VeriSLO Usage Guide

Thibaut Blanc, Amaury Mazoyer, Juliette Ponsonnet, Hugo Salou

December 22, 2025

Contents

Todo list

1. Installation

See https://gitlab.aliens-lyon.fr/verislo/verislo/-/blob/main/README.md#installation.

2. Annotations

2.1. Annotations' syntax

Program annotations in VeriSLO are special OCaml attributes. There are currently four types of annotations:

- `[@ret ⟨var⟩]` specifies that the return value of the code block it is attached to is named *var* in future annotations.
- `[@post "⟨cond$_1$⟩",...,"⟨cond$_n$⟩"]` specifies the postcondition of the block it is attached to. Conditions cond_i must be valid Iris code (not exactly, *see* section 2.3). If multiple postconditions are given, *e.g.* for `"⟨cond$_1$⟩", "⟨cond$_2$⟩"`, the separating conjonction $\text{cond}_1 * \text{cond}_2$ is taken.
- `[@pre "⟨cond$_1$⟩",...,"⟨cond$_n$⟩"]` same as `post`, but for preconditions.
- `[@invariant "⟨inv$_1$⟩",...,"⟨inv$_n$⟩"]` same as `post`, but can only be applied on loops. Specifies the invariant of the loop.

2.2. Placing annotations

There are places where annotations are forbidden. Here are the rules to follow when placing annotations:

- `invariant` annotations can only be placed on `while` and `for` loops.

```
while ⟨cond⟩ do ⟨body⟩ done [@invariant "⟨inv⟩"]

for i = ⟨e1⟩ to ⟨e2⟩ do ⟨body⟩ done [@invariant "⟨inv⟩"]
```

- `pre` annotations can only be place on function definitions or in matching.

```
fun [@pre "⟨cond⟩"] x -> ⟨body⟩

function
  | ⟨pattern1⟩ [@pre "⟨cond1⟩"] -> ⟨body1⟩
  | ⟨pattern2⟩ [@pre "⟨cond2⟩"] -> ⟨body2⟩
  ⋮

match ⟨expr⟩ with
  | ⟨pattern1⟩ [@pre "⟨cond1⟩"] -> ⟨body1⟩
  | ⟨pattern2⟩ [@pre "⟨cond2⟩"] -> ⟨body2⟩
  ⋮
```

- `post` annotations
- `ret` annotations cannot be placed on expressions that do not return a value, such as `if` without `else`, `while` loops, `for` loops, *etc*.

Amaury: Post annotations' placement rules

In addition, some annotation placements are equivalent to others:

- Expression

  $$\texttt{let [@post "}\langle \textit{cond} \rangle\texttt{"] x = }\langle \textit{body} \rangle\texttt{ (in }\langle \textit{expr} \rangle\texttt{)}^{?}$$

  is equivalent to

  $$\texttt{let x = (}\langle \textit{body} \rangle\texttt{ [@post "}\langle \textit{cond} \rangle\texttt{"]) (in }\langle \textit{expr} \rangle\texttt{)}^{?}.$$

- Expression

  $$\texttt{let [@pre "}\langle \textit{cond}_1 \rangle\texttt{"] [@post "}\langle \textit{cond}_2 \rangle\texttt{"] (rec)}^{?}\texttt{ f x = }\langle \textit{body} \rangle\texttt{ (in }\langle \textit{expr} \rangle\texttt{)}^{?}$$

  is equivalent to

  $$\texttt{let (rec)}^{?}\texttt{ f = (fun [@pre "}\langle \textit{cond}_1 \rangle\texttt{"] [@post "}\langle \textit{cond}_2 \rangle\texttt{"] x -> }\langle \textit{body} \rangle\texttt{) (in }\langle \textit{expr} \rangle\texttt{)}^{?}.$$

2.3. Syntactic sugar

Consider the following example, we have mutable state in our code and we would want to prove property about the value of that state. We can, as a simple instance of this problem, consider the following code.

```
let x = ref 0 in
x := 5
```

In Iris, in order to talk about x's value, we need to use assertion $\mathtt{x} \mapsto \mathtt{v}$ where we interpret x as a location (this is especially true in VeriSLO). Then, we are free to prove (or use) properties about v (*i.e.* x's value). For example, an assertion proving that x's value is odd and greater than three would look like

$$\exists \mathtt{v}, \quad \mathtt{x} \mapsto \mathtt{v} \quad * \quad \mathrm{odd}(\mathtt{v}) \quad * \quad \mathtt{v} \geq 3. \tag{1}$$

(Note that we should only add one $\mathtt{x} \mapsto \mathtt{v}$ assertion as we get an absurdity if we have strictly more than one.) In a VeriSLO assertion, you can simply talk about the value !x as if it was a variable, *e.g.*

$$\mathrm{odd}(!\mathtt{x}) \quad * \quad !\mathtt{x} \geq 3$$

and the *syntactic sugar* will transform it into the expected assertion (1). For the special case that one "part" of the annotation (cond_i from section 2.1) looks like $!\mathtt{x} = \mathtt{a}$, the $\exists$v at the beginning becomes useless, as we have one candidate for !x, and we simply substitute by a everywhere: for example, we do the following translation:

$$!\mathtt{x} = 3 \quad * \quad \mathrm{odd}(!\mathtt{x}) \qquad \rightsquigarrow \qquad \mathrm{odd}(3) \quad * \quad \mathtt{x} \mapsto 3,$$

and

$$!\mathtt{x} = 3 \quad * \quad !\mathtt{x} = 4 \quad * \quad \mathrm{odd}(!\mathtt{x}) \qquad \rightsquigarrow \qquad \mathrm{odd}(3) \quad * \quad 3 = 4 \quad * \quad \mathtt{x} \mapsto 3.$$

Hugo

Once more syntactic sugar is implemented, document it here. (For example, OCaml code inside Hoare triples.)

3. List of unsupported features

Below is a (probably non exhaustive) list of the OCaml features not supported (yet) by VeriSLO:

- mutual recursion
- recursive definition of values
- labelled or optional arguments
- polymorphic variant
- arrays
- downto
- let operators
- lazy
- objects
- when clauses
- declaration of primitive operations
- modules (and everything related to them)
- classes
- records (except references)
- extensible constructors
- int32, int64 and native-int litterals
- float litterals
- char litterals
- string litterals
- physical equality
- shallow pattern matching
- exceptions
- continuations

A. Source code

The source code can be found at `https://gitlab.aliens-lyon.fr/verislo/verislo/`.