Agenda

- Semantic analysis
Type equivalence, compatibility and suitability

- When different types can be used together depends on the context and is governed by language specific rules.

- **Type equivalence**: when can two objects of two different types be considered equivalent.

- **Type compatibility**: when can two objects of two different types be considered compatible.

- **Type suitability**: when can two objects of two different types be considered suitable for one another.

- Assignment usually requires compatibility, expression operands usually require suitability.
### Structural & Name Type Equivalence

```c
typedef struct {
    int A;
    float B;
} X;

typedef X Z;

typedef struct {
    int C;
    float D;
} Y;
```

**X** and **Y** are structurally equivalent, but *not* name equivalent

**X** and **Z** are name equivalent
(therefore structurally equivalent as well)
Type Compatibility

- Type compatibility is used to check if the value of some given type is compatible with another given type.
  - Can RHS value of assignment be assigned to LHS?
  - Can the value expression for a given function argument be passed as the corresponding formal argument?
- Same types are compatible, as are name equivalent types.
- Different types may be compatible if language supports implicit casts.
  - Pass a `char` as an `int` (widening) or `int` as `char` (narrowing, may truncate).
  - Signed to unsigned, and vice versa.
  - Floating point to integer, and vice versa (rounding, loss of precision).
Type Suitability

- Type compatibility is used to check if values can be used together with some language operator/operation

- Does operand match a unary operator?

- Do both operands match the binary operator?

- Do both operands of a binary operator match each other?

- Do both values of a ternary conditional match each other?
Type Suitability

Not Suitable

\n\not \ 1 \hspace{1cm} - \text{true}

\hspace{1cm} 1 \ + \ \text{true}
\hspace{1cm} \text{false or 2}
Type Suitability

Suitable!

```
1 == 2
true == false
```

Not Suitable

```
0 == false
true != 1
```
Type Suitability

Are $X$ and $Y$ type suitable?

Can they be *unified* to a common type? (if $X$ were float, and $Y$ were integer...)
Visibility & accessibility analysis

• Is the usage of a name legal at a given point in the program, according to the language scoping rules?

• Design of symbol table data structures should be congruent with language rules
  • Symbol tables per major scope, with link to parent scope, possibly with access modifiers/conditions

• Typical scoping rules allow names to be *shadowed* (early declaration becomes inaccessible), but not necessarily *deleted*

• Language specific constructs:
  • C++ "*friend*” keyword lets classes access private fields/methods
Visibility & accessibility analysis

class P {
    public int x;
    Private int z;
}

class Q extends P {
    public int y;

    public void M() {
        int z = x + y;
    }
}
Usage analysis

- The application of type equivalence, type compatibility, type suitability, accessibility and visibility

- Is the *use* of a name consistent with its *definition*?
  - Is a name used as a constant/variable/type actually a constant/variable/type?
  - Is a name used as a scalar actually a scalar? Is a variable used as an array actually an array, and is the dimensionality of its use compatible?
  - Is a name used as a function/procedure actually a function/procedure? Is the argument list compatible with the formally declared parameters?
  - Are the operands for all operators correct for that operator?

- Runtime implications
  - Array bounds checking
  - Use of uninitialized variables
Usage analysis

type u32 = int
var x int
var y u32
var a [100]bool
func f(p int) int { ... }

<table>
<thead>
<tr>
<th>Statement</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>u32 = 32</td>
<td>Assignment to a type name</td>
</tr>
<tr>
<td>f(0) = 1</td>
<td>Assignment to a function return value</td>
</tr>
<tr>
<td>x = a</td>
<td>Assign array to scalar</td>
</tr>
<tr>
<td>x[y] = 1</td>
<td>Array subscript on scalar</td>
</tr>
<tr>
<td>x = y.foo.bar</td>
<td>Field access on scalar</td>
</tr>
<tr>
<td>A = f(x, a[0])</td>
<td>Assign int to bool array element</td>
</tr>
<tr>
<td></td>
<td>Wrong # of parameters to f</td>
</tr>
</tbody>
</table>
Performing semantic analysis

• High level overview:

  • Recursively AST starting from the top

  • Collect declarations and populate symbol tables associated to the nearest enclosing scope

  • Process statements for context and correctness

  • Process all expressions, performing name resolution, type checking and usage analysis

  • Optionally replace certain classes of nodes (Ident to Symbol, for example), and add annotations and useful links throughout
Performing semantic analysis

• Sandwich approach:
  • Prepare to handle recursive processing
  • Do recursive visitation
  • Analyze results afterwards
Processing declarations

• Performed at each scope or language construct that can introduce new names (function body, class definition, etc.)

• Processing:
  • Collect declarations: list of names and associated types
  • Lookup each name to check for possible redeclaration
  • Add name to symbol table, with associated entry linking to declaration and all relevant attributes
  • Process initial value expression (if present)
  • Recursively process sub-structures (such as records and functions)
Processing declarations

\begin{verbatim}
var x, y, y, z integer

var t integer = true
\end{verbatim}
Processing declarations

- Design considerations:
  - Does order of declaration matter?
  - How is initialization handled?
Initialization order

```c
int x = 10;
int y = x;
struct { int x } s;
int *sx = &s.x;
```
Initialization order

Variable-length arrays (VLAs)

```c
int x = 10;
char a[x];
```
Handling mutual recursion

```java
func p() {
    ...
}

func q() {
    p()
}
```
Handling mutual recursion

```go
func p() {
    q()       // legal?
}

func q() {
    p()
}
```
Handling mutual recursion

```latex
func p() {
    x = 488    // legal?
}

var x integer
```
Forward prototypes

void q();

void p() {
    q();
}

void q() {
    p();
}
Processing declarations

- Design considerations:
  - What does the symbol table entry for a forward prototype look like?
  - Perform one or two passes
  - One pass:
    - Collect and process names in order, once
  - Two pass:
    - Perform first pass as usual, but don’t recurse into functions
    - On second pass, recurse into function bodies, with all symbols accessible available
Processing statements

• Sandwich processing:
  • Recursively process each sub-structure
    • Expressions, scopes
  • Consider handling language constructs like break’s within loops, or return’s
Variable processing

\[ \text{ident} \left( \text{("argList")} \mid \text{["expr"]} \mid "." \text{field} \mid \downarrow \right)^* \]

- Regular expression describing the general form of a variable reference
- This form expresses:
  - Function invocations
  - Single dimension array subscripting
  - Multiple sub-structure field lookup
  - Pointer dereferencing \(\downarrow\)
  - Multiple repeated iterations of all of the above
Variable processing

ident ("(" argList ")" | "[" expr "]" | "." field | ↑ )*

- Lookup `ident` in current ambient symbol table
- Could be:
  - A local variable
  - A formal parameter
  - A function/procedure
  - The name of a class or module or package
Variable processing

- Check that preceding name referred to a function
- Get symbol table entry for that name
- Verify length of \textit{argList} against formal parameter list
- Check type compatibility & accessibility of each \textit{argList} with corresponding formal parameter
- Expression return type is same as function return type

\[ \text{ident} \left( \left( \left( \text{"argList"} \right) \right) \mid \left( \left[ \text{"expr"} \right] \right) \mid \text{"." field} \mid \uparrow \right)^* \]
Variable processing

- Check that preceding name referred to an array
- Get symbol table entry for that name
- Verify that the type of $expr$ is suitable to use as an index
- Expression return type is same as array subscript type
- Consider the types when indexing a multi-dimensional array…

```
ident ( "(" argList ")" | "[" expr "]" | "." field | ↑ )*
```
Variable processing

ident ( "(" argList ")" | "[" expr "]" | "." field | ↑ )*

- Check that preceding name referred to something that contains named sub-fields (think class, struct/union record, package, module, etc.)
- Get symbol table entry for that name
- Lookup the name field, ensuring that it exists and get its symbol table entry
- Verify that field is visible & accessible
- Expression return type is same as declared field type
Variable processing

- Check that preceding thing is of a pointer-to-something type

- Expression return type is same as that something
Variable processing

\[ \text{ident} \left( \left( \text{"" argList ""} \right) \mid \left[ \text{"" expr ""} \right] \mid \text{"." field} \mid \uparrow \right)^* \]

\[ \begin{align*}
\text{VarRefExpr(name)} \\
\text{FuncCallExpr(callable, args)} \\
\text{SubsExpr(expr, indexer)} \\
\text{FieldExpr(expr, field)} \\
\text{DerefExpr(expr)}
\end{align*} \]
Variable processing

\[ C.x.f[i](a) \]

\[
\text{DerefExpr(}
  \text{FuncCallExpr(}
    \text{SubsExpr(}
      \text{FieldExpr(}
        \text{FieldExpr(}
          \text{VarRefExp('C')},
          'x'),
          'f'),
        VarRefExpr('i')),
      [ VarRefExpr('a') ]
    )
  )
\]