Lecture 18

Semantics Analysis

Awanish Pandey

Department of Computer Science and Engineering
Indian Institute of Technology
Roorkee

March 17, 2025

Take aways from the last class

- Type System and Type checking

Take aways from the last class

- Type System and Type checking
- Type expressions and type constructor

Take aways from the last class

- Type System and Type checking
- Type expressions and type constructor
- Rules for symbol table entry

Take aways from the last class

- Type System and Type checking
- Type expressions and type constructor
- Rules for symbol table entry
- Type checking for expressions

Take aways from the last class

- Type System and Type checking
- Type expressions and type constructor
- Rules for symbol table entry
- Type checking for expressions
- Type checking for statements

Equivalence of Type expression

- Structural equivalence: Two type expressions are equivalent if

Equivalence of Type expression

- Structural equivalence: Two type expressions are equivalent if
  - either these are same basic types

Equivalence of Type expression

- Structural equivalence: Two type expressions are equivalent if
  - either these are same basic types
  - or these are formed by applying same constructor to equivalent types

Equivalence of Type expression

- Structural equivalence: Two type expressions are equivalent if
  - either these are same basic types
  - or these are formed by applying same constructor to equivalent types
- Name equivalence: types can be given names

Equivalence of Type expression

C uses structural equivalence except for structs.

- Structural equivalence: Two type expressions are equivalent if
  - either these are same basic types
  - or these are formed by applying same constructor to equivalent types
- Name equivalence: types can be given names
  - Two type expressions are equivalent if they have the same name

Function to test structural equivalence

Algorithm bool structureEquivalence(s, t)

```
1: if s and t are same basic types then
2:     return true
3: else if s == array(s1, s2) and t == array(t1, t2) then
4:     return return structureEquivalence(s1, t1) && structureEquivalence(s2, t2)
5: else if s == s1 × s2 and t == t1 × t2 then
6:     return structureEquivalence(s1, t1) && structureEquivalence(s2, t2)
7: else if s == pointer(s1) and t == pointer(t1) then
8:     return structureEquivalence(s1, t1)
9: else if s == s1 → s2 and t == t1 → t2 then
10:    return structureEquivalence(s1,t1) && structureEquivalence(s2,t2)
11: else
12:    return false
13: end if
```

Name equivalence

- Consider the following code

```
typedef int Integer
int next,last;
Integer p,q,r;
```

Name equivalence

- Consider the following code

```
typedef int Integer
int next,last;
Integer p,q,r;
```

- Do the variables next, last, p, q and r have identical types ?

Name equivalence

- Consider the following code

```
typedef int Integer
int next,last;
Integer p,q,r;
```

- Do the variables next, last, p, q and r have identical types ?
- Name equivalence views each type name as a distinct type

Name equivalence

- Consider the following code
```
typedef int Integer
int next,last;
Integer p,q,r;
```
- Do the variables next, last, p, q and r have identical types ?
- Name equivalence views each type name as a distinct type
- Under name equivalence $next = last$ and $p = q = r$, however, $next \neq p$

Name equivalence

- Consider the following code

```
typedef int Integer
int next,last;
Integer p,q,r;
```

- Do the variables next, last, p, q and r have identical types ?
- Name equivalence views each type name as a distinct type
- Under name equivalence $next = last$ and $p = q = r$, however, $next \neq p$
- Under structural equivalence all the variables are of the same type

Notes

- C uses structural equivalence for all types except structures.

Notes

- C uses structural equivalence for all types except structures.
- It uses the acyclic structure of the type graph

Notes

- C uses structural equivalence for all types except structures.
- It uses the acyclic structure of the type graph
- Type names must be declared before they are used

Notes

- C uses structural equivalence for all types except structures.
- It uses the acyclic structure of the type graph
- Type names must be declared before they are used
  - However, allow pointers to undeclared structure types

Notes

- C uses structural equivalence for all types except structures.
- It uses the acyclic structure of the type graph
- Type names must be declared before they are used
  - However, allow pointers to undeclared structure types
  - All potential cycles are due to pointers to structure

Type Conversion

- Consider expression like $x + i$ where x is of type *floats* and i is of type *integer*

Type Conversion

- Consider expression like $x + i$ where x is of type *floats* and i is of type *integer*
- Internal representations of *integers* and *floats* are different in a computer

Type Conversion

- Consider expression like $x + i$ where x is of type *floats* and i is of type *integer*
- Internal representations of *integers* and *floats* are different in a computer
- Different machine instructions are used for operations on *integers* and *floats*

Type Conversion

- Consider expression like $x + i$ where x is of type *floats* and i is of type *integer*
- Internal representations of *integers* and *floats* are different in a computer
- Different machine instructions are used for operations on *integers* and *floats*
- The compiler has to convert both the operands to the same type

Type Conversion

- Consider expression like $x + i$ where x is of type *floats* and i is of type *integer*
- Internal representations of *integers* and *floats* are different in a computer
- Different machine instructions are used for operations on *integers* and *floats*
- The compiler has to convert both the operands to the same type
- Language definition specifies what conversions are necessary.

Type Conversion

- Consider expression like $x + i$ where x is of type *floats* and i is of type *integer*
- Internal representations of *integers* and *floats* are different in a computer
- Different machine instructions are used for operations on *integers* and *floats*
- The compiler has to convert both the operands to the same type
- Language definition specifies what conversions are necessary.
- Usually conversion is to the type of the left hand side

Type Conversion

- Consider expression like $x + i$ where x is of type *floats* and i is of type *integer*
- Internal representations of *integers* and *floats* are different in a computer
- Different machine instructions are used for operations on *integers* and *floats*
- The compiler has to convert both the operands to the same type
- Language definition specifies what conversions are necessary.
- Usually conversion is to the type of the left hand side
- Type checker is used to insert conversion operations:
  $x + i \rightarrow float + inttofloat(i)$

Type Conversion

- Consider expression like $x + i$ where x is of type *floats* and i is of type *integer*
- Internal representations of *integers* and *floats* are different in a computer
- Different machine instructions are used for operations on *integers* and *floats*
- The compiler has to convert both the operands to the same type
- Language definition specifies what conversions are necessary.
- Usually conversion is to the type of the left hand side
- Type checker is used to insert conversion operations:
  $x + i \rightarrow float + inttofloat(i)$
- Type conversion is called implicit if done by compiler.

Type Conversion

- Consider expression like $x + i$ where x is of type *floats* and i is of type *integer*
- Internal representations of *integers* and *floats* are different in a computer
- Different machine instructions are used for operations on *integers* and *floats*
- The compiler has to convert both the operands to the same type
- Language definition specifies what conversions are necessary.
- Usually conversion is to the type of the left hand side
- Type checker is used to insert conversion operations:
  $x + i \rightarrow float + inttofloat(i)$
- Type conversion is called implicit if done by compiler.
- It is limited to the situations where no information is lost

Type Conversion

- Consider expression like $x + i$ where x is of type *floats* and i is of type *integer*
- Internal representations of *integers* and *floats* are different in a computer
- Different machine instructions are used for operations on *integers* and *floats*
- The compiler has to convert both the operands to the same type
- Language definition specifies what conversions are necessary.
- Usually conversion is to the type of the left hand side
- Type checker is used to insert conversion operations:
  $x + i \rightarrow float + inttofloat(i)$
- Type conversion is called implicit if done by compiler.
- It is limited to the situations where no information is lost
- Conversions are explicit if programmer has to write something to cause conversion

Type checking for expressions

- $E \rightarrow num \quad E.type = int$

Type checking for expressions

- $E \rightarrow num \quad E.type = int$
- $E \rightarrow num.num \quad E.type = float$

Type checking for expressions

- $E \rightarrow num \quad E.type = int$
- $E \rightarrow num.num \quad E.type = float$
- $E \rightarrow id \quad E.type = lookup(id.entry)$

Type checking for expressions

- $E \rightarrow num \quad E.type = int$
- $E \rightarrow num.num \quad E.type = float$
- $E \rightarrow id \quad E.type = lookup(id.entry)$
- $E \rightarrow E_1 op E_2$

Type checking for expressions

- $E \rightarrow num \quad E.type = int$
- $E \rightarrow num.num \quad E.type = float$
- $E \rightarrow id \quad E.type = lookup(id.entry)$
- $E \rightarrow E_1 op E_2 \quad E.type = if \ E_1.type == int \ \&\& \ E_2.type == int$ $then \ int$

Type checking for expressions

- $E \rightarrow num \quad E.type = int$
- $E \rightarrow num.num \quad E.type = float$
- $E \rightarrow id \quad E.type = lookup(id.entry)$
- $E \rightarrow E_1 op E_2 \quad E.type = if \ E_1.type == int \ \&\& \ E_2.type == int$
  $then \ int$
  $elseif \ E_1.type == int \ \&\& \ E_2.type == float$
  $then \ float$

Type checking for expressions

- $E \rightarrow num \quad E.type = int$
- $E \rightarrow num.num \quad E.type = float$
- $E \rightarrow id \quad E.type = lookup(id.entry)$
- $E \rightarrow E_1 op E_2 \quad E.type = if \ E_1.type == int \ \&\& \ E_2.type == int$
  $then \ int$
  $elseif \ E_1.type == int \ \&\& \ E_2.type == float$
  $then \ float$
  $elseif \ E_1.type == float \ \&\& \ E_2.type == int$
  $then \ float$

Type checking for expressions

- $E \rightarrow num \quad E.type = int$
- $E \rightarrow num.num \quad E.type = float$
- $E \rightarrow id \quad E.type = lookup(id.entry)$
- $E \rightarrow E_1 op E_2 \quad E.type = if \ E_1.type == int \ \&\& \ E_2.type == int$
  $then \ int$
  $elseif \ E_1.type == int \ \&\& \ E_2.type == float$
  $then \ float$
  $elseif \ E_1.type == float \ \&\& \ E_2.type == int$
  $then \ float$
  $elseif \ E_1.type == float \ \&\& \ E_2.type == float$
  $then \ float$

implicit type conversion have to be done.

Overloaded functions and operators

- Overloaded symbol has different meaning depending upon the context

Overloaded functions and operators

- Overloaded symbol has different meaning depending upon the context
- In maths + is overloaded; used for integer, real, complex, matrices

Overloaded functions and operators

- Overloaded symbol has different meaning depending upon the context
- In maths $+$ is overloaded; used for integer, real, complex, matrices
- Overloading is resolved when a unique meaning for an occurrence of a symbol is determined

Overloaded functions and operators

- Overloaded symbol has different meaning depending upon the context
- In maths + is overloaded; used for integer, real, complex, matrices
- Overloading is resolved when a unique meaning for an occurrence of a symbol is determined
- Suppose only possible type for 2, 3 and 5 is integer and Z is a complex variable

Overloaded functions and operators

- Overloaded symbol has different meaning depending upon the context
- In maths + is overloaded; used for integer, real, complex, matrices
- Overloading is resolved when a unique meaning for an occurrence of a symbol is determined
- Suppose only possible type for 2, 3 and 5 is integer and Z is a complex variable
  - then 3*5 is either integer or complex depending upon the context

Overloaded functions and operators

- Overloaded symbol has different meaning depending upon the context
- In maths + is overloaded; used for integer, real, complex, matrices
- Overloading is resolved when a unique meaning for an occurrence of a symbol is determined
- Suppose only possible type for 2, 3 and 5 is integer and Z is a complex variable
  - then 3*5 is either integer or complex depending upon the context
  - in 2*(3*5),

Overloaded functions and operators

- Overloaded symbol has different meaning depending upon the context
- In maths + is overloaded; used for integer, real, complex, matrices
- Overloading is resolved when a unique meaning for an occurrence of a symbol is determined
- Suppose only possible type for 2, 3 and 5 is integer and Z is a complex variable
  - then 3*5 is either integer or complex depending upon the context
  - in 2*(3*5),3*5 is integer because 2 is integer

Overloaded functions and operators

- Overloaded symbol has different meaning depending upon the context
- In maths + is overloaded; used for integer, real, complex, matrices
- Overloading is resolved when a unique meaning for an occurrence of a symbol is determined
- Suppose only possible type for 2, 3 and 5 is integer and Z is a complex variable
  - then 3*5 is either integer or complex depending upon the context
  - in 2*(3*5),3*5 is integer because 2 is integer
  - in Z*(3*5),

Overloaded functions and operators

- Overloaded symbol has different meaning depending upon the context
- In maths + is overloaded; used for integer, real, complex, matrices
- Overloading is resolved when a unique meaning for an occurrence of a symbol is determined
- Suppose only possible type for 2, 3 and 5 is integer and Z is a complex variable
  - then 3*5 is either integer or complex depending upon the context
  - in 2*(3*5),3*5 is integer because 2 is integer
  - in Z*(3*5), 3*5 is complex because Z is complex

Type Resolution

- Try all possible types of each overloaded function (possible but brute force method!)

Type Resolution

- Try all possible types of each overloaded function (possible but brute force method!)
- Keep track of all possible types

Type Resolution

- Try all possible types of each overloaded function (possible but brute force method!)
- Keep track of all possible types
- Discard invalid possibilities

Type Resolution

- Try all possible types of each overloaded function (possible but brute force method!)
- Keep track of all possible types
- Discard invalid possibilities
- At the end, check if there is a single unique type

Type Resolution

- Try all possible types of each overloaded function (possible but brute force method!)
- Keep track of all possible types
- Discard invalid possibilities
- At the end, check if there is a single unique type
- Overloading can be resolved in two passes:

Type Resolution

- Try all possible types of each overloaded function (possible but brute force method!)
- Keep track of all possible types
- Discard invalid possibilities
- At the end, check if there is a single unique type
- Overloading can be resolved in two passes:
  - Bottom up: compute set of all possible types for each expression

Type Resolution

- Try all possible types of each overloaded function (possible but brute force method!)
- Keep track of all possible types
- Discard invalid possibilities
- At the end, check if there is a single unique type
- Overloading can be resolved in two passes:
  - Bottom up: compute set of all possible types for each expression
  - Top down: narrow set of possible types based on what could be used in an expression

Determining set of possible types

- $E' \rightarrow E \quad E'.types = E.types$

Determining set of possible types

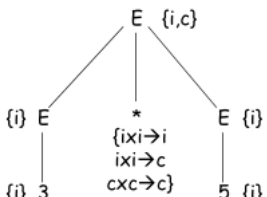
- $E' \rightarrow E \quad E'.types = E.types$
- $E \rightarrow id \quad E.types = lookup(id)$

Determining set of possible types

- $E' \rightarrow E \quad E'.types = E.types$
- $E \rightarrow id \quad E.types = lookup(id)$
- $E \rightarrow E_1(E_2) \quad E.types = \{t |$ there exists an s in $E_2.types$ and $s \rightarrow t$ is in $E_1.types\}$

Determining set of possible types

- $E' \rightarrow E \quad E'.types = E.types$
- $E \rightarrow id \quad E.types = lookup(id)$
- $E \rightarrow E_1(E_2) \quad E.types = \{t |$ there exists an s in $E_2.types$ and $s \rightarrow t$ is in $E_1.types\}$



Narrowing the set of possible types

- Ada requires a complete expression to have a unique type

Narrowing the set of possible types

- Ada requires a complete expression to have a unique type
- Given a unique type from the context we can narrow down the type choices for each expression

Narrowing the set of possible types

- Ada requires a complete expression to have a unique type
- Given a unique type from the context we can narrow down the type choices for each expression
- If this process does not result in a unique type for each subexpression, then a type error is declared for the expression

- $E' \rightarrow E \quad E'.types = E.types$
  $E.unique = if \ E'.types == t \ then \ t$
  $else \ type\_error$

- $E' \rightarrow E \quad E'.types = E.types$
  $E.unique = if \ \ E'.types == t \ \ then \ t$
  $else \ \ type\_error$
- $E \rightarrow id \quad E.types = lookup(id)$

- $E' \rightarrow E \quad E'.types = E.types$
  $E.unique = if\ \ E'.types == t\ \ then\ t$
  $else\ \ type\_error$
- $E \rightarrow id \quad E.types = lookup(id)$
- $E \rightarrow E_1(E_2) \quad E.types = \{t|$ there exists an s in $E_2.types\ and\ s \rightarrow t$ is in $E_1.types\}$
  $t = E.unique$
  $S = \{s|s \in E_2.types$ and $(s \rightarrow t) \in E_1.types\}$
  $E_2.unique = if\ \ S == s$ then s else $type\_error$
  $E_1.unique = if\ \ S == s\ \ then\ s \rightarrow t$ else $type\_error$