Implementing a VSOP Compiler

Cyril SOLDANI, Florent DE GEETER, Pascal FONTAINE

March 23, 2023

Part III

Semantic Analysis

1 Introduction

In this assignment, you will do the semantic analysis of VSOP programs, according to the semantic rules described in the VSOP manual. You will reuse the lexer and parser developed in the previous assignments (you can of course improve them if needed).

To allow easy testing of your generated semantic checker, your program will dump the annotated *abstract syntax tree* (AST) corresponding to given input VSOP source file on standard output, in the format described in section 2.

As VSOP source code files can contain semantic errors, you will need to detect those (in addition to lexical and syntax errors), and print error messages when they occur. The way semantic errors should be handled is not precisely described in the VSOP manual. This documents gives some more information about the way your compiler should handle those errors in section 3.

Some guidance regarding how you should test your semantic checker is given in section 4.

Some guidance regarding how you can implement your type checker is given in section 5.

Finally, the way you should submit your work for evaluation is described in section 6.

This assignment is due **at the latest** for Tuesday the **18th of April** 2023 (23:59 CET).

2 Output Format

The output format for this assignment is the same as the one for the previous assignment, except that expressions will be annotated with their type, using the notation:

`<expr> : <type>`

where `<expr>` is printed as previously, and `<type>` is one of `unit`, `bool`, `int32`, `string` or a class name.

Example

For the source code of the linked list example in the VSOP manual (figure 1), your semantic checker should output something like:

```
[Class(List, Object, [],
   [Method(isNil, [], bool, [true : bool] : bool),
    Method(length, [], int32, [0 : int32] : int32)]),
 Class(Nil, List, [], []),
 Class(Cons, List, [Field(head, int32), Field(tail, List)],
   [Method(init, [hd : int32, tl : List], Cons,
      [Assign(head, hd : int32) : int32, Assign(tail, tl : List) : List, self
       : Cons]
      : Cons),
    Method(head, [], int32, [head : int32] : int32),
    Method(isNil, [], bool, [false : bool] : bool),
    Method(length, [], int32,
      [BinOp(+, 1 : int32, Call(tail : List, length, []) : int32) : int32]
      : int32)]),
 Class(Main, Object, [],
   [Method(main, [], int32,
      [Let(xs, List,
         Call(New(Cons) : Cons, init,
           [0 : int32,
            Call(New(Cons) : Cons, init,
              [1 : int32,
               Call(New(Cons) : Cons, init, [2 : int32, New(Nil) : Nil])
               : Cons])
            : Cons])
         : Cons,
         [Call(self : Main, print, ["List has length " : string]) : Object,
          Call(self : Main, printInt32, [Call(xs : List, length, []) : int32])
          : Object, Call(self : Main, print, ["\x0a" : string]) : Object, 0
          : int32]
         : int32)
       : int32]
      : int32)])]
```

assuming the class `Object` has methods:

```
print(s : string) : Object { (* implementation *) }
printInt32(i : int32) : Object { (* implementation *) }
```

3 Error Handling

Your semantic checker should detect and report semantic errors in input VSOP source files. Error messages should begin with

```
<filename>:<line>:<col>: semantic error:
```

where the position indicates where the error was detected.

As for the syntax analysis assignment, the exact error message and positions reported are up to you. Use you best judgment to provide useful error messages.

Here follows a non-exhaustive list of things to check, along with some (non normative) reporting examples.

You should report if:

- an expression does not conform to expected type, *e.g.* in a field initializer, a method argument, *etc*.

```
class List {
    isNil() : bool { true }
    length() : int32 { 0 }
}

(* Nil is nothing more than a glorified alias to List *)
class Nil extends List { }

class Cons extends List {
    head : int32;
    tail : List;

    init(hd : int32, tl : List) : Cons {
        head <- hd;
        tail <- tl;
        self
    }

    head() : int32 { head }
    isNil() : bool { false }
    length() : int32 { 1 + tail.length() }
}

class Main {
    main() : int32 {
        let xs : List <- (new Cons).init(0, (new Cons).init(
                                         1, (new Cons).init(
                                         2, new Nil))) in {
            print("List has length ");
            printInt32(xs.length());
            print("\n");
            0
        }
    }
}
```

Figure 1: VSOP source code of a linked list

What position to report is not trivial. Two common approaches are bottom-up and top-down reporting. For the code

```
class C {
    f() : int32 {
        "an ignored string";
        let x : bool <- { 1; "a" } in not x // Last expression of block
    }
}
```

Bottom-up reporting would give something like

```
input.vsop:4:25: semantic error: expected type bool,
    but found type string.
input.vsop:2:17: semantic error: expected type int32,
    but found type bool.
```

The first message complains that the type of `{ 1; "a" }` is not bool, while the second complains that the whole method body should be of type int32.

Top-down reporting would give something like

```
input.vsop:4:30: semantic error: expected type bool,
    but found type string.
input.vsop:4:39: semantic error: expected type int32,
    but found type bool.
```

the first message reports the position of `"a"`, while the second report the position of `not x`.

In VSOP, we have the feeling the top-down approach generally gives better error messages, but you are free to report those errors as you see fit, as long as you report them;

- a class is redefined. Don't forget that `Object` is predefined;
- a field or method is illegally redefined, *e.g.*

```
class C {
    i : int32;
    s : string;
    i : bool;
}
```

```
input.vsop:4:5: semantic error: redefinition of field i,
    first defined at 2:5.
```

- a method has several formal arguments with the same name;
- a type is used that is not defined in the file, *e.g.*

```
class C {
    g : G;
}
// End-of-file
```

```
input.vsop:2:9: semantic error: unknown type G.
```

- an identifier is used that is not defined in the scope;
- an overridden method formal argument or return type does not match its parent method type, *e.g.*

```
class P { add() : int32 { 42 } }
class C extends P {
```

```
3       add(i : int32) : int32 { i + 42 }
4   }
```

```
input.vsop:3:5: semantic error: overriding method add
    with different type
```

- some classes are involved in a cycle, *e.g.*

```
1   class A extends C { }
2   class B extends A { }
3   class C extends B { }
```

```
input.vsop:1:17: semantic error: class A cannot extend child class C.
input.vsop:2:17: semantic error: class B cannot extend child class A.
input.vsop:3:17: semantic error: class C cannot extend child class B.
```

- a dispatch is made to an invalid method, *e.g.*

```
1   class Empty { }
2   class NotEmpty extends Empty { f() : unit { () } }
3   class Bogus {
4       g() : unit {
5           let e : Empty <- new NotEmpty in
6           e.f(); // Static type of e is Empty
7           ()
8       }
9   }
```

```
input.vsop:6:11: semantic error: class Empty has no method named f.
```

- an input file has no class `Main`, or if the class `Main` has no `main` method. You *can* report it at position 1:1 or at end of file (somewhat arbitrary).
- the `main` method of the `Main` class has not the right signature;
- both branches of a conditional don't agree, *e.g.*

```
1   class C {
2       g() : int32 {
3           if inputInt() = 0 then 42
4           else false;
5           -1
6       }
7   }
```

```
input.vsop:4:14: semantic error: expected type int32,
    but found type bool.
```

This list is not exhaustive. Read the manual carefully to find other error conditions that should be reported.

4 Testing Your Semantic Checker

You are expected to test your semantic checker by writing some VSOP input files, and calling your semantic checker on them. Use both *valid* input files, which are semantically correct, and *invalid* ones, that should trigger errors. Try to test all language features, and all specific errors you should detect. Don't forget to try a few nested constructs, like a while loop inside an if-then-else branch, or *vice versa*.

Don't assume your lexer and parser are bug-free. If you encounter some strange semantic errors, or if the output is different from what you expected, it may be worth calling `vsopc` again with the `-l` or `-p` argument to check if previous phases are correct.

5 Practical Advice

As the declaration order of classes, fields and methods does not matter in VSOP, it is hard to analyze a VSOP program in one pass. *E.g.* in the code

```
class C extends P { (* ... *) }
(* many other classes ... *)
class P { (* ... *) }
```

When you first meet `P` on the first line, you do not know yet whether or not it is defined and whether or not it is a child of class `C`. It is only when you have seen the whole file that you can answer those questions.

You could try to queue checks to be made later and/or use a smart data structure to check a VSOP program in a single pass, but it would quickly become quite complex.

Instead, do as many passes as you need to do the job easily. For example, you could do a first pass over your AST, just considering class declarations (ignoring their fields and methods), recording which classes are defined, and what their parents are. Then you can do a check for cycles, followed by a second pass, over the whole AST this time, and report any use of an undefined class. A third pass might record the types of methods and fields for each class, without inspecting their body or initializer. Finally, you could do the complete typechecking in a fourth pass over the whole AST, using the previously gathered information.

This is just an example. You can use different passes, but use several! Don't be overly concerned with the performance of your compiler at this stage. It is better to obtain a working compiler than a fast-but-wrong one, and the performance bottlenecks will not be there anyway (those are generally in the lexer, and in optimization passes).

A second advice is to note that, in VSOP, the expected type for an expression is often known before inspecting the expression itself. A method body should conform to its return type, the type of the last expression of a block should conform to the expected type for the block (other expression types are not known in advance, but cannot lead to type errors, as long as the expressions themselves are well typed), the type of a condition should be `bool`, the type of the body of a `let` should conform to the type expected for the `let`, *etc*.

You can take advantage of this fact to implement a top-down type checking, making easy to report typing errors in a top-down manner (as suggested in section 2).

6 How to Submit your Work?

You will submit your work for evaluation on the submission platform.

You can provide your test VSOP input files in a `vsopcompiler/tests` sub-folder. Any file with `.vsop` extension in that folder will be tested against the reference parser by the testing script.

Your code will be executed as follows. Your built `vsopc` executable will be called with the arguments `-c <SOURCE-FILE>` where `<SOURCE-FILE>` is the path to the input VSOP source code.

Your program should then output the typed AST on standard output, and eventual error messages on standard error, as described above. You program should still handle arguments `-l <SOURCE-FILE>` and `-p <SOURCE-FILE>` as in previous assignments.

The submission platform will check that your submission is in the right format, and test your semantic checker. If you want to be able to use that feedback, don't wait until the last minute to submit your checker (you can submit up to 40 times, only the last submission will be taken into account).

Also note that **5% of your final grade** will be determined directly by the automated tests for this assignment. It is thus doubly in your interest to ensure that your code passes all the tests (and early enough).

Plagiarism. Cooperation (even between groups) is allowed, but all cooperation must be clearly referred to in the final report, and we will have **no tolerance at all for plagiarism** (writing together, reusing/sharing code or text). Code available on the web that serves as a source of inspiration must also be properly referred in the report.

Participation. **Each student has to participate** in the project, and do her/his fair share of work. Letting others in the group do all work will be considered as plagiarism.