Type Checking and Semantic Analysis

Copyright and AI Notice

These slides are intended for studying and teaching at Nazarbayev University in Astana, Kazakhstan.

If you want to redistribute or adapt these slides for different purposes, check the license first.

No generative AI was used in the preparation of these slides.

Contents

In this set of slides we explain semantic analysis. Semantic analysis involves looking up variable names, type checking, selecting overloads, and inserting implicit conversions.

We first give general principles and options. After that we give an example procedure, that is close to C.

These slides are not easy, but fundamental understanding of semantic analysis is necessary for understanding realistic compilers, which are at the core of every program.

Name Look-Up

One has to keep track of the following problems:

- Scopes of local identifiers are introduced and removed in stack-like fashion.
- Identifiers can use namespaces or package names. In that case, one has to look in the proper environment.
- Identifiers can have incomplete definitions, which are completed later.
- All of this has to be efficient.

Untyped Languages?

In assembly language, there exist no typed data. All data are just bytes. Types are built-in in the instructions. The instruction use the data, assuming that they have certain types.

This means that the programmer has to remember how much space data needs, and which are the proper instructions that can be applied. There is no protection against applying instructions on unintended data.

One cannot say that assembly language is untyped. Correctly said, instructions have built-in types, the types exist in the mind of the programmer, and it is the task of the programmer to make sure that the typed instructions are applied consistently on the same group of data.

Type Checking

We decide that we will help the programmer a bit by adding the types to the language. Now we can:

- decide automatically how much place a variable needs in memory.
- know which operations can be applied on a given piece of data, and reject operations that cannot be part of a correct algorithm.
- allow the programmer to write operations in imprecise fashion. We replace them by exact operations that fit to the type of the data.

We first explain general principles of type checking. After that, we introduce a concrete type system and a checking procedure, which are based on C (and a little bit on C^{++})

Static vs. Dynamic Type Checking

- Static type checking means that types are checked (and imprecise names are resolved) before the program is executed. It has the advantage that the programmer gets immediate feed back for the complete program, and no further checks are needed at run time. Static type checking is used by Java, modern C, and C^{++}
- Dynamic type checking means that types are checked (and imprecise names are resolved) at run time. The main advantage is that it offers flexibility. Sometimes, it is impossible to know in advance the type of the data. Disadvantage is that it slows down execution speed.

  Dynamic type checking is used by Python, Perl and Object-Oriented languages.

The third option, including types in instructions, and leaving correct application to the programmer, is used only in assembly language, and it was used by the first versions of C.

In order to implement dynamic type checking, one needs to add type information to the data. This information has to be checked at runtime, when the exact operation is selected.

Java and C^{++} mostly use static type checking, but allow dynamic type checking for calling virtual functions in object hierarchies.

In case more than one function (or operator) is applicable, we have to decide which one to use. If it is done at compile time, it is called overload resolution. If it is done at run time, it is called dynamic dispatch.

C uses overload resolution on built-in operators, but not on user functions.

C^{++} uses overload resolution both on built-in operators and on user functions. The rules on user functions are pretty complicated, and the result of three years of hard work.

I think that Java copied the rules from C^{++} mostly.

Type Checking: Bottom Up vs Top Down

Type checking can be done in two ways:

bottom-up: Given an AST of form $f(t_1, \ldots, t_n)$, first obtain the type of each t_i. After that, check if there is an applicable f and use the return type of f for $f(t_1, \ldots, t_n)$.

top-down: Look up the possible definitions of f, check which types $t_1, \ldots, t_n$ must have, and try to coerce $t_1, \ldots, t_n$ into the needed types.

Bottom Up Type Checking

C, C^{++} and Java use bottom-up type checking.

C^{++} uses bottom-up checking, which the exception of curly initializers, which are checked top down. I think it was a bad idea.

Ada uses a mixture of bottom-up and top-down type checking. According to Stroustrup, some members of the Ada standard committee consider the Ada rules a failure.

Intensional vs. Extensional Type Equivalence

The question is: Does one consider **structs** (or records or classes) with the same fields equal?

For example:

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


struct type2
   { int a1; int a2; };
```

Do we allow assignments between `type1` and `type2`, or parameter passing?

Two **structs** are extensionally equal if they have the same field names in the same order, and the types of these fields are either exactly equal or extensionally equal.

Intensional Equality

Most modern programming languages, e.g. C, C^{++} and Java, do not use extensional equality.

Instead, **structs** are always defined together with a name, and only the name matters. This is called intensional type equivalence.

Extensional type equivalence is not compatible with information hiding, and the user should be allowed to introduce different types for the same data when it is used to represent different things. (For example, speed, position and acceleration are all represented by three **double**s but should have different types.)

Intensional equality is easier for us, compiler-builders, because checking equality of names it is easier than comparing structure.

Type Checking for C

We will explain how type checking works for C.

C uses static type checking, ASTs are checked bottom up. During type checking, some implicit conversions are inserted.

We give examples on the next slide.

Some Examples in *C*

```
int a = 4;
double b = a + 4;
   // + has definition on int,int, which returns an int.
   // The result has to be converted to double.

double c = a + b;
   // + has a definition on double,double. In order to
   // use it, a must be converted to double.
if(c) c = 1.0;
   // c (which is double) is implicitly converted to bool.
```

A Concrete Type System

We introduce a type system that should be good enough for most of C and also for C^{++}.

We assume the following primitive types: **bool**, **char**, **int**, **unsigned int**, and **double**.

In addition, we assume the following type constructors:

- If T is a type, $n \geq 0$ is an unsigned integer constant, then **array**(n, T) is a type.
- If T is a type, then **pntr**(T) is a type.

(We won't worry about `size_t` in this course.)

Struct/Class

Defining **struct** (or **record** or **class**) types is a bit harder than it seems:

First attempt: If $T_1, \ldots, T_n$ are types, $v_1, \ldots, v_n$ are field names, then $(v_1\colon T_1, \ldots, v_n\colon T_n)$ is a type.

It would work for simple structs, like (re: **double**, im: **double**), it does not work for structs with forward or self reference.

Non-Recursiveness of Struct

The types of fields of structs may contain pointers to types that are not yet defined:

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

Of course, the definition

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

is out of question.

A Type Container for Struct

We assume that we have a type container that stores names of **structs**.

We add the rule: If v is an identifier that has a **struct** definition in the type container, then v is a type.

We can put the definition list := (elem: **int**, next: **pntr**(**list**)) in the container.

This also solves the extensionality problem, we just compare the identifiers.

Distinction between Lvalue and Rvalue

In order to understand how ASTs are typed in C, one needs to understand the distinction between lvalue and rvalue.

An lvalue is something that can be assigned to, an rvalue is just a value.

An lvalue represents a reference to memory. In the register/stack machine, lvalues will become pointers.

In the expression `x = a.f ++`, variable `a` is an lvalue, `a.f` is still an lvalue, `a.f ++` is an rvalue, `x` is an lvalue, and the complete expression `x = a.f ++` is an lvalue, because assignment in C returns the address of the assigned variable.

Type Checking: Some Notation

During typechecking, we will attach types to the nodes in the AST. Unfortunately, the resulting AST is not easy to write down, because the types can be complex. The easiest way would be to write the types in the tree. For example $+(a, 3.0)$ after type checking becomes

$$+_{\mathbf{double,rval}}(\ \mathbf{load}_{\mathbf{double,rval}}(\ a_{\mathbf{double,lval}}\ ),\ 3.0_{\mathbf{double,rval}}\ ).$$

This is pretty unreadable. Instead of writing the types, we assume that they are hidden in the nodes, and that we can get them by field acces. We write t.type for the type of the top node of t, and t.lr for the lr-value of t. If $t = +(\mathbf{load}(a), 3.0)$, then t.type $= \mathbf{double}$, and t.lr $= \mathbf{rval}$.

In an unchecked AST, the fields are unitialized.

Inserting Loads: **makerval**

We need a few helper functions:

Function **makerval**(t) ensures that AST t has an rval-type. It can be implemented as follows:

AST **makerval**(AST t) :

- If t.lr = **lval**, then construct t' = **load**(t). Set t'.type = t.type and set t'.lr = **rval**. Return t'.
- Otherwise, return t.

Example:

Conversion Penalties

In general, overload resolution is implemented by putting penalties on conversions. If the types are exactly equal, the penalty is 0. If conversion is not possible, the penalty is $+\infty$.

In general, conversions that are guaranteed to succeed without information loss, should be preferred over conversions that may lose information.

For example, converting `char` to `int` is guaranteed to have no information loss, while converting `int` to `char` may result in losing information.

C^{++} uses conversion levels, which need a lecture in itself to explain. We will introduce a workable simplification on the next slide.

Penalties of Type Conversions

We introduce a penalty function, which is a bit improvised.

There are three types of conversions

1. between primitive types,
2. from arrays to pointers, and
3. between pointer types.

In order to attach penalties to conversions between primitive types, we use the following sequence:

bool, **char**, **integer**, **unsigned**, **double**.

Converting to the right in this sequence has a penalty of 1 per step. Converting to the left has a penalty of 10 per step.

Penalties of Type Conversions (2)

We allow converting **pntr**(**void**) to **pntr**(T) with a penalty of 1, if T is not void.

C also allows conversion between pointer and **unsigned**, but we will not allow that.

We write **penalty**(T_1, T_2) for the penalty of converting T_1 to T_2.

We have
penalty(**bool**, **integer**) $= 2$, **penalty**(**double**, **integer**) $= 20$,
and and **penalty**(**pntr**(**void**), **pntr**(**int**)) $= 1$.

Conversion Function

Function $\mathbf{convert}_T(t)$ converts t into type T. This function should only be called only on types for which the conversion is possible.

AST $\text{convert}_T(\text{ AST } t\ )$:

- If t.type $\neq T$, then construct $t' = \mathbf{conv}(t)$. Set t'.type $= T$, and set t'.lr $=$ **rval**.
- Otherwise, return t.

In our setting, conversions are only possible on rvalues. This is not the case for C^{++}.

Converting Arrays to Pointers

In C, lvalue arrays are implicitly converted to rvalue pointers. In order to do this, we define the function **checkdecay**(t) :

- If t.type has form **array**(n, T) and t.lr = **lval**, then construct $t' =$ **decay**(t). Set t'.type = **pntr**(T), and set t'.lr = **rval**. Return t'.
- Otherwise, return t.

Function **checkdecay** must be applied on every tree that might be an lvalue array.

These are variables, lvalue field selections, and pointer accesses `*`.

Typecheck Function

On the slides that follow, we define a function **check** that recursively walks through the AST, and assigns types and lr-values to the nodes. It inserts **load**s and **conv**s where needed.

It is fairly close to C, but not exactly the same.

We assume that the parser replaces field access `t.f` by $\mathbf{select}_f(t)$, and `t -> f` by $\mathbf{select}_f(\star(t))$.

We also assume that the parser replaces array access `p[t]` by $\star(+(p,t))$.

We assume that `p ++` is replaced by $\mathbf{xpp}(p)$, and `++ p` by $\mathbf{ppx}(p)$. Similarly, `p --` and `-- p` are replaced by $\mathbf{xmm}(p)$ and $\mathbf{mmx}(p)$.

Checking Variables

If t is a variable v, then look up the declaration of v in the variable store. Generate an error if v is not found. Otherwise, let T be the type of v in the variable store.

Set t.type $= T$ and set t.lr $=$ **lval**. Return **checkdecay**(t).

Checking Constants

If t is a constant c, then the tokenizer defined a type T for c.

Set $t.\text{type} = T$, $t.\text{lr} = \textbf{rval}$. Return t.

String Constants

If t is a quoted character constant, (for example `"hello, world"`), then let $T = \textbf{array}(n, \textbf{char})$, where n is the length of the string constant. Assign $t.\text{type} = T$, and assign $u.\text{lr} = \textbf{lval}$. Return $\textbf{checkdecay}(t)$.

Note that, because T is an array, **decay** will convert it into $\textbf{pntr}(\textbf{char}) \,/\, \textbf{rval}$

In the real C-language, the **char** would be **const**, but we are not considering constness here.

Checking Built-In Binary Operators

If t has form $\mathbf{op}(t_1, t_2)$ with **op** among
`+,-,*,/,<,>,<=,>=,!=,==,` $\ldots$, then start by recursively calling

$$u_1 = \mathbf{makerval}(\ \mathbf{check}(t_1)\ ), \quad u_2 = \mathbf{makerval}(\ \mathbf{check}(t_2)\ ).$$

We assume that **op** has a set of exact definitions with signatures $U_1(T_1, T_1), \ldots, U_k(T_k, T_k)$. ($U_j(T_j, T_j)$ means that the exact operator expects two arguments of type T_j, and returns a value of type U_j.)

Find a j for which

$$\mathbf{penalty}(\ u_1.\text{type},\ T_j\ ) + \mathbf{penalty}(\ u_2.\text{type},\ T_j\ )$$

is minimal and not $+\infty$.

Construct AST $u = \mathbf{op}(\ \mathbf{convert}_{T_j}(u_1),\ \mathbf{convert}_{T_j}(u_2)\ )$. Assign $u.\text{type} = U_j$, and $u.\text{lr} = \mathbf{rval}$. Return u.

For example for ==, the set of overloads must be something like:

$\mathbf{bool}(\mathbf{bool}, \mathbf{bool})$, $\mathbf{bool}(\mathbf{char}, \mathbf{char})$, $\mathbf{bool}(\mathbf{integer}, \mathbf{integer})$,
$\mathbf{bool}(\mathbf{unsigned}, \mathbf{unsigned})$, $\mathbf{bool}(\mathbf{double}, \mathbf{double})$,
$\mathbf{bool}(\mathbf{pntr}(T), \mathbf{pntr}(T))$.

For +, the set of overloads is probably:

$\mathbf{bool}(\mathbf{bool}, \mathbf{bool})$, $\mathbf{char}(\mathbf{char}, \mathbf{char})$, $\mathbf{integer}(\mathbf{integer}, \mathbf{integer})$,
$\mathbf{unsigned}(\mathbf{unsigned}, \mathbf{unsigned})$, $\mathbf{double}(\mathbf{double}, \mathbf{double})$,
$\mathbf{pntr}(T)(\mathbf{pntr}(T), \mathbf{integer})$, $\mathbf{pntr}(T)(\mathbf{pntr}(T), \mathbf{unsigned})$.

Checking Simple Assignment

If t has form $=(t_1, t_2)$, then start by computing

$$u_1 = \textbf{check}(t_1), \quad u_2 = \textbf{makervalue}(\textbf{check}(t_2)).$$

If u_1.lr is not **lval**, then generate an error message.

If $\textbf{penalty}(u_1.\text{type}, u_2.\text{type}) = \infty$, then also generate an error message.

Otherwise, let $T = u_1.\text{type}$. Construct u is $=(\ u_1,\ \textbf{convert}_T(u_2)\ )$. Set $u.\text{type} = T$, and $u.\text{lr} =$ **lval**. Return u.

Checking Updating Assignment

Checking updating assignment operators is not really different from checking binary operators. The difference is that the left argument must be **lval** and cannot be converted.

If t has form $\mathbf{op}(t_1, t_2)$ with **op** among `+=`, `-=`, `*=`, `/=`, then first compute

$$u_1 = \mathbf{check}(t_1), \qquad u_2 = \mathbf{makerval}(\ \mathbf{check}(t_2)\ ).$$

If u_1.lr is not **lval**, then generate an error message.

Assume that **op** has a set of exact definitions of form $U_1(T_1, T_1), \ldots, U_k(T_k, T_k)$.

Find a j for which $T_j = u_1$.type, and $\mathbf{penalty}(u_2.\text{type}, T_j)$ is minimal and not $+\infty$.

Construct $u = \mathbf{op}(u_1, \mathbf{convert}_{T_j}(u_2)\ )$. Assign u.type $= U_j$, and assign u.lr = **lval**. Return u.

Checking Field Selection

If t has form $\mathbf{select}_f(t_1)$, then first recursively compute $u_1 = \mathbf{check}(t_1)$.

If u_1.type is not an identifier, or the identifier does not have a struct definition in the type container, then generate an error message.

If the definition of u_1.type does not have a field f, generate an error message.

Let T be the defined type of field f.

Assign t.type $= T$, and assign t.lr $= u_1$.lr.

Return $\mathbf{checkdecay}(t)$.

Function $\mathbf{checkdecay}(t)$ inserts a **decay** operation if t.type has form $\mathbf{array}(n, T)$, and t.lr $= \mathbf{lval}$.

Checking Unary $\star$-operator

The $\star$-operator converts a $\mathbf{pntr}(T)$ as rvalue into T as lvalue.

If t has form $\star t_1$, then recursively compute
$u_1 = \mathbf{makervalue}(\ \mathbf{check}(t_1)\ )$.

If u_1.type is not of form $\mathbf{pntr}(T)$, then generate an error message.

Otherwise, construct $u = \mathbf{conv}(u_1)$, set u.type $= T$ and set
u.lr $= \mathbf{lval}$.

Return $\mathbf{checkdecay}(u)$.

Checking Unary &-operator

The &-operator converts a type T as lvalue into $\mathbf{pntr}(T)$ as rvalue:

If t has form $\&t_1$, then recursively compute $u_1 = \mathbf{check}(t_1)$.

If u_1.lr is not **lval**, then generate an error message.

Otherwise, construct $u = \mathbf{conv}(u_1)$, set u.type $= \mathbf{pntr}(T)$ and set u.lr $=$ **rval**. Return u.

Increasement/Decreasement Operators

If t has form **op**(t) with **op** among **xpp**, **xmm**, **ppx**, **mmx**, start by computing $u_1 =$ **check**(t_1).

If u_1.lr is not **lval**, then generate an error message.

If u_1.type is not of form
bool, **char**, **int**, **unsigned**, **double**, **pntr**(X), then generate an error message.

Assign t.type $= u_1$.type. If op $=$ **xpp** or **xmm**, then assign t.lr $=$ **rval**. In the other two cases, assign t.lr $=$ **lval**.

(This is because **xpp** and **ppx** return a copy of the previous value of u_1)

Checking the Ternary Conditional Operator

If t has form ?(t, t_1 t_2), then recursively call

$$u = \textbf{makerval}(\,\textbf{check}(t)\,), \quad u_1 = \textbf{check}(t_1), \quad u_2 = \textbf{check}(t_2).$$

If $\textbf{penalty}(u.\text{type}, \textbf{bool}) = \infty$, then generate an error message.

Otherwise, set $u = \textbf{convert}_{\textbf{bool}}(u)$. If $\textbf{penalty}(\,u_1.\text{type},\, u_2.\text{type}\,) < \textbf{penalty}(\,u_2.\text{type},\, u_1.\text{type}\,)$, then set $t.\text{type} = u_2.\text{type}$. Otherwise, set $t.\text{type} = u_1.\text{type}$.

If either $u_1.\text{lr} = \textbf{rval}$ or $u_2.\text{lr} = \textbf{rval}$, set $u_1 = \textbf{makerval}(u_1)$, and set $u_2 = \textbf{makerval}(u_2)$. Set $t.\text{lr} = \textbf{rval}$. Otherwise, set $t.\text{lr} = \textbf{lval}$.

Return t.

Checking a User Defined Function

If t has form $f(t_1, \ldots, t_n)$, with f an identifier, then recursively compute

$$u_1 = \textbf{makerval}(\ \textbf{check}(t_1)\ ), \ldots, u_n = \textbf{makerval}(\ \textbf{check}(t_n)\ ).$$

C has no overloading on user defined functions, so that there exist either no, or one function declaration with name f. If no function exists, then generate an error.

Otherwise, assume that f is declared with type $U(T_1, \ldots, T_n)$. Check that for all u_i, $\quad \textbf{penalty}(u_i.\text{type}, T_i) \neq +\infty$.

Construct $u = f(\ \textbf{convert}_{T_1}(u_1), \ldots, \textbf{convert}_{T_n}(u_n)\ )$. Assign $u.\text{type} = U$, and assign $u.\text{lr} = \textbf{rval}$. Return u.

Examples

Consider the example from the beginning: `c = a + b`. The AST is $=(c, +(a, b))$. The declarations are `int a; double b; double c`.
Calling **check**($=(c, +(a, b))$) results in
$=(c, +(\mathbf{conv}(\mathbf{load}(a)), \mathbf{load}(b)))$.

The types and lr-values of the nodes are on the next slide.

The types and lr-values are as follows:

part of AST	type	lr
c	**double**	**lval**
a	**int**	**lval**
$\mathbf{load}(a)$	**int**	**rval**
$\mathbf{conv}(\mathbf{load}(a))$	**double**	**rval**
b	**double**	**lval**
$\mathbf{load}(b)$	**double**	**rval**
$+(\mathbf{conv}(\mathbf{load}(a)), \mathbf{load}(b))$	**double**	**rval**
$=(c, +(\mathbf{conv}(\mathbf{load}(a)), \mathbf{load}(b)))$	**double**	**lval**

Examples (2)

Consider the expression `x = a.f ++`, using declarations `double x; struct aaaa { int f; int g; } a`. The AST equals $=(\ x,\ \mathbf{xpp}(\mathbf{select}_f(a))\ )$. The result of **check** equals $=(\ x, \mathbf{conv}(\mathbf{xpp}(\mathbf{select}_f(a)))\ )$. The types and lr-values are as follows:

part of AST	type	lr
x	**double**	**lval**
a	aaaa	**lval**
$\mathbf{select}_f(a)$	**int**	**lval**
$\mathbf{xpp}(\mathbf{select}_f(a))$	**int**	**rval**
$\mathbf{conv}(\mathbf{xpp}(\mathbf{select}_f(a)))$	**double**	**rval**
$=(\ x,\ \mathbf{conv}(\mathbf{xpp}(\mathbf{select}_f(a)))\ )$	**double**	**lval**

Examples (3)

Consider `*p ++ = *q++`, assuming `char* p; char* q;`

The AST t equals $=(*(\mathbf{xpp}(p)), *(\mathbf{xpp}(q))\ )$. **check** returns $=(\ \mathbf{conv}(\mathbf{xpp}(p)),\ \mathbf{load}(\mathbf{conv}(\mathbf{xpp}(q)))\ )$, with

part of AST	type	lr
p	**pntr(char)**	**lval**
$\mathbf{xpp}(p)$	**pntr(char)**	**rval**
$\mathbf{conv}(\mathbf{xpp}(p))$	**char**	**lval**
q	**pntr(char)**	**lval**
$\mathbf{xpp}(q)$	**pntr(char)**	**rval**
$\mathbf{conv}(\mathbf{xpp}(q))$	**char**	**lval**
$\mathbf{load}(\mathbf{conv}(\mathbf{xpp}(q))$	**char**	**rval**
$=(\ \mathbf{conv}(\mathbf{xpp}(p)),\ \mathbf{load}(\mathbf{conv}(\mathbf{xpp}(q)))\ )$	**char**	**lval**

C^{++}

Generalizing **check** to C^{++} is possible, but a lot of work. Not really because it is hard, but because C^{++} has many complicated rules:

- One has to consider CV-qualifiers: `const` and `volatile`.
- C^{++} has inheritance. It has templates.
- C^{++} uses references instead of lr-value distinction.
- Expressions may contain temporary variables. One needs a conversion for those.
- C^{++} has pretty complicated overload selection rules.
- Primitive operators can be overloaded.

Conclusion

We tried to explain the general principles of type checking, of semantic analysis on ASTs, and we worked out the details for a concrete language, *C*.

The challenge of teaching compiler construction is finding a way of including realistic languages, while at the same time avoiding getting lost in too many details.