CS143
Compilers

Lecture 7

July 25, 2001

Type Checking

- Having learned how to compute attribute values for all of the constructs in a source language, we are now ready to begin semantic analysis, our final stage of analysis before synthesizing target code.
- One of the most essential checks a compiler performs is type checking, the process of verifying that the operators in a source program are acting on the appropriate operands, based on the rules imposed by the specification of the language. The information gathered during type checking helps determine the operations to be performed by the target language.

Type Systems

- A type system is a specification of the types of data supported by a language, accompanied by a set of rules that define the types of language constructs.
- Using the information described by a type system, one can construct a translation scheme that describes, by means of attributes, how the types of grammar symbols (i.e. language constructs) are to be determined during compilation from the types of other symbols, usually child nodes in the parse tree.
- In addition, semantic actions can be added in order to detect, report, and recover from semantic errors.

Type Expressions

- *Type expressions* lend a useful, universal notation for describing the types of language constructs and facilitating type checking.
- Type expressions can be constructed as follows:
  - All basic types supported by the language are type expressions.
  - Special basic types, indicating semantic errors or the absence of a value, are type expressions.
  - A *type constructor* applied to a type expression is a type expression. Common constructors include arrays, cartesian products, records, pointers, and functions.

Static and Dynamic Type Checking

- Type checking can be performed statically, at compile time, or dynamically, at run time by the target program.
- In order to perform dynamic checking, values of objects must be accompanied by information about their types, leading to inefficiency in the target code.
- A *sound* type system is a type system that ensures that type errors can always be detected at compile time.
- A language is said to be *strongly typed* if it can guarantee that programs accepted by the compiler can run without type errors. In practice, though, some checks cannot be performed statically, such as ensuring array indices are valid.

Error Recovery

- As with lexical and syntax analysis, error recovery is a priority during type checking.
- Once an expression is determined to have an invalid type, due to the incompatibility of an operator with one or more of its operands, it is advisable to assign the expression a valid type so that meaningful analysis can continue.
- For this reason, the design of a type system should include provisions for errors, incorporated into the rules that assign types to constructs.

Equivalence of Type Expressions

- Often, it is necessary to compare two type expressions and determine whether they are equivalent.
- It follows that a compiler should have an efficient algorithm for determining whether two arbitrary type expressions are equivalent. This can be accomplished by choosing an appropriate representation for type expressions, encoding basic types and constructors used in such expressions.
- There are two standards for determining equivalence: structural equivalence and name equivalence.

Structural Equivalence of Type Expressions

- Two type expressions are said to be *structurally equivalent* if they are either the same basic type, or are formed by applying the same constructor to two other structurally equivalent types.
- In practice, structural equivalence is often applied loosely, in order to fit the semantic rules of the language. For example, many languages allow arrays of variable sizes, but of fixed dimension, to be passed as arguments to functions.

Names for Type Expressions

- In many languages, types can be named. This yields the standard of *name equivalence*, in which two type expressions are considered equivalent if they have the same name.
- While this standard of equivalence can cause operations to be deemed invalid while structural equivalence would indicate that they are valid, this standard makes type checking more efficient since determining compatibility of type expressions is trivial.

Cycles in Representations of Types

- It is natural to represent a type expression using a graph, where nodes represent basic types and constructors, and edges represent the application of constructors to other type expressions.
- In types such as records, it is very common to have type expressions defined recursively. When using structural equivalence, the graph for such a type expression has a cycle.
- In practice, cycles are avoided by using a combination of name and structural equivalence to determine equivalence of type expressions.

Type Conversions

- Even when a program is deemed valid in that all operators are applied to operands of valid types, it is frequently necessary to convert operands to objects of a different type. This eliminates the need for a separate operator for each acceptable combination of types of operands.
- Type conversions can be performed implicitly by the compiler; such conversions are called coercions. In other cases, the programmer must explicitly specify the conversion to be performed. Implicit conversion can be very helpful in improving the efficiency of the target program.

Overloading of Functions and Operators

- An *overloaded* operator is an operator that has different meanings depending on the types of its operands. In most source languages, for instance, arithmetic operators are overloaded. However, target languages tend not to overload operators, as they contain machine-dependent instructions that require specific types.
- Whenever an overloaded operator is used, it must be *resolved* so that a unique operation is determined. This process is known as *operator identification*.

Set of Possible Types of Subexpressions

- In many cases, overloaded operators can be resolved simply by examining the types of the operands, but in some cases the type of an expression depends on its context. This yields a set of valid types that a given expression may have.
- By determining a unique type for an expression's context, we can narrow the set of possible types that the expression can have.
- This process requires heavy use of both synthesized and inherited attributes.

Polymorphic Functions

- A *polymorphic function* is a function that can accept arguments of varying types. Many operators in common source languages are polymorphic.
- While such functions can complicate type checking, they are highly desirable because they facilitate the implementation of abstract operations that act on data of varying types, such as determining the length of a list without caring about the types of the elements of the list.

Type Variables

- To facilitate type checking of polymorphic functions, *type variables* can be used. Such variables are essentially type expressions containing variables as well as fixed type expressions.
- Type variables aid in the process of *type inference*, which is the determination of the type of a language construct from the way in which it is used.
- Type variables can be used to ensure that the usage of a given construct is consistent throughout the source program.

Substitutions, Instances and Unification

- A natural question is, given two type variables, how do we know if they are compatible?
- This question is answered by the process of *unification*, which entails substituting the variables in each type expression with other type expressions in an attempt to attain structural equivalence.
- Unification proceeds by representing each type expression as a graph and attempting to "merge" the graphs by matching constructors, basic types, and variable substitutions.