V – A Formal Mathematical Language

Short paper

Victor Makarov

February 19, 2023

Abstract

The V language is based on ZFC set theory and intended for use mostly in education. So every proof step is quite simple and can be checked by a human. The most important syntactic construct in V is the class term. Class terms are used for defining theories, the lambda, set builder, and restricted quantification notations. For testing the V language the following 3 modules (modules in V are similar to articles in Mizar) were written:

- root: elementary set theory: 5134 lines, 143 proofs, 47 equational proofs; some of the theorems were declared as axioms, they have to be proved later;
- group: the basics of group theory, including the isomorphism theorems: 5402 lines, 508 proofs, 326 equational proofs;
- altg: the basics of the permutation theory, necessary for defining the alternating group: about 1700 lines, 136 proofs, 60 equational proofs;

Also, a computer program (a proof checker) has been written(26300 lines in C++), which now is working in an experimental mode. The 3 modules have been successfully checked by the proof checker. The processing times for the root, group, and altg modules are 35sec,53sec, and 14sec respectively on the laptop Lenovo ThinkPad Yoga 11e 5th Gen.

The modules can be downloaded at [2].

1 Introduction

The V language is based on ZFC set theory [6]. Jean Dieudonne, one of the most prominent mathematicians of the last century stated [5, p. 215]:
"The theory of sets, so conceived, embraces all mathematical theories, each of which is defined by the assignment of a certain number of letters (the "constants" of the theory) and relations that involve these letters (the "axioms" of the theory): for example, the theory of groups contains two constants, G and m (representing respectively the set on which the group is defined, and the law of composition), and the relations express first that m is a mapping from G x G to G, and second the classical properties of the law of composition."
In other words, any mathematical theory is an extension of the set theory. See also [8, 4].

Note, that some relatively novel mathematical theories such as category theory may need to use additional axioms such as Tarski's axiom[7].

One of the most successful formal mathematical languages based on set theory is the Misar language [7]. But the Mizar language is rather complex; in particular, its context-free grammar has about 165 nonterminals [1]. It seems, that the V language is simpler: see a context-free grammar for V at [3].

The (relative) simplicity of the V language has been achieved, in particular, by using a uniform syntax for the lambda, set-builder, and restricted quantification notations. Also, the V language has no special construct for proving by cases – it can be done by using the existing syntactic constructs(see Section 5.1).

A remote ancestor of the V language is the MSL language [10]

2 Modules

All mathematical knowledge (i.e. definitions and proofs) is kept in *modules*(modules are similar to articles in Mizar). In the computer, each module is kept in a separate file with the extension .v.

A module has the following form (the V keywords are in bold):
module M;
use $M_1, ..., M_k$;
[body]

Here: M: the *module name*(a nonempty sequence of digits and letters);
M_i: the names of modules used in the module M; $i = 0, ..., k; k \geq 0$. These modules are called *the parent modules* of the module M.
body: a sequence of the main constructs: definitions, theory contributions, proofs.

3 Theories

In V, theories are introduced using *class terms*[9]. A class term in V has the form

$$\{x_1, ..., x_k; A_1; ...; A_m\}$$

where x_i is a name, called the name defined in the class term, $i = 1, ...k, k \geq 1$;
A_j are formulas, $j = 1, ...m, m \geq 1$, called the axioms of the class term(some axioms can actually be axiom schemata).

In a standard notation, it denotes the class term $\{x_1, ..., x_k | A_1 \& ... \& A_m\}$. Note, that this notation makes it difficult to parse {x—P} if the symbol '—' is an infix operator, because in this case, {x—P} may denote a singleton.

Not every class term denotes a set [9]. If it does, it is called a *set term*.

A theory T, which is an extension of ZFC, and has the primitive names $c_1, ..., c_k, k \geq 1$, and the axioms $A_1, ..., A_m, m \geq 1$ can be introduced as $T := \{c_1, ..., t_k; A_1, ..., A_m\}$.

For example, group theory is introduced in V as follows:

```
group := {elg, * ;    Axm := * in fn(elg#elg, elg);
  AxAssoc:= A[x,y,z: elg, x*(y*z) = (x*y)*z ];
  Axeinv := Ex[{e,inv; Axe := e in elg; Axinv := inv in fn(elg,elg)},
   (Axre := A[x:elg, x * e = x]) & (Axrinv := A[x: elg, x * inv(x) = e])];
}; // end group
```

The construct `x,y,z: elg` in the axiom AxAssoc will be converted automatically to the class term {x,y,z; x in elg; y in elg; z in elg }. The restricted quantifier (Ex[) makes e, inv visible in the enclosing scope (i.e. in group).

For defining theories, also *restricting class terms* can be used. For example, finite group theory can be introduced as fingroup:= group && (Axfin:= fin(elg)) where fin(elg) means "elg is a finite set".

For developing a theory T, *theory contributions* are used. They also can be called *T-contributions*. They have the form T:: [body], where T is a theory name, and the body is a sequence of the main constructs. The syntax of this construct (as well as all language design) was influenced by the C++ programming language [11].

The body can use the primitive names of the theory T, as well as the names defined in the previous T-contributions, more generally, all the names whose scope is T. Such names are called *T-names*. For example, the names elg,*,e, and inv are some group-names

A *T-model* is any term z such that the formula $z \in T$ can be proved. If N is a T-name, z is a T-model, then the dot term $N.z$ denotes the value of N in the T-model z. For example, if z is a group (i.e. a pair (G, f), where G, f are some terms), then $e.z$ denotes the unit of the group z and $elg.z$ denotes G.

4 Introducing basic logical and set-theoretic notation

For introducing basic logic and set-theoretic notation *name declarations* are used.

A name declaration has the form dcl$[N, t_1, ..., t_k, t_r]$; and determines the syntax of using the name N; Here: N: a name (an identifier or an operator); t_i: the type of i-th argument,$i = 1, ..., k, k \geq 0$; t_r: the result type.

The name declaration allows to construct the *dcl-terms* which, up to abstract syntax, have the form N$(z_1, ..., z_k)$, and the type of the dcl-term is t_r (if the arguments $z_1, ..., z_k$ fit to the corresponding types $t_1, ..., t_k$). For example, if p, q are variables of the type bool, the type of the dcl-term $p \& q$ is bool(see the

declaration of '&' below), and because the formulas are the terms of the type bool, it also is a formula. The infix notation $p\&q$ is possible because '&' is an infix name in V [3].

A type denotes a nonempty collection: some examples of types: are any, set, bool, int, nat, and abterm(tells that a class term can be at this place).

The following name declarations are taken from the module root:

```
dcl[bool,set];
dcl[true, bool];
dcl[false, bool];                // this is a comment;
dcl[~, bool, bool];              // negation          \~  \lnot
dcl[&, bool, bool, bool];        // conjunction       \&
dcl[or, bool, bool, bool];       // disjunction       \lor
dcl[->, bool, bool, bool];       // implication       \to
dcl[==, bool, bool, bool];       // equivalence       \equiv
dcl[~==, bool, bool, bool];      // anti equivalence  \not\equiv
dcl[xor, bool, bool, bool];      // exclusive or      \oplus
dcl[Exist, bvar, bool, bool];    // Exist(x,P(x)):   exists x such that P(x);
dcl[Exist1, bvar, bool, bool];   // Exist1(x,P(x)):  exists exactly one ...;
dcl[Existx, bvar, bool, bool];   // Existx(x,P(x)):  x is visible also in
dcl[Exist1x, bvar, bool, bool]; // Exist1x(x,P(x)): the enclosing scope;
dcl[All,bvar, bool, bool];       // All(x,P(x)theorem;
dcl[=,any,any,bool];             // equality;
dcl[~=,any,any,bool];            // inequality;
dcl[!!, bool, bool];             // is an axiom; below d denotes a class term;
dcl[A[,abterm,bool,bool];        // the restricted universal quantifier;
dcl[E[,abterm,bool,bool];        // the restricted existential quantifier);
dcl[Ex[,abterm,bool,bool];       // Ex[d,P]: d-names are visible ...
dcl[E1[,abterm,bool,bool];       // E1[d,P]: Exists exactly one ...
dcl[F[,abterm,any,FN];           // Lambda notation: F[d,g];
dcl[R[,abterm,any,set];          // R[d,g] = im(F[d,g]);
dcl[U[,abterm,set,2];            // 2: type(U[d,w]) = type(w); U[d,g] = union(R[d,g]);
dcl[!, bool, bool];              // possibly, a theorem;
dcl[II,set,set];                 // II(A) intersection of all sets in A
dcl[{}, set ];                   // the empty set
dcl[<: ,set,set, bool];          // set inclusion
dcl[<<, set, set, bool];         // strict set inclusion
dcl[\/ ,set,set, set];           // union of two sets
dcl[/\ ,set,set, set];           // intersection of two sets
dcl[--, set,set, set];           // set difference
dcl[union,set, set];             // union of the set
dpsetset := dcl[#,set,set,set]; // direct product
dcl[~, set,set,bool];            // eqiupollent
dcl[in,any,class,bool];          // \in
dcl[int, set];                   // the integers
dcl[nat, set];                   // the set of the natural numbers;
dcl[arb, set, any];              // !! Axarb := S ~= {} -> arb(S) in S;
dcl[nemp,set,bool];              // nemp(S) == S ~= {};
dcl[most1,set,bool];             // most1(S): S has at most one element;
dcl[atl2,fn(set,bool)];          // atl2(X): X has at least 2 elements;
dcl[one,set,bool];               // one(S): S has exactly one element;
dcl[the,set,any];                // !! Axthe := one(S) -> the(S) in S;
```

If P,Q are formulas, d is a class term of the form $\{x_1, ..., x_k; Q\}$ then A[d,P] and E[d,P] denote the formulas $All(x_1, ...All(x_k, Q \to P)...)$ and $Exist(x_1, ...Exist(x_k, Q\&P)...)$ respectively.

The parent of a name declaration in the abstract syntax tree(AST) can be only the module, the theory contributions or the proof.

The formulas Existx(x,P(x)), Exist1x(x,P(x)), Ex[d,P] and E1x[d,P] can only be used in proofs and theory

contributions and the parent of such a formula in the AST must be the proof or the theory contribution itself. They mean that the name x and the d-names will be visible in the enclosing scope.

5 Proofs

A proof has the following form:
Proof T;
¡Proof steps¿
end Proof T;
where T is the name of a formula to be proved.

There are the following proof steps: is steps, by steps, byeq steps, and assume steps.

An is step has the form F; **is** J; where F is a formula, and J is a justification - a keyword or a formula. Some real examples(taken from the module root) of is steps are:

```
MP := (p -> q) & p -> q;      is Taut;        // Modus Ponens,
```

The formula MP must be a propositional tautology.

```
F1 := X1 -- {a} <: X1;            is TDIn;         // ! TDIn := X -- Y <: X;
```

The formula F1 must be an instance of the formula TDIn, which was proved before. When checking that a term t is an instance of another term t1, a substitution s is computed, so we can say that t is a s-instance of t1 or t is a s-value of t1.

```
F0 := a in X1/\Y1
F2 := X1/\Y1 ~= {};  is Tinnemp(F0); //  Tinnemp := x in X -> X ~= {};
```

Here Tinnemp(F0) is an MP-justification. The proof checker calculates the resolvent of the formulas `Tinnemp := x in X -> X ~= {}` and `F0 := a in X1/\Y1`, receiving the formula `X1/\Y1 ~= {}`, and F2 must be an instance of that formula.

A by step, in the simplest case, has the form F; **by** E; G; where F and G are formulas and E is an equality A=B or an equivalence A==B. The proof checker compares F with G top down, left to right. For each subterm t of F which is not equal to the corresponding subterm t' in G, the subterm t must be an s-instance of the term A and the subterm t' must be the s-value of B.

A byeq step, in the simplest case, has the form F; **byeq** E; where F is an equality P=Q or an equivalence P==Q, E is also an equality A=B or an equivalence A==B. The byeq step is correct if the by step (P; **by** E; Q); is correct.

An assume step has the form F; **assume**; where F is a formula. To check the correctness of the assume step, the proof checker uses the term variable GOAL. The initial value of GOAL is the formula T to be proved. When checking the assume step, the following cases are possible:
Case 1: GOAL is an implication `P -> Q`. In this case, F must be equal to P. GOAL changes its value to Q.
Case 2; GOAL is s formula A[d,P]. In this case, F must be one of the d-axioms. After handling all such assume steps, GOAL changes its value to P.
Case 3: F is the negation of GOAL. GOAL changes its value to false.

In all cases, the formula F will be written into the list of local truths(the formulas proved in the other proof steps will be also written to the list) and all free variables in F will be written into the list of local constants.

At the end of the proof, when all the proof steps have been successfully checked, GOAL must be in the list of local truths.

5.1 Proving by cases

Unlike [7, 12], the V language has no special syntactic constructs for proving by cases. Suppose, we have proved a formula P or Q and wish to prove a formula T. For this it is sufficient to prove 2 lemmas: $P \to T$ and $Q \to T$.

In V, it can be done as follows:

```
L0 := P or Q;          <proof of L0>
T; by Case2(L0);       L1 & L2;
L1 := P -> T;          <proof of L1>
L2 := Q -> T;          <proof of L2>
```

Here Case2 is the tautology `p1 or p2 -> (q == (p1 -> q) & (p2 -> q))` defined in the root module.

Checking the by step above the proof checker calculates Case2(L0), receiving
`q == (P -> q) & (Q -> q)`. The by step becomes
`T; by q == (P -> q) \& (Q -> q); L1 & L2`.
Checking this by step, the proof checker finds out that T is an s-instance of q and s = (q: T). The s-value of `(P -> q) & (Q -> q)` is `(P -> T) & (Q -> T)`, which is equal to L1 & L2.

6 Equational proofs

An equational proof has the following form:
EqProof E;
¡Prelude: possible proof steps¿
¡Main part: a sequence of formulas, separated by by_elements¿
end EqProof E;

Here E is (in a simple case) a possibly signed equality. If E is A=B then -E denotes B=A. Below is an equational proof taken from the module group(elg denotes the carrier set of the group):

```
! Laaa := A[a:elg, a*a = a -> a = e];
 EqProof Laaa;
F0 := a in elg;        assume;
F00 := a*a = a;        assume;
F01 := a * inv(a) = e; is Axrinv; // ! Axrinv := A[x: elg, x * inv(x) = e];
F02 := a*e = a;        is Axre;   // ! Axre   := A[x:elg, x * e = x];
F1 := a;               by -F02;
F2 := a*e;             by -F01;
F3 := a*(a*inv(a));    by AxAssoc(a,a,inv(a));
F4 := (a*a)*inv(a);    by F00;
F5 := a * inv(a);      by F01;
F6 := e;
 end EqProof Laaa;
```

In this example, the prelude is steps F0, F00, F01, and F02, and the main part is steps F1-F6.

7 Conclusion

For testing the V language the following 3 modules (modules in V are similar to articles in Mizar) were written:

- root: elementary set theory: 5134 lines, 143 proofs, 47 equational proofs; some of the theorems were declared as axioms, they have to be proved later;
- group: the basics of group theory, including the isomorphism theorems: 5402 lines, 508 proofs, 326 equational proofs;
- altg: the basics of the permutation theory, necessary for defining the alternating group: about 1700 lines, 136 proofs, 60 equational proofs;

Also, a computer program (a proof checker) has been written(26300 lines in C++), which now is working in an experimental mode. The 3 modules have been successfully checked by the proof checker. The processing times for the root, group, and altg modules are 35sec,53sec, and 14sec respectively on the laptop Lenovo ThinkPad Yoga 11e 5th Gen.

The modules can be downloaded at [2].

References

[1] Mizar syntax. http://mizar.uwb.edu.pl/version/current/doc/syntax.txt, 2015.

[2] V folder. https://drive.google.com/drive/folders/1n63f1eym0cEXsy6i3d_qf7XniqoV8p5y?usp=sharing, 2023.

[3] V syntax. https://drive.google.com/file/d/1SNoJji98IcdnVdpMgRBW1Psu9DLGECo3/view?usp=sharing, 2023.

[4] Jacques Carette. Answering to a proponent of type theory. https://mathoverflow.net/questions/90820/set-theories-without-junk-theorems. But until I see type theory taught as a foundational course, and elementary textbooks eschew sets, I will not believe that set theory is not still the 'foundations'. Jacques Carette Mar 11, 2012 at 13:42.

[5] Jean Dieudonné and IG Macdonald. *A Panorama of Pure Mathematics, as seen by N. Bourbaki.* Academic Press, 1982.

[6] Abraham Adolf Fraenkel, Yehoshua Bar-Hillel, and Azriel Levy. *Foundations of set theory.* Elsevier, 1973.

[7] Adam Grabowski, Artur Kornilowicz, and Adam Naumowicz. Mizar in a nutshell. *Journal of Formalized Reasoning*, 3(2):153–245, 2010.

[8] John Harrison. Let's make set theory great again. In *Invited talk at the 3rd Conference on Artificial Intelligence and Theorem Proving (AITP 2018)*, 2018.

[9] Azriel Levy. *Basic set theory.* Courier Corporation, 2012.

[10] V. P. Makarov. MSL — a mathematical specification language. In Anil Nerode and Mikhail Taitslin, editors, *Logical Foundations of Computer Science — Tver '92*, pages 305–313, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg. doi:10.1007/BFb0023884.

[11] Bjarne Stroustrup. *The C++ programming language.* Pearson Education, 2013.

[12] Makarius Wenzel et al. The isabelle/isar reference manual, 2004.