Why Types Matter

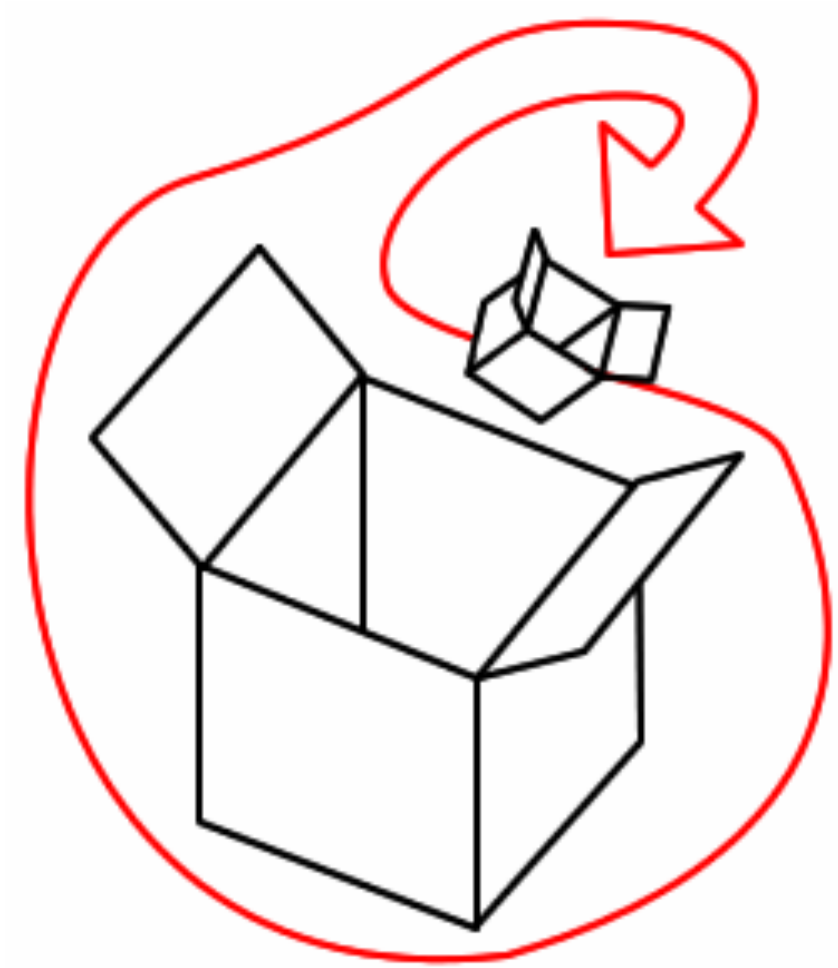
Zheng Gao

CREST, UCL

Russell's Paradox

Let *R* be the set of all sets that are not members of themselves. **Is *R* a member of itself**?

- If so, this contradicts with *R*'s definition
- If not, by definition, *R* should contain itself



Formalism in naïve set theory:

Let $R = \{x \mid x \notin x\}$, then $R \in R \iff R \notin R$

bar·ber
par·a·dox

The Barber Paradox

There is a town with a male barber who shaves all and only those men who do not shave themselves. **Who shaves the barber**?

The Barber Paradox

There is a town with a male barber who shaves all and only those men who do not shave themselves. **Who shaves the barber**?

- If the barber does not shave himself, according to the rule he must shave himself.

The Barber Paradox

There is a town with a male barber who shaves all and only those men who do not shave themselves. **Who shaves the barber**?

- If the barber does not shave himself, according to the rule he must shave himself.

- If he does shave himself, according to the rule he will not shave himself.

The Barber Paradox

There is a town with a male barber who shaves all and only those men who do not shave themselves. **Who shaves the barber**?

Naïve set theory contains contradiction

- If the ba[illegible]ding to the rule he r[illegible]
- If he does shave himself, according to the rule he will not shave himself.

Types to the Rescue

Constructs a hierarchy of types.

Any object is built only from those of higher types, which prevents circular referencing.

1) a barber as a citizen of the town, who shaves himself

and

2) a barber as a professional, who shaves others

are of different types.

Type Theory

An alternative to set theory as a foundation for mathematics, in which each term has a type

Simply typed λ-calculus is one of the many forms of type theory, which consists of

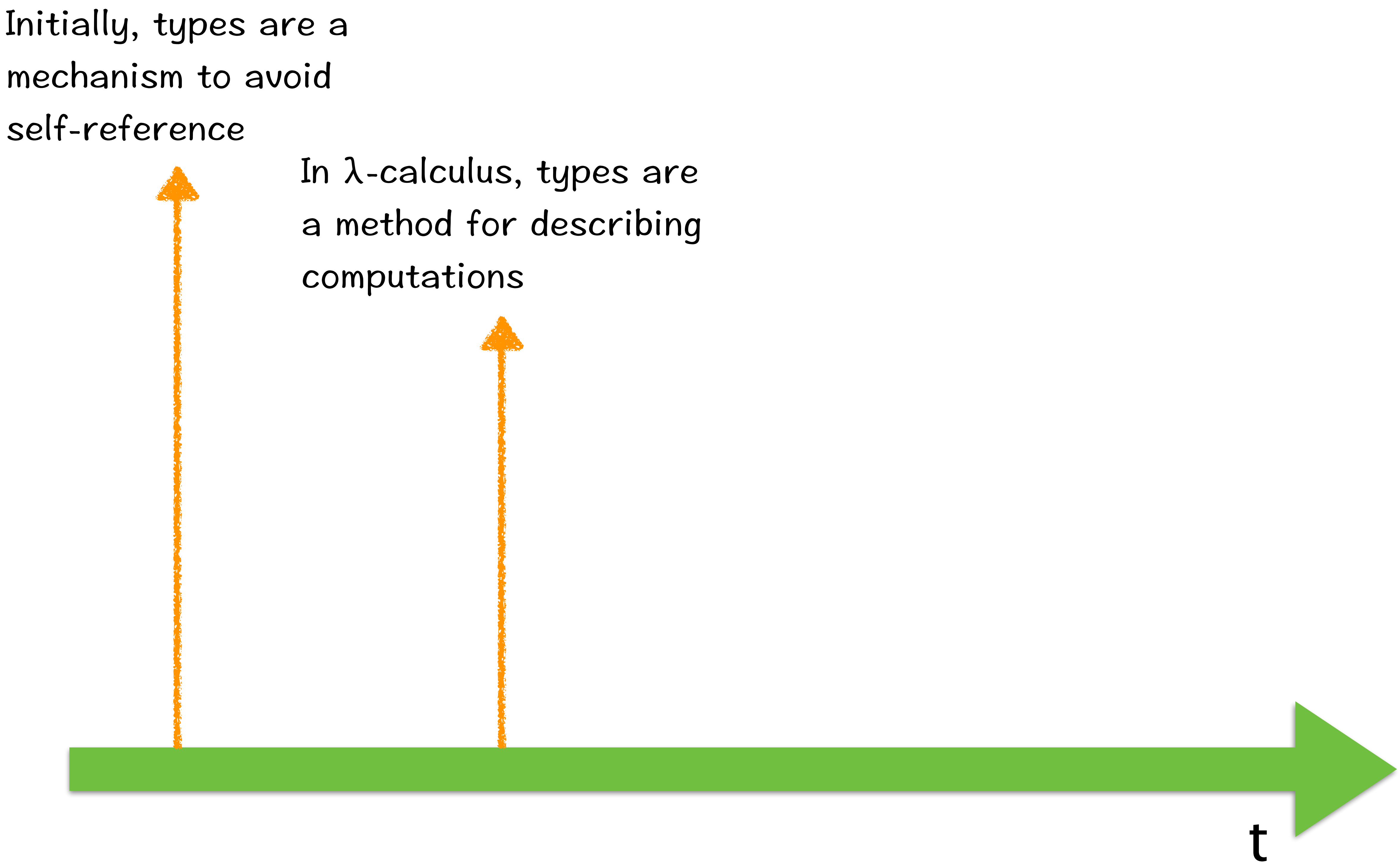
- Base types
- Only one type constructor, $\longrightarrow$, used to model the type of functions

The Evolution of Types

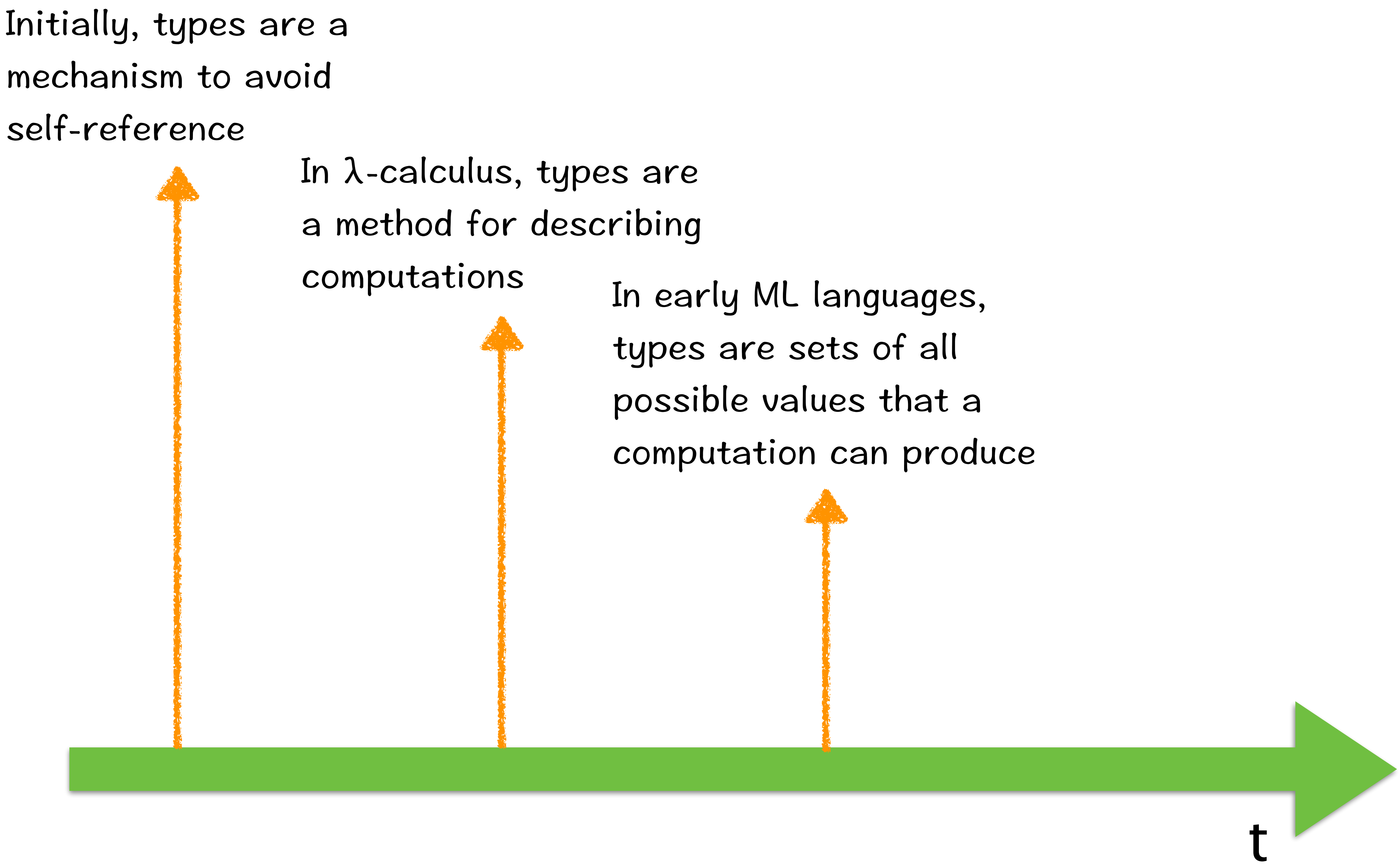
The Evolution of Types

Initially, types are a
mechanism to avoid
self-reference

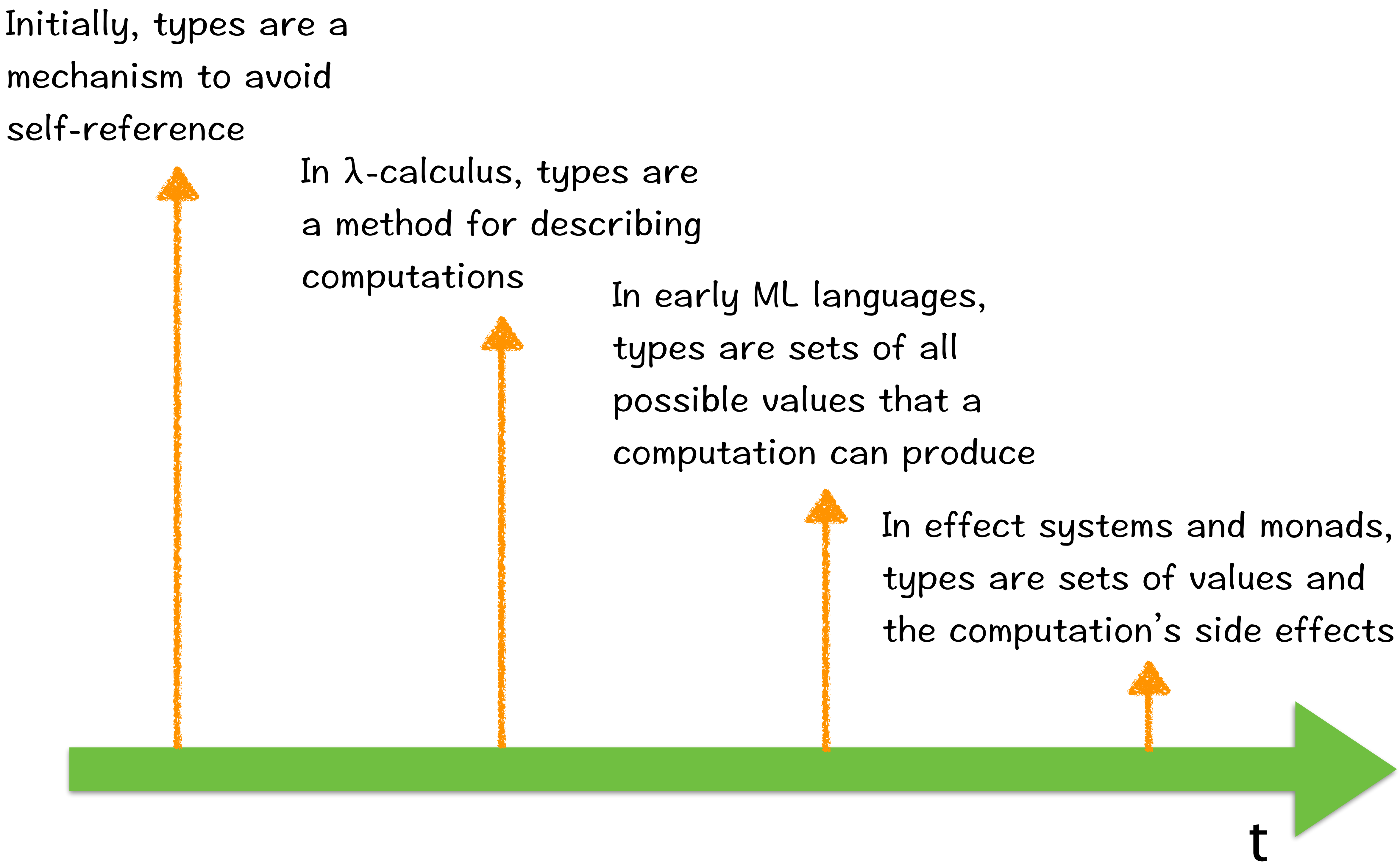
The Evolution of Types



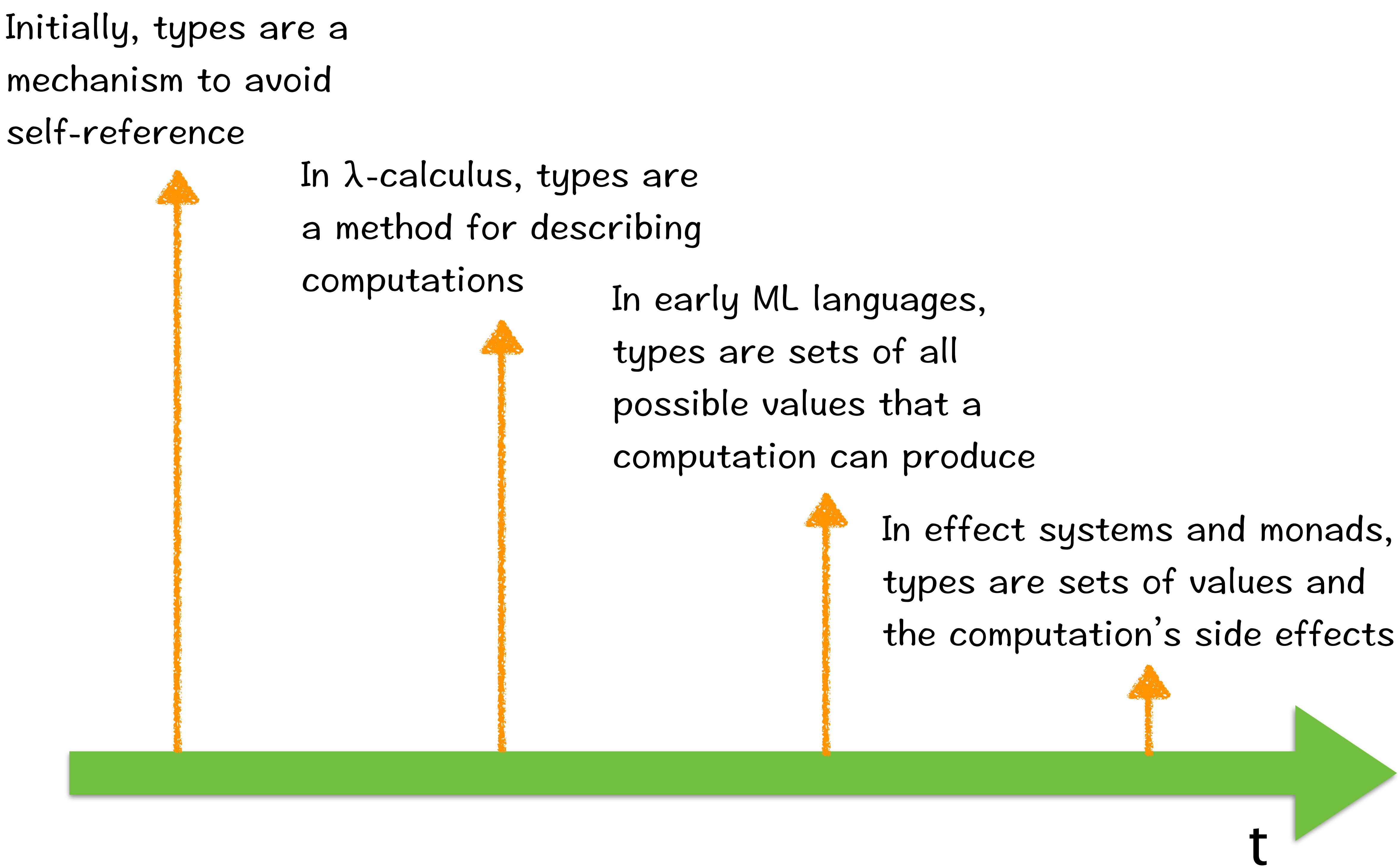
The Evolution of Types



The Evolution of Types



The Evolution of Types

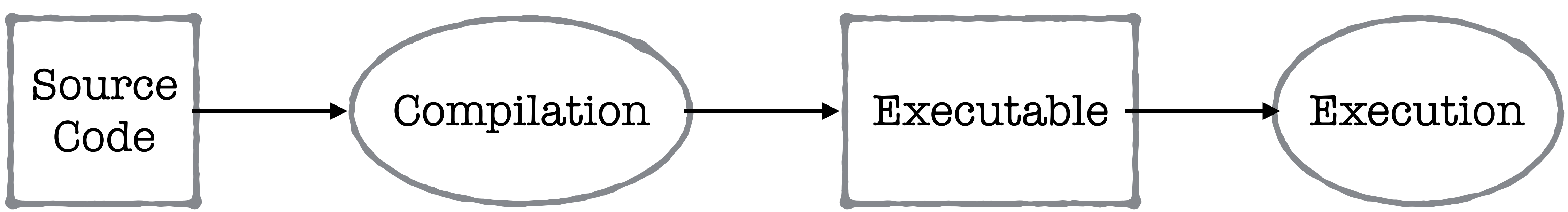


Type System

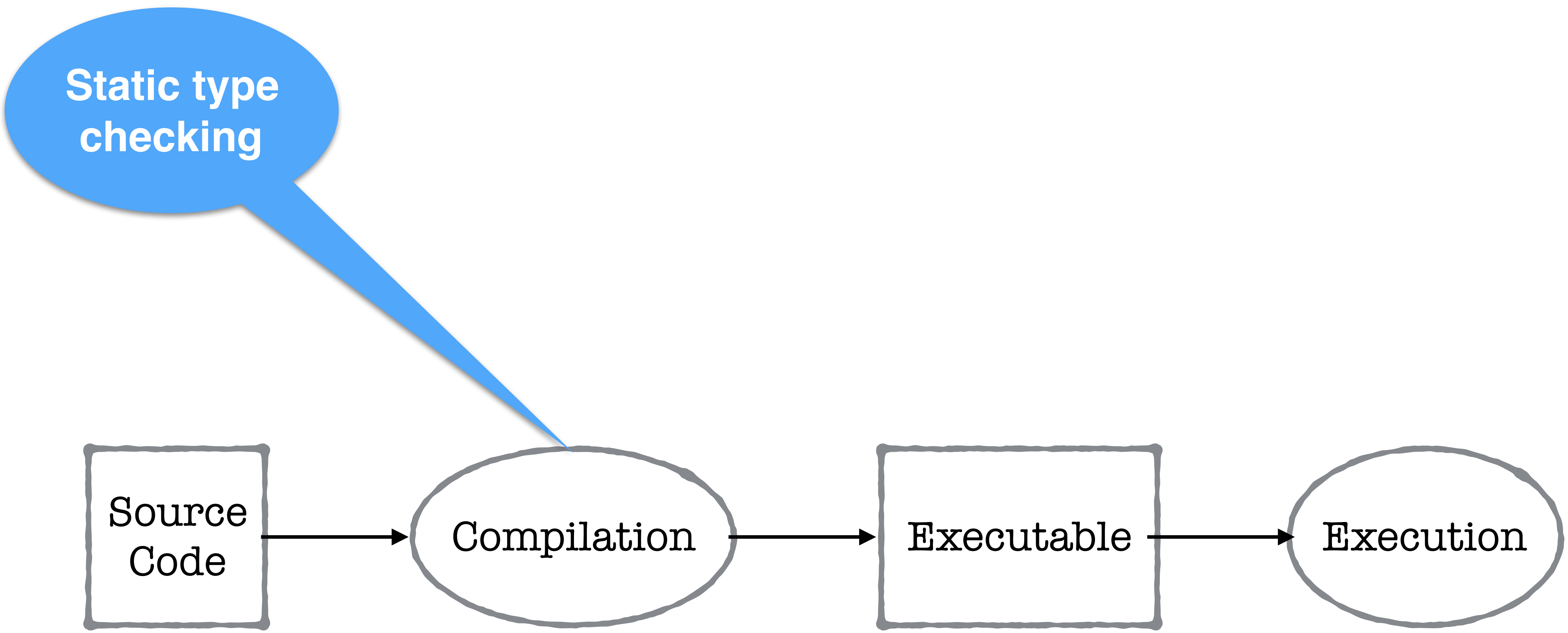
A tractable method that assigns types to syntactic phrases that compose a program, and automatically checks whether the usage of these phrases comply with their types

An over-approximation of the run-time behaviour of program terms

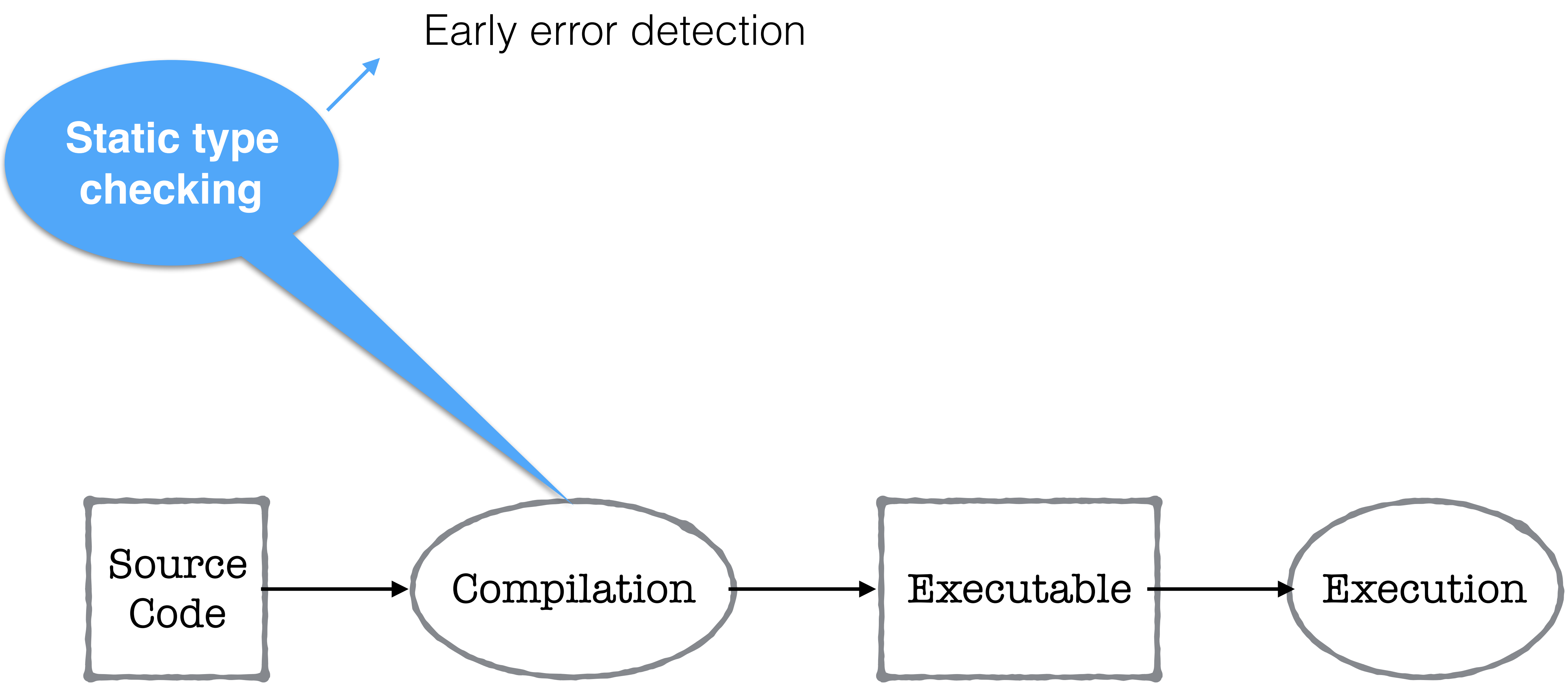
Static & Dynamic Type Checking



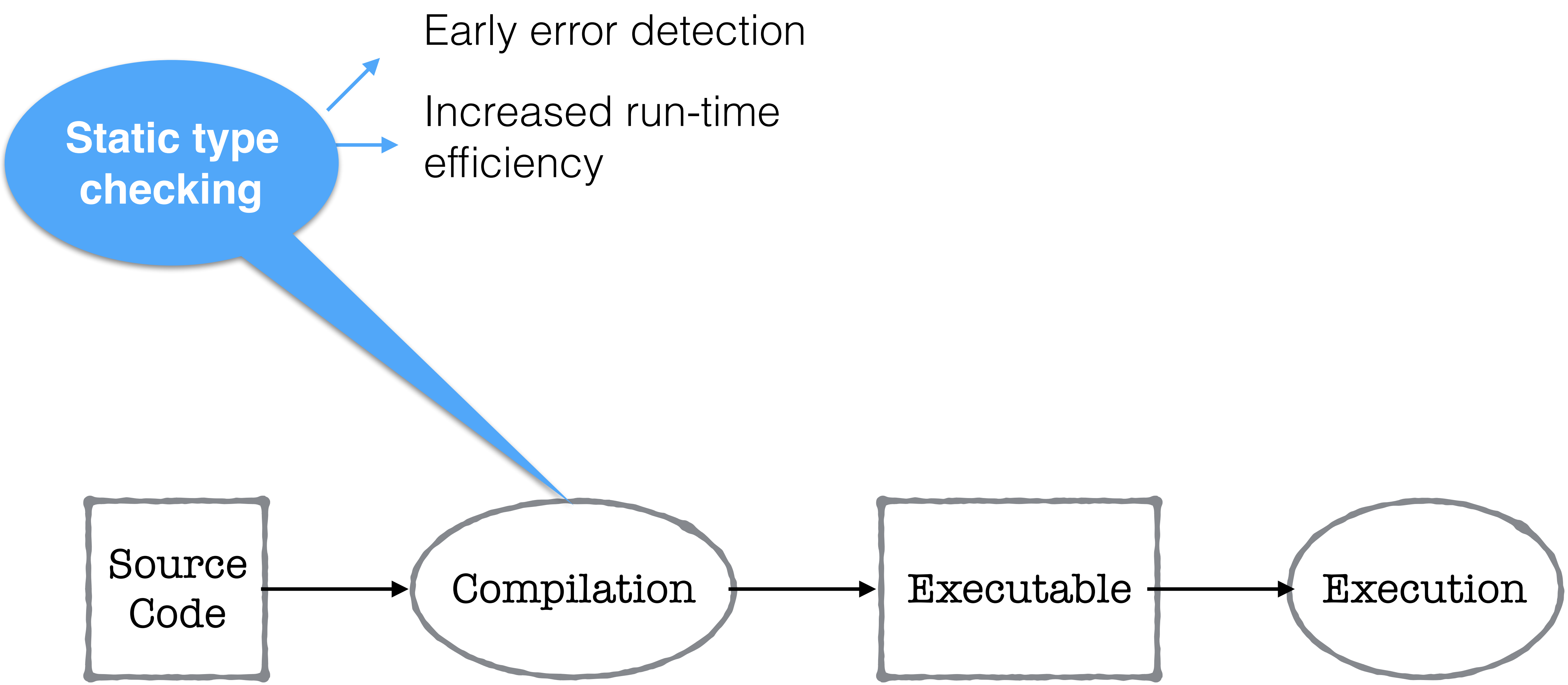
Static & Dynamic Type Checking



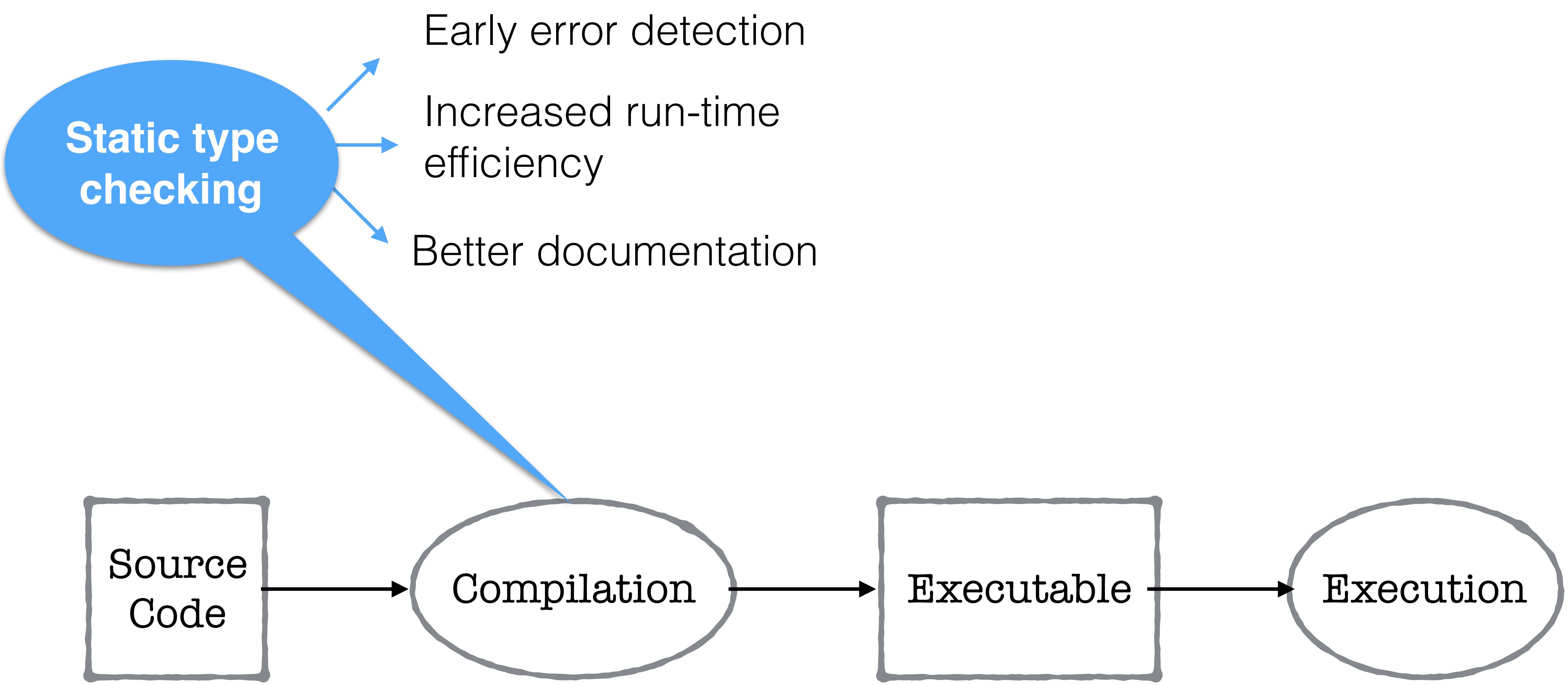
Static & Dynamic Type Checking



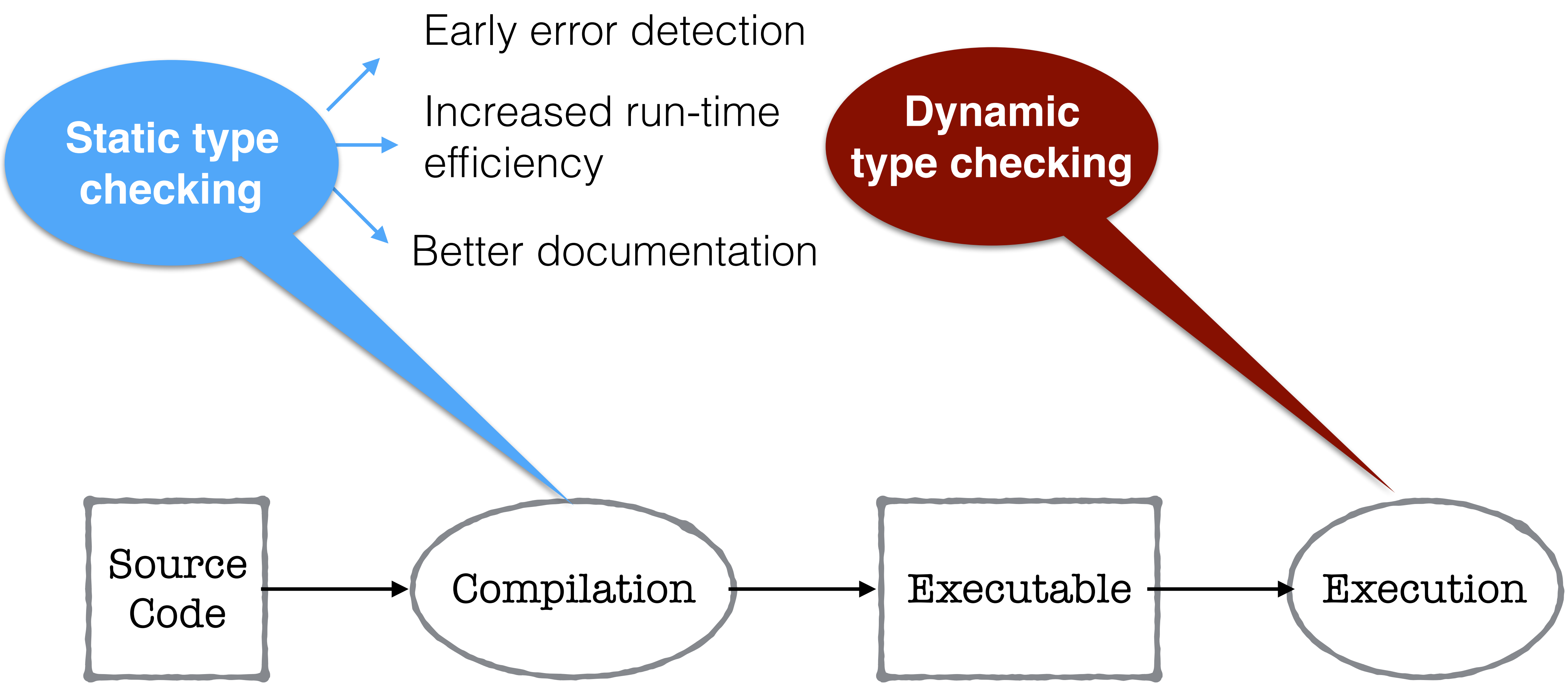
Static & Dynamic Type Checking



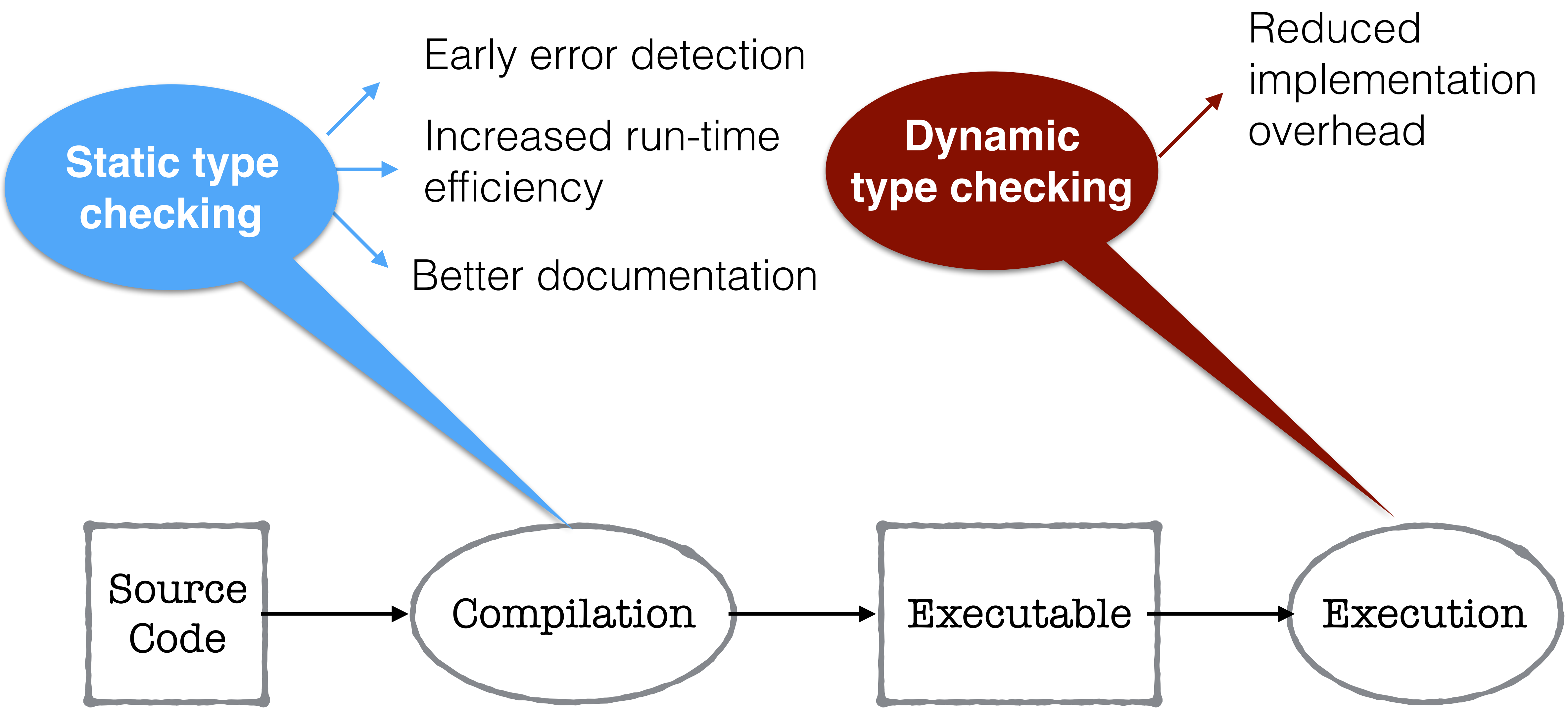
Static & Dynamic Type Checking



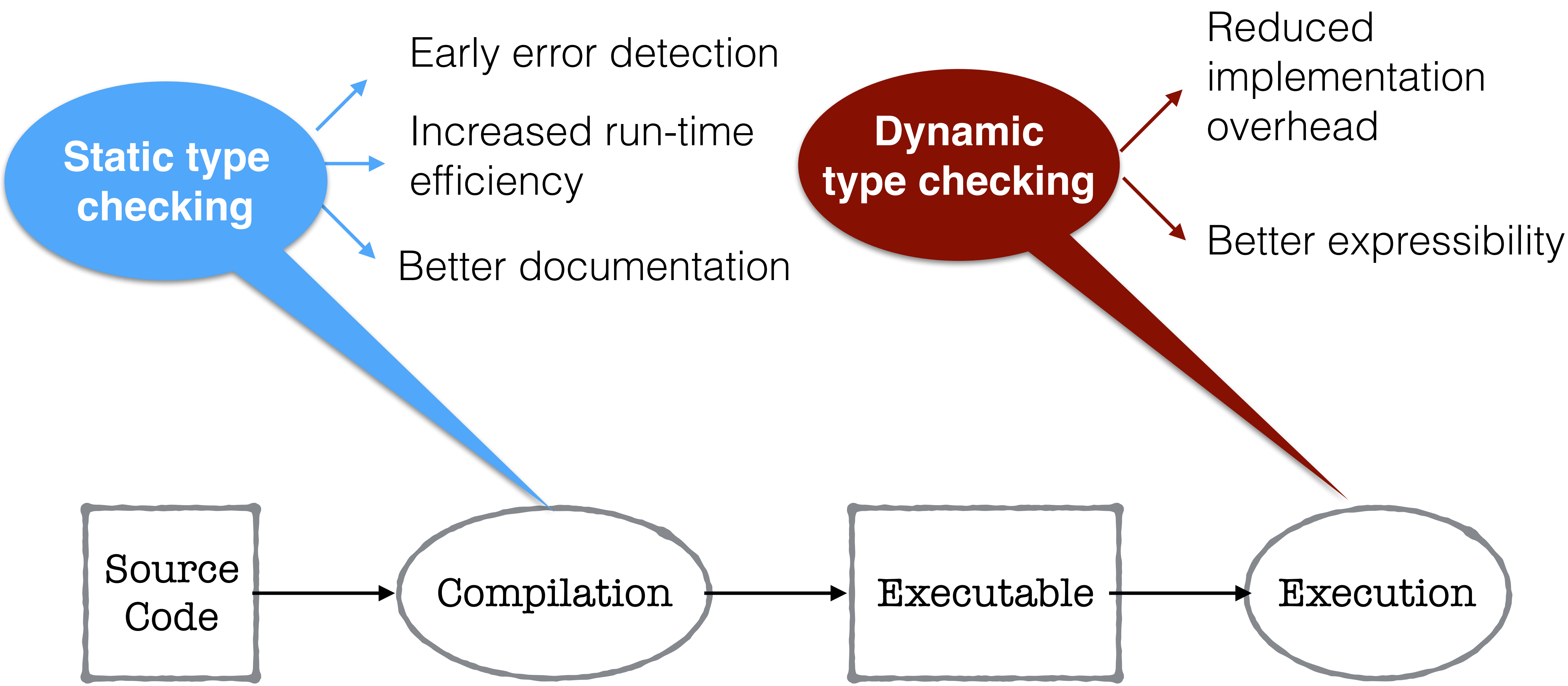
Static & Dynamic Type Checking



Static & Dynamic Type Checking



Static & Dynamic Type Checking



Why We Care

Generally, almost all real-world programming languages have type systems which offers multiple benefits.

Specifically for GI/GP, type systems have the promise to guide the search and avoid the construction of invalid individuals.

Java's Static Type Checking

Suppose we have:

```
class A {
    A me() {
        return this;
    }

    public void doA() {
        System.out.println("Do A");
    }
}

class B extends A {
    public void doB() {
        System.out.println("Do B");
    }
}

class C extends A{
    public void doC() {
        System.out.println("Do C");
    }
}
```

Java's Static Type Checking

```
new B().me().doB();

new B().me().doA();

((B) new B().me()).doB();

((C) new B().me()).doC();
```

Java's Static Type Checking

```
new B().me().doB();
```

Illegal. Compiler thinks new B().me() returns an object of class A, but at run-time, this returns an objects of class B.

```
new B().me().doA();
```

```
((B) new B().me()).doB();
```

```
((C) new B().me()).doC();
```

Java’s Static Type Checking

```java
new B().me().doB();
```

Illegal. Compiler thinks new B().me() returns an object of class A, but at run-time, this returns an objects of class B.

```java
new B().me().doA();
```

Legal.

```java
((B) new B().me()).doB();
```

```java
((C) new B().me()).doC();
```

Java's Static Type Checking

```
new B().me().doB();
```

Illegal. Compiler thinks new B().me() returns an object of class A, but at run-time, this returns an objects of class B.

```
new B().me().doA();
```

Legal.

```
((B) new B().me()).doB();
```

Legal.

```
((C) new B().me()).doC();
```

Java's Static Type Checking

`new B().me().doB();`	Illegal. Compiler thinks new B().me() returns an object of class A, but at run-time, this returns an objects of class B.
`new B().me().doA();`	Legal.
`((B) new B().me()).doB();`	Legal.
`((C) new B().me()).doC();`	Legal. But throws cast exception at run-time.

Hindley Milner's Type System

One of the most famous type systems for the typed λ-calculus with parametric polymorphism:

- A fast (nearly linear time) algorithm that automatically infer types of the constructs from their usage

- A set of typing rules, e.g.

$$\frac{\Gamma \vdash e_0 : \tau \rightarrow \tau' \qquad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0\ e_1 : \tau'} \quad [\mathrm{App}]$$

HM Example

Let us assume that we have a function myFunc of type:

$$\text{myFunc} : \text{ADT} \longrightarrow \text{int}$$

And we want to infer the type of a function someFunc

someFunc (x) + myFunc (x)

Step One

someFunc (x) + myFunc (x)

Step One

someFunc (x) + myFunc (x)

↓

x : α

Step Two

someFunc (x) + myFunc (x)

↓ ↓

$x : \alpha$ $\quad$ $ADT \longrightarrow int$

Step Two

someFunc (x) + myFunc (x)

$x : \alpha$ ADT $\longrightarrow$ int

α = ADT

Step Two

someFunc (x) + myFunc (x)

$x : \alpha$ ↓ $ADT \longrightarrow int$

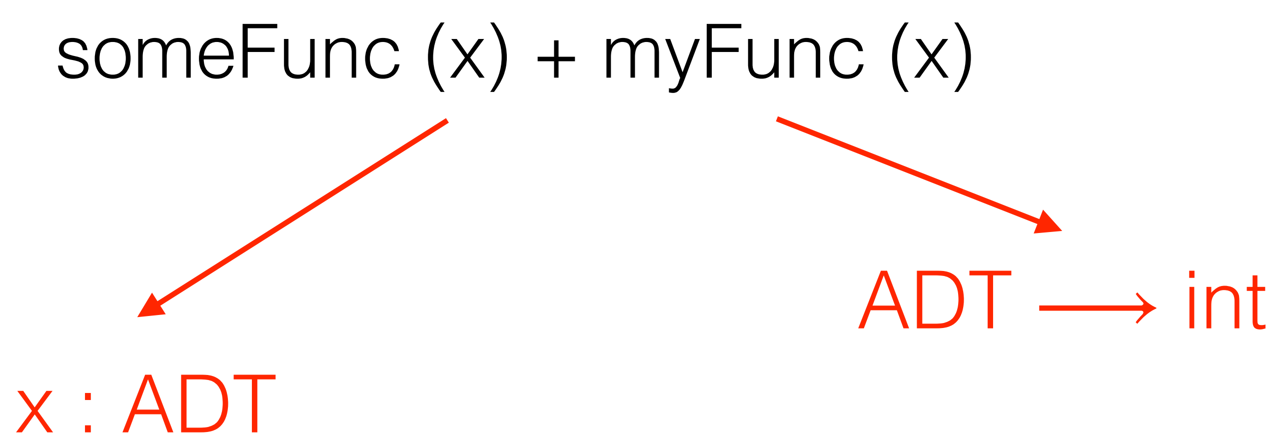
↓

$\alpha = ADT$

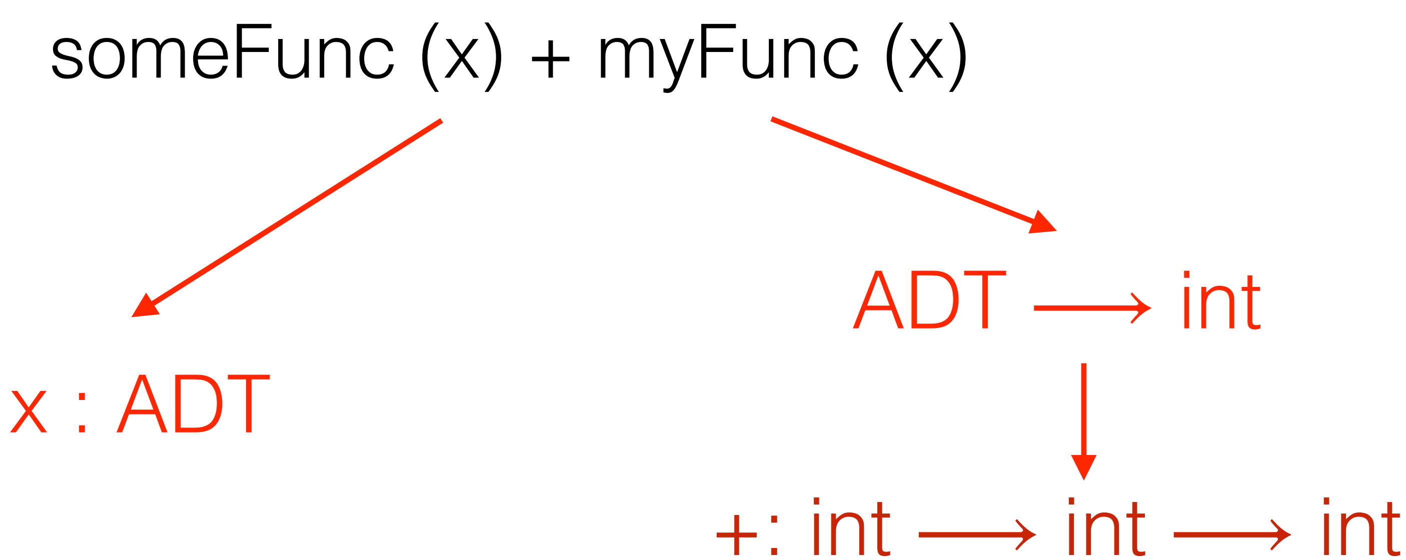
↓

$x : ADT$

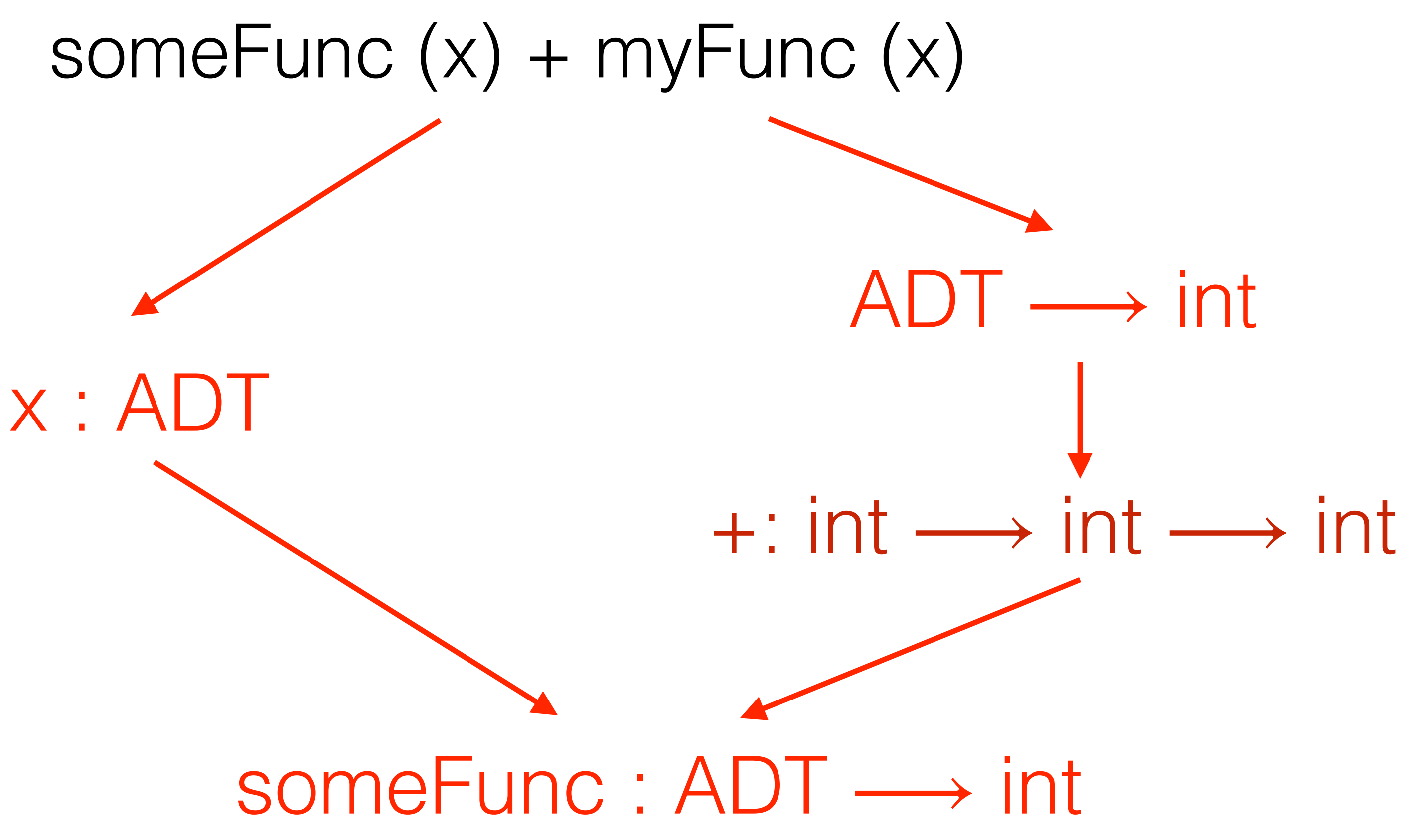
Step Three



Step Three



Step Three



Polymorphism

The provision of
a single interface
to entities of
different types

Polymorphism {

Parametric

Ad Hoc

Inclusion

Parametric Polymorphism

Generic programming in programming languages

```
class List<T> {
    class Node<T> {
        T elem;
        Node<T> next;
    }
    Node<T> head;
    int length() { ... }
}

List<B> map(Func<A,B> f, List<A> xs) {
    ...
}
```

Rank-N polymorphic function is a function whose parameters are Rank-(N-1) polymorphic

Ad Hoc Polymorphism

Function overloading in programming languages

```
function Add( x, y : Integer ) : Integer;
begin
    Add := x + y
end;

function Add( s, t : String ) : String;
begin
    Add := Concat( s, t )
end;
```

Inclusion Polymorphism

Inheritance creates inclusion polymorphism (subtyping)

```
abstract class Animal {
    abstract String talk();
}

class Cat extends Animal {
    String talk() {
        return "Meow!";
    }
}

class Dog extends Animal {
    String talk() {
        return "Woof!";
    }
}
```

Inclusion Polymorphism

Inheritance creates inclusion polymorphism (subtyping)

```
abstract class Animal {
    abstract String talk();
}

class Cat extends Animal {
    String talk() {
        return "Meow!";
    }
}

class Dog extends Animal {
    String talk() {
        return "Woof!";
    }
}
```

Cat < Animal

Dog < Animal

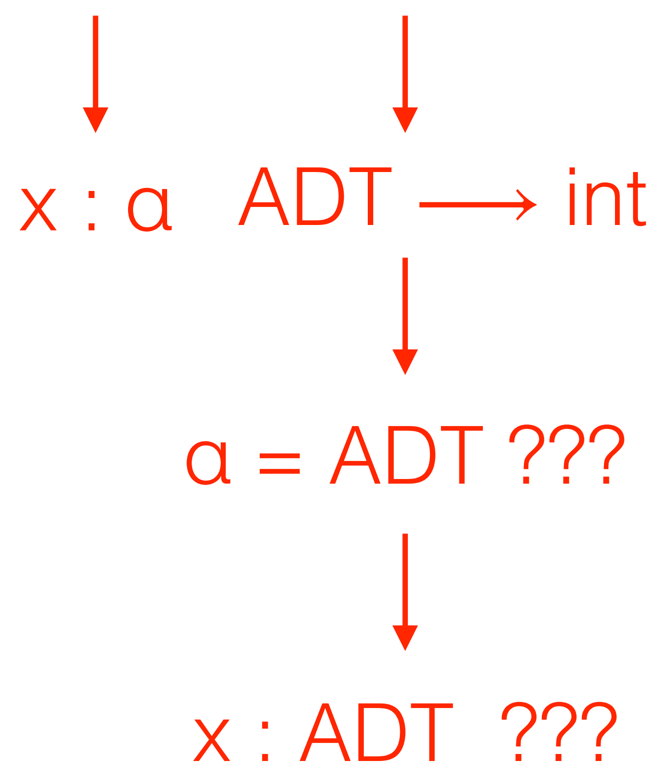
HM Limitations

- Limited to rank 1 parametric polymorphism
- Does not support ad hoc polymorphism
- No notion of subtyping

Limitation Example One

Suppose we have subtyping B < A, any function that takes arguments of type A is expected to takes arguments of type B as well.

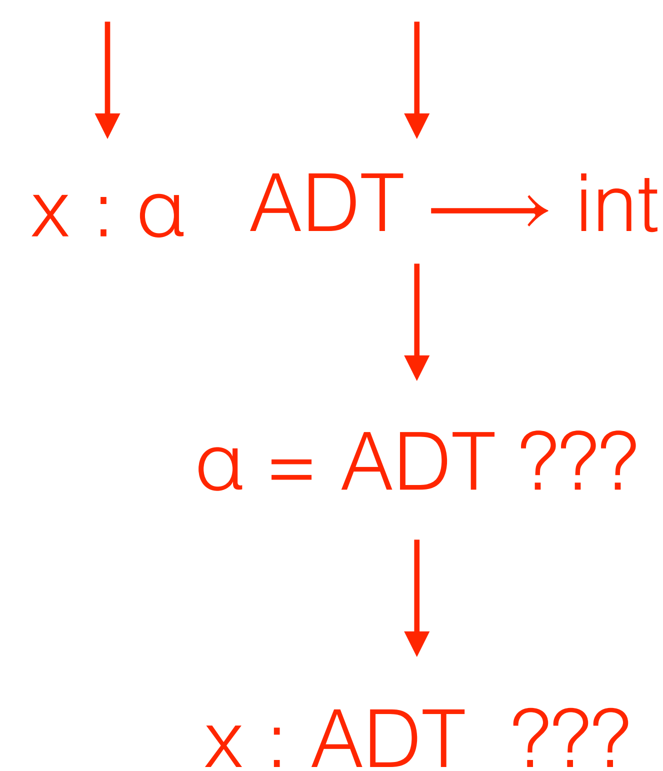
someFunc (x) + myFunc (x)



Limitation Example One

Suppose we have subtyping B < A, any function that takes arguments of type A is expected to takes arguments of type B as well.

someFunc (x) + myFunc (x)



α could be any subtype of ADT

Limitation Example Two

In HM, an assumption set may contain at most one typing assumption for an construct

The operator < , for example, has types:

$\text{char} \longrightarrow \text{char} \longrightarrow \text{bool}$

$\text{int} \longrightarrow \text{int} \longrightarrow \text{bool}$

But it does not have the type:

$\forall \alpha . \alpha \longrightarrow \alpha \longrightarrow \text{bool}$

So any single typing is either too narrow or too wide

Intersection Types

Allow a term to have multiple types by introducing a type constructor ∧, a universal type ω used for untypable constructs, and the following typing rules:

$$\frac{M : (\sigma_1 \wedge \sigma_2)}{M : \sigma_1} \quad \frac{M : (\sigma_1 \wedge \sigma_2)}{M : \sigma_2} \qquad (\wedge E)$$

$$\frac{M : \sigma_1 \quad M : \sigma_2}{M : (\sigma_1 \wedge \sigma_2)} \qquad (\wedge I)$$

In practice, intersection types enable **function overloading**.

Union Types

The dual notion of intersection types, which introduces a type constructor $\vee$ and similar typing rules.

In C / C++, union types are the construct **union**

Example

Consider the following code snippet in C++:

```
typedef struct {
    char c;
    bool b;
} ADT;

typedef union {
    int i;
    ADT a;
} unionType;

void foo(unionType x, int y) {};
void foo(unionType x, float y) {};
```

The type of function **foo** would be:

$$((\text{int} \vee \text{ADT}) \longrightarrow \text{int} \longrightarrow \text{void}) \wedge ((\text{int} \vee \text{ADT}) \longrightarrow \text{float} \longrightarrow \text{void})$$

Example

Consider the following code snippet in C++:

```
typedef struct {
    char c;
    bool b;
} ADT;

typedef union {
    int i;
    ADT a;
} unionType;

void foo(unionType x, int y) {};
void foo(unionType x, float y) {};
```

The type of function **foo** would be:

$$((\text{int} \vee \text{ADT}) \longrightarrow \text{int} \longrightarrow \text{void}) \wedge ((\text{int} \vee \text{ADT}) \longrightarrow \text{float} \longrightarrow \text{void})$$

Example

Consider the following code snippet in C++:

```
typedef struct {
    char c;
    bool b;
} ADT;

typedef union {
    int i;
    ADT a;
} unionType;

void foo(unionType x, int y) {};
void foo(unionType x, float y) {};
```

The type of function **foo** would be:

$$((\text{int} \vee \text{ADT}) \longrightarrow \text{int} \longrightarrow \text{void}) \wedge ((\text{int} \vee \text{ADT}) \longrightarrow \text{float} \longrightarrow \text{void})$$

Example

Consider the following code snippet in C++:

```
typedef struct {
    char c;
    bool b;
} ADT;

typedef union {
    int i;
    ADT a;
} unionType;

void foo(unionType x, int y) {};
void foo(unionType x, float y) {};
```

The type of function **foo** would be:

$$((\text{int} \vee \text{ADT}) \longrightarrow \text{int} \longrightarrow \text{void}) \wedge ((\text{int} \vee \text{ADT}) \longrightarrow \text{float} \longrightarrow \text{void})$$

Example

Consider the following code snippet in C++:

```
typedef struct {
    char c;
    bool b;
} ADT;

typedef union {
    int i;
    ADT a;
} unionType;

void foo(unionType x, int y) {};
void foo(unionType x, float y) {};
```

The type of function **foo** would be:

$$((\text{int} \vee \text{ADT}) \longrightarrow \text{int} \longrightarrow \text{void}) \wedge ((\text{int} \vee \text{ADT}) \longrightarrow \text{float} \longrightarrow \text{void})$$

Retype

A general tool that automatically replaces certain types, together with the corresponding operations if necessary, of a program with new ones.

Potential Applications

Reducing energy consumption

Precision tracking and improvement for FP programs

New mutation operators in GI/GP

Taint analysis

Symbolic execution

Auto-transplantation

Intersection Types in Retype

We use intersection types to cleanly model function overloading, because Retype may generate new overloads of an existing operator.

Consider the following code snippet:

```
int main() {
    int a, b;
    b = a + 2;
    a = b + ext_func(a);
    return 0;
}
```

<u>Assumption</u>: an external function ext_func of type int $\longrightarrow$ int

<u>Objective</u>: retype int to ADT

Intersection Types in Retype

We use intersection types to cleanly model function overloading, because Retype may generate new overloads of an existing operator.

Consider the following code snippet:

Before retyping

```
int main() {
    int a, b;
    b = a + 2;
    a = b + ext_func(a);
    return 0;
}
```

Assumption: an external function ext_func of type int ⟶ int

Objective: retype int to ADT

Intersection Types in Retype

We use intersection types to cleanly model function overloading, because Retype may generate new overloads of an existing operator.

Consider the following code snippet:

Before retyping

```
int main() {
    int a, b;
    b = a + 2;
    a = b + ext_func(a);
    return 0;
}
```

$+ : \text{int} \longrightarrow \text{int} \longrightarrow \text{int}$

<u>Assumption</u>: an external function ext_func of type int $\longrightarrow$ int

<u>Objective</u>: retype int to ADT

Intersection Types in Retype

We use intersection types to cleanly model function overloading, because Retype may generate new overloads of an existing operator.

Consider the following code snippet:

Before retyping

```
int main() {
    int a, b;
    b = a + 2;
    a = b + ext_func(a);
    return 0;
}
```

$+: int \longrightarrow int \longrightarrow int$

$+: int \longrightarrow int \longrightarrow int$

<u>Assumption</u>: an external function ext_func of type int ⟶ int

<u>Objective</u>: retype int to ADT

Intersection Types in Retype

We use intersection types to cleanly model function overloading, because Retype may generate new overloads of an existing operator.

Consider the following code snippet:

```
int main() {
    ADT a, b;
    b = a + 2;
    a = b + ext_func(a);
    return 0;
}
```

After retyping

+: ADT ⟶ ADT ⟶ ADT

+: ADT ⟶ **int** ⟶ ADT

<u>Assumption</u>: an external function ext_func of type int ⟶ int

<u>Objective</u>: retype int to ADT