Equivalence of Type Expressions

- Recall:
  - Basic types are type expressions.
  - Certain operators (such as tuples, records, arrays, functions, classes, inheritance, …) can be applied to other type expressions to create new type expressions. Such operators are called type constructors.
  - We may also give (user-defined) names to type expressions (think `typedef` in C or names of classes in C++). Such type names are also valid type expressions.
- When should we consider two type expressions T_1 and T_2 to be equivalent ($T_1 \equiv T_2$)?
  1. If we don't have type names, then structural equivalence is the natural choice.
  2. If we do have type names, then we could choose either structural equivalence or name equivalence.
  3. With type names, we can have the possibility of circular types. How do we handle these?

Structural Equivalence

TypeExpr → *BasicType* | *TypeConstr TypeExpr*

BasicType → **int** | **char** | …

TypeConstr → **Array** | **Prod** | **Ptr** | **Func** | …

- Definition of $T_1 \equiv_s T_2$.
  - If $T_1 \in \mathit{BasicType} \wedge T_2 \in \mathit{BasicType}$, then $T_1 = T_2$.
  - If $T_1 = \mathbf{Array}(N_1, T_1') \wedge T_2 = \mathbf{Array}(N_2, T_2')$ then $N_1 = N_2 \wedge T_1' \equiv_s T_2'$.
  - Similarly for other type constructors.
  - Otherwise false.

- In other words, the ASTs of T_1 and of T_2 must be identical.

Structural Equivalence

TypeExpr → *BasicType* | *TypeConstr TypeExpr*

BasicType → **int** | **char** | ...

TypeConstr → **Array** | **Prod** | **Ptr** | **Func** | ...

- Definition of $T_1 \equiv_S T_2$.
  - If $T_1 \in \mathit{BasicType} \wedge T_2 \in \mathit{BasicType}$, then $T_1 = T_2$.
  - If $T_1 = \mathbf{Array}(N_1, T_1') \wedge T_2 = \mathbf{Array}(N_2, T_2')$ then $N_1 = N_2 \wedge T_1' \equiv_S T_2'$.
  - Similarly for other type constructors.
  - Otherwise false.

- In other words, the ASTs of T_1 and of T_2 must be identical.

- [C] Do the identifiers `f` and `g` declared as `int *f(void);` and `int (*g)(void);` have equivalent types?

Bit Operations for Structural Equivalence

TypeExpr → *BasicType* | *TypeConstr TypeExpr*

BasicType → **int** | **char** | ...

TypeConstr → **Array** | **Prod** | **Ptr** | **Func** | ...

- This idea works if the type constructors are unary.
    - E.g., drop the length of arrays and the argument types of functions.
- Encode basic types with some number of bits.
- Have a unique bit pattern for each type constructor.
- Concatenate bits to get a unique pattern for each type.
- Used in early C compilers.

Equivalence When Type Names Occur

TypeExpr → *BasicType* | *TypeConstr TypeExpr* | *TypeName*

BasicType → **int** | **char** | …

TypeConstr → **Array** | **Prod** | **Ptr** | **Func** | …

TypeName → **TypeDef id =** *TypeExpr*

Equivalence When Type Names Occur

TypeExpr → *BasicType* | *TypeConstr TypeExpr* | *TypeName*

BasicType → **int** | **char** | ...

TypeConstr → **Array** | **Prod** | **Ptr** | **Func** | ...

TypeName → **TypeDef id** = *TypeExpr*

- [Structural Equivalence] Definition of $T_1 \equiv_S T_2$.
  - If $T_1 \in \mathit{BasicType} \land T_2 \in \mathit{BasicType}$, then $T_1 = T_2$.
  - If $T_1 = \mathbf{Array}(N_1, T_1') \land T_2 = \mathbf{Array}(N_2, T_2')$ then $N_1 = N_2 \land T_1' \equiv_S T_2'$. ...
  - If $T_1 \in \mathit{TypeName} \land T_2 \in \mathit{TypeName}$, then $T_1 \equiv_S T_2$.
  - Otherwise false.

Equivalence When Type Names Occur

TypeExpr → *BasicType* | *TypeConstr TypeExpr* | *TypeName*

BasicType → **int** | **char** | ...

TypeConstr → **Array** | **Prod** | **Ptr** | **Func** | ...

TypeName → **TypeDef id** = *TypeExpr*

- [Structural Equivalence] Definition of $T_1 \equiv_S T_2$.
  - If $T_1 \in$ *BasicType* $\wedge$ $T_2 \in$ *BasicType*, then $T_1 = T_2$.
  - If $T_1 = \mathbf{Array}(N_1, T_1') \wedge T_2 = \mathbf{Array}(N_2, T_2')$ then $N_1 = N_2 \wedge T_1' \equiv_S T_2'$. ...
  - If $T_1 \in$ *TypeName* $\wedge$ $T_2 \in$ *TypeName*, then $T_1 \equiv_S T_2$.
  - Otherwise false.
- [Name Equivalence] Definition of $T_1 \equiv_N T_2$.
  - If $T_1 \in$ *BasicType* $\wedge$ $T_2 \in$ *BasicType*, then $T_1 = T_2$.
  - If $T_1 = \mathbf{Array}(N_1, T_1') \wedge T_2 = \mathbf{Array}(N_2, T_2')$ then $N_1 = N_2 \wedge T_1' \equiv_N T_2'$. ...
  - If $T_1 \in$ *TypeName* $\wedge$ $T_2 \in$ *TypeName*, then $T_1 = T_2$.
  - Otherwise false.

Type Names and Cyclic Type Expressions

- Consider the following type expressions.
  - [Pascal]
    ```
    type link = ^cell;
         cell = record
                  info: integer;
                  next: link
                end;
    ```
  - [C]
    ```
    struct cell {
        int info;
        struct cell *next;
    };
    ```
- What is the structure of these types?
  - Acyclic representation.
  - Cyclic representation.

Real-Life Examples

- C [C17 ballot, §6.2.7]
  Two types have compatible type if their types are the same. [...] For two structures, corresponding members shall be declared in the same order. For two structures or unions, corresponding bit-fields shall have the same widths. For two enumerations, corresponding members shall have the same values.

Real-Life Examples

- C [C17 ballot, §6.2.7]
  Two types have compatible type if their types are the same. [...] For two structures, corresponding members shall be declared in the same order. For two structures or unions, corresponding bit-fields shall have the same widths. For two enumerations, corresponding members shall have the same values.

- Java [JLS16, §4.3.4]
  Two reference types are the same compile-time type if they are declared in compilation units associated with the same module (§7.3), and they have the same binary name (§13.1), and their type arguments, if any, are the same, applying this definition recursively.

  [...]

  At run time, several reference types with the same binary name may be loaded simultaneously by different class loaders. These types may or may not represent the same type declaration. Even if two such types do represent the same type declaration, they are considered distinct.

  Two reference types are the same run-time type if:
  • They are both class or both interface types, are defined by the same class loader, and have the same binary name (§13.1) [...].
  • They are both array types, and their component types are the same run-time type (§10 (Arrays)).