Compiler Construction: Introduction

Lectures 8
Semantic Analysis

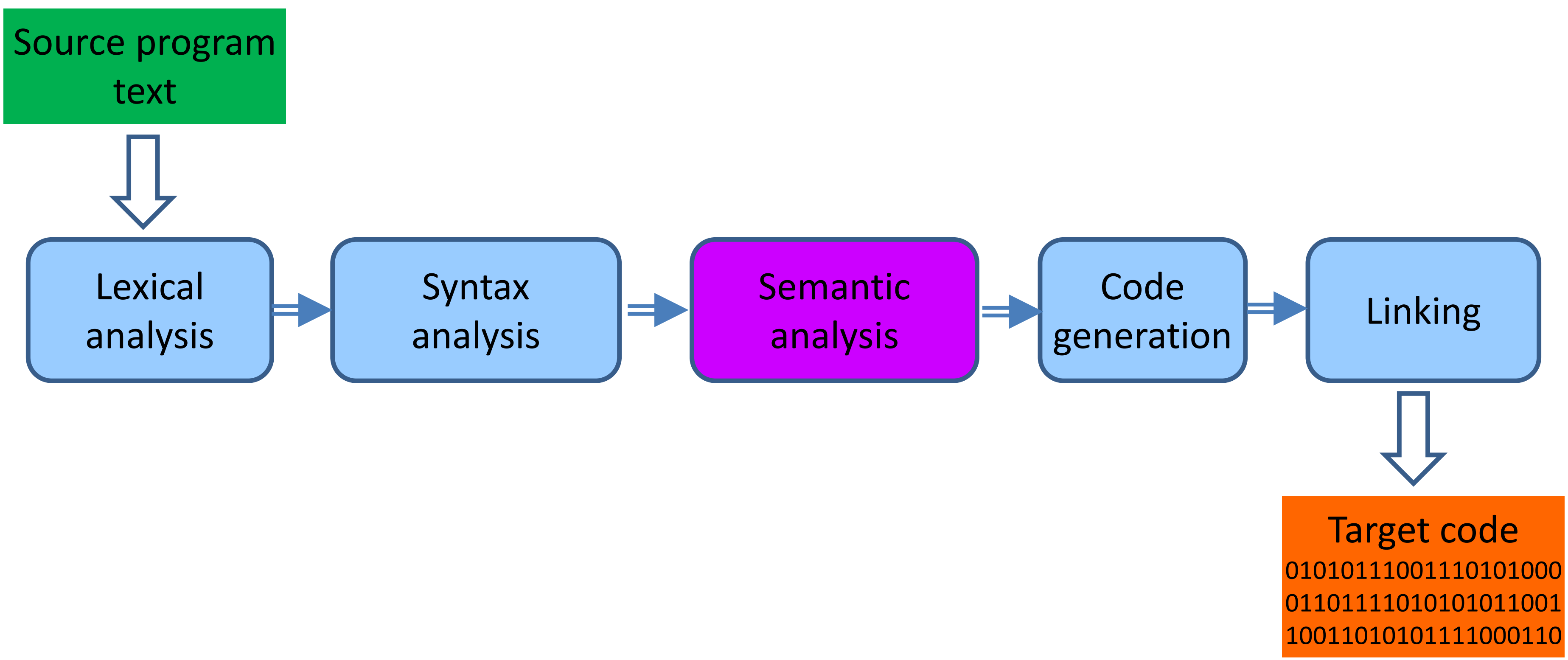
Eugene Zouev

Fall Semester 2018

Innopolis University

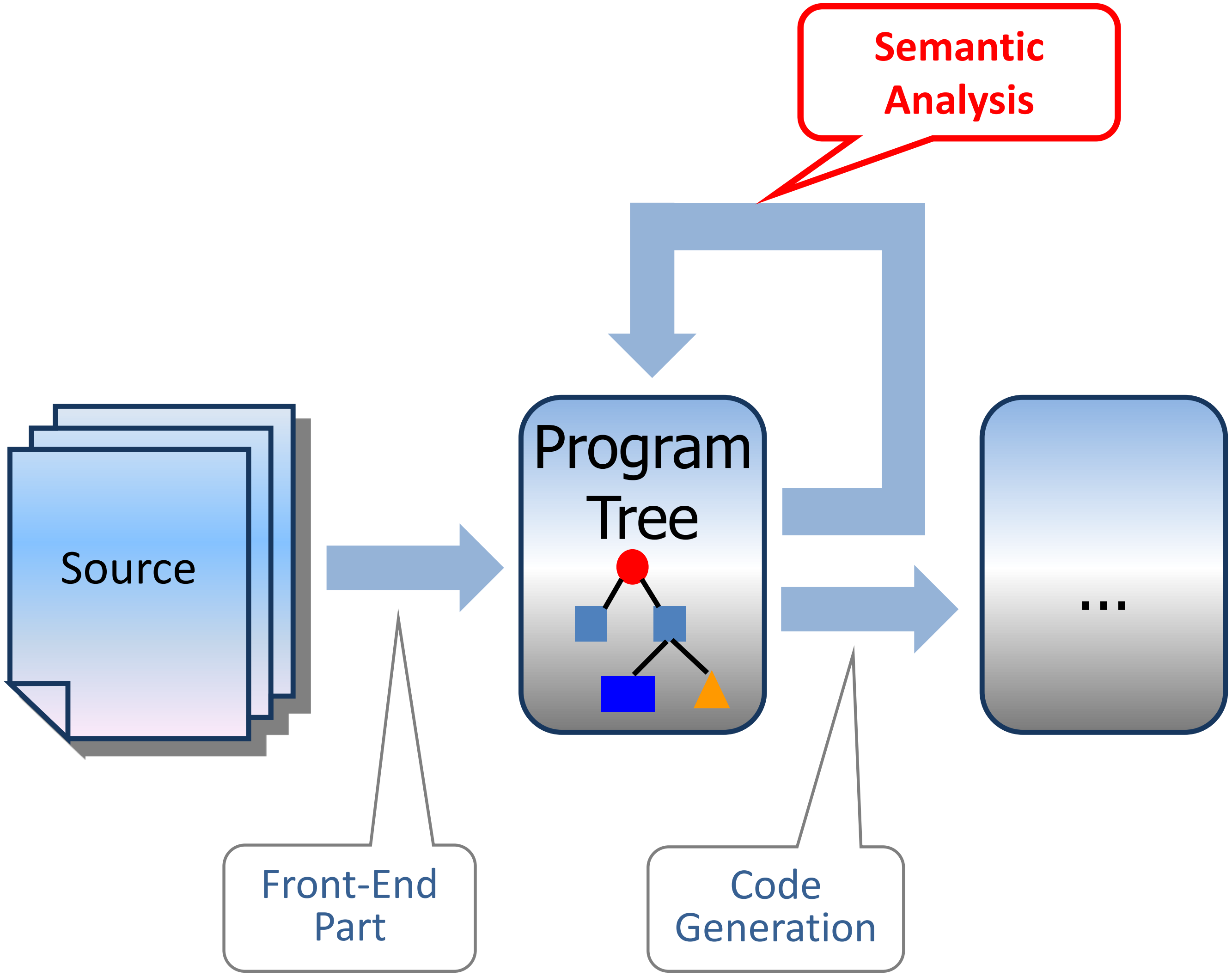
Compilation: An Ideal Picture

A program written by a human
(or by another program)



A program binary image suitable for immediate execution by a machine

Semantic analysis



Type representation (1)

From the previous lecture

C++ type system:

- Fundamental types: integer, float, character, ...
- Class and enumeration types
- Type modifiers: constants, pointers, references, pointers to class members
- Functional types, arrays
- Families of types (templates)

Many ways for defining new types, for example:

- Reference to pointer **int***& rp = p;
- Pointer to function **double**& (*f)(**const** C*);
- Array of pointers to pointers to class members

C<**int**,**float**>::***char** A[10];

Many complex & non-obvious conversion rules

Type representation (2)

From the previous lecture

Solution for C++:

- Represent types as **type chains**

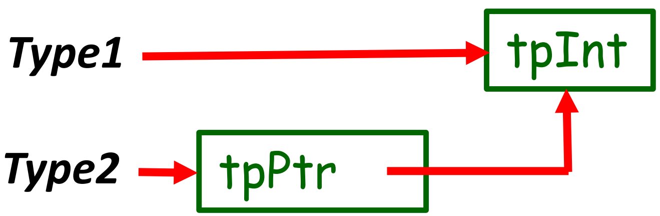
int	tpInt
int*	tpPtr,tpInt
long unsigned int**	tpPtr,tpPtr,tpULI
const int	tpConst,tpInt
const int*	tpPtr,toConst,tpInt
const int *const	tpConst,tpPtr,tpConst,tpInt
const C*[10]	tpArr,10,tpPtr,tpConst,tpClass,C
int& (*f)(**float**)**const**	tpPtr,f
C::***int**	tpPtrMemb,C,tpInt
...	
f	tpMembFun,tpRef,tpInt,1,tpFloat

Type representation (3)

Typical operations on types (and on type chains)

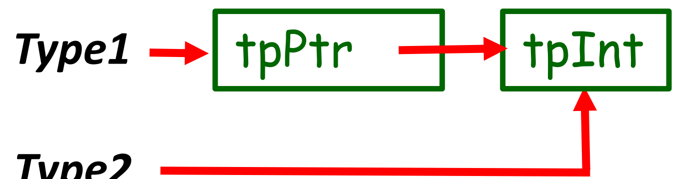
Taking address:

int -> int*



Dereferencing:

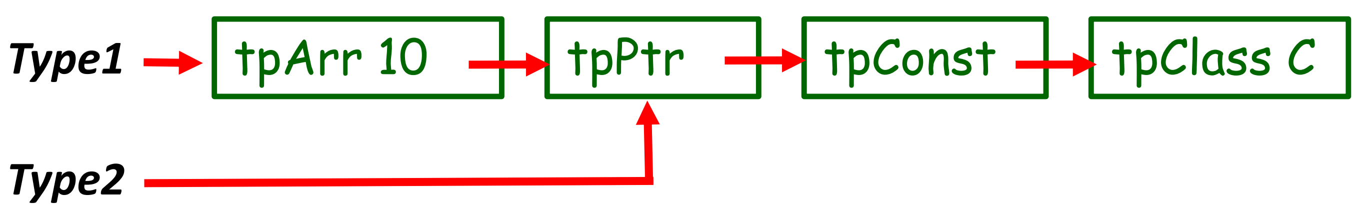
int* -> int



Access to the type of an array element:

tpArr,10,tpPtr,tpConst,tpClass,C ->

tpPtr,tpConst,tpClass,C

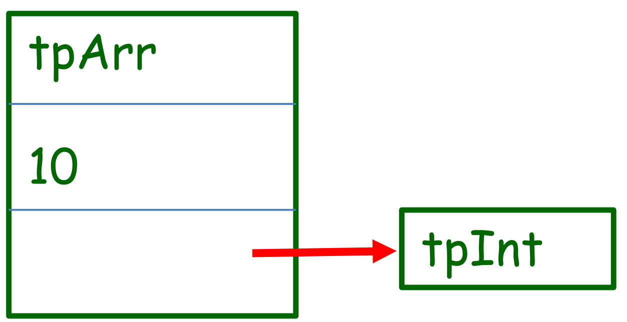


Type representation (4)

Compound types representation

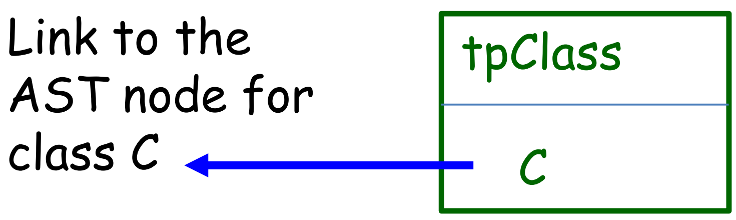
Compound type: array

int[10]



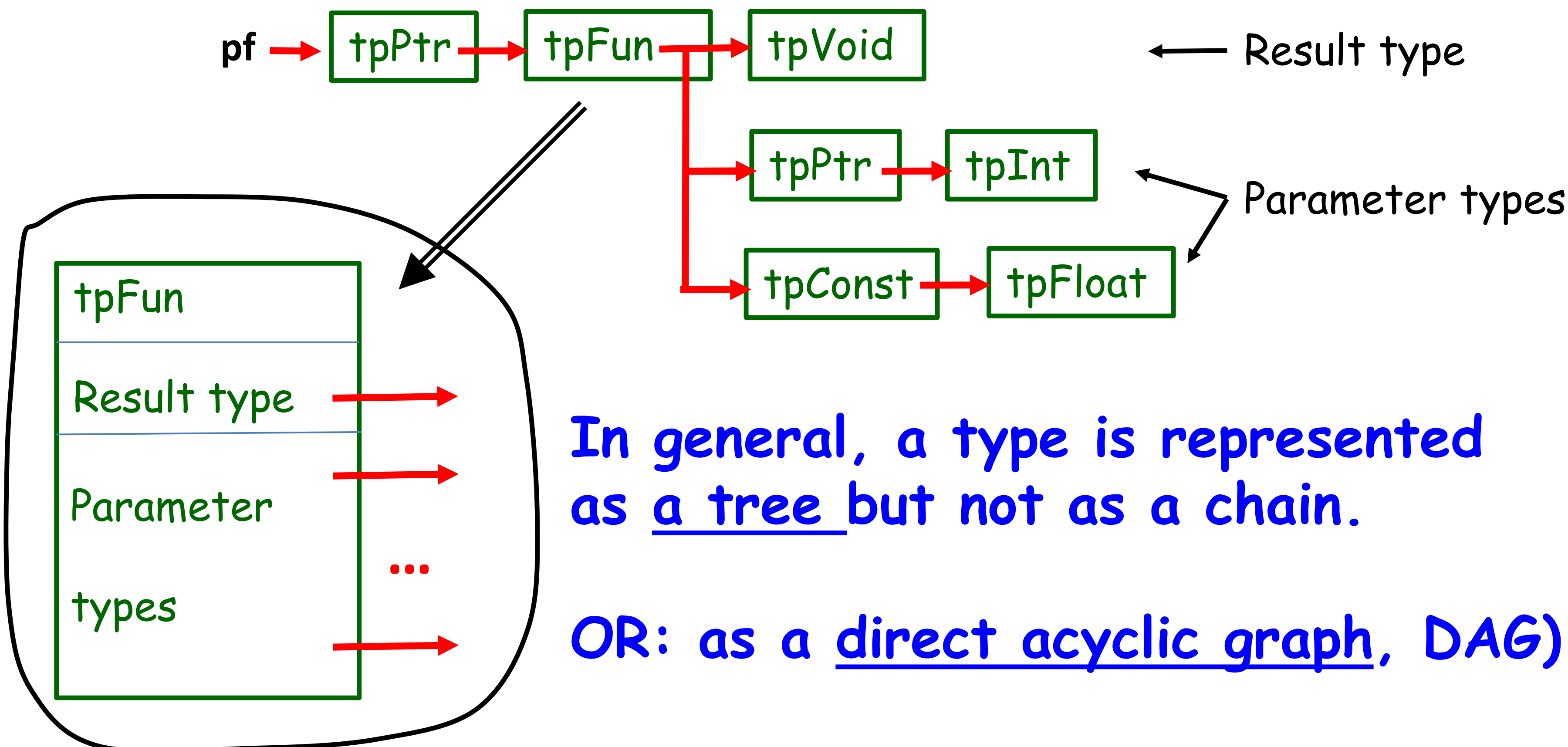
Compound type: class

class *C*



Type representation (5)

```
typedef void (*pf)(int*, const float);
```



In general, a type is represented as a tree but not as a chain.

OR: as a direct acyclic graph, DAG).

What is semantic analysis for?

```
void f(int p)
{
    int a, b;
    *a = 777;
    return xyz*a+b*f;
}
...
f();
int x = f(1,2);
```

Illegal operation (dereferencing) for an object of type int

Using uninitialized variables a and b

Undeclared variable xyz

Illegal operand types for operator *

Returning a value in void function

Illegal number of arguments in calls to function f

Illegal position for call to function f

Syntactically perfect program!..

What is semantic analysis for?

Some remarks

1. Errors like “undeclared identifier» are typically detected on syntax analysis stage – while building symbol tables and/or program tree.
2. Errors like “uninitialized variable” usually are not detected by all compilers because it requires deeper control flow and data flow analysis.
3. Analysis of the code snippet ...xyz*a... typically results in a message like “illegal operand types for * operator”. Formally that’s true but in fact the reason is that xyz is not declared – this is an example of and ***induced error***.

Semantic analysis

- Typically semantic analysis runs on the program tree built on previous compilation stages (while syntax analysis).
- Semantic analysis is typically implemented as a series of tree traverses with some actions related to the source language semantics.
- The more complex semantics is the more passes (traverses) are needed.
  - For relatively simple languages semantic analysis can be done **together** with syntax analysis while building the program tree.
  - Usually, the last tree walk implements target code generation – either an intermediate representation (like C--) or assembler code.
  - Often, before code generation, some **additional stages** after semantic analysis are necessary like building CFG & SSA representations…

Semantic analysis

- One or several semantic actions are performed on each tree walk.
- What's the result of each tree walk?
  - Either a modified program tree with **the same node types**; perhaps complex nodes get replaced for simpler ones.
    Example is *C#* compiler: after each tree walk the tree consists of the same node types.
  - Or a modified program tree **with different node types** that are more primitive but are "closer" to the target architecture.
    Example is Scala compiler: node types representing source program constructs get replaced for more primitive nodes ("ICode"), and the JVM (or MSIL) code is generated form ICode finally.

Semantic analysis

The result of each tree walk is typically twofold:

- The tree changes **its structure**: some nodes/subtrees are added or removed, some nodes/subtrees get replaced for other nodes/subtrees…
- Tree nodes are annotated ("decorated" ☺) by attributes reflecting various semantic features; the attributes are deduced during the analysis process.

=> The Abstract Syntax Tree (AST) is converted to the **Annotated** Syntax Tree (AAST).

(An alternative solution is **attribute grammars**.)

Semantic analysis: Actions

Four categories of actions while semantic analysis:

- Semantic checks

  Operand types consistency in expressions
  Проверки корректности конструкций (деструктор)

- Semantic conversions

  Replacing conversions for function calls

  Replacing infix operators for operator function calls

  Inserting necessary type conversions

  Template instantiating

- Identification of hidden semantics

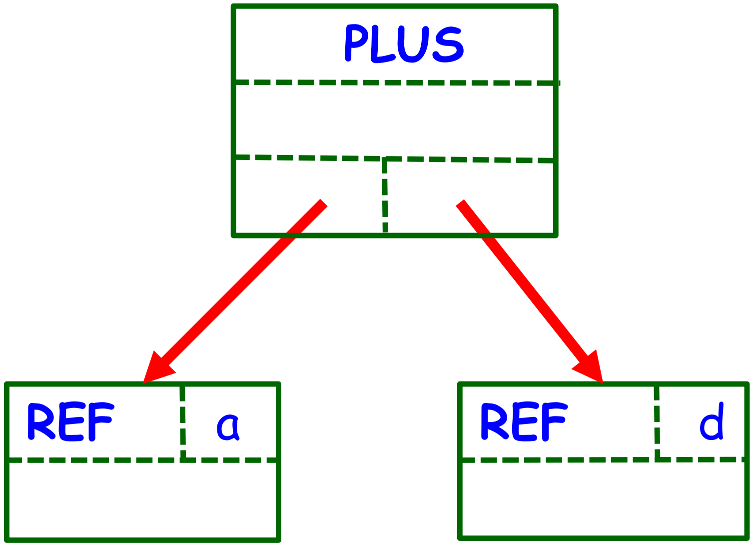
  Implicit destructor calls
  Temporary objects

- Optimizations (!)

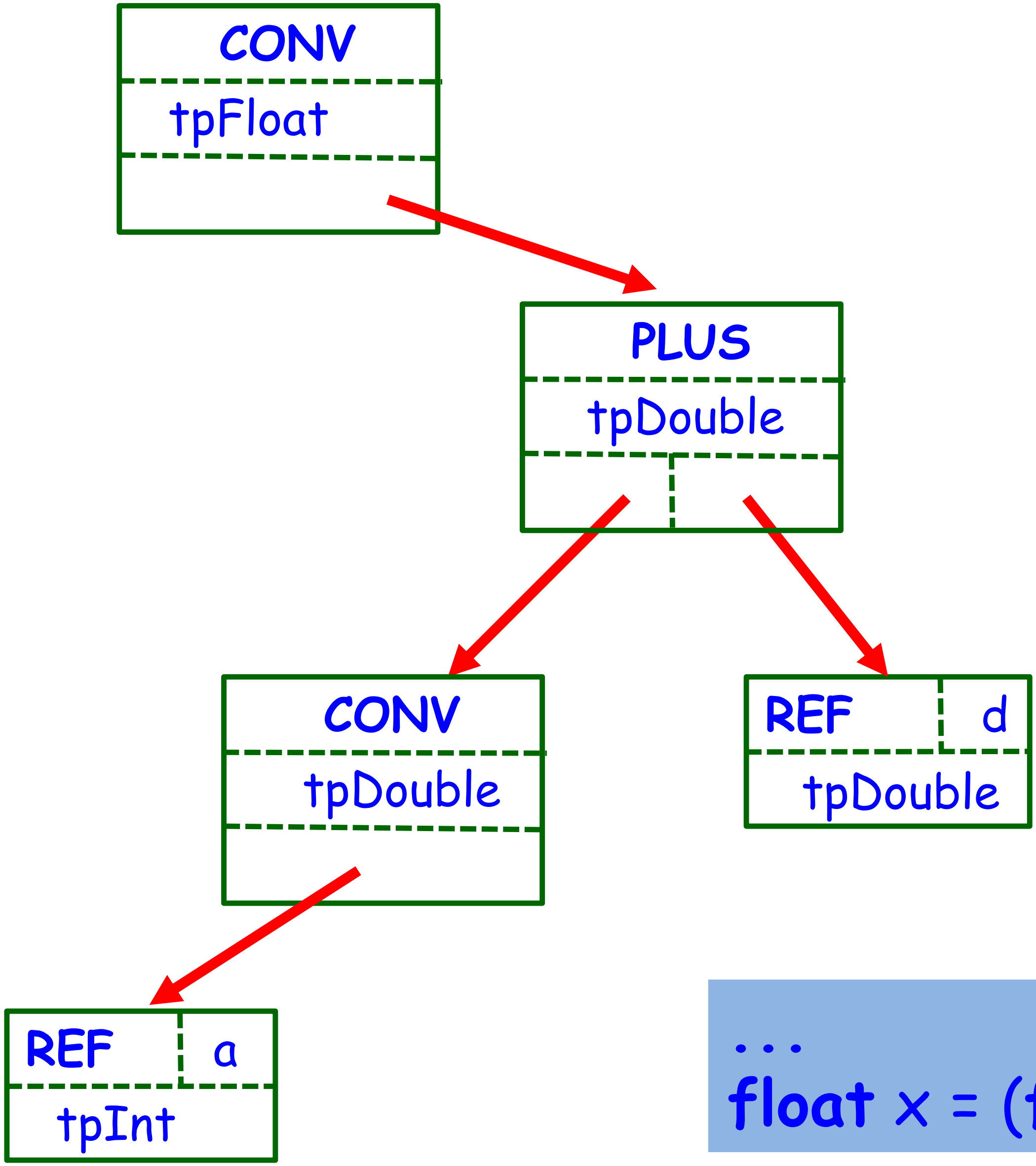
Semantic analysis: Example 1

Standard conversions

```
int a = 3;
double d = 7.55;
float x = a + d;
```



Semantic analysis: Example 1




```
...
float x = a + d;
```

```
...
float x = (float)((double)a + d);
```

Semantic analysis: Example 2

Initialization semantics

```
class C { ... };
...
C c1;
C c2(1);
C c3(c2);
C c4 = 7;
C c5 = c1;
```

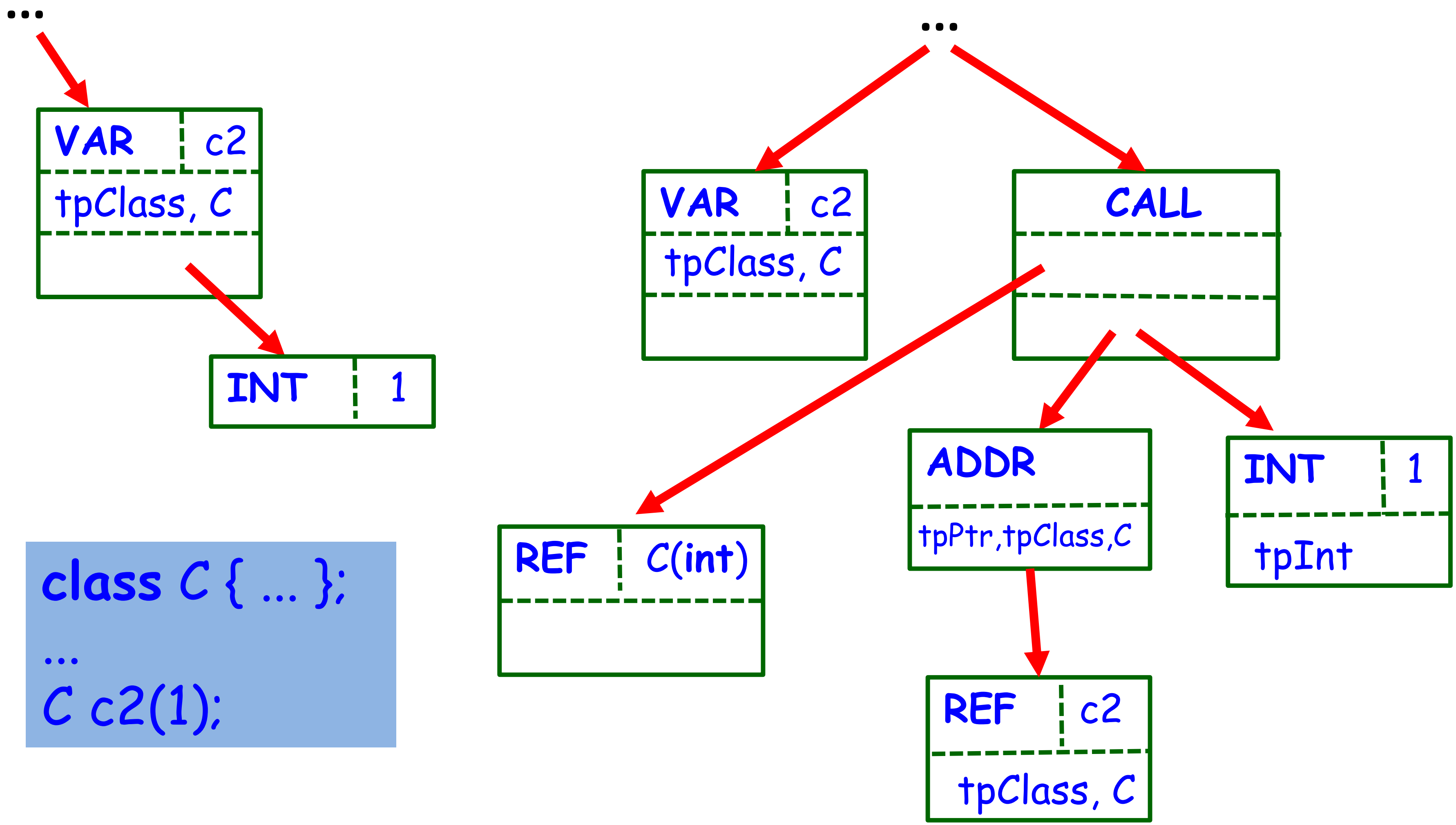
- Allocate memory for c1 object;
- Call **default constructor** of C for c1.

- Allocate memory for c2 object;
- Call **C(int)** constructor for c2.

- Allocate memory for c3 object;
- Call **copy constructor** for c3.

- Allocate memory for c4 object;
- Create temporary object tmp;
- Call **C(int)** constructor for tmp;
- Call **copy constructor** for c4.

Semantic analysis: Example 2




```
class C { ... };
...
C c2(1);
```

Semantic analysis: Example 3

User-defined conversions

```
class C {
private:
    bool m;
public:
    operator bool() { return m; }
};
...
C c;
...
if ( c ) ...
```

```
if ( (bool)c ) ...
```

```
if ( c.operator bool() ) ...
```

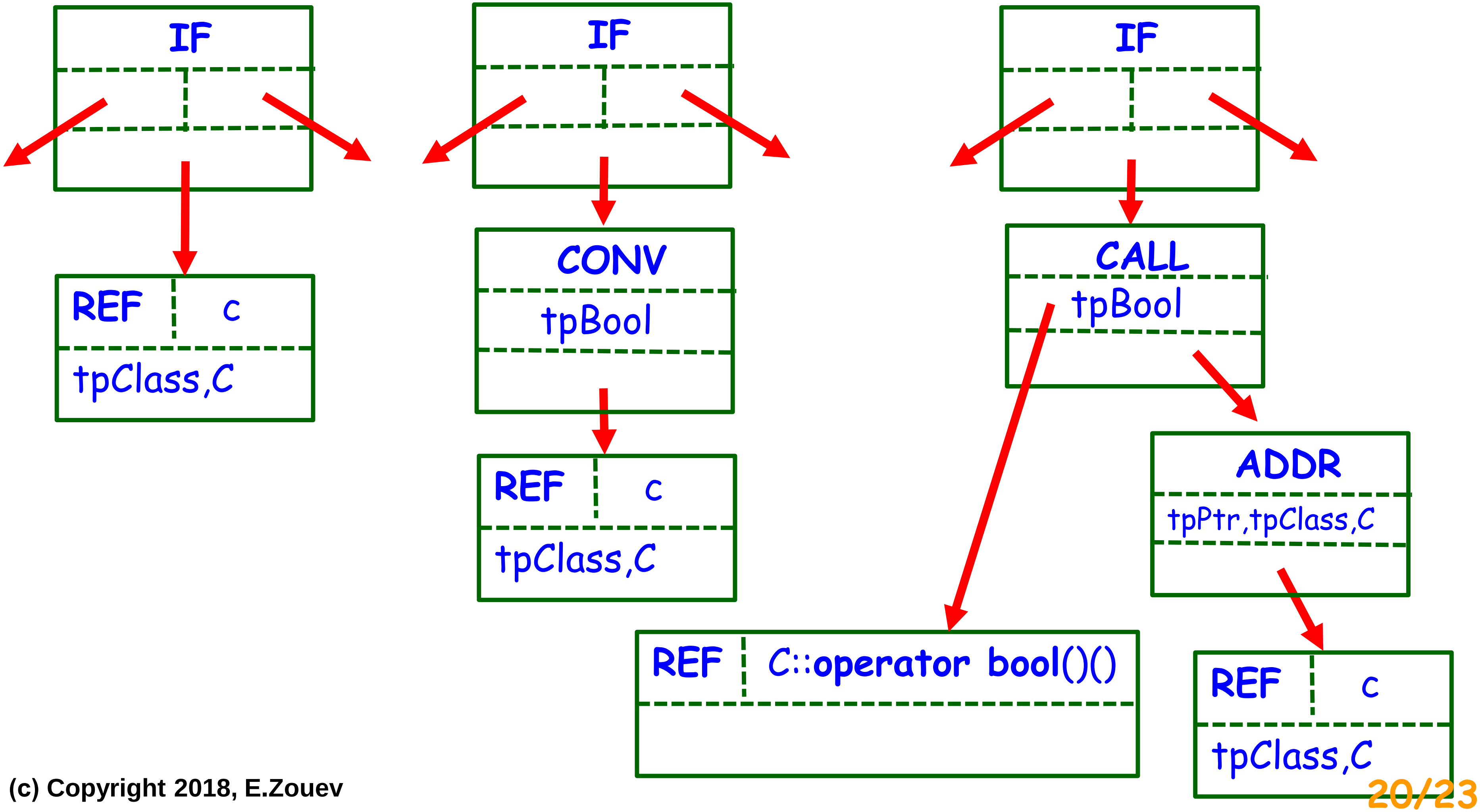
Semantic analysis: Example 3

if (c) …

if ((bool)c) …

if (c.operator bool()) …

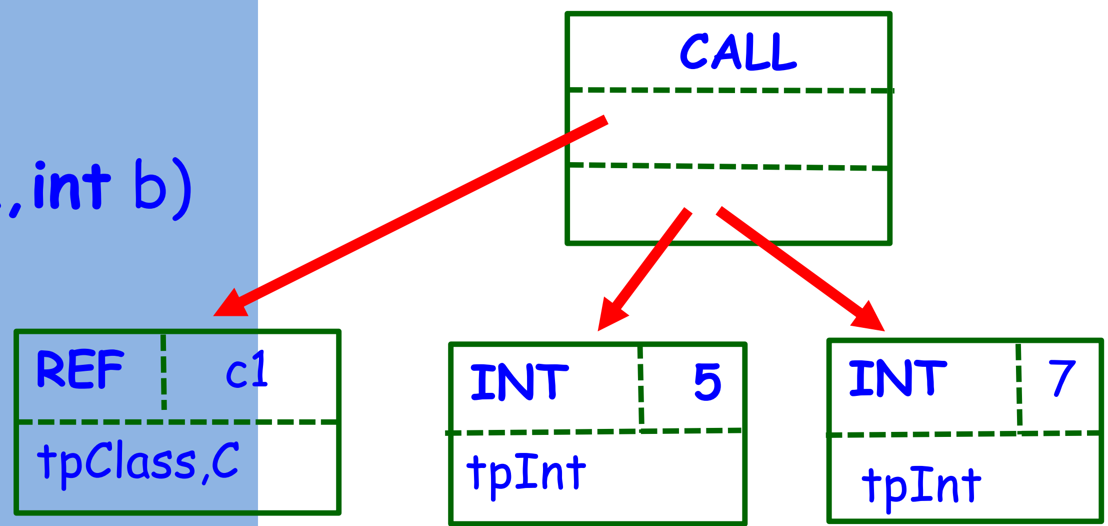
if (C::operator bool(&c)) …



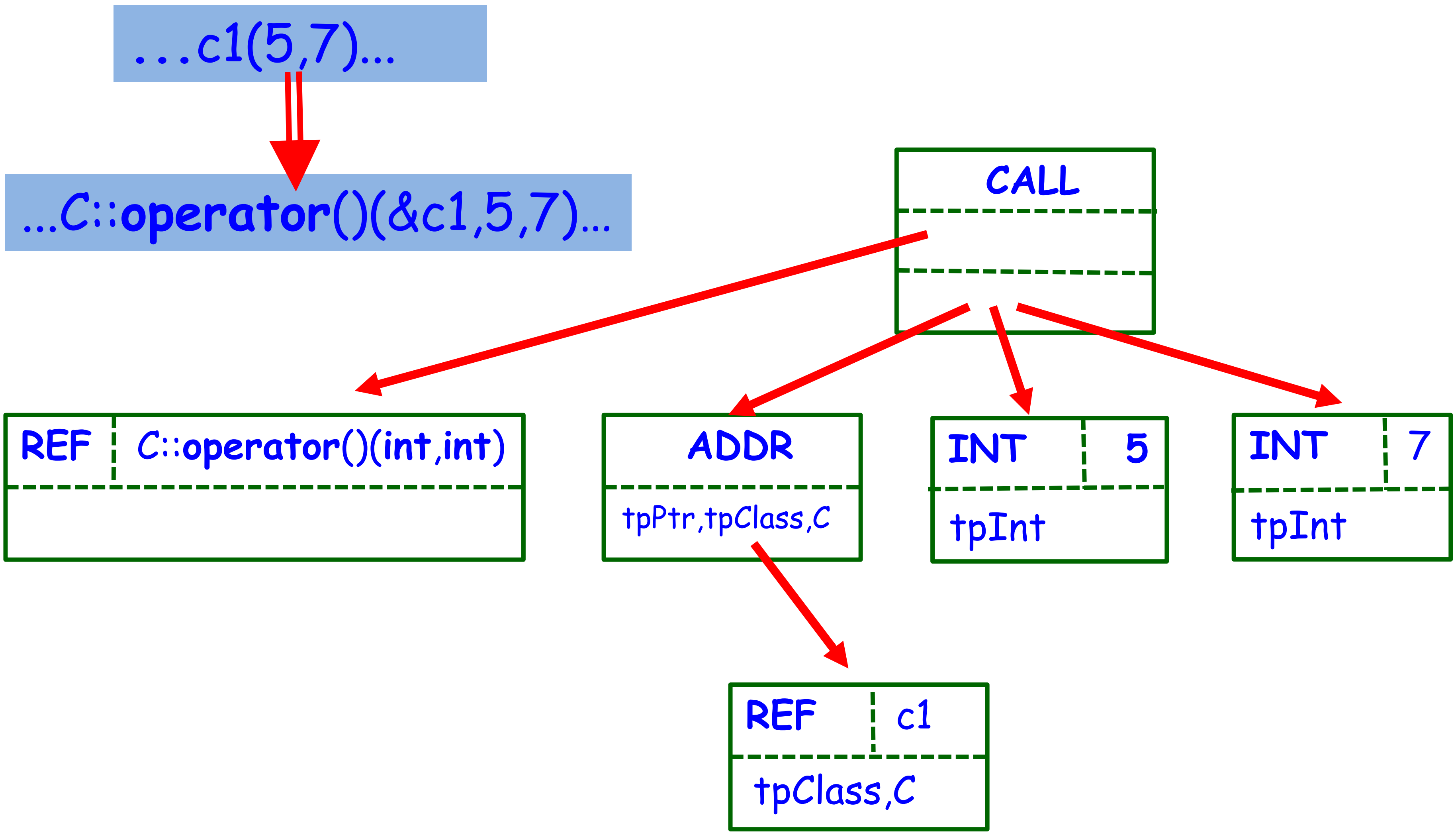
Semantic analysis: Example 4

Functional objects ("functors")

```
class C {
public:
    int operator()(int a, int b)
    { return a+b; }
};
...
C c1;
...
int res = c1(5,7);
```



Semantic analysis: Example 4



Semantic analysis: Example 5

Calculating constant expressions

```
const int Factor = 10;
int A[Factor*7-1];
```