Type Checking

Output of Parser

The parser either

1. evaluates expressions immediately.
2. generates executable code (usually in reverse Polish notation).
3. Generates an abstract syntax tree.

1,2 are possible only for very simple languages.

Polish Notation

Polish Notation is called after (the nationality of) Jan Łukasiewicz.

Consider the expression: `x = x + ( y + z ) * 4`.

In Polish notation: `= x + x * + y z 4`.

In reverse Polish notation (RPN): `x x y z + 4 * + =`.

Evaluating RPN with a stack machine:

Expression: `x x y z + 4 * + =`

```
push( x ); // Push x on the stack.
push( x ); // Push value of x on the stack.
push( y ); // Push value of y on the stack.
push( z ); // Push value of z on the stack.
eval( + ); // Take two values from top of stack,
           // add them, and push result back.
push( 4 ); // Push 4 on the stack.
eval( * );   // Take two values from top of stack,
  // multiply them, and push result back.
eval( + ); // Take two values from top of stack,
  // add them, and push result back.
assign;  // Take value from top of stack,
         // Take variable from top of stack,
         // and write value into variable.
```

Moral

Expressions have tree structure. They are evaluated in postfix order:

First recursively evaluate the arguments, then apply the top operator on the arguments.

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

```
   block(
      assign( sum, 0 ),
      for( declare( unsigned int, i, 0 ),
               < ( i, n ), pref++( i ),
         block( +=( sum, *( i, i )) ) )
   )
```

```
// Copy a string from from to to:

void strcpy( const char* from, char* to )
{
   while( *from )
      * ( to ++ ) = * ( from ++ );
}


funcdef( strcpy,  void,
   parlist( const_pointer( char ), from,
            pointer( char ), to ),
block(
   while( *( from ),
      =( *( postf++( to )), *( postf++( from )) ) )
)
```

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.

Name Look Up

Name lookup is non-trival:

- Local identifiers are introduced and removed in stack-like fashion.
- Identifiers can consist of multiple components (namespaces).
- Identifiers have to be looked up in different namespaces (ADT, fields of structures, **using** directives.)
- Identifiers can have incomplete definitions, which are completed later.
- All this has to be efficient.

Type Checking

Role of type checking:

1. Determine if the operators in the AST can be matched to operators in the intermediate representation.
2. If an operator can be mapped in more than one way, determine which one should be used.
3. Same for user defined functions: Check if called functions have a definition. If there is more than one definition, determine which one should be used.
4. Insert implicit conversions.

- For example, + has different definitions on **int**, **real**, **pointers**, **strings**, etc.

  A pointer (to any type) and a **int** can be added. Two pointers cannot be added.

- In C^{++} a user defined function (operator) can have different definition. The compiler has to be determine if (**(1)** there is fit, and if there is **(2)** a best fit.

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 convenient 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 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 must have 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

Structs can refer to themselves (either directly or through other structs.

A **struct** cannot contain another struct of its own type, but it may contain a **pointer** or **reference** to itself:

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

If there would be a type constructor $\mathbf{struct}_{E,N}(t_1, t_2)$, then one would need a fixed point operator to define

$$\text{list} = \text{fix } L \ \mathbf{struct}(\ \text{elem}\!:\mathbf{int},\ \text{next}\!:\mathbf{pointer}(L)\ ).$$

The situation is similar to functions: Functions cannot contain themselves but they can call themselves if you give them a name.

This, in combination with the fact that in most programming languages, **struct**-equivalence is intensional, suggest that recursion can be easily expressed by giving a name to the **struct**, without need of fixed point operator.

The definition of list creates the following typing rules:

$$\begin{array}{lcl} l\!:\text{list} & \Rightarrow & (l.\text{elem})\!:\text{int} \\ l\!:\text{list} & \Rightarrow & (l.\text{next})\!:\textbf{pointer}(\text{list})) \end{array}$$

Type Constructors

- If T is a type, then $\mathbf{array}(T)$ and $\mathbf{array}(n, T)$ are types.
- If T is a type, then $\mathbf{pntr}(T)$ and $\mathbf{ref}(T)$ are types.
- If T is a type, then $\mathbf{const}(T)$ is a type.

We assume that types of form $\mathbf{const}(\mathbf{const}(T))$, $\mathbf{ref}(\mathbf{ref}(T))$, $\mathbf{array}(n, \mathbf{const}(T))$, $\mathbf{array}(\mathbf{const}(T))$, and $\mathbf{const}(\mathbf{ref}(T))$ cannot occur.

Type Checking

A type checker in a compiler must do the following:

1. Decide, for every function or operatator that occurs in the AST, if there is a concrete function that fits. If more than one concrete function fits, decide which one will be used.

2. If necessary, insert conversions into the AST.

Type Checking (2)

Type checking can be done in two ways:

bottom-up: Given tree of form $f(t_1, \ldots, t_n)$, compute the possible types of each t_i. After that, check if there is an applicable f and compute the possible types for $f(t_1, \ldots, t_n)$.

top-down: Knowing the set of possible types that f should have, determine for each t_i, the set of types that t_i can have.

Theoretically, the results should be the same. In practice, one direction might lead to a combinatorial explosion, especially if implicit conversions are involved.

Type Checking (3)

C and Java use bottom-up type checking. Every term has a unique, main type. Other types can be obtained by conversions.

C^{++} mostly uses bottom-up, except for curly initializers, which are checked top down.

In some other languages (ADA), a term can have different, unrelated types. In that case, tree-automata techniques must be used.

Const

I try to define a type checking algorithm for C^{++} that is not totally crazy.

First define a helper predicate $\mathrm{CV}(T_1, T_2)$ which is true if T_1 can be 'converted' into T_2 by inserting **const**s or **volatile**s.

If $\mathrm{CV}(T_1, T_2)$ is true, then a T_1 can be bound to an argument that requires a T_2.

Const (2)

- $\mathrm{CV}(T, T)$.
- $\mathrm{CV}(\ \mathbf{ref}(T_1),\ \mathbf{ref}(\mathbf{const}(T_2))\ )$ if and only if $\mathrm{CV}(T_1, T_2)$.
- $\mathrm{CV}(\ \mathbf{ref}(\mathbf{const}(T_1)),\ \mathbf{ref}(\mathbf{const}(T_2))\ )$ if and only if $\mathrm{CV}(T_1, T_2)$.
- $\mathrm{CV}(\ \mathbf{func}(T_1; T_{1,1}, \ldots, T_{1,m}),\ \mathbf{func}(T_2; T_{2,1}, \ldots, T_{2,n})\ )$ if and only if $m = n$, $\mathrm{CV}(T_1, T_2)$ and for each i with $1 \leq i \leq n$, $\mathrm{CV}(T_{2,i}, T_{1,i})$.

Const (3)

Rules for **array** and **pntr**:

- CV($\mathbf{pntr}(T_1)$, $\mathbf{pntr}(\mathbf{const}(T_2))$) if and only if CV(T_1, T_2).
- CV($\mathbf{pntr}(\mathbf{const}(T_1))$, $\mathbf{pntr}(\mathbf{const}(T_2))$) if and only if CV(T_1, T_2).
- CV($\mathbf{array}(n_1, T_1)$, $\mathbf{array}(n_2, T_2)$) if and only if $n_1 = n_2$ and CV(T_1, T_2).
- CV($\mathbf{array}(T_1)$, $\mathbf{array}(T_2)$) if and only if CV(T_1, T_2).

Const (4)

Const is trickier than it seems at first:

```
int a = 2015;
int* p1 = &a;
const int *p2 = p1;
```

This is OK, we have CV(**pntr**(**int**), **pntr**(**const**(**int**))).

The reverse should be forbidden:

```
const int a = 2015;
const int* p1 = &a;
int *p2 = p1;        // Now we can change a;
```

Indeed, not CV(**pntr**(**const**(**int**)), **pntr**(**int**)).

Const (5)

Surprisingly, we also don't have:

CV(**pntr**(**pntr**(**int**)), **pntr**(**pntr**(**const**(**int**)))).

```
const int a = 2015;
int* p = nullptr;

int** p1 = &p;
const int** p2 = &p;  // Is forbidden!

*p2 = &a;
**p1 = 2016;    // changes a!
```

We do have:

CV(**pntr**(**pntr**(**int**)), **pntr**(**const**(**pntr**(**const**(**int**))))).

Conversions

We assume a set $\mathcal{C}$ of conversion costs. The cost can be natural numbers or values of a finite set.

We assume that cost can be compared: $c_1 \sqsubset c_2$ if c_1 is stricly cheaper (better) than c_2.

If conversion from T_1 to T_2 is possible, then $\mathbf{conv}(T_1, T_2)$ returns a pair (g, c), consisting of a conversion function g and the cost of the conversion.

If no conversion is possible, $\mathbf{conv}(T_1, T_2)$ returns $\perp$.

Conversions (2)

- If $\mathrm{CV}(T_1, T_2)$ then $\mathbf{conv}(T_1, T_2) = (\mathbf{id}, c)$, where $c = 1A$ if $T_1 = T_2$, and $c =$ 1B otherwise.
- $\mathbf{conv}(\ T,\ \mathbf{const}(\mathbf{ref}(T))\ ) = (\mathbf{box}_T, 1A)$. (Needed for temporaries.)
- In C/C^{++}, array references are implicitly transformed into pointers:

  $\mathbf{conv}(\ \mathbf{ref}(\mathbf{array}(n, T)),\ \mathbf{pntr}(T)\ ) = (\mathbf{array2pntr}_T,\ \text{1A})$,
  $\mathbf{conv}(\ \mathbf{ref}(\mathbf{const}(\mathbf{array}(n, T))),\ \mathbf{pntr}(\mathbf{const}(T))\ ) =$
  $(\mathbf{array2pntr}_T,\ \text{1A})$, and
  $\mathbf{conv}(\ \mathbf{ref}(\mathbf{array}(n, T)),\ \mathbf{pntr}(\mathbf{const}(T))\ ) =$
  $(\mathbf{array2pntr}_T,\ \text{1B})$.

  The conversion function $\mathbf{array2pntr}_T$ does nothing. (The function does nothing.)

  The same conversions apply to $\mathbf{array}(T)$.

Conversions (3)

- If type T is a primitive type, then $\mathbf{conv}(\mathbf{ref}(\mathbf{const}(T)),\ T) = \mathbf{conv}(\mathbf{ref}(T),\ T) = (\mathbf{load}_T, 1A)$.
- If type T is non-primitive, but has a copy constructor $\mathbf{copy}_T$ with argument type $U = \mathbf{ref}(T)$ or $\mathbf{ref}(\mathbf{const}(T))$, and $\mathrm{CV}(U', U)$, then $\mathbf{conv}(U', T) = (\mathbf{copy}_T, 1\mathrm{C})$.

Disclaimer: These rules are probably too simple. Also, there is no mention of rvalue references.

Diclaimer 2: It appears that **conv** ignores top level **const**, because **const** makes only sense in the scope of **pntr** or **ref**.

Conversions (3)

- Assume that T_1 has form U_1, $\mathbf{const}(U_1)$, $\mathbf{ref}(U_1)$, or $\mathbf{ref}(\mathbf{const}(U_1))$, with U_1 a primitive type.

  Assume that T_2 has form U_2 or $\mathbf{const}(U_2)$, with U_2 also a primitive type.

  - If there exists no conversion from U_1 to U_2, then $\mathbf{conv}(T_1, T_2) = \perp$.
  - Otherwise, let $\mathbf{conv}_{U_1,U_2}$ be the conversion operator. If the conversion is guaranteed to be non-lossy, then set $c = 2$, otherwise set $c = 3$.
  - If T_1 contains **ref**, then $\mathbf{conv}(T_1, T_2) = (\mathbf{conv}_{U_1,U_2}(\mathbf{load}_{U_1}(\,\cdot\,)),\ c)$. Otherwise, $\mathbf{conv}(T_1, T_2) = (\mathbf{conv}_{U_1,U_2},\ c)$.

Conversions (4)

- If there exists a user defined conversion operator $\mathbf{op}_{U_1,U_2}$, which converts U_1 to U_2, and T_1 can be converted to U_1 with level $\sqsubseteq 3$, and U_2 can be converted to T_2 with level $\sqsubseteq 3$, then

$$\mathbf{conv}(T_1, T_2) = (\ g_2(\mathbf{op}_{U_1,U_2}(g_1(\ \cdot\ ))),\ 4\ ),$$

where g_1 is obtained from $\text{conv}(T_1, U_1) = (g_1, c_1)$, and g_2 is obtained from $\text{conv}(U_2, T_2) = (g_2, c_2)$.

C^{++}-rules, taken from C^{++} lecture

C^{++} distinguishes levels of conversions:

Level 1A The conversion from T_1 to T_2 has level 1A if $T_1 = T_2$ (no conversion) or it involves conversion from arrays or functions to pointers (which is unavoidable because nothing else can be done with them).

Level 1B The level from T_1 to T_2 is level 1B if consists of a level 1A conversion, followed by possible insertions of **const**.

Level 1C Application of a a copy constructor.

Level 2 The conversion from T_1 to T_2 has level 2, if both T_1, T_2 are integral. (**bool, char, int, short, long, unsigned**), and the conversions from T_1 to T_2 is guaranteed to be without loss. The conversion from **float** to **double** is also level 2.

Level 3 The conversion from T_1 to T_2 has level 3, if both T_1, T_2 are integral, but the conversion from T_1 to T_2 is possibly lossy. (For example from **int** to **char**, from **unsigned int** to **int**, or from **int** to **double**.

Also conversions from a derived class to a base class are level 3.

Level 4 The conversion from T_1 to T_2 is level 4 if it involves a user defined conversion. (A one argument constructor.)

Type Checking Algorithm

Function **maintype**(t) returns a pair (t', T), where t' is obtained from t by possibly inserting type conversions, and T is the main type of t'.

Type Checking Algorithm

tree $\times$ type **maintype**(term t)

- if t is a constant (number, bool, char, etc.), then return (t, T), where T is the given type of t.
- if t is an identifier and t is declared with type T, then return $(\ t, \mathbf{ref}(T)\ )$.

  (In C/C^{++}, variables always have reference type.)
- if t is an identifier without declaration, then generate an error, and possibly return $(t, \mathbf{int})$.

- Otherwise, t has form $f(t_1, \ldots, t_n)$.

  For each i, recursively let $(t'_i, T_i) = \textbf{maintype}(t_i)$.

  Let $\mathcal{F}_1$ be the set of functions that can be overloadings of f, according to the language definition. If $\mathcal{F}_1$ is empty, create an error message.

  Set $\mathcal{F}_2 = \emptyset$. For each $\hat{f} \in \mathcal{F}_1$, let $\textbf{func}(U;\ U_1, \ldots, U_n)$ be the type of $\hat{f}$. Check (by calling **conv**) that for each i $(1 \leq i \leq n)$, the type T_i is convertible to U_i. If yes, then add $\hat{f}$ to $\mathcal{F}_2$.

  If the resulting set $\mathcal{F}_2$ is empty, create an error message.

  If $\mathcal{F}_2$ contains more than one function $\hat{f}$, check, using the overload resolution rules, if a unique $\hat{f}$ can be selected. If yes, return $(\ \hat{f}(g_1(t'_1), \ldots, g_n(t'_n)),\ U\ )$, where the g_i are the conversions obtained by calling $\textbf{conv}(T_i, U_i)$.

  If no unique $\hat{f}$ exists, generate an error 'ambiguous overload of f'.

Field Selection Functions

We assume that terms of form $t.f$, where f is a field of a **struct**, are replaced (by the parser) by terms of form $\mathbf{sel}_f(t)$, where $\mathbf{sel}_f$ is a field selection function.

Every **struct** S that has a field with name f and type F, must provide overloads of $\mathbf{sel}_f$ with types $\mathbf{func}(\mathbf{ref}(F); \mathbf{ref}(S))$, and $\mathbf{func}(\mathbf{ref}(\mathbf{const}(F)); \mathbf{ref}(\mathbf{const}(S)))$.

For very small structs, one can also provide $\mathbf{func}(F; S)$.

Essential Functions

For C/C^{++}, one needs the following functions:

- For every primitive type T, an operator $\mathbf{load}_T$ of type $\mathbf{func}(T;\mathbf{ref}(\mathbf{const}(T)))$.

  Non-primitive types can have copy constructors with similar type.

- For every primitive type T, an assignment operator $\mathbf{assign}_T$ of type $\mathbf{func}(T;\mathbf{ref}(T),T)$.

Essential Functions (2)

- For every type T, an operator $*$ with types $\mathbf{func}(\mathbf{ref}(T); \mathbf{pntr}(T))$ and $\mathbf{func}(\mathbf{ref}(\mathbf{const}(T)); \mathbf{pntr}(\mathbf{const}(T)))$.
- For every type T, an operator & with types $\mathbf{func}(\mathbf{pntr}(T); \mathbf{ref}(T))$ and $\mathbf{func}(\mathbf{pntr}(\mathbf{const}(T)); \mathbf{ref}(\mathbf{const}(T)))$.
- For every type T, a conversion function $\mathbf{array2pntr}_T$ with types

$$\mathbf{func}(\mathbf{pntr}(T);\ \mathbf{ref}(\mathbf{array}(n, T)))$$
$$\mathbf{func}(\mathbf{pntr}(\mathbf{const}(T));\ \mathbf{ref}(\mathbf{const}(\mathbf{array}(n, T))))$$
$$\mathbf{func}(\mathbf{pntr}(\mathbf{const}(T));\ \mathbf{ref}(\mathbf{array}(n, T)))$$

(Same types for $\mathbf{array}(T)$)

Essential Functions (3)

- For every type T, and every type I that can be used for indexing, one needs an indexing operator $\mathbf{index}_T$ of type $\mathbf{func}(\mathbf{ref}(T); \mathbf{pntr}(T), I)$ and $\mathbf{func}(\mathbf{ref}(\mathbf{const}(T)); \mathbf{pntr}(\mathbf{const}(T)), I)$.
- For every type T, an operator $\mathbf{box}_T$ of type $\mathbf{func}(\mathbf{ref}(\mathbf{const}(T)); T)$.

  (**box**-operators are used for temporaries.)
- For every integral type T, and for every pointer type T, postincrement and postdecrement operators of type $\mathbf{func}(T; \mathbf{ref}(T))$.
- For every integral type T, and for every pointer type T, preincrement and predecrement operators of type $\mathbf{func}(\mathbf{ref}(T); \mathbf{ref}(T))$.

Essential Functions (4)

- Conversion operators, arithmetic operators.
- If a class T has a constructor with arguments $T_1, \ldots, T_n$, it is viewed as a function with type

  $$\mathbf{func}(T;\ T_1, \ldots, T_n).$$

  (This also applies to copy-constructor, if there is one.)
- A member function of class T with arguments $T_1, \ldots, T_n$ and return value U, that is not a constructor, has type

  $$\mathbf{func}(U;\ \mathbf{ref}(T), T_1, \ldots, T_n).$$

  Even when the object is treated as pointer inside the member function, to the outside it behaves as a reference.

  If the member function is **const**, then $\mathbf{ref}(T)$ must be replaced by $\mathbf{ref}(\mathbf{const}(T))$.

Higher-Order Types

In maintype, it was assumed that functions have types of form $\mathbf{func}(T; T_1, \ldots, T_n)$.

In reality, one often writes, 'For every type T, we have an operator of type ... This means that we need higher-order types:

$$\Pi \, U_1 \cdots U_m \;\; \mathbf{func}(T; T_1, \ldots, T_n).$$

Before f can be applied, the variables $U_1, \ldots, U_m$ must be instantiated with concrete types, (either by unification or in some other, creative way).

Higher-Order Types (2)

Examples of higher-order types (taken from C) are

$$
\begin{aligned}
&*{:}\,\Pi X\ \mathbf{func}(\ \mathbf{ref}(X);\ \mathbf{pntr}(X)\ ),\\
&*{:}\,\Pi X\ \mathbf{func}(\ \mathbf{ref}(\mathbf{const}(X));\ \mathbf{pntr}(\mathbf{const}(X))\ ),\\
&\mathbf{index}_X{:}\,\Pi X\ \mathbf{func}(\ \mathbf{ref}(X); \mathbf{pntr}(X), \mathbf{int}\ ).
\end{aligned}
$$

Most Specific Functions

Overloading in C^{++} is resolved by selecting the most specific function, which must be unique.

Suppose that we have functions:

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

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

If we have

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

then neither of the versions of `f` is more specific than the other.

Curly Initializer Lists

In order to handle Curly initializer lists, **conv** has to be extended quite a lot.

Conclusion

The challenge of teaching compiler construction is to find a reasonable compromise between saying nothing meaningful at all, or saying only ugly things.