TOIL a Threaded Object Interpreted Language

Introduction

This paper describes the development of a design for an object oriented language which uses a threaded interpreter model, similar to Forth, and which supports multiple inheritance.

The design grew out of a small scripting language that was developed for the first attempt at the SASSY project (Software Architecture Support System). The initial aim was to re-factor the SASSY code to make it more modular, and separating out the script interpreter was the first step. Once that task was completed the option of extending the scripting language into a more complete general purpose language needed to be investigated.

The TOIL language is very simple, and should be viewed as a sort of symbolic assembler language. Sophisticated language constructs should be implemented in a high level language and a compiler should then generate the TOIL symbolic code. A future project will develop such a language and compiler.

Background

The SASSY project uses a small scripting language to access an ontology knowledge database and assemble the information into a software architecture document.

The original design was quite simple. It used a collection of function objects to perform the underlying operations, such as accessing the database or manipulating the stack or adding a paragraph to the document. A data stack provides temporary storage, and a return address stack is used to keep track of return addresses from function calls. The code was simply a table of numbers, each of which was either the index of a function object or the index of a function within the code, or a number which could be used as a jump offset for the function objects that performed that operation. The interpreter would be given the index of the starting function and would process the function objects until it returned from the first function. This was an entirely acceptable design in the context of that version of SASSY.

The first refinement was to remove the code table from the interpreter and put the code for each function into an “exec” function object. This meant the interpreter just had a list of function objects rather than the combined structure in the original design (where the values in the code table had to be interpreted as either being the address of a function object or a location in the code table). In this design the return address stack was no longer required as it used the C++ stack.

The second refinement was to re-introduce an explicit program counter and return address stack so that the interpreter could be single stepped, one instruction at a time. This was done in anticipation of merging the interpreter with a previously written scheduler (not a part of the current project).

This version was then built as a shared library which can be used by any program that requires scripting capability.

The scripting version relies on the containing program to manage the data objects. This is fine for a scripting language, but for a full language we also need to have structured data. If we are going to have data structures, we might as well have objects, and if we have objects it would be nice to have multiple inheritance – otherwise we are just rehashing a multitude of other interpreted languages for little benefit.

This is how the ideas for TOIL were arrived at.

Data Objects

A scripting language needs a means of handling a variety of data object types in a consistent manner. The design chosen for TOIL (derived from the design used in SASSY) is to have reference counted smart pointers and wrapper objects as shown in the following diagram.

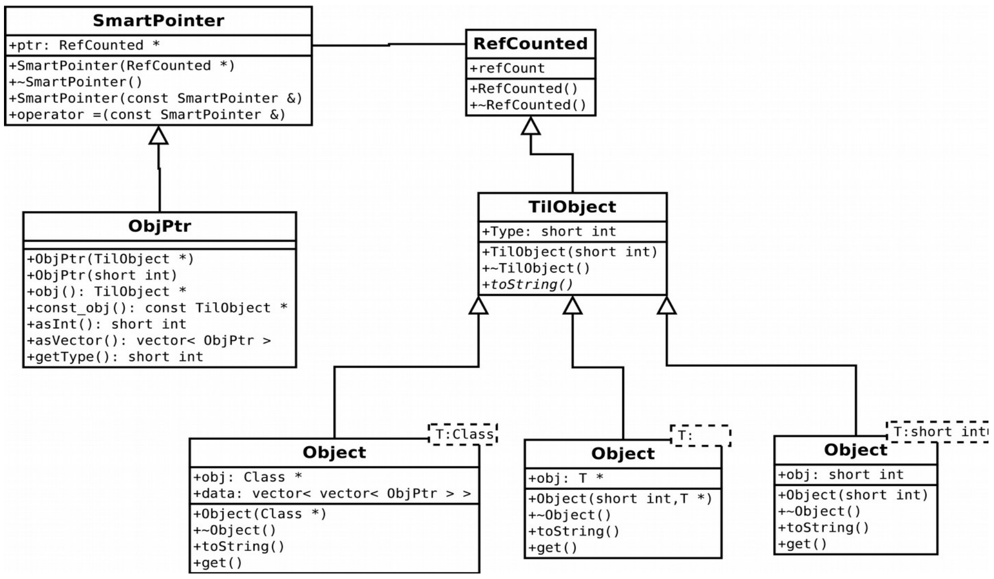

Figure 1: Data Classes

Object is a template class which manages an instance of whatever classes the program needs. A specialisation for short int avoids having to have a separate object for such a small data item, and another for the interpreter's Class class includes the referenced data for the class as a vector of ObjPtrs.

The problem with using a reference counted design is that if a loop of objects should be formed then they are never removed from the system, causing a memory leak. Later work will include an Object Reference class that is similar to an ObjPtr but does not contribute toward the reference count. These can be used to access objects without the danger of introducing loops. Since the recent C++ standard includes shared pointers and weak pointers these may replace some of these classes.

Object Oriented

Whereas the original design used the base of the data stack for any long term storage, or constant values, we now need to keep these values in objects, and references to data items will mean references to the current object.

To manage this we introduce the object stack. When we make a call into an object we move it from the data stack on to the object stack and start a new section in the return address stack. The return address stack is therefore implemented as a vector of vector of addresses. When the last value is popped off and the vector becomes empty it is popped off, and so is the top of the object stack.

Load and save operations are used to move data between the object on top of the object stack to the data stack for use in calculations etc. A “new” operator (function object) creates a new instance of an object on the data stack. It is up to the programmer to write and call the code that initialises the object, otherwise it will have the default values defined for the class. A “this” operator copies the current object back on to the data stack so that an object can refer to itself.

The TOIL system does not include any global data or functions. All data is held in objects.

Multiple Inheritance

A defining feature of TOIL is that it should include multiple inheritance – a class should be able to include more than one parent class.

The main difficulty implementing multiple inheritance is that the offsets calculated for a parent class may not be correct when the parent is added to a child class. Normally the compiler will generate an offset for each function and data item relative to the base of the class. It is then a simple matter of adding the offset to the address of the object to get the desired location. For single inheritance, the derived class just has its functions and data added on top of the pile, and all the previous offsets remain valid. However this does not work for multiple inheritance since the second, and subsequent parents are added on top of the others, invalidating the offsets.

The solution used by TOIL is to replace the absolute offsets with relative offsets. A parent class can then continue to use its offsets, no matter where it lies in the overall class structure. To make this work it is necessary to remember where each function was when it was called, and for this we use what we sometimes call the "Where The Hell Are We" stack. The offset is pushed on as we call a function object, and popped off again on returning. The initial value of the offset for each function object is set when it is loaded into the class.

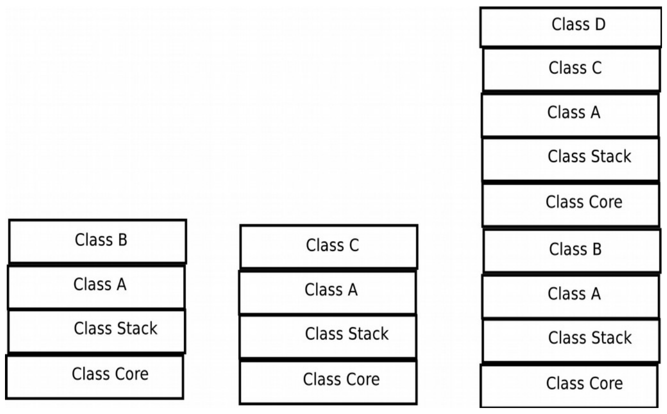

Figure 2: Class Stacks

Parent Classes

The functions objects of a class are held in a single vector within a class object. When a class is created the vectors for each parent (and their parents) are loaded first, and then a new vector added for the new class. We thus have our functions in a vector of vectors of functions, and hence the addresses are given by a pair of offsets – the addresses within TOIL are thus two dimensional relative addresses.

A similar structure is used for the data attributes of each class and object.

Diamond Inheritance

The other difficulty introduced by multiple inheritance is the often called the "diamond inheritance" problem, based on the shape of the inheritance graph. A class A can be the parent of two other classes, B and C. If class D then inherits from both B and C there are then two instances of A (and its parents) included. This may be a problem, but not necessarily – it all depends on what A does, and therefore there can be no automatic solution.

TOIL addresses this issue by allowing attributes of a class to be marked as “private”. Since all the attributes are ObjPtrs it is easy enough to merge the duplicates into a single object which is done automatically by the “new” operator, unless it is marked as private.

This scheme works for the data values. For functions there is no harm in having multiple references to the same function as they are effectively identical (unless they have private data). If you needed to access the two versions of a method in class A you would have to do it indirectly via functions in B and C. Care needs to be taken that methods in the parent classes have sufficiently distinct names. (It is expected that a compiler would generate names that included some form of name mangling that included the class name.)

Polymorphism

If a class redefines a function previously defined in one of its parent classes then the function replaces the parents copy within its own class stack. (*The current implementation is broken if there is more than one instance of the function in the class stack.)*

A call to a function in another object (on the data stack) must state what class it is expecting to find. The actual class of the object then provides the location of the expected class in the class stack for the object (or an error if its not a base class for the object).

Data Types

The TOIL system has an extensible data typing system. There are some core types, such as short (16 bit) integers, strings, logical (boolean), a class type, and some container types – vector, and map. The type system can be extended by the plug-in libraries (see below) to include new fundamental types (for example a complex number type).

Each type must have a unique name and numeric id, and may have function objects that implement the following operations:

- toString – converts a value of the type into a string representation;
- fromString – converts a string representation into a value of the type;
- less – returns true if the first item is less than the second;
- equal – returns true if they have the same value; and
- copy – returns a duplicate of the item.

In addition the interpreter will maintain a two dimensional map into which function objects that convert from one type to another are placed. The system will attempt a conversion if the types on the stack are not identical.

Parser

The parser for TOIL performs more as a symbolic linker than as a conventional language parser. Its job is to convert the symbolic form of the program into a numerical form that is used by the interpreter.

A single pass design is used. This should make the loading process quite fast. It keeps the design quite simple.

Generally the parser works by translating the text symbols into numerics by looking them up in the dictionary for the current class. The base symbols are loaded from a set of libraries – this is described further below.

The symbols are read from a source file, which may include other source files (etc.). If a file name is relative it is treated as being relative to the previously loaded file (or the location from which TOIL was started). The symbols are white space delimited so the language is entirely free format (very much unlike Python). The only exceptions are comments which continue to the end of the current line and the value string assigned to variables, which also includes the remainder of the current line (which makes string constants possible).

The parser recognises the following symbols:

- lib – the following symbol is used to load either a built in class (such as string) or a shared library plug-in that contains a user written class.
- include – the following symbol is treated as a source file (or URL), the current read location is pushed onto a stack and the new file read. When the file is exhausted it is popped off the stack and the previous file resumed.
- # - a comment – the remainder of the line is discarded.
- { } - signifies the beginning and end of a class definition.
- [ ] - the beginning and end of a list of parent classes.
- message – is used to identify the starting point for the script. It is followed by a class name and a function name.
- var – identifies the definition of a variable. It is followed by the type, the name of the variable and a string that will be converted into its initial value.
- pvar – as for var, but flagged as being private so that it will not be collapsed if more than one instance is included in a class.
- : - identifies the beginning of a function.
- ; - identifies the end of a function. This corresponds to the "return" function object unless the function name is "main", in which case the "stop" function object is used.
- call – identifies a call from one object to another. It expects a class name and function name to follow it.
- Numerics – short integer values can be placed directly into the code for use as counters etc.

The parser also recognises some words and generates short sequences of code. The aim is to make programming in TOIL a bit less error prone. The design currently includes the following symbols:

- if else – each of which must be followed by exactly one function name. It removes the value from the top of the stack and calls the first function if its non-zero, or the second one otherwise;
- foreach – which must be followed by exactly one function name. It expects a vector object on top of the stack and pushes on a loop counter. It then pushes on the indexed entry and calls the function. On return it increments the loop counter and repeats;
- for – (Not implemented yet) executes its function for a fixed number of times using a loop counter;
- while – (Not implemented yet) executes its function until a boolean value is false;
- until – (Not implemented yet) executes its function at least once and until a boolean is true.

Errors and Debugging

The system provides some minimal support for detecting errors and attempting to resolve them. A "trace" function object can be used to enable or disable tracing so that the user can determine the state of the system as the code executes. If an error is detected the system reports the name of the function object and the error message, and then unwinds the stack, reporting the instructions at each level.

Libraries

The aim is to produce an extensible system where a user can provide libraries of operators for special purposes, such as matrices or graphics. Each library can define one or more classes and add function objects to it. A user class can then include these classes as its parents to make the functions available to the classes own functions.

The system includes some built-in classes:

- Core – provides the essential operators for the system such as if, jump, call, return and exec.
- Stack – provides basic stack operations such as dup, swap, etc.
- Logic – provides basic boolean algebra.
- Integer – provides numerics on 64 bit integers
- Real – provides numeric operations on double precision floating point numbers.
- String – provides string manipulation operations.
- Type – provides manipulations of the type system.
- Vector – provides a basic array.
- Map – provides an associative array.

Some standard plug-ins are also provided. These include operating system functions, file handling and so on. These will be provided as plug-ins so that the user can control their use, and thus prevent undesired access to their system.

Plug-in libraries will use the framework provided by the CFI library.

Remaining Work

The core of the TOIL system is now functional, but considerable work is required to complete the task so that it is a general purpose language.

Short Term

The outstanding tasks to get the system fully functional include

- Object References – These are required to prevent memory leaks.
- Operators – We still need to create operators for logic, arithmetic and string handling. Operators for type handling would enable the scripts to do type manipulations..
- Exceptions – Support for exception handling needs to be built into the design.
- Plug-ins – required for operating system functions and file handling etc.

- A programming manual listing the operators would be useful.