Implementing a VSOP Compiler

Cyril SOLDANI, Pr. Pierre GEURTS

March 23, 2016

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).

Note that a new `unit` type was introduced, which will require a slight modification in your lexer and parser. It is described in section 2.

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 3.

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 4.

Some guidance regarding how you should test your semantic checker is given in section 5.

Some guidance regarding how you can implement your type checker is given in section 6.

Finally, the way you should submit your work for evaluation is described in section 7.

This assignment is due at the latest for Wednesday the **13th of April** 2016 (23:59 CET).

2 Unit Type

A new type has been added to the VSOP language: `unit`.

The only inhabitant of `unit` is `()`, also called unit. `unit` is used in VSOP where `void` would be used in C, except that it has a valid value `()` which can be used as any other value (*e.g.* passed as argument to a method, stored into a variable, *etc.*).

You will need to modify your lexer by adding a new keyword `"unit"`.

You will need to extend your parser with the two following rules:

```
type = "unit";
expr = "(" ")";
```

3 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, your semantic checker should output something like:

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

assuming the class `IO` has methods:

```
print(s : string) : IO { (* implementation *) }
printInt(i : int32) : IO { (* implementation *) }
```

4 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.*

  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 the block
  }
}
```

  Bottom-up reporting would give something like

```
input.vsop:4:21: semantic error: expected type bool, but found type
string.
input.vsop:2:15: 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:4:26: semantic error: expected type bool, but found type
string.
input.vsop:4:35: 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:4:3: semantic error: redefinition of field i.
```

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

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

```
input.vsop:2:7: semantic error: unknown type G.
```

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

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

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

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

```
class A extends C { }
class B extends A { }
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.*

```
class Empty { }
class NotEmpty extends Empty { f() : unit { () } }
class Bogus {
  f() : unit {
    let e : Empty <- new NotEmpty in
    e.f(); // Static type of e is Empty
    ()
  }
}
```

```
input.vsop:6:7: semantic error: class Empty has no method named f.
```

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

```
class C {
  g() : int32 {
    if inputInt() = 0 then 42
    else false;
    -1
  }
}
```

```
input.vsop:4:10: 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.

5 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 `-lex` or `-parse` argument to check if previous phases are correct.

6 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 very 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, based on 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 3).

7 How to Submit your Work?

Your work should be submitted through the submission platform.

You should submit an archive named `vsopcompiler.tar.xz`. That archive should contain a `vsopcompiler` folder with your code, and a `Makefile` to build it. A skeleton `vsopcompiler` folder and `Makefile` is provided on the assignments web page. You can actually build your compiler with whatever tool you want (if you need building at all), but still provide a `Makefile` with the targets described here.

If something in your implementation is not obvious from your **commented** source code, explain it in a short report in a file `report.txt` or `report.pdf`. You can also describe the main difficulties you encountered in that file.

Finally, 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 semantic checker by the testing script.

Your code will be built as follows. From inside your `vsopcompiler` folder, `make install-tools` will be run. This should install any software needed to build your compiler (`make` and `llvm-dev`) are already installed. You can use `sudo` if you need administrator privileges. Then, `make vsopc` will be issued to build your compiler, which should, obviously, be named `vsopc`. Appending the letter `c` (for **C**ompiler) to the name of the language is a common convention for compilers.

Your code will be executed as follows. Your built `vsopc` executable will be called with the arguments `-check <SOURCE-FILE>` where `<SOURCE-FILES>` is the path to the input VSOP source code. Your program should then output the annotated AST on standard output, and/or eventual error messages on standard error, as described above.

The reference platform is an amd64 machine, powered by Debian Jessie (a GNU\Linux distribution). You can use the virtual machine provided on the assignments web page. Alternatively, you can use `debian/jessie64` with vagrant or `debian:jessie` with docker, or your own Debian-based machine.

We will try to add test scripts to the submission platform that 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 semantic checker (you can submit as many times as you want until the deadline, only the last submission will be taken into account).