Matthew Aguirre
Independent Study Report:
Flexible Design of Interpreters

Fall 2001

# 1. Introduction

The purpose of this research is to develop a flexible interpreter for testing purposes. The overall goal of the independent study is to further the ability for the existing interpreter to handle the addition of new data types. Most interpreters have built in data types which it knows how to use and handle, but the addition of user-defined types to an interpreter has three significant difficulties: dealing with new operations on the type(s), coping with implicit type conversions, and handling literal values for the new type. This research addresses these issues by handling new operations provided on new types, type checking, type conversions, and literal values on user-defined types.

All code for this project is written in C++ for MSVC 6.0 and 7.0 Beta 2. ANTLR 2.7.1 using Java 1.3 became an essential component of the interpreter. The Token, lexicalAnalyzer, and Parser classes were replaced by the use of ANTLR's code that is generated by grammar files created during the research mentioned above. This code is now used to parse and validate the grammar of a test script and a tree walker is used to execute the script as stored in the parse tree.

# 2. TypeManager

A new class, TypeManager, was added to the system, which adds to the ease of handling and creating data types within the interpreter itself. The TypeManager keeps an instance of each of the data representations available to the system through a registration method provided in the interface. Conversion information is also registered upon the first request for a conversion. This information is stored in an adjacency matrix. The index values for the matrix can be used to gain access to the data types by referencing a vector that holds correlations between index values and string identifiers for types.

Public methods in the TypeManager allow for converting, testing for type compatibility, testing for type existence, and type instantiation. The TypeManager is used to instantiate all objects and to convert one type to another, where applicable. Many

of these methods are called through corresponding methods in the intepreterValue class interface. The interpreter uses interpreterValue objects to represent data objects during execution of interpreted script; that is, interpreterValue objects contain data representations such as integers and reals. Conversion of data types is used within the *CopyFrom*() method, and instantiation methods are accessed within the constructors for interpreterValue objects. Registration of data types is called during static construction of exemplars of each data representation. During matrix initialization, each data representation is polled for its conversion information. Interaction between objects is shown in Figure 1. The interfaces for TypeManager and interpreterValue are shown in Figures 2 and 3 respectfully.

The TypeManager class is a key component in adding the ability for the interpreter to do type casting while still enabling new types to be added to the interpreter. The TypeManager class incorporates the pre-existing idea of maintaining a single instance of each data type with the idea of separation and encapsulation. The single instantiation of each data representation is separated from the abstract representation class and stored and encapsulated in the TypeManager class. The idea of registering a data type and its properties within the system was implemented for adding new types to the interpreter and storing information about these types. Information about each type includes descriptions of possible conversions, an archetype of each data type, and whether a new type has been added. Instantiation of data types is done through methods that allow the ability to clone, create with a value, create with default value, and create from a literal.

## *2.1. Data Type Storage*

Storage of the data representations within the TypeManager is done with a partial map object. For additional lookup purposes, a vector of type names is also used. This partial map object stores MetaData objects. A MetaData object contains fields for the archetype of each data representation, an index value for indexing into the vector of type names and into an adjacency matrix, and a flag to denote whether the representation's conversions have been registered. Registration of the data type with the TypeManager occurs during a single static initialization of each type of data representation; the abstract

base type registers the *this* pointer and the name of each designated type. A figure of the layout of the TypeManager class is provided as Figure 4.

The registration method in the TypeManager class permits the registration of a type that does not yet have a representation; this allows a data type to register a conversion to a new data type that does not exist in the interpreter. This enables the data types to not rely on the existence of other existing data types, only on the possible existence of the data type. This decision lets additional types be defined and registered at run-time.

An adjacency matrix is used to store information about conversions of data types. A conversion is an object that is instantiated and stored in a list by the exemplar constructor of each data type. The conversion object consists of *from_type*, *to_type*, and *classification* fields. The *from_type* is the type from which to convert, the *to_type* is the desired type to have as a result of converting, and the *classification* is the type of conversion. Currently, the only classifications supported by the interpreter are *widening* and *narrowing*.

Each type of conversion has an associated weight in which to reflect the classification and to ensure a "shortest path" between conversions. This shortest path between conversions is used to guarantee the use of widening conversions whenever possible (but not more than needed) and to use narrowing conversions only when necessary. To do this, the weight of a widening conversion is designated as "1" and the weight of a narrowing conversion is designated as "n + 1," where n is the current number of data representations defined in the interpreter. These values are updated in a private method *update_weights()*. This method resets all of the weight values as stored in a vector; constant values are used to index into the vector at the desired indices (i.e., *NarrowingIndex* and *WideningIndex*). This allows for the easy addition of new types of conversions into the interpreter.

## 2.2. Conversions

Conversion between data types is performed using an adjacency matrix. The interpreter has a lazy approach to generating the shortest path matrix: the shortest path information is generated from the matrix the first time that a conversion is requested.

This decision also allows for new data types to be introduced at run-time such that if a change is made in the matrix by adding a new data representation, the shortest path is then re-calculated for all types to account for the new data type.

The MetaData object associated with each data type has a flag that is marked if its conversions have been registered and stored in the matrix. As the matrix is being initialized for the first time (or because a new type has been added), this flag is checked. If the type has had its conversions stored, then it is skipped; otherwise the list of conversions is retrieved from the representation and lookups are performed to find the *to_type* as mentioned above. The *to_type* is then used to lookup the MetaData of the representation that is stored in the partial map. The index value that is stored in the MetaData object corresponds to its column in the adjacency matrix. This entry in the adjacency matrix is marked as allowing a conversion between the two types. The cost is initialized according to the classification of the conversion, and the path initialized to show the conversion. This path is later used to easily concatenate paths together to show that other conversions are possible other than those explicitly stated by each data type.

To calculate the shortest conversion path between all data types in the adjacency matrix, Floyd's Algorithm was used to calculate and store the shortest conversion path from each data type to another (Figure 5). An infinite, or impossible conversion, is represented as an empty path with a cost of zero. As conversions are requested by the interpreter, the conversion path is "traversed." The data type is converted from one type to another until the desired type is reached; an error data type is returned if there is no possible conversion.

## *2.3. Data Type Instantiation.*

All instantiations of data types are done through the TypeManager. The desired type is sent as text (i.e., "Integer" and "Real") as an argument to the *make* method of the TypeManager. This is done in the interpreterValue constructor, which takes as an argument the data type. Within this constructor, the TypeManager is referenced and the *make* method, which takes the data type as an argument, is called. This method references into the Type_Map object to retrieve the appropriate MetaData object. The

appropriate *make* method is then called on the desired interpreterValueRep object to return a newly instantiated interpreterValueRep.

An additional make method allows the ability to pass in an initial value. Literal values are also used to create data types. If the interpreter encounters the value "42" or "3.14," it must be able to interpret these values correctly and create the appropriate data types to represent the data (such as Integer or Real). This is done by using a *create_from_literal* method that takes the Text representation of the literal as an argument. The TypeManager then calls the *create_from_literal* method on each data representation class. If the data type can interpret and create correctly a value from the Text representation, then a new object is instantiated with the value that was passed in. Otherwise, the TypeManager keeps looking to see if it can find a type that can interpret the literal value.

Another create_from_literal method was added which takes a TokenBuffer. This method of creating literals supports creating literals of any data type in the interpreter.

## 3. Supporting New Operations

One of the issues that must be faced as a result of adding new types to a system is that of supporting new operations. Support for new operations in the interpreter was added by having all mathematical and logical operations (except assignment) use a single method, *callMethod*, within each data representation class. The use of the *callMethod* method provides for easy addition of new operations to be added to user defined types.

Along with the *callMethod* method, a new method named *method_signature* was added. This method returns a list of formal parameters expected by the object when a particular method is called. With these two methods in place, all mathematical and logical operations are now performed through the *CallMethod* method in interpreterValue. Any new methods for specific data types are also handled through these two methods. An example of this is the One_Way_List. This data type supports *Add_Right*, *Advance*, and other methods used to access data. These methods are "declared" and "described" in *method_signature*. Declaration of the method means that if a call is made to this method, it will be recognized. Description of the method involves creating a list of the formal parameters of the method.

A problem arises by using method_signature to create a list of formal parameters expected by the method call: some methods allow the use of any type as an argument (i.e., operator << for ostream). To give this flexibility to an object, a new method was added to TypeManager that returns a text value "*Any_Type*" for the formal parameter. This allows for any data type representation to be passed as a parameter where the parameter formal type is "*Any_Type*".

Since all operations on a data representation are performed through *callMethod*, knowledge of operations is separated from the interpreter. This provides for new operations to be easily added without changing the behavior of the interpreter or requiring changes to the parsing and execution engine of the interpreter.

# 4. Type Checking

Previous to the work done on the interpreter, a method call was required to have the correct type passed into it as an argument. The reason for this was because of the inability for the interpreter to perform implicit casts so that compatible types could be used in place of the formal required types. Originally in the interpreter, many of the built-in operators for types were overloaded by the interpreterValue class. In turn, each of these methods would invoke the appropriate operation on the interpreterValueRep object that is held internal to it. This was changed so that most operations are invoked by use of the *callMethod* in the interpreterValue and interpreterValueRep classes. The assignment operator was left to allow an interpreterValue to be assigned the value of another interpreterValue. This assignment does not copy the contents of the internal representations, but instead replaces it with whatever is on the right hand side of the assignment. In order to perform an assignment to the representation, the method CopyFrom in the interpreterValue class is called.

To allow for conversions as necessary, a check for type compatibility before the method call is executed is performed. To aid in this, the addition of a *method_signature* method was added to each representation. This method returns information such as the formal argument list for the desired method and whether or not the method is defined. If the method is not defined, an error is returned without entering execution of the method. If the method is defined, the formal argument list is then compared to the actual argument

list used in the method call. Each actual argument is compared to its corresponding formal argument to ensure type compatibility. If type compatibility is not found, an error is returned stating this. As stated in the previous section, some methods can take any data type representation. To represent this, the TypeManager has a value "*Any_Type*" to act as a placeholder in the parameter list. When the test for type compatibility is made between the actual parameter and "*Any_Type*," the TypeManager returns that the conversion is possible.

One catch that has occurred though the use of method calls in this fashion is that the object on the "left" of a binary operation is the data type that has "control." That is, if a statement is Integer plus Real, the result will be an Integer, so any value that is stored by the Real is truncated during conversion for use in the addition method. If the statement had been Real plus Integer, the Integer value would be converted to a Real and the sum would be the correct, un-truncated value.

## 5. Implicit/Explicit Casting

Implicit and explicit casting abilities were added to the interpreter. An implicit cast is a cast that is performed when a specific data type is needed that is not the current data type. The interpreter will then try to cast the current type to the desired type. This is done by the interpreter in one place: the *CopyFrom* method in the intepreterValue class. An example of this is if the user wishes to convert a Real into an Integer. First a new Integer type is created by use of the *make* method provided in the TypeManager class. Then the *CopyFrom* method on the newly instantiated interpreterValue object which holds the Integer representation is called with the interpreterValue containing the Real as an argument.

Internal to this method, a test is made to make sure that the "from type" can be converted to the "to type" (Real and Integer, respectfully, in the example). If the conversion is possible, then the call to the TypeManager to perform the conversion is made and a new interpreterValueRep of the desired to_type is returned. The current interpreterValueRep held by the target interpreterValue object is deleted and the pointer is assigned to the newly returned representation. Implicit casting is performed by the interpreter during method execution and the assignment operand.

Adding explicit casting required the modification of the EBNF to support a place where casting statements are to be done. Following the Java 1.3 grammar specification, this cast statement was made the lowest priority unary operation available and was inserted just after multiplicative operations (the highest order binary operation). This new EBNF is depicted in Figure 6. Slight modifications to the ANTLR grammar were made so that the type cast was the lowest order unary operation. The most current grammar for Java that is provided for ANTLR places the postfix expression at a lower precedence than type cast. An explicit cast uses the same method for performing the implicit type conversion.

# 6. Literal Handling

Literal handling by the parser works with all data types introduced into the system. Integer and Real literals were already in place by the parser and lexical analyzer for the system. A problem arises when a programmer wishes to have a literal for a newly defined type such as a One_Way_List. This data type is a list of integers and has no built-in literal handling by the parser.

The first problem presented is how to make the parser know when a literal is being encountered. For now, the decision was made to mark a literal in the following fashion, "data_type(literal_information)." Here, data_type is a type that is recognized by the TypeManager and the literal information is in the form of Tokens that can be in any format. This is valid so long as the data type knows how to handle the tokens with which it is presented.

This method of representing literals is syntactically similar to that of a method call. For this reason, the literal handling is done at the same priority level as a method call. To denote the difference between a method call and a literal, the identifier is checked. If the identifier is a recognized data type in TypeManager, it is a literal; otherwise, it is a method. Since the method call and literal handler use the same rule within the parser, they must also both call argList(). A flag is set that a literal is being processed once the test to TypeManager is complete. In *argList*, when this flag is checked, the TokenBuffer is retrieved from the parser and is passed to the constructor of interpreterValue along with the data type.

In the constructor for interpreterValue that takes the TokenBuffer, the TypeManager's *create_from_literal* method which takes a TokenBuffer is called. This method then references the MetaData object within the Partial Map and calls the *make* method of the proper data representation which takes a TokenBuffer. This is where the individual representation must know how to read and parse the data from the token stream. An example of this is for One_Way_List where a valid literal is OWList(1 2 3 4). When the One_Way_List representation receives the token stream, it uses the LT() method to search for ")" and until this value is found, the value is assumed to be an integer and it is added to the list. Each token on the stream is consumed until the ")" token is found. With this, the new interpreterValueRep pointer is returned and the interpreterValue object has been created and initialized to the literal value.

Now that the object has been created, the desired effect is to place this value into the Abstract Source Tree (AST) so that when it is traversed in the tree walker, the literal value can be read straight from the tree. To do this, a new AST derived class, interpreterAST, was created. This derived class overloads all required methods to ensure correct operation. In most cases, the methods are empty or return some value immediately. A new constructor which takes an interpreterValue is added along with a method to return the interpreterValue. After the literal has been evaluated and stored in the interpreterValue, an interpreterAST object is instantiated with this new interpreterValue. This object is then cast to a RefAST object and is placed into a virtual node in the tree. Now in the tree walker, the only thing needed to do to retrieve the value is to take the first child of the LITERAL_VALUE token which is stored in the tree. Casting and calling the *getValue* method on the child returns the stored interprterValue that was created from the literal.

# 7. ANTLR

ANTLR was used to replace the lexicalAnalyzer, Token, and Parser classes in the interpreter. ANTLR is a tool that is used to parse a source file to generate an AST. These trees are generated by following rules as set in a grammar file. These rules are used to reflect a grammar that is shown by the EBNF in Figure 6. The grammar file for

generating the AST is a heavily modified and stripped down version of the public domain Java 1.3 grammar available on ANTLR's web site.

A heavily stripped down and modified version of the Java tree walker grammar was also used. This is the "engine" that drives the execution of the interpreter code. The previous use of recursive descent parsing required that during the execution of loops, once the condition to execute the loop became false, the parser would still have to parse through the tokens in the lexicalAnalyzer to get to the next point of execution after the loop. This required a flag to be passed to each method of the parser to determine whether or not to execute the code. The use of ANTLR's AST remedies this problem by using trees and nodes to store the code to be executed. A *while* statement is a branch of a tree in the interpreter where "while" becomes the root node of this branch. If the condition for executing the body of the loop is false, then the node is skipped along with all execution. The root node becomes the marker that denotes the beginning of execution, and to execute the body of the loop, the tree is simply traversed. *If-else* statements are handled in the same way, where if a condition is true, the *then* portion of the statement is traversed, otherwise the *else* portion is traversed. If the *else* clause does not exist, execution continues with the next statement.

The tradeoff of using this tree walker is that all code for the interpreter must be placed inside of this tree walker grammar file.

# 8. Conclusion

The work that was done on the interpreter has added new features and flexibility to the interpreter. The ability to explicitly and implicitly cast was added which allowed the ability to perform assignments between data types. This casting was then used to expand the mathematical/logical/method operations performed by the interpreter. Since all logical and mathematical operations were changed to use the *callMethod* method of a data representation, the ability to do checking of a method signature for compatible types and perform type conversions was added. This added the ability for mixed mode expressions in mathematical operations. The use of the *callMethod* method also provides for easy addition of new operations to user defined types. Since all operations on a data representation are performed through *callMethod*, knowledge of operations is separated

from the interpreter and new operations may be easily added without the consequence of creating changes in the behavior of the interpreter. Additional literal handling was added. A literal of any type, including new user-defined types, may now be created within the interpreter. An example of this is a One_Way_List which is a list of integers and has no built-in literal handler in the parser. This list may now be created as a literal in the interpreter by the following statement: "OWList(1 2 3 4);." This statement creates a new One_Way_List with the values 1, 2, 3, and 4.

Although the issues that arise from adding new types to an interpreter have been addressed, there is still work to be done. Future work to be completed includes changing the literal handling to have more of the work handled in the parser and tree-walker. This is because the current implementation is "clunky" since the user must specify the literal type each time a literal is created. A more desirable scheme would be to use a follow-set to determine when literals are found in the token stream. In addition, possible changes to the mathematical operations for mixed mode expressions should be studied. As of now, the data type on the left hand side of an operation has "control" of the operation. For example, consider the expression 1 + 3.14. In this example, 1 is an integer and when the binary addition method is invoked, the 3.14 real value is cast to an integer. The result of the expression is 4. This may be acceptable if this value is being assigned to an integer, but if the value is a float, the result is being truncated.

Figure 1: Interfaces and object interaction

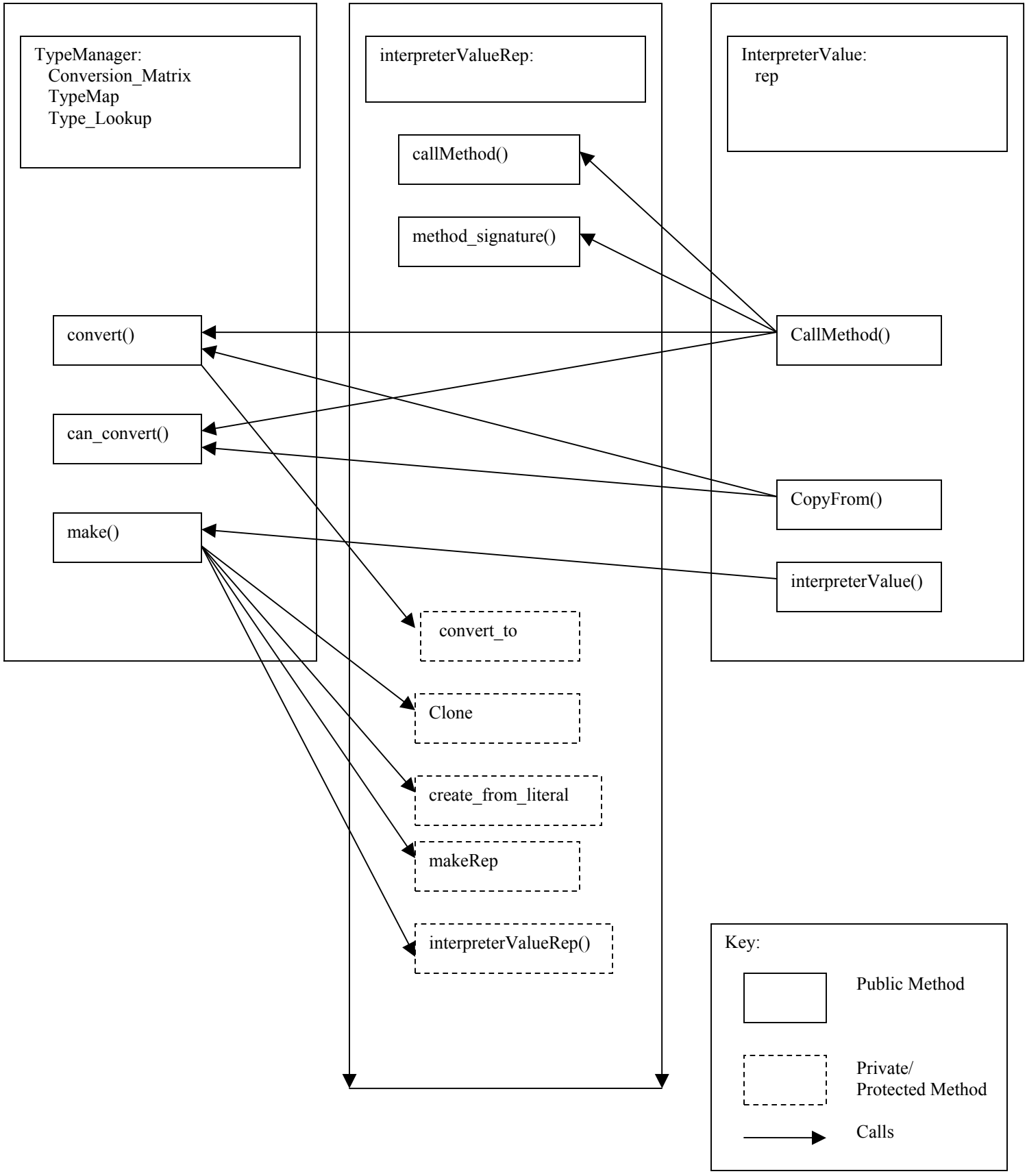

Figure 2: TypeManager Interface

```
class TypeManager {
public:
        typedef Text TypeIdentifier;
        ~TypeManager();

        interpreterValueRep* convert(TypeManager::TypeIdentifier to_type, interpreterValueRep* from_type);
        bool can_convert(TypeIdentifier to_type, interpreterValueRep* from_type);

        void register_type(TypeManager::TypeIdentifier type_name, interpreterValueRep* archetype);
        bool type_exists(TypeManager::TypeIdentifier type);

        //Create new interpreterRep objects of the desired types/values
        interpreterValueRep* make(TypeManager::TypeIdentifier type);
        interpreterValueRep* make(TypeManager::TypeIdentifier type, void* init_value);
        interpreterValueRep* make(TypeManager::TypeIdentifier type, interpreterValueRep* init_value);
        interpreterValueRep* make_error(Text message, int line = -1, int col = -1);

        // Create a new interpreter ValueRep object by parsing a user-
        // defined literal representation:
        interpreterValueRep* create_from_literal(Text& literal);
        interpreterValueRep* create_from_literal(TypeManager::TypeIdentifier type,
                ANTLR_USE_NAMESPACE(antlr) TokenBuffer* stream);

        //should be the only way that a TypeManager can be accessed
        static TypeManager& getInstance();
        static TypeManager::TypeIdentifier anyType();
};
```

Figure 3: interpreterValue interface

```
class interpreterValue
{
public:
        // constructors
        interpreterValue (TypeManager::TypeIdentifier type_name = "Void");
        interpreterValue (Text& literal, Create_Action how_to);
        interpreterValue (Text type, ANTLR_USE_NAMESPACE(antlr) TokenBuffer* stream);
        interpreterValue (Text message, int line, int col);
        ~interpreterValue ();

        bool isError ();
        void Put_To (std::ostream& stream);
        interpreterValue callMethod (Text name, List_of_Values& params);
        void method_signature(Text name, list<TypeManager::TypeIdentifier>& formal, bool& valid);

        // change type and value
        void resetAsError  (Text message, int line = -1, int col = -1);
        void resetAsVoid ();
        bool isBoolCompatible ();
        operator bool ();

        // Swap operator
        void operator &= (interpreterValue& rhs);

        void operator = (interpreterValue& rhs);
        void CopyFrom (interpreterValue& rhs);
        interpreterValue Clone ();
        interpreterValue (const interpreterValue& rhs);

        Text getType();
};
```

Figure 4: Detailed TypeManager

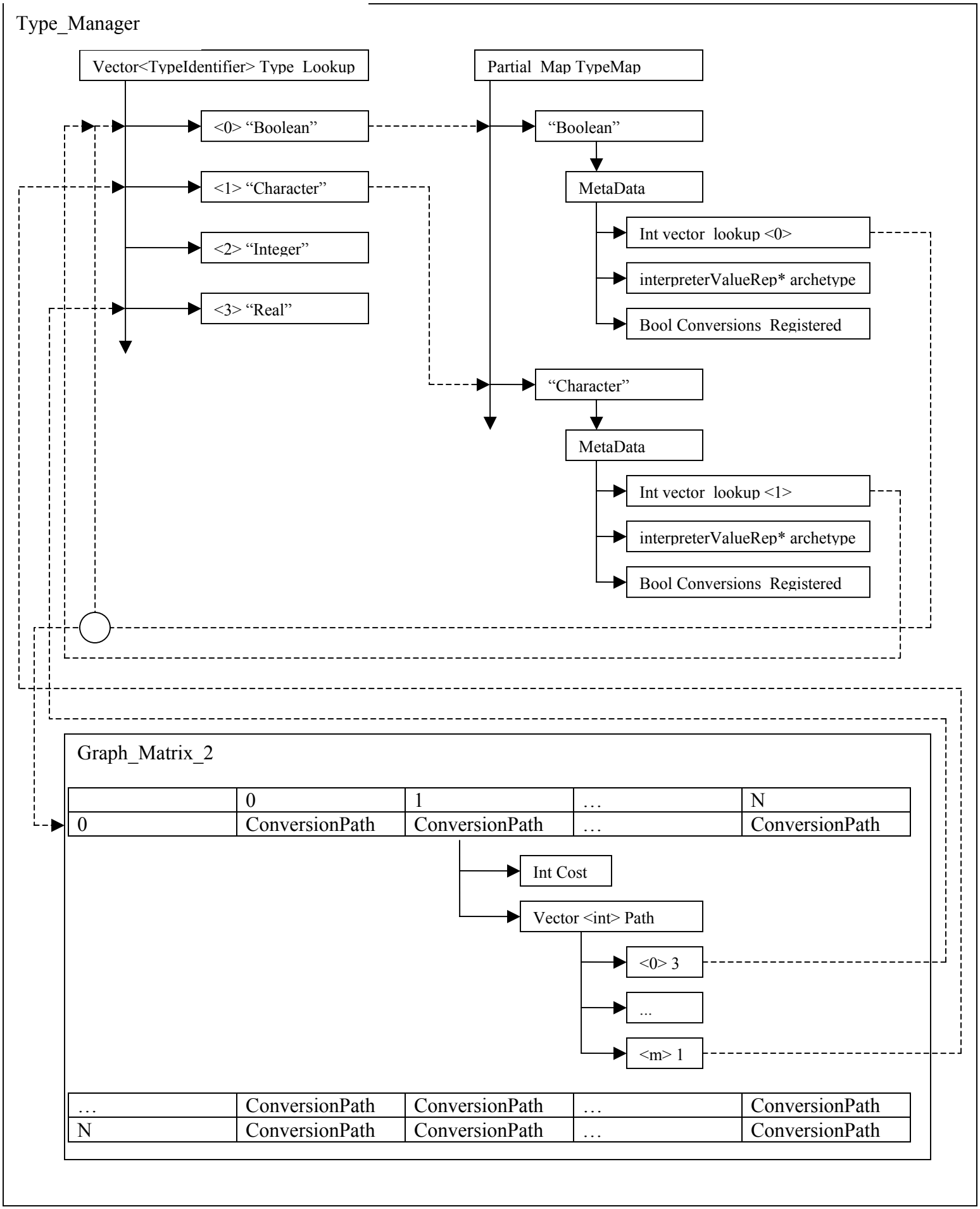

Figure 5: Floyd's Algorithm

```
for(int k = 0; k < loop_limit; k++)
        for(int i = 0; i < loop_limit; i++)
                for(int j = 0; j < loop_limit; j++)
                        if(GraphMatrix_2[i][j] > GraphMatrix_2[i][k] + GraphMatrix_2[k][j])
                                GraphMatrix_2[i][j] = GraphMatrix_2[i][k] + GraphMatrix_2[k][j];
```

Figure 6: EBNF

```
<file> ::= <comment>+ | <block>+
<comment> ::= <commentline> | <commentblock>
<block> ::= { <statement>+ }
<commentLine> ::= // (A-Za-z0-9\s)+ \n
<commentBlock> ::= /* (A-Za-z0-9\s\n)+ */
<statement> ::= <declaration> | <ifstmnt> | <whilestmnt> | <block> | <expression> ;  | ;
<declaration> ::= <varType> <declList> ;
<declList> ::= identifier [ <initialization> ]
                | identifier [ <initialization>] , <declList>
<initialization> ::= = <expression>
<ifstmnt> ::= if ( <expression> ) <statement> [ <elsestmnt> ]
<elsestmnt> ::= else <statement>
<whilestmnt> ::= while ( <expression> ) <statement>
<varType> ::= int | char | string | float | boolean | identifier
<expression> ::= <assignmentExpression>
<assignmentExpression> ::= <conditionalExpression>
                            ( ( = | += | -= | *= | /= | %= ) <assignmentExpression>)?
<conditionalExpression> ::= <logicalOrExpression>
<logicalOrExpression> ::= <logicalAndExpression> ( || <logicalAndExpression>)*
<logicalAndExpression> ::= <inclusiveOrExpression> ( && <inclusiveOrExpression>)*
<equalityExpression> ::= <relationalExpression> (( != | == ) <relationalExpression>)*
<relationalExpression> ::= <shiftExpression> ( ( < | > | <= | >= ) <shiftExpression>)*
<shiftExpression> ::= <additiveExpression> /* (( <<  | >> |  >>> ) <additiveExpression>)* */
<additiveExpression> ::= <multiplicativeExpression> (( + | - ) <multiplicativeExpression>)*
<multiplicativeExpression> ::= <castExpression> (( * | / | % ) <castExpression>)*
<castExpresssion>::= '('varType')' <unaryExpression>
<unaryExpression> ::=   ++ <unaryExpression>
                    |    --  <unaryExpression>
                    |    -    <unaryExpression>
                    |    +   <unaryExpression>
                    |      <unaryExpressionNotPlusMinus>
<unaryExpressionNotPlusMinus> ::= ~ <unaryExpression>
                                  |  !  <unaryExpression>
                                  |    <postfixExpression>
<postfixExpression> ::=  <primaryExpression> [ ++ | -- ]
<primaryExpression> ::= identifier | <invocation> | '(' <expression> ')' | varType'('PCDATA')'
<invocation> ::= identifier . identifier ( [ <parameterList> ] )
<parameterList> ::=  identifier
                   |  identifier , <parameterList>
```