

Lingua versus Dafny and Rocq

Andrzej Blikle

May 8th, 2025

1 General view

The idea of building programs in a way that guarantees their correctness is not new. To my knowledge, the inadequacy of the approach of first writing a program and only then trying to prove it correct was pointed out for the first time by Edsger Dijkstra in 1976 in his book *A Discipline of Programming* [8] where he writes:

Between the lines (of his book) the reader may have caught a few more general messages. The first message is that it does not suffice to design a mechanism of which we hope that it will meet its requirements, but we must design it is such a form that we can convince ourselves — and anyone else for that matter — that it will, indeed, meet its requirements. And, therefore, instead of first designing a program and then trying to prove its correctness, we develop correctness proof and program hand in hand (our emphasis). (In actual fact, the correctness proof is developed slightly ahead of the program: after having chosen the form of the correctness proof, make the program so that it satisfies the proof's requirements.)¹

Dijkstra formalized this idea using his *weakest preconditions*.

A little later (1977 – 1981), Andrzej Blikle published four papers ([1] - [4]) technically different from Dijkstra's approach but in a similar spirit and philosophy. And with the same goal.

In recent years, Dijkstra's ideas have been implemented by the authors of the Dafny Environment and their followers ([9] - [12]) under the name *correct-by-construction*. A comparable approach has been developed since 1984 in INRIA, recently under the name *The Rocq Prover* (see <https://rocq-prover.org>). Among other vehicles they are offering an ecosystem supporting correctness proofs of C programs.

In our approach (described in [5]) we are offering two universal mathematical methods:

- A. a method of designing programming languages in such a way that they have denotational semantics; as an example of an application of this method a (Python-like) language **Lingua** has been designed,
- B. a method of building provably correct programs in **Lingua**-like languages.

First major difference between **Lingua** and the two other approaches is that the latter restrict their concern only the **B** problem. Besides, they do not define any mathematical semantics of programming languages which they use. This semantics seems to be understood as implicit in their Hoare-like program-correctness proof rules, although they never claim that fact explicitly. In both these approaches, semantics is not an issue.

Therefore the only field where our approaches are may be compared is **B**.

Our approach to **B** splits into two steps. In **Step-1** we are building a program-correctness theory for languages with denotational semantics. On this ground we define the concept of *clean total correctness* of programs. Programs correct in this sense terminate their execution without abortion. We also define a number of program-construction rules that given correct programs return such programs as their results. The soundness of these rules is proved on the ground of our theory.

Given program constructors in a **Lingua**-like language, programmers can build their programs with a guarantee of correctness. Although they do not need to prove the correctness of programs ex-post, in the course of program development they have to prove the validity of some mathematically simple, but possibly textually large, formulas. Carrying such proofs by hand may be time-consuming and error-prone. Here our **Step-2**

¹ Another mention of this idea is in Dijkstra's [7].

begins. It consists in building a formalized **D-theory** (D for denotations) whose model includes our denotational model of **Lingua** plus the corresponding tools for correct-program construction. Our **D-theory**, which is actually under construction, must include two basic fundamentals:

1. a formalize language; here a “union” of **Lingua-V** and **MetaSoft** is offering a formal, although not (yet) formalized prototype,
2. a set of axioms that must be rich enough to formulate and prove the correctness of programs and the soundness of our program-construction rules.

Since the logic of **D-theory** is classical, we may try to adapt some existing theorem-provers to work for us.

D-theory must be built in such a way that our “world of denotations” constitutes its model. And, of course, this fact has to be proved! Once such a proof is completed we gain confidence that **D-theory** is consistent, which practically means that whatever we prove in it, is true (Kurt Gödel theorem).

Now, coming back to **Dafny** and **Rocq** their approaches corresponds to our **Step-2**. They are building formalized theories without formally building their models, hence — without proving their consistencies. They only assume that their axioms are adequate for the programming languages that they use (e.g., C in **Rocq**), but they can’t prove it, since they do not define a model for their theory.

2 Major technical differences between Lingua-V and Dafny

CONCEPT	LINGUA	DAFNY
denotations	The development of a many-sorted algebra of denotations is the “founding step” in designing Lingua. At this stage the language designer decides about all future mechanisms of the language.	No such concepts are explicitly defined.
syntax	Syntax is derived from an earlier constructed algebra of denotations in three steps: the derivation of abstract syntax, of concrete syntax and of colloquial syntax. First two of them are many-sorted algebras.	Syntax is defined by BNF equations, and is “final”, hence corresponds to our colloquial syntax. The definition of syntax is essentially the “founding step” of the language.
semantics	Abstract and concrete syntaxes are defined in a way that guarantee the existence of (unique) homomorphisms into the algebra of denotations. These homomorphisms are their denotational semantics. The semantics of the (final) colloquial syntax is a composition of a recovery function that turns colloquial syntax into concrete syntax, and the semantics of concrete syntax.	In the source report [11] only syntax is defined formally. Semantics is described informal and may be guessed to be implicit in Hoare-like proof rules. Although the authors never say that explicitly, they seem to regard these rules as evident, hence tacitly assuming that the implementation (semantics) of Dafny insures their soundness.
values	Typed data or objects.	No such concept is explicitly defined.
abstract errors	All domains of values include abstract errors and all constructors “react” to them. Besides, states may carry errors and, therefore, the denotations of program components also react to errors. The mechanism of handling errors is formalized in the semantics of the language.	No such concept is mentioned. At least, I have not identified any comments about errors.
types	Finitistic structured elements that unambiguously identify sets of values called the clans of types.	Informally understood as sets of values plus corresponding constructors. There are several categories of built-in types plus some mechanisms for creating user-defined types. The description of

		types mixes the syntax with an (informal) semantics of type- and value declarations.
value expressions	The denotations of value expressions are partial state-to-value functions, and may return errors as values. Expressions do not generate side-effects.	Value expressions are called right-hand-side expressions and constitute a very rich syntactic category. They are used both in programs and in their specifications. They may have side effects, e.g. if they include method calls.
reference expressions	Their denotations are total functions from states to references or errors.	They are called left-hand-side expressions and are defined in Sec. 9.14 of [11]. An example is $a[i]$, where a is an array. It might be interesting to analyze them in the context of de Bakker paradox.
boolean expressions (BE)	Their denotations are partial 3-valued predicates based on McCarthy's calculus.	No formal definition, but it is pointed out that the evaluation of BE may generate an error in a way that corresponds to McCarthy's calculus. From [10] (Sec. 2.0): Another name for a boolean expression is logical formula (or just formula) or predicate, which simply means an expression that for a given state evaluates to either false or true. One can also think of a predicate as characterizing a set of states.
type expressions	Their denotations are total functions from states to types or errors.	Not explicitly defined.
conditions	Syntactically constitute a superset of BE, but semantically are based on Kleene's rather than McCarthy's calculus.	One may guess that conditions are just expressions with boolean values, although not all such conditions may be used in programs. This issue is technically complicated and I did not have enough patience to study it.
ghost versus compiled items	Not introduced explicitly, but may be said to be implicit in conditions.	Ghost items are not seen by a compiler but by a theorem prover. Ghost items are used entirely in specifications.
constants	Do not introduced as a specific category.	Introduced as a particular case of variables called "constant variables". Declarable.
metaprograms	pre pr-con specified instruction post po-con Specified instructions may include assertions.	Called just "programs", but including the same elements although in a different arrangement: requires pr-con assures po-con specified instruction
program validation (correct by construction)	Correct metaprograms <u>are built</u> in a step-wise way by sound construction rules. Usually, an application of a rule requires proving an implication concerning program items (variables, types, methods, etc.). During the development of programs, preconditions, postconditions and specinstructions are constructed and modified. Lingua's correctness is ex ante	Developed specified programs <u>are proved correct</u> . Programs may be built from other programs (lemmas), but — as far as I understood — when "a single" program is developed, it is first developed as a whole, and then proved correct. Dafny 's correctness is ex post.

To a large extent a technical comparison of **Lingua** with **Rocq** is similar.

3 References

- [1] Blikle Andrzej, *Toward Mathematical Structured Programming*, Formal Description of Programming Concepts (Proc. IFIP Working Conf. St. Andrews, N.B Canada 1977, E.J Neuhold ed. pp. 183-2012, North Holland, Amsterdam 1978
- [2] Blikle Andrzej, *On Correct Program Development*, Proc. 4th Int. Conf. on Software Engineering, 1979 pp. 164-173
- [3] Blikle Andrzej, *On the Development of Correct Specified Programs*, IEEE Transactions on Software Engineering, SE-7 1981, pp. 519-527
- [4] Blikle Andrzej, *The Clean Termination of Iterative Programs*, Acta Informatica, 16, 1981, pp. 199-217.
- [5] Blikle Andrzej, Chrzastowski-Wachtel Piotr, Jablonowski Janusz and Tarlecki Andrzej, *A Denotational Engineering of Programming Languages*, a book in progress, 2024, <https://moz-nainaczej.com.pl/what-has-been-done/the-book>
- [6] Bordis Tabea, Runge Tobias, Schaefer Ina, *Correctness-by-Construction for Feature-Oriented Software Product Lines*, Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '20), November 16–17, 2020, Virtual, USA. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3425898.3426959>
- [7] Dijkstra Edsger, W., *A constructive approach to the problem of program correctness*, BIT 8 (1968)
- [8] Dijkstra Edsger, W., *A Discipline of Programming*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 1976
- [9] Leino K. Rustan M., *This is Boogie 2*, Manuscript KRML 178, working draft 24 June 2008, <https://www.microsoft.com/en-us/research/project/boogie-an-intermediate-verification-language/publications/>
- [10] Leino K. Rustan M., *Program Proofs*, MIT Press 2023,
- [11] Leino K. Rustan M., Cok David R., and the Dafny contributors, *Dafny Reference Manual*, August 29, 2024, <http://dafny.org/dafny/DafnyRef/DafnyRef> ,
- [12] Getting Started with Dafny: A Guide, <https://dafny.org/latest/OnlineTutorial/guide>