Type Systems

- Recall: The set of types in a programming language, along with the rules that use types to specify program behavior, are collectively called a type system.

Type Systems

- Recall: The set of types in a programming language, along with the rules that use types to specify program behavior, are collectively called a type system.
- Examples
  - [Java: JLS16]
    §4.2.1 The values of the integral types are integers in the following ranges: [...] For `int`, from -2147483648 to 2147483647, inclusive [...]
    §4.2.2 If an integer operator other than a shift operator has at least one operand of type `long`, then the operation is carried out using 64-bit precision, and the result of the numerical operator is of type `long`. If the other operand is not `long`, it is first widened to type `long` by numeric promotion. Otherwise, the operation is carried out using 32-bit precision, and the result of the numerical operator is of type `int`. If either operand is not an `int`, it is first widened to type `int` by numeric promotion. [...]
    §4.3.1 An *object* is a *class instance* or an *array*. The reference values (often just *references*) are pointers to these objects, and a special null reference, which refers to no object. [...]

Type Systems

- Recall: The set of types in a programming language, along with the rules that use types to specify program behavior, are collectively called a type system.
- Examples
  - [Java: JLS16]
    §4.2.1 The values of the integral types are integers in the following ranges: [...] For `int`, from -2147483648 to 2147483647, inclusive [...]
    §4.2.2 If an integer operator other than a shift operator has at least one operand of type `long`, then the operation is carried out using 64-bit precision, and the result of the numerical operator is of type `long`. If the other operand is not `long`, it is first widened to type `long` by numeric promotion. Otherwise, the operation is carried out using 32-bit precision, and the result of the numerical operator is of type `int`. If either operand is not an `int`, it is first widened to type `int` by numeric promotion. [...]
    §4.3.1 An *object* is a *class instance* or an *array*. The reference values (often just *references*) are pointers to these objects, and a special null reference, which refers to no object. [...]
  - [C: C17 Ballot]
    §6.5.3.2 The unary `&` operator yields the address of its operand. If the operand has type "*type*", the result has type "pointer to *type*". If the operand is the result of a unary `*` operator, neither that operator nor the `&` operator is evaluated and the result is as if both were omitted, except that the constraints on the operators still apply and the result is not an lvalue. Similarly, if the operand is the result of a `[ ]` operator, neither the `&` operator nor the unary `*` that is implied by the `[ ]` is evaluated and the result is as if the & operator were removed and the `[ ]` operator were changed to a + operator. Otherwise, the result is a pointer to the object or function designated by its operand.

Type Expressions

- Every language construct has a type associated with it. This type will be denoted by a type expression.

Type Expressions

- Every language construct has a type associated with it. This type will be denoted by a type expression.
- Intuition
  - 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.

Type Expressions

- Every language construct has a type associated with it. This type will be denoted by a type expression.
- Intuition
  - 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.
- We will need to have notions of equivalence and inclusion among type expressions.
  - For type expressions s and t, $s \equiv t$ means that these expressions "represent the same type".
  - For type expressions s and t, $s \prec t$ means that a construct of type s can be used in a context requiring a construct of type t.

A Concrete Example

- A type system modeled after C’s type system.

A Concrete Example

- A type system modeled after C's type system.
- Basic types
  - Arithmetic types: `char`, `int`, `uint`, `long`, `ulong`, `float`, `double`.
  - The `void` type specifies an empty set of values.
  - The `error` type will be used to signal a type inference error.

A Concrete Example

- A type system modeled after C's type system.
- Basic types
  - Arithmetic types: `char`, `int`, `uint`, `long`, `ulong`, `float`, `double`.
  - The `void` type specifies an empty set of values.
  - The `error` type will be used to signal a type inference error.
- Derived types
  - Arrays: If T is a type expression and N is an integer literal, then $\mathbf{ARRAY}(N, T)$ is a type expression denoting the type of an array with elements of type T.

A Concrete Example

- A type system modeled after C's type system.
- Basic types
  - Arithmetic types: `char`, `int`, `uint`, `long`, `ulong`, `float`, `double`.
  - The `void` type specifies an empty set of values.
  - The `error` type will be used to signal a type inference error.
- Derived types
  - Arrays: If T is a type expression and N is an integer literal, then $\mathbf{ARRAY}(N, T)$ is a type expression denoting the type of an array with elements of type T.
  - Products (tuples): If T_1 and T_2 are type expressions, then their Cartesian product $T_1 \times T_2$ is a type expression. The operator $\times$ is left-associative.

A Concrete Example

- A type system modeled after C's type system.
- Basic types
  - Arithmetic types: `char`, `int`, `uint`, `long`, `ulong`, `float`, `double`.
  - The `void` type specifies an empty set of values.
  - The `error` type will be used to signal a type inference error.
- Derived types
  - Arrays: If T is a type expression and N is an integer literal, then $\mathbf{ARRAY}(N, T)$ is a type expression denoting the type of an array with elements of type T.
  - Products (tuples): If T_1 and T_2 are type expressions, then their Cartesian product $\mathrm{T}_1 \times T_2$ is a type expression. The operator $\times$ is left-associative.
  - Records: If $I_1, \ldots, I_k$ are distinct identifiers, and $T_1, \ldots, T_k$ are type expressions, then $\mathbf{RECORD}(I_1 \colon T_1, \ldots, I_k \colon T_k)$ is a type expression denoting a record type with k named fields.

A Concrete Example

- A type system modeled after C's type system.
- Basic types
  - Arithmetic types: `char`, `int`, `uint`, `long`, `ulong`, `float`, `double`.
  - The `void` type specifies an empty set of values.
  - The `error` type will be used to signal a type inference error.
- Derived types
  - Arrays: If T is a type expression and N is an integer literal, then $\mathbf{ARRAY}(N, T)$ is a type expression denoting the type of an array with elements of type T.
  - Products (tuples): If T_1 and T_2 are type expressions, then their Cartesian product $T_1 \times T_2$ is a type expression. The operator $\times$ is left-associative.
  - Records: If $I_1, \ldots, I_k$ are distinct identifiers, and $T_1, \ldots, T_k$ are type expressions, then $\mathbf{RECORD}(I_1 \colon T_1, \ldots, I_k \colon T_k)$ is a type expression denoting a record type with k named fields.
  - Pointers: If T is a type expression, then $\mathbf{PTR}(T)$ is a type expression denoting the type "pointer to an entity of type T".

A Concrete Example

- A type system modeled after C's type system.
- Basic types
  - Arithmetic types: `char`, `int`, `uint`, `long`, `ulong`, `float`, `double`.
  - The `void` type specifies an empty set of values.
  - The `error` type will be used to signal a type inference error.
- Derived types
  - Arrays: If T is a type expression and N is an integer literal, then $\mathbf{ARRAY}(N, T)$ is a type expression denoting the type of an array with elements of type T.
  - Products (tuples): If T_1 and T_2 are type expressions, then their Cartesian product $T_1 \times T_2$ is a type expression. The operator $\times$ is left-associative.
  - Records: If $I_1, \ldots, I_k$ are distinct identifiers, and $T_1, \ldots, T_k$ are type expressions, then $\mathbf{RECORD}(I_1 \colon T_1, \ldots, I_k \colon T_k)$ is a type expression denoting a record type with k named fields.
  - Pointers: If T is a type expression, then $\mathbf{PTR}(T)$ is a type expression denoting the type "pointer to an entity of type T".
  - Functions: If D and R are type expressions denoting a domain type and a range type, then $D \to R$ is a type expression denoting the type of a function mapping elements of D to elements of R. $\to$ is right-associative.