[PROVE]

22nd December 2020

1 Introduction

This document is work in progress. All source code [1] as well as the most recent version of this document [2] can be found on GitHub. The purpose of this document to introduce notation and vocabulary in order to facilitate efficient communication about this project. Underlying concepts are explained verbally and/or formally in order to detect any possible misconceptions of the matter. Furthermore this document could serve as an introduction to the project, if somebody wants to join. Especially with respect to this last point it might be useful to include a short discussion of the following points at the beginning of this document (this is a TODO list):

- a short verbal explanation of the purpose and form of the [PROVE] language
- a list of terms we have agreed upon to describe certain entities, e.g. statement, formula, formulator, variable, constant, identifier, ... (we might have to have a look at our agreements again)
- to be continued ...

At the moment this document starts off right, where we ended our last discussion, so no great explanations of things that we have agreed upon are present at the moment.

2 Terminology

In our last discussion, we have agreed on the following terminology:

- statement
- formula
- formulator

Further I would like to suggest the following terms. Some of the given explanations might be a bit confusing at this point, but their meaning will become clearer at a later point in this document.

- character (all legal characters, which are not square brackets)
- symbol (a concatenation of characters, delimited by square brackets or the end or the beginning of the file)
- identifier (every symbol, which is not a formulator)
- constant (an identifier, which is not allowed to be replaced by another identifier)
- variable (an identifier, which must be replaced by another identifier)
- constant statement (a statement, which is not comprised of any variables)
- implication (a formula only containing implication formulators)
- equality (a formula only containing equality formulators)
- ordinary formula (a formula, which contains no implication or equality symbols)
- known identifier (an identifier, which is a constant in the context of the currently processed statement)
- reachable constant (an identifier or a constant statement, which can be validated - i.e. is reachable when searching backwards in the graph representation of the proof - given all the known identifiers and constant statements in the context of the currently processed statement)
- backtracking (searching backwards in the graph - which is a special form of planar tree - for reachable constants)
- branch exploration (searching forwards for reachable constants in sub-branches, while backtracking)

3 Syntax

3.1 EBNF

3.1.1 A first sketch

The syntax of the [PROVE] language can conveniently be described by means of the extended Backus-Naur form[3] (EBNF), which is a meta syntax notation defined in the ISO/IEC 14977 [4] standard. 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 following meta characters:

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

Using this notation the [PROVE] language can be described by the following EBNF:

$$\langle \texttt{formula} \rangle = \left\{ \langle \texttt{statement} \rangle \langle \texttt{formulator} \rangle \middle| \langle \texttt{formulator} \rangle \langle \texttt{statement} \rangle \right\}$$

$$\langle \texttt{statement} \rangle = \texttt{"["} \left(\langle \texttt{formula} \rangle \middle| \langle \texttt{string} \rangle \middle| \langle \texttt{statement} \rangle \right) \texttt{"]"} \left\{ \texttt{"["} \left(\langle \texttt{formula} \rangle \middle| \langle \texttt{string} \rangle \middle| \langle \texttt{statement} \rangle \right) \texttt{"]"} \right\}$$

$$\langle \texttt{formulator} \rangle = \texttt{"="} \middle| \texttt{"=>"} \middle| \langle \texttt{string} \rangle$$

$$\langle \texttt{string} \rangle = \langle \texttt{symbol} \rangle \left\{ \langle \texttt{symbol} \rangle \right\}$$

$$\langle \texttt{symbol} \rangle = \left(\texttt{"a"} \middle| \texttt{"b"} \middle| \ldots \middle| \texttt{"z"} \middle| \texttt{"A"} \middle| \texttt{"B"} \middle| \ldots \middle| \texttt{"Z"} \middle| \texttt{"0"} \middle| \texttt{"1"} \middle| \ldots \middle| \texttt{"9"} \middle| \texttt{"+"} \middle| \texttt{"-"} \middle| \texttt{"/"} \middle| \texttt{"*"} \middle| \texttt{"\%"} \middle| \texttt{"\^{}"} \middle| \texttt{"\&"} \middle| \texttt{"."} \middle| \texttt{"?"} \middle| \texttt{"!"} \right)$$

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, ⟨formula⟩ is the initial production. One might notice that a ⟨formula⟩ might be empty and a ⟨statement⟩ may contain such an empty ⟨formula⟩. We can reason about whether this makes sense or not, and it might be necessary to revise these definitions, but currently I just regard these situations as edge cases with no harmful implications.

3.1.2 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 ⟨`symbol`⟩ can be any letter, number or one of the 10 special characters as indicated in the EBNF. Examples of valid ⟨`symbol`⟩s are `a`, `T`, `7`, `&`.
- A ⟨`string`⟩ is a sequence of one or more symbols. Examples of valid ⟨`string`⟩s are: `a7we`, `T`, `76`, `&q`. Note that `T` is a ⟨`string`⟩ and a ⟨`symbol`⟩ at the same time. This does not mean that the grammar is ambiguous. A single `T` will always be interpreted in this way, but it depends on the context, whether we are interested in the fact of `T` being a ⟨`string`⟩ or `T` being a ⟨`symbol`⟩.
- A ⟨`formulator`⟩ is either a ⟨`string`⟩ or any one of the terminals `=` or `=>`. Examples of valid ⟨`formulator`⟩s are `a7we`, `T`, `=`, `=>`. Note that `=T` is not a valid ⟨`formulator`⟩. Seeing that `T` can be a ⟨`formulator`⟩ (comprised of a ⟨`string`⟩) in one case and just a ⟨`string`⟩ (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 ⟨`string`⟩s only within ⟨`statement`⟩s. The user of the language has great freedom in naming formulators and identifiers (TODO: define formulators and identifiers in this document, i.e. not the productions, but what they represent), 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 ⟨`string`⟩ 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`⟩e are `[a]in[b]`, `[a]=[b][c]`, `[[a]a7we[b]]=>[c]`.

3.2 Improving the sketch (and second thoughts on this improvement)

As intuitive and beautiful (due to its simplicity) this first sketch might look, there are a few problems with it. Firstly a proof might be given in form of a statement or in form of a formula. Secondly, when parsing a ⟨`statement`⟩ we might have to look far ahead after encountering the leading `[` in order to determine, whether we are handling a ⟨`formula`⟩ or a ⟨`statement`⟩. One way to address this issue is, to start parsing a ⟨`formula`⟩, after reading a `[`, but then, when not finding a ⟨`formulator`⟩, not to report an error, but to proceed with the steps that would follow when parsing a statement.

To encode this behaviour in the EBNF and to address the issue of having the proof as a whole in form of a statement or a formula, we can introduce a helper production ⟨`expr`⟩(for expression) containing either a ⟨`formula`⟩ or a ⟨`statement`⟩.

$$
\begin{aligned}
\langle \texttt{expr} \rangle &= \langle \texttt{statement} \rangle \Big| \langle \texttt{formula} \rangle \\
\langle \texttt{formula} \rangle &= \Big\{ \langle \texttt{statement} \rangle \langle \texttt{formulator} \rangle \Big| \langle \texttt{formulator} \rangle \langle \texttt{statement} \rangle \Big\} \\
\langle \texttt{statement} \rangle &= \texttt{"["} \Big(\langle \texttt{expr} \rangle \Big| \langle \texttt{string} \rangle \Big) \texttt{"]"} \Big\{ \texttt{"["} \ \Big(\langle \texttt{expr} \rangle \Big| \langle \texttt{string} \rangle \Big) \ \texttt{"]"} \ \Big\} \\
\langle \texttt{formulator} \rangle &= \texttt{"="} \Big| \texttt{"=>"} \Big| \langle \texttt{string} \rangle \\
\langle \texttt{string} \rangle &= \langle \texttt{symbol} \rangle \Big\{ \langle \texttt{symbol} \rangle \Big\} \\
\langle \texttt{symbol} \rangle &= \Big(\texttt{"a"} \Big| \texttt{"b"} \Big| \ldots \Big| \texttt{"z"} \Big| \texttt{"A"} \Big| \texttt{"B"} \Big| \ldots \Big| \texttt{"Z"} \Big| \texttt{"0"} \Big| \texttt{"1"} \Big| \ldots \Big| \texttt{"9"} \Big| \texttt{"+"} \Big| \texttt{"-"} \Big| \texttt{"/"} \Big| \texttt{"*"} \Big| \texttt{"\%"} \Big| \texttt{"\^{}"} \Big| \texttt{"\&"} \Big| \texttt{"."} \Big| \texttt{"?"} \Big| \texttt{"!"} \Big)
\end{aligned}
$$

Second thoughts: It might be worth considering to disallow proofs to be given in form of a statement. When it comes to *verification,* we will see that it makes a difference, whether a formula stands 'freely' or whether it is inside an assumed statement and might not require verification. Considering a free standing statement as an assumption would than render the whole proof as an assumption and prohibit its verification.

4 Scanning and Parsing

When processing a `[PROVE]` source file, we first convert the stream of characters the source file is comprised of to a stream of tokens, which have a meaning in the context of the grammar. The tokens can be enumerated by the following (quite self-explanatory) definition in `token.h`:

```
typedef enum {
    TOK_EOF,      /* end-of-file */

    TOK_LBRACK,   /* left bracket */
    TOK_RBRACK,   /* right bracket */

    TOK_IMPLY,    /* implication */
    TOK_EQ,       /* equality */
    TOK_SYM       /* symbol */
} TType;
```

This token stream is fed to the parser, which then validates the adherence to the EBNF as given above. Inside the parser functions, which are validating the syntactic correctness of the source file, further function calls and checks for semantic correctness and mathematical correctness will take place.

5 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, I can see 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. Therefore I recommend against the precedence approach.

Instead I suggest the following semantic rule, which will address this issue:

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

Consider the following code:

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

Translated to English this means 'Let a and b be identifiers, then we have an identifier c'. We do not know, where c comes from, since it was not introduced before. Therefore I suggest the following rule (we will see in the proof of Russell's paradox that we might have to relax this rule a little bit, when dealing with equalities):

- new identifiers must only be introduced before the first `=>` in a formula

Further I suggest the following rule for obvious reasons:

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

Side note: Every formula, involving formulators other than `=` or `=>` and comprised of n statements, can be viewed as an n-ary operator. In that sense single identifiers are formulae as well, defining a nullary operation. Maybe it is not necessary to make a difference between formulators and identifiers. One argument against that would be though that formulators never function as variables - well.. maybe they can (think of isomorphisms).

6 Representation

6.1 Example: Russell's Paradox

A first example, we considered, is the following proof of Russell's Paradox. The purpose of this [PROVE] code is to provide a valid proof as a starting point for investigating further, which rules a data type representation should follow and which properties it must have.

Source-Code 1: Russels Paradox

```
[A] [ [x] => [ [[x]in[A]] = [[[x]in[x]]=>[False]] ] ] =>
[ [A] => [[[A]in[A]] = [[[A]in[A]]=>[False]]] ] =>
[ [[A]in[A]] = [[[A]in[A]]=>[False]] ] =>
[ [[A]in[A]] =>
        [[[A]in[A]]=>[False]] =>
        [False]] =>
[ [[A]in[A]] =>
[False]] =>
[ [A]in[A] ] =>
[False]
```

In the following subsections, we will showcase a possible representation of the structure of this proof as a graph and the C implementation of this graph.

6.2 Graph representation

An intuitive approach for representing the structure of this proof is by means of a graph (in particular a special planar tree).

6.2.1 Nodes

Every graph is comprised of a set of nodes. The nodes used in the current realisation of the [PROVE] software, are represented by the following structure, which can be found in `pgraph.h`:

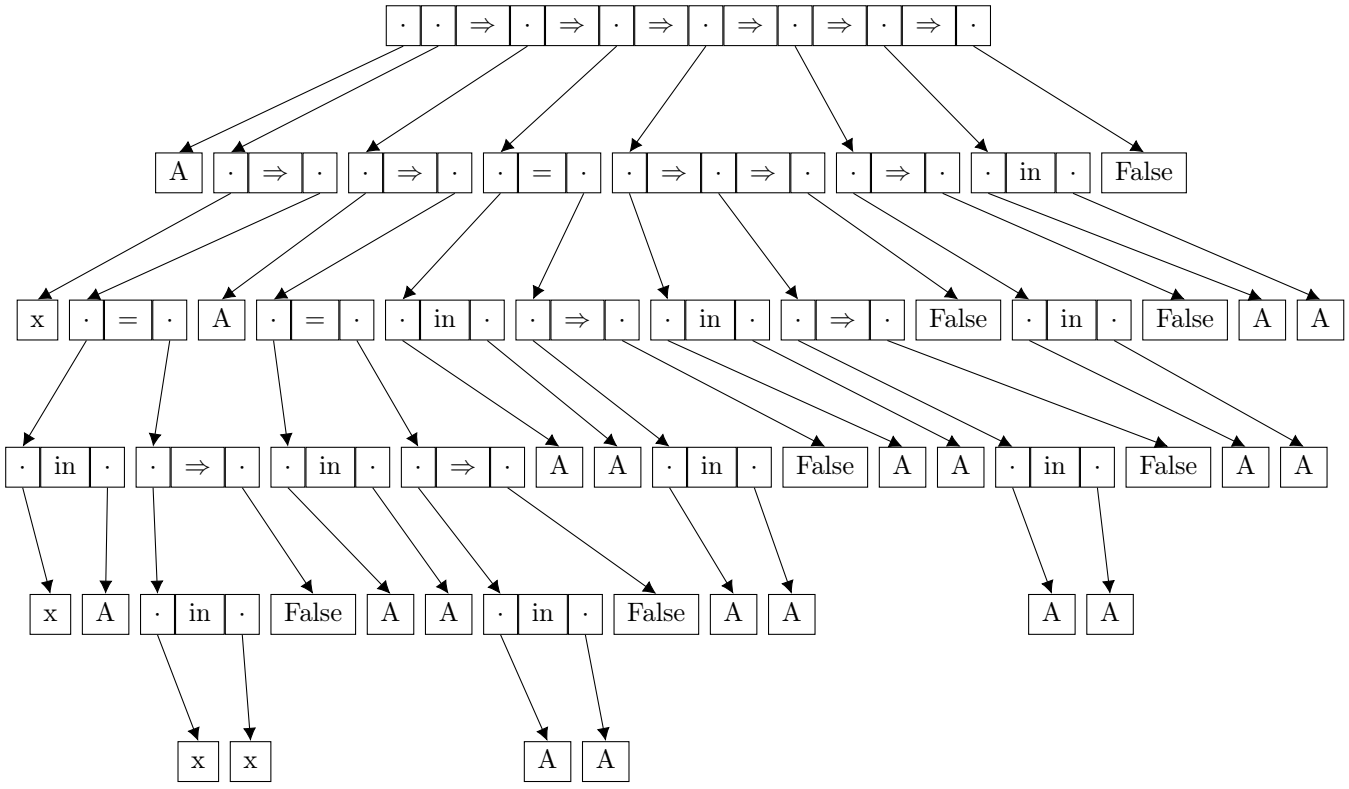
```
typedef struct Pnode {
        struct Pnode* parent;
        struct Pnode* child;
        struct Pnode* left;
        struct Pnode* right;
        struct Pnode* prev_const;/* link to previous constant in the tree */
        char** symbol; /* using a double pointer to let known identifiers
                                        point to the same char* in memory */
        NFlags flags;
        int num; /* number of the current node in pre-order traversal of the tree */
        Variable* var; /* link to the first variable in
                                        sub-tree (for substitution) */
} Pnode;
```

The nodes are each doubly linked the nodes adjacent to them in order to enable us to navigate through the graph. Every pair of brackets is represented as a node. During the creation of the graph nodes are created as children or to the right of the current node. When encountering an opening bracket, a child-node is created and the child becomes the current node. When a closing bracket is encountered, we move leftwards through the nodes as far as possible and then to the parent node. When encountering the next opening bracket, we check whether the current node has already children and if so, we create a node to the right of the current one and move to it, before creating a new child node.

6.2.2 Graph

Using the previously introduced nodes and applying the rules for the creation of the graph to Russell's Paradox (Source-Code 1), we can create the following graph:

Figure 1: Graph representation of Source-Code 1



6.3 Flags

6.3.1 NFLAGs

For the later verification of the proof it is useful to store all necessary information about each node in the corresponding structure. We can do this by means of flags, which are integer values of powers of 2. Using bitwise binary operators these flags can be set and unset for each node respectively. The enumeration defining the node specific flags (NFLAGs) can be found in `pgraph.h`:

```
typedef enum {
        NFLAG_NONE = 0,
        NFLAG_IMPL = 1,
        NFLAG_EQTY = 2,
        NFLAG_FMLA = 4,
        NFLAG_ASMP = 8,
        NFLAG_LOCK = 16,
        NFLAG_NEWC = 32,
        NFLAG_FRST = 64
} NFlags;
```

For convenience we say that a node is an assumption, if the child statement or formula is an assumption. The flags have the following meaning:

Flag	Meaning
IMPL	The node is part of a formula, containing implication formulators.
EQTY	The node is part of a formula, containing equality formulators.
FMLA	The node is part of an ordinary formula.
ASMP	The node is an assumption (only of interest in implications).
LOCK	The ASMP flag is 'locked', i.e. all nodes at lower levels are also assumptions.
NEWC	The node contains a newly introduced constant (identifier).
FRST	The node appears before the first formulator in a formula.

Note that only certain combinations of NFLAGs can occur. For example at most one of the flags IMPL, EQTY and FMLA must be set. Furthermore some flags might be set or unset in certain context, where they do not make sense, but they are not of interest.

For example:

- The FRST flag might be set or unset in a node, which is part of a statement. Since the FRST flag is only of interest in a formula, we disregard this setting to keep the code simple.
- The ASMP flag is set for all statements occurring before the first formulator, since the first statement in an implication is an assumption. It is also set in ordinary formulae, but we disregard it.

One important fact about the ASMP flag is that it 'trickles down' the tree, i.e. if a node has the ASMP flag set, then its children will also have the ASMP flag set. This behaviour is ensured by the LOCK flag.

6.3.2 GFLAGs

When traversing the graph at a later stage for verification, we might want to store the current state of the graph during this process. This state is not specific to any node, but to the graph as a whole. We can therefore call the flags used for this purpose GFLAGs. Their definition can also be found in `pgraph.h`:

```
typedef enum {
        GFLAG_NONE = 0,
        GFLAG_VRFD = 1,
        GFLAG_SUBD = 2,
        GFLAG_BRCH = 4,
        GFLAG_WRAP = 8
} GFlags;
```

The flags have the following meaning:

Flag	Meaning
VRFD	The last statement at a lower level was verified (no further verification required).
SUBD	A variable in the currently processed branch has been substituted.
BRCH	A branch in currently traversed.
WRAP	Nodes in a formula 'wrap around' when moving rightwards (useful for equalities).

6.4 Differences between implication and equality

This subsection requires some more investigation and discussion.

`=>`	`=`
New identifiers become variables at parent level.	New identifiers are unknown at parent level.
unidirectional	bidirectional

6.5 Variables and constants

Identifiers can be either variables, constants or unknown depending on the context (i.e. the node from whose perspective we intend to make use of that identifier). In the following, we will say that a node is *below* another node, when it is its child or below its child. Further we will say that a node is a *younger sibling* of another node, if it is right of that node or right of a younger sibling of that node. We can now define the following:

- An identifier is a constant for all nodes, which are below the younger siblings of the parent node of the identifier.
- An identifier is a variable for all nodes, which are below the younger siblings of the parent node of the parent node of the identifier.
- An identifier is unknown for every other node.

7 Verification

Each statement potentially has to be verified. Since some statements are assumptions and others are just a part of an ordinary formula, we have to decide, which statements we have to verify and which not to. A reasonable approach seems to only verify implications, which are not assumptions (equalities are a bit tricky - this might require some more discussion). Following this idea, we can trigger verification at the end of the parse function for a ⟨`statement`⟩ in `pparser.c` (the code has been stripped of the debugging output to improve its readability):

```
1 /* verification is triggered here */
2 if (HAS_NFLAG_IMPL(pnode) && !HAS_NFLAG_ASMP(pnode)) {
3         init_reachable(pnode);
4         while (next_reachable_const(pnode)) {
5                 if(same_as_rchbl(pnode)) {
6                         SET_GFLAG_VRFD
7                 }
8         }
9 }
```

What happens here is the following:

- The current node (`pnode`) is checked for having the IMPL flag set and the ASMP flag unset.
- In line 3 the verification is initialised.
- From line 4 to line 8, we loop through all nodes, which are reachable from the perspective of `pnode` .
- If the current node is similar (TODO: define this properly - but for now it's quite obvious) to the reachable node currently inspected (line 5), then set the VRFD global flag.

The core of this verification lies in the function in the following function, which can be found in `pgraph.h`:

Source-Code 2: next reachable constant statement

```
1  unsigned short int next_reachable_const(Pnode* pnode) {
2          /* branch exploration */
3          if (HAS_GFLAG_BRCH) {
4                  if (!next_in_branch(pnode)) {
5                          exit_branch();
6                          UNSET_GFLAG_BRCH
7                  }
8                  return TRUE;
9          }
10
11         /* substitution */
12         if (HAS_GFLAG_SUBD){
13                 if (next_known_id()) {
14                         sub_vars();
15                         return attempt_explore(pnode);
16                 } else {
```

```
                        finish_sub();
                        return next_reachable_const(pnode);
                }
        }

        /* backtracking */
        if (move_left(&reachable) ||
                         (move_up(&reachable) && move_left(&reachable))) {
                if (HAS_SYMBOL(reachable) ?  move_left(&reachable) :  TRUE) {
                        if (reachable->var != NULL) {
                                init_sub(pnode);
                                sub_vars();
                        }
                        return attempt_explore(pnode);
                }
        }
        return FALSE;
}
```

The verification can be subdivided into three steps: backtracking, substitution and branch exploration. Each of these steps work on a pointer `reachable` pointing to the potentially next reachable node.

7.1 Backtracking

During backtracking `reachable` moves leftwards and upwards in the tree (there is always only one of these options available) skipping all formulators on the way. After each move it is determined, whether the node contains any variables. If so, substitution is initialised. Next an attempt to explore the branch (the subtree below the node pointed to by `reachable`) is started. If it is unsuccessful, the next reachable node has been found. Otherwise branch exploration is started.

7.2 Substitution

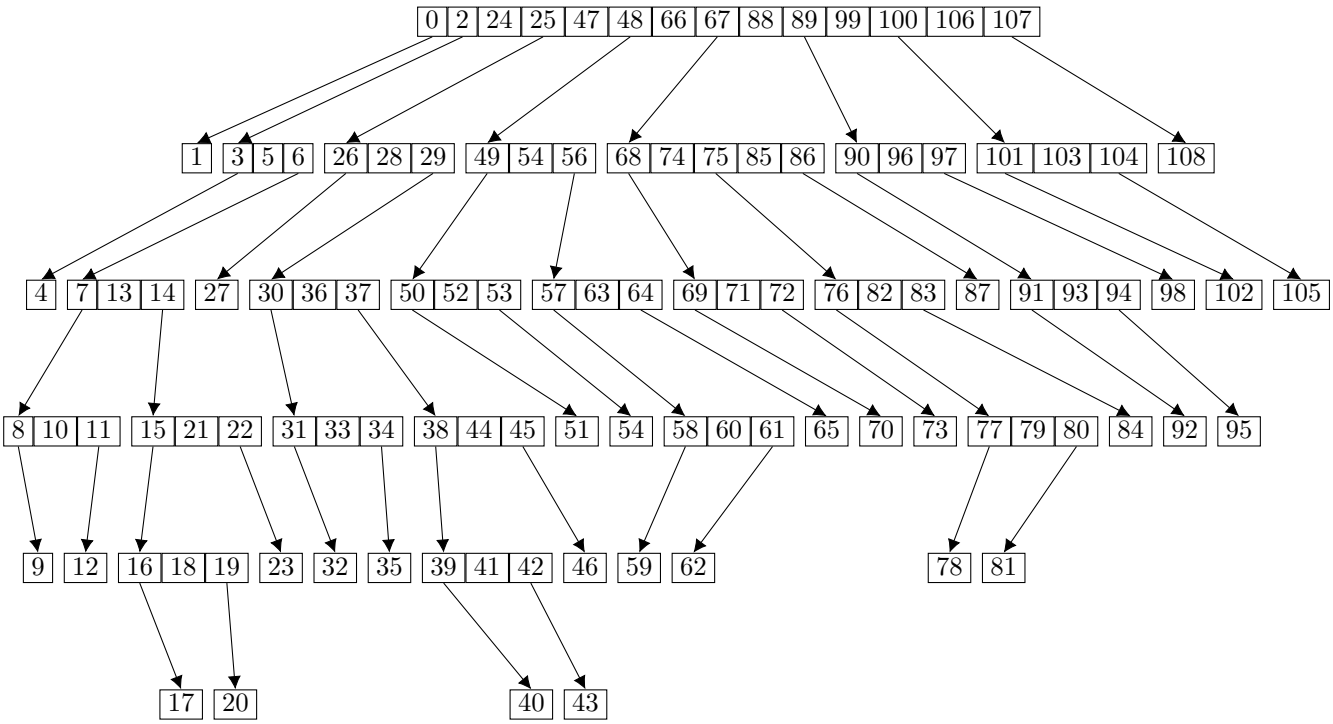
If during backtracking it was determined that substitution is needed, we loop through all known identifiers and substitute each of them into the node currently pointed to by `reachable`. For each substituted identifier an attempt of branch exploration is started.

7.3 Branch exploration

A branch is said to be explorable, if the assumptions 'guarding' (I like this word, maybe it should be introduced to the terminology) the branch are satisfied by any of the known constant statements. In that case the branch is traversed in the opposite direction as during backtracking. Every statement in the branch not depending on an unsatisfied assumption is then reachable.

8 Output

Figure 2: node numbers in pre-order traversal of figure 1



Source-Code 3: `[prove]` output after processing Source-Code 1

```
[A][[x]=>[[[x]in[A]]=[[[x]in[x]]=>[False]]]]=>
[[A]=>
        [[[A]in[A]]=[[[A]in[A]]=>[False]]]{29}<26><6#><2><6#><2><0>]*{25}<6><2#><0>=>
[[[A]in[A]]=[[[A]in[A]]=>[False]]]{48}<29#><25><6#><2><0>=>
[[[A]in[A]]=>
        [[[A]in[A]]=>
                [False]{83}<76><68><56><64#><49><48><29><37><45#><30><25><6><14><22#><7><2>
                <0>]*{75}<68><56#><64><49><48><29><37#><45><30><25><6><14#><22><7><2><0>=>
        [False]{86}<83#><75><68><56><64#><49><48><29><37><45#>
        <30><25><6><14><22#><7><2><0>]*{67}<29><25><6><2><0>=>
[[[A]in[A]]=>
        [False]{97}<90><75><83#><86#><67><56><64#><49><48><29><37><45#>
        <30><25><6><14><22#><7><2><0>]*{89}<49><56#><48><29><30><37#><25><6><7><14#><2><0>=>
[[A]in[A]]{100}<49#><56><48><29><30#><37><25><6><7#><14><2><0>=>
[False]{107}<100><97#><89><75><83#><86#><67><56><64#>
<49><48><29><37><45#><30><25><6><14><22#><7><2><0>
```

Currently the [PROVE] software produces hardly more than debugging output. It can be very helpful though for understanding the process of verification. To refer to the nodes of the graph, the nodes are numbered in pre-order traversal [6] as illustrated in figure 2. The corresponding output (Source-Code 3 [some line breaks have been added to improve readibility]) can be understood as follows:

- When verifying a node, [PROVE] prints the number of the node within curly braces (e.g. `{83}`).
- The number of every reachable node that the node to be verified is checked against is printed within angle brackets (e.g. `<76>`).
- If the verification was successful, a hash is added to the number of the reachable node (e.g. `<64#>`).
- If verification of the current node is not necessary, because the VRFD flag is set, a star is printed before the node number (e.g. `*{67}`

Currently no verification steps are skipped and nodes are often verified multiple times. This is desired at the current stage for debugging purposes and to illustrate the underlying verification mechanism.

Most semantic rules are also still not checked and some of them still have to be defined. One very important issue, which I would like to discuss, is how identifiers are allowed to be introduced. For the verification of this proof I had to relax a few restrictions that I have originally imposed on this issue, but without further elaboration of this, I suggest a proper discussion of the matter.

References

[1] https://github.com/g-regex/prove

[2] https://github.com/g-regex/prove/blob/master/doc/doc.pdf

[3] https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form

[4] https://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf

[5] https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form

[6] https://en.wikipedia.org/wiki/Tree_traversal#Pre-order_(NLR)