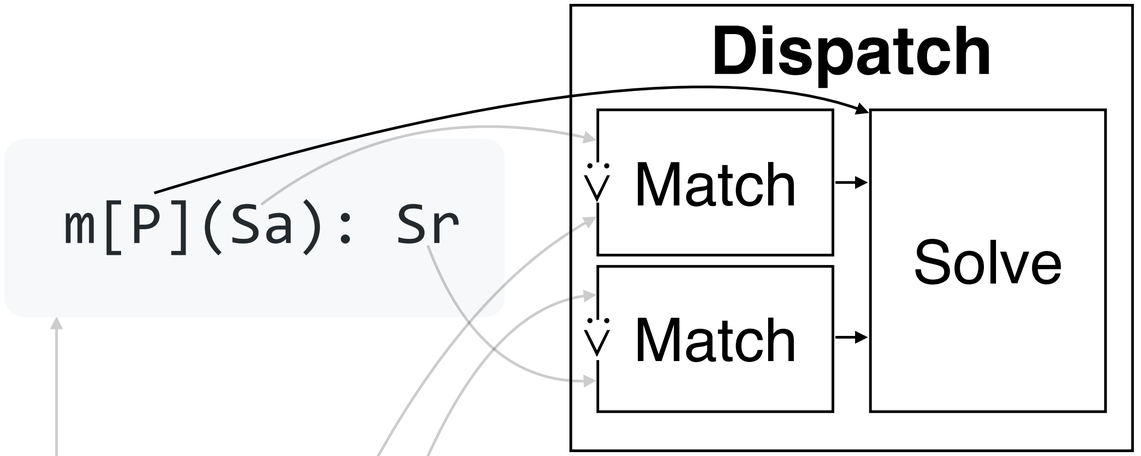
PLRG





`m(e)`

Compile time



`m[P](Sa): Sr`

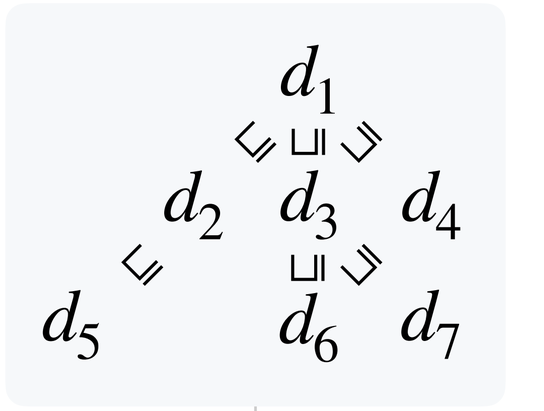
d_1 d_2 d_3 d_4 d_5 d_6 d_7

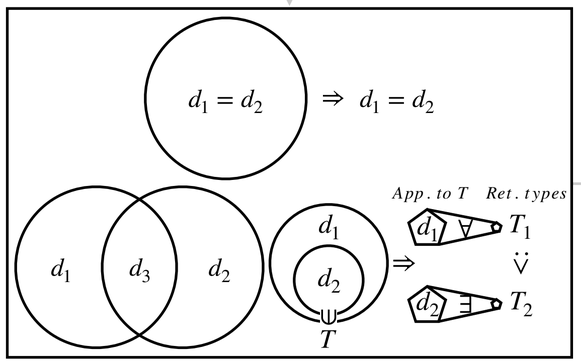
`m(v): Tr`

`Ta`

Run time

PLRG

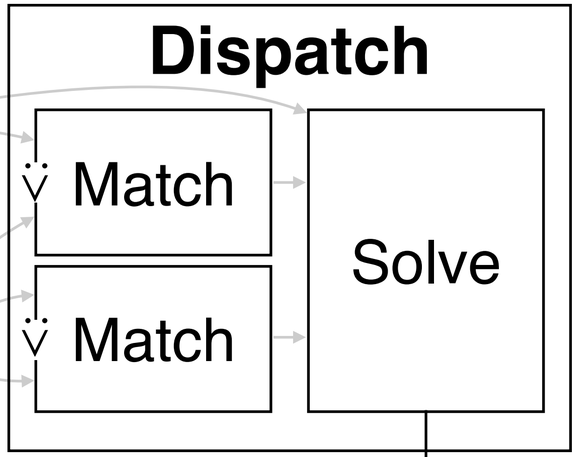




`m(e)`

Compile time

`m[P](Sa): Sr`



P=T

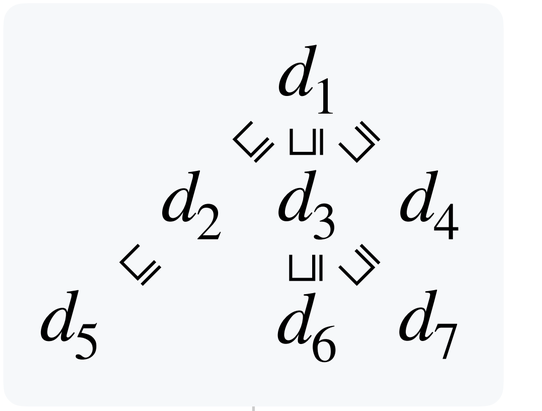
d_1 d_2 d_3 d_4 d_5 d_6 d_7

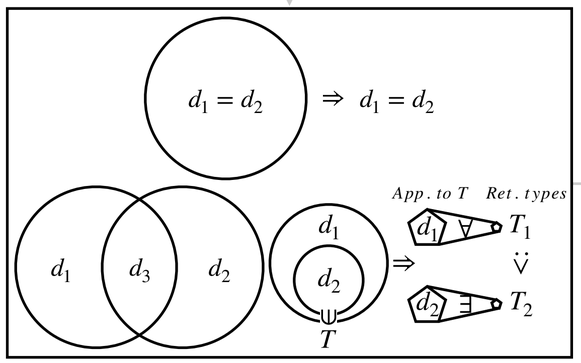
`m(v): Tr`
`..`
`Ta`

$[T/P]\ d_1$

Run time

PLRG

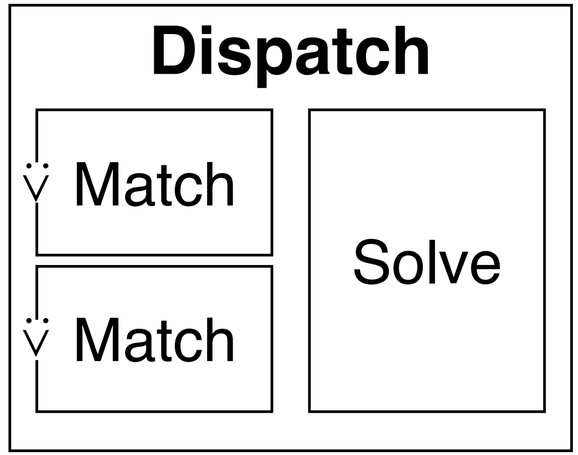




`m(e)`

Compile time

`m[P](Sa): Sr`



d_1 d_2 d_3 d_4 d_5 d_6 d_7

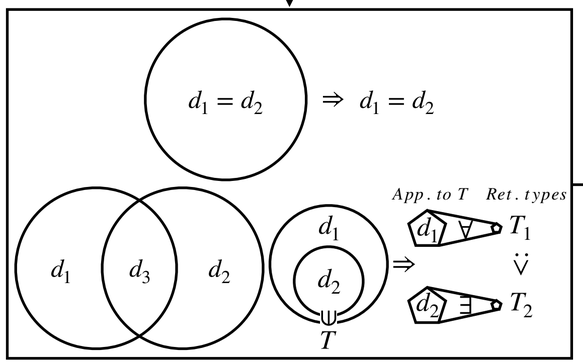
`m(v): Tr`

`Ta`

Run time

PLRG

```
makeArray[T](t: T): Array[T] = ...
makeArray[T](i: Number): Array[T] = ...
```




```
makeArray(i)
```

```
makeArray[T](i: Number): Array[T]
```

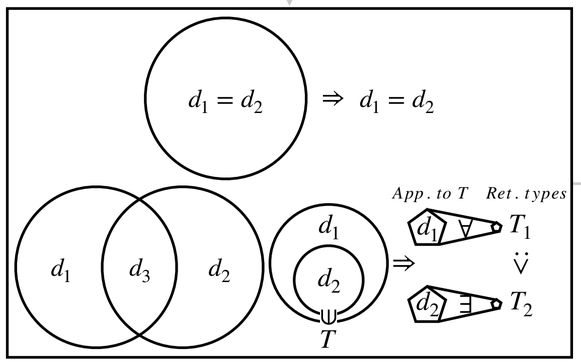
```
makeArray[T](t: T): Array[T]
```

Compile time

Run time

PLRG

```
makeArray[T](t: T): Array[T] = ...
makeArray[T](i: Number): Array[T] = ...
```



```
makeArray[T](i: Number): Array[T]
```

```
makeArray[T](t: T): Array[T]
```

```
makeArray(i)
```

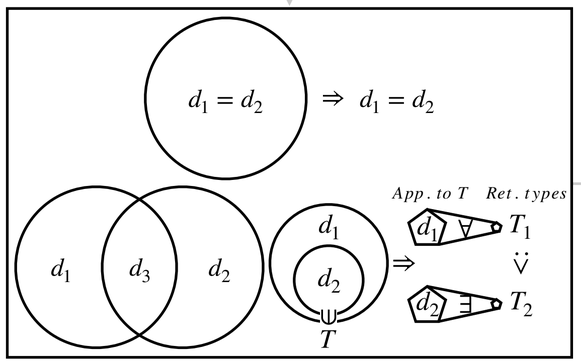
```
makeArray(i): Array[Object]
```

Compile time

Run time

PLRG

```
makeArray[T](t: T): Array[T] = ...
makeArray[T](i: Number): Array[T] = ...
```




```
makeArray[T](i: Number): Array[T]
```

```
makeArray[T](t: T): Array[T]
```

```
makeArray(i)
```

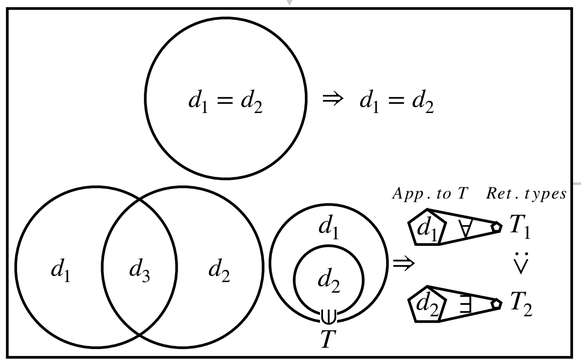
```
makeArray(1): Array[Object]
```

Compile time

Run time

PLRG

```
makeArray[T](t: T): Array[T] = ...
makeArray[T](i: Number): Array[T] = ...
```



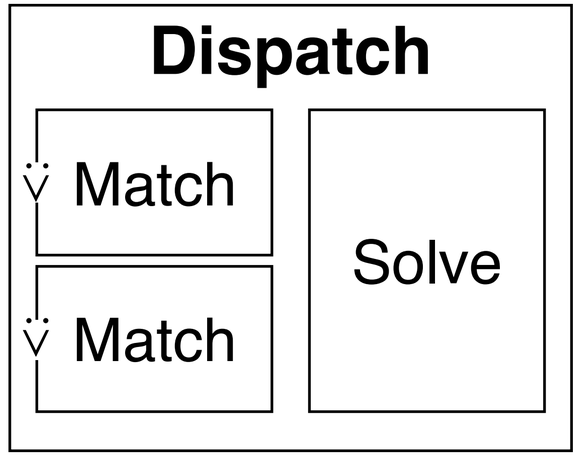

```
makeArray(i)
```

```
[T](Number): Array[T]
```

```
makeArray[T](i: Number): Array[T]     makeArray[T](t: T): Array[T]
```

```
makeArray(1): Array[Object]
```

Int

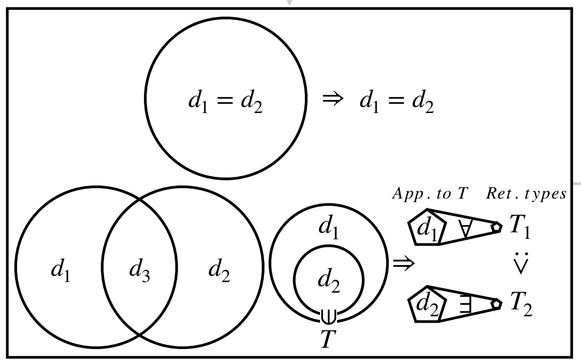


Compile time

Run time

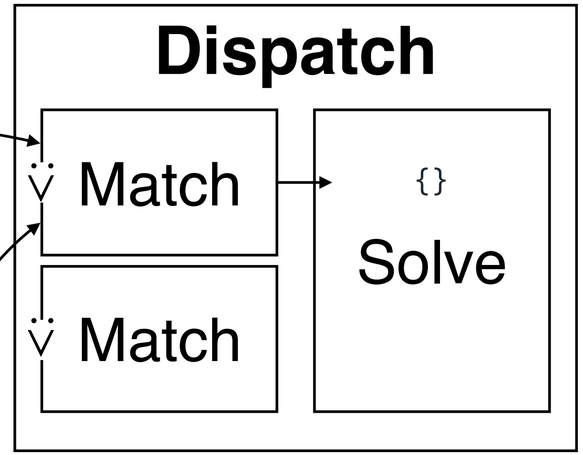
PLRG

```
makeArray[T](t: T): Array[T] = ...
makeArray[T](i: Number): Array[T] = ...
```




```
makeArray(i)
```

Compile time




```
[T](Number): Array[T]
```

```
makeArray[T](i: Number): Array[T]    makeArray[T](t: T): Array[T]
```

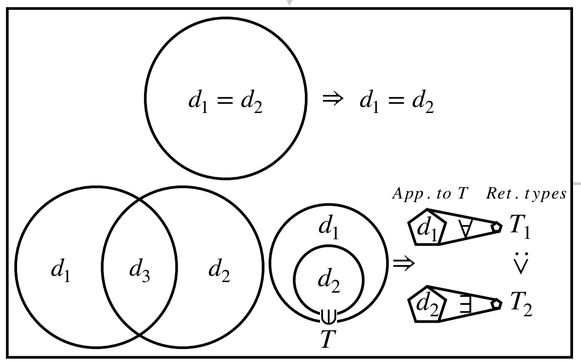
```
makeArray(1): Array[Object]
```

```
Int
```

Run time

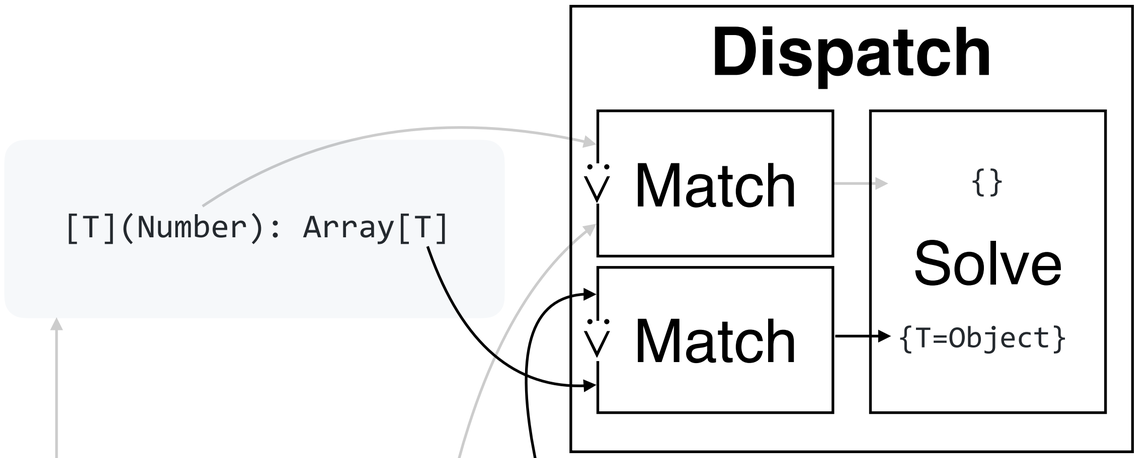
PLRG

```
makeArray[T](t: T): Array[T] = ...
makeArray[T](i: Number): Array[T] = ...
```




```
makeArray(i)
```

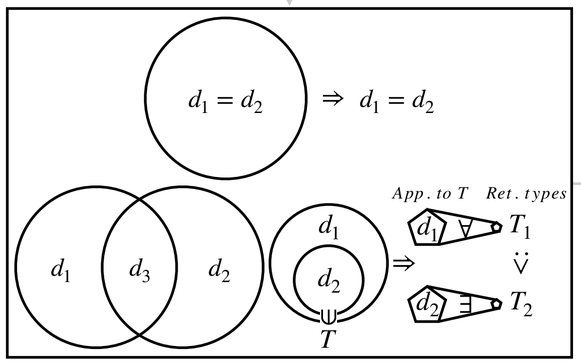
Compile time



Run time

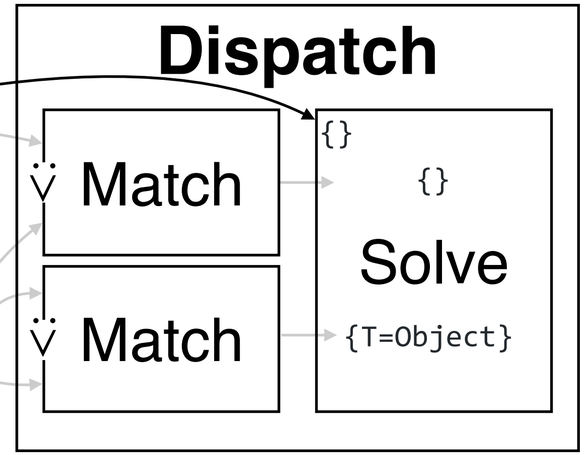
PLRG

```
makeArray[T](t: T): Array[T] = ...
makeArray[T](i: Number): Array[T] = ...
```




```
makeArray(i)
```

Compile time




```
[T](Number): Array[T]
```

```
makeArray[T](i: Number): Array[T]
```

```
makeArray[T](t: T): Array[T]
```

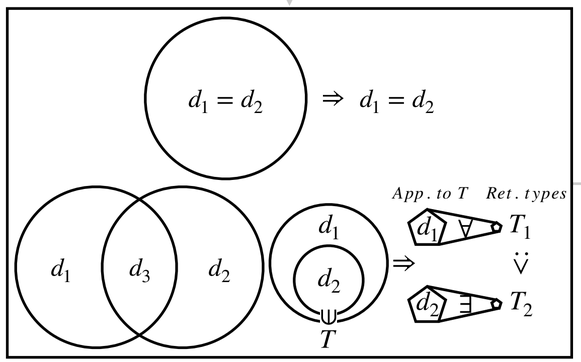
```
makeArray(1): Array[Object]
```

```
Int
```

Run time

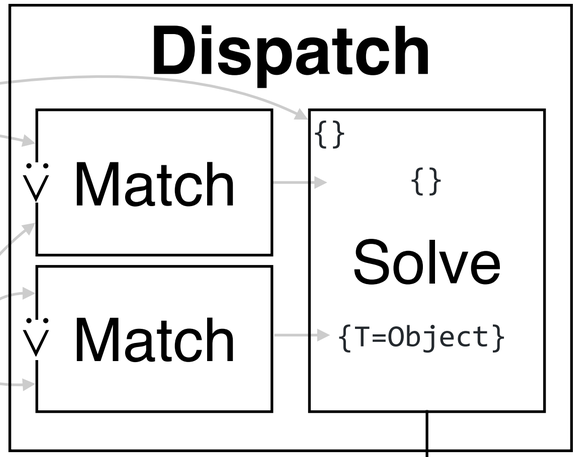
PLRG

```
makeArray[T](t: T): Array[T] = ...
makeArray[T](i: Number): Array[T] = ...
```




```
makeArray(i)
```

Compile time




```
[T](Number): Array[T]
```

```
T=Object
```

```
makeArray[T](i: Number): Array[T]
```

```
makeArray[T](t: T): Array[T]
```

```
makeArray(1): Array[Object]
```

```
Int
```

Run time

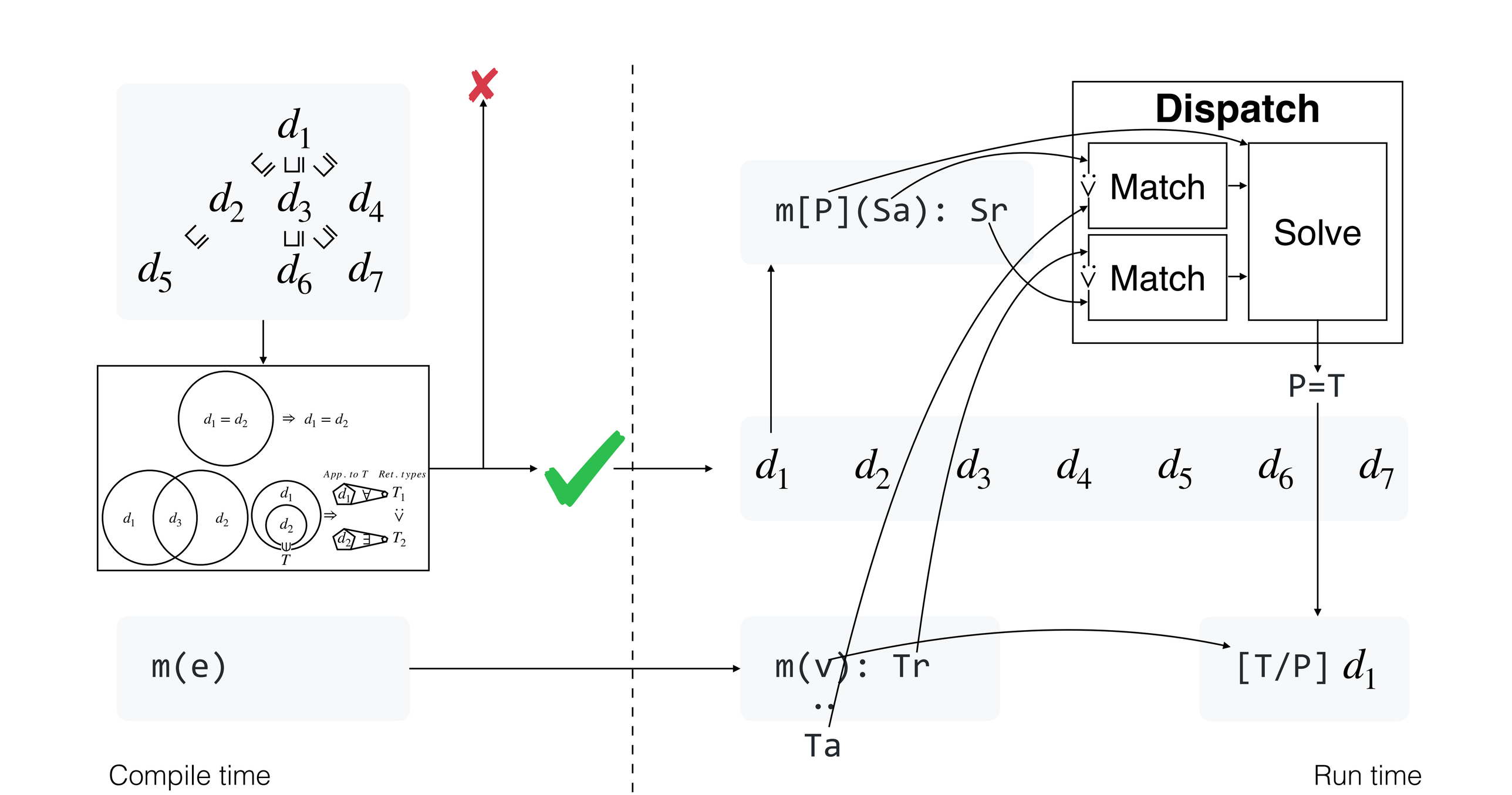
PLRG

Theorem

A well-formed program never results in an ambiguous method call.

Theorem

A well-formed program never results in a type error.


d_1
d_2 d_3 d_4
d_5 d_6 d_7
$d_1 = d_2 \Rightarrow d_1 = d_2$
App. to T Ret. types
T_1
T_2
T
m(e)
m[P](Sa): Sr
Dispatch
Match
Match
Solve
P=T
d_1 d_2 d_3 d_4 d_5 d_6 d_7
m(v): Tr
Ta
[T/P] d_1
Compile time
Run time

PLRG