A Compiler Designed for the VSOP Language

Lycoops Vincent
Montefiore Institute
School of Applied Sciences
University of Liege
Liege, Belgium
Email: vlycoops@student.ulg.ac.be

Mathonet Gregoire
Montefiore Institute
School of Applied Sciences
University of Liege
Liege, Belgium
Email: gregoire.mathonet@student.ulg.ac.be

***Abstract*—In this report, we focus on explaining our different implementation choices, as well as our personal choices for the interpretation of a VSOP source code file.**

I. Introduction

The VSOP language is a ULg designed programming language aiming at being Turing-complete and providing the most used functionalities of well-known programming languages while focusing on keeping a lightweight syntax and little syntactic sugar.
In this context, we are building a compiler for this language that should be able to reach the LLVM transition format, to apply any other backed compiler afterwards. The designed compiler should also report errors in a convenient fashion.
We now describe each part of the frontend of the compiler in the following sections.

II. Lexer

The lexer uses a JFlex produced regular expression analyzer to cut the input file into a stream of tokens. It tries not to stop on the first error to report as many errors as possible. Literals are determined by a look-ahead of either a whitespace character or an operator, so that ending the file with a literal will ouput an error. However, as VSOP is class-defined, the last character of any file should be a closing bracket, and this should not be a problem. To ensure compatibility with tests, however, we use a stream reader that adds a whitespace to the end of the stream, allowing those matchings to occur.
Upon error, the lexer tries to determine which kind of error happened to produce more useful feedback to the user. This way, if the problems happens while parsing the beginning of a non-decimal number, the lexer will be able to output a specific message, for instance.
A final EOF token is added to the end of the stream, token used by java_cup but which is not rendered while displaying the found tokens.

III. Parser

The parser reads the stream of symbols generated by the lexer to give them as feed to the java_cup implementation of the VSOP grammar. The value associated to a mathematical expression is still not computed on the fly, even if can be known at compile time.
Each generated node of the tree is an ASTNode that still embbeds the information of position in the source code if it is a terminal symbol, or the beginning of a syntactically determined block remembered (for instance, a class, but not a parenthesis). The terminals typically have a value, either a string or an integer one, while all nodes can have properties such as position. A type is associated to the leaves of the created AST so that the type of all object identifiers is already known at this time.
The dumping is done recursively from the root node, exploring children's state.
Error recovery is kept minimal, it only tries to parsing correctly three next tokens. If this is possible, the generation of the tree is continued, but by default the generated tree is not outputted as it may refer to garbage only. The position of the error is reported to the last correctly parsed token if the symbol involved is not a terminal, or at the position of the terminal itself if known. So far, the position of elements is not recorded upwards so that some errors might still not be accurate in position.

IV. Semantics analyzer

A first uncomplete pass is performed in the AST, in order to find all the classes, and then to populate all classes with their methods. At this step, no check is done to validate inheritance nor method overriding.
From a first complete pass in prefix order in the AST, the analyzer creates the scope of each node, by adding elements for all children of a node that declares a new element in the scope. From the same pass, the semantic analyzer creates a tree of classes that extend from each other and verifies that each atom used as a leaf (not in a declaration) is indeed present in the scope of this leaf. A second pass is then issued in postfix order, to add a type information to each statement and block of code. The leaves' type is already known from the parser, and the type of a supernode is uniquely determined by the operation applied to children expression of this operation. At the same time, the analyzer

verifies that the involved operations exists, are correctly called, with arguments matching in type and number. Thanks to the "extends tree", the analyzer is able to substitute a child class to a parent one to retrieve a legible operation.

V. LLVM IR GENERATION

Section still under work.
The generation comes down to building a OOP-free representation of an OOP language: each method is turned into a global function whose name is prefixed with the name of its class, and that takes as first argument a structure representing the class: a structure that contains but the fields of the class.
The new operator invokes the constructor of the class, that issues a "super();"-like call if it extends something else than Object. The fields in a class are ordered with highest in extends-tree first so that those calls are easily managed.
The code is then emitted recursively, and the entry point of the program is set to a new Main.main().

A. Direct LLVM emission

The compiler was originally designed to emit native LLVM directly, but in order to ease the integration of new functionalities, we have rather focused on generating C code, that CLang can compile either to LLVM or native code, and adding functionalities on top. We actually first though we would emit native LLVM code, but java really has a poor support for LLVM, that is why we changed our mind. You can find a stub for LLVM that never could produce anything in the src directory.

B. C generation

This C generation is done as depicted above, only the "let" keyword has to be transformed into larger closures, with randomly generated new values. This is actuallt very close to direct LLVM generation and we think directly generating LLVM wouldn't be too hard to implement, only to the matter of learning its syntax.

VI. VSOP EXTENDED

We added the following functionalities to extend the initial language, we describe here how they can be used.

A. Garbage collection

The language is garbage collected at run-time.

B. IO

The language can dump out values to standard output using a class that will extend the built-in IO. Facilities for inputting values also exist and comply with the language semantics.

C. Default parameters to methods

Methods arguments can have default values, using a syntax such as $name : type : defaultvalue$. These default values are overridden when the method is called with other arguments, and if not enough parameters are given, the compiler tries to add the remaining from their default values (one should therefore have the last arguments predefined, or there will be an error).

D. Defined dispatch

Using the syntax $object < type.method(params)$, one can cast $object$ to $type$ and use the method defined as in $type$. Obviously the extension must exist and the method be defined at the upper level as well.

E. Missing operators

Added OR, GREATER, GREATER_EQUAL, SHIFT_LEFT and SHIFT_RIGHT operators.

F. Numeric types

The float type has been added, with facilities for all operations and printing. POW operations still requires an int32 as exponent. The format of floats is to be $[0-9]+.[0-9]+f$. One can also switch between int32 and float types by using the @ symbol.
The byte type has been added, and represents only one byte of information. This type should be usable almost anywhere where an int32 is expected and casted down or up depending upon the situation.

G. Arrays

One can use arrays, in a much like-java fashion, except that no protection against out of index exists. You can allocate arrays of any type, or arrays of arrays, the compiler checks that you access arrays as their type, and that you do not dereference them too much. The arrays are given by address, and assigning an array to another only copies the address, not the real contents. These arrays also support easy matrix semantics, that is, it is allowed to allocate over several dimensions with one instruction, where all dimensions will have the same length.

H. Public, protected and private

All methods and fields must be declared public, protected or private, using the + , ~ and - modifiers. Assignments to fields obtained through a method are allowed, eg: foo.getBar().baz <- 2; is a legal instruction.

I. Cast to children

The compilers allows forced casts to child classes, although this may cause a run-time error. Those casts have to be explicitely requested.

J. Include

In order to reuse part of the code, one can include file for compilation using the # directive, followed by a filename. These directives must be given at the beginning of a file, any number of such may be given. This file may be given relative or absolute, and will be included only if no other previous file included a file whose hash is the same. Note that this is not compilation avoidance, as the so given files are plain source code that will be recompiled.

K. Automatic generation of getters and setters

The syntax + in front of a field name allows the compiler to automatically generate a getter and a setter for the field, named getField and setField respectively, where the setter returns the self pointer in order to chain calls.

L. Wrapper types

The compiler automatically includes a file that declares wrappers for basic types, the classes Unit, Bool, String, Float and Integer. When possible, the compiler will wrap basic types in a wrapper clas when required (such as in a assignment towards such a class), and unwrap them if needed (such as in mathematical expression or array dereferencing or passing as argument to a method that expects basic types).

M. Easier constructor use

For default uses of the NEW operator, support of inlined constructors has been added, that is, new Class(args) is allowed and will dispatch a call to the init method of the newly generated object right after its creation.

N. For loop on arrays

Arrays support for-loop constructs, where type is inferred, using the construct ”for item : array : array_length do”. If the expression given as array is not an array or if the length is not an integer, an error is thrown. This construction is recommended as easy as does not involve a function.

O. Several local variables at once

The syntax of the ”let” keyword has been extended to allow one to declare several variables at once in a ”let” construct, using comma’s between each declaration.

P. Equals, code and enhanced methods

The Object class defines an equals method that is nothing but address comparison. Children classes are free to redefine locally this mechanism, as is done by the String class that uses the C strcmp function on its value field. The String class is thus extended to allow for comparison, concatenation etc, that the native type does not support.
In the same fashion, the code method computes a hash of the object, usually of the address, unless overriden as for String for instance.
Other wrapper types also have their equals method overriden and some helper methods.

Q. Object erasing

Unlike many OO languages, VSOP extended can allow for object explicit destruction, using the ”erase” keywork. This should only be used for values that should not remain in memory at run time for undefined amounts of time, such as for cryptography libraries. The ”erase” operation always returns a null object whose type is the type of the destructed object. Only objects can be destroyed. If a public or protected method named ”delete” that does not take arguments and returns self exists in the extension tree of the destroyed class, it is called before destruction happens, to garble up memory that should not be recoverable.

R. External interfaces

In order to not re-invent the wheel over and over, VSOP extended can use any C defined function in the following way: you have to add the includes you will need using -ci parameter of vsopc, and the libraries using -cl. Now, you can interface by creating a class whose names begins with External. All methods of such a class are taken to be defined outside, with a C function of the same name, taking the same arguments except the self pointer omitted. The correspondence of types is int32(int), string(char*), float(float), byte(char), bool(char). You should just return something from the method, but it won’t ever get evaluated, so often 0. Check the TestExtended.ve file for example. You should only interface other structure to Object, having them passed by reference.

S. Interval testing

The syntax ”expression <> a:b” returns true if the expression is within bounded range a:b, working for integer types only obviously.

VII. Conclusion

We presented our still under work compiler for the VSOP language. As one can see, it boils down to assemble the right generators finely tuned for a given language, but generic options to generate compilers now exist.

ACKNOWLEDGMENT

The authors would like to thank the developers of the tools we used to build this compiler, namely JFlex, java_cup and LLVM so far.

REFERENCES

[1] G. Klein, *JFlex*, http://jflex.de/

[2] Technical University Munich, *java_cup*, http://www2.cs.tum.edu/projects/cup/

[3] LLVM, *LLVM IR specification*, http://llvm.org/docs/LangRef.html