Type Checking

Output of Parsing

We now understand pretty well how parsing works. We have seen that it is possible to attach program fragments (actions) to grammar rules, that are carried out when the rule is reduced.

We have also seen that it is possible to make reductions emit statements, so that the total output of the parser will be an executable programme.

Unfortunately, this method works only for very simple programming languages, (simple type checking rules, not too many forward references), and it makes optimization very difficult.

In the rest of these slides, we will assume that the parser outputs a tree (usually this tree is called abstract syntax tree), which can be further processed.

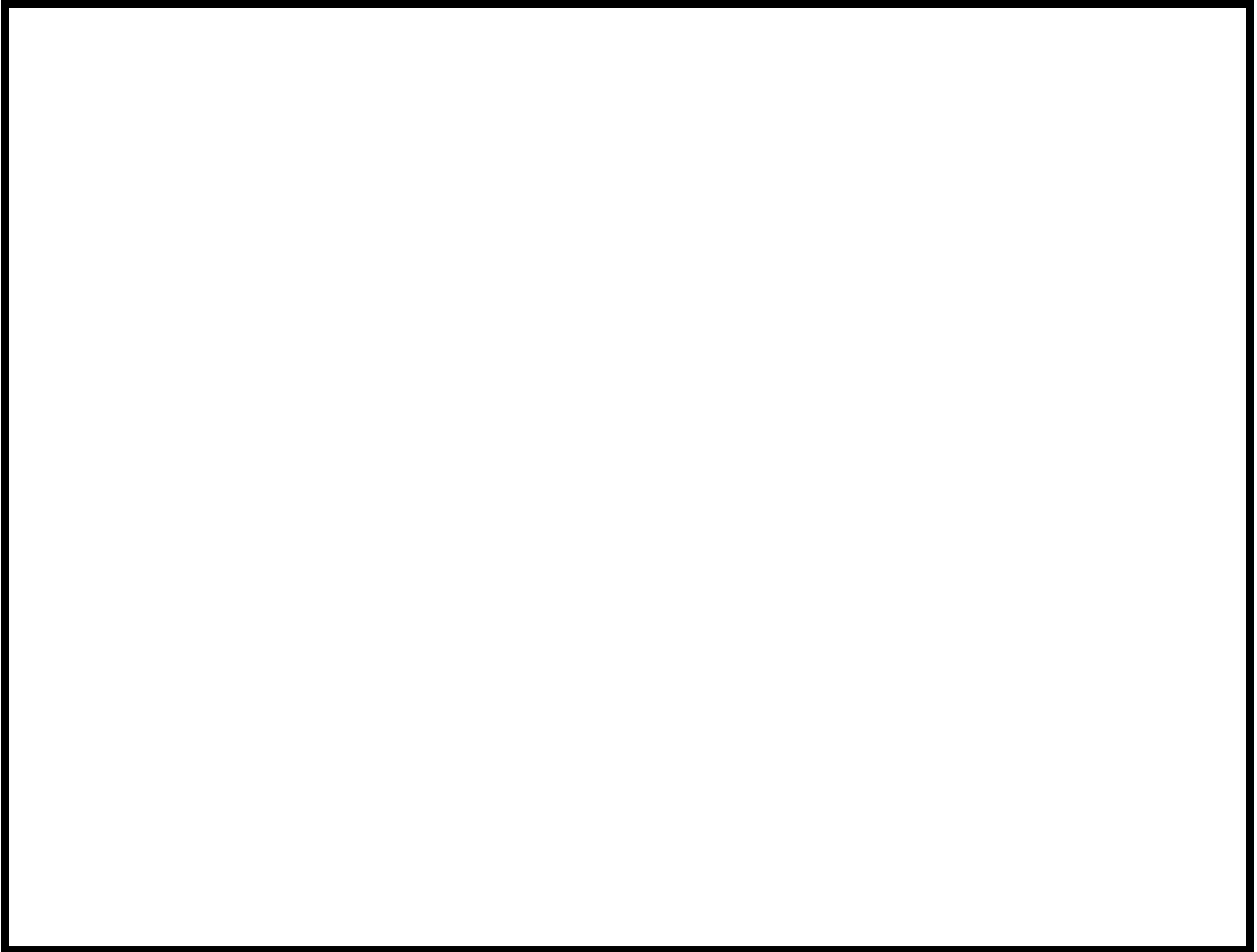
Abstract Syntax Trees

The abstract syntax tree (AST) is a simple tree representation of the parse tree. For example

```
sum = 0;
for( unsigned int i = 0; i < n; ++ i )
{
   sum += i * i;
}
```

can be represented by

```
statementlist(
   assign( sum, 0 ),
   for( init( i, 0 ), <( i, n ), pref++( i ),
      +=( sum, *( i, i )) )
)
```



Abstract Syntax Trees

In most cases, the AST is more or less equal to the parse tree, but with some of the obvious junk removed.

(1) $S \rightarrow S + T$

(2) $S \rightarrow T$

(3) $T \rightarrow T * U$

(4) $T \rightarrow U$

(5) $U \rightarrow \textbf{num}$

(6) $U \rightarrow \textbf{ident}$

(7) $U \rightarrow (S)$

Only rules 1,3 and possibly also rules 5,6, need to be represented in the AST.

Type Systems

Humans have the annoying habit of giving the same name to different operators.

- For example, + has different definitions on **int**, **real**, **complex**, etc.
- In class hierarchies, a method with the same name can be defined at many points in the class hierarchy.

The poor compiler has to find out:

1. which version of the operator was meant, and
2. whether there exists an applicable operator under the given name at all.

An ambiguous name that will be resolved at compile time, is called overloaded. When the name will be resolved at run time, it is called polymorphic.

Type Systems

Why is it acceptable to give the same name to different operators?

Probably because the different operators have very much in common. This formally means that they satisfy a common axiom system. (share an important set of properties that can be considered as defining the operator.)

For example, all instantiations of + are commutative and associative and distribute over multiplication. (when we ignore rounding errors.)

Overwritten methods in class hierarchies should have so much in common with their ancestors that it justified to give them the same name.

Run Time versus Compile Time Type Resolution

- If we are lucky, we can guarantee at compile time that an operator exists, and we can decide at compile time, which operator will be called. This means that the compiler can tell the user immediately, when no proper operator exists, and that no further checks are necessary at run time.
- If we are a little bit less lucky, we can still guarantee at compile time that a proper operator exists, (and give an error message when no operator exists), but we cannot decide anymore which operator will be called at run time. This situation usually occurs with **virtual** functions in C^{++}, or overloaded methods in Java.
- If we are totally unlucky, we can say nothing useful at compile time. This is the case for some interpreted languages, like Perl and Python.

Run Time Selection

If type selection has to be postponed until run time, then every object need a field (of some **enum** type) that indicates its type.

When a polymorphic function is called, one must use **switch( )**, or a chain of **if**s to select the proper function.

This makes execution of the code less efficient.

Recursive Definition of Types

In most programming languages, the set of possible types can be given by a recursive definition. This means that we have a set of rules of form

1. T is a type.
2. If $T_1, \ldots, T_n$ are types, then $\phi(T_1, \ldots, T_n)$ is also a type.

The types that are obtained by a rule of the first form are called primitive types.

The types that are obtained by a rule of the second form, are called derived or compound types.

Often, primitive types correspond to the types that are built-in into the processor, but this is not necessarily the case.

We see that a type is simply some kind of tree.

Primitive Types

Primitive types typically include **bool, char, int, unsigned int, float, double** and maybe some others.

Are **structs** and **classes** primitive or compound?

They may appear compound at first, but:

Intentional/Extensional Type Equivalence

```
struct type1
{
   int a1; int a2;
};

struct type2
{
   int a1; int a2;
};

type1 v1;
v1. a1 = 4;
v1. a2 = 5;
type2 v2 = v1;   // Do we allow this?
```

Intensional/Extensional Type Equivalence

Extensional Type Equivalence Types are considered equal if they are built-up in the same way. Extensional type equivalence would allow the assignment on the previous slide. Extensional type equivalence can be easily implemented by always expanding the type definitions.

Intensial Type Equivalence Types are considered equal only if they have the same name. The modern view prefers intensial type equivalence, because it allows the construction of richer type systems, and it is consistent with hiding of implementation.

Non-Recursiveness of Struct

If one would have a type constructor for **struct**, then the definition of some **structs** would be not well-founded anymore:

```
struct list
{
   int elem;
   struct list* next;
};
```

If one would have a constructor $\mathbf{struct}_{E,N}(t_1, t_2)$, then one would have $L = \mathbf{struct}_{E,N}(\ \mathbf{int},\ \mathbf{pointer}(L)\ )$.

This, in combination with the fact that in most programming languages, **struct**-equivalence is intensional, suggest that **struct** types should be treated as primitive, user-defined types. I will later explain how it is done.

Type Checking

A type checker in a compiler must do the following:

1. For polymorphic operators, the type checker should check that an instance can be selected at run time. (Not all languages require this, but C^{++} and Java do)
2. Decide, for overloaded functions, which version must be used.
3. Decide if implicit conversions are necessary. (For example, converting **int** to **double**, or changing references into objects by copy constructors.) If conversions are necessary, they must be inserted.

Type Checking

For most languages, bottom up type checking is sufficient:

In a term of form $f(t_1, \ldots, t_n)$, first compute all possible types of all t_i. Then select then compute all possible types of $f(t_1, \ldots, t_n)$.

Type Checking Algorithm

Definition A type judgement is an object of form $t{:}T$, where t is a term, and T a type. The meaning of $t{:}T$ is that t has type T.

If the type checking algorithm typecheck(t) can find a type for t, then it returns a judgement $t'{:}T$, where t' is obtained from t by inserting conversions.

Type Checking Algorithm

typejudgement typecheck(term t)

if t is a primitive object (number, bool, etc.) then

return t: T, where T is the standard type of T.

if t is an identifier and t is declared, then

return t: T, where T is the type with which t is declared.

(In C/C^{++}, this will be always a reference type.)

if t is an undeclared identifier, then

generate an error. (and possibly return t: int.)

Otherwise, t must have form $f(t_1, \ldots, t_n)$.

For each i, let $S_i :=$ conversions(typecheck(t_i)).

Try to find a function declaration $f'{:}T_1 \times \cdots \times T_n \rightarrow T$,
such that for each i, there is a type judgement of form
$t_i'{:}T_i \in S_i$, and f' has name f.

If no f' exists, then report 'could not find definition of f'.

If more than one f' exists, and no f' is strictly
more specific than all other possible functions, then
report 'ambiguous overload of f'.

Let $f' :=$ the most specific applicable function.

return $f'(t_1', \ldots, t_n'){:}T$.
(Note that we have replaced
$f(t_1, \ldots, t_n)$ by $f'(t_1', \ldots, t_n')$.

Type Checking Algorithm

setoftypejudgements conversions(typejudgement $t{:}T$)

{

setoftypejudgements $S := \{t{:}T\}$.

As long as there exist a type judgement $u{:}U \in S$, and a conversion function $c{:}U \rightarrow V$, s.t. either

no judgement with type V occurs in S, or

there is a judgement $v{:}V$ in S, but term v is more complex then $c(u)$, then

add $c(u){:}V$ to S (and possibly remove $v{:}V$ from S.)

If no more conversions can be added, then return S

}

Higher-Order Types

In the typechecking algorithm, we assumed that functions have types of form $T_1 \times \cdots \times T_n \rightarrow T$.

In reality, function types may contain universally quantified type variables:

$$\Pi\ U_1 \cdots U_m\ \ T_1 \times \cdots \times T_n \rightarrow T.$$

Such types are called higher-order types.

Before f can be applied, the variables $U_1, \ldots, U_m$ must be instantiated with concrete types.

Examples of higher-order types are:

$$*\colon \Pi X\ \mathrm{pointer}(X) \rightarrow \mathrm{ref}(X), \quad \&\colon \Pi X\ \mathrm{ref}(X) \rightarrow \mathrm{pointer}(X),$$

and

$$[\ ]\colon \Pi X\ \mathrm{pointer}(X) \times \mathrm{int} \rightarrow \mathrm{ref}(X).$$

Remarks

The typechecking algorithm requires that among the applicable functions, there is one that is most specific. This is how C^{++} is defined. The ADA language would be more liberal.

constness cannot be handled by conversion functions, because of a combinatorial explosion. I will come to this topic later.

In C^{++}, not all possible conversions would be added, because user defined conversions cannot be chained. The exact rules are quite complicated.

Let's try an example:

$f_A: A \times A \to A$

$f_B: B \times B \to B$

$f_C: C \times C \to C$

$\mathrm{conv}_{AB}: (A \to B)$

$\mathrm{conv}_{BC}: (B \to C)$

Check $f(f(a,b),c)$, $f(f(f(a,a),b),c)$

Typecheck `( i + j ) * 4.0` in C, with declarations

```
int i, j;
```

Typecheck `f(1,2)`, with declarations:

```
double f( int, double );
int f( double, int );
```

An example that ADA would accept, while C^{++} rejects it:

$f_A \colon A \to B$

$f_B \colon B \to A$

$a \colon A$

$a \colon B$

$p \colon A \to C$

Check $p(f(a))$, $p(f^2(a))$, or $p(f^3(a))$.

At the final stage, p will only accept A, so that in all terms of form $p(f^i(a)$, the types can be uniquely determined.

C^{++} would reject it, because it requires that every term has a fixed type, independent of its context.

(In ADA, typechecking can be viewed as running a non-deterministic tree automaton. Conversion functions are ϵ transitions.)

Most Specific Functions

In case, more than one function is applicable, the type checking algorithm (for C^{++}) selects the most specific. Suppose that we have functions:

```
int f( int, int );
int f( double, double );
```

and expression `f(1,2)`. In that case, the first version of `f` should be preferred, because it is the most specific.

If we would have

```
int f( int, double );
int f( double, int );
```

then neither of the versions of `f` is more specific than the other, so that the compiler cannot decide.

Most Specific Functions

Definition We write $T_1 \prec T_2$ if type T_1 is strictly more specific than T_2.

If $T_1 \prec T_2$, then T_1 is implictly convertible into T_2, but the converse need not be the case. For example, in C^{++}, the types **char** and **int** can be freely converted into each other, but at the same time, **char** $\prec$ **int**, and **int** $\not\prec$ **char**.

For function types, we define
$(S_1 \times \cdots \times S_n \rightarrow U) \prec (T_1 \times \cdots \times T_n \rightarrow V)$ if

1. For each i, either $S_i = T_i$ or $S_i \prec T_i$, and
2. there is one i, for which $S_i \prec T_i$.

If we have a set F of applicable functions, and there is one $f \in F$, s.t. for all $f' \neq f$, we have $f \prec f'$, then we can select f as most specific function.

Constness and Subtyping without Conversion

Consider:

```
int prod1( const int& x1, const int& x2 )
{
   return x1 * x2;
}

int prod2( int& x1, int& x2 )
{
   return x1 * x2;
}
```

```
int apply( int f( const int&, int& ), int a, int b)
{
   return f(a,b);
}

apply( prod1, 3, 4 ); // Should be forbidden.
apply( prod2, 4, 5 ); // Should be possible.
```

If one thinks about it, only the first should be forbidden, and the second should be fine. In practice, the compiler (gcc) forbids both. I don't know if this is the C^{++}-standard, or if the implementation is incomplete.

Subtyping without Conversions (Inheritance, Constness)

Let us write $A \sqsubseteq B$ if type A is a subtype of B, and no conversion is necessary.

This means that any function that requires a B, can be given an A without any conversion.

If A inherits from B, then pointer$(A) \sqsubseteq$ pointer(B), and ref$(A) \sqsubseteq$ ref(B).

In Java, one would also have $A \sqsubseteq B$, (because in Java, every Object is a pointer.)

For **const**, we have $A \sqsubseteq$ const(A).

This is a bit surprising, because const looks like an adjective, but in reality is a modifier.

Subtyping without Conversions (2)

In addition, one has the following rules:

If $A \sqsubseteq B$, then pointer$(A) \sqsubseteq$ pointer(B), and ref$(A) \sqsubseteq$ ref(B).

If $B_1 \sqsubseteq A_1, \ldots, B_m \sqsubseteq A_m$, and $A \sqsubseteq B$, then

$$A_1 \times \cdots \times A_m \to A \quad \sqsubseteq \quad B_1 \times \cdots \times B_m \to B.$$

(It seems that gcc doesn't apply this last rule. I am not sure what is the standard.)

Disclaimer: I am not completely certain if const and inheritance can be handled by the same relation $\sqsubseteq$. It seems to be the case.