# CS 131 Compilers: Discussion 11: Operational Semantic and Runtime Resourse Allocation

**杨易为 季杨彪 尤存翰**
{yangyw,jiyb,youch}@shanghaitech.edu.cn

2021 年 5 月 17 日

## 1 Type System Cont.

### 1.1 Last time

```
defn(I, T, [def(I, T) | _]).
defn(I, T, [def(I1, _) | R]) :- dif(I, I1), defn(I, T, R)
typeof(X, T, Env) :- defn(X, T, Env).
```

### 1.2 Type Inference in Rust/C++

Rust is a good language to do type inference utilizing the trait. Type inference is the process of automatic detection of the type of an expression.

```
fn main() {
    let mut things = vec![];
    things.push("thing");
}
```

Here, the type of *things* is inferred to be $Vec < \&str >$ because of the value we push into *things*.

The type inference is based on the standard Hindley-Milner (HM) type inference algorithm.

Before C++20, we may find it difficult to write good type inference in Generic Type. For example

```
template<typename T> T add(T in1, T in2) {
    return in1 + in2;
}
class Complex {
public:
    double real = 0;
    double imaginary = 0;
};
```

The compiler will emit such error:

1. no match for 'operator+' (operand types are 'Complex' and 'Complex')
2. 'Complex' is not derived from 'const std::cxx11::basic_string<_CharT, _Traits, _Allocator>' …
3. mismatched types 'const_CharT*' and 'Complex'

Since this template function is already available for integers and double precision numbers, the cause of the error is not in the template function itself, but in the call to the template using the actual data Complex. but the problem is that the error message does not indicate which call is the problem.

Also, the error message indicates that there is no matching + operator, which is the real cause. But then Complex does not inherit from various classes and other accompanying information only adds to the

confusion. The authors believe that any programmer who has used generic programming will have had a similar experience.

The solution is simple: just overload the + operator for Complex

### 1.3 Type Coercion

Type coercion is the automatic or implicit conversion of values from one data type to another (such as strings to numbers). Type conversion is similar to type coercion because they both convert values from one data type to another with one key difference —type coercion is implicit whereas type conversion can be either implicit or explicit.

```
// implicit conversion of classes:
#include <iostream>
using namespace std;

class A {};

class B {
public:
  // conversion from A (constructor):
  B (const A& x) {}
  // conversion from A (assignment):
  B& operator= (const A& x) {return *this;}
  // conversion to A (type-cast operator)
  operator A() {return A();}
};

int main ()
{
  A foo;
  B bar = foo;    // calls constructor
  bar = foo;      // calls assignment
  foo = bar;      // calls type-cast operator
  return 0;
}
```

## 2 Operational Semantic

Suppose we have already defined the operational semantics of basic arithmetic expressions involving the integers, variables, and booleans as follows (respectively):

$$\frac{\cdots}{\Gamma \vdash e_1 : n, \Gamma}$$

and

$$\frac{\cdots}{\Gamma \vdash b_1 : b, \Gamma}$$

$\Gamma$ here represents our mapping between variables and their values. We use the convention $e$ for arithemtic expressions and $b$ for boolean expressions. Note that as of now, expressions do not modify the state of our program. Write the operational semantics for the following types of constructs:

1. Assignment to an arithmetic expression as follows: $x := e$. Note an assignment returns void.
2. A sequence of statements $s_1$ and $s_2$ as follows: $s_1; s_2$. Note a sequence returns void.

3. An if statement as follows: if $b$ then $s_1$ else $s_2$. Note an if statement must only execute its corresponding branch.

## 2.1 Runtime support for functions