[PROVE]

27th January 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. A formulator does not contain any square brackets but is delimited by a closing square bracket or the beginning of the file at its beginning and an opening square bracket or the end of the file at its end.
- An *identifier* is a sequence of legal characters and is delimited by an opening square bracket at its beginning and by a closing square bracket at its end.
- To refer to formulators or identifiers more generally, we will use the term *symbol* in this document.
- 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 from a list that may be extended. Opening and closing square brackets, and the literals used in 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 6.1).
- 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* symbols, 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

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

$\vert$	indicates ‘or’
(. . .)	groups productions and/or terminals together; only useful in combination with $\vert$
[. . .]	indicates optionality
{. . . }	indicates zero or more repetitions

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

$$
\begin{aligned}
\langle\texttt{expr}\rangle &= \langle\texttt{statement}\rangle \Big| \langle\texttt{formula}\rangle \\
\langle\texttt{formula}\rangle &= \Big\{ \langle\texttt{statement}\rangle\langle\texttt{formulator}\rangle \Big| \langle\texttt{formulator}\rangle\langle\texttt{statement}\rangle \Big\} \\
\langle\texttt{statement}\rangle &= \text{‘[’}\Big(\langle\texttt{expr}\rangle \Big| \langle\texttt{symbol}\rangle\Big)\text{‘]’}\Big\{\text{‘[’}\Big(\langle\texttt{expr}\rangle \Big| \langle\texttt{symbol}\rangle\Big)\text{‘]’}\Big\} \\
\langle\texttt{formulator}\rangle &= \text{‘=’}\Big|\text{‘=>’}\Big|\langle\texttt{symbol}\rangle \\
\langle\texttt{symbol}\rangle &= \langle\texttt{character}\rangle\Big\{\langle\texttt{character}\rangle\Big\} \\
\langle\texttt{character}\rangle &= \Big(\text{‘a’}\Big|\ldots\Big|\text{‘z’}\Big|\text{‘A’}\Big|\ldots\Big|\text{‘Z’}\Big|\text{‘0’}\Big|\ldots\Big|\text{‘9’}\Big|\text{‘+’}\Big|\text{‘-’}\Big|\text{‘/’}\Big|\text{‘*’}\Big|\text{‘\%’}\Big|\text{‘\^{}’}\Big|\text{‘\&’}\Big|\text{‘.’}\Big|\text{‘?’}\Big|\text{‘!’}\Big|\text{‘:’}\Big|\text{‘\_’}\Big)
\end{aligned}
$$

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

production is to be listed first, so in this case, ⟨expr⟩ 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.

3.1 Examples

Here are some examples of productions and some motivations for the design of the syntax. We will work our way up - from bottom to top - through the EBNF.

- A ⟨`character`⟩ can be any letter, number or any one of the special characters indicated in the EBNF. Examples of valid ⟨`character`⟩s are `a`, `T`, `7`, `&`.
- A ⟨`symbol`⟩ is a sequence of one or more characters. Examples of valid ⟨`symbols`⟩s are: `a7we`, `T`, `76`, `&q`. Note that `T` is a ⟨`symbol`⟩ and a ⟨`character`⟩ at the same time. This does not mean that the grammar is ambiguous. A single `T` will always be interpreted as a ⟨`character`⟩, which - when not followed or preceded by other ⟨`character`⟩s - is the only character of a ⟨`symbol`⟩. It depends on the context, whether we are interested in the fact of `T` being a ⟨`symbol`⟩ or `T` being a ⟨`character`⟩.
- A ⟨`formulator`⟩ is either a ⟨`symbol`⟩ or any one of the terminals (fixed, not "unpack-able" elements of an EBNF) `=` or `=>`. Examples of valid ⟨`formulator`⟩s are `a7we`, `T`, `=`, `=>`. Note that `=T` is not a valid ⟨`formulator`⟩. Seeing that `T` can be a ⟨`formulator`⟩ (containing of a ⟨`symbol`⟩) in one case and just a ⟨`symbol`⟩ (not packed inside a ⟨`formulator`⟩) in another case, one might worry about ambiguity. However, having a close look at the EBNF, we see that ⟨`formulator`⟩s only occur in between ⟨`statement`⟩s and ⟨`symbols`⟩s only within ⟨`statement`⟩s. The user of the language has great freedom in naming formulators and identifiers, so it is up to the user to choose them wisely. It could for example be confusing to name a formulator `7`, but `add7` makes intuitively sense.
- A ⟨`statement`⟩ is a concatenation of one or more pairs of square brackets each containing either a ⟨`formula`⟩, a ⟨`symbol`⟩ or a ⟨`statement`⟩. Examples of valid ⟨`statement`⟩s are `[]`, `[a][b][[a]in[b]]`, `[[]]`, `[[a]or[b]]`.
- A ⟨`formula`⟩ is comprised of at least one ⟨`statement`⟩ and at least one ⟨`formulator`⟩. The order of occurrence does not matter. Examples of valid ⟨`formula`⟩s are `[a]in[b]`, `[a]=[b][c]`, `[[a]a7we[b]]=>[c]`.

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

To determine, which expressions require a justification, we will also need a few rules:

- If an expression is an implication (i.e., all formulators are `=>`), then every statement occurring after the first formulator is not justified.
- Every expression in an equality (i.e., all formulators are `=`) is already justified. (Remark: An equality has all the properties a two-sided implication would have, but in addition to that it also allows for substituting one statement with another)
- If an expression is justified, all statements contained within it are justified as well.
- If and only if an expression is not justified and part of an implication, it requires verification (see Section 6).

Justified statements, which are part of an implication, will be called *assumptions*.

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

```
[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 symbol 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.

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 symbol, but not both. Processing the code bracket by bracket and symbol by symbol, we now perform the following steps:

- When encountering a `[`, we check, whether the current node has a child or carries a symbol. 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 symbol, the symbol 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

(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`: expression in node is already justified

🟦 `NEWC`: node contains a newly introduced constant

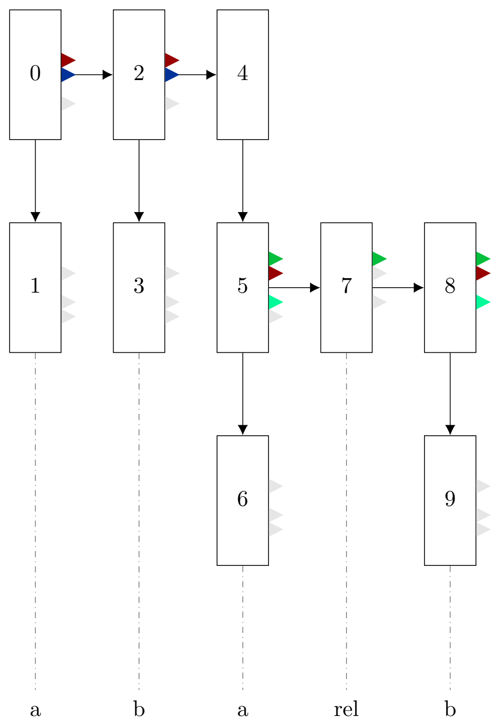
🟩 `LOCK`: `ASMP` flag is locked for the entire subtree

🟦 `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 1



The tree shown in figure 3 (b) was automatically generated by `[PROVE]` following the steps described above. The nodes have been number 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 `JUST` flag indicates that the statement contained by a node is an assumption. These flags are set and unset in a variable, which is part of the structure of the node in memory. This setup simplifies the verification process by storing already gained information and thereby avoiding repeated computation of frequently used steps.

Remark: Some of these flags are set for nodes, which they are not applicable to. This has technical reasons (mainly related to computational efficiency) and is nothing we have to worry about. Further, some flags are not set, where they could be. This has technical reasons as well. E.g. node 4 in figure 3 could carry the `JUST` flag. This flag would be set, if we would continue reading more code, but since node 4 is not part of an implication, we do not have to worry about it not being set.

6 Verification

Verifying a proof means, checking that every conclusion can be justified. With respect to `[PROVE]` code, we imposed certain conditions on statements to be an assumption. A conclusion is a statement satisfying the following conditions:

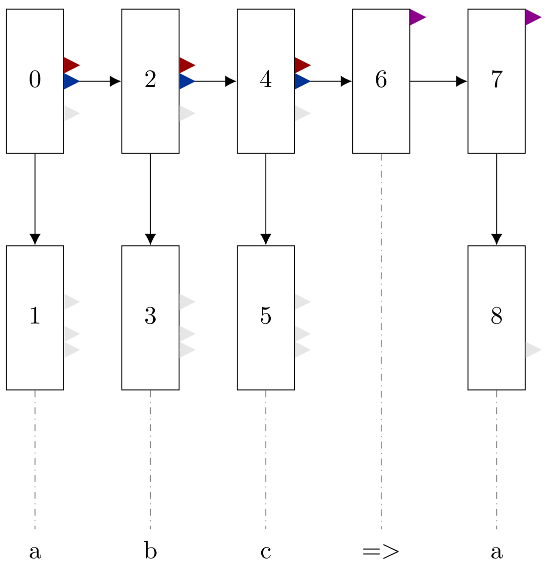
- The statement is not justified already.
- The statement must be part of an implication (i.e. a formula, which contains (only!) `=>` formulators).

To illustrate these ideas, let us now look at the following implication:

Source Code 2: Example

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

Figure 4: Representation of source code 2



Remark on figure 4: Nodes 0, 2 and 4 are part of an implication but do not have the `IMPL` flag set. This has technical reasons: The flags indicating the type of formula (i.e. `IMPL`, `EQTY` and `FMLA`) are set after encountering the first formulator. When stepping through the nodes leftwards to finally get to the parent level, these flags are copied over. Since this does not happen at the topmost level, these flags are not set. As we are not interested in these flags at those positions, we do not have to worry about that.

In source code 2 we make three assumptions: We state that there are three identifiers `a`, `b` and `c`. Note the flags in the graph telling us that the statements 0, 2 and 4 are already justified. The identifiers themselves are not expressions, so the JUST flags are not applicable to them. After the `=>` formulator, we have another expression stating that there is an `a`. This is an unjustified statement and requires verification. In order to verify statement 7, we compare the sub-tree with node 8 (node 7's child) as its root against the sub-trees with the nodes 1, 3 and 5 as their roots respectively. Two sub-trees are found to be similar, when the following two 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 symbol, the other node must carry the same symbol.

In the example of source code 2 the compared sub-trees consist of only a single node, each carrying a symbol. 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, namely the following:

- Variables and constants
- Reachability

6.1 Variables and constants

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 3: Example

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

Figure 5: Representation of source code 3

(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: expression in node is already justified

▶ NEWC: node contains a newly introduced constant

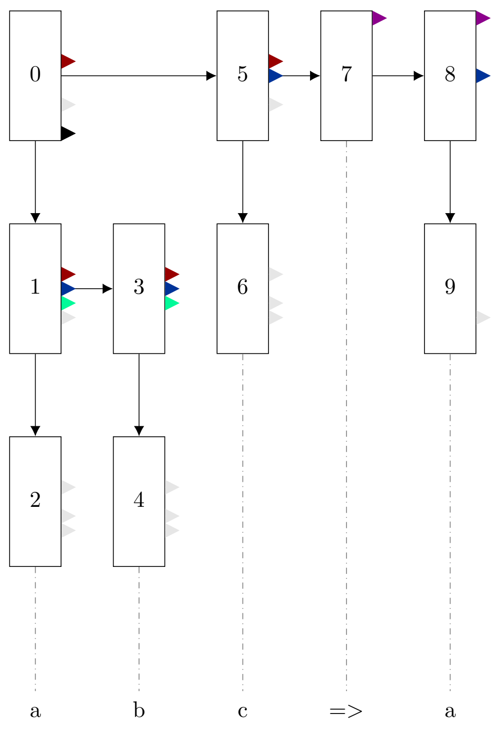
▶ LOCK: ASMP flag is locked for the entire subtree

▶ 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 3



Notice that statement 8 cannot be verified. It now carries the NEWC flag indicating that a new constant has been introduced - it would be verifyable when adding an intermediate step: `[c=a]`. Node 0 on now carries the VAR flag telling us that its sub-tree contains at least one variable. This is, because the constants, declared in statement 0, are only constants within that sub-tree: As can be seen in figure 5 (b), the nodes 1 and 3 - just as the nodes 0 and 2 in source code 2 - still carry the NEWC flag. For all nodes to the right of node 0 the identifiers `a` and `b` are treated as variables. They cannot be used directly, but have to be replaced by another sub-tree (i.e. statement) or symbol.

6.2 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 4: Example

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

Figure 6: Representation of source code 4

(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`: expression in node is already justified

🟦 `NEWC`: node contains a newly introduced constant

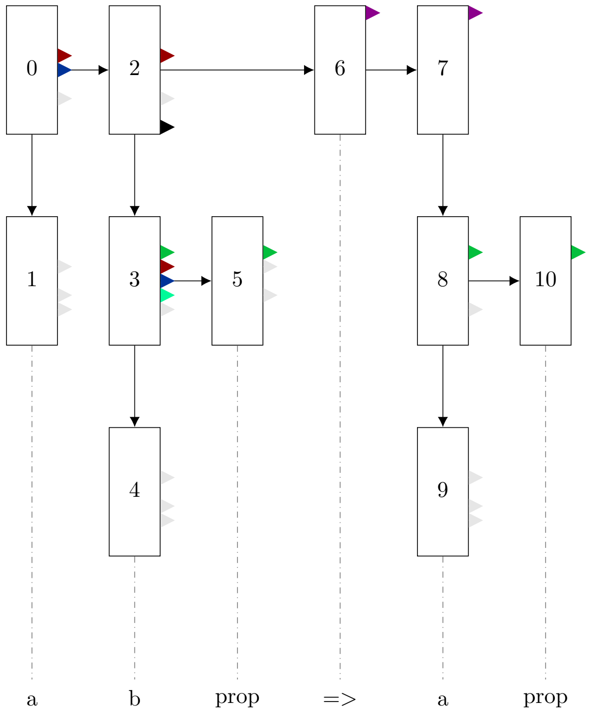
🟩 `LOCK`: `ASMP` flag is locked for the entire subtree

🟦 `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 4



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 4. 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 6: 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`).

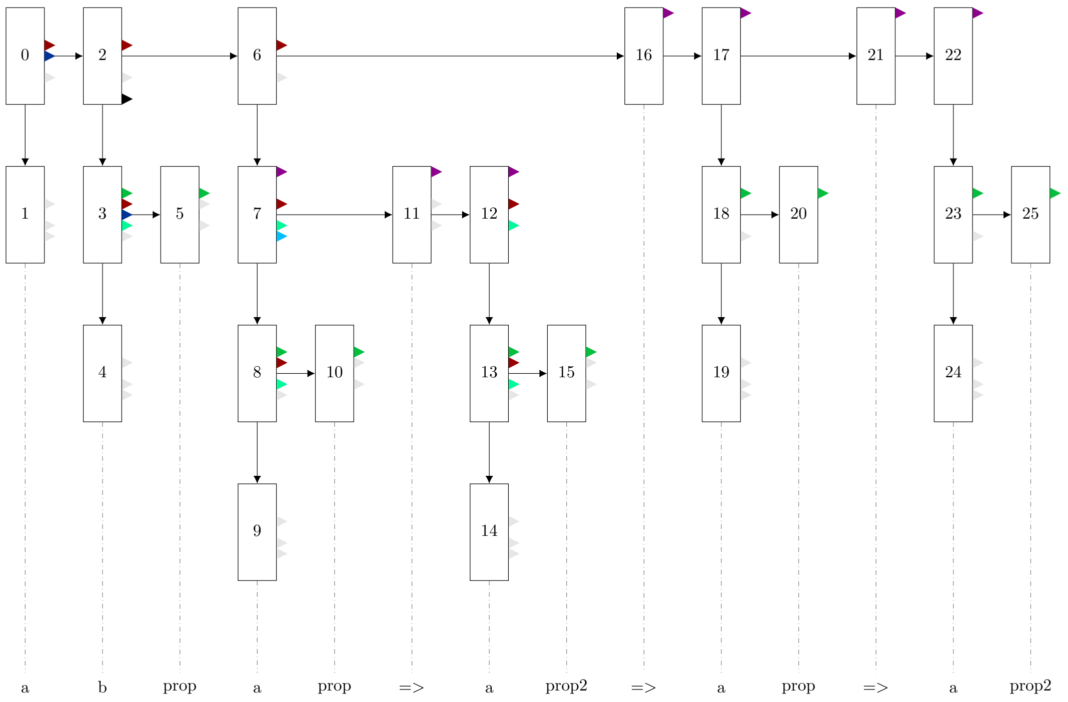
To get deeper insight in how the reachability in sub-trees is determined, let us look at a slightly more involved example. In source code 5 we make three assumptions: There is an identifier `a`, there is a property `[b]prop`, which can by applied to any identifier, and if `a` has this property, then 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 5: Example

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

Figure 7: Representation of source code 5



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).