[PROVE]

2nd February 2021

1 Introduction

This document is work in progress. All source code as well as the most recent version of this document can be found on GitHub (see: https://github.com/g-regex/prove/blob/main/doc/doc.pdf). The purpose of this document to introduce notation, vocabulary and some underlying principles in order to facilitate efficient communication about this project. Furthermore this document could serve as an introduction to the project, if anybody wants to join the project.

2 Terminology

Let us start with some basic definitions

- A *statement* is a concatenation of one or more pair/s of square brackets enclosing another statement, a formula or an identifier.
- A *formula* is a concatenation of statements and formulators. The empty string is a special case of a formula. Apart from this special case every formula must contain at least one statement and at least one formulator.
- A *formulator* is a sequence of legal (see figure 2) characters delimited by either square brackets or the beginning or the end of the file.
- An *identifier* is a formulator, which is delimited by an opening square bracket at its beginning and by a closing square bracket at its end.
- To refer to statements or formulas more generally, we will use the term *expression* in this document.
- To refer to a formula, which is not an equality and not an implication, the term *ordinary formula* will be used throughout the document.
- When we speak of a *character* in this document, we mean a uppercase or lowercase Latin letter, a number or a special character (see figure 2). Opening and closing square brackets, and the reserved formulators `=`, `=>` are not referred to as characters in this document.
- The terms *variable* and *constant* can be used to refer to an identifier. Which term is to be used depends on the context, we are using it in (see Subsection 6.3).
- The term *real constant* refers to a formula, comprised of exactly one formulator and exactly one empty statement.

3 Syntax

The syntax of the [PROVE] language can conveniently be described by means of the extended Backus-Naur form (EBNF), which is a meta syntax notation defined in the ISO/IEC 14977 standard (International Organization for Standardization 1996). Using an EBNF enables us to recursively and unambiguously describe nested patterns by grouping syntactic elements to so called ⟨production⟩s. It must always be possible to 'unpack' a production in a unique and finite way, resulting in a (sequence of) *terminal* strings, indicated by quotation marks. In order to correctly interpret the [PROVE] EBNF it is sufficient to understand the meaning of the meta characters shown in figure 1.

|	indicates 'or'
(...)	groups productions and/or terminals together; only useful in combination with |
[...]	indicates optionality
{...}	indicates zero or more repetitions

Figure 1: Meta characters of the Extended Backus-Naur Form

$$
\begin{aligned}
\langle \texttt{expression} \rangle &= \langle \texttt{statement} \rangle \,\big|\, \langle \texttt{formula} \rangle \\
\langle \texttt{formula} \rangle &= \langle \texttt{formulator} \rangle \,\Big|\, \Big\{ \langle \texttt{statement} \rangle \langle \texttt{formulator} \rangle \,\Big|\, \langle \texttt{formulator} \rangle \langle \texttt{statement} \rangle \Big\} \\
\langle \texttt{statement} \rangle &= \text{`['} \langle \texttt{expression} \rangle \text{`]'} \Big\{ \text{`['} \langle \texttt{expression} \rangle \text{`]'} \Big\} \\
\langle \texttt{formulator} \rangle &= \text{`='} \,\big|\, \text{`=>'} \,\big|\, \langle \texttt{character} \rangle \Big\{ \langle \texttt{character} \rangle \Big\} \\
\langle \texttt{character} \rangle &= \text{`a'} \,\big|\, \ldots \,\big|\, \text{`z'} \,\big|\, \text{`A'} \,\big|\, \ldots \,\big|\, \text{`Z'} \,\big|\, \text{`0'} \,\big|\, \ldots \,\big|\, \text{`9'} \,\big|\, \text{`+'} \,\big|\, \text{`-'} \,\big|\, \text{`/'} \,\big|\, \text{`*'} \,\big|\, \text{`\%'} \,\big|\, \text{`\^{}'} \,\big|\, \text{`\&'} \,\big|\, \text{`.'} \,\big|\, \text{`?'} \,\big|\, \text{`!'} \,\big|\, \text{`:'} \,\big|\, \text{`\_'}
\end{aligned}
$$

Figure 2: Extended Backus-Naur Form of the [PROVE] language

Using this notation the [PROVE] language can be described by the EBNF shown in figure 2. Every EBNF needs to have an initial production as a starting point. It is convention that this distinguished production is to be listed first, so in this case, ⟨expression⟩ is the initial production. As one might notice a valid [PROVE] file might contain either a statement or a formula. This design choice has been made in order to account for axioms and theorems that might be given in the human readable form of statements in the [PROVE] language. Proofs on the other hand will be given in the form of a formula, having only `=>` formulators at its outermost level. We could easily think of other formulas, which will not satisfy this requirement and thereby neither constitute an axiom/theorem nor a proof. A [PROVE] file containing such a formula would be syntactically correct but would not have any meaning.

4 Semantic

When generating a couple of syntactically correct snippets of `[PROVE]` code, one realises that not all code adhering to the rules of the EBNF makes sense - at least not without introducing more conventions.

Consider the following code:

```
[a]and[b]=[c]=>[b][c]
```

The use of `=` and `=>` is ambiguous in this context, since `=` could refer to everything on its right hand side or just to `[c]`. Similarly `=>` could refer to everything to its left hand side or just to `[c]`. This problem could be addressed by introducing an order of precedence (e.g. non reserved formulators, `=`, `=>`). However, the implications of this approach should be carefully considered. Precedence can easily be encoded in the EBNF, without changing the set of valid code patterns. By doing so we could elegantly encode a part of the language's meaning in its syntactic structure. As tempting as this idea might be, there is a negative side effect coming along with this approach. Introducing precedence between formulators will set brackets *implicitly*. Since the visibility and meaning of identifiers (variables/constants) depends on the depth of nesting of statements (i.e. on brackets), the approach of precedence might lead to counter-intuitive code.

Instead the following semantic rule is introduced:

- `=`, `=>` and other formulators must not be mixed ✓

It might also be convenient to restrict the form an equality can have. Especially with respect to a straight-forward implementation, it would make sense to only let single statements (i.e. only one pair of square brackets at the outermost level) to be equal to each other.

Therefore the following rules are suggested:

- Equalities are only such formulas, which are a concatenation of single statements and `=`, where no single statement is next to another single statement and no `=` is next to another `=` ✓
- An equality must start and end with a single statement. ✓

Further, the following rule for implications is suggested:

- `=>` must not appear at the end of a ⟨`statement`⟩

5 Representation

Let us have a look at the following statement:

Source Code 1: Example

```
1 [a][b][[a]rel[b]]
```

An intuitive approach for representing the structure of this code is by thinking of it as a tree, where every statement and every formulator is represented by a node. It is worth mentioning that this tree can be interpreted as a binary tree (each node has one parent node and at most two children - one at the left and one at the right). However it is more intuitive to not think of this binary tree in its usual form, but to rotate it leftwards. Then every node can have (at most two) children - either below or to the right - and a parent (exactly one - expect for the root) to the left or above it.

As we see, there are several ways of thinking of a node as a child or a parent to another node (i.e. does a node lie in between another node and the root or does it lie above another node in the graphical representation). For the rest of the document, we will use the following terminology, always referring to the proposed graphical representation of the tree:

- A *child* is a node, which lies below another node.
- A *parent* is a node, which lies above another node.
- The nodes to the sides of another node are referred to as the *left* and *right* nodes respectively.

5.1 Graph creation

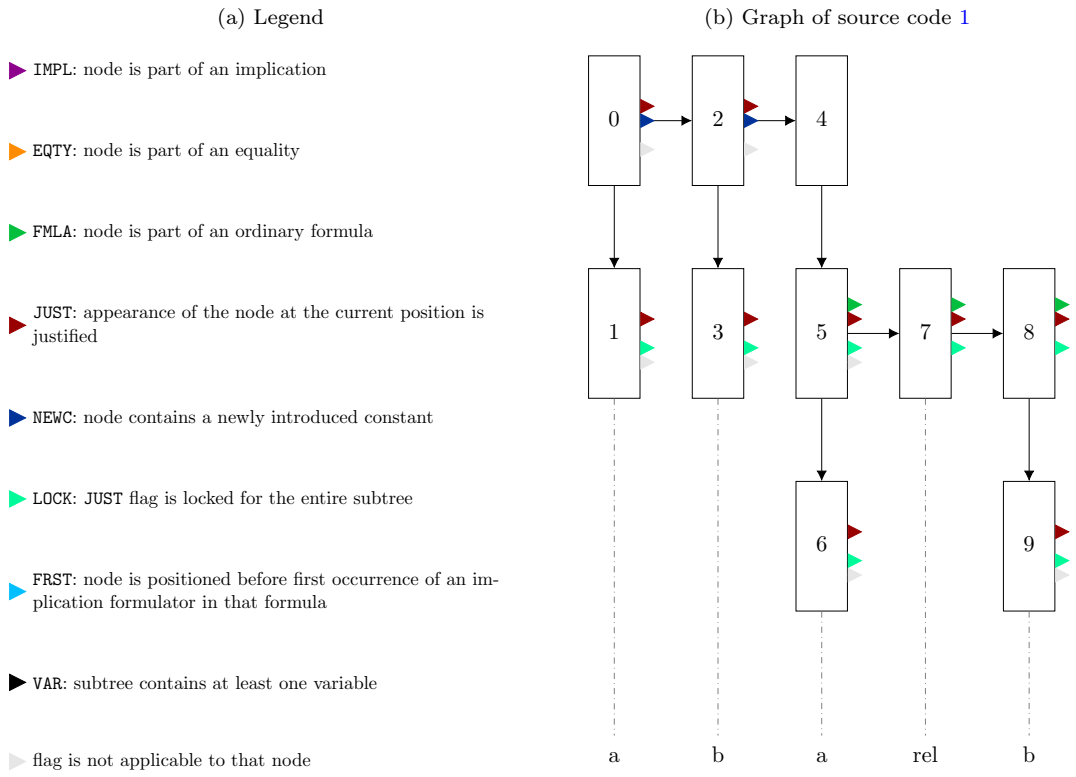
When creating a tree for some corresponding `[PROVE]` code, we start with a single node at the root of the tree. Since we refer to nodes in terms relative to another node, we have to keep track of our current position in the tree, while creating it. This position will be referred to as the *current* node. In the beginning the root is the current node.

Every node can have a number of properties, some of which exclude others. For example, every node can have a child or carry a formulator, but not both. Processing the code bracket by bracket and formulator by formulator, we now perform the following steps:

- When encountering a `[`, we check, whether the current node has a child or carries a formulator. If not, we create a child to the current node. This child becomes the current node. Otherwise we create a node to the right and a child to that node. That node then becomes the current node.
- When encountering a formulator, the formulator property of the current node is set.
- When encountering a `]`, we move from one node to another leftwards until we encouter the leftmost node, which has a parent. Then we move to that parent, which becomes the current node.

On the next page a graphical representation of source code 1 can be found.

Figure 3: Representation of source code 1



The tree shown in figure 3 (b) was automatically generated by `[PROVE]` following the steps described above. The nodes have been numbered in the order they have been created, which corresponds to a pre-order traversal numbering of the tree. The flags drawn on the righthand side of each node encode information, which has been gathered during the creation of the graph. E.g. the `NEWC` flag indicates that the statement contained by a node introduces a new constant (see Subsection 6.3). These flags correspond to single bits in the structure corresponding to each node in memory. Using these flags simplifies later tasks by storing already gained information and thereby avoiding repeated computation of frequently performed steps.

Remark: Some of these flags are set for nodes, which they are not applicable to and some flags are not set, where they could be. This has technical reasons (mainly related to computational efficiency) and is nothing we have to worry about.

5.2 The flags IMPL, EQTY and FMLA

During graph creation the flags IMPL, EQTY and FMLA are set respectively according to the following rules:

- When the current node contains an `=>` formulator, the IMPL flag is set.
- When the current node contains an `=` formulator, the EQTY flag is set.
- When the current node contains any other formulator, the FMLA flag is set.
- When a new node to the right of the current node is created, the respective state of the flags IMPL, EQTY and FMLA of the current node are copied over to the new node.
- When encountering a `]` and moving to the left-most node of the current sub-tree (see Subsection 5.1), the states of the flags IMPL, EQTY and FMLA are copied over from right to left.

5.3 The flags JUST and LOCK

One of the flags used is the JUST flag (short for *justified*). It only has a meaning in combination with the IMPL flag. The JUST flag is set according the following rules:

- When a new child node is created and the JUST flag or the LOCK flag of the current node is set, the JUST flag *and* the LOCK flag of the new child node are set as well.
- When a new child node is created and for the current node none of the flags IMPL, EQTY or FMLA are set, the flags JUST and LOCK are set for the new child node.
- When a new node to the right of the current node is created and any of the flags EQTY, FMLA or LOCK of the current node are set, the JUST flag of the new node to the right is set. The state of the LOCK flag is carried over to the new node.
- When a new node to the right of the current node is created and the IMPL flag of the current node is *not* set, the JUST flag of the *current* node is set.

The only function of the LOCK flag is to ensure that all nodes in a sub-tree below a node with a JUST flag set has also the JUST flag set.

The JUST flag can be interpreted as follows: If the JUST flag is set, the appearance of the corresponding node in the tree is justified in the sense that we accept its appearance in the currently processed subtree without verifying it (see Section 6). We do not speak of *justification*, since a missing JUST flag does not imply that there is no good reason for the corresponding node to appear at that position in the tree. However, the meaning of the combination of the IMPL flag and the JUST flag will be elaborated in the beginning of Section 6.

6 Verification

Verifying a node means finding another node at an eligible position in the tree (see Subsection 6.4), which is similar to the current node (see Subsection 6.2). Whether a node has to be verified depends on the combination of the state of its corresponding `IMPL` flag and `JUST` flag. The following rules apply:

- If both, the `IMPL` flag and the `JUST` flag, are set, the corresponding node contains an assumption - a statement not depending on any other statements. Such a node does not require verification.
- If neither the `IMPL` flag nor the `JUST` flag are set, the corresponding node does not require verification.
- If the `IMPL` flag is set and the `JUST` flag is unset, the corresponding node requires verification.

6.1 Output

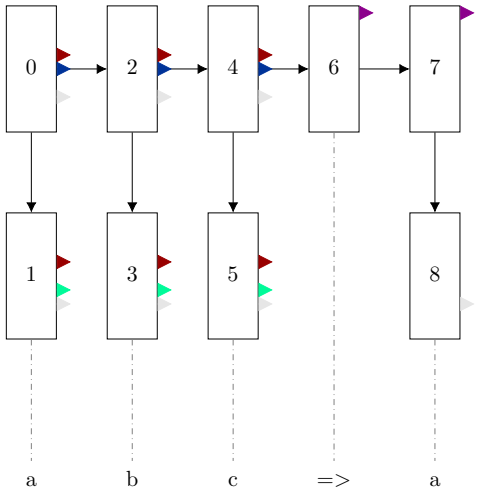
The `[PROVE]` software currently only produces debugging output. However, this output might help understanding the basic principles of the software and could be developed further to some meaningful and user-friendly output.

The debugging output is a modified version the original `[PROVE]` code, in which line breaks, tabstops and verification information has been inserted. On the next page you will find a short explanation of the debugging information and an example.

Source Code 2: Example

```
1 [a][b][c]=>[a]
```

Figure 4: Representation of source code 2



Source Code 3: `[PROVE]` output for Source Code 2

```
1 [a][b][c]=>
2 [a]{7}<4><2><0#>
```

Source Code 2 shows a trivial example `[PROVE]` code, which requires a verification step after processing the last statement. As figure 4 shows, node 7 has the `IMPL` flag set and the `JUST` flag unset. Thus node 7 requires verification.

Source Code 3 shows the verification information, which has to be interpreted as follows:

- If a node requires verification, the number of the node is printed in braces (e.g. `{7}`) after the corresponding statement.
- The number of every node, the node to be verified is compared to, is printed in angular brackets (e.g. `<4>`).
- If the sub-trees *below* the two compared nodes are similar to each other, a hash is printed within the angular brackets (e.g. `<0##>`).

6.2 Similarity of sub-trees

Two sub-trees can only be compared to each other, after all variables (with respect to the parent of the root of the respective sub-tree) have been substituted (see Subsection 6.3). The compared sub-trees are found to be similar, when the following conditions hold:

- Each node from the one sub-tree has a corresponding node in the other subtree.
- If one of the nodes has a node to its right, the corresponding node does so as well.
- If one of the nodes has a child, the corresponding node does so as well.
- If one of the nodes carries a formulator, the other node must carry the same formulator.

In the example of source code 2 the compared sub-trees consist of only a single node, each carrying a formulator. The similarity of the sub-trees under nodes 0 and 7 (with the nodes 1 and 8 as their roots respectively) is trivial. If we want to verify slightly more involved formulas, we have to discuss some more concepts. We will do so in the following subsections.

6.3 Variables and constants

As mentioned above, identifiers (i.e. formulas, which consist of a single formulator) can be variables or constants depending on the context. The context is given by the tree and the position of a specific node, from whose perspective we will determine the status of an identifier.

An identifier can have two different states with respect to a specific node: *unknown* and *constant*.

6.3.1 The flags `NEWC` and `VAR`

The state of an identifier can be determined by looking at the flags `NEWC` and `VAR`:

- If the `NEWC` flag of a node is set, then the identifier is a constant for all nodes to the right of that node and their respective sub-trees.
- If the `VAR` flag of a node is set, then the subtree below that node contains at least one node with the `NEWC` flag set. The identifiers in these nodes are subject to substitution and unknown to the nodes (and their respective sub-trees) to the right of the node with the set `VAR` flag. Of course this unknown status will be overwritten as soon as an identifier will be declared.

6.3.2 Examples

To illustrate the concept of variables and constants, let us slightly modify source code 2 and put square brackets around the first two statements:

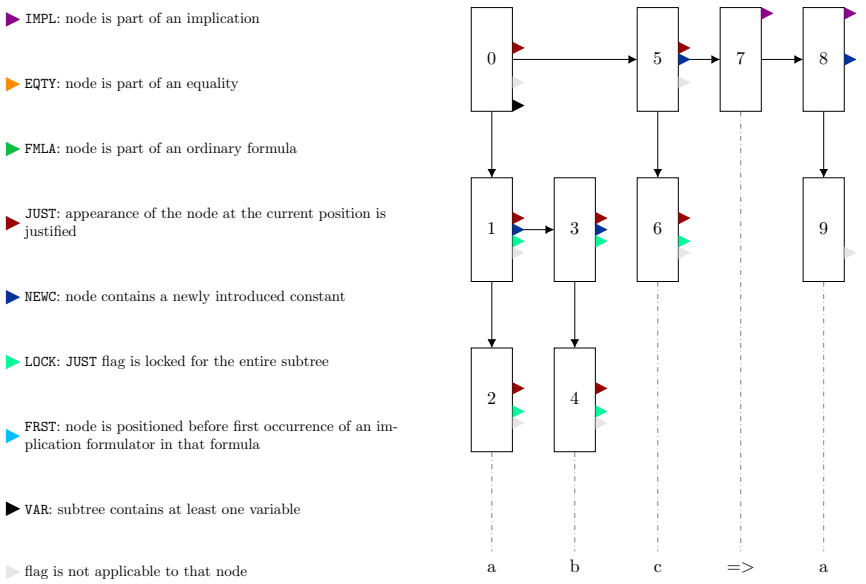
Source Code 4: `examples/var1.prove`

```
1 [[a][b]][c]=>[a]
```

Figure 5: Representation of source code 4

(a) Legend

(b) Graph of source code 4



Source Code 5: `[PROVE]` output for `examples/var1.prove`

```
1 [[a][b]][c]=>
2 [a]{8}<5><0>verification failed on line 1, column 17
```

Notice that statement 8 cannot be verified. It now has the `NEWC` flag set, indicating that a new constant has been introduced - it would be verifiable when adding one additional assumption: `[c=a]`.

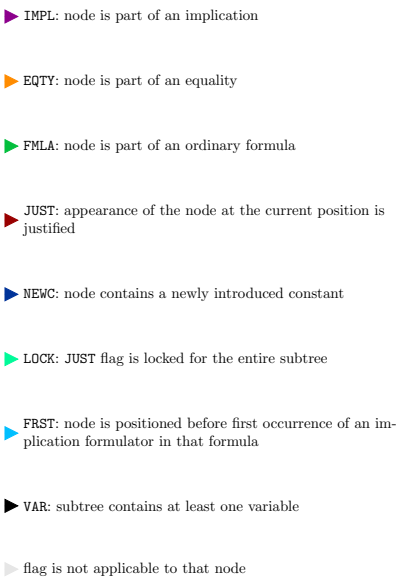
We might want to change this behaviour and allow new identifiers to be introduced anywhere in the code (i.e. also after a `=>` formulator). Then, Source Code 4 could successfully be verified - but for a different reason as if we would have used the suggested additional assumption.

Source Code 6: `examples/var2v1.prove`

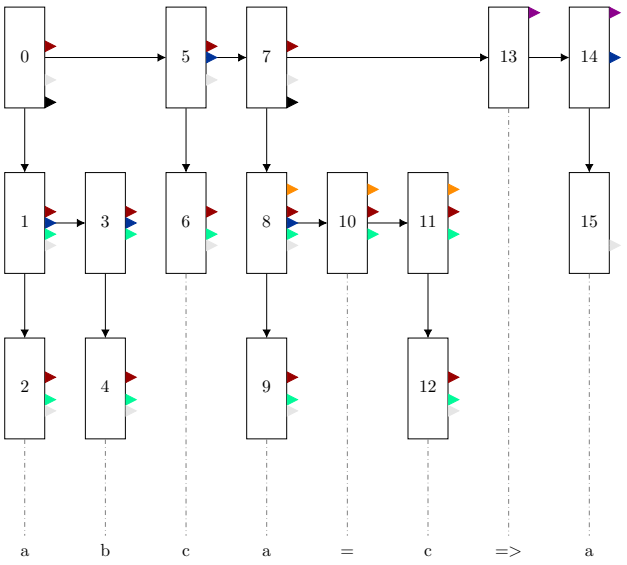
```
1 [[a][b]][c][[a]=[c]]=>[a]
```

Figure 6: Representation of source code 6

(a) Legend



(b) Graph of source code 6



Source Code 7: `[PROVE]` output for `examples/var2v1.prove`

```
1 [[a][b]][c][[a]=[c]]=>
2 [a]{14}<11#><8#><7><5><0>
```

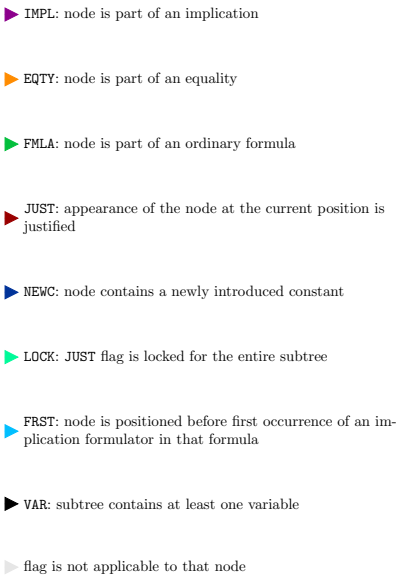
Here `[[a]=[c]]` is part of the initial assumptions. The verification is successful as node 14 (`[a]`) is found to be similar both nodes 11 and 8, which are equal by the statement of node 7.

Source Code 8: `examples/var3v1.prove`

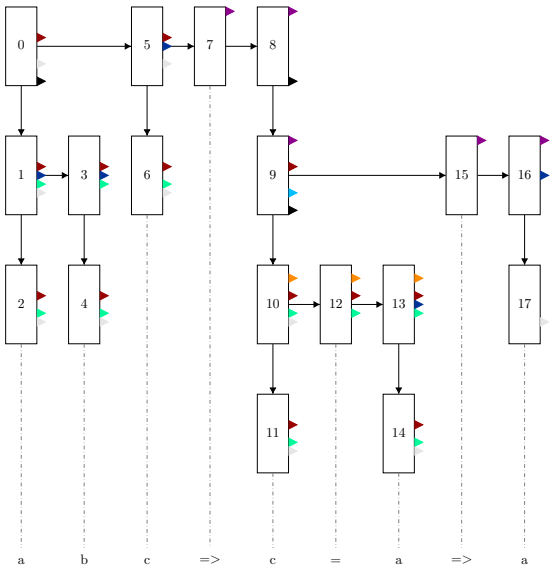
```
1 [[a][b]][c]=>[[[c]=[a]]=>[a]]
```

Figure 7: Representation of source code 8

(a) Legend



(b) Graph of source code 8



Source Code 9: `[PROVE]` output for `examples/var3v1.prove`

```
1 [[a][b]][c]=>
2 [[[c]=[a]]=>
3         [a]{16}<13#><10#><9><5><0>]
```

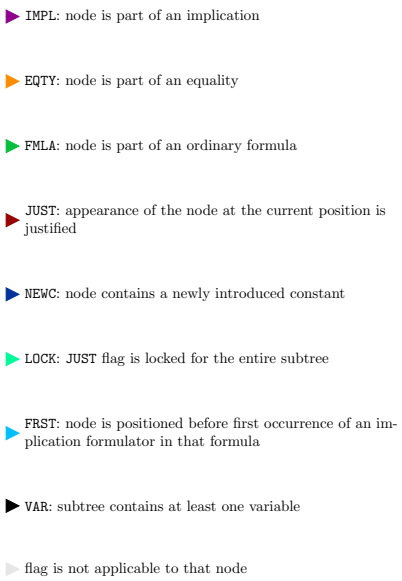
This is a variation of Source Code 6, where `[c=a]` is an assumption in an implication at a deeper level.

Source Code 10: `examples/var4v1.prove`

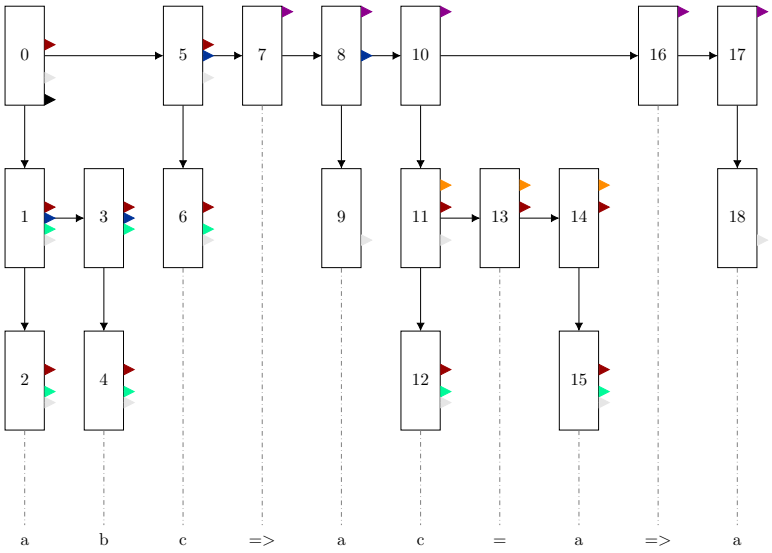
```
1 [[a][b]][c]=>[a][[c]=[a]]=>[a]
```

Figure 8: Representation of source code 10

(a) Legend



(b) Graph of source code 10



Source Code 11: `[PROVE]` output for `examples/var4v1.prove`

```
1 [[a][b]][c]=>
2 [a]{8}<5><0>verification failed on line 1, column 17
3 [[c]=[a]]{10}<8><5><0><0><0><0>verification failed on line 1, column 26
4 =>
5 [a]{17}<14#><11><10><8#><5><0><0><0><0><0><0><0><0><0>
```

Here, nodes 8 and 10 cannot be verified, since there are no nodes in eligible positions which they are similar to.

Source Code 12: `examples/var4v2.prove`

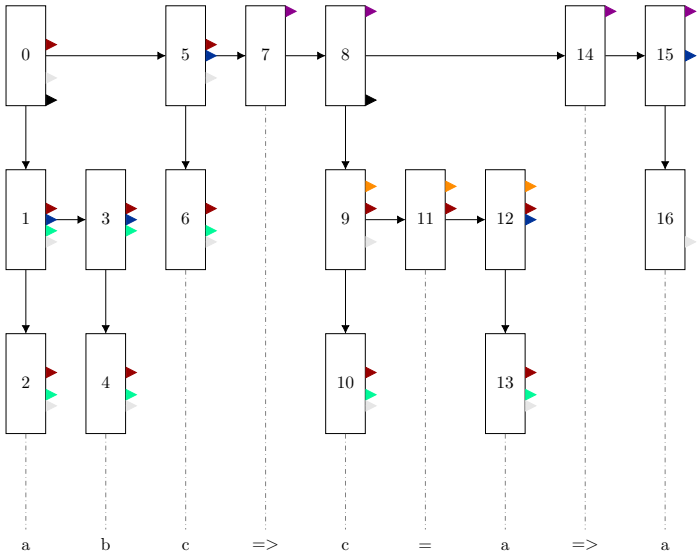
```
1 [[a][b]][c]=>[[c]=[a]]=>[a]
```

Figure 9: Representation of source code 12

(a) Legend

- IMPL: node is part of an implication
- EQTY: node is part of an equality
- FMLA: node is part of an ordinary formula
- JUST: appearance of the node at the current position is justified
- NEWC: node contains a newly introduced constant
- LOCK: JUST flag is locked for the entire subtree
- FRST: node is positioned before first occurrence of an implication formulator in that formula
- VAR: subtree contains at least one variable
- flag is not applicable to that node

(b) Graph of source code 12



Source Code 13: `[PROVE]` output for `examples/var4v2.prove`

```
1 [[a][b]][c]=>
2 [[c]=[a]]{8}<5><0>verification failed on line 1, column 23
3 =>
4 [a]{15}<12#><9#><8><5><0>
```

Here, same problems as in Source Code 10 arise.

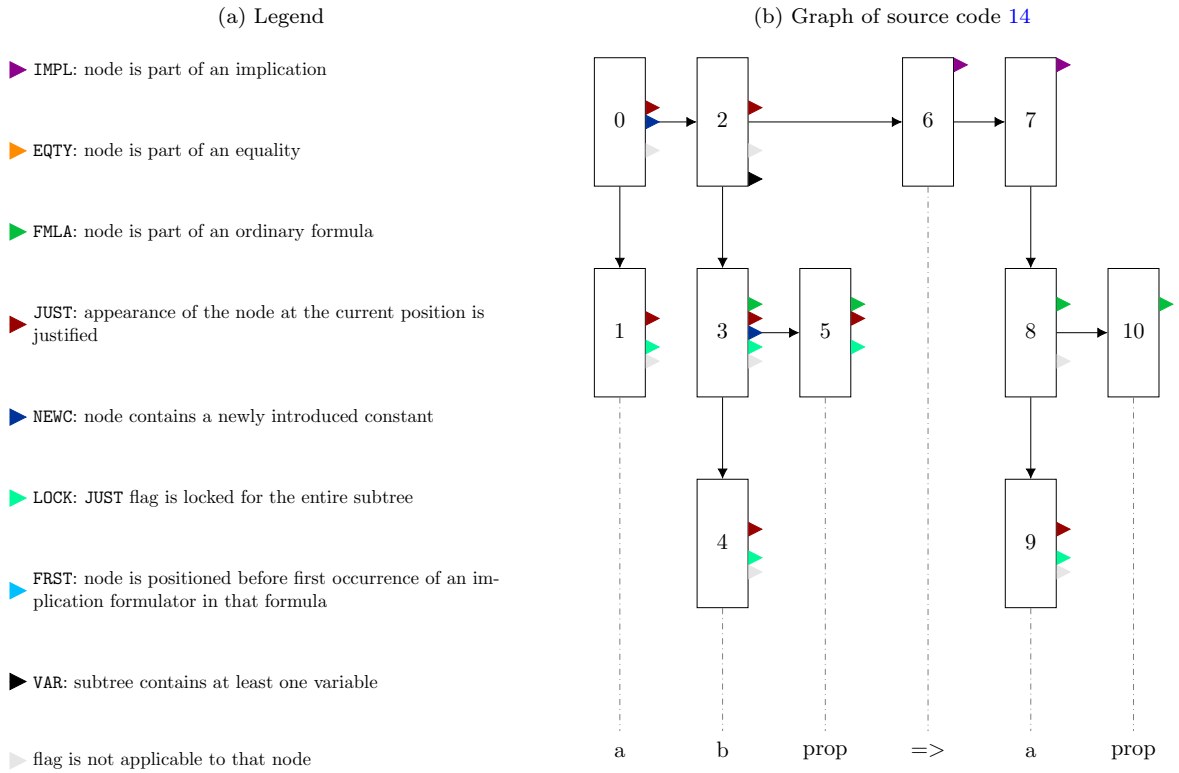
6.4 Reachability

As mentioned before verification of a node involves comparing the current node against a number of sub-trees.
To select the sub-trees the current node is compared against, we have to introduce the concept of *reachability*.

Source Code 14: Example

```
1 [a][[b]prop]=>[[a]prop]
```

Figure 10: Representation of source code 14



To determine which nodes are reachable from the current node, we go back in the tree node by node moving to the left and upwards. All of these nodes containing statements are reachable from the current node. Further, the following holds:

- If a reachable statement contains variables, all variables have to be substituted by reachable nodes containing no variables.

- If a reachable statement contains another statement, that statement is also reachable.
- If a reachable statement contains an implication, the conclusions of the implication are reachable, when the assumptions can be verified.
- If a reachable statement contains an equality, all statements in that equality are reachable, if at least one statement can be verified.

The verification of assumptions might depend on the substitution of variables.

Let us consider source code 14. Translated to English it says: There is an identifier `a` and there is a property `[b]prop`, which can be applied to any variable `b`. It follows that `a` has this property.

Now let us understand, how the `[PROVE]` code can be verified using the tree from figure 10: Nodes 0 and 2 and their respective sub-trees are assumptions and do not have to be verified. Node 7 is part of an implication and not already justified - in fact it is the only node in the tree, which satisfies these conditions - so it has to be verified. The nodes 0 and 2 are the only nodes, which are reachable from node 7. Node 0 is a constant statement (i.e. a statement not containing any variables), whereas node 2 is a variable statement. Before attempting to compare node 7 with node 2 the variable has to be substituted. The variable node in the sub-tree below statement 2 is node number 4 (`b`), which is to be replaced by the children of all constant statements (one at a time of course), which are reachable from node 7. In this case the only constant statement, which is reachable from node 7 is statement 0 (`[a]`), so we substitute node 4 (`b`) by the child of node 0, namely node 1 (`a`), before we compare the sub-tree under node 2 (now `[a]prop`) against the sub-tree under node 7 (`[a]prop`).

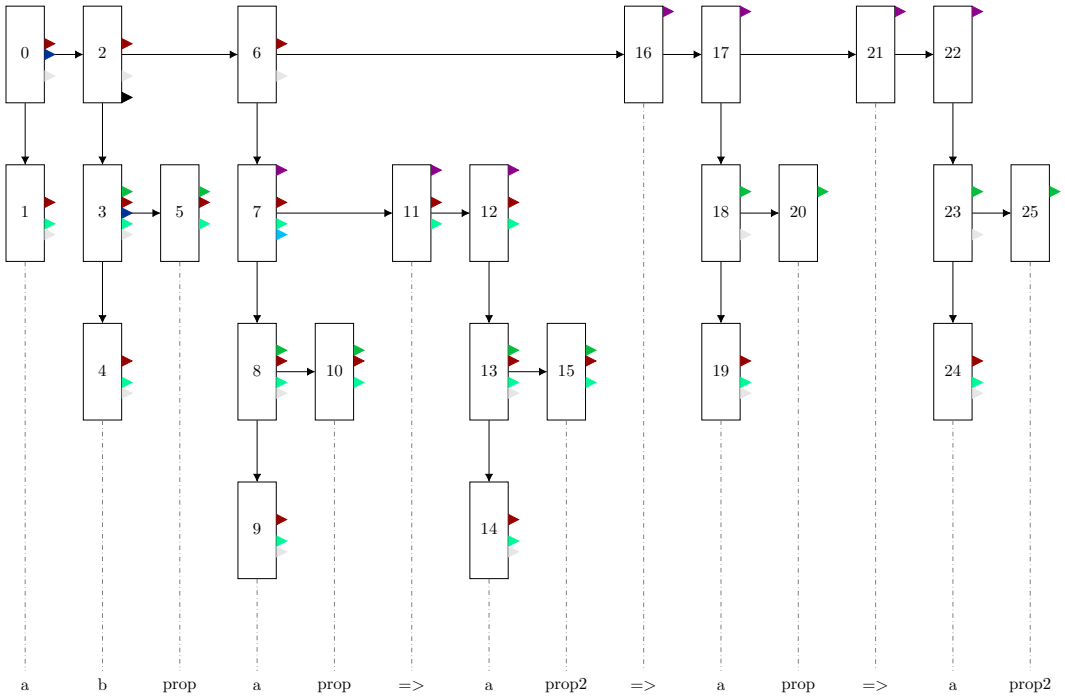
To get deeper insight in how the reachability in sub-trees is determined, let us look at a slightly more involved example. In source code 15 we make three assumptions: There is an identifier `a`, there is a property `[b]prop`, which can by applied to any identifier, and if `a` has this property, then it also has a second property. In this example the statements 17 and 22 require verification. Statement 17 can - like in the previous example - be justified by statement 2. For statement 22 however, we have to explore the sub-tree below node 6. Node 6 contains an implication, which depends on the assumption of statement 7, so before we can determine, whether any of the statements in implication 6 are reachable, we have to verify assumption 7. Assumption 7 therefore has to be compared against all constant nodes, which are directly reachable from node 22, namely nodes 17 and 0. It turns out that assumption 7 can be verified using statement 17, which lets us reach statement 12 from node 22, which we intend to verify. Since the statements (`[a]prop2`) in node 22 and node 12 are similar, the verification is successful.

Note that without statement 17, statement 22 could not be verified, since assumption 7 is compared against all *constant* statements, which are directly reachable from node 22.

Source Code 15: Example

```
[a][[b]prop][[[a]prop]=>[[a]prop2]]=>[[a]prop]=>[[a]prop2]
```

Figure 11: Representation of source code 15



References

International Organization for Standardization (1996). *ISO/IEC 14977 : 1996(E)*. URL: https://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf (visited on 24/01/2021).