Design Patterns for Teaching Type Checking in a Compiler Construction Course

Francisco Ortin, Daniel Zapico, Juan Manuel Cueva

Abstract. A course in compiler construction seeks to develop an understanding of well-defined fundamental theory and typically involves the production of a language processor. In a graduate degree in software engineering, the development of a compiler contributes significantly to the developer's comprehension of the practical application of theoretical concepts.

Different formal notations are commonly used to define type systems, and some of them are used to teach the semantic analysis phase of language processing. In the traditional approach, attribute grammars are probably the most widely used ones. This paper shows how object-oriented design patterns represented in UML can be used to both teach type systems and develop the semantic analysis phase of a compiler. The main benefit of this approach is two-fold: better comprehension of theoretical concepts because of the use of notations known by the students (UML diagrams), and improvement of software engineering skills for the development of a complete language processor.

Keywords: Compiler construction, design pattern, type checker, type system, software engineering, semantic analysis.

I. Introduction

A course in compiling techniques is an important part of computing core curricula [1]. The process of compiler construction is supported by well-defined theory and exploit concepts, principles, and software development skills drawn from other related disciplines, such as programming, software engineering, computer architecture and organization, and operating systems [1]. The teaching of compiling techniques is usefully augmented by a realistic example of the systematic construction, or software engineering, of a compiler as exemplified by computer science curricula [2]-[3]. Software engineering techniques can help students understand concepts that underpin the compilation process.

The Compiler Construction course at the University of Oviedo (Spain) is taught in the first year of a software engineering graduate program. This course is offered to graduate students over two quarters in the classical “phase order” of the compilation process seen in so many compiler texts [4]. The first quarter covers an introduction followed by analysis issues, such as lexical analysis, syntax analysis (or parsing), and semantic analysis. The second quarter covers symbol tables, runtime-memory organization, code generation, and interpreter implementation.

The prerequisite knowledge expected of students in the graduate course include sufficient programming experience with object-oriented programming languages (Java or C++) and object-oriented technologies, and the ability to read and reason about UML diagrams, data structures, and object-oriented design. The course comprises two hours per week of lectures, plus three hours per week of practical work which includes a weekly assignment and a final assessed project.

The graduate program emphasizes software engineering techniques and seeks to develop the practical skills necessary to design and implement larger-scale software systems. A course in compiling techniques, within the context of a program intended to develop software engineering abilities, involves the systematic construction of a compiler by practical application of the underlying theory to each phase of compiler design and implementation.

This paper focuses on the importance of type systems as a basis for developing a better understanding of the semantic analysis phase of the compilation process. Several formalisms are commonly used in the design and verification of the type systems of programming languages, but they are not applied in the commercial implementation of compiler type checkers [5]. The main reasons are two-fold: first, (1) formal methods are focused on proving properties of type systems rather than implementing typecheckers; second, (2) the code generated by these tools is inefficient and usually difficult to debug and trace, because the generated code comes from a general translation of mathematical specifications.

First, concepts such as type expression, type coercion, type equivalence, polymorphism and overriding are introduced to the students. Later, the design of each concept is presented using design patterns in UML and implementations of the designs are shown in different object-oriented programming languages. Java and C++ are the two programming languages known by the students. Depending on the undergraduate courses which he or she undertakes, a student might feel more confident in one of these two languages than the other one. C++ is the programming language of choice for the examples presented in this paper.

The rest of the paper is structured as follows. Section 2 shows different approaches to modeling type expressions. Section 3 presents type expression equivalence. An object-oriented design to reduce memory and CPU consumption is described in Section 4. Methods for developing type coercion features and parametric polymorphism are described in sections 5 and 6, respectively. Section 7 compares this course to a traditional approach and Section 8 analyzes the evaluation of this course. The conclusions are presented in section 9.

II. Type Expressions

Typechecking is the analysis that detects semantic inconsistencies and anomalies, guarantees that entities match their declaration and establishes this analysis compliance with a given type system; the algorithm that performs type checking is implemented by a type checker [6]. A language processor typically implements a type checker in its semantic (contextual) analysis phase [4]. The main benefits of static (compile-time) type checking are [7] error detection, abstraction, performance, safety, and documentation.

A language processor has to represent the types of the language internally when implementing a type checker. A type expression is an internal (and also external in the case of the ML language) representation of the type of a language syntactic construction [4]. Therefore, any language processor must model all the type expressions of the language being processed in order to perform type checking apart from the rest of translation tasks.

Type expressions are based on the constructive definition of types: a type is either a set of basic types (also known as predefined or built-in types) or a constructed type, composed from other types. A basic type is an atomic type whose internal structure

cannot be modified (specified) by the programmer (e.g., *integer*, *float*, or *boolean*). A constructed type is built by the programmer, applying type constructors (e.g. *record*, *array*, *set*, or *class*) to other types, either basic or constructed ones.

A. Representation of Type Expressions

Internal representation of type expressions could vary depending on the features of both the language to be processed and the language used to implement the compiler. For example, representing type expressions of a language that only has built-in types would be as easy as using integer, enumerable, or even character variables. This representation, however, would not be valid if the language to be processed supports type constructors. The type constructor, e.g., “pointer to type T” could create an infinite number of type expressions.

When type constructors are present, a schema to model infinite type expressions is needed. One feasible solution might be using character strings to represent type expressions, via some kind of type expression language. An example of this approach is detailed in [4]. The most serious disadvantage of this approach is the need to process the string as a program to extract component types of a constructed type, at the expense of too many computational resources.

A simpler and more efficient solution is using recursive data structures by means of objects. This structure is easier to manipulate than character strings and requires less computational power when processing a compound type expression. Some routines are common to most type expression, and particular routines are necessary for specific type expressions. By using object orientation, these common and specific routines can be properly set to each type expression by means of classes and polymorphism.

B. The *Composite* Design Pattern

The problem of representing recursively composite structures in a hierarchal way appears in several contexts in computer science. The object-oriented **Composite** pattern [8] is used to model and solve this sort of problem. This pattern models both primitive and recursively composed structures, offering a uniform way to manipulate these heterogeneous structures. An example of the Composite design pattern is represented by the class diagram in Fig. 1. The elements of the Composite design patterns are as follows.

1. **Component** (*TypeExpression*). A (usually abstract) class that declares the interface of all the elements (simple and compound) defining their uniform processing. Every method can offer a default implementation (*asterisk*, *squareBrackets*, *brackets*, and *dot*) or be declared abstract to be implemented by the *Component* subclasses (*getBytes* and *typeExpression*).
2. **Leaf** (*Character*, *Integer*, *Void*, *Boolean*, and *Error*). These elements represent leaf nodes in the hierarchical structure. A leaf object does not own a “child” type. The methods of the *Leaf* class define the concrete operations for this specific node.
3. **Composite** (*Pointer*, *Array*, *Struct*, and *Function*). This element models structures built by composition of others, managing references to the types to which the type constructor was applied. The *Composite* implements its methods taking into account the referred types, even obtaining partial results from the operations implemented by each of the *Composite* child nodes.

C. Implementation of Type Expressions

When applying the *Composite* design pattern to implement type expressions, every built-in type will be a leaf node, and every compound type will be a separate *Composite* class. The *TypeExpression* class will hold operations of the semantic analysis phase; that is, operations common to all types (equivalence, coercion, inference, or unification). *TypeExpression* will also hold operations for the code-generation phase (size or low-level representation). Placing these methods in the root class of the hierarchy implies a uniform treatment of all types according to the interface of the *Component* class, regardless of their internal structure.

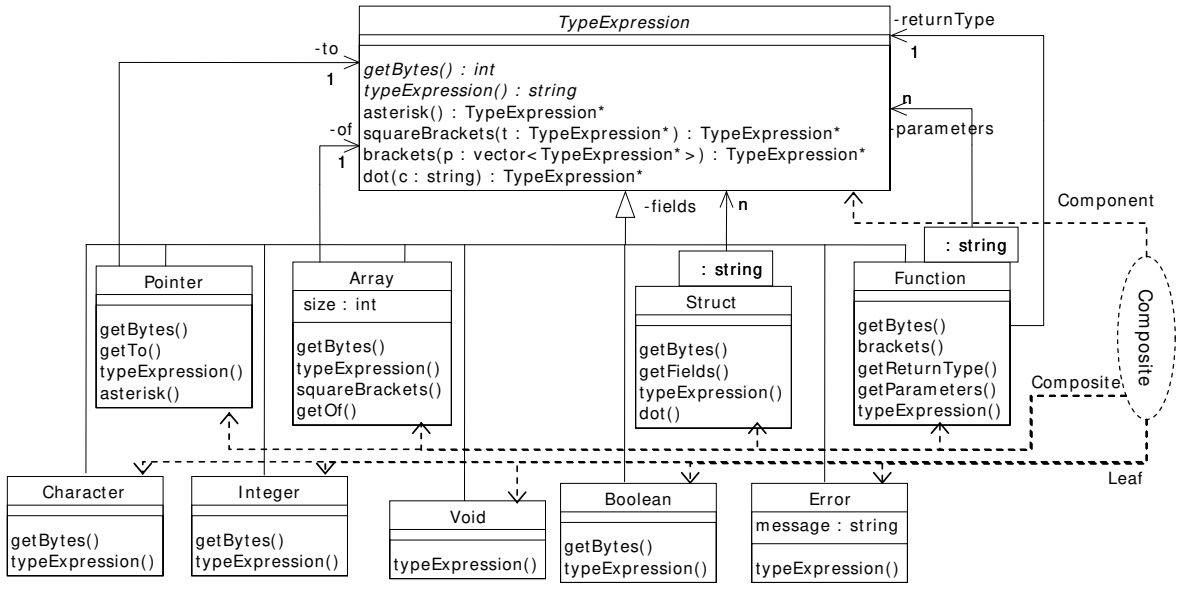

Figure 1. Class diagram to model some type expressions of the C programming language.

When most types have the same behavior for a specific operation, that behavior will be implemented as a method of the *TypeExpression* class. Then, if required, each subclass could override the general operation defined in the *TypeExpression* class; otherwise, the default behavior will be inherited. Moreover, every sub-class can define specific methods according to the type expression each class represents, without

declaring those specific methods in the base class. That is, these methods define messages of that specific type expression, and they are not applicable to the rest of the existing types.

Figure 1 shows the preliminary design of type expressions of part of the C programming language, following the *Composite* design pattern. The *Component* role is played by the *TypeExpression* abstract class, defining the general behavior of all type expressions. Some of the messages accepted by the *TypeExpression* class have a default implementation. These sample messages are as follows.

- *asterisk*, *squareBrackets*, *brackets*, and *dot*. These methods have two responsibilities. First is inferring the resulting type when a language operator is applied to an expression. Many methods receive other type expressions as parameters. For instance, a function invocation (*brackets*) requires the type of each argument to infer the returning type. The second responsibility is type checking, which needs to know whether a language operator could be applied to an expression. If the operation makes no sense for a specific type expression (a type error is committed), an *Error* type expression will be returned. The new instance of the *Error* class created will store the error description.

  The default implementation of all of these methods indicates that such operations are not semantically defined for the corresponding type, giving back an *Error* object to the caller. For each operator that can be syntactically applied to a type, the programmer will have to override the corresponding method. For example, the type *Pointer* class implements the *asterisk* method because the “*” operator is allowed

over pointers. This method returns a type expression representing the type the pointer references.

- *typeExpression*. Returns a character string representing the type expression of the type modeled, using a notation employed in many compilers [4]. This method facilitates debugging tasks at the same time as reducing memory consumption. Further explanation is given in the next section. An example of a C++ implementation of this method for the *Function* class is shown in Fig. 2. In this example, a type expression that represents a function with two arguments, a pointer to an integer and a real, and returns nothing, will be translated to the string "*(Pointer(Integer),Float)->Void*".
- *getBytes*. Returns the number of bytes needed to hold a variable of each type. This message is an example of how the types of the language can also represent responsibilities of the code generation phase.

```
string Function::typeExpression() const {
    ostringstream o;
    if (parameters.size()) {
          o<<"(";
          for (unsigned i=0;i< parameters.size();i++)
                o<< parameters [i]->typeExpression()<<
                      (i<parameters.size()-1 ? ',' : ')');
    }
    o<<"->"<<returnType->typeExpression();
    return o.str();
}
```

Figure 2. Sample implementation for computing type expressions.

Compound types have associations with base types; a pointer requires the type the pointer points to (*to*); an array requires the type the array "collects" (*of*); a *struct* requires a collection of its fields (*fields*), qualified by the name of each field (*string*); a function requires an association with the type of the value(s) the function returns (*returnType*) and a collection of its parameters qualified by name (*parameters*). Since all the

associations point to the root class of the hierarchy, compound types can be composed of any type, including themselves, by means of polymorphism.

Additionally, this design facilitates type construction using existing analysis tools, such as lex/yacc, Antlr, and JavaCC. Type expressions are built in a syntax-directed manner by means of type constructors. Basic types are created first and later used to build more complex types in the semantic routines of analysis tools.

D. An Example of Use

This section introduces a sample scenario of the design presented above, representing type expressions for a subset of the C programming language. Lexical and syntactic features have been specified by means of lex and yacc, respectively, and the remaining parts of the processor have been implemented in ISO/ANSI C++. When a variable is defined (lines 1 to 9 in Fig. 3), the variable is associated with its type expression (a pointer to the *TypeExpression* class) in a symbol table. In these semantic routines, types are created by invoking the appropriate type constructors so that complex types are composed while yacc is performing reductions.

Although this example is of a single-pass compiler, this design is also suitable for multi-pass compilers. In that case, type expressions would be constructed by executing the visit methods of the **Visitor** design pattern [8]-[9].

Function statements are sequences of expressions. Each expression has a type, i.e., a pointer to *TypeExpression*. For every reduction performed by yacc, the type of the sub-expression is inferred by applying the suitable operator. This inference is processed by sending the appropriate message to the type of the sub-expression.

Figure 3 shows a trace of all of the type expressions inferred in each statement of the main program. Source code with its line numbers is shown in the left part; whereas, the string returned when sending the message *typeExpresssion* to each inferred type can be seen in the right part of Fig. 3. For example, a *Pointer* type is obtained from the first sub-expression in line 13 (*pointer* variable); the *asterisk* is invoked in the next reduction so that the pointed type is inferred (*Integer*); the next step is getting the type of *vector* from the symbol table; then, the *squareBrackets* message is passed to *Array* (the type of *vector*), using the *Integer* type expression (previously inferred) as an argument; the final type inferred is *Integer*.

```
 1:      int vector[10];
 2:      int *pointer;
 3:      int **doublePointer;
 4:      char **v[10];
 5:      char *w[10];
 6:      struct Date { int day, month, year; };
 7:      struct Date date;
 8:      int *f(int,char*);
 9:      void p(int*);
10:      int main() {
11:        date;                 Struct((day x Integer)x
                                     (month x int)x(year x Integer))
12:        v;                    Array(10,Pointer(Pointer(Character)))
13:        vector[*pointer];     Integer
14:        **v[**doublePointer]; Character
15:        w[*f(3,w[1])];        Pointer(Character)
16:        p(f(date.day,w[2]));
17:      }
```

Figure 3. Type inference trace of an example program.

The processes of type inference and type checking are offered in a uniform way. Namely, dissimilar algorithms can be applied without distinguishing between different type expressions. All the type expressions are treated uniformly, obtaining heterogeneous behavior through polymorphism. Figure 4 illustrates this uniformity feature using yacc in a single-pass compiler. The *asterisk*, *squareBrackets*, and *dot* messages are responsible for this heterogeneous functionality, regardless of the type expression that is being used.

```
exp: '(' exp ')'                { $$=$2; }
   | '*' exp                    { $$=$2->asterisk(); }
   | exp '[' exp ']'            { $$=$1->squareBrackets($3); }
   | exp '.' ID                 { $$=$1->dot($3); }
   | INT_LITERAL                { $$=new Integer($1); }
   | CHAR_LITERAL               { $$=new Character($2); }
   | ID                         { $$=symbolTable.find($1); }
```

Figure 4. Uniform type inference and type checking with yacc.

III. Type Equivalence

In the semantic analysis phase of a compiler, the use of equivalent types should be verified each time a type needs to be checked. Hence, a key issue in the design of a type system is the formulation of the conditions that two objects must satisfy to be considered equivalent.

Different approaches of type equivalence are offered by several programming languages, mainly classified into two families [10].

1) **Structural equivalence**. Two types are structurally equivalent only if they have the same structure; that is, either they are the same basic type, or they have been built applying the same type constructor to structurally equivalent types. Languages, such as ML, Algol-68 and Modula 3, employ a type system based on structural equivalence. C++ implements structural equivalence, except for classes, structs, and unions.
2) **By-name equivalence**. Every type has a unique name. Hence, two types are equivalent only if they have the same name. Ada and Java are two languages that use by-name equivalence.

The design of the second family is easier than the first one because by-name equivalence is as simple as comparing unique identifiers of each type. However, with

structural equivalence the structure of each of the types should be recursively compared.

A. Implementation of Type Equivalence

Using the presented design, the implementation of a structural equivalence algorithm becomes a relatively simple task. Since every type should be comparable, an *equivalentTo* method is added to the *TypeExpression* class. This method checks whether or not each type is equivalent to the one passed as a parameter.

In the case of simple types, the *equivalentTo* method simply checks if the type of the parameter is the same as the implicit object –the default implementation of the method in the *TypeExpression* root class.

In the case of complex types, the type equivalence process should also verify that both types have been built using the same type constructor. In addition, a recursive comparison of each child type expression should be performed. A C++ implementation of both cases is shown in Fig. 5.

```
bool TypeExpression::equivalentTo(const TypeExpression *te) const {
    return typeid(*this)==typeid(*te); /* Runtime Type Information */
}
bool Struct::equivalentTo(const TypeExpression *te) const {
    const Struct *record=dynamic_cast<const Struct*>(te);
    if (!record) return false;
    if (fields.size()!=record->fields.size()) return false;
    map<string, TypeExpression*>::const_iterator it1,it2;
    for (it1=fields.begin(),it2=record->fields.begin();
                        it1!=fields.end();++it1,++it2) {
      if (it1->first!=it2->first) return false;
      if (!it1->second->equivalentTo(it2->second)) return false;
    }
    return true;
}
```

Figure 5. Example implementation of structural equivalence.

Using the RunTime Type Information (RTTI) provided by the standard ISO/ANSI C++ programming language, a default implementation returns whether or not the two type

expressions are the same class. In the case of *Struct*, type structures should have the same number of fields; each field name must match; and all of the field's type expressions need to be structurally equivalent.

IV. Type Representation by Means of Directed Acyclic Graphs

Applying the class diagram showed in Fig. 1, type expressions may be created with a tree-like object structure. Tree structures duplicate type representation for different syntactic constructions with the same type. This object duplication could become critical when creating complex type expressions for real programs, involving an unacceptable number of duplicated objects at program compilation. This high memory consumption and the excessive computation needed to create and explore these object structures requires the redesign of type expressions.

A. The *Flyweight*, *Builder*, and *Singleton* Design Patterns

These design problems can be solved by applying the **Flyweight** design pattern [8]. This pattern is based on identifying objects that are shared simultaneously in many contexts, representing shared object states apart from particular object states. As the number of objects increases, their shared state (*Flyweight*) is represented by a single object; whereas, other instances of smaller size represent individual information of each particular object.

The distinction between shared and particular states is straightforward when representing type expressions. In a source code input, a language processor will recognize a considerable number of symbols of the same type. Symbol information is individual; each variable must have an associated offset in memory, a scope, a type,

and an identifier. However, information held by a type could be shared by symbols and syntactic constructions of the same type. Types manage responsibilities such as size, equivalence, coercion, inference, and low level representation.

As a result, separating symbols from type expressions and reusing the second ones will offer a better processor throughput and lower memory consumption. The *Flyweight* design pattern provides a model to avoid redundant creation of objects –in this case, type expressions. A type factory is responsible for getting and creating (when needed) type expressions of every syntactic construction, thus releasing the language processor programmer from this task.

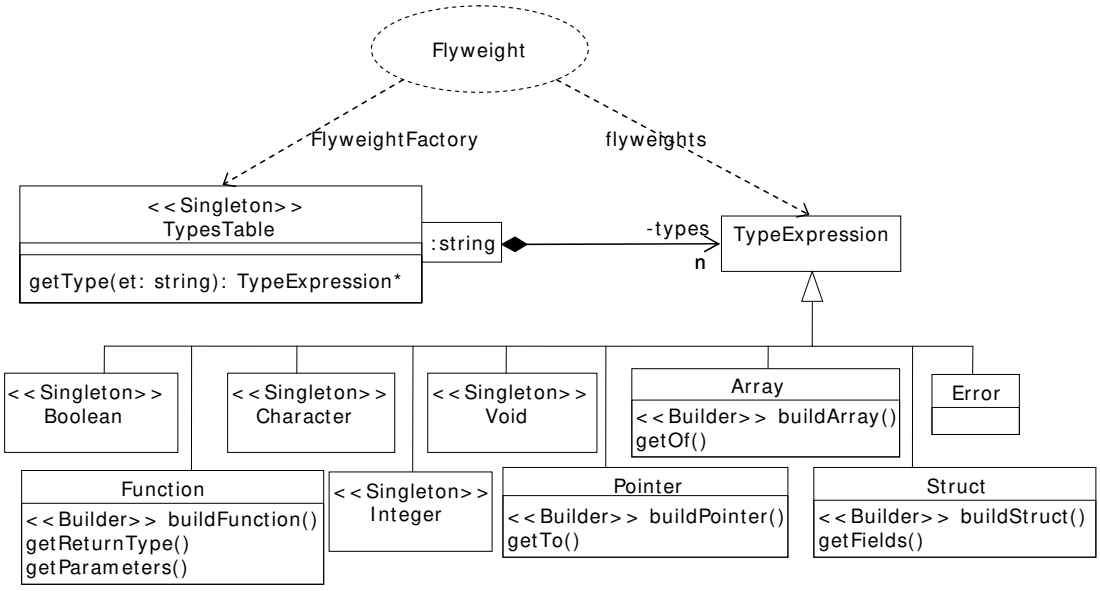

Figure 6. Adding a types table (Flyweight design pattern).

To summarize, the *TypesTable* class (Fig. 6) creates type expressions following the next criteria.

1. **Encapsulation of type access**. The *getType* method of *TypesTable* will be the only way to access a specific type. A char string that represents the type expression with a textual notation is passed to this method as a parameter. This

method is responsible for locating the object associated with each type expression and creating the necessary instance if the corresponding object has not been created before. Thus, the use of this method guarantees that no duplicate types are present, returning a pointer to the correct *TypeExpression* instance.

2. **Prohibition of creation and destruction of types**. To guarantee that no type duplication exists, the construction and destruction of type instances will not be allowed. In C++, this prohibition can be achieved by declaring the constructors and destructors of all the types as protected, and making *TypesTable* a friend class of every type. In other programming languages like Java, this implementation is easily obtained by using the package information-hiding level. In case the language requires explicit destruction of objects (as C++ does), the types table will be responsible for deleting all the types when the table is released.
3. **Heterogeneous construction of type expressions**. The process of creating the appropriate type from a char string that represents a type expression is not a trivial task. The string should be analyzed and the appropriate type constructors must be invoked to compose the recursive object structure. This process is only performed in type expression construction. Afterwards, whenever an existing type is needed, a pointer to *TypeExpression* is obtained directly by hashing the type-expression string. The creation of type expressions is performed with another design pattern called **Builder** that separates the construction of complex objects [8]. Classes that represent compound types will implement a class (static) method

responsible for building the compound types of the type expressions. For instance, the *buildFunction* method will parse the string "*(Pointer(Integer),Float)->Void*" and will construct the *Function* type expression from the types from which the function is composed, "*(Pointer(Integer),Float)*" and "*Void*". This process is a mutual recursive process, i.e., these methods will use the *getType* method that has been previously invoked, obtaining the "*Pointer(Integer)*", "*Integer*", and "*Float*" type expressions.

4. **A unique instance for each simple type**. To ensure that only one instance of each simple type is created, the **Singleton** design pattern [8] is also used (notice that composite objects do not have to be unique since they depend on their aggregate types).

By using the *Flyweight* design pattern, the object structure that represents types of the language will not be a tree-like structure with repeated structures anymore. Instead, type expressions will be represented by Directed Acyclic Graphs (DAG) that guarantee a fully shared term representation [11]. Symbols of the example program in Fig. 3 will now be represented with the DAG shown in Fig. 7.

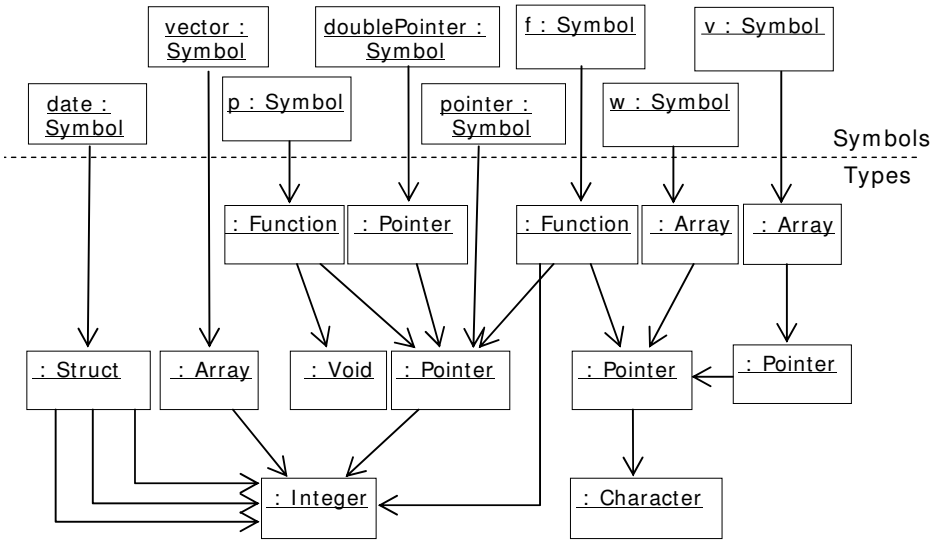

Figure 7. Type expressions object diagrams using a DAG structure.

B. Implementation of Structural Equivalence Using DAGs

Since uniqueness in type expressions is guaranteed by the design described above, the implementation of structural equivalence –the *equivalentTo* method– is strongly simplified. Now no longer need to be recursively compared type instances of the *TypeExpression* class. Object identity will provide this equivalence. In the C++ programming language is as easy as comparing object's memory addresses instead of their structures; in Java this comparison should be done between references using == instead of the *equals* method (Fig. 8). Hence, two type expressions will be equivalent if they are exactly the same object. The computing time for this process is extremely low in comparison with the time needed to compare the structure of two trees [12].

```
bool TypeExpression::equivalentTo(const TypeExpression *te) const {
        return this == te;
}
```

Figure 8. Implementation of structural equivalence using DAG structures.

V. Type Coercion

Programming languages usually allow conversions of the inferred types, either explicitly (performed by the programmer) or implicitly (performed by the compiler). Implicit conversions are also called coercion, promotion, or implicit cast. Most of the languages provide type coercion in contexts where the conversion means no information loss. For example, implicit casting from integer to real is done in C++, Pascal, Java, and C#; whereas, Modula-2 does not offer coercion at all.

Type coercion extends the definition of type equivalence in a slight way. For example, two types can be non-equivalent, but a promotion could exist so that they would become compatible. Language specifications state explicitly the places where coercion can be applied and under which circumstances. Two common examples are arguments passed to a function and the expression on the right hand side of an assignment.

A. Implementation of Type Coercion

A new *coercion* method will be placed in the root class of the hierarchy (*TypeExpression*). This message returns whether or not the implicit object type could coerce to the type passed as a parameter. The returning type expression is the coerced type, or *null* to indicate that no coercion is possible.

The *coercion* operation is implemented extending the *equivalentTo* functionality. Type equivalence will be the default implementation (Fig. 9); and if a type defines type coercion differently from type equivalence, this method should be overridden. Then, every *coercion* method of the derived type expressions will specify types to which each object can promote. As an example, an array that collects elements of type T in C++ can

be promoted to a pointer to elements of type T. Since a function defines no coercion, the default definition (type equivalence) is appropriate. Fig. 9 shows two sample implementations of different type coercions.

```
const TypeExpression *TypeExpression::coercion(
                                const TypeExpression *te) const {
    return this->equivalentTo(te)?te:0; // * Default type coercion
}
const TypeExpression *Array::coercion(const TypeExpression *te) const {
    const TypeExpression *tc=TypeExpression::coercion(te);
    if (tc) return tc; // * Type equivalence
    const Pointer *pointer=dynamic_cast<const Pointer*>(te);
    if (pointer && of->equivalent(pointer->getTo()) )
          return pointer;
    return 0;
}
```

Figure 9. Default type coercion and the specific implementation of the Array type.

Type coercion depends on the language being processed, but with this design, any modification of coercions is easily adapted from one language to another. The code that implements type coercion is placed in each *coercion* method of the hierarchy. Modifying promotion rules becomes simple and straightforward. Another typical scenario where type coercion should be defined is sub-type polymorphism. In those type systems that offer inheritance, every class must permit coercion to its super-class(es).

VI. Type Polymorphism

Universal or parametric polymorphism is a language property that permits part of a program to have different types [6]. Polymorphism occurs when a variable or a function can be defined with a set of types. By means of inheritance, many object-oriented languages provide polymorphism restricted to type hierarchies; this kind of polymorphism is commonly defined as inclusion polymorphism (or sub-type polymorphism) [13]-[14]. Object-oriented languages commonly refer to universal polymorphism as generics.

A. Type Variables and Unification

The main addition of polymorphic type systems are type variables. A type variable represents any type expression as an instance of the type variable. A type variable can be any type inside another type expression. ML, Haskell, and OCaml are functional languages that provide polymorphism; C++, Eiffel, Java 5, and C# 2.0 are object-oriented languages that implement polymorphism as well.

A language processor that provides polymorphism has to implement a unification algorithm; the process of finding a substitution for each type variable with another type expression, according to the use of a polymorphic operator or function [15]. An example is a function that receives a parameter of generic type and returns the address of the parameter. If this function is invoked with an integer as parameter, the return type should be inferred as a pointer to an integer. This inference mechanism is performed by a unification algorithm.

Only polymorphism features of the C++ and Java 5 type systems (languages known by all the students) are described here. Both are basic polymorphic type systems, but more complex type systems of languages such as ML (that require the implementation of occur check) could also be developed following the same design criteria. The code in Fig. 10 is an example of a valid input for a C++ processor.

```
1:      int vector[10];
2:      template <typename T> struct Record { T field; };
3:      template <typename T> struct List {
4:          T one;
5:          T* many; };
6:      template <typename T> T *f(T);
7:      template <typename T> T g(T,T);
8:      template <typename T1, typename T2> T2 *h(T1,T2);
9:      struct Record<bool> recordBool;
10:     struct List<int> intList;
11:     int main() {
12:         f(3);                    Pointer(Integer)
13:         g(3,'3');                Integer
```

```
14:          h(3.3,vector);          Pointer(Array(10,Integer))
15:          f<int>(true);           Pointer(Integer)
16:      }
```

Figure 10. Input program using polymorphism.

This C++ program requires a process of unification to infer types. For example, in line 14 the *h* function is invoked with two parameters; the types of both arguments should be inferred; and the resulting types must be unified to the type variables T1 and T2. Afterwards, the type checker could infer that the type returned by the function is a pointer to an array of integers (a pointer to T2). Apart from unification, type coercion is also used in lines 13 and 15.

B. Implementation of Type Polymorphism

Following the designs presented in this paper, a new kind of type expression will be added, *TypeVariable*. This class must have an attribute to indicate the name of the type variable, necessary in the unification algorithm. In addition, a *unify* method will be added to all of the elements of the hierarchy. This method receives a pair of parameters: a type expression to unify, and the list of substitutions –an association of type variables with type expressions. The unified type is returned by the method.

To implement the *unify* method in each class, it must be taken into account that

1. simple types do not hold type variables. They will simply return an invocation to the *coercion* method. Thus, this implementation will be the default behavior in *TypeExpression*.
2. compound types must check if they represent the same type constructor. If so, they will invoke the *unify* method of each component type. As a result, a type expression with the same constructor type, composed of the unifications of each child type, will be returned.

3. the type variable should find its substitution. The type variable will search for itself into the substitutions list (second parameter). If the variable is found, the unification of the type expression associated with this type variable is returned. Otherwise, the type to unify (first parameter) is assigned to the type variable and returned.

The *unify* method will be invoked during type checking every time a function is called. Type variables can also be explicitly set by the programmer (lines 9, 10, and 15 of Fig. 10).

VII. Comparison with the Traditional Approach

A. The Attribute Grammar Approach

In a typical compiler course, attribute grammars are the main formalism used to describe type checkers of programming languages [4], [16]. Attribute grammars are introduced to the students after the syntax analysis topic. An attribute grammar is a formal system that allows a user to define attributes in an augmented, context-free grammar and the relationships to be established among them [17].

Before using attribute grammars, instructors should describe to the students some topics, such as notation, inherited and synthesized attributes, well-defined (non-circular) attribute grammars, S-attributed and L-attributed grammars, and translation to imperative programs [18]. The explanation of these topics is not a trivial task because attribute grammars are declarative languages, and students are accustomed to programming in imperative languages.

The problem of the traditional approach is two-fold. First, many exercises must be performed because the students are not accustomed to programming in declarative languages; second, no tools are available to translate full-featured attribute grammars to object-oriented imperative languages [5].

Attribute grammars are high-level mechanisms to describe type features in a syntax-directed way. However, the implementation of type checkers could not be directly obtained from this formalism, involving an important drawback in a software engineering compiler construction course. As an example, if a one-pass compiler is going to be developed using yacc, a non-circular attribute grammar should be translated to an equivalent S-attributed grammar; yacc is an imperative LALR syntax-directed parser generator that only supports synthesized attributes. Afterwards, declarative rules of attribute grammars must be translated to imperative action routines in yacc.

The S-attributed grammar in Fig. 11 is an example of the type checker implemented in yacc, previously shown in Fig. 4. Following this approach, type equivalence, coercion, and polymorphism are features that are really difficult to express [4].

Grammar	Rules
$exp \rightarrow$ (exp_1)	exp.type := exp_1.type
$exp \rightarrow$ '*' exp_1	exp.type := if exp_1.type = *pointer*(*t*) then *t* else *error*
$exp \rightarrow exp_1$ [exp_2]	exp.type := if exp_1.type =*array*(*s*,*t*) and exp_2.type = integer then *t* else *error*
$exp \rightarrow exp_1$. ID	exp.type := if exp_1.type = *record*($\alpha \times (s \times t) \times \beta$) and ID.name = *s* then *t* else *error*
$exp \rightarrow$ INT_LITERAL	exp.type := *integer*
$exp \rightarrow$ CHAR_LITERAL	exp.type := *char*
$exp \rightarrow$ ID	exp.type := find(ID.name)

Figure 11. Classical example of type inference using attribute grammars [4].

B. The Design Patterns Approach

The approach presented in this paper substitutes UML design patterns to specify type systems for the high-level syntax-directed formalism of attribute grammars. Both are high-level notations, but since UML is not syntax-directed, separation of language's syntax from type checking issues is easier. Therefore, instructors can focus their explanations on type checking. Following this scheme, type checking could be performed both in a syntax-directed way (as the one pass compiler in Fig. 4) and following a multi-pass scheme by means of the *Visitor* design pattern [8]-[9].

At the higher level, design patterns facilitate the comprehension, abstraction, encapsulation, reutilization, and modularization of type checking algorithms, being directly applicable to practical compiler construction. At the lower level, example implementations of methods clarify type inference and type checking algorithms, the responsibilities of the semantic analysis phase, and how different topics of type systems are interconnected.

Other formalisms, such as typed lambda calculi and operational semantics [19], are rarely used to teach type systems in a compiler construction course. These notations are mainly oriented to design type systems rather than to implement type checkers.

VIII. Assessment

The effectiveness of the compiler course presented in this paper has been evaluated after assessing students' feedback, students' performance, and the features of the compiler developed as the final project.

A. Students Feedback

In an anonymous survey, 33 students completed the questionnaire of ten items (Table 1), using a five-point scale (where 1 = strongly agree and 5 = strongly disagree). The survey is aimed at evaluating if, in their opinion, the designs presented in this paper facilitate the comprehension of type system concepts and the implementation of a compiler, and improving their software engineering skills. The survey also asked the students if they think the practical development of a compiler applying object-oriented techniques strengthens the abilities related to other subjects.

With some variation, the majority of students agreed that the objectives of the presented approach were achieved. The average evaluation (4.27) showed that the effectiveness of this work was notably successful; the agreement evaluation of almost all the topics was over four. The students expressed that design patterns help implement a compiler and facilitate the comprehension of theoretical concepts, and provide good practice to improve their software engineering skills.

1	UML design patterns have facilitated the comprehension of the theoretical concepts	4.45
2	Object-oriented design patterns have made the development of a compiler an easy task	4.61
3	UML design patterns have been appropriated to implement the semantic analysis of the compiler	4.33
4	The practical development of a compiler has strengthened other subjects' objectives	4.03
5	Design patterns have helped produce more maintainable code	4.67
6	The development of a compiler is a good practice to improve software engineering skills	4.42
7	The development of a compiler is a good practice to improve programming skills	4.12
8	Analyzing object-oriented type system design patterns reduces the time necessary to develop a type checker	4.27
9	Interesting and significant less time explaining theoretical issues provides more time to develop a compiler	3.73
10	Learning to apply compiler lessons in the development of a complete compiler is more important than understanding and practicing formal notations	4.06
	Average:	**4.27**

Table I: Results of the survey. Strongly disagree (1), disagree (2), neutral (3), agree (4), strongly agree (5).

The only question that had an assessment lower that four (almost agree) was question nine. Probably this result came from some students' opinions that the design-level abstraction is enough to implement a compiler.

B. Students Performance

The course was originally taught using attribute grammars to describe type systems [4]. In 2003, the design patterns described in this paper were introduced in the course together with attribute grammars. This year, the students developed a single-pass compiler using yacc. In 2004, attribute grammars were suppressed from the course, developing a multi-pass compiler using the *Visitor* design pattern [8]-[9]. Table II shows the performance evolution of students in these years.

Student performance has improved over the years. In 2002, the percentage of students that passed the compiler course was 70.65%. This rate has gradually increased to 98.55% obtained in the last course. A numeric grading system with a four-point scale has been used: brilliant (4), very good (3), notable (2), pass (1) and fail (0). Average student grades have also increased from 1.8 to 2. Interestingly, students in 2003, obtained the worst marks; this was the year that both attribute grammars and design patterns were taught.

Year of Study	With attribute grammars	With design patterns	Student pass rate (%)	Average student grades (0-4)	Hours used in compiler construction
2002	√		70.65	1.82	119.22
2003	√	√	87.62	1.78	117.14
2004		√	92.65	2.07	114.6
2005		√	98.55	2.05	110

Table II: Evolution of student performance from 2002 to 2005.

The estimated number of hours for a student to develop the final project is based on the complexity of the compiler. To avoid plagiarism, the number of features students

should implement is gradually increased in each examination. Therefore, presenting the compiler in the June exams implies 90 hours, 120 hours if presented in September, and 150 hours at the end of the year. As Table II shows, hours used by students in their final project has been gradually reduced (from 119 to 110).

These results may represent that object-oriented design patterns are an adequate mechanism to teach type systems. The improvement of the pass rates and grades of students may be a result of a better comprehension of concepts. At the same time, the practical approach presented in this paper seems to facilitate the development of the final project, lowering the number of hours taken by the students. Although, these results might be also from other causes, no plagiarism between students has been detected, and no other relevant factor has been introduced in the course.

C. Qualitative Benefits

Since UML designs have been used for teaching both type system theory and implementation issues, a better use of time and a better comprehension of the concepts have been achieved. Time saved by using UML instead of attribute grammars has been employed to enhance the quality of the compiler to be developed. Therefore, in years 2005 and 2006, new features such as type coercion, type equivalence, and basic type polymorphism, were included. Moreover, the architecture was changed from old single-pass compiler to present multi-pass compiler [20]-[21], doubly improving the software engineering exercise of developing a real language processor.

IX. Conclusions

In a graduate program centered in the software engineering area, the main objective of a compiler course is the development of a language processor as a practical exercise in compiler and software construction. In the design, and verification of programming languages type systems, several formalisms were used but most of them are unknown to graduate students. When teaching type checking using object-oriented design patterns, theoretical concepts are specified as responsibilities of classes and methods. This approach facilitates the comprehension of type checking issues and the compiler development process. Designs are based on concepts, such as type expression, primitive type, type constructor, type equivalence, type coercion, and type polymorphism. These concepts do not belong to a specific paradigm and appear in most type systems.

The object-oriented design patterns presented in this paper can be implemented in any programming language that supports classes, encapsulation, inheritance, and polymorphism. They can be used both in dynamic and static type systems, with existing compiler generation tools, and with both single-pass and multi-pass compilers.

This approach has been applied to a compiler course of a graduate degree in software engineering, obtaining a gradual increase of student performance and an encouraging feedback. Besides learning traditional type systems concepts, the students implement a real type checker, and they strengthen their software engineering and programming skills. This approach seems to be a satisfactory combination of compiler theory and practice.

Acknowledgements

The development of this project has been partially funded by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation, project TIN2004-03453.

This work has been supported by the Laboratory of Object-Oriented Technologies (OOTLab) research group [22] and the Advanced Programming Languages and Technologies (APLT) subgroup [23] of the University of Oviedo (Spain).

The authors of this paper would also like to thank the anonymous reviewers for their helpful comments and Vanessa Menendez-Covelo for reviewing the final version of the article.

References

[1] IEEE-ACM, “Computing Curricula 2001 Computer Science”. 2001.

[2] W. G. Griswold, “Teaching Software Engineering in a Compiler Project Course”, Journal on Educational Resources in Computing, vol. 2, no. 4, pp. 1-18, Dec. 2002.

[3] H. Liu, “Software engineering practice in an undergraduate compiler course”, IEEE Transactions on Education, vol. 36, no. 1, pp. 104-107, Feb.1993.

[4] A. H. Aho, J. D. Ullman, and R. Sethi, Compilers: Principles, Techniques, and Tools. Reading, MA: Addison Wesley, 1985.

[5] M.L. Scott, Programming Language Pragmatics. San Francisco, CA: Morgan Kaufmann Publishers, 2000.

[6] L. Cardelli, The Computer Science and Engineering Handbook. Boca Raton, FL: CRC Press, 2004, pp. 2208-2236.

[7] B. C. Pierce, Types and Programming Languages. Cambridge, MA: MIT Press, 2002.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison Wesley, 1995.

[9] D. Watt and D. Brown, Programming Language Processors in Java: Compilers and Interpreters. New York, NY: Prentice Hall, 2000.

[10] J. Welsh, M. J. Sneeringer, C. A. R. Hoare, “Ambiguities and Insecurities in Pascal”, Software Practice and Experience, vol. 7, no. 6, pp. 685-696, Nov. 1977.

[11] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier, “Compiling language definitions: the ASF+SDF compiler”, Transactions on Programming Languages and Systems, vol. 24, no. 4, pp. 334-368, July 2002.

[12] F. Baader and T. Nipkow, Term Rewriting and All That. New York, NY: Cambridge University Press, 1998.

[13] C. Strachey, “Fundamental concepts in programming languages”, Higher-Order and Symbolic Computation, vol. 13, no. 1-2, pp. 11-49, April 2000.

[14] A. Eliens, Principles of Object-Oriented Software Development, 2nd ed. Reading, MA: Addison Wesley, 2000.

[15] R. Milner, “A Theory of Type Polymorphism in Programming”, Journal of Computer and System Sciences, vol. 17, no. 3, pp. 348-375, Dec. 1978.

[16] C. N. Fischer and R. J. LeBlanc, Crafting a Compiler. Menlo Park, CA: Benjamin-Cummings, 1988.

[17] D. E. Knuth, “Semantics of context-free languages”, Mathematical Systems Theory, vol. 2, no. 2, pp. 127-145, 1968.

[18] J. Paakki, “Attribute grammar paradigms –a high-level methodology in language implementation”, ACM Computing Surveys, vol. 27, no. 2, pp. 196-255, June 1995.

[19] A. M. Pitts, “Operational Semantics and Program Equivalence”, Lecture Notes in Computer Science, vol. 2395, pp. 378-412, Sep. 2000.

[20] A. W. Appel, Modern Compiler Implementation in Java. New York, NY: Cambridge University Press, 1997.

[21] D. Galles, Modern Compiler Design. Reading, MA: Addison Wesley, 2005.

[22] J. M. Cueva, Object-Oriented Technologies Laboratory Research Group. [Online]. Available: http://www.ootlab.uniovi.es, Mar. 2007.

[23] J. E. Labra, A. A. Juan, R. Izquierdo, F. Ortin, J. M. Cueva, and M. C. Luengo, Advanced Programming Languages and Technologies. [Online]. Available: http://www.di.uniovi.es/~labra/APLT, Mar. 2007.