The Pitfalls of Ansible's Variable and Template Expression Semantics

Opdebeeck, Ruben; De Roover, Coen

Publication date:
2021

The Pitfalls of Ansible's Variable and Template Expression Semantics

Ruben Opdebeeck, Coen De Roover
Software Languages Lab, Vrije Universiteit Brussel, Belgium
{ropdebee,cderoove}@vub.ac.be

I. Presentation Abstract

Ansible is a widely-used Infrastructure-as-Code (IaC) language for managing the configuration of machines in a digital infrastructure. The reliability of configuration definition files, which Ansible calls "playbooks", is of the upmost importance. However, Ansible employs semantics unlike those found in traditional programming languages, the unexpected behaviour of which could surprise developers. Next to forming a steep learning curve for newcomers, this semantics also hinders both manual and mechanical verification. In this presentation, we will show a number of potential pitfalls caused by a combination of unconventional semantic properties of Ansible variables and template expressions.

The purpose of this talk is three-fold:

1) To spread awareness of the unconventional semantics of Ansible and possible pitfalls to practitioners.
2) To entice tool builders to work on code analysers and bug detectors related to these pitfalls.
3) To stimulate language designers to address these pitfalls with safer alternatives.

A. Background

Ansible practitioners specify the desired configuration of their machine infrastructure in "playbooks", which contain one or more "plays", each targeting a group of machines. A play consists of multiple tasks, each describing a step necessary to configure the machine, which Ansible executes sequentially. Ansible also offers variables and expressions as a means to store and manipulate data. Expressions are specified in the Jinja2 templating language, where segments enclosed in double braces are evaluated by a template engine and substituted for the result. For example, the expression `{{ my_list | first }}` will run the value of a variable `my_list` through the `first` filter, which returns the first element of a sequence. These expressions can be bound to variables for later use in the playbook, or can be provided as arguments to tasks, to be used by Ansible's internals.

B. Example

To illustrate one example of the unexpected behaviour which will be presented, consider the code depicted in Listing 1, which shows the contents of two files. The `playbook.yml` file contains the playbook, consisting of one play which will execute two tasks. The play also binds two *play variables* (lines 5–6), `x` and `y`. Here, `x` is bound to `1`, while `y` is bound to a template expression which will produce the value of `x`. The play's first task (line 8) executes the `include_vars` action, which loads variables specified as key-value pairs in a separate file into a global scope in the play. The loaded file, `my_variables.yml` (lines 13–14), binds a variable `x` to 2. Note that this does not modify the value of `x` bound on line 5, it merely binds a new variable which happens to have the same name. Finally, the second task (lines 9–12) binds its own variable `x` to 3 in a task-local scope, and prints the value of the variable `y`.

```
1  ---
2  # Content of "playbook.yml"
3  - hosts: localhost
4    vars:
5      x: 1
6      y: "{{ x }}"
7    tasks:
8      - include_vars: "my_variables.yml"
9      - debug:
10         msg: "y is {{ y }}"
11       vars:
12         x: 3
13 ---
14 # Content of "my_variables.yml"
15 x: 2
```

Listing 1: Contrived example of unconventional Ansible semantics.

C. Unconventional Semantics

Given that there are three bindings for `x`, and `y` is defined in terms of `x`, a natural question arises: What will this second task print?

A reader who is used to traditional semantics may immediately proclaim that it will print "y is 1". After all, when `y` was bound, the value of `x` was 1, and `y`'s expression produces `x`, so `y` must be 1. However, the first unconventional property of Ansible's variables and template expressions is that *variables are not names for values*. Instead, a variable is a name given to an expression. The expression is evaluated only when the variable is dereferenced, and anew for every subsequent dereference. Template expressions are therefore lazy, and are more akin to lambdas than to raw expressions. In fact, one can interpret that the "value" of `x` is not 1, but rather that it is a literal expression which produces 1, and is only evaluated once `x` is dereferenced.

Nonetheless, the reader may assume that this lambda-like expression closes over the scope in which it is defined, and therefore resolves the name `x` to the original binding on line 5. On the contrary, the second unconventional property of template expressions is that they *do not close over the defining scope*. Therefore, evaluating `y` on line 10 will resolve the name `x` starting from the scope in which the expression is evaluated, not in the scope in which it was first bound.

A plausible conclusion would thus be that the second task will print "y is 3", as the task binds the variable `x` to 3 in its own scope. This brings us to the third unconventional property of Ansible's variable lookup, namely its *complex variable precedence* system consisting of 22 individual precedence rules [1]. These rules state that variables included through `include_vars` tasks take precedence over those in task-local scopes. Therefore, the binding of `x` to 2 of line 14, included through the first task, is in a "superposition"-like state: it is both globally visible, and also shadows locally-scoped variables in any subsequent task. This finally leads us to a surprising answer to the question at the beginning of this section. The second task (lines 9–12) will print "y is 2".

The example shown above is but one of multiple examples of potential pitfalls caused by unconventional semantics that we will present during this talk. Although the example is contrived, it is easy to imagine that these pitfalls can lead to bugs which are difficult to investigate in large Ansible codebases. The situation becomes particularly dire for Ansible roles, which can be considered libraries and are often maintained by third parties [2]. When a role executes an `include_vars` task, the included variables will break encapsulation and override variables local to the play that loads the role. We found that nearly 19% (4808) of the roles included in the Andromeda dataset [3] contain at least one `include_vars` statement, thus leading to many possibilities of unexpected variable shadowing in practice.

REFERENCES

[1] "Ansible documentation: Understanding variable precedence," https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html#understanding-variable-precedence.

[2] R. Opdebeeck, A. Zerouali, C. Velázquez-Rodríguez, and C. De Roover, "Does Infrastructure as Code adhere to Semantic Versioning? An analysis of Ansible role evolution," in *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM20)*. IEEE, 2020, pp. 238–248.

[3] R. Opdebeeck, A. Zerouali, and C. De Roover, "Andromeda: A dataset of Ansible Galaxy roles and their evolution," in *Proceedings of the 2021 International Conference on Mining Software Repositories (MSR21)*, 2021, pp. 580–584.