Concepts of Object-Oriented Programming

Peter Müller

Chair of Programming Methodology

Autumn Semester 2012

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Meeting the Requirements

Cooperating Program Parts with Well-Defined Interfaces

→ Objects (data + code)
→ Interfaces
→ Encapsulation

Classification and Specialization

→ Classification, subtyping
→ Polymorphism
→ Substitution principle

Highly Dynamic Execution Model

→ Active objects
→ Message passing

Correctness

→ Interfaces
→ Encapsulation
→ Simple, powerful concepts

Topics in this Section

- Cooperating program parts …
    - How do we define components?

- … with well-defined interfaces
    - What is the interface of a component?
    - How do we describe the interface of a component?
    - How do we make sure clients use a component correctly?

Topics in this Section

- Cooperating program parts …
    - How do we define components?

- … with well-defined interfaces
    - What is the interface of a component?
    - How do we describe the interface of a component?
    - How do we make sure clients use a component correctly?

Types and contracts

4. Types

4.1 Bytecode Verification

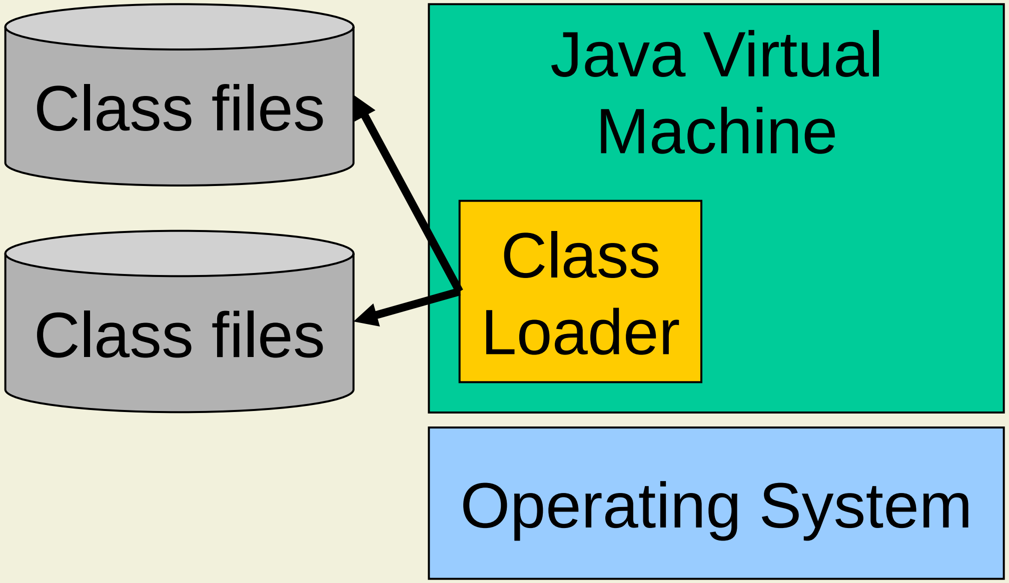
4.2 Parametric Polymorphism

Mobile Code: Motivation

- Download and execution of code, e.g., Java applets
    - Web pages
    - Mobile devices
- Upload of code
    - Customizing servers
- Automatic distribution of code and patches in distributed systems

Class Loaders

- Programs are compiled to bytecode
  - Platform-independent format
  - Organized into class files
- Bytecode is interpreted on a virtual machine
- Class loader gets code for classes and interfaces on demand
- Programs can contain their own class loaders



Example: Specialized Class Loader

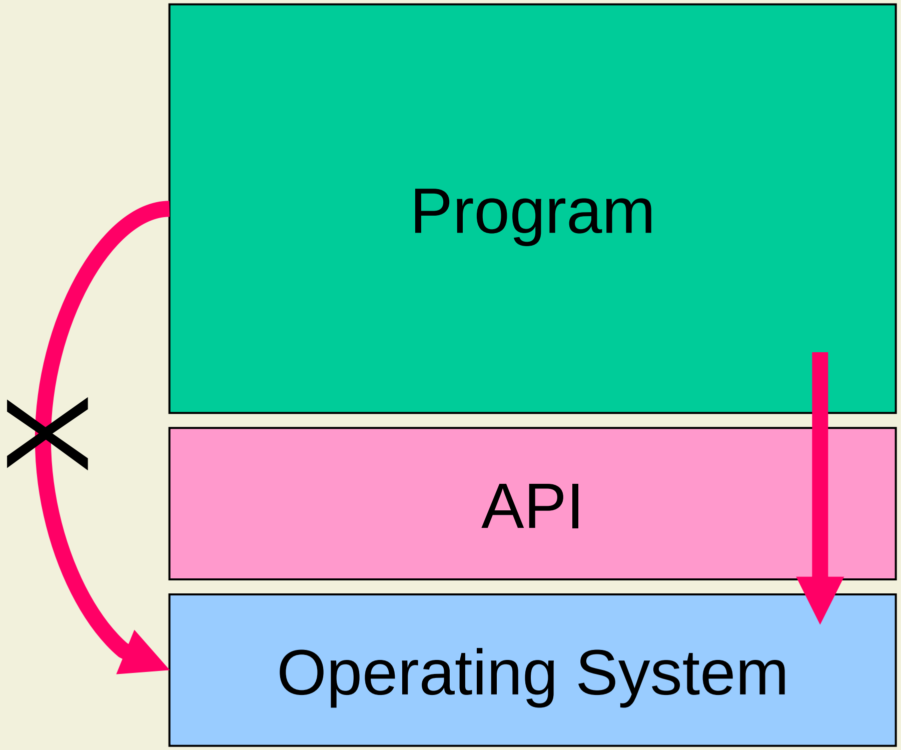
Error handling partly omitted

```
public class MyLoader extends ClassLoader {
  byte[ ] getClassData( String name ) { … }

  public synchronized Class loadClass( String name )
      throws ClassNotFoundException {

    Class c = findLoadedClass( name );
    if ( c != null ) return c;

    try { c = findSystemClass( name );  return c; }
    catch ( ClassNotFoundException e ) { }

    byte[ ] data = getClassData( name );
    return defineClass( name, data, 0, data.length );  }
}
```

Java

Security for Java Programs

- Sandbox
  - Applets get access to system resources only through an API
  - Access control can be implemented
- Security relies on
  - Type safety
  - Code does not by-pass sandbox



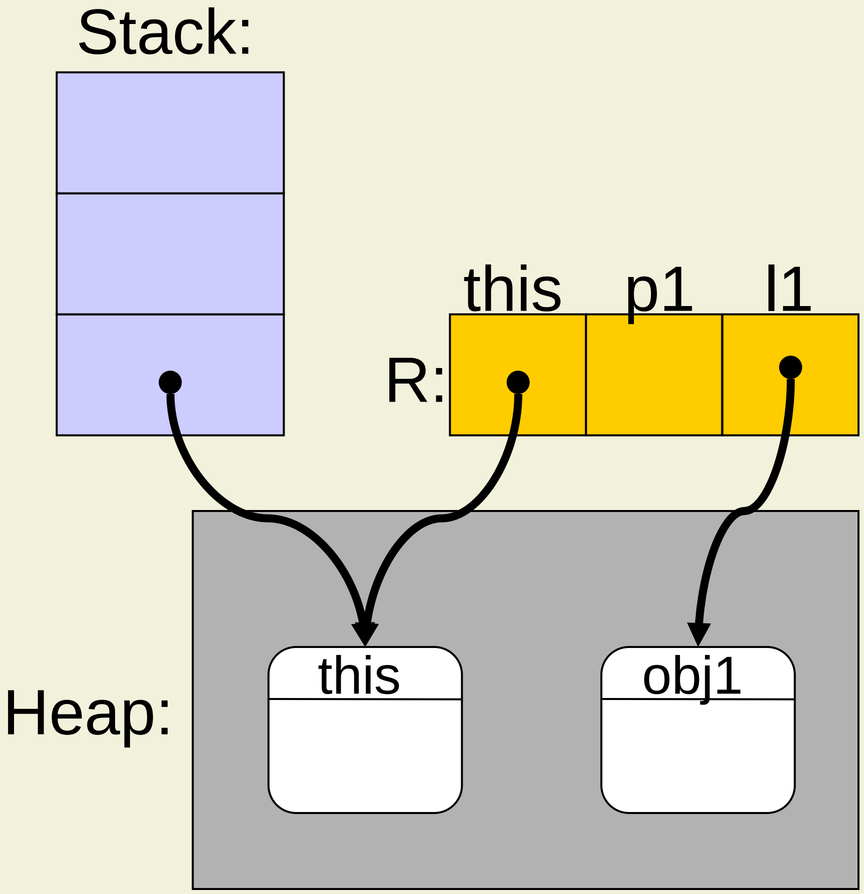
Security in Mobile Environments

- Mobile code cannot be trusted
  - Code may not be type safe
  - Code may destroy or modify data
  - Code may expose personal information
  - Code may crash the underlying VM
  - Code may purposefully degrade performance (denial of service)

- How to guarantee a minimum level of security?
  - Untrusted code producer
  - Untrusted compiler

Java Virtual Machine

- JVM is stack-based
- Most operations pop operands from a stack and push a result
- Registers store method parameters and local variables
- Stack and registers are part of the method activation record

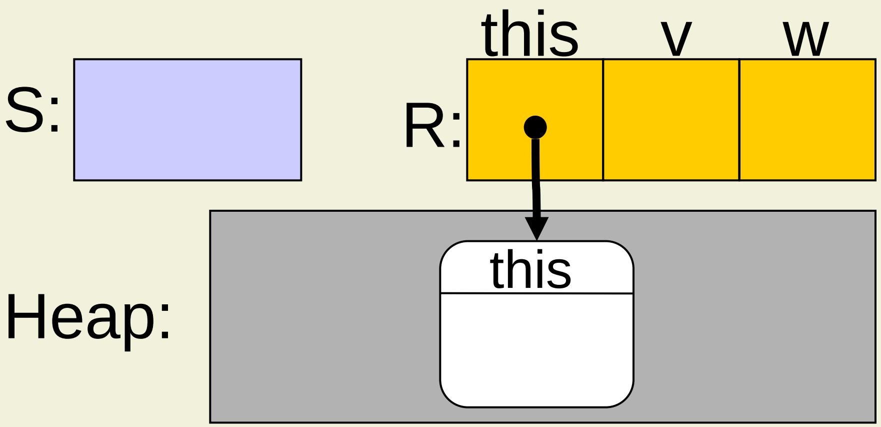


Java Bytecode

- Instructions are typed
- Load and store instructions access registers
- Control is handled by intra-method branches (goto, conditional branches)

```
class C {
  void m( ) {
    int v;
    Object w;
    v = 5;
    w = this;
  }
}
```

```
iconst 5
istore 1
aload 0
astore 2
return
```

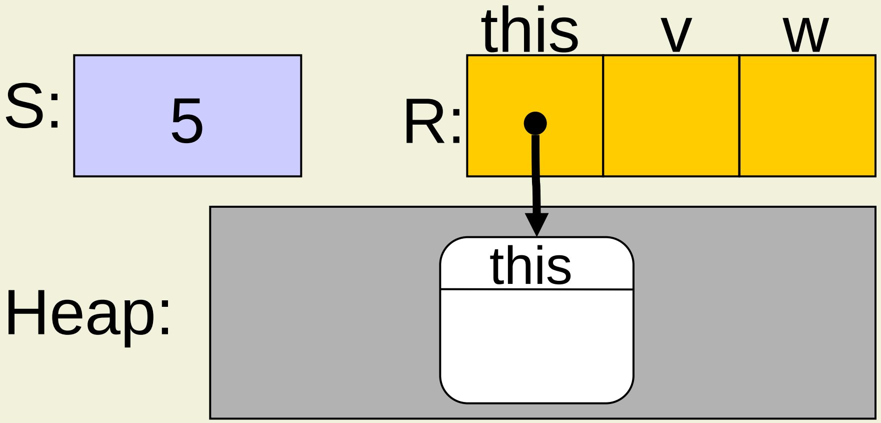


Java Bytecode

- Instructions are typed
- Load and store instructions access registers
- Control is handled by intra-method branches (goto, conditional branches)

```
class C {
  void m( ) {
    int v;
    Object w;
    v = 5;
    w = this;
  }
}
```

```
iconst 5
istore 1
aload 0
astore 2
return
```

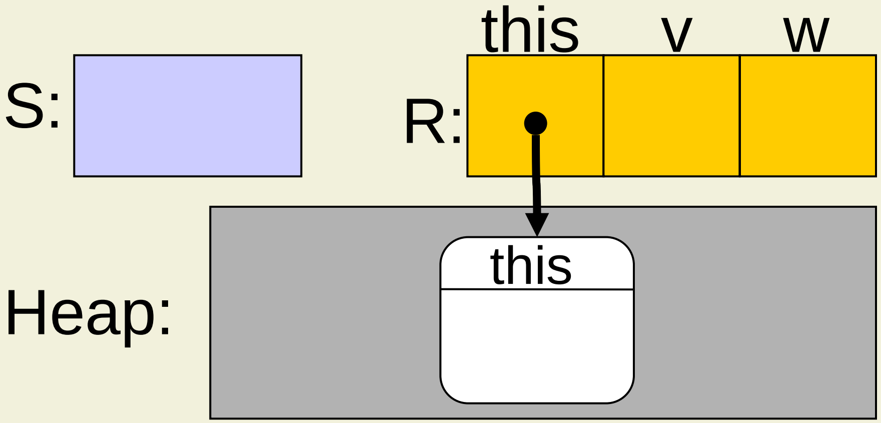


Java Bytecode

- Instructions are typed
- Load and store instructions access registers
- Control is handled by intra-method branches (goto, conditional branches)

```
class C {
  void m( ) {
    int v;
    Object w;
    v = 5;
    w = this;
  }
}
```

```
iconst 5
istore 1
aload 0
astore 2
return
```

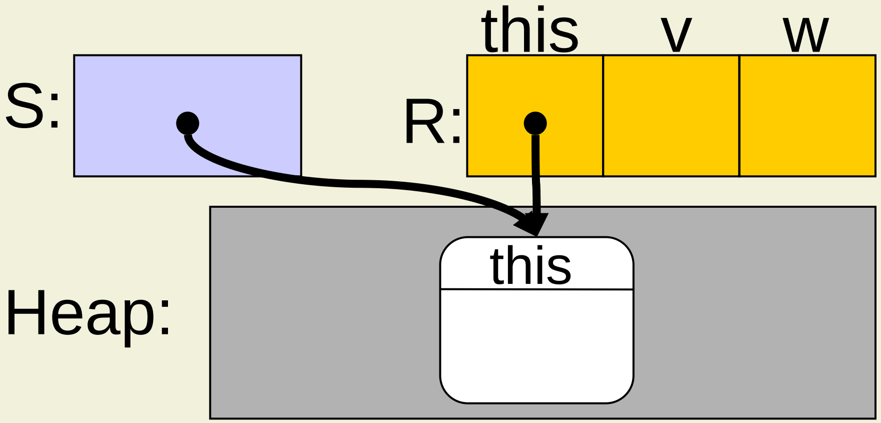


Java Bytecode

- Instructions are typed
- Load and store instructions access registers
- Control is handled by intra-method branches (goto, conditional branches)

```
class C {
  void m( ) {
    int v;
    Object w;
    v = 5;
    w = this;
  }
}
```

```
iconst 5
istore 1
aload 0
astore 2
return
```

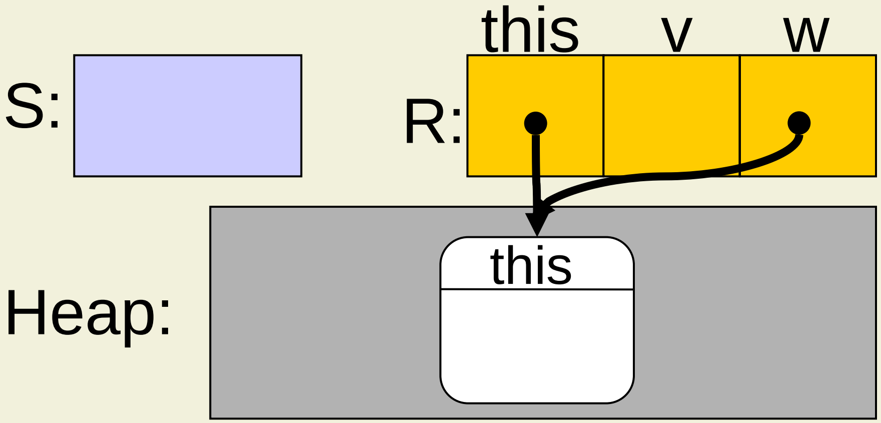


Java Bytecode

- Instructions are typed
- Load and store instructions access registers
- Control is handled by intra-method branches (goto, conditional branches)

```
class C {
  void m( ) {
    int v;
    Object w;
    v = 5;
    w = this;
  }
}
```

```
iconst 5
istore 1
aload 0
astore 2
return
```

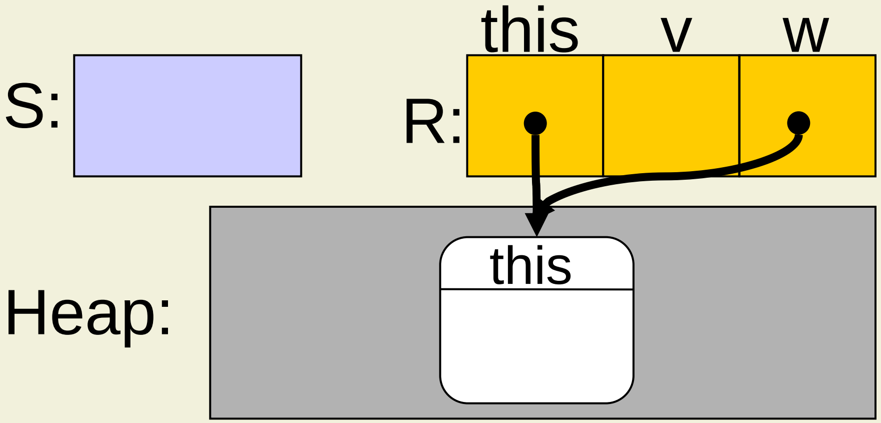


Java Bytecode

- Instructions are typed
- Load and store instructions access registers
- Control is handled by intra-method branches (goto, conditional branches)

```
class C {
  void m( ) {
    int v;
    Object w;
    v = 5;
    w = this;
  }
}
```

```
iconst 5
istore 1
aload 0
astore 2
return
```



Bytecode Verification

- Proper execution requires that
  - Each instruction is type correct
  - Only initialized variables are read
  - No stack over- or underflow occurs
  - Etc.

- Java Virtual Machine guarantees these properties
  - By bytecode verification when a class is loaded
  - By dynamic checks at run time

Bytecode Verification via Type Inference

- The Bytecode verifier simulates the execution of the program
- Operations are performed on types instead of values
- For each instruction, a rule describes how the operand stack and local variables are modified

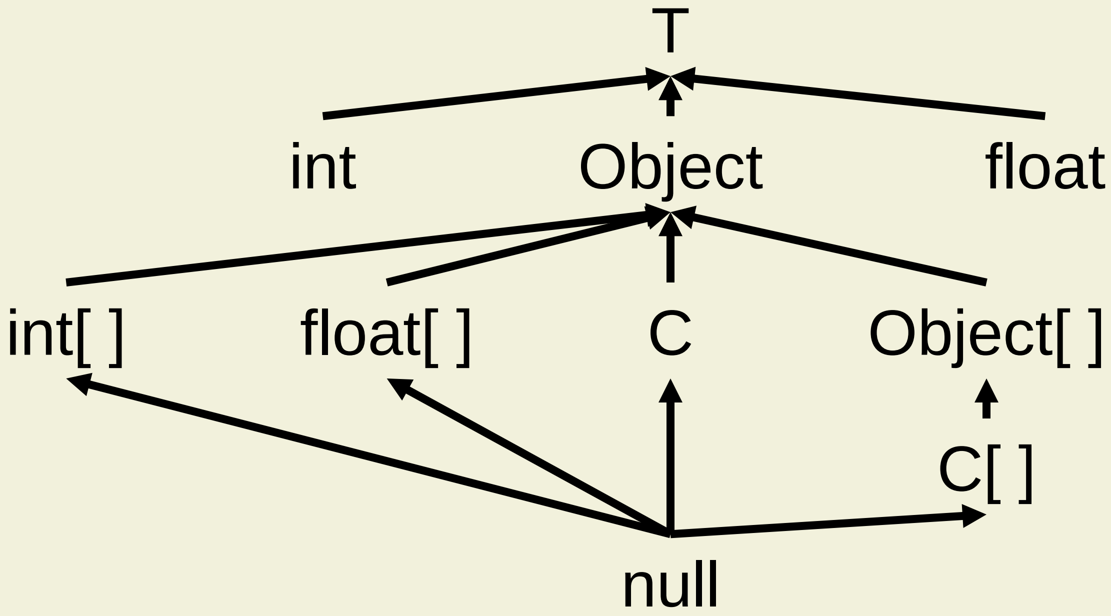
$$i: (S,R) \rightarrow (S',R')$$

$$iadd: \ (int.int.S,R) \rightarrow (int.S,R)$$

- Errors are denoted by the absence of a transition
  - Type mismatch
  - Stack over- or underflow

Types of the Inference Engine

- Primitive types
- Object and array reference types
- null type for the null reference
- T for uninitialized registers



Selected Rules

- Maximum stack size (MS) and maximum number of parameters and local variables (MR) are stored in the classfile
- Rule for method invocation uses method signature (no jump)

iconst n:

$(S,R) \rightarrow (int.S,R)$, if $|S| < MS$

iload n:

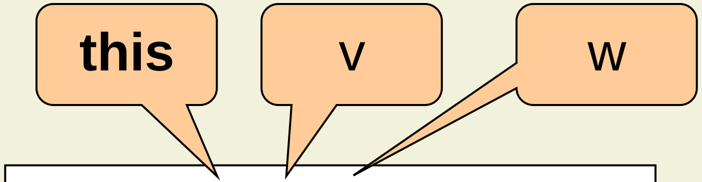
$(S,R) \rightarrow (int.S,R)$,
if $0 \leq n < MR \wedge R(n) = int \wedge |S| < MS$

astore n:

$(t.S,R) \rightarrow (S,R\{ n \leftarrow t \})$,
if $0 \leq n < MR \wedge t <: Object$

invokevirtual $C.m.\sigma$:
$(t'_n \ldots t'_1.t'.S,R) \rightarrow (r.S,R)$, if
$\sigma = r (t_1,\ldots,t_n) \wedge t' <: C \wedge t'_i <: t_i$

Example



int v;	iconst 5	([ ] , [C,T,T]) →
Object w;	istore 1	(int , [C,T,T]) →
v = 5;	aload 0	([ ] , [C,int,T]) →
w = **this**;	astore 2	(C , [C,int,T]) →
	return	([ ] , [C,int,C])

int v;	iconst 5	([ ] , [C,T,T]) →
Object w;	istore 1	(int , [C,T,T]) →
v = 5;	iload 1	([ ] , [C,int,T]) →
w = v;	astore 2	(int , [C,int,T])
	return	stuck

astore expects an object type on top of the stack!

Smallest Common Supertype

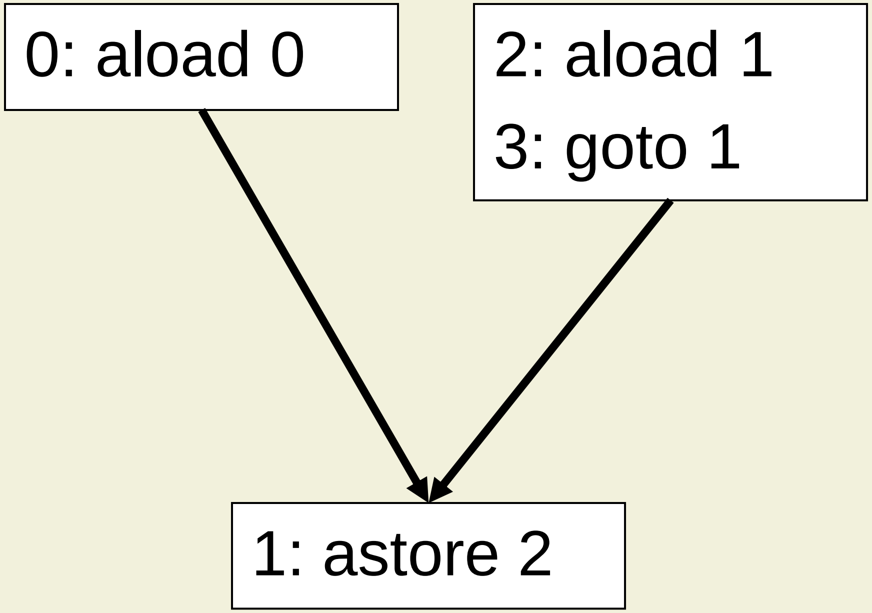
- Branches lead to joins in control flow

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```

Smallest Common Supertype

- Branches lead to joins in control flow
- Instructions can have several predecessors

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```



```
0: aload 0
```

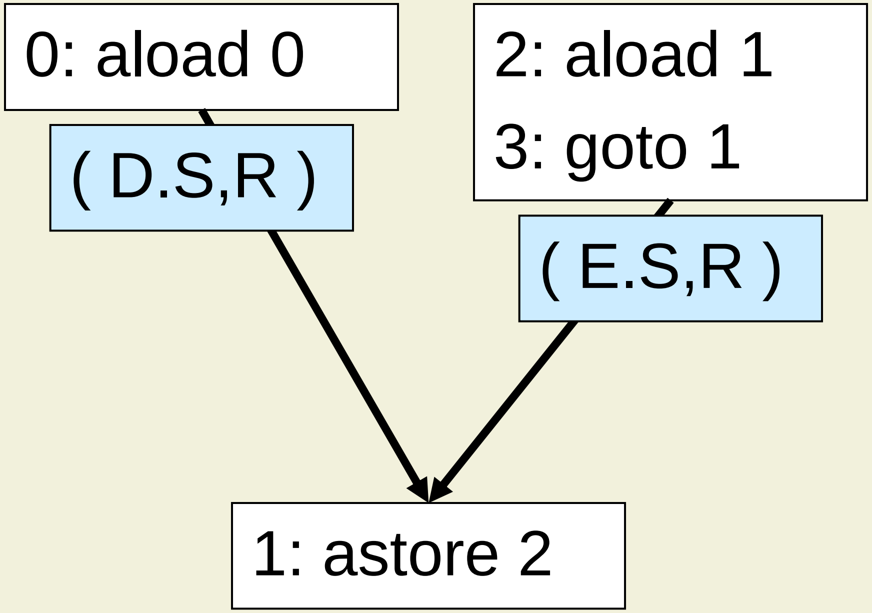
```
2: aload 1
3: goto 1
```

```
1: astore 2
```

Smallest Common Supertype

- Branches lead to joins in control flow
- Instructions can have several predecessors

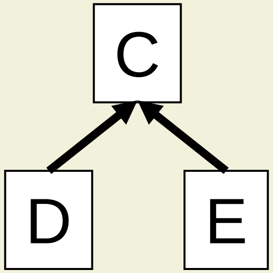
```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```

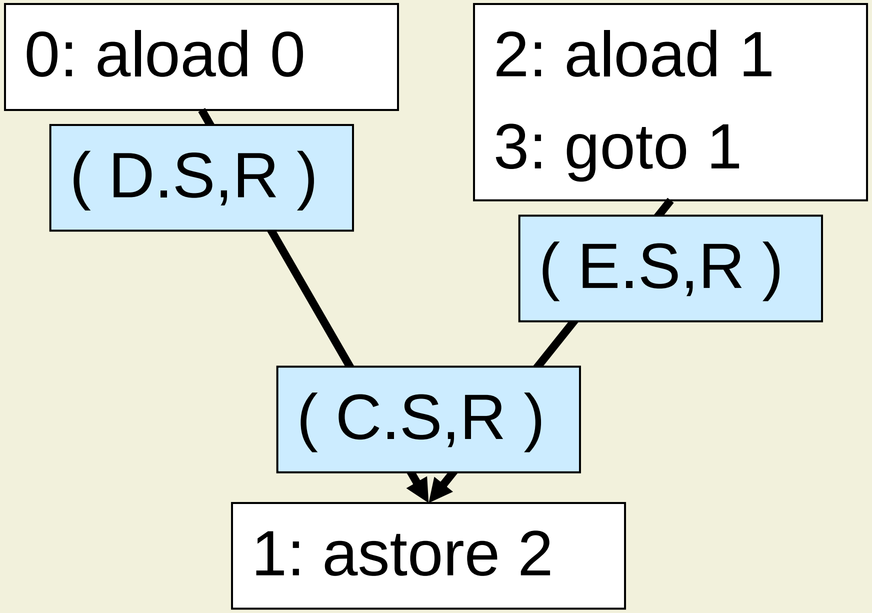


Smallest Common Supertype

- Branches lead to joins in control flow
- Instructions can have several predecessors
- Smallest common supertype is selected (T if no other common supertype exists)

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```





0: aload 0

(D.S,R)

2: aload 1
3: goto 1

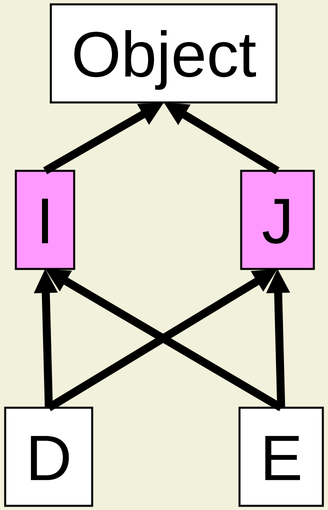
(E.S,R)

(C.S,R)

1: astore 2

Handling Multiple Subtyping

- With multiple subtyping, several smallest common supertypes may exist
- JVM solution
    - Ignore interfaces
    - Treat all interface types as Object
    - Works because of single inheritance of classes
- Problem
    - **invokeinterface** I.m cannot check whether target object implements I
    - Run-time check is necessary



Inference Algorithm

- Inference is a fixpoint iteration

```
in( 0 ) := ( [ ] , [ $P_0$,…,$P_n$,T,…,T ] )
worklist := { i | $instr_i$ is an instruction of the method }
while worklist $\neq \emptyset$ do
  i := min( worklist )
  remove i from worklist
  out( i ) := apply_rule( $instr_i$, in( i ) )
  foreach q in successors( i ) do
    in( q ) := pointwise_scs( in( q ), out( i ) )
    if in( q ) has changed then worklist := worklist $\cup$ { q }
  end
end
```

Pointwise SCS

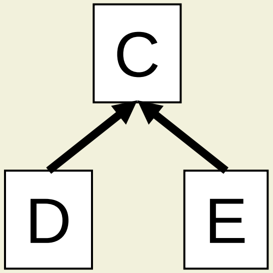
- scs(s,t) is the smallest common supertype of s and t

$$\begin{aligned}\text{pointwise\_scs}(\ & ([\, s_1.....s_k \,] , [\, t_0,\ldots,t_n \,]) , \\ & ([\, s'_1.....s'_k \,] , [\, t'_0,\ldots,t'_n \,])\ ) = \\ ([\, \text{scs}(s_1,s'_1).....\text{scs}(s_k,s'_k) \,] &, [\, \text{scs}(t_0,t'_0),\ldots,\text{scs}(t_n,t'_n) \,])\end{aligned}$$

- pointwise_scs is undefined for stacks of different heights
  - Bytecode verification results in an error

Inference Example

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```



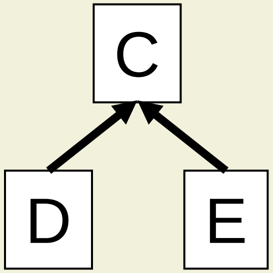
	in	out
0:	([ ] , [D,E,T])	
1:		
2:		
3:		

worklist

0 1 2 3

Inference Example

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```



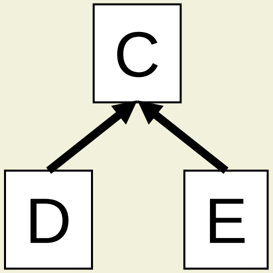
worklist

1 2 3

	in	out
0:	([ ] , [D,E,T])	
1:		
2:		
3:		

Inference Example

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```



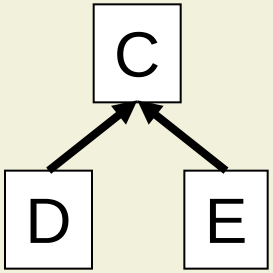
	in	out
0:	([ ] , [D,E,T])	([D] , [D,E,T])
1:		
2:		
3:		

worklist

1 2 3

Inference Example

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```



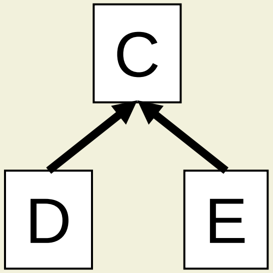
	in	out
0:	([ ] , [D,E,T])	([D] , [D,E,T])
1:	([D] , [D,E,T])	
2:		
3:		

worklist

1 2 3

Inference Example

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```



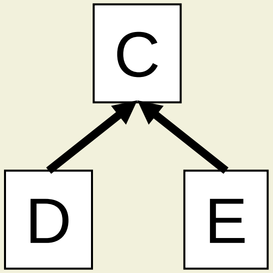
	in	out
0:	([ ] , [D,E,T])	([D] , [D,E,T])
1:	([D] , [D,E,T])	
2:		
3:		

worklist

2 3

Inference Example

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```



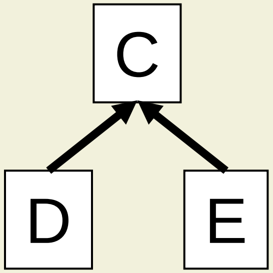
worklist

2 3

	in	out
0:	([ ] , [D,E,T])	([D] , [D,E,T])
1:	([D] , [D,E,T])	([ ] , [D,E,D])
2:		
3:		

Inference Example

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```



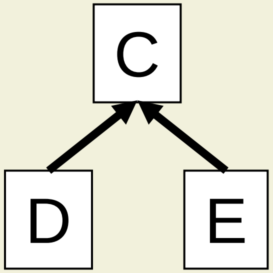
	in	out
0:	([ ] , [D,E,T])	([D] , [D,E,T])
1:	([D] , [D,E,T])	([ ] , [D,E,D])
2:	([ ] , [D,E,D])	
3:		

worklist

2 3

Inference Example

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```



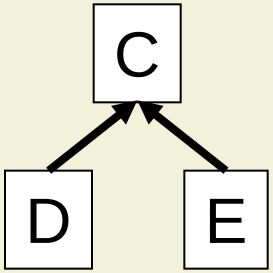
worklist

3

	in	out
0:	([ ] , [D,E,T])	([D] , [D,E,T])
1:	([D] , [D,E,T])	([ ] , [D,E,D])
2:	([ ] , [D,E,D])	
3:		

Inference Example

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```



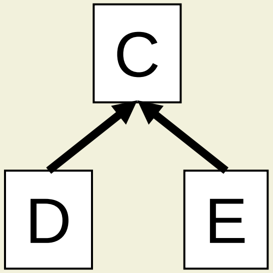
	in	out
0:	([ ] , [D,E,T])	([D] , [D,E,T])
1:	([D] , [D,E,T])	([ ] , [D,E,D])
2:	([ ] , [D,E,D])	([E] , [D,E,D])
3:		

worklist

3

Inference Example

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```



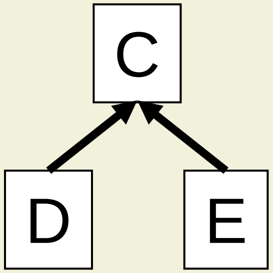
worklist

3

	in	out
0:	([ ] , [D,E,T])	([D] , [D,E,T])
1:	([D] , [D,E,T])	([ ] , [D,E,D])
2:	([ ] , [D,E,D])	([E] , [D,E,D])
3:	([E] , [D,E,D])	

Inference Example

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```

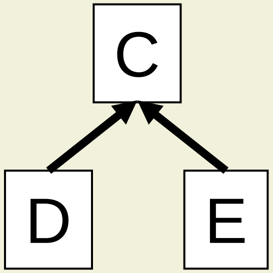


worklist

	in	out
0:	([ ] , [D,E,T])	([D] , [D,E,T])
1:	([D] , [D,E,T])	([ ] , [D,E,D])
2:	([ ] , [D,E,D])	([E] , [D,E,D])
3:	([E] , [D,E,D])	

Inference Example

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```

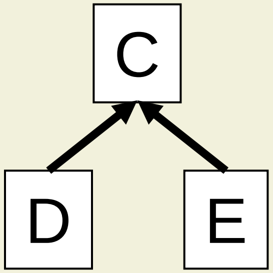


worklist

	in	out
0:	([ ] , [D,E,T])	([D] , [D,E,T])
1:	([D] , [D,E,T])	([ ] , [D,E,D])
2:	([ ] , [D,E,D])	([E] , [D,E,D])
3:	([E] , [D,E,D])	([E] , [D,E,D])

Inference Example

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```

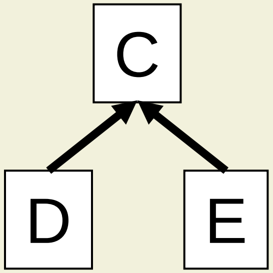


	in	out
0:	([ ] , [D,E,T])	([D] , [D,E,T])
1:	([D] , [D,E,T]) ([C] , [D,E,T])	([ ] , [D,E,D])
2:	([ ] , [D,E,D])	([E] , [D,E,D])
3:	([E] , [D,E,D])	([E] , [D,E,D])

worklist

Inference Example

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```



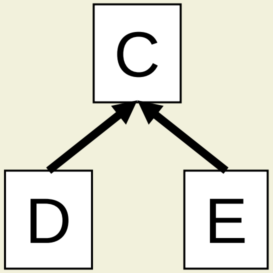
worklist

1

	in	out
0:	([ ] , [D,E,T])	([D] , [D,E,T])
1:	([D] , [D,E,T]) ([C] , [D,E,T])	([ ] , [D,E,D])
2:	([ ] , [D,E,D])	([E] , [D,E,D])
3:	([E] , [D,E,D])	([E] , [D,E,D])

Inference Example

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```

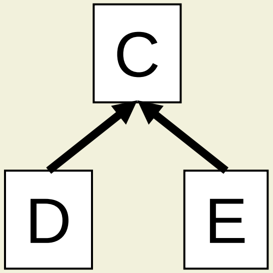


	in	out
0:	([ ] , [D,E,T])	([D] , [D,E,T])
1:	([D] , [D,E,T]) ([C] , [D,E,T])	([ ] , [D,E,D])
2:	([ ] , [D,E,D])	([E] , [D,E,D])
3:	([E] , [D,E,D])	([E] , [D,E,D])

worklist

Inference Example

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```

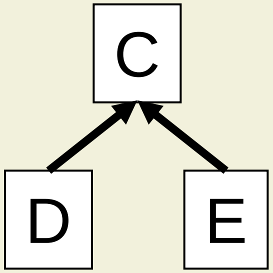


worklist

	in	out
0:	([ ] , [D,E,T])	([D] , [D,E,T])
1:	([D] , [D,E,T]) ([C] , [D,E,T])	([ ] , [D,E,D]) ([ ] , [D,E,C])
2:	([ ] , [D,E,D])	([E] , [D,E,D])
3:	([E] , [D,E,D])	([E] , [D,E,D])

Inference Example

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```

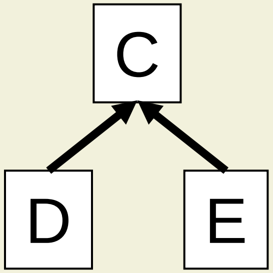


	in	out
0:	([ ] , [D,E,T])	([D] , [D,E,T])
1:	([D] , [D,E,T]) ([C] , [D,E,T])	([ ] , [D,E,D]) ([ ] , [D,E,C])
2:	([ ] , [D,E,D]) ([ ] , [D,E,C])	([E] , [D,E,D])
3:	([E] , [D,E,D])	([E] , [D,E,D])

worklist

Inference Example

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```



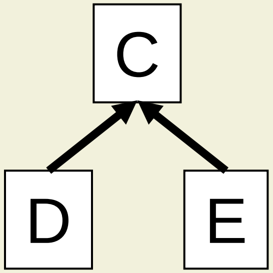
worklist

2

	in	out
0:	([ ] , [D,E,T])	([D] , [D,E,T])
1:	([D] , [D,E,T]) ([C] , [D,E,T])	([ ] , [D,E,D]) ([ ] , [D,E,C])
2:	([ ] , [D,E,D]) ([ ] , [D,E,C])	([E] , [D,E,D])
3:	([E] , [D,E,D])	([E] , [D,E,D])

Inference Example

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```

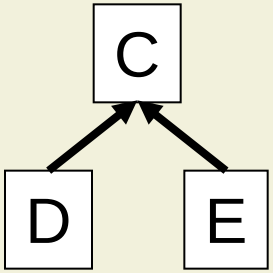


	in	out
0:	([ ] , [D,E,T])	([D] , [D,E,T])
1:	([D] , [D,E,T]) ([C] , [D,E,T])	([ ] , [D,E,D]) ([ ] , [D,E,C])
2:	([ ] , [D,E,D]) ([ ] , [D,E,C])	([E] , [D,E,D])
3:	([E] , [D,E,D])	([E] , [D,E,D])

worklist

Inference Example

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```

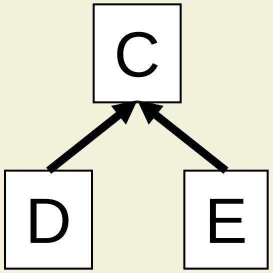


	in	out
0:	([ ] , [D,E,T])	([D] , [D,E,T])
1:	([D] , [D,E,T]) ([C] , [D,E,T])	([ ] , [D,E,D]) ([ ] , [D,E,C])
2:	([ ] , [D,E,D]) ([ ] , [D,E,C])	([E] , [D,E,D]) ([E] , [D,E,C])
3:	([E] , [D,E,D])	([E] , [D,E,D])

worklist

Inference Example

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```

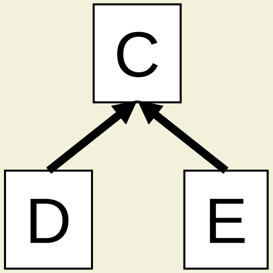


worklist

	in	out
0:	([ ] , [D,E,T])	([D] , [D,E,T])
1:	([D] , [D,E,T]) ([C] , [D,E,T])	([ ] , [D,E,D]) ([ ] , [D,E,C])
2:	([ ] , [D,E,D]) ([ ] , [D,E,C])	([E] , [D,E,D]) ([E] , [D,E,C])
3:	([E] , [D,E,D]) ([E] , [D,E,C])	([E] , [D,E,D])

Inference Example

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```



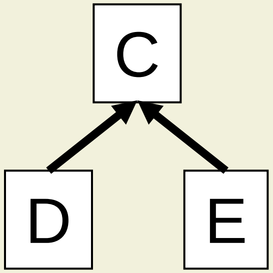
worklist

3

	in	out
0:	([ ] , [D,E,T])	([D] , [D,E,T])
1:	([D] , [D,E,T]) ([C] , [D,E,T])	([ ] , [D,E,D]) ([ ] , [D,E,C])
2:	([ ] , [D,E,D]) ([ ] , [D,E,C])	([E] , [D,E,D]) ([E] , [D,E,C])
3:	([E] , [D,E,D]) ([E] , [D,E,C])	([E] , [D,E,D])

Inference Example

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```

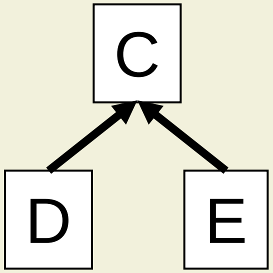


	in	out
0:	([ ] , [D,E,T])	([D] , [D,E,T])
1:	([D] , [D,E,T]) ([C] , [D,E,T])	([ ] , [D,E,D]) ([ ] , [D,E,C])
2:	([ ] , [D,E,D]) ([ ] , [D,E,C])	([E] , [D,E,D]) ([E] , [D,E,C])
3:	([E] , [D,E,D]) ([E] , [D,E,C])	([E] , [D,E,D])

worklist

Inference Example

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```

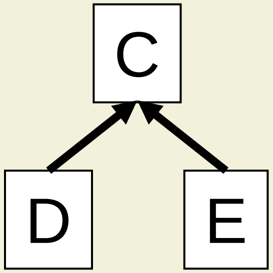


	in	out
0:	([ ] , [D,E,T])	([D] , [D,E,T])
1:	([D] , [D,E,T]) ([C] , [D,E,T])	([ ] , [D,E,D]) ([ ] , [D,E,C])
2:	([ ] , [D,E,D]) ([ ] , [D,E,C])	([E] , [D,E,D]) ([E] , [D,E,C])
3:	([E] , [D,E,D]) ([E] , [D,E,C])	([E] , [D,E,D]) ([E] , [D,E,C])

worklist

Inference Example

```
0: aload 0
1: astore 2
2: aload 1
3: goto 1
```



worklist

	in	out
0:	([ ] , [D,E,T])	([D] , [D,E,T])
1:	([D] , [D,E,T]) ([C] , [D,E,T]) ([C] , [D,E,T])	([ ] , [D,E,D]) ([ ] , [D,E,C])
2:	([ ] , [D,E,D]) ([ ] , [D,E,C])	([E] , [D,E,D]) ([E] , [D,E,C])
3:	([E] , [D,E,D]) ([E] , [D,E,C])	([E] , [D,E,D]) ([E] , [D,E,C])

Type Inference: Discussion

- Advantages
  - Determines the most general solution that satisfies the typing rules
  - Might be more general than what is permitted by compiler
  - Very little type information required in class file
- Disadvantages
  - Fixpoint computations may be slow
  - Solution for interfaces is imprecise and requires run-time checks
- Alternative: type checking (since Java 6)

Bytecode Verification via Type Checking

- Extend class file to store type information

([int] , [C,int,T])

- Type information can be declared for each bytecode instruction
- Type information required at the beginning of all basic blocks:
  - At jump target
  - At entry point of exception handler

Includes all join points

- Computation of SCS no longer necessary
  - Avoid fixpoint computation and interface problem

Type Checking Algorithm

- Use and check declared types wherever available
- Infer types otherwise

```
foreach basic block of a method body do
  in := types( start )
  foreach { i | instr_i is an instruction of basic block } do
    in := apply_rule( instr_i, in )
    foreach q in successors( i ) do
      if types( q ) is declared then
        check that in is assignable to types( q )
        in := types( q )
      end
  end
end
```

Type Checking Algorithm

- Use and check declared types wherever available
- Infer types otherwise

```
foreach basic block of a method body do
  in := types( start )
  foreach { i | $instr_i$ is an instruction of basic block } do
    in := apply_rule( $instr_i$, in )
    foreach q in successors( i ) do
      if types( q ) is declared then
        check that in is assignable to types( q )
        in := types( q )
      end
  end
end
```

Required types

Type Checking Algorithm

- Use and check declared types wherever available
- Infer types otherwise

```
foreach basic block of a method body do
  in := types( start )
  foreach { i | instr$_i$ is an instruction of basic block } do
    in := apply_rule( instr$_i$, in )
    foreach q in successors( i ) do
      if types( q ) is declared then
        check that in is assignable to types( q )
        in := types( q )
      end
    end
end
```

Required types

Check conditions and infer next configuration

Type Checking Algorithm

- Use and check declared types wherever available
- Infer types otherwise

```
foreach basic block of a method body do
  in := types( start )
  foreach { i | instr_i is an instruction of basic block } do
    in := apply_rule( instr_i, in )
    foreach q in successors( i ) do
      if types( q ) is declared then
        check that in is assignable to types( q )
        in := types( q )
      end
  end
end
```

Required types

Check conditions and infer next configuration

Check declared types

Type Checking Algorithm

- Use and check declared types wherever available
- Infer types otherwise

```
foreach basic block of a method body do
  in := types( start )
  foreach { i | $instr_i$ is an instruction of basic block } do
    in := apply_rule( $instr_i$, in )
    foreach q in successors( i ) do
      if types( q ) is declared then
        check that in is assignable to types( q )
        in := types( q )
      end
  end
end
```

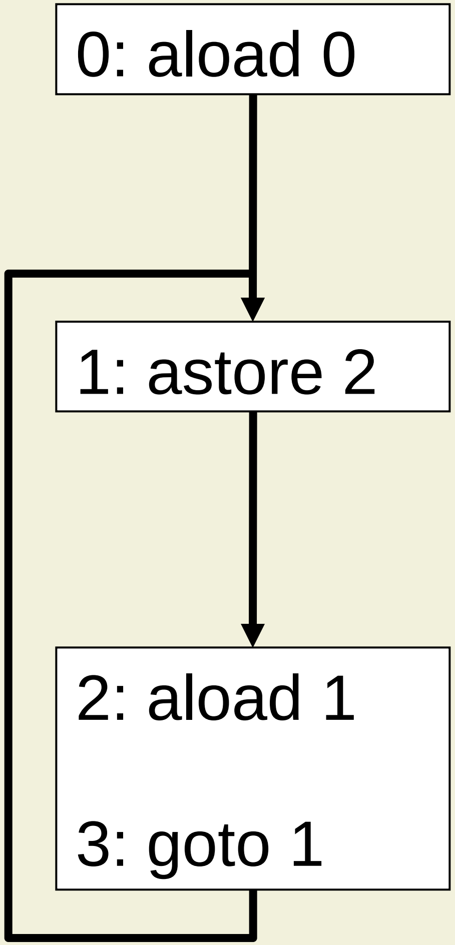
Required types

Check conditions and infer next configuration

Check declared types

Use declared types if $instr_i$ is not a jump

Type Checking Example

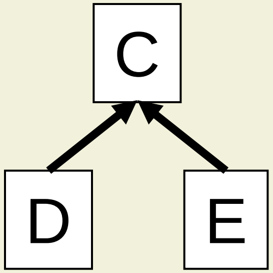


0: aload 0

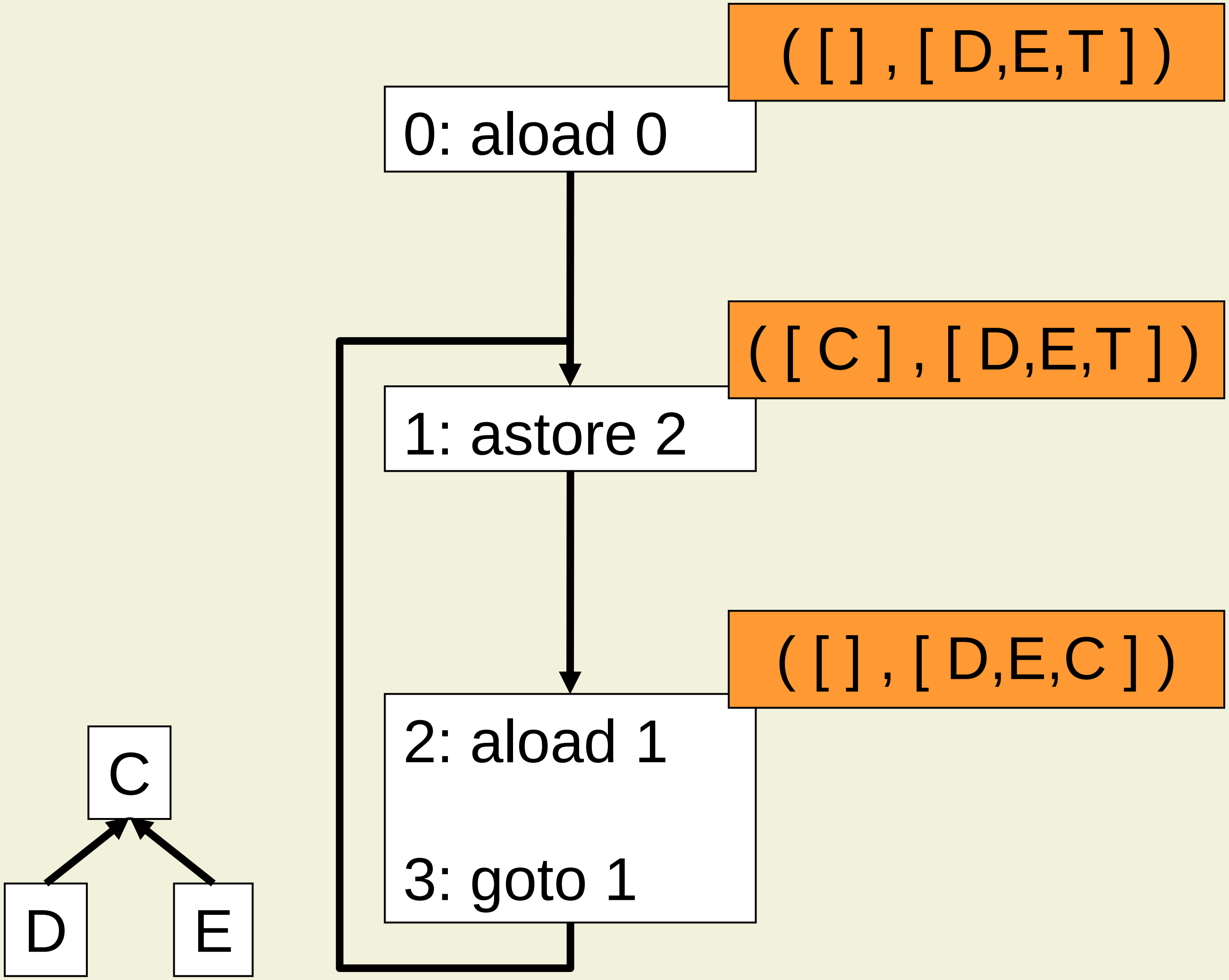
1: astore 2

2: aload 1

3: goto 1



Type Checking Example



([ ] , [D,E,T])

0: aload 0

([C] , [D,E,T])

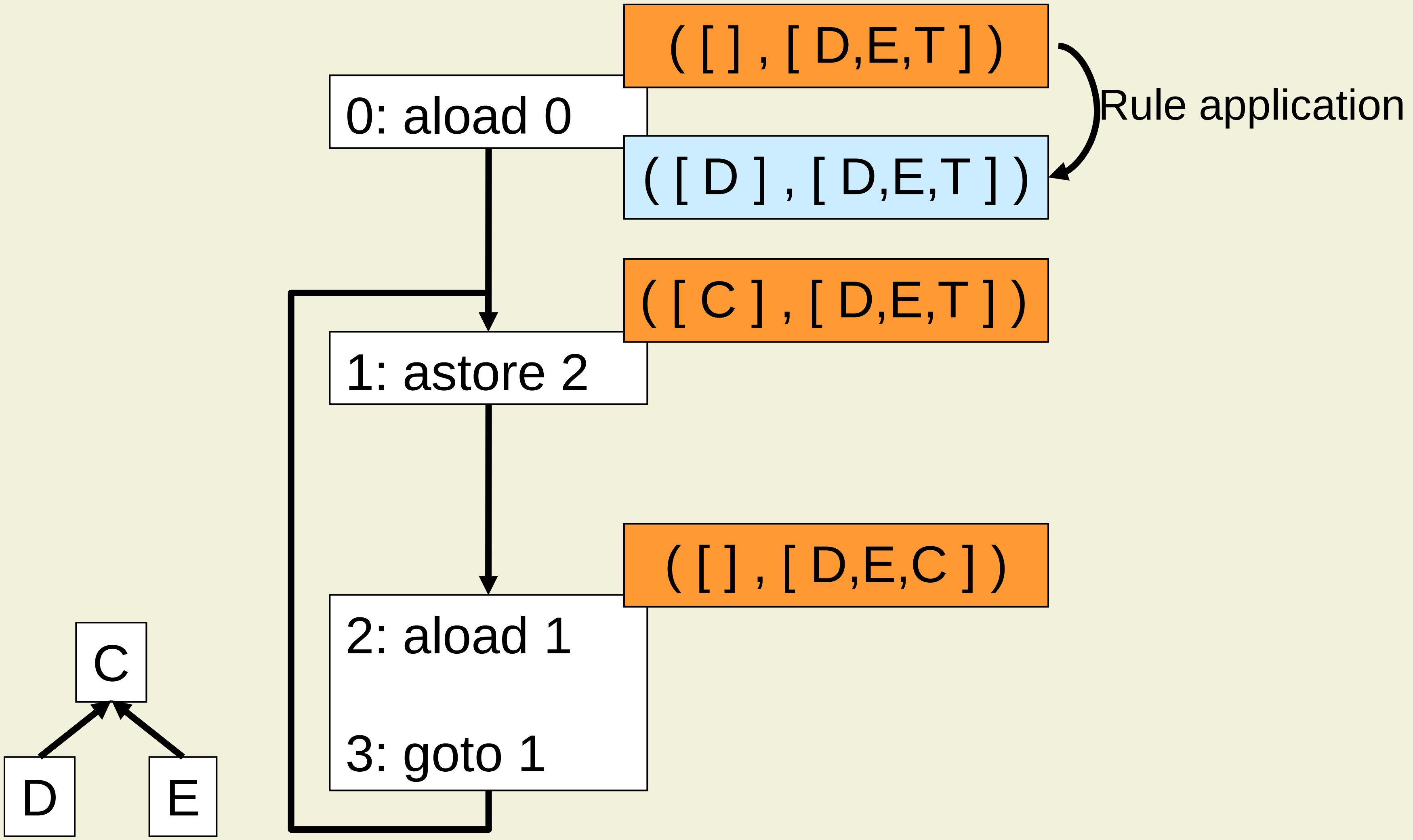
1: astore 2

([ ] , [D,E,C])

2: aload 1

3: goto 1

Type Checking Example



([ ] , [D,E,T])

0: aload 0

([D] , [D,E,T])

Rule application

([C] , [D,E,T])

1: astore 2

([ ] , [D,E,C])

2: aload 1

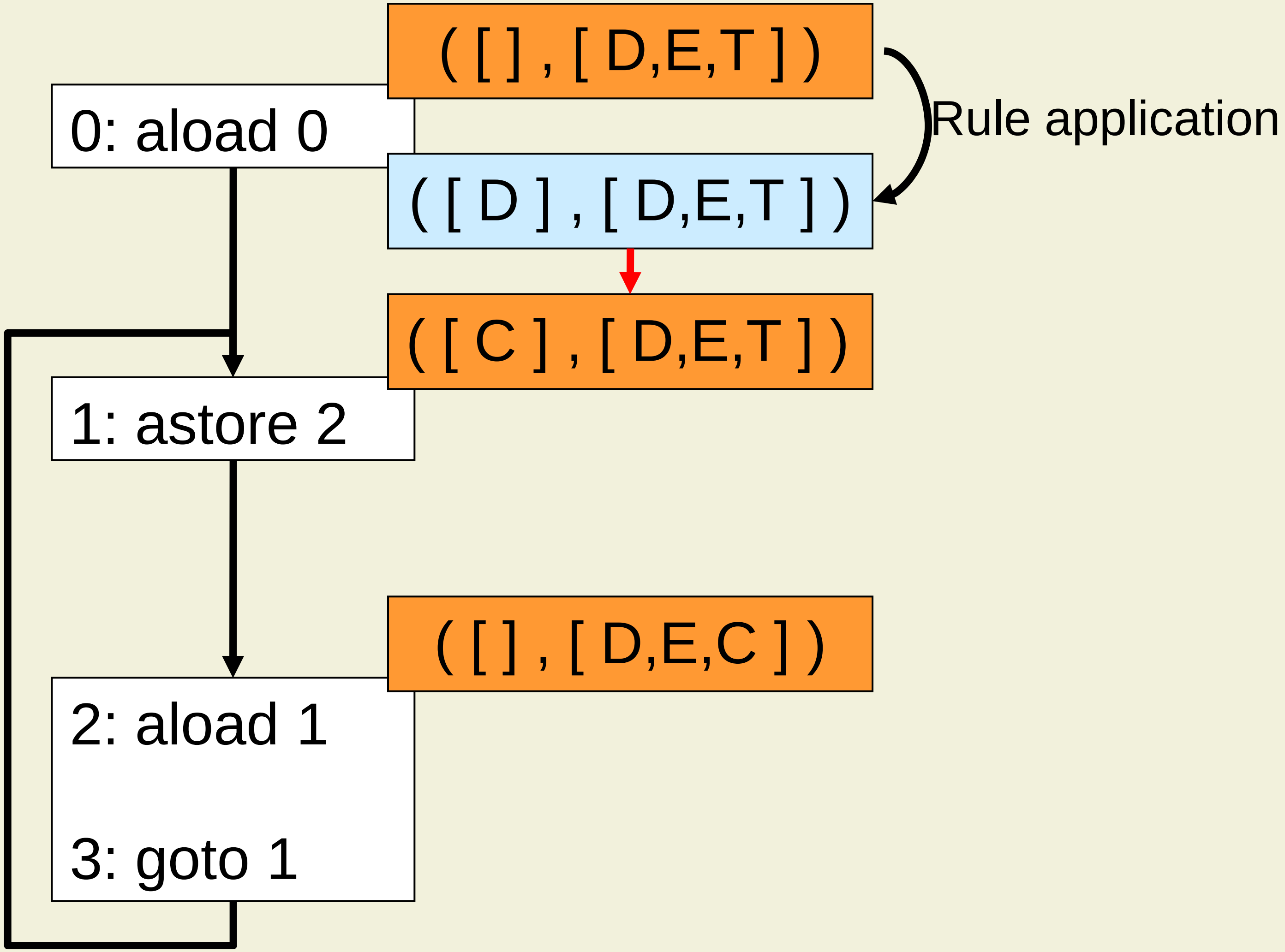
3: goto 1

C

D

E

Type Checking Example



0: aload 0

1: astore 2

2: aload 1

3: goto 1

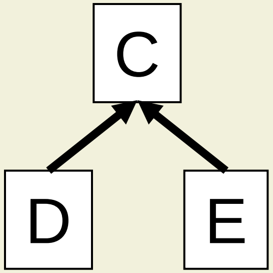
([ ] , [D,E,T])

Rule application

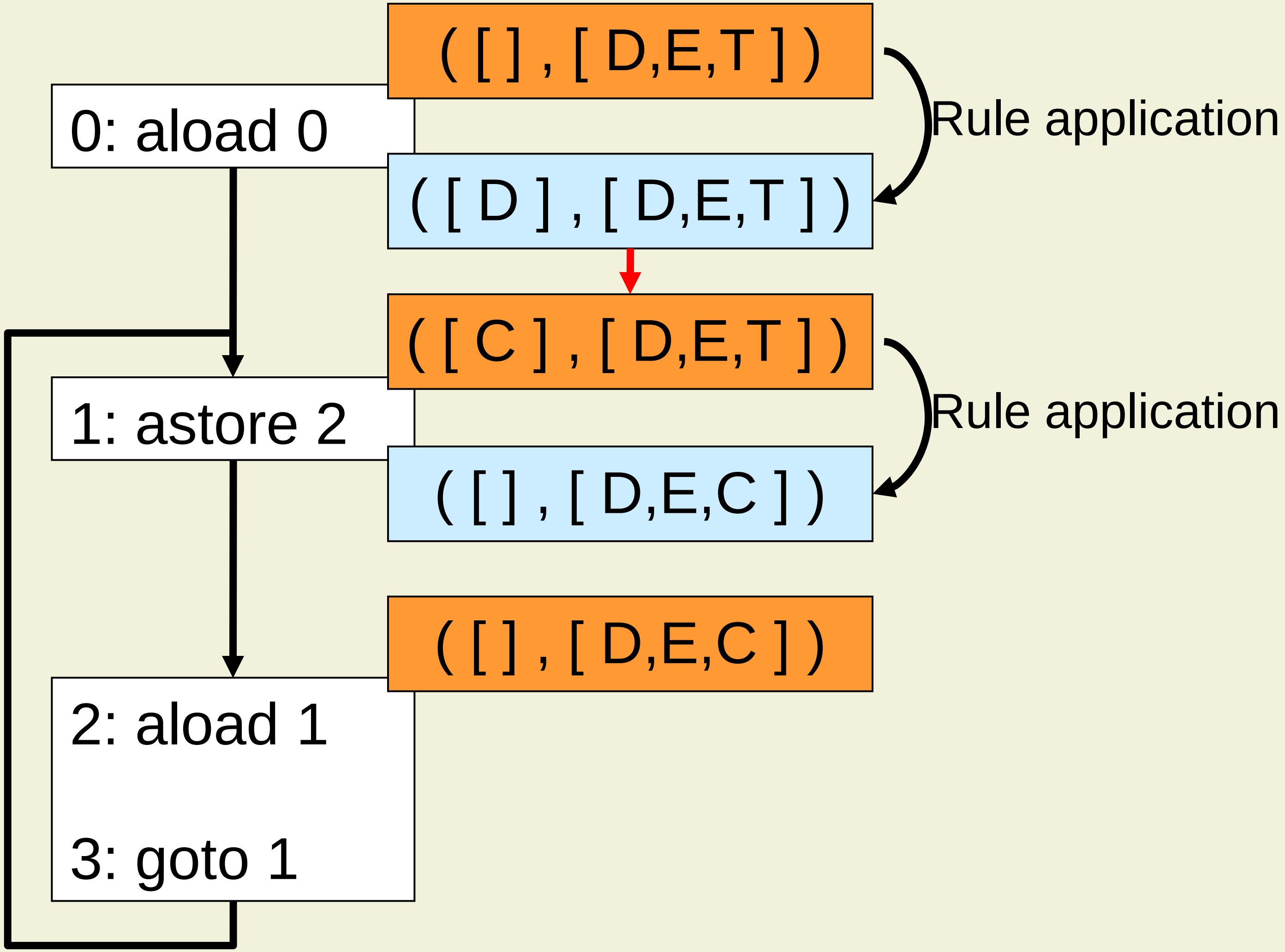
([D] , [D,E,T])

([C] , [D,E,T])

([ ] , [D,E,C])



Type Checking Example



0: aload 0

1: astore 2

2: aload 1

3: goto 1

([ ] , [D,E,T])

([D] , [D,E,T])

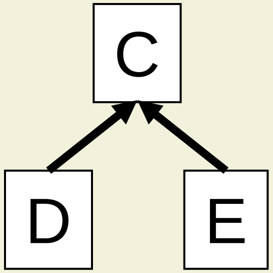
([C] , [D,E,T])

([ ] , [D,E,C])

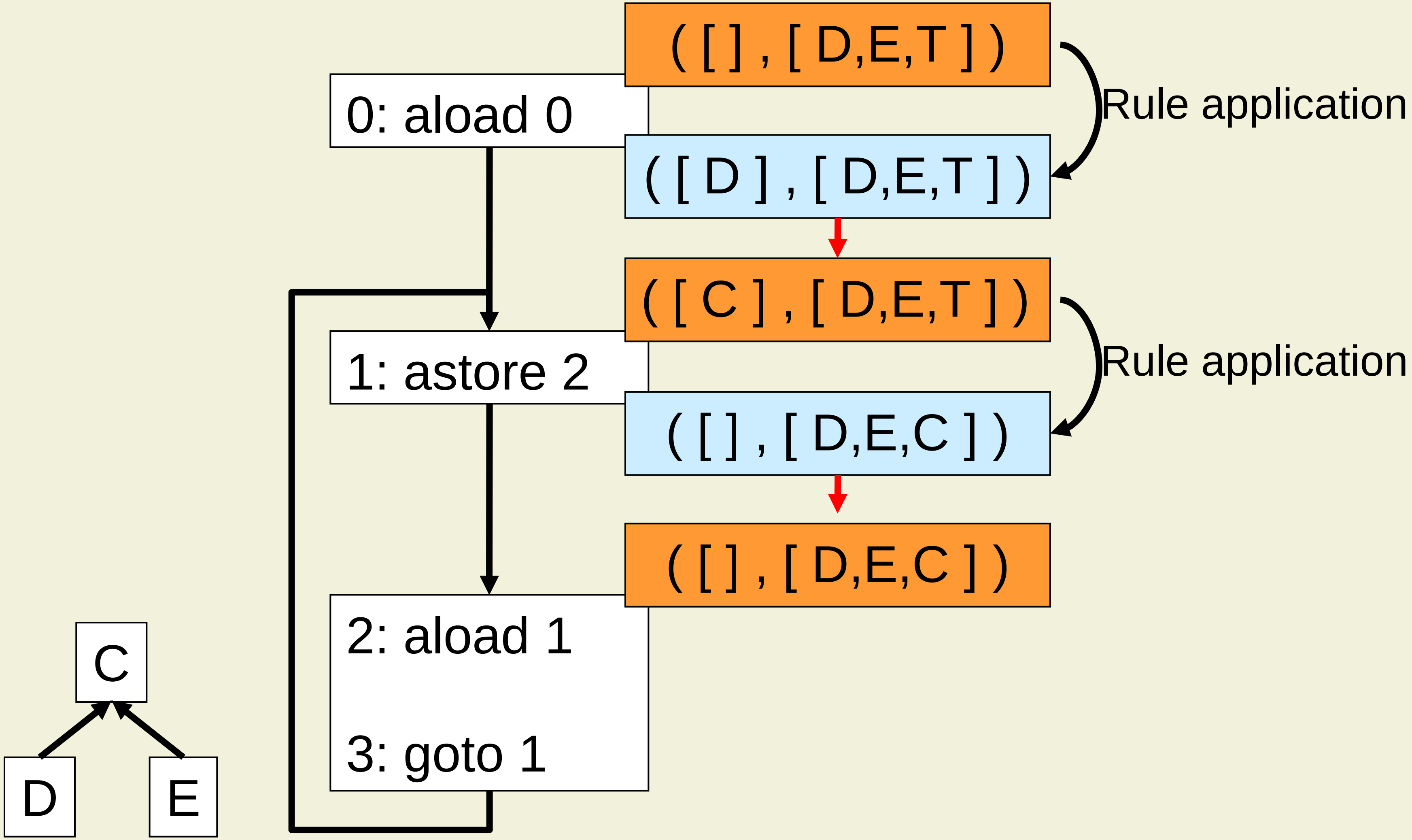
([ ] , [D,E,C])

Rule application

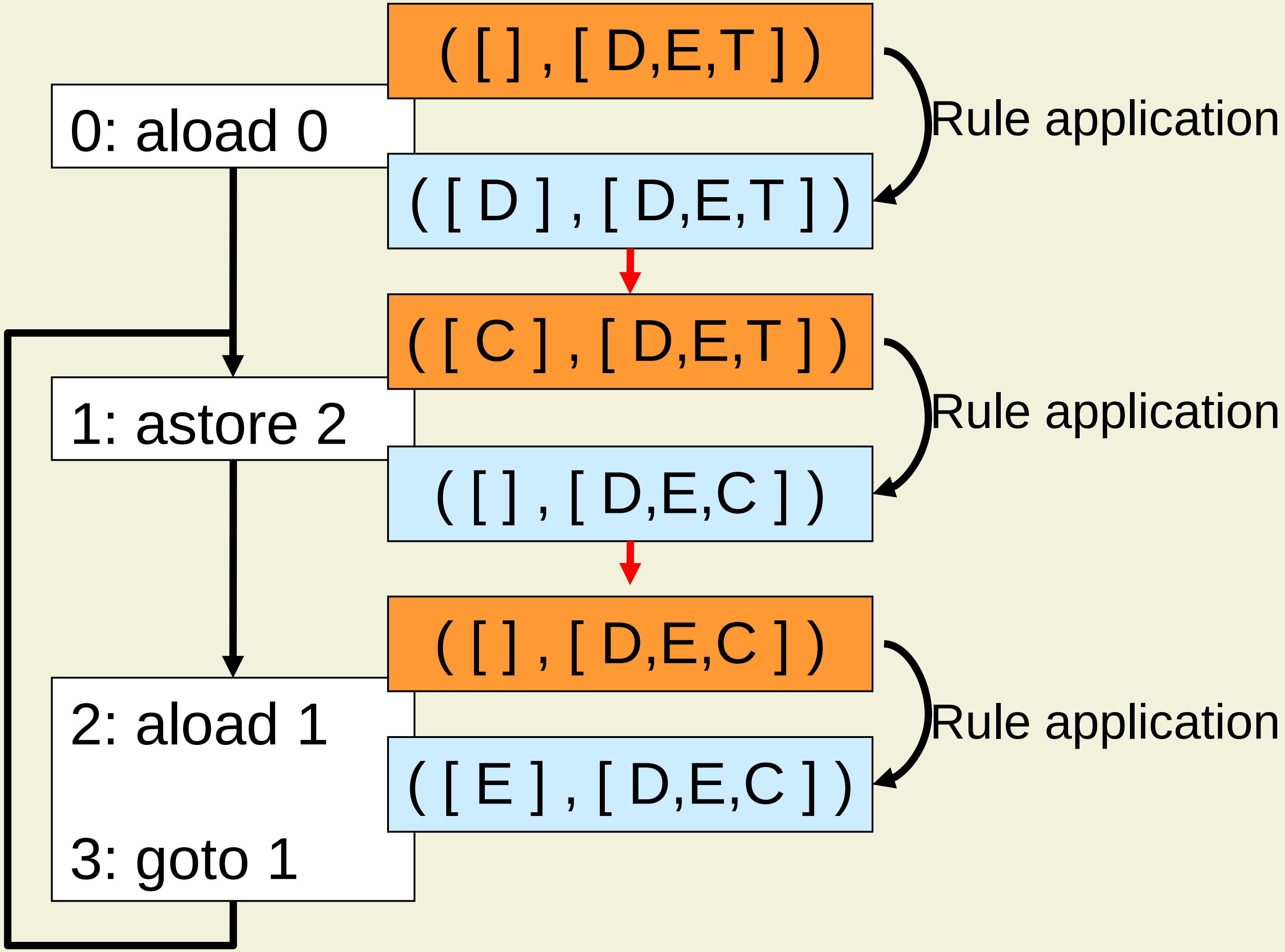
Rule application



Type Checking Example



Type Checking Example



0: aload 0

([ ] , [D,E,T])

Rule application

([D] , [D,E,T])

1: astore 2

([C] , [D,E,T])

Rule application

([ ] , [D,E,C])

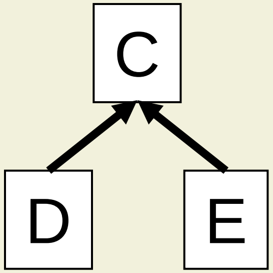
2: aload 1

3: goto 1

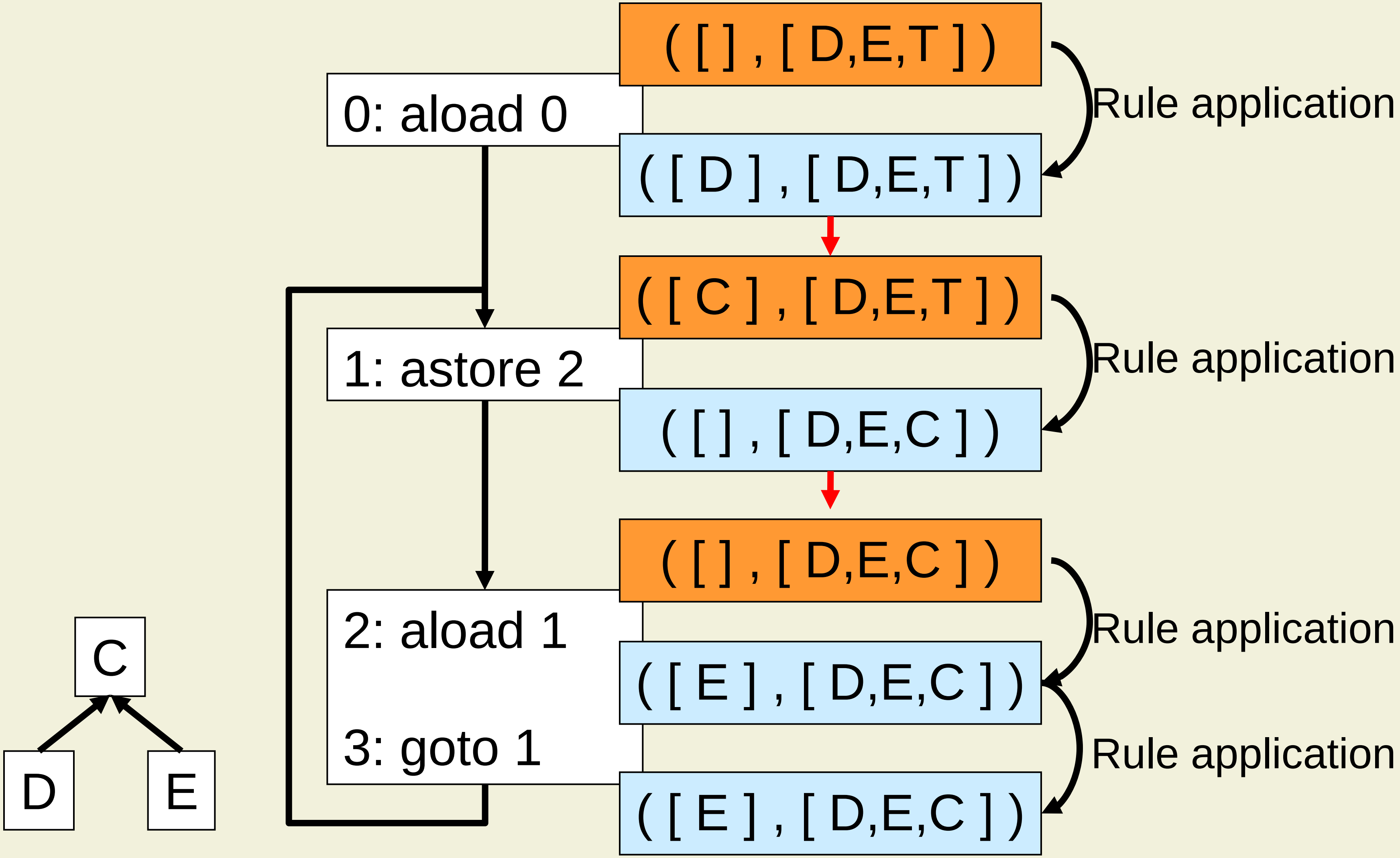
([ ] , [D,E,C])

Rule application

([E] , [D,E,C])



Type Checking Example



0: aload 0

1: astore 2

2: aload 1

3: goto 1

([ ] , [D,E,T])

Rule application

([D] , [D,E,T])

([C] , [D,E,T])

Rule application

([ ] , [D,E,C])

([ ] , [D,E,C])

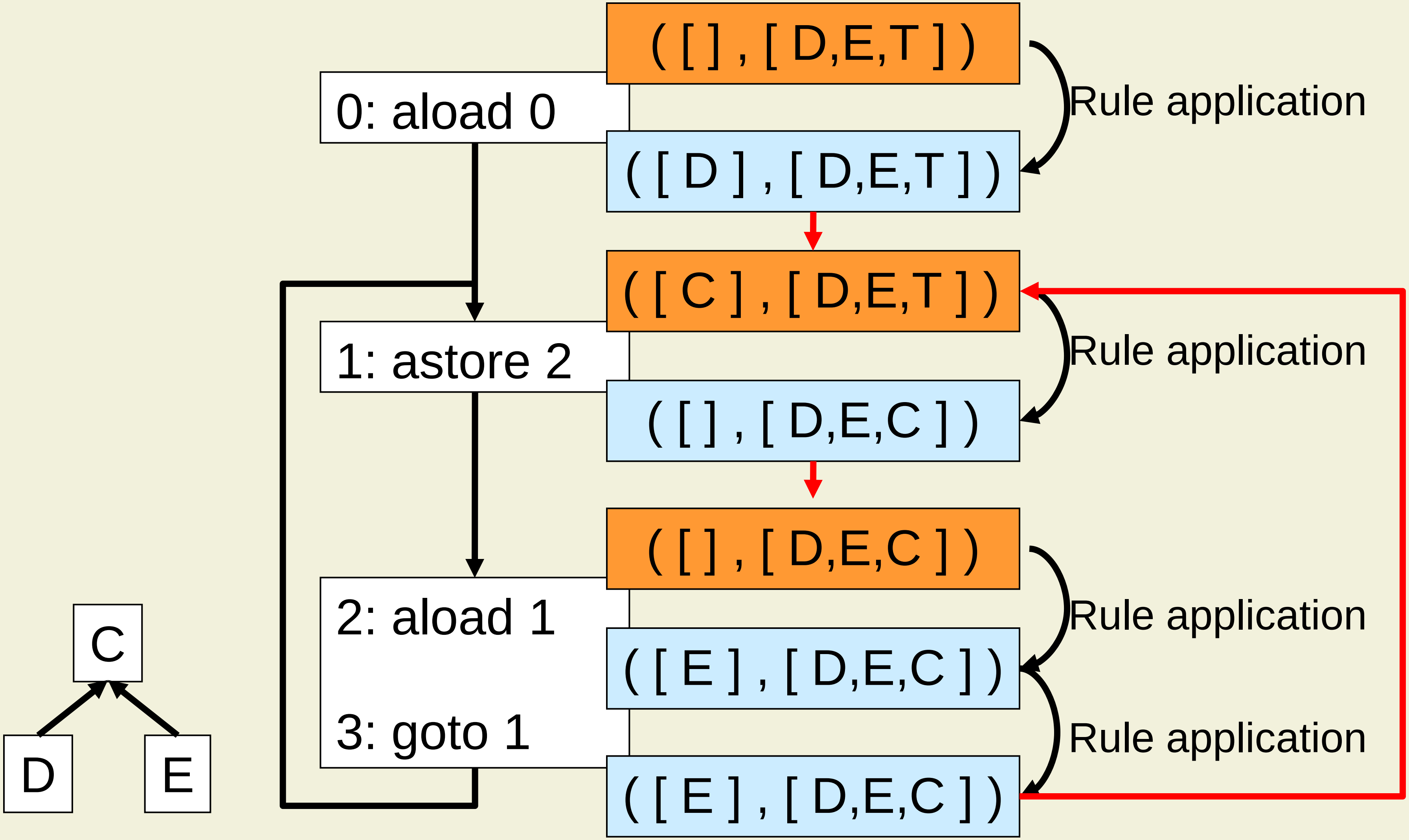
Rule application

([E] , [D,E,C])

Rule application

([E] , [D,E,C])

Type Checking Example



0: aload 0

1: astore 2

2: aload 1

3: goto 1

([ ] , [D,E,T])

Rule application

([D] , [D,E,T])

([C] , [D,E,T])

Rule application

([ ] , [D,E,C])

([ ] , [D,E,C])

Rule application

([E] , [D,E,C])

Rule application

([E] , [D,E,C])

Bytecode Verification: Summary

- Bytecode verification enables secure mobile code
  - For programs written in typed bytecode
- Bytecode verification can be done via type inference or type checking
- Some run-time type checks are still necessary
  - For instance, casts and co-variant arrays

Type Inference for Source Programs

- Type inference can also be done on source code
  - For example, C# 3.0 and Scala infer types of local variables
  - Reduce annotation overhead, especially with generics

```
def sum( a: Array[ Int ] ): Int = {
  val it = a.elements
  var s = 0;
  while( it.hasNext ) { s = s + it.next }
  s
}
```

Scala

Type Inference for Source Programs

- Type inference can also be done on source code
  - For example, C# 3.0 and Scala infer types of local variables
  - Reduce annotation overhead, especially with generics

```
def sum( a: Array[ Int ] ): Int = {
  val it = a.elements
  var s = 0;
  while( it.hasNext ) { s = s + it.next }
  s
}
```

Scala

```
def client = {
  var a = 1
  a = "Hello"
}
```

Scala

Type Inference for Source Programs

- Type inference can also be done on source code
  - For example, C# 3.0 and Scala infer types of local variables
  - Reduce annotation overhead, especially with generics
- Type annotations can still be used to support inference

```
def sum( a: Array[ Int ] ): Int = {
  val it = a.elements
  var s = 0;
  while( it.hasNext ) { s = s + it.next }
  s
}
```

Scala

```
def client = {
  var a = 1
  a = "Hello"
}
```

Scala

```
def client = {
  var a: Any = 1
  a = "Hello"
}
```

Scala

Inference of Method and Field Types

- Inference of method signatures generally requires knowledge of all implementations
- Inference of field types generally requires knowledge of all assignments to the field
- Inference of these types is non-modular
  - Or based on speculation

```
class A {
  var f = 5;
  def foo( p: Int ) = {
    p
  }
}
```

Scala

Inference of Method and Field Types

- Inference of method signatures generally requires knowledge of all implementations
- Inference of field types generally requires knowledge of all assignments to the field
- Inference of these types is non-modular
  - Or based on speculation

```
class A {
  var f = 5;
  def foo( p: Int ) = {
    p
  }
}
```

Scala

Inference:
f: Int

Inference of Method and Field Types

- Inference of method signatures generally requires knowledge of all implementations
- Inference of field types generally requires knowledge of all assignments to the field
- Inference of these types is non-modular
  - Or based on speculation

```
class A {
  var f = 5;
  def foo( p: Int ) = {
    p
  }
}
```

ala

Inference:
f: Int

Inference:
foo returns Int

Inference of Method and Field Types

- Inference of method signatures generally requires knowledge of all implementations
- Inference of field types generally requires knowledge of all assignments to the field
- Inference of these types is non-modular
  - Or based on speculation

```
class A {
  var f = 5;
  def foo( p: Int ) = {
    p
  }
}
```

ala

Inference:
f: Int

Inference:
foo returns Int

```
class B extends A {
  f = "Hello";
  override def foo( p: Int ) = {
    "Hello"
  }
}
```

Scala

4. Types

4.1 Bytecode Verification

4.2 Parametric Polymorphism

Polymorphism Revisited

- Not all polymorphic code is best expressed using subtype polymorphism
- Recovering precise type information requires downcasts

```
class Queue {
  Object elem;
  Queue next;
  void enqueue( Object e ) { … }
  Object dequeue( ) { … }
}
```

Java

```
Queue q = new Queue( );
String s = “Hello”;
q.enqueue( s );
String t = ( String ) q.dequeue( );
```

Java

Polymorphism Revisited

- Not all polymorphic code is best expressed using subtype polymorphism
- Recovering precise type information requires downcasts
- Subtype relations are sometimes not desirable
  - E.g., covariant arrays

```
class Queue {
  Object elem;
  Queue next;
  void enqueue( Object e ) { … }
  Object dequeue( ) { … }
}
```

Java

```
Queue q = new Queue( );
String s = “Hello”;
q.enqueue( s );
String t = ( String ) q.dequeue( );
```

Java

```
static void fill( Object[ ] a, Object val )
{ … }
```

Java

Parametric Polymorphism

- Classes and methods can be parameterized with types
- Clients provide instantiations for type parameters
- Modularity: generic code is type checked once and for all (without knowing the instantiations)

```
class Queue<T> {
  T elem;
  Queue<T> next;
  void enqueue( T e ) { … }
  T dequeue( ) { … }
}
```

Java

```
Queue<String> q;
q = new Queue<String>( );
String s = “Hello”;
q.enqueue( s );
String t = q.dequeue( );
```

Java

```
static <T> void fill( T[ ] a, T val )
{ … }
```

Java

Type Checking Generic Code

- Type checking a generic class often requires information about its type arguments
  - Availability of methods
- Constraints can be expressed by specifying upper bounds on type parameters

```
class Queue<T> {
  T elem;
  Queue<T> next;

  void enqueue( T e ) {
    if( next == null ) { … }
    else {
      if( e.compareTo( elem ) <= 0 ) {
        next.enqueue( elem );
        elem = e;
      } else next.enqueue( e );
    }
  }
  …
}
```

Java

Upper Bounds: Example

```
interface Comparable<T> {
  int compareTo( T o );
}
```

Java

```
class Queue<T extends Comparable<T>> {
  T elem;
  Queue<T> next;

  void enqueue( T e ) {
    if( next == null ) { … }
    else {
      if( e.compareTo( elem ) <= 0 ) {
        next.enqueue( elem );
        elem = e;
      } else next.enqueue( e );
    }
  }
  …
}
```

Java

Upper Bounds: Example

```
interface Comparable<T> {
  int compareTo( T o );
}
```

Java

```
class Queue<T extends Comparable<T>> {
  T elem;
  Queue<T> next;

  void enqueue( T e )
    if( next == null ) { ..
    else {
      if( e.compareTo( elem ) <= 0 ) {
        next.enqueue( elem );
        elem = e;
      } else next.enqueue( e );
    }
  }
  …
}
```

Java

Typecheck under the assumption
T <: Comparable<T>

Upper Bounds: Example

```
interface Comparable<T> {
  int compareTo( T o );
}
```

Java

```
Queue<String> q;
// String implements
      Comparable<String>
```

Java

```
class Queue<T extends Comparable<T>> {
  T elem;
  Queue<T> next;

  void enqueue( T e )
    if( next == null ) { ..
    else {
      if( e.compareTo( elem ) <= 0 ) {
        next.enqueue( elem );
        elem = e;
      } else next.enqueue( e );
    }
  }
  …
}
```

Java

Typecheck under the assumption
T <: Comparable<T>

Upper Bounds: Example

```
interface Comparable<T> {
  int compareTo( T o );
}
```

Java

```
class Queue<T extends Comparable<T>> {
  T elem;
  Queue<T> next;

  void enqueue( T e )
    if( next == null ) { ..
    else {
      if( e.compareTo( elem ) <= 0 ) {
        next.enqueue( elem );
        elem = e;
      } else next.enqueue( e );
    }
  }
  …
}
```

Java

Typecheck under the assumption
T <: Comparable<T>

```
Queue<String> q;
// String implements
      Comparable<String>
```

Java

```
Queue<Person> q;
// Person does not
      implement
      Comparable<Person>
```

Java

Subtyping and Generics

```
class Queue<T extends Comparable<T>> { … }
```

Subtyping and Generics

```
class Queue<T extends Comparable<T>> { … }
```

- Generic types are subtypes of their declared supertypes

```
Object o = new Queue<String>( );
```

Subtyping and Generics

```
class Queue<T extends Comparable<T>> { … }
```

- Generic types are subtypes of their declared supertypes
- Type variables are subtypes of their upper bounds

```
Object o = new Queue<String>( );
```

```
void foo( T p ) {
  Comparable<T> v = p;
}
```

Subtyping and Generics

```
class Queue<T extends Comparable<T>> { … }
```

- Generic types are subtypes of their declared supertypes
- Type variables are subtypes of their upper bounds
- How about different instantiations of the same generic class?

```
Object o = new Queue<String>( );
```

```
void foo( T p ) {
  Comparable<T> v = p;
}
```

```
List<Person> o;
o = new List<Student>( );
o = new List<Object>( );
```

Covariant Type Arguments

- Covariance:
  If S <: T then
  C<S> <: C<T>

```
class Queue<T> {
  void enqueue( T e ) { … }
  T dequeue( ) { … }
}
```

Covariant Type Arguments

- Covariance:
  If S <: T then
  C<S> <: C<T>

```
class Queue<T> {
  void enqueue( T e ) { … }
  T dequeue( ) { … }
}
```

```
void put( Queue<Object> q ) {
  q.enqueue( “Hello” );
}
```

Covariant Type Arguments

- Covariance:
  If S <: T then
  C<S> <: C<T>

```
class Queue<T> {
  void enqueue( T e ) { … }
  T dequeue( ) { … }
}
```

```
void put( Queue<Object> q ) {
  q.enqueue( “Hello” );
}
```

Not type safe if q had type Queue<Integer>

Covariant Type Arguments

- Covariance:
  If S <: T then
  C<S> <: C<T>

```
class Queue<T> {
  void enqueue( T e ) { … }
  T dequeue( ) { … }
}
```

```
void put( Queue<Object> q ) {
  q.enqueue( “Hello” );
}
```

Not type safe if q had type Queue<Integer>

```
Object get( Queue<Object> q ) {
  return q.dequeue( );
}
```

Covariant Type Arguments

- Covariance:
  If S <: T then
  C<S> <: C<T>

```
class Queue<T> {
  void enqueue( T e ) { … }
  T dequeue( ) { … }
}
```

- Covariance is unsafe when a generic type argument is used for variables that are written by clients
  - Mutable fields
  - Method arguments

```
void put( Queue<Object> q ) {
  q.enqueue( “Hello” );
}
```

Not type safe if q had type Queue<Integer>

```
Object get( Queue<Object> q ) {
  return q.dequeue( );
}
```

Contravariant Type Arguments

- Contravariance:
  If S <: T then
  C<T> <: C<S>

```
class Queue<T> {
 void enqueue( T e ) { … }
 T dequeue( ) { … }
}
```

Contravariant Type Arguments

- Contravariance:
  If S <: T then
  C<T> <: C<S>

```
class Queue<T> {
  void enqueue( T e ) { … }
  T dequeue( ) { … }
}
```

```
void put( Queue<String> q ) {
  q.enqueue( “Hello” );
}
```

Contravariant Type Arguments

- Contravariance:
  If S <: T then
  C<T> <: C<S>

```
class Queue<T> {
  void enqueue( T e ) { … }
  T dequeue( ) { … }
}
```

```
void put( Queue<String> q ) {
  q.enqueue( “Hello” );
}
```

```
String get( Queue<String> q ) {
  return q.dequeue( );
}
```

Contravariant Type Arguments

- Contravariance:
  If S <: T then
  C<T> <: C<S>

```
class Queue<T> {
  void enqueue( T e ) { … }
  T dequeue( ) { … }
}
```

```
void put( Queue<String> q ) {
  q.enqueue( “Hello” );
}
```

```
String get( Queue<String> q ) {
  return q.dequeue( );
}
```

Not type safe if q had type Queue<Object>

Contravariant Type Arguments

- Contravariance:
  If S <: T then
  C<T> <: C<S>

```
class Queue<T> {
  void enqueue( T e ) { … }
  T dequeue( ) { … }
}
```

- Contravariance is unsafe when a generic type argument is used for variables that are read by clients
  - Fields
  - Method results

```
void put( Queue<String> q ) {
  q.enqueue( "Hello" );
}
```

```
String get( Queue<String> q ) {
  return q.dequeue( );
}
```

Not type safe if q had type Queue<Object>

Java/C# Solution: Non-Variance

- Generic types in Java/C# are non-variant (neither co- nor contravariant)
- Non-variance is statically type safe
  - No run-time checks needed

```
class Queue<T> {
  void enqueue( T e ) { … }
  T dequeue( ) { … }
}
```

Java

```
Queue<Object> o;
o = new Queue<String>( );
```

Java

```
Queue<String> o;
o = new Queue<Object>( );
```

Java

Java/C# Solution: Non-Variance

- Generic types in Java/C# are non-variant (neither co- nor contravariant)
- Non-variance is statically type safe
  - No run-time checks needed
- Non-variance is sometimes overly restrictive

```
class Queue<T> {
  void enqueue( T e ) { … }
  T dequeue( ) { … }
}
```

Java

```
Queue<Object> o;
o = new Queue<String>( );
```

Java

```
Queue<String> o;
o = new Queue<Object>( );
```

Java

```
class Random<T> {
  T next( ) { … }
}
```

Java/C#: Generics vs. Arrays

- Recall: Java/C# arrays are covariant
- But an array T[ ] is not much different from a class Array<T>

```
Object[ ] o;
o = new String[ 5 ];
```

Java

```
Queue<Object> o;
o = new Queue<String>( );
```

Java

Java/C#: Generics vs. Arrays

- Recall: Java/C# arrays are covariant
- But an array T[ ] is not much different from a class Array<T>
- Run-time checks
  - Covariant arrays require run-time checks for each update
  - Covariant generics would need checks for field updates and argument passing
- Covariant generics would require more run-time checks in more bytecode instructions

```
Object[ ] o;
o = new String[ 5 ];
```

Java

```
Queue<Object> o;
o = new Queue<String>( );
```

Java

Eiffel Solution: Covariance

- Generic types in Eiffel are covariant
- Design is consistent with covariance for method arguments and fields
  - But not statically type safe

```
class Queue[ T ] … end
```

Eiffel

```
o: Queue[ ANY ]
s: Queue[ STRING ]
create s.make
o := s
```

Eiffel

```
o: Queue[ ANY ]
s: Queue[ STRING ]
create o.make
s := o
```

Eiffel

Scala Solution: Variance Annotations

- By default, generic types in Scala are non-variant

```
class Queue[ T ] {
  def enqueue( e: T ) = { … }
  def dequeue: T = { … }
}
```

Scala

```
Queue[ AnyRef ] o;
o = new Queue[ String ]( );
```

Scala

```
Queue[ String ] o;
o = new Queue[ AnyRef ]( );
```

Scala

Scala Solution: Variance Annotations

- By default, generic types in Scala are non-variant
- Programmers can supply variance annotations to allow co- and contravariance

```
class Queue[ T ] {
  def enqueue( e: T ) = { … }
  def dequeue: T = { … }
}
```

Scala

```
Queue[ AnyRef ] o;
o = new Queue[ String ]( );
```

Scala

```
Queue[ String ] o;
o = new Queue[ AnyRef ]( );
```

Scala

Scala Solution: Variance Annotations

- By default, generic types in Scala are non-variant
- Programmers can supply variance annotations to allow co- and contravariance
- Type checker imposes restrictions on use of variance annotations

```
class Queue[ T ] {
  def enqueue( e: T ) = { … }
  def dequeue: T = { … }
}
```

Scala

```
Queue[ AnyRef ] o;
o = new Queue[ String ]( );
```

Scala

```
Queue[ String ] o;
o = new Queue[ AnyRef ]( );
```

Scala

Covariance Annotations

- A covariance annotation (+) is useful when type variable occurs only in positive positions
  - Result type
  - Types of immutable fields

```
class Random[ +T ] {
  def next: T = { … }
}
```

Scala

```
val r: Random[ AnyRef ] =
             new Random[ String ]( )
val a = r.next
```

Scala

Covariance Annotations

- A covariance annotation (+) is useful when type variable occurs only in positive positions
  - Result type
  - Types of immutable fields

```
class Random[ +T ] {
  def next: T = { … }
}
```

Scala

```
val r: Random[ AnyRef ] =
          new Random[ String ]( )
val a = r.next
```

Scala

- Type checker prevents other occurrences

```
class Random[ +T ] {
  def next: T = { … }
  def initialze( i: T ) = { … }
}
```

Scala

Contravariance Annotations

- A contravariance annotation (-) is useful when type variable occurs only in negative positions
    - Parameter type

```
class OutputChannel[ -T ] {
  def write( x: T ) = { … }
}
```

Scala

```
val o: OutputChannel[ String ] =
     new OutputChannel[ AnyRef ]( )
o.write( “Hello” )
```

Scala

Contravariance Annotations

- A contravariance annotation (-) is useful when type variable occurs only in negative positions
  - Parameter type

```scala
class OutputChannel[ -T ] {
  def write( x: T ) = { … }
}
```

Scala

```scala
val o: OutputChannel[ String ] =
      new OutputChannel[ AnyRef ]( )
o.write( “Hello” )
```

Scala

- Type checker prevents other occurrences

```scala
class OutputChannel[ -T ] {
  def write( x: T ) = { … }
  def lastWritten: T = { … }
}
```

Scala

Working with Non-Variant Generics

- How can we write code that works with many different instantiations of a generic class?

Working with Non-Variant Generics

- How can we write code that works with many different instantiations of a generic class?

- Solution 1: Method type arguments

```
static <T> void printAll( Collection<T> c ) {
  for ( T e : c ) { System.out.println( e ); }
}
```

Java

Working with Non-Variant Generics

- How can we write code that works with many different instantiations of a generic class?
- Solution 1: Method type arguments

```
static <T> void printAll( Collection<T> c ) {
  for ( T e : c ) { System.out.println( e ); }
}
```

Java

- Solution 2: Wildcards

```
static void printAll( Collection<?> c ) {
  for ( Object e : c ) { System.out.println( e ); }
}
```

Java

Wildcards and Existential Types

- A wildcard represents an unknown type

```
static void printAll( Collection<?> c ) {
  for ( Object e : c ) { System.out.println( e ); }
}
```

Java

- Interpretation as existential type
  - “There exists a type argument T such that c has type Collection<T>”
  - Existential quantifier is instantiated automatically by the type system

```
Collection<String> c = new ArrayList<String>( );
…
printAll( c );
```

Java

Wildcards and Existential Types

- A wildcard represents an unknown type

```
static void printAll( Collection<?> c ) {
  for ( Object e : c ) { System.out.println( e ); }
}
```

Java

- Interpretation as existential type
  - “There exists a type argument T such that c has type Collection<T>”
  - Existential quantifier is instantiated automatically by the type system

```
Collection<String> c = new ArrayList<String>( );
…
printAll( c );
```

Wildcard instantiated with String

Java

Wildcard Examples

```
static Collection<?> id( Collection<?> c ) {
  return c;
}
```

Wildcard Examples

```
static Collection<?> id( Collection<?> c ) {
  return c;
}
```

Two existential types

Wildcard Examples

Correct: type checker instantiates type argument with c’s type argument

```
static Collection<?> id( Collection<?> c ) {
  return c;
}
```

Two existential types

Wildcard Examples

Correct: type checker instantiates type argument with c’s type argument

```
static Collection<?> id( Collection<?> c ) {
  return c;
}
```

Two existential types

```
Collection<String> c = new ArrayList<String>( );
Collection<String> d = id( c );
```

Wildcard Examples

Correct: type checker instantiates type argument with c’s type argument

```
static Collection<?> id( Collection<?> c ) {
  return c;
}
```

Two existential types

```
Collection<String> c = new ArrayList<String>( );
Collection<String> d = id( c );
```

Type error: existential types might have different instantiations (modular type checking)

Wildcard Examples (cont’d)

```
static void merge( Collection<?> c, Collection<?> d ) {
  for( Object e : c ) { d.add( e ); }
}
```

Wildcard Examples (cont’d)

```
static void merge( Collection<?> c, Collection<?> d ) {
  for( Object e : c ) { d.add( e ); }
}
```

Two existential types

Wildcard Examples (cont'd)

```
static void merge( Collection<?> c, Collection<?> d ) {
  for( Object e : c ) { d.add( e ); }
}
```

Two existential types

Type error: d might expect elements of different type

Wildcard Examples (cont’d)

```
class Spooler {
  Collection<?> task;

  void setTask( Collection<?> c ) {
    task = c;
  }
  void print( ) {
    for ( Object e : task ) { System.out.println( e ); }
  }
}
```

Wildcard Examples (cont'd)

```
class Spooler {
  Collection<?> task;

  void setTask( Collection<?> c ) {
    task = c;
  }
  void print( ) {
    for ( Object e : task ) { System.out.println( e ); }
  }
}
```

Correct: type checker instantiates task's type argument with c's

Wildcard Examples (cont'd)

```
class Spooler {
  Collection<?> task;

  void setTask( Collection<?> c ) {
    task = c;
  }
  void print( ) {
    for ( Object e : task ) { System.out.println( e ); }
  }
}
```

Cannot be simulated with method type parameters

Correct: type checker instantiates task's type argument with c's

Wildcard Examples (cont’d)

```
class Spooler {
  Collection<?> task;

  void setTask( Collection<?> c ) {
    task = c;
  }
  void print( ) {
    for ( Object e : task ) { System.out.println( e ); }
  }
}
```

Cannot be simulated with method type parameters

Correct: type checker instantiates task’s type argument with c’s

Works because every Java object has toString method

Constrained Wildcards

```
static void printFormatted( Collection<?> c ) {
  for ( Object e : c ) {
    String s = e.format( 80 );
    System.out.println( s );
  }
}
```

Constrained Wildcards

```
static void printFormatted( Collection<?> c ) {
  for ( Object e : c ) {
    String s = e.format( 80 );
    System.out.println( s );
  }
}
```

Type error: elements might not support method format

Constrained Wildcards: Upper Bounds

```
interface Format {
  String format( int width );
}
```

```
static void printFormatted( Collection<? extends Format> c ) {
  for ( Format e : c ) {
    String s = e.format( 80 );
    System.out.println( s );
  }
}
```

Constrained Wildcards: Upper Bounds

```
interface Format {
  String format( int width );
}
```

```
static void printFormatted( Collection<? extends Format> c ) {
  for ( Format e : c ) {
    String s = e.format( 80 );
    System.out.println( s );
  }
}
```

Typecheck under the assumption
? <: Format

Constrained Wildcards: Upper Bounds

```
interface Format {
  String format( int width );
}
```

```
static void printFormatted( Collection<? extends Format> c ) {
  for ( Format e : c ) {
    String s = e.format( 80 );
    System.out.println( s );
  }
}
```

Typecheck under the assumption ? <: Format

```
Collection<Object> c = new ArrayList<Object>( );
printFormatted( c );
```

Constrained Wildcards: Upper Bounds

```
interface Format {
  String format( int width );
}
```

```
static void printFormatted( Collection<? extends Format> c ) {
  for ( Format e : c ) {
    String s = e.format( 80 );
    System.out.println( s );
  }
}
```

Typecheck under the assumption ? <: Format

```
Collection<Object> c = new ArrayList<Object>( );
printFormatted( c );
```

Compile-time error: Object is not a subtype of the upper bound Format

More Bounded Wildcards

```
class Cell<T> {
  T value;

  void copyFromT( Cell<T> other ) {
    value = other.value;
  }







}
```

More Bounded Wildcards

```
class Cell<T> {
  T value;

  void copyFromT( Cell<T> other ) {
    value = other.value;
  }









}
```

Overly restrictive

More Bounded Wildcards

```
class Cell<T> {
  T value;

  void copyFromT( Cell<T> other ) {
    value = other.value;
  }

  void copyFrom( Cell<?> other ) {
    value = other.value;
  }




}
```

More Bounded Wildcards

```
class Cell<T> {
  T value;

  void copyFromT( Cell<T> other ) {
    value = other.value;
  }

  void copyFrom( Cell<?> other ) {
    value = other.value;
  }




}
```

Not type correct

More Bounded Wildcards

```
class Cell<T> {
  T value;

  void copyFromT( Cell<T> other ) {
    value = other.value;
  }

  void copyFrom( Cell<? extends T> other ) {
    value = other.value;
  }




}
```

More Bounded Wildcards

```
class Cell<T> {
  T value;

  void copyFromT( Cell<T> other ) {
    value = other.value;
  }

  void copyFrom( Cell<? extends T> other ) {
    value = other.value;
  }







}
```

Typecheck under the assumption
? <: T

More Bounded Wildcards

```
class Cell<T> {
  T value;

  void copyFromT( Cell<T> other ) {
    value = other.value;
  }

  void copyFrom( Cell<? extends T> other ) {
    value = other.value;
  }

  void copyTo( Cell<?> other ) {
    other.value = value;
  }
}
```

Typecheck under the assumption ? <: T

More Bounded Wildcards

```
class Cell<T> {
  T value;

  void copyFromT( Cell<T> other ) {
    value = other.value;
  }

  void copyFrom( Cell<? extends T> other ) {
    value = other.value;
  }

  void copyTo( Cell<?> other ) {
    other.value = value;
  }
}
```

Typecheck under the assumption ? <: T

Not type correct

More Bounded Wildcards

```
class Cell<T> {
  T value;

  void copyFromT( Cell<T> other ) {
    value = other.value;
  }

  void copyFrom( Cell<? extends T> other ) {
    value = other.value;
  }

  void copyTo( Cell<? super T> other ) {
    other.value = value;
  }
}
```

Typecheck under the assumption ? <: T

Wildcard can also have lower bounds

More Bounded Wildcards

```
class Cell<T> {
  T value;

  void copyFromT( Cell<T> other ) {
    value = other.value;
  }

  void copyFrom( Cell<? extends T> other ) {
    value = other.value;
  }

  void copyTo( Cell<? super T> other ) {
    other.value = value;
  }
}
```

Typecheck under the assumption
? <: T

Typecheck under the assumption
T <: ?

Wildcard can also have
lower bounds

Wildcards vs. Additional Type Parameters

- Often wildcards can be replaced by additional class or method type arguments
  - C# does not have wildcards

```
static <T> void printAll( Collection<T> c ) {
  for ( T e : c ) { System.out.println( e ); }
}
```

Java

```
static void printAll( Collection<?> c ) {
  for ( Object e : c ) { System.out.println( e ); }
}
```

Java

Wildcards vs. Method Type Parameters

```
void copyFrom( Cell<? extends T> other ) {
  value = other.value;
}
```

```
<S extends T> void copyFrom( Cell<S> other ) {
  value = other.value;
}
```

Wildcards vs. Method Type Parameters

```
void copyFrom( Cell<? extends T> other ) {
  value = other.value;
}
```

```
<S extends T> void copyFrom( Cell<S> other ) {
  value = other.value;
}
```

```
Cell<String> s;
Cell<Object> o;
s = new Cell<String>();
o = new Cell<Object>();
o.copyFrom( s );
```

Wildcards vs. Method Type Parameters

```
void copyFrom( Cell<? extends T> other ) {
  value = other.value;
}
```

```
<S extends T> void copyFrom( Cell<S> other ) {
  value = other.value;
}
```

```
Cell<String> s;
Cell<Object> o;
s = new Cell<String>();
o = new Cell<Object>();
o.copyFrom( s );
```

Identical client code: instantiations of wildcard and method type argument are inferred

Wildcards vs. Method Type Parameters

```
void copyFrom( Cell<? extends T> other ) {
  value = other.value;
}
```

```
<S extends T> void copyFrom( Cell<S> other ) {
  value = other.value;
}
```

```
void copyTo( Cell<? super T> other ) {
  other.value = value;
}
```

```
<S super T> void copyTo( Cell<S> other ) {
  other.value = value;
}
```

```
Cell<String> s;
Cell<Object> o;
s = new Cell<String>();
o = new Cell<Object>();
o.copyFrom( s );
```

Identical client code: instantiations of wildcard and method type argument are inferred

Wildcards vs. Method Type Parameters

```
void copyFrom( Cell<? extends T> other ) {
  value = other.value;
}
```

```
<S extends T> void copyFrom( Cell<S> other ) {
  value = other.value;
}
```

```
Cell<String> s;
Cell<Object> o;
s = new Cell<String>();
o = new Cell<Object>();
o.copyFrom( s );
```

Identical client code: instantiations of wildcard and method type argument are inferred

```
void copyTo( Cell<? super T> other ) {
  other.value = value;
}
```

```
<S super T> void copyTo( Cell<S> other ) {
  other.value = value;
}
```

Java does not support lower bounds for type parameters

Wildcards vs. Class Type Parameters

```
class Wrapper {
  Cell<?> data;
}
```

```
class Wrapper<T> {
  Cell<T> data;
}
```

Wildcards vs. Class Type Parameters

```
class Wrapper {
  Cell<?> data;
}
```

```
Wrapper w = new Wrapper( );
w.data = new Cell<String>( );
w.data = new Cell<Object>( );
```

```
class Wrapper<T> {
  Cell<T> data;
}
```

Wildcards vs. Class Type Parameters

```
class Wrapper {
  Cell<?> data;
}
```

```
Wrapper w = new Wrapper( );
w.data = new Cell<String>( );
w.data = new Cell<Object>( );
```

```
class Wrapper<T> {
  Cell<T> data;
}
```

```
Wrapper w = new Wrapper<Object>( );
w.data = new Cell<String>( );
w.data = new Cell<Object>( );
```

Wildcards vs. Class Type Parameters

```
class Wrapper {
  Cell<?> data;
}
```

```
Wrapper w = new Wrapper( );
w.data = new Cell<String>( );
w.data = new Cell<Object>( );
```

Instantiation can change over time

```
class Wrapper<T> {
  Cell<T> data;
}
```

```
Wrapper w = new Wrapper<Object>( );
w.data = new Cell<String>( );
w.data = new Cell<Object>( );
```

Wildcards vs. Class Types Parameters

```
class Wrapper {
  Cell<?> data;
}
```

```
Wrapper w = new Wrapper( );
w.data = new Cell<String>( );
w.data = new Cell<Object>( );
```

Instantiation can change over time

```
class Wrapper<T> {
  Cell<T> data;
}
```

```
Wrapper w = new Wrapper<Object>( );
w.data = new Cell<String>( );
w.data = new Cell<Object>( );
```

With type argument, instantiation is fixed when object is created

Wildcards vs. Type Parameters (cont’d)

```
interface Comparator<T> {
  int compare( T fst, T snd );
}
```

```
class TreeSet<E> {
  TreeSet( Comparator<E> c ) { … }
  … }
```

Wildcards vs. Type Parameters (cont'd)

```
interface Comparator<T> {
  int compare( T fst, T snd );
}
```

```
class TreeSet<E> {
  TreeSet( Comparator<E> c ) { … }
  … }
```

TreeSet needs to compare set elements

Wildcards vs. Type Parameters (cont’d)

```
interface Comparator<T> {
  int compare( T fst, T snd );
}
```

```
class TreeSet<E> {
  TreeSet( Comparator<E> c ) { … }
  … }
```

TreeSet needs to compare set elements

```
class Person { … }
```

```
class Student extends Person { … }
```

```
class PersonComp implements Comparator<Person> {
  int compare( Person fst, Person snd ) { … }
}
```

Wildcards vs. Type Parameters (cont’d)

```
interface Comparator<T> {
  int compare( T fst, T snd );
}
```

```
class TreeSet<E> {
  TreeSet( Comparator<E> c ) { … }
  … }
```

TreeSet needs to compare set elements

```
class Person { … }
```

```
class Student extends Person { … }
```

```
class PersonComp implements Comparator<Person> {
  int compare( Person fst, Person snd ) { … }
}
```

Universal comparator for all persons

Wildcards vs. Type Parameters (cont'd)

```
interface Comparator<T> {
  int compare( T fst, T snd );
}
```

```
class TreeSet<E> {
  TreeSet( Comparator<E> c ) { … }
  … }
```

TreeSet needs to compare set elements

```
class Person { … }
```

```
class Student extends Person { … }
```

```
class PersonComp implements Comparator<Person> {
  int compare( Person fst, Person snd ) { … }
}
```

Universal comparator for all persons

```
TreeSet<Student> s = new TreeSet<Student>( new PersonComp() );
```

Wildcards vs. Type Parameters (cont’d)

```
interface Comparator<T> {
  int compare( T fst, T snd );
}
```

```
class TreeSet<E> {
  TreeSet( Comparator<E> c ) { … }
  … }
```

TreeSet needs to compare set elements

```
class Person { … }
```

```
class Student extends Person { … }
```

```
class PersonComp implements Comparator<Person> {
  int compare( Person fst, Person snd ) { … }
}
```

Universal comparator for all persons

```
TreeSet<Student> s = new TreeSet<Student>( new PersonComp() );
```

Wildcards vs. Type Parameters (cont'd)

```
interface Comparator<T> {
  int compare( T fst, T snd );
}
```

```
class TreeSet<E> {
  TreeSet( Comparator<E> c ) { … }
  … }
```

TreeSet needs to compare set elements

```
class Person { … }
```

```
class Student extends Person { … }
```

```
class PersonComp implements Comparator<Person> {
  int compare( Person fst, Person snd ) { … }
}
```

Universal comparator for all persons

```
TreeSet<Student> s = new TreeSet<Student>( new PersonComp() );
```

Compile-time error: PersonComp is not a subtype of Comparator<Student>

Wildcards vs. Type Parameters (cont’d)

```
interface Comparator<T> {
  int compare( T fst, T snd );
}
```

```
class TreeSet<E> {
  TreeSet( Comparator<? super E> c ) { … }
  … }
```

```
class Person { … }
```

```
class Student extends Person { … }
```

```
class PersonComp implements Comparator<Person> {
  int compare( Person fst, Person snd ) { … }
}
```

Wildcards vs. Type Parameters (cont'd)

```
interface Comparator<T> {
  int compare( T fst, T snd );
}
```

```
class TreeSet<E> {
  TreeSet( Comparator<? super E> c ) { … }
  … }
```

TreeSet needs to compare at least set elements

```
class Person { … }
```

```
class Student extends Person { … }
```

```
class PersonComp implements Comparator<Person> {
  int compare( Person fst, Person snd ) { … }
}
```

Wildcards vs. Type Parameters (cont'd)

```
interface Comparator<T> {
  int compare( T fst, T snd );
}
```

```
class TreeSet<E> {
  TreeSet( Comparator<? super E> c ) { … }
  … }
```

TreeSet needs to compare at least set elements

```
class Person { … }
```

```
class Student extends Person { … }
```

```
class PersonComp implements Comparator<Person> {
  int compare( Person fst, Person snd ) { … }
}
```

```
TreeSet<Student> s = new TreeSet<Student>( new PersonComp() );
```

Wildcards vs. Type Parameters (cont’d)

```
interface Comparator<T> {
  int compare( T fst, T snd );
}
```

```
class TreeSet<E> {
  TreeSet( Comparator<? super E> c ) { … }
  … }
```

TreeSet needs to compare at least set elements

```
class Person { … }
```

```
class Student extends Person { … }
```

```
class PersonComp implements Comparator<Person> {
  int compare( Person fst, Person snd ) { … }
}
```

```
TreeSet<Student> s = new TreeSet<Student>( new PersonComp() );
```

Wildcards vs. Type Parameters (cont’d)

```
interface Comparator<T> {
  int compare( T fst, T snd );
}
```

```
class TreeSet<E> {
  TreeSet( Comparator<? super E> c ) { … }
  … }
```

TreeSet needs to compare at least set elements

```
class Person { … }
```

```
class Student extends Person { … }
```

```
class PersonComp implements Comparator<Person> {
  int compare( Person fst, Person snd ) { … }
}
```

```
TreeSet<Student> s = new TreeSet<Student>( new PersonComp() );
```

Wildcard instantiated with Person, which is a supertype of Student

Subtyping and Generics: Wildcards

- The bounds for a wildcard determine the set of possible instantiations

```
Cell<? extends Person> c;
Cell<? super PhDStudent> d;
```

- S is a subtype of T if for each type argument, the set of possible instantiations for S is a subset of the set of possible instantiations for T

Subtyping and Generics: Wildcards

- The bounds for a wildcard determine the set of possible instantiations

```
Cell<? extends Person> c;
Cell<? super PhDStudent> d;
```

- S is a subtype of T if for each type argument, the set of possible instantiations for S is a subset of the set of possible instantiations for T

```
c = new Cell<Student>( );
```

Subtyping and Generics: Wildcards

- The bounds for a wildcard determine the set of possible instantiations

```
Cell<? extends Person> c;
Cell<? super PhDStudent> d;
```

- S is a subtype of T if for each type argument, the set of possible instantiations for S is a subset of the set of possible instantiations for T

```
c = new Cell<Student>( );
```

Instantiation is fixed (singleton set)

Subtyping and Generics: Wildcards

- The bounds for a wildcard determine the set of possible instantiations

```
Cell<? extends Person> c;
Cell<? super PhDStudent> d;
```

- S is a subtype of T if for each type argument, the set of possible instantiations for S is a subset of the set of possible instantiations for T

```
c = new Cell<Student>( );
```

Instantiation is fixed (singleton set)

```
Cell<? extends Student> e = …;
c = e;
```

Type Erasure

- Java introduced generics in version 1.4
- For backwards compatibility, Sun did not want to change the virtual machine

- Generic type information is erased by compiler
  - C<T> is translated to C
  - T is translated to its upper bound
  - Casts are added where necessary
- Only one classfile and only one class object to represent all instantiations of a generic class

Type Erasure: Example

```
class Cell<T extends Object> {
  T value;

  void set( T v ) {
    value = v;
  }

  T get( ) {
    return value;
  }
}
```

```
class Cell {
  Object value;

  void set( Object v ) {
    value = v;
  }

  Object get( ) {
    return value;
  }
}
```

```
Cell<String> c;
c = new Cell<String>( ) ;
c.set( “Hello” );
String s = c.get( );
```

```
Cell c;
c = new Cell( );
c.set( “Hello” );
String s = ( String ) c.get( );
```

Erasure: Missing Run-Time Information

```
void foo( Cell<?> c ) {
  if( c instanceof Cell<String> )
    …
}
```

Java

Erasure: Missing Run-Time Information

```
void foo( Cell<?> c ) {
  if( c instanceof Cell<String> )
    …
}
```

Java

Compile-time error: generic types not allowed with **instanceof**

Erasure: Missing Run-Time Information

```
void foo( Cell<?> c ) {
  if( c instanceof Cell<String> )
    …
}
```

Java

Compile-time error: generic types not allowed with **instanceof**

```
Class c = Cell<String>.class;
```

Java

Erasure: Missing Run-Time Information

```
void foo( Cell<?> c ) {
  if( c instanceof Cell<String> )
    …
}
```

Java

Compile-time error: generic types not allowed with **instanceof**

```
Class c = Cell<String>.class;
```

Jav

Compile-time error: class object of generic types not available

Erasure: Missing Run-Time Information

```
void foo( Cell<?> c ) {
  if( c instanceof Cell<String> )
    …
}
```

Java

Compile-time error: generic types not allowed with **instanceof**

```
Class c = Cell<String>.class;
```

Java

Compile-time error: class object of generic types not available

```
Cell<String>[ ] a;
a = new Cell<String>[10];
```

Java

Erasure: Missing Run-Time Information

```java
void foo( Cell<?> c ) {
  if( c instanceof Cell<String> )
    …
}
```

Java

Compile-time error: generic types not allowed with **instanceof**

```java
Class c = Cell<String>.class;
```

Compile-time error: class object of generic types not available

```java
Cell<String>[ ] a;
a = new Cell<String>[10];
```

Compile-time error: arrays of generic types not allowed

Run-Time Information for Generics in C#

```
void Foo( object c ) {
  if( c is Cell<string> )
    …
}
```

C#

Run-Time Information for Generics in C#

```
void Foo( object c ) {
  if( c is Cell<string> )
    …
}
```

C#

C# does not have wildcards

Run-Time Information for Generics in C#

```
void Foo( object c ) {
  if( c is Cell<string> )
    …
}
```

C#

C# does not have wildcards

C# can perform dynamic type test

Run-Time Information for Generics in C#

```
void Foo( object c ) {
  if( c is Cell<string> )
    …
}
```

C#

C# does not have wildcards

C# can perform dynamic type test

```
System.Type type = typeof( Cell<string> );
```

C#

Run-Time Information for Generics in C#

```
void Foo( object c ) {
  if( c is Cell<string> )
    …
}
```

C#

C# does not have wildcards

C# can perform dynamic type test

```
System.Type type = typeof( Cell<string> );
```

C#

C# has run-time representation

Run-Time Information for Generics in C#

```
void Foo( object c ) {
  if( c is Cell<string> )
    …
}
```

C#

C# does not have wildcards

C# can perform dynamic type test

```
System.Type type = typeof( Cell<string> );
```

C#

C# has run-time representation

```
Cell<String>[ ] a;
a = new Cell<string>[10];
```

C#

Run-Time Information for Generics in C#

```
void Foo( object c ) {
  if( c is Cell<string> )
    …
}
```

C#

C# does not have wildcards

C# can perform dynamic type test

```
System.Type type = typeof( Cell<string> );
```

C# has run-time representation

```
Cell<String>[ ] a;
a = new Cell<string>[10];
```

C# can perform run-time check for array update

Erasure: Missing Run-Time Checks

```
String demo( Cell<?> c ) {
  Cell<String> cs = ( Cell<String> ) c;
  …
  return cs.val;
}
```

Java

Erasure: Missing Run-Time Checks

```java
String demo( Cell<?> c ) {
  Cell<String> cs = ( Cell<String> ) c;
  …
  return cs.val;
}
```

Java

```java
void main( ) {
  Cell<Object> co = new Cell<Object>( );
  co.val = new Integer( 5 );
  demo( co );
}
```

Java

Erasure: Missing Run-Time Checks

No run-time check for this cast!

```
String demo( Cell<?> c ) {
  Cell<String> cs = ( Cell<String> ) c;
  …
  return cs.val;
}
```

Java

```
void main( ) {
  Cell<Object> co = new Cell<Object>( );
  co.val = new Integer( 5 );
  demo( co );
}
```

Java

Erasure: Missing Run-Time Checks

```
String demo( Cell<?> c ) {
  Cell<String> cs = ( Cell<String> ) c;
  …
  return cs.val;
}
```

No run-time check for this cast!

Type invariant broken: object in cs is not of type Cell<String>!

```
void main( ) {
  Cell<Object> co = new Cell<Object>( );
  co.val = new Integer( 5 );
  demo( co );
}
```

Java

Erasure: Missing Run-Time Checks

```
String demo( Cell<?> c ) {
  Cell<String> cs = ( Cell<String> ) c;
  …
  return cs.val;
}
```

No run-time check for this cast!

Type invariant broken: object in cs is not of type Cell<String>!

Run-time error: cs.val is not a string

```
void main( ) {
  Cell<Object> co = new Cell<Object>( );
  co.val = new Integer( 5 );
  demo( co );
}
```

Java

Other Consequences of Erasure

- Static fields are shared by all instantiations of a generic class

```
class Count<T> {
  static int c = 0;
}
```

Other Consequences of Erasure

- Static fields are shared by all instantiations of a generic class

```
class Count<T> {
  static int c = 0;
}
```

```
Count<string>.c = 1;
Count<object>.c = 2;
```

C#

Other Consequences of Erasure

- Static fields are shared by all instantiations of a generic class

```
class Count<T> {
  static int c = 0;
}
```

```
Count<string>.c = 1;
Count<object>.c = 2;
```

C#

```
Count.c = 1;
Count.c = 2;
```

Java

Other Consequences of Erasure

- Static fields are shared by all instantiations of a generic class

```
class Count<T> {
  static int c = 0;
}
```

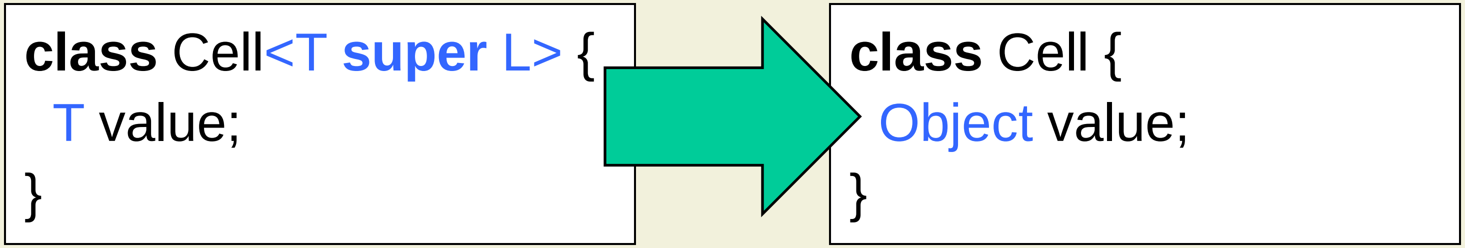
```
Count<string>.c = 1;
Count<object>.c = 2;
```

C#

```
Count.c = 1;
Count.c = 2;
```

Java

- Lower bounds for type parameters



```
class Cell<T super L> {
  T value;
}
```

```
class Cell {
  Object value;
}
```

Other Consequences of Erasure

- Static fields are shared by all instantiations of a generic class

```
class Count<T> {
  static int c = 0;
}
```

```
Count<string>.c = 1;
Count<object>.c = 2;
```

C#

```
Count.c = 1;
Count.c = 2;
```

Java

- Lower bounds for type parameters

```
class Cell<T super L> {
  T value;
}
```

```
class Cell {
  Object value;
}
```

  - Enforcing lower bound would require many run-time checks (e.g., for each field update)

C++ Templates

- Templates allow classes and methods to be parameterized
- Clients provide instantiations for template parameters

```
template<class T> class Queue {
  T elem;
  Queue<T>* next;
public:
  void enqueue( T e ) { … };
  T dequeue( void ) { … };
};
```

C++

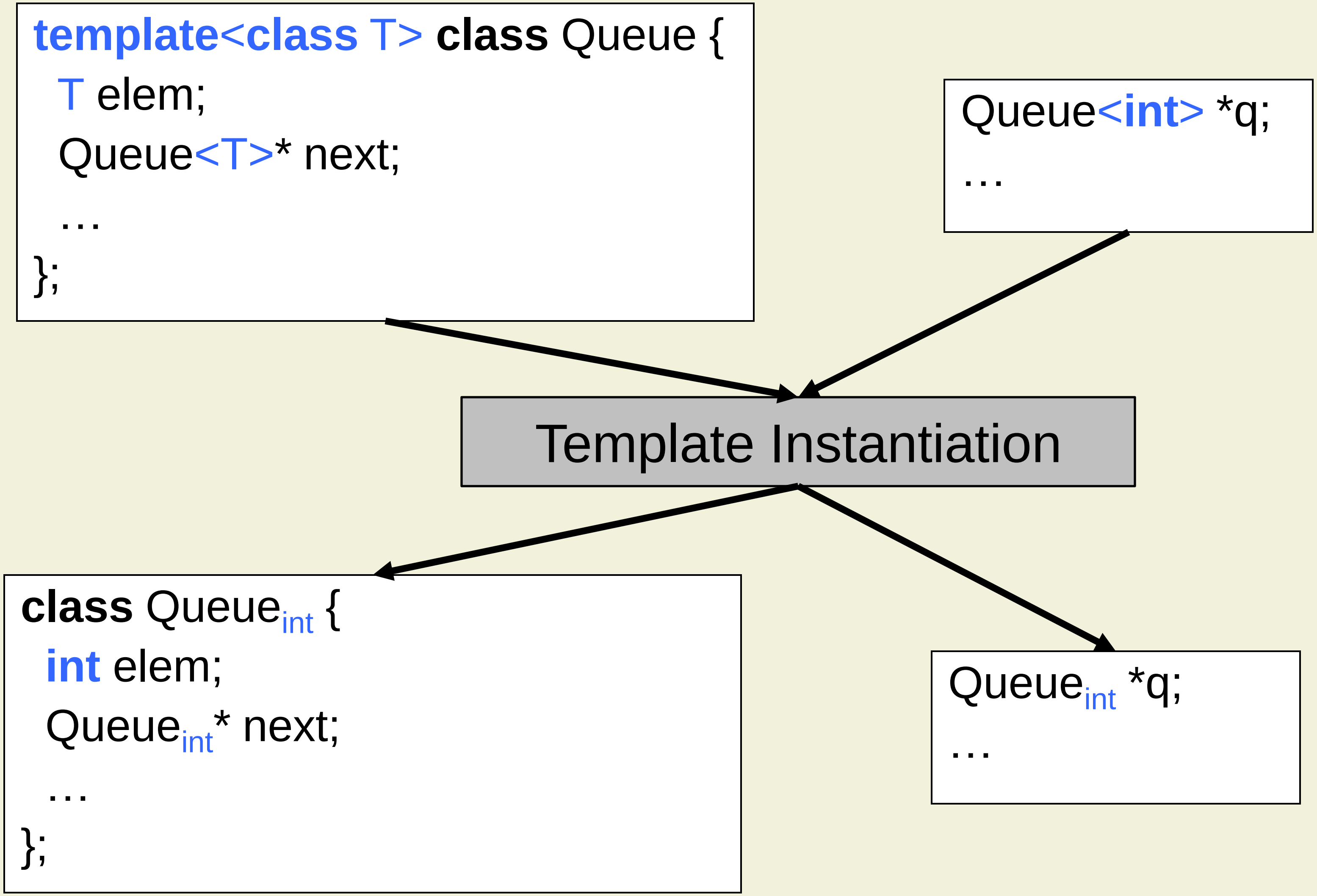
```
Queue<int> *q;
q = new Queue<int>( );
int s = 5;
q->enqueue( s );
int t = q->dequeue( );
```

C++

```
template<class T> void fill( T a[ ], T v )
{ … };
```

C++

Template Instantiation



```
template<class T> class Queue {
  T elem;
  Queue<T>* next;
  …
};
```

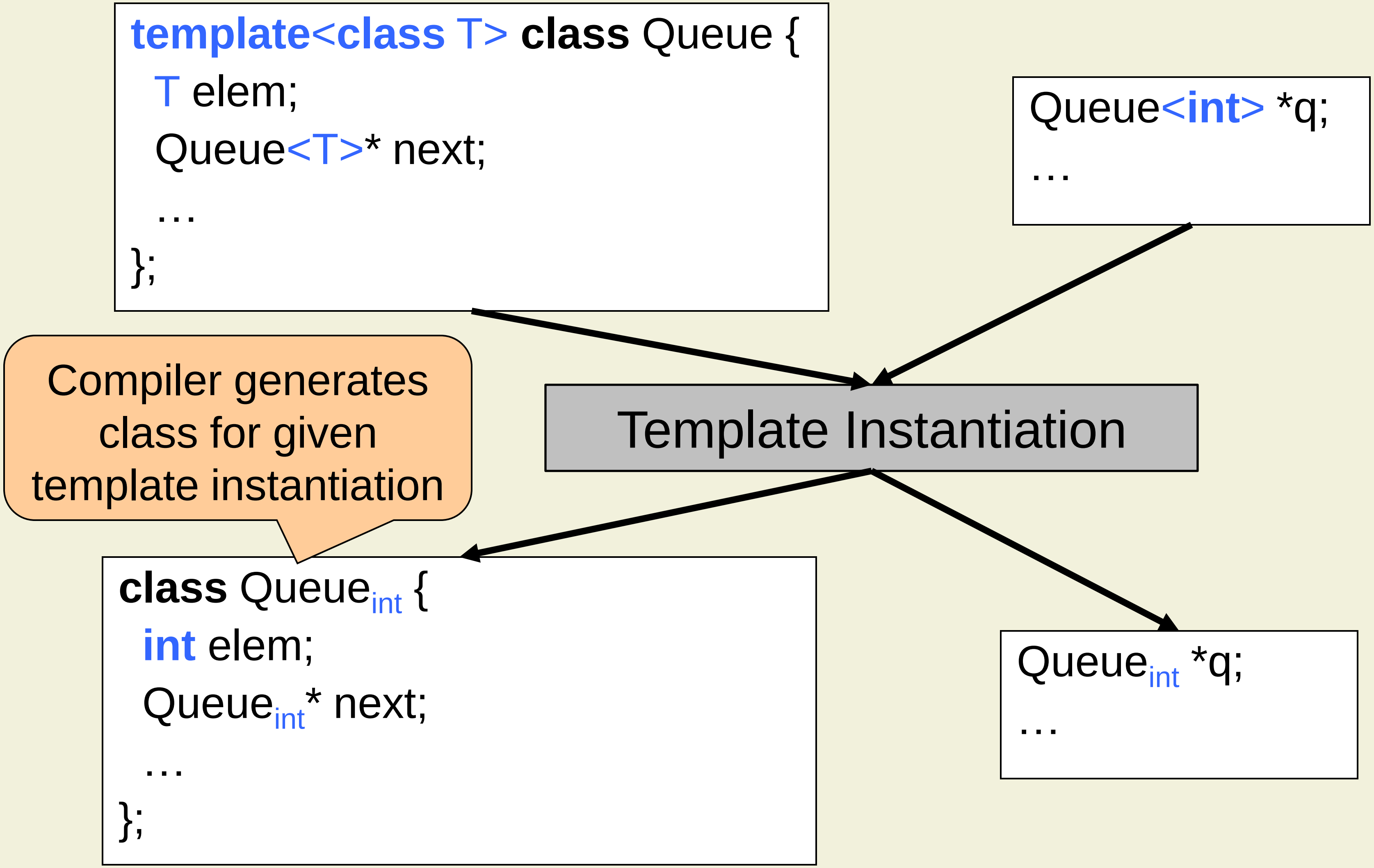
```
Queue<int> *q;
…
```

Template Instantiation

class Queue$_{\text{int}}$ {
  int elem;
  Queue$_{\text{int}}$* next;
  …
};

Queue$_{\text{int}}$ *q;
…

Template Instantiation



```
template<class T> class Queue {
  T elem;
  Queue<T>* next;
  …
};
```

```
Queue<int> *q;
…
```

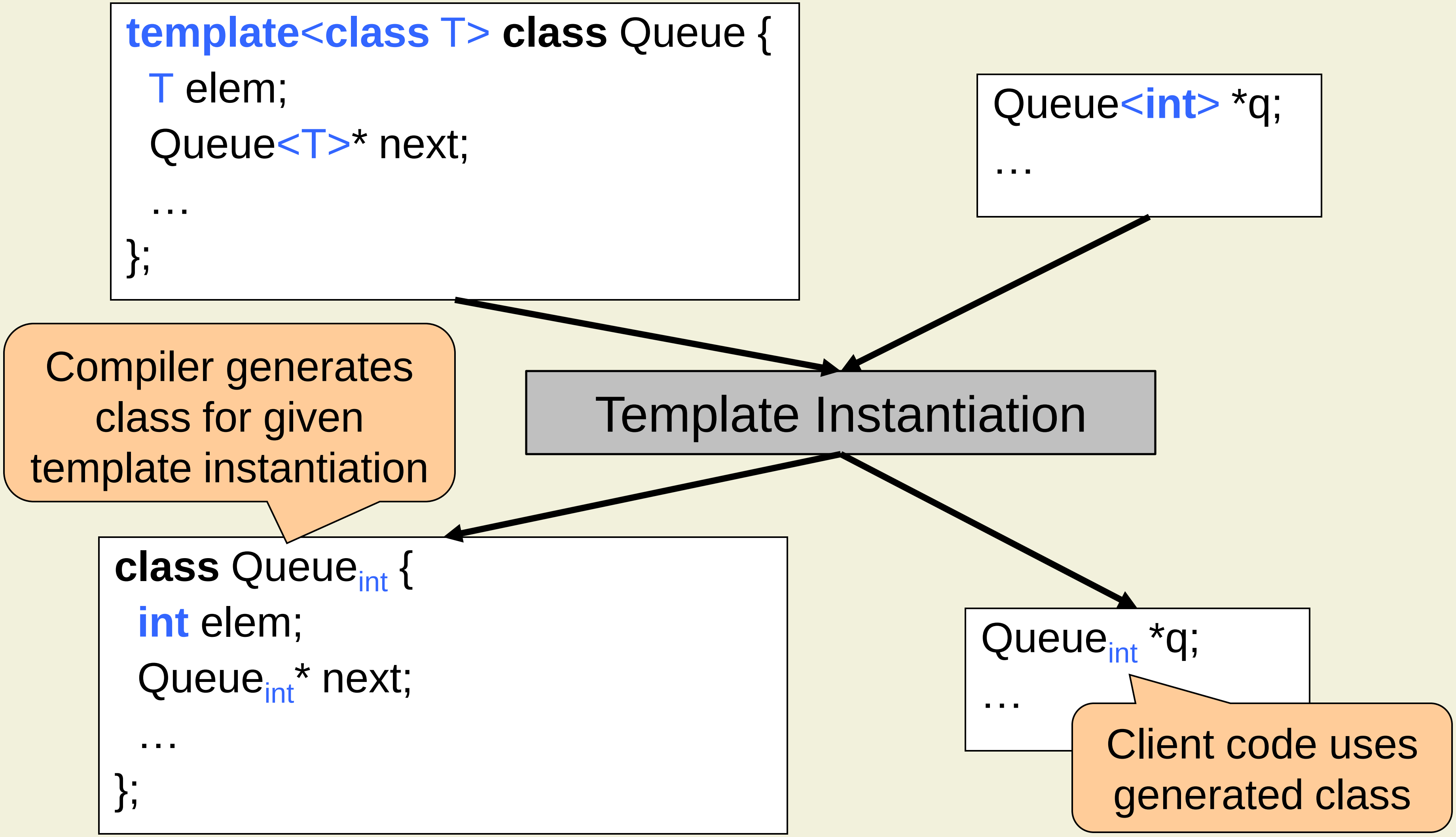
Template Instantiation

Compiler generates class for given template instantiation

class $\text{Queue}_{\text{int}}$ {
  int elem;
  $\text{Queue}_{\text{int}}$* next;
  …
};

$\text{Queue}_{\text{int}}$ *q;
…

Template Instantiation



```
template<class T> class Queue {
  T elem;
  Queue<T>* next;
  …
};
```

```
Queue<int> *q;
…
```

Template Instantiation

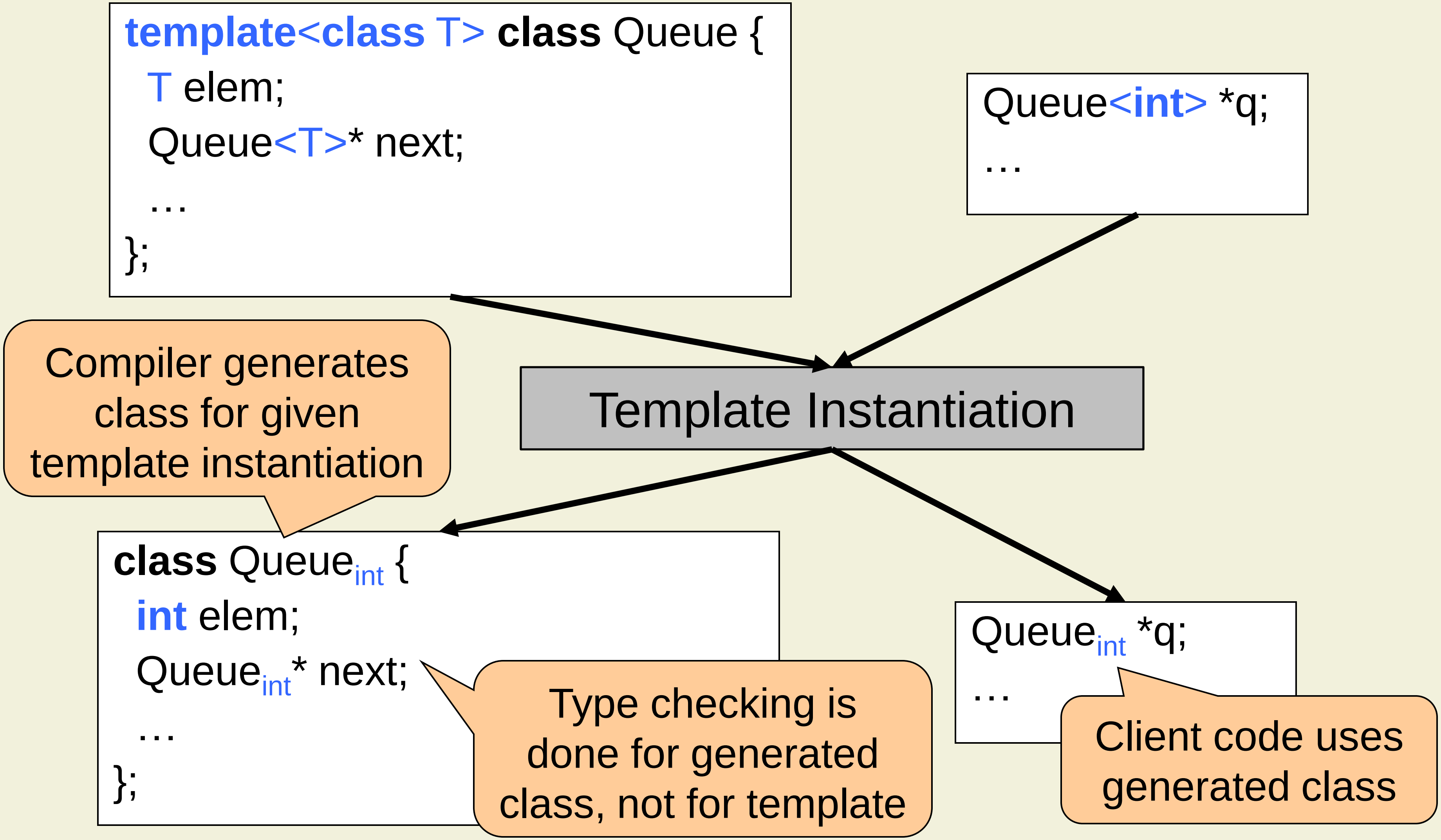
Compiler generates class for given template instantiation

class Queue$_{int}$ {
  int elem;
  Queue$_{int}$* next;
  …
};

Queue$_{int}$ *q;
…

Client code uses generated class

Template Instantiation



```
template<class T> class Queue {
  T elem;
  Queue<T>* next;
  …
};
```

```
Queue<int> *q;
…
```

Template Instantiation

Compiler generates class for given template instantiation

class Queue$_{int}$ {
  int elem;
  Queue$_{int}$* next;
  …
};

Type checking is done for generated class, not for template

Queue$_{int}$ *q;
…

Client code uses generated class

Templates and Type Checking

```
template<class T> class Queue {
  T elem;  Queue<T>* next;
public:
  void enqueue( T e ) {
    if( next == NULL ) {
      elem = e;  next = new Queue<T>( );
    } else {
      if( e.compareTo( elem ) <= 0 )
        next->enqueue( elem ); elem = e;
      else next->enqueue( e );
    }
  };
  T dequeue( void ) { return “Hello”; };
};
```

Templates and Type Checking

```
template<class T> class Queue {
  T elem;  Queue<T>* next;
public:
  void enqueue( T e ) {
    if( next == NULL ) {
      elem = e;  next = new Queue<T>( );
    } else {
      if( e.compareTo( elem ) <= 0 )
      { next->enqueue( elem ); elem = e; }
      else next->enqueue( e );
    }
  };
  T dequeue( void ) { return “Hello”; };
};
```

Templates and Type Checking

```
template<class T> class Queue {
  T elem;  Queue<T>* next;
public:
  void enqueue( T e ) {
    if( next == NULL ) {
      elem = e;  next = new Queue<T>( );
    } else {
      if( e.compareTo( elem ) <= 0 )
      { next->enqueue( elem ); elem = e; }
      else next->enqueue( e );
    }
  };
  T dequeue( void ) { return "Hello"; };
};
```

Compiler does not type check template code

Templates and Type Checking

```
template<class T> class Queue {
  T elem;  Queue<T>* next;
public:
  void enqueue( T e ) {
    if( next == NULL ) {
      elem = e;  next = new Queue<T>( );
    } else {
      if( e.compareTo( elem ) <= 0 )
      { next->enqueue( elem ); elem = e; }
      else next->enqueue( e );
    }
  };
  T dequeue( void ) { return "Hello"; };
};
```

Compiler does not check availability of methods

Compiler does not type check template code

Templates and Type Checking

```
template<class T> class Queue {
  T elem;  Queue<T>* next;
public:
  void enqueue( T e ) {
    if( next == NULL ) {
      elem = e;  next = new Queue<T>( );
    } else {
      if( e.compareTo( elem ) <= 0 )
      { next->enqueue( elem ); elem = e; }
      else next->enqueue( e );
    }
  };
  T dequeue( void ) { return “Hello”; };
};
```

```
Queue<int> *q;
q = new Queue<int>( );
```

Compiler does not check availability of methods

Compiler does not type check template code

Templates and Type Checking

```
template<class T> class Queue {
  T elem;  Queue<T>* next;
public:
  void enqueue( T e ) {
    if( next == NULL ) {
      elem = e;  next = new Queue<T>( );
    } else {
      if( e.compareTo( elem ) <= 0 )
      { next->enqueue( elem ); elem = e; }
      else next->enqueue( e );
    }
  };
  T dequeue( void ) { return “Hello”; };
};
```

```
Queue<int> *q;
q = new Queue<int>( );
```

Compiler does not check availability of methods

Compiler does not type check template code

Templates and Type Checking

```
template<class T> class Queue {
  T elem;  Queue<T>* next;
public:
  void enqueue( T e ) {
    if( next == NULL ) {
      elem = e;  next = new Queue<T>( );
    } else {
      if( e.compareTo( elem ) <= 0 )
      { next->enqueue( elem ); elem = e; }
      else next->enqueue( e );
    }
  };
  T dequeue( void ) { return “Hello”; };
};
```

Compiler does not check availability of methods

Compiler does not type check template code

```
Queue<int> *q;
q = new Queue<int>( );
```

Compiles even though template is instantiated

Templates and Type Checking

```
template<class T> class Queue {
  T elem;  Queue<T>* next;
public:
  void enqueue( T e ) {
    if( next == NULL ) {
      elem = e;  next = new Queue<T>( );
    } else {
      if( e.compareTo( elem ) <= 0 )
      { next->enqueue( elem ); elem = e; }
      else next->enqueue( e );
    }
  };
  T dequeue( void ) { return "Hello"; };
};
```

Compiler does not check availability of methods

Compiler does not type check template code

```
Queue<int> *q;
q = new Queue<int>( );
```

Compiles even though template is instantiated

```
Queue<int> *q;
q = new Queue<int>( );
int s = 5;
q->enqueue( s );
int t = q->dequeue( );
```

Templates and Type Checking

```
template<class T> class Queue {
  T elem;  Queue<T>* next;
public:
  void enqueue( T e ) {
    if( next == NULL ) {
      elem = e;  next = new Queue<T>( );
    } else {
      if( e.compareTo( elem ) <= 0 )
      { next->enqueue( elem ); elem = e; }
      else next->enqueue( e );
    }
  };
  T dequeue( void ) { return “Hello”; };
};
```

Compiler does not check availability of methods

Compiler does not type check template code

```
Queue<int> *q;
q = new Queue<int>( );
```

Compiles even though template is instantiated

```
Queue<int> *q;
q = new Queue<int>( );
int s = 5;
q->enqueue( s );
int t = q->dequeue( );
```

Templates and Type Checking

```
template<class T> class Queue {
  T elem;  Queue<T>* next;
public:
  void enqueue( T e ) {
    if( next == NULL ) {
      elem = e;  next = new Queue<T>( );
    } else {
      if( e.compareTo( elem ) <= 0 )
      { next->enqueue( elem ); elem = e; }
      else next->enqueue( e );
    }
  };
  T dequeue( void ) { return "Hello"; };
};
```

Compiler does not check availability of methods

Compiler does not type check template code

```
Queue<int> *q;
q = new Queue<int>( );
```

Compiles even though template is instantiated

Compile-time errors: template methods not type correct

```
Queue<int> *q;
q = new Queue<int>( );
int s = 5;
q->enqueue( s );
int t = q->dequeue( );
```

Templates and Type Checking (cont’d)

- Template code is not type checked
  - Type errors are not detected before instantiation
- No need for upper bounds on type parameters
  - Availability of methods is not checked anyway
  - Template has to document (informally) what it expects from its type arguments

- Different instantiations of templates are unrelated
  - Use template methods to write polymorphic methods
- Templates do not require run-time support
  - Run-time types correspond to generated classes

Template Meta-Programming

```
template <int n> class Fact {
public:
  static const int val = Fact<n-1>::val * n;
};
```

Template Meta-Programming

```
template <int n> class Fact {
public:
  static const int val = Fact<n-1>::val * n;
};
```

Template parameters need not be types

Template Meta-Programming

```
template <int n> class Fact {
public:
  static const int val = Fact<n-1>::val * n;
};
```

Template parameters need not be types

```
class Fact<0> {
public:
  static const int val = 1;
};
```

Template Meta-Programming

```
template <int n> class Fact {
public:
  static const int val = Fact<n-1>::val * n;
};
```

Template parameters need not be types

```
class Fact<0> {
public:
  static const int val = 1;
};
```

Templates can be specialized

Template Meta-Programming

```
template <int n> class Fact {
public:
  static const int val = Fact<n-1>::val * n;
};
```

Template parameters need not be types

```
class Fact<0> {
public:
  static const int val = 1;
};
```

Templates can be specialized

```
int main( ) {
  printf( "fact 3 = %d\n", Fact<3>::val );
  printf( "fact 4 = %d\n", Fact<4>::val );
  printf( "fact 5 = %d\n", Fact<5>::val );
  return 0;
}
```

Template Meta-Programming

```
template <int n> class Fact {
public:
  static const int val = Fact<n-1>::val * n;
};
```

Template parameters need not be types

```
class Fact<0> {
public:
  static const int val = 1;
};
```

Templates can be specialized

Compiler generates these instantiations

```
int main( ) {
  printf( "fact 3 = %d\n", Fact<3>::val );
  printf( "fact 4 = %d\n", Fact<4>::val );
  printf( "fact 5 = %d\n", Fact<5>::val );
  return 0;
}
```

Template Meta-Programming

```
template <int n> class Fact {
public:
  static const int val = Fact<n-1>::val * n;
};
```

Template parameters need not be types

```
class Fact<0> {
public:
  static const int val = 1;
};
```

Templates can be specialized

Compiler generates these instantiations

```
int main( ) {
  printf( "fact 3 = %d\n", Fact<3>::val );
  printf( "fact 4 = %d\n", Fact<4>::val );
  printf( "fact 5 = %d\n", Fact<5>::val );
  return 0;
}
```

Through constant propagation, values are computed by compiler

Generic Types vs. Templates: Summary

Generic Types

Templates

Generic Types vs. Templates: Summary

Generic Types

- Modular type checking of generic class
  - Overhead (e.g., upper bounds)

Templates

- Type checking per instantiation
  - Flexibility like with structural typing

Generic Types vs. Templates: Summary

Generic Types

- Modular type checking of generic class
  - Overhead (e.g., upper bounds)
- Run-time support desirable

Templates

- Type checking per instantiation
  - Flexibility like with structural typing

Generic Types vs. Templates: Summary

Generic Types

- Modular type checking of generic class
  - Overhead (e.g., upper bounds)
- Run-time support desirable

Templates

- Type checking per instantiation
  - Flexibility like with structural typing
- No need for run-time support

Generic Types vs. Templates: Summary

Generic Types

- Modular type checking of generic class
    - Overhead (e.g., upper bounds)
- Run-time support desirable
- No meta-programming

Templates

- Type checking per instantiation
    - Flexibility like with structural typing
- No need for run-time support

Generic Types vs. Templates: Summary

Generic Types

- Modular type checking of generic class
  - Overhead (e.g., upper bounds)
- Run-time support desirable
- No meta-programming

Templates

- Type checking per instantiation
  - Flexibility like with structural typing
- No need for run-time support
- Meta-programming is Turing-complete