Efficient Fail-Fast Dynamic Subtype Checking

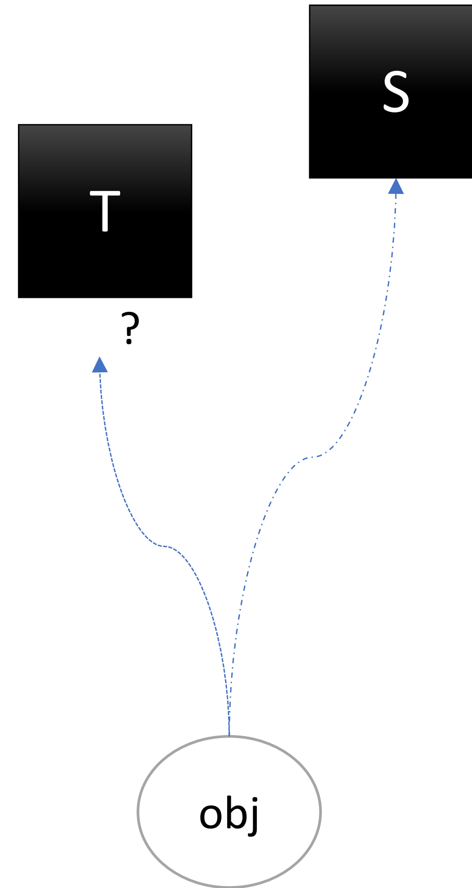
Rohan Padhye and Koushik Sen

UC Berkeley

VMIL 2019

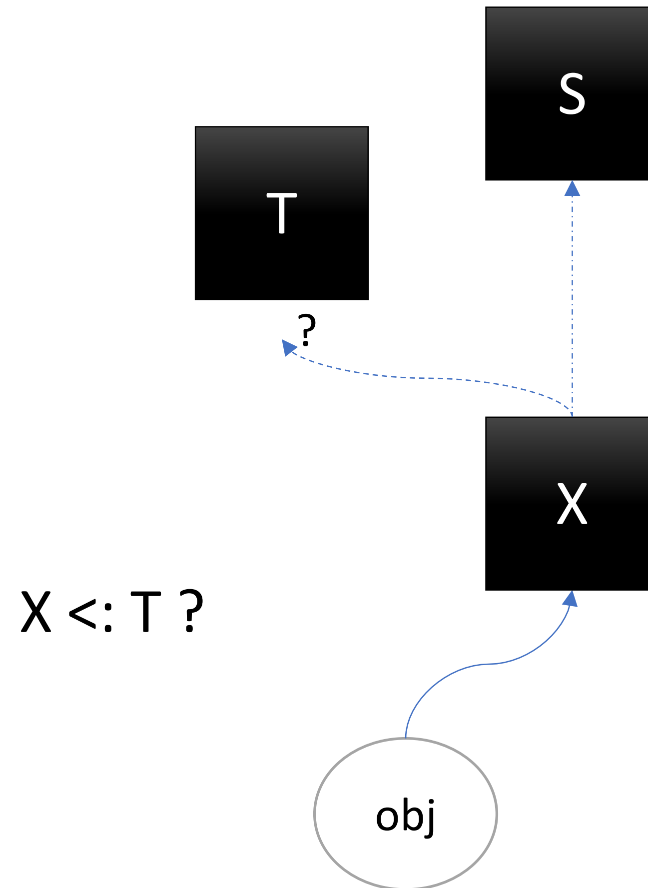
Dynamic Subtype Checking

```
S obj = .....

if (obj instance of T) {
    ....
}
```

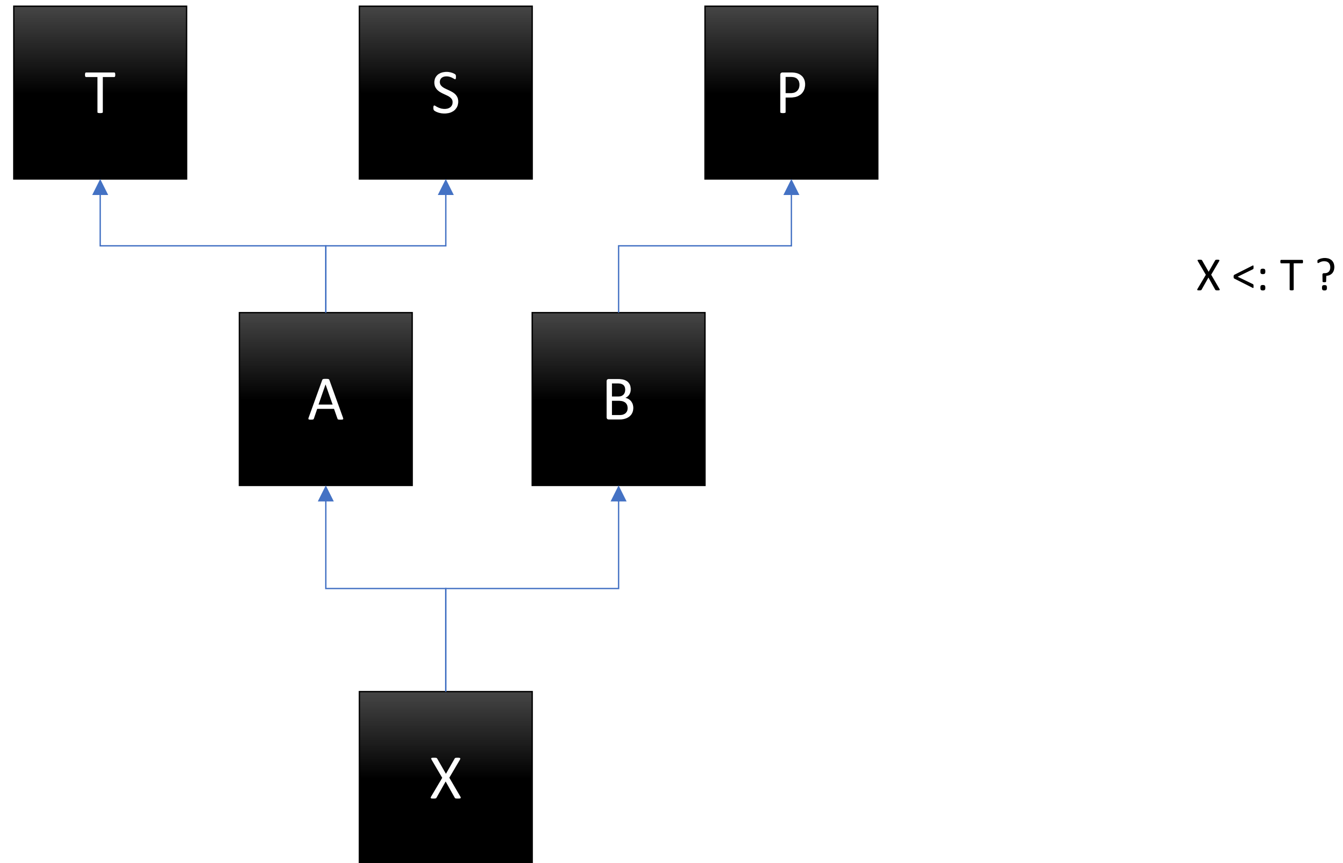


Dynamic Subtype Checking

```
S obj = new X()

if (obj instance of T) {
    ....
}
```

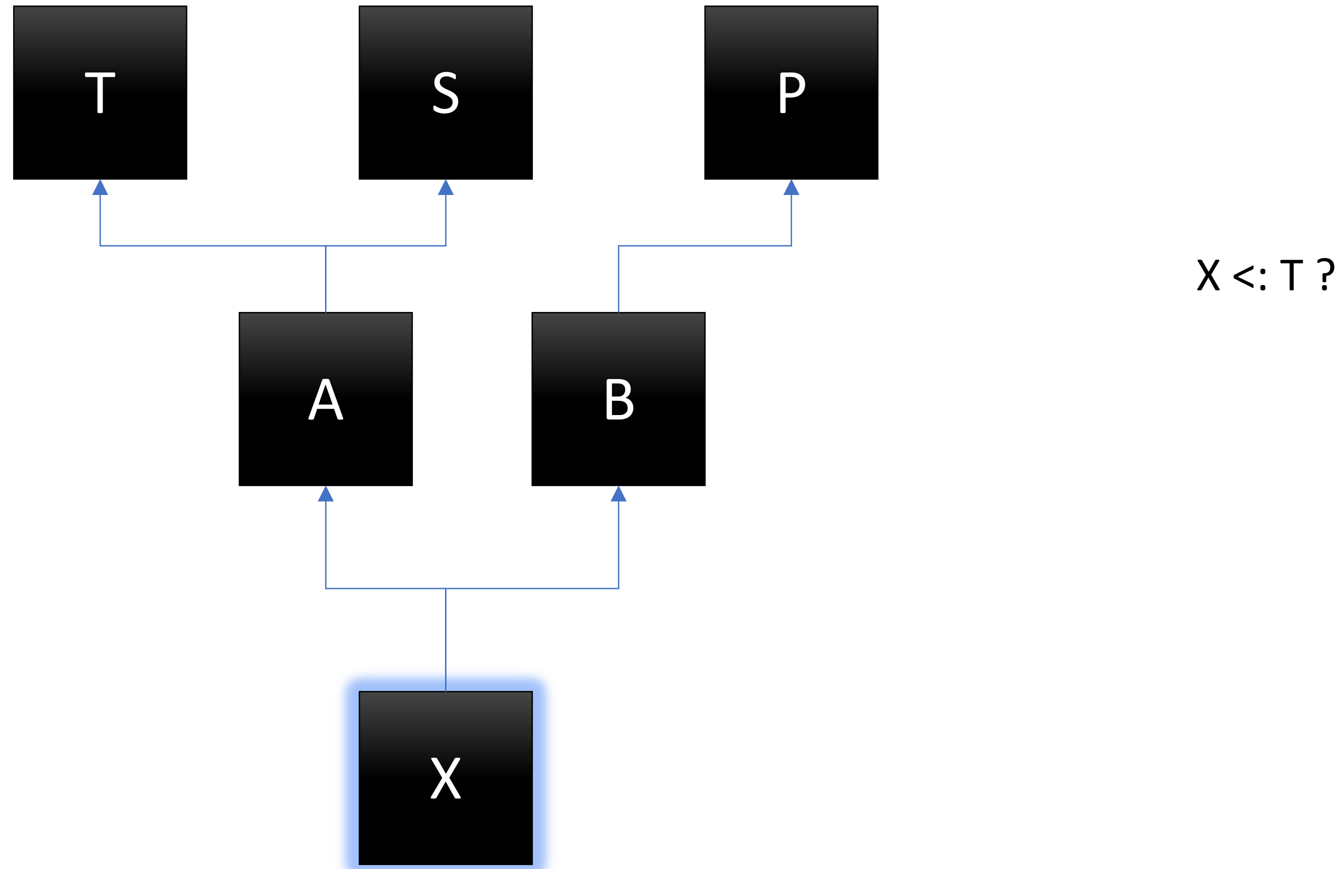


General Solution: Linear Search

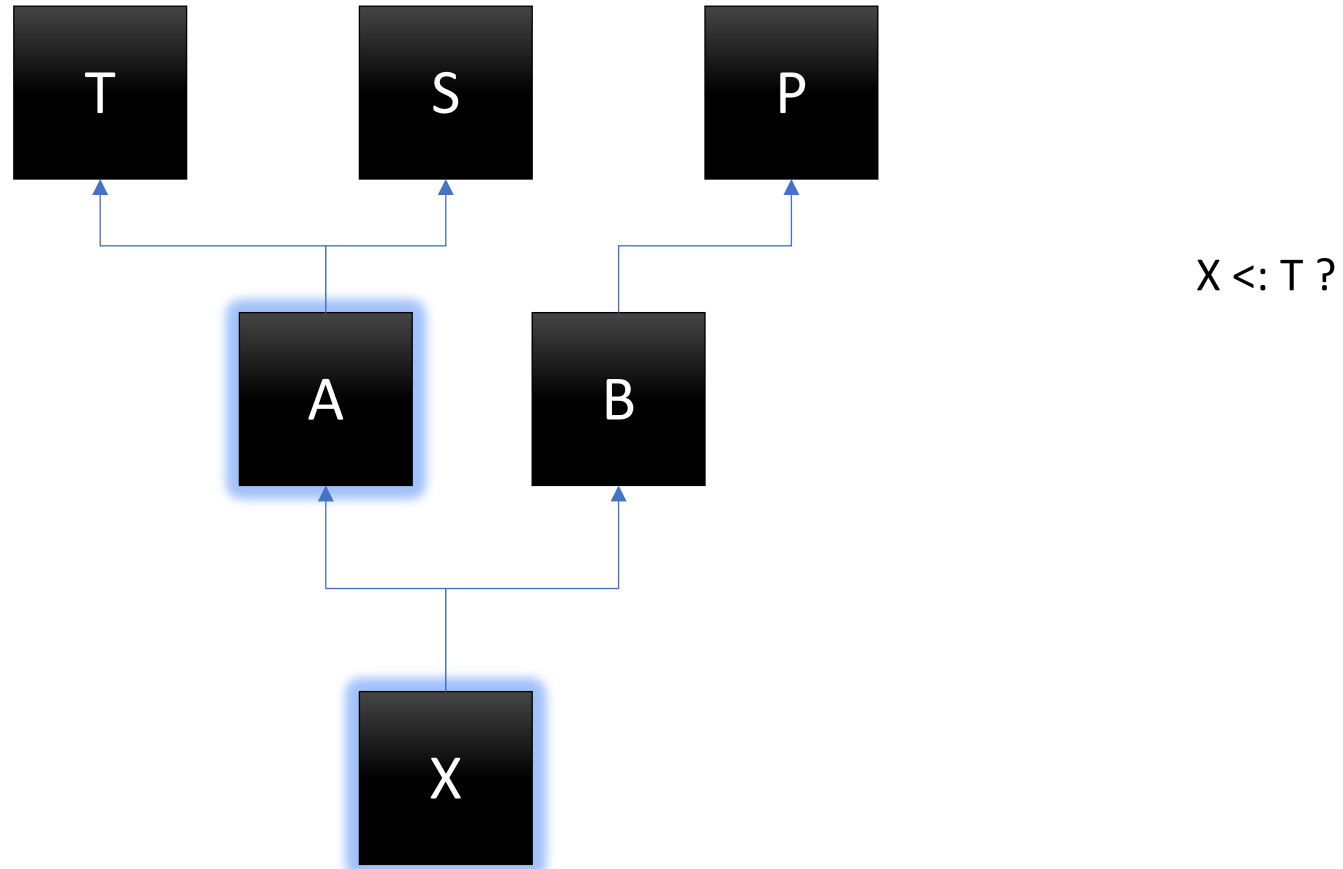


X <: T ?

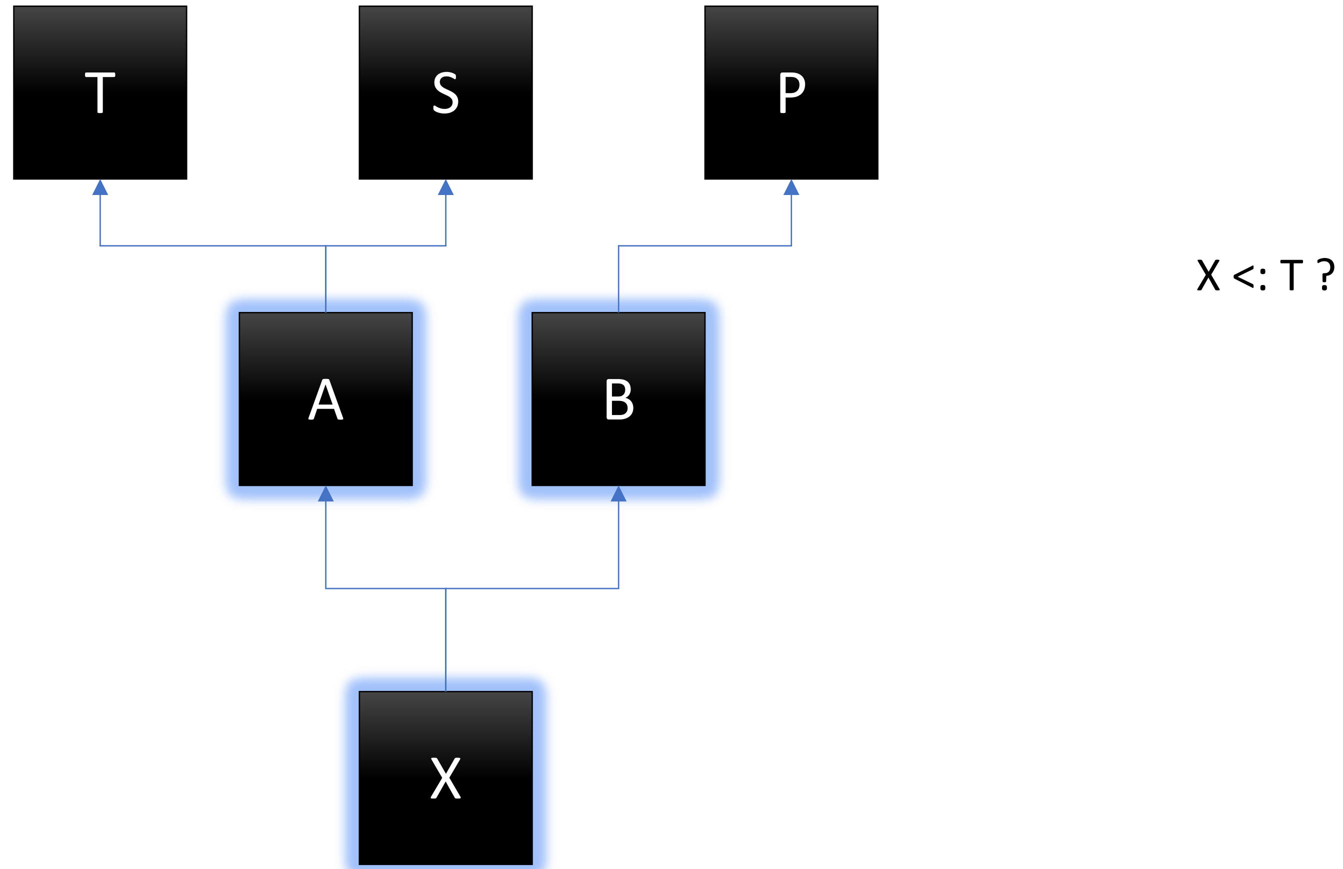
General Solution: Linear Search



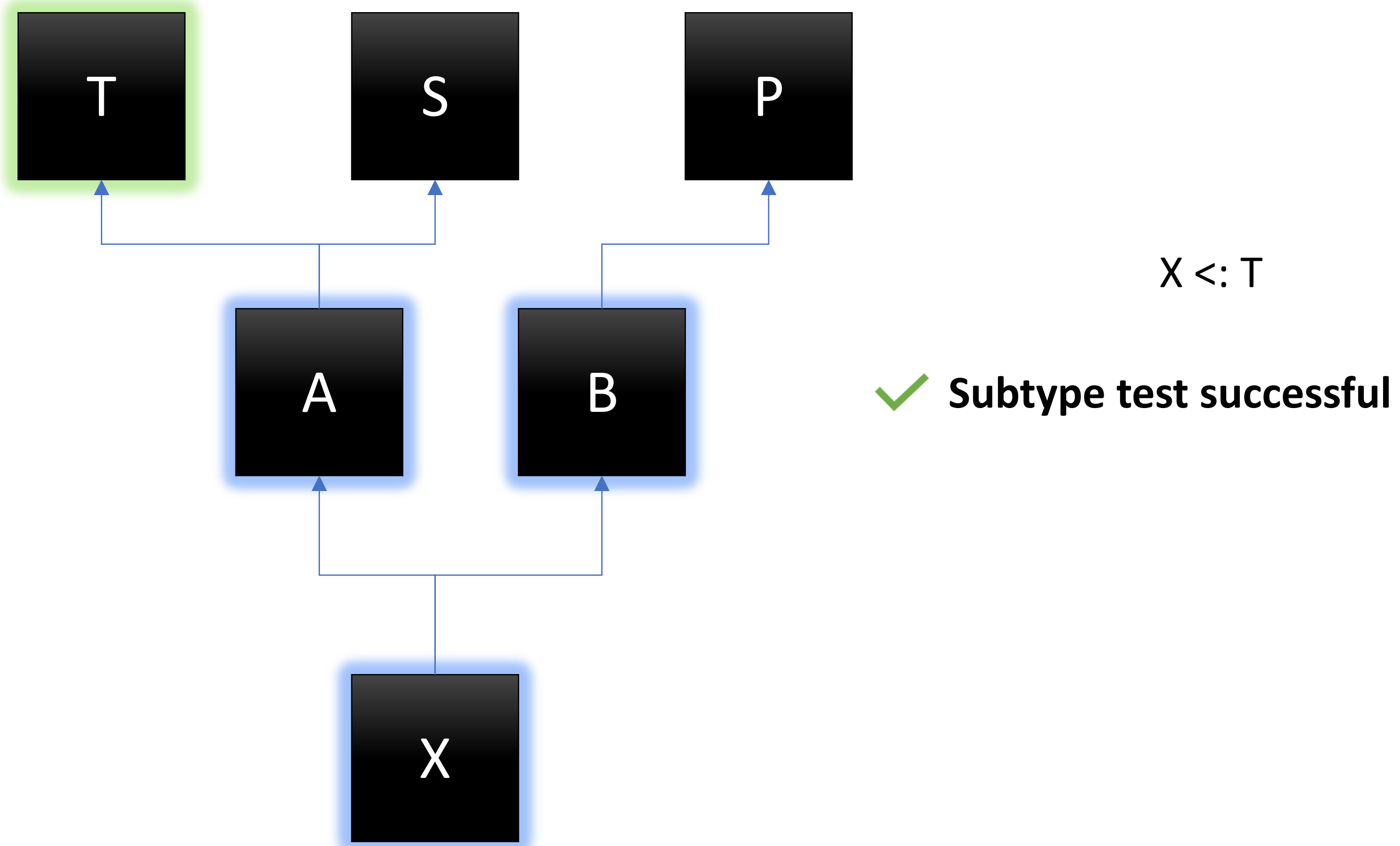
General Solution: Linear Search



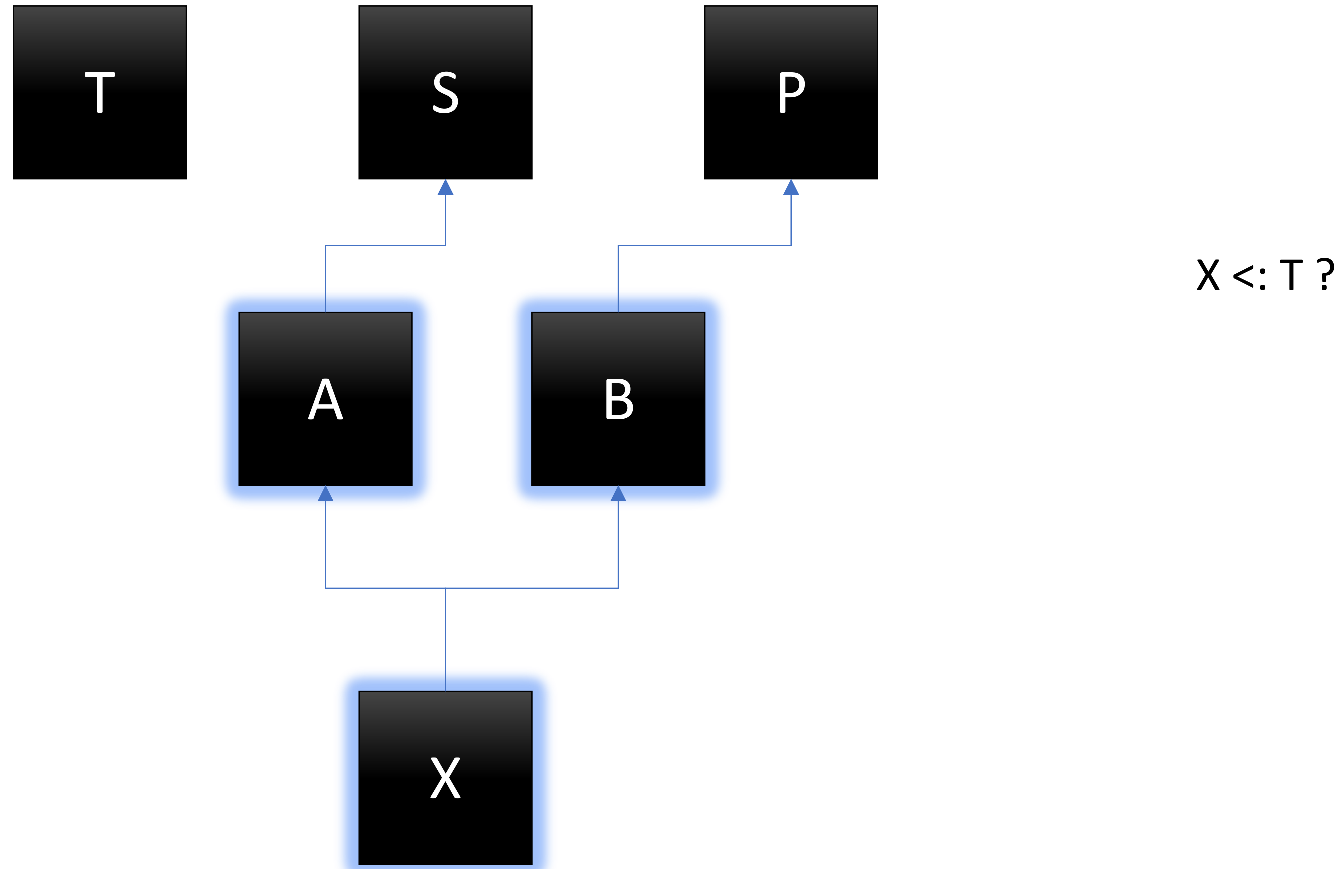
General Solution: Linear Search



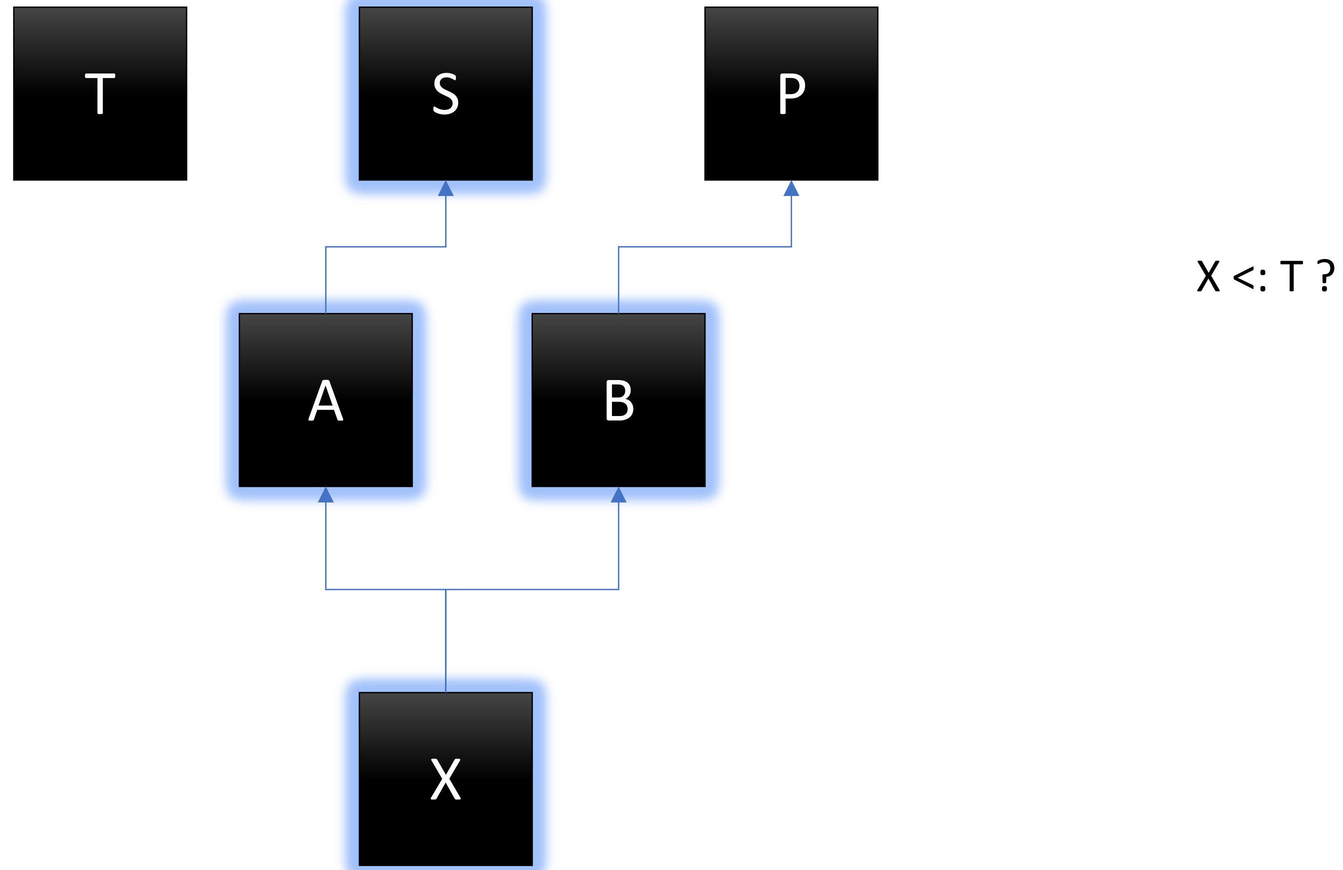
General Solution: Linear Search



General Solution: Linear Search

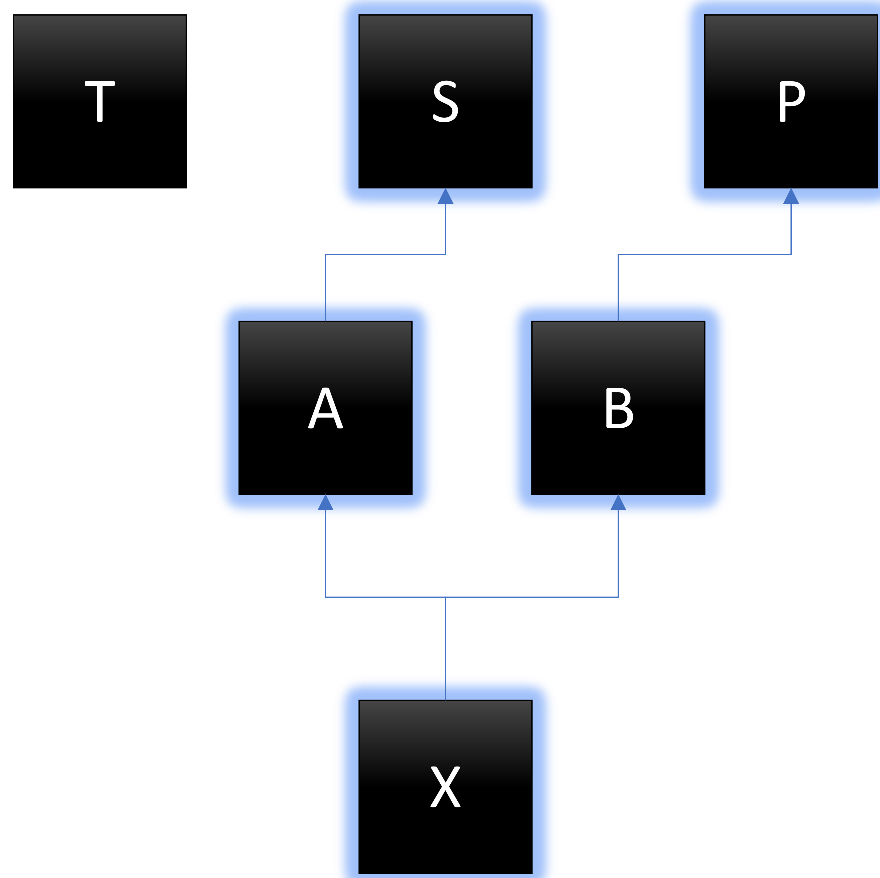


General Solution: Linear Search



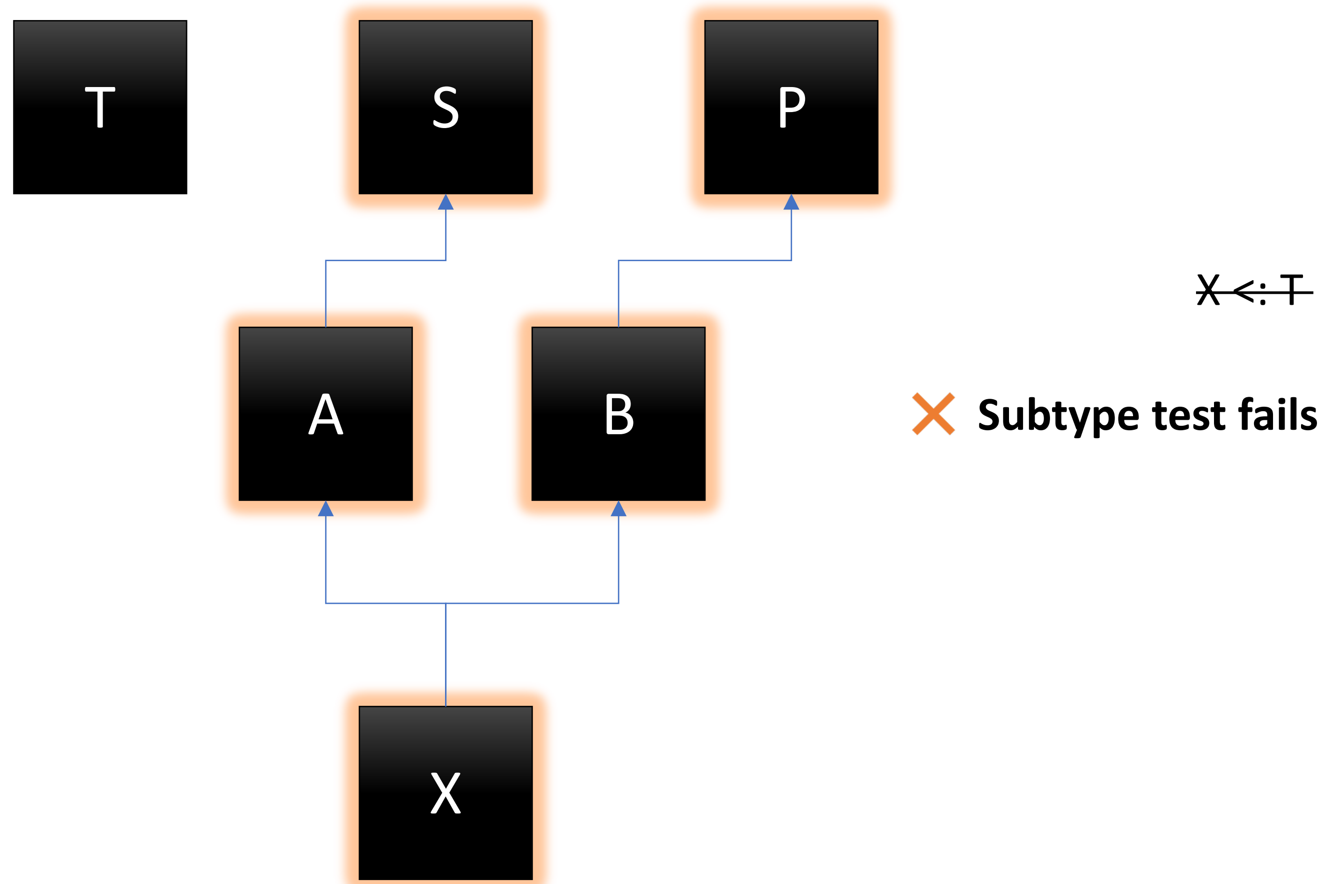
X <: T ?

General Solution: Linear Search



X <: T ?

General Solution: Linear Search



Implementations must consider trade-offs

Constant time?

Constant space? (per-class)

Supports multiple inheritance?

Supports open hierarchies?

Existing Schemes

Table 1. Summary of dynamic subtype checking strategies.

Scheme	Constant Space	Constant Time	Multiple Inheritance	Open Hierarchy
Schubert et al. [14]	✓	✓	✗	✗
Cohen's display [5]	✗	✓	✗	✓
NHE [10]	✗†	✓	✓	✗
Packed encoding [15]	✗†	✓	✓	✓‡
PQ-Encoding [17]	✗†	✓	✓	✗
R&B [13]	✗†	✓	✓	✓‡
Gibbs and Stroustrup [8]	✓	✓	✓	✗
Perfect Hashing [6]	✗	✓	✓	✓
HotSpot JVM [4]	✓/✗	✗	✓	✓
LLVM [1]	✓	✗	✓	✗

† The per-class space requirement is very small in practice.
‡ Requires non-trivial recomputation when dynamically loaded classes change the hierarchy.

Existing Schemes

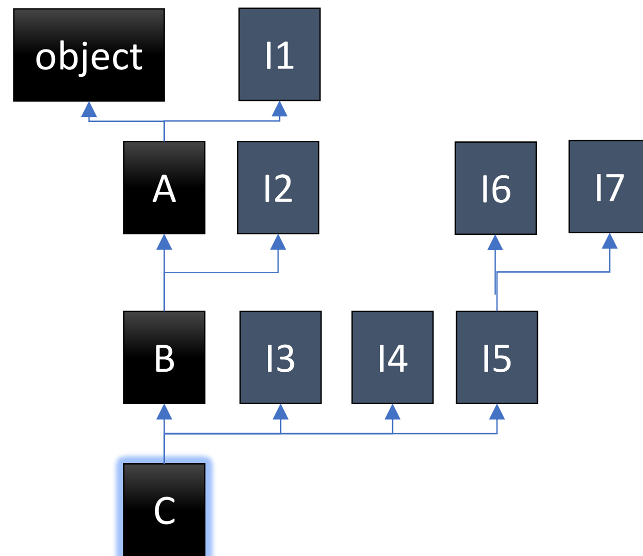
Table 1. Summary of dynamic subtype checking strategies.

Scheme	Constant Space	Constant Time	Multiple Inheritance	Open Hierarchy
Schubert et al. [14]	✓	✓	✗	✗
Cohen's display [5]	✗	✓	✗	✓
NHE [10]	✗†	✓	✓	✗
Packed encoding [15]	✗†	✓	✓	✓‡
PQ-Encoding [17]	✗†	✓	✓	✗
R&B [13]	✗†	✓	✓	✓‡
Gibbs and Stroustrup [8]	✓	✓	✓	✗
Perfect Hashing [6]	✗	✓	✓	✓
HotSpot JVM [4]	✓/✗	✗	✓	✓
LLVM [1]	✓	✗	✓	✗

† The per-class space requirement is very small in practice.
‡ Requires non-trivial recomputation when dynamically loaded classes change the hierarchy.

Case Study: HotSpot JVM

```
class A implements I1 { … }
class B extends A implements I2 { … }
interface I5 extends I6, I7, I2 { … }
class C extends B implements I3, I4, I5 { … }
```



Metadata for C

Primary

depth	super
0	object
1	A
2	B
3	C
4	
5	
6	
7	

Secondary

super
I1
I2
I6
I7
I5
I3
I4

Case Study: HotSpot JVM

```
class A implements I1 { … }
class B extends A implements I2 { … }
interface I5 extends I6, I7, I2 { … }
class C extends B implements I3, I4, I5 { … }
class D extends C
class E extends D
class F extends E
class G extends F
class H extends G
```

Metadata for H

Primary

depth	super
0	object
1	A
2	B
3	C
4	D
5	E
6	F
7	G

Secondary

super
I1
I2
I6
I7
I5
I3
I4
H

Case Study: HotSpot JVM

```
X <: C ?
  X.primary[3] == C?

X <: D ?
  X.primary[4] == D?

X <: H ?
  X.secondary_check(H)

X <: I5 ?
  X.secondary_check(I5)
```

Metadata for C

Primary

depth	super
0	object
1	A
2	B
3	C
4	
5	
6	
7	

Secondary

super
I1
I2
I6
I7
I5
I3
I4

Metadata for H

Primary

depth	super
0	object
1	A
2	B
3	C
4	D
5	E
6	F
7	G

Secondary

super
I1
I2
I6
I7
I5
I3
I4
H

Case Study: HotSpot JVM

```
X <: H ?
 X.secondary_check(H)

X <: I5 ?
 X.secondary_check(I5)
```

```
X.secondary_check(T) := {
  if (X.cache == T) return true;
  if (X == T) return true;
  foreach S in X.secondaries {
    if (S == T) {
      X.cache = S
      return true;
    }
  }
  return false;
}
```

Metadata for H

Secondary

super
I1
I2
I6
I7
I5
I3
I4
H

Case Study: HotSpot JVM

Observations:

1. Fast path for success
2. Failure == linear search

```
X.secondary_check(T) := {
  if (X.cache == T) return true;
  if (X == T) return true;
  foreach S in X.secondaries {
    if (S == T) {
      X.cache = S
      return true;
    }
  }
  return false;
}
```

Is this assumption always true?

Are there workloads where dynamic subtype tests often fail?

Case Study: Scala's Pattern Matching

```
obj match {
  case x:A => x.method_on_A()
  case y:B => y.method_on_B()
  case z:C => z.method_on_C()
  …
}
```

```
if (obj instanceof A) {
  A x = (A) obj;
  x.method_on_A();
} else if (obj instanceof B) {
  B y = (B) obj;
  y.method_on_B();
} else if (obj instanceof C) {
  C z = (C) obj;
  z.method_on_C();
} …
```

Profiling Scala's Pattern Matching

Small workload: `scalac Hello.scala`

- 47,597 `instanceof` tests
  - **93% failed**

Large workload: `sbt compile # builds scalac`

- 3.1 *billion* `instanceof` tests
  - **76% failed**
  - 45 million secondary scans

Cast Study: LLVM Compiler Infrastructure

```
if (AllocationInst *AI = dyn_cast<AllocationInst>(Val)) {
  …
} else if (CallInst *CI = dyn_cast<CallInst>(Val)) {
  …
} else if …
```

```
static bool isLoopInvariant(const Value *V, const Loop *L) {
  if (isa<Constant>(V) || isa<Argument>(V) || isa<GlobalValue>(V))
    return true;
  // Otherwise, it must be an instruction...
  return !L->contains(cast<Instruction>(V)->getParent());
}
```

Cast Study: LLVM Compiler Infrastructure

m::CallInst:

Inheritance diagram: class **CallInst**

llvm::CoroIdInst
llvm::AnyCoroIdInst
llvm::AnyCoroIdRetconInst
llvm::AnyCoroSuspendInst
llvm::CoroSuspendInst
llvm::BinaryOpIntrinsic
llvm::ConstrainedFPIntrinsic
llvm::CoroAllocaAllocInst
llvm::CoroAllocaFreeInst
llvm::CoroAllocaGetInst
llvm::CoroAllocInst
llvm::CoroBeginInst
llvm::CoroEndInst
llvm::CoroFrameInst
llvm::CoroFreeInst
llvm::CoroPromiseInst
llvm::CoroSaveInst
llvm::CoroSizeInst
llvm::CoroSubFnInst
llvm::DbgInfoIntrinsic
llvm::GCProjectionInst
llvm::InstrProfIncrementInst
llvm::InstrProfValueProfileInst
llvm::MemIntrinsicBase < MemIntrinsic >
llvm::MemIntrinsicBase < Derived >
llvm::MemIntrinsicBase < AtomicMemIntrinsic >
llvm::MemIntrinsicBase < AnyMemIntrinsic >
llvm::VACopyInst
llvm::VAEndInst
llvm::VAStartInst
llvm::IntrinsicInst
llvm::CallInst
llvm::CallBase
llvm::Instruction
llvm::User
llvm::Value
llvm::ilist_node_with_parent< Instruction, BasicBlock >
llvm::ilist_node_with_parent< NodeTy, ParentTy, Options >
llvm::ilist_node< NodeTy, Options... >
llvm::ilist_node< Instruction, Options... >
llvm::ilist_node< T, Options >
llvm::ilist_node_impl< ilist_detail::compute_node_options< NodeTy, Options... >::type >
llvm::ilist_node_impl< ilist_detail::compute_node_options< T, Options... >::type >
llvm::ilist_node_impl< ilist_detail::compute_node_options< Instruction, Options... >::type >
llvm::ilist_node_impl< OptionsT >
OptionsT::node_base_type
ilist_detail::compute_node_options::type::node_base_type< NodeTy, Options... >
ilist_detail::compute_node_options< T, Options... >::type::node_base_type
ilist_detail::compute_node_options::type::node_base_type< Instruction, Options... >

Profiling the LLVM Compiler Infrastructure

Small workload: `clang++ Hello.cpp`

- 5.5 million `dyn_cast<T>/isa<T>` operations
  - **74% failed**

Large workload: `clang selfie.c # 10K LoC`

- 93.7 million `dyn_cast<T>/isa<T>` operations
  - **78% failed**

Takeaway: In some workloads…

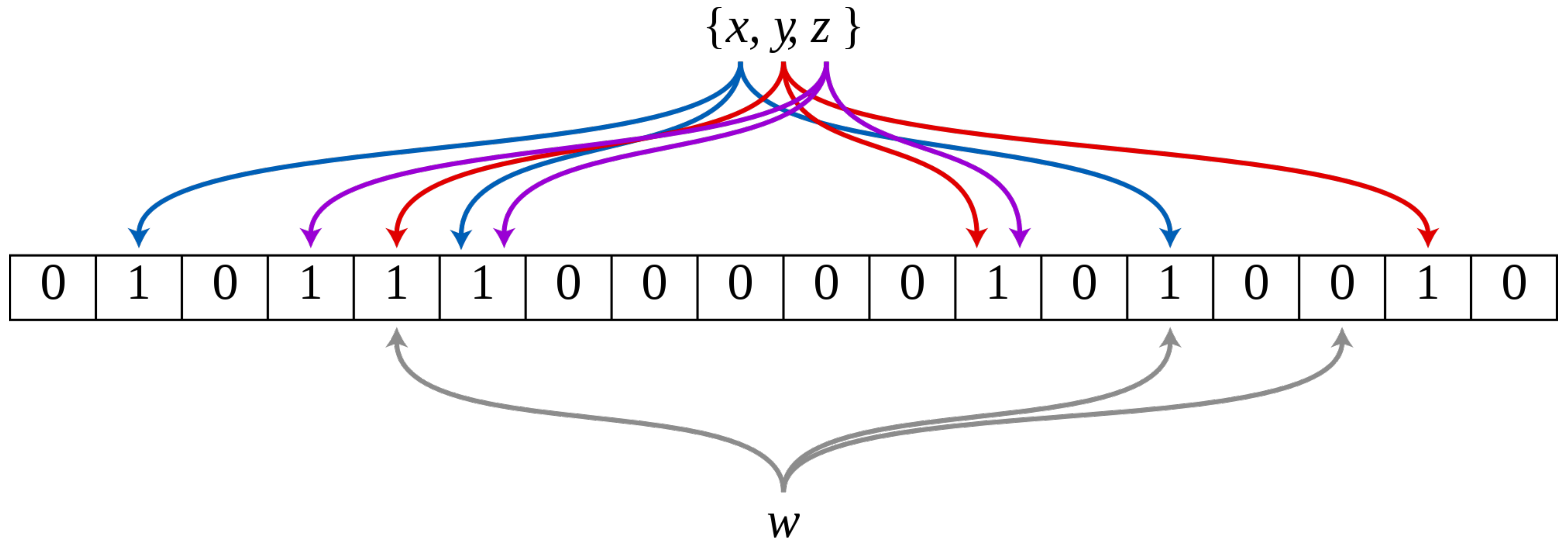
Dynamic subtype tests often fail

But fast path is optimized for successful tests ☹

Can we fail fast when linear search is likely?

(with no overhead for the current fast path)

Solution: **Bloom Filters**



Fail-Fast using Bloom Filters

For each type T:

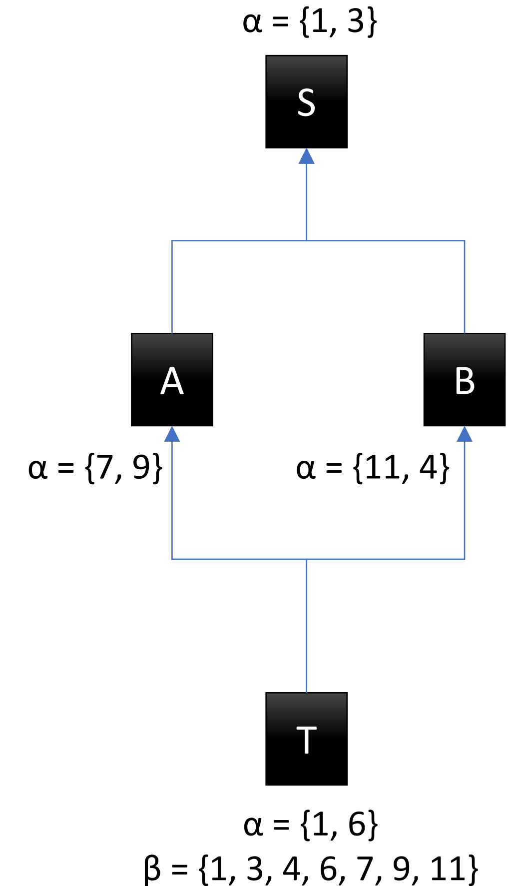
$\alpha(T)$:= k distinct integers, chosen randomly from $[1..m]$

$\beta(T) := \alpha(T) \cup \alpha(S_1) \cup \alpha(S_2) \cup \ldots \cup \alpha(S_n)$

where $S_1, S_2, \ldots S_n$ are all the (transitive) super-types of T

Invariant:

$$T <: S \Rightarrow \alpha(S) \subseteq \beta(T)$$



X

α = {5, 8}

X <: T ? **No**

Y

α = {7, 11}

Y <: T ? **Maybe**

Fail-Fast using Bloom Filters

For each type T:

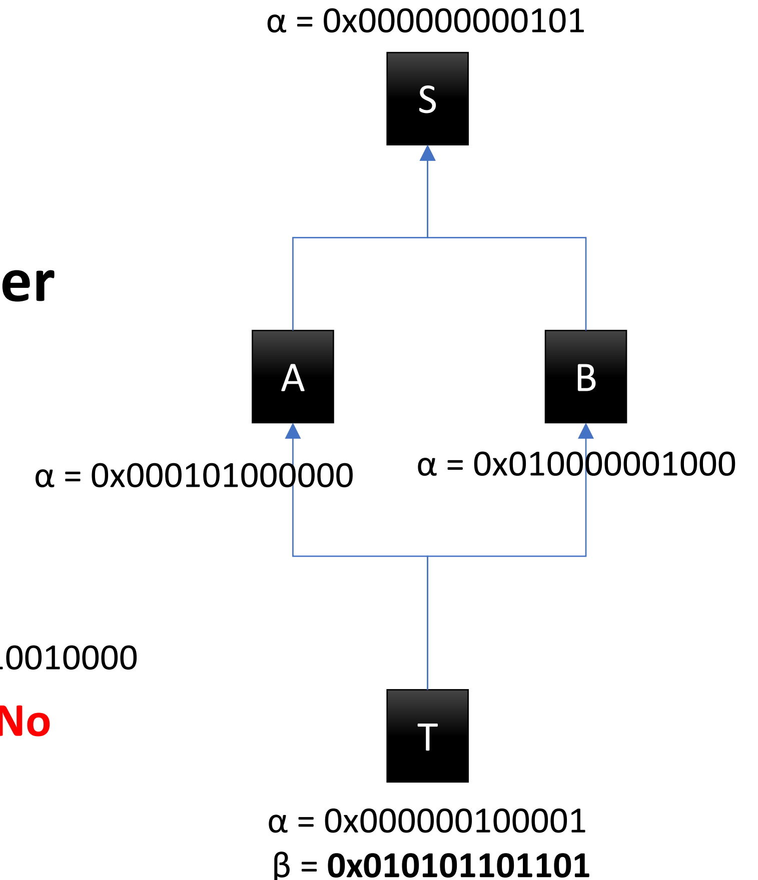
$\alpha(T)$:= **compile_time_random(parity=*k*)   // m-bit integer**

$\beta(T)$:= $\boldsymbol{\alpha(T) \mid \alpha(S_1) \mid \alpha(S_2) \mid \ldots \mid \alpha(S_n)}$

where $S_1, S_2, \ldots S_n$ are all the (transitive) super-types of T

Invariant:

$$\mathbf{T <: S \Rightarrow \alpha(S)\ \&\ \beta(T) = \alpha(S)}$$



Fail-Fast using Bloom Filters

```
// is object `o` an instance of type `T`?
boolean fail_fast_instanceof(S o, type T) {
  if (type(o).beta & T.alpha != T.alpha) {
    return false;
  } else {
    return slow_instanceof(o, T); // linear scan
  }
}
```

Worst-case only when false positive in bloom filters

Choosing parameters

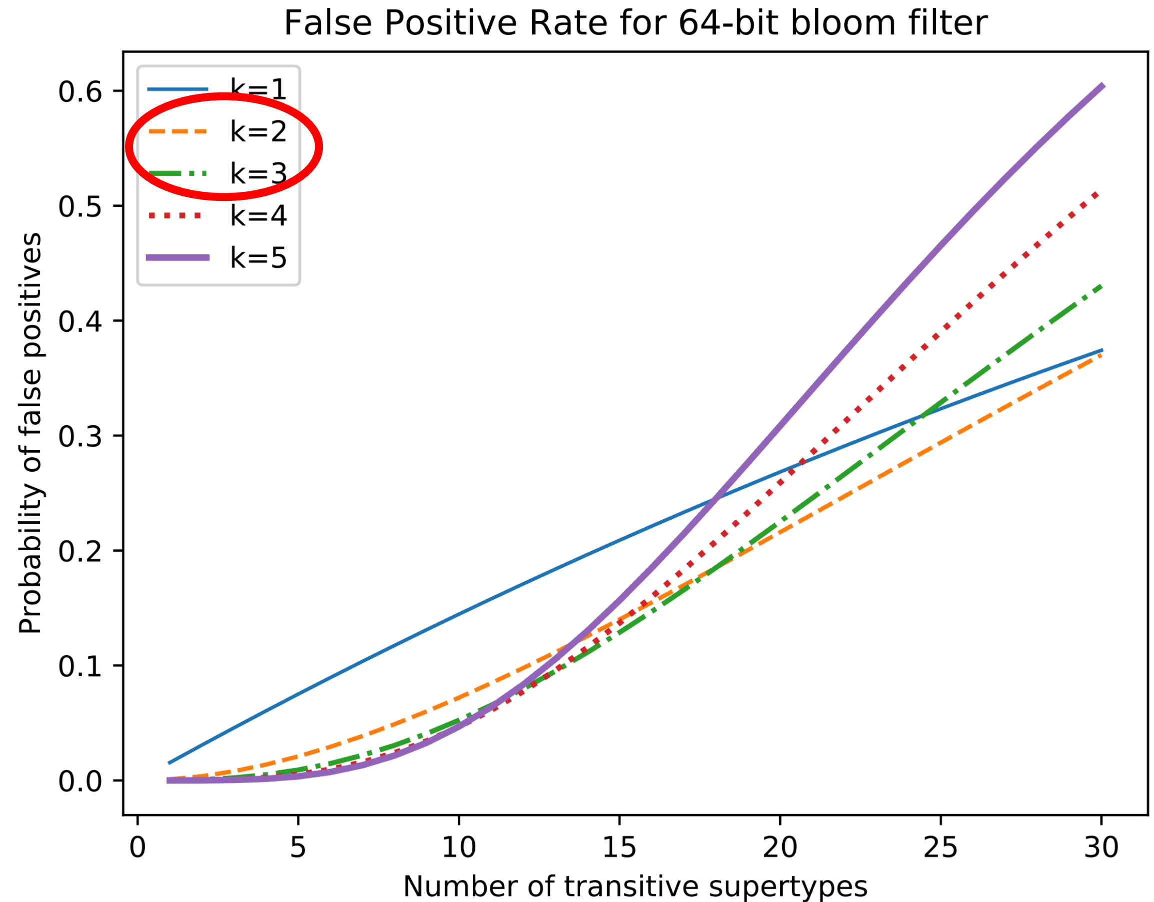
m = size of machine word

k = **parity ??**

n = num. of transitive supertypes

False positive rate:

$$p = (1 - e^{-\frac{k \times n}{m}})^k$$



Preliminary Evaluation (JVM HotSpot)

```
class Foo extends Bar implements Baz, Qux {









  /* members of Foo */
}
```

Preliminary Evaluation (JVM HotSpot)

```
class Foo extends Bar implements Baz, Qux {
  public static final long __alpha__
                 = FailFast.genAlpha();

  public static final long __beta__
             = Foo.__alpha__ | Bar.__beta__ |
               Baz.__beta__  | Qux.__beta__;

  @Override public long __getBeta__() {
    return Foo.__beta__;
  }

  /* members of Foo */
}
```

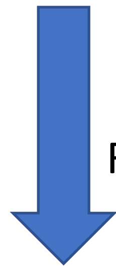
Preliminary Evaluation (JVM HotSpot)

```
obj match {
  case x:A => x.method_on_A()
  case y:B => y.method_on_B()
  case z:C => z.method_on_C()
  …
}
```

```
if (obj instanceof A) {
  A x = (A) obj;
  x.method_on_A();
} else if (obj instanceof B) {
  B y = (B) obj;
  y.method_on_B();
} else if (obj instanceof C) {
  C z = (C) obj;
  z.method_on_C();
} …
```

Preliminary Evaluation (JVM HotSpot)

```
if(o instanceof T){...}
```



Rewrite if T is a secondary type

```
if (o.__getBeta__() & T.__alpha__ == T.__alpha__
    && o instanceof T) { ... }
```

Preliminary Evaluation (JVM HotSpot)

	Baseline [4]		With Fail-Fast		
Benchmark	**Time**	**Worst Case**	**Time**	**Speedup**	**Worst Case**
Single Negative	51.458 ± 0.126 ns	100%	35.663 ± 0.092 ns	1.44×	0%

```
if(o instanceof T){...}
```

Preliminary Evaluation (JVM HotSpot)

	Baseline [4]		With Fail-Fast		
Benchmark	Time	Worst Case	Time	Speedup	Worst Case
Single Negative	51.458 ± 0.126 ns	100%	35.663 ± 0.092 ns	1.44×	0%
2-Random Case Match	46.248 ± 0.127 ns	33%	39.314 ± 0.082 ns	1.18×	0%

```
obj match {
  case x:A => x.method_on_A()
  case y:B => y.method_on_B()
}
```

trait Base
trait A extends Base { def method_on_A(): Int }
trait B extends Base { def method_on_B(): Int }

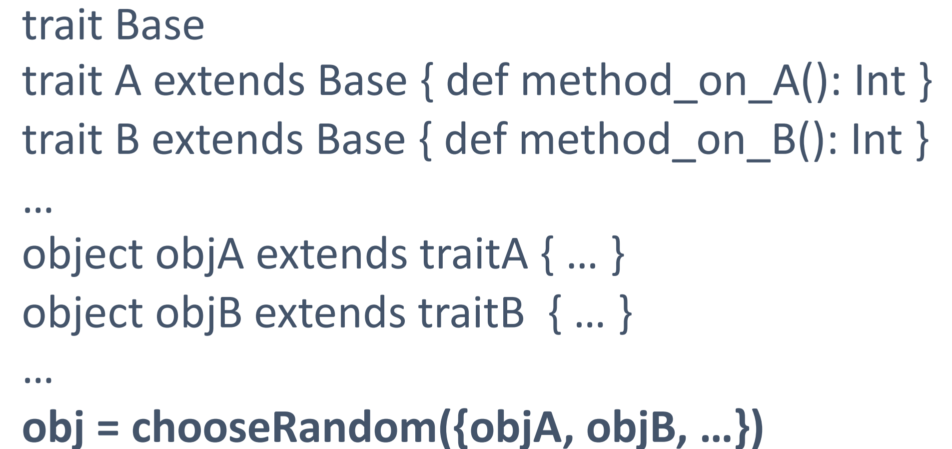
object objA extends traitA { … }
object objB extends traitB { … }

obj = chooseRandom({objA, objB})

Preliminary Evaluation (JVM HotSpot)

	Baseline [4]		With Fail-Fast		
Benchmark	**Time**	**Worst Case**	**Time**	**Speedup**	**Worst Case**
Single Negative	51.458 ± 0.126 ns	100%	35.663 ± 0.092 ns	1.44×	0%
2-Random Case Match	46.248 ± 0.127 ns	33%	39.314 ± 0.082 ns	1.18×	0%
5-Random Case Match	75.090 ± 0.226 ns	67%	47.550 ± 0.130 ns	1.58×	0%

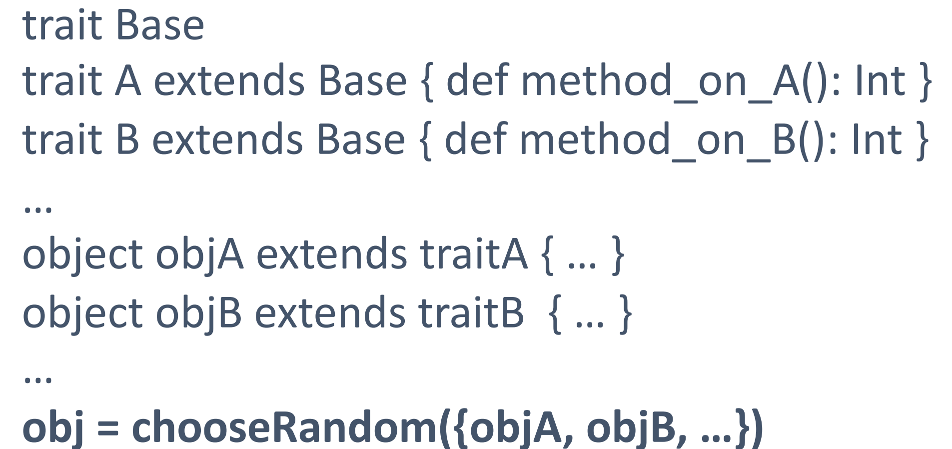
```
obj match {
  case x:A => x.method_on_A()
  case y:B => y.method_on_B()
  case z:C => z.method_on_C()
  case u:D => u.method_on_D()
  case v:E => v.method_on_E()


}
```



Preliminary Evaluation (JVM HotSpot)

	Baseline [4]		With Fail-Fast		
Benchmark	**Time**	**Worst Case**	**Time**	**Speedup**	**Worst Case**
Single Negative	51.458 ± 0.126 ns	100%	35.663 ± 0.092 ns	1.44×	0%
2-Random Case Match	46.248 ± 0.127 ns	33%	39.314 ± 0.082 ns	1.18×	0%
5-Random Case Match	75.090 ± 0.226 ns	67%	47.550 ± 0.130 ns	1.58×	0%
10-Random Case Match	116.031 ± 0.582 ns	82%	50.722 ± 0.228 ns	2.29×	0%

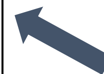
```
obj match {
  case x:A => x.method_on_A()
  case y:B => y.method_on_B()
  case z:C => z.method_on_C()
  case u:D => u.method_on_D()
  case v:E => v.method_on_E()
  …
  case q:H => q.method_on_H()
}
```



Preliminary Evaluation (JVM HotSpot)

	Baseline [4]		With Fail-Fast		
Benchmark	**Time**	**Worst Case**	**Time**	**Speedup**	**Worst Case**
Single Negative	51.458 ± 0.126 ns	100%	35.663 ± 0.092 ns	1.44×	0%
2-Random Case Match	46.248 ± 0.127 ns	33%	39.314 ± 0.082 ns	1.18×	0%
5-Random Case Match	75.090 ± 0.226 ns	67%	47.550 ± 0.130 ns	1.58×	0%
10-Random Case Match	116.031 ± 0.582 ns	82%	50.722 ± 0.228 ns	2.29×	0%
10-Random Cases + 10-Type Noise	143.057 ± 0.424 ns	82%	52.286 ± 0.205 ns	2.74×	9%

```
obj match {
  case x:A => x.method_on_A()
  case y:B => y.method_on_B()
  case z:C => z.method_on_C()
  case u:D => u.method_on_D()
  case v:E => v.method_on_E()
  …
  case q:H => q.method_on_H()
}
```



```
trait Base extends N1, N2, N3, … N10
trait A extends Base { def method_on_A(): Int }
trait B extends Base { def method_on_B(): Int }
…
object objA extends traitA { … }
object objB extends traitB  { … }
…
obj = chooseRandom({objA, objB, …})
```

Summary

Dynamic subtype tests often fail (in some workloads)

Worst-case linear search occurs (in production VMs)

Bloom filters can enable fail-fast refutations (high probability)

expected constant time + constant space + multiple inheritance + open hierarchy