A note on climate science and verified programming

The TiPES-WP6 team with Cezar Ionescu and Patrik Jansson

2020-10-26

Parts of this note have been taken almost verbatim from (Ionescu et al., 2018) and (Botta et al., 2017).

Verified programming: what the heck is that?

In his presentation "*Understand and communicate the impacts of Tipping Point uncertainties on accountable policies*" of the work done in WP6 during the first 6 months of the H2020 EU TiPES project, M. Crucifix pointed out that our approach is based on three pillars: *climate science*, *decision theory* and ***verified programming***.

Most climate scientists will probably agree that understanding and communicating the impacts of tipping point uncertainties on climate policies requires contributions from climate science. And many will agree that understanding the impacts of uncertainties on climate policies (no matter whether these are about the presence or the magnitude of abrupt transitions or about the value of model parameters at which structural changes in relevant features of the model take place) requires some understanding of the decision processes for which such policies are envisaged and, hence, of decision theory[1].

But what has *verified programming* to do with all this? What has verified programming to do with climate science, and, most importantly, what *is* verified programming and ***who needs it***?

Who needs verified programming?

In a nutshell, verified programming is a methodology for writing programs that can be checked to be correct by a *type checker*.

A type checker is itself a program (perhaps written in a language that is not that of the programs it checks) and a program is correct if it fulfills a *specification*. For example, a specification for a program R that is meant to compute square roots of positive real numbers might look like

$$\forall x \in \mathbb{R}, \quad 0 \leq x \Rightarrow R(x) * R(x) = x \tag{1}$$

[1]Decision theory is a rather broad notion. Accountable climate policy advice necessarily requires contributions from different disciplines including, among others, control theory, expert elicitation and game theory. In TiPES WP6 we will follow (Webster, 2000), (Webster, 2008) and focus of control theory and sequential decision problems under non-deterministic and stochastic uncertainty, see (Botta et al., 2020).

Some machinery is needed in order to turn (1) into a *formal* statement that a type checker can actually process. Also, the notion that a square root program R is correct if it can be shown to fulfill (1) by a type checker is not without problems: what if the type checker itself is incorrect? And what does this mean? These are interesting and relevant question but we do not need to be concerned with them here.

For the purpose of this discussion, we only need to realize that (1) assigns a *meaning* to R. It is a precise and yet concise, description of what R is required to do. It is certainly more concise and more understandable than R itself can possibly be, especially when R is written in an imperative programming language[2].

If the type checker verifies R to fulfill (1), we can be sure that, as long as x is non-negative and R terminates on x, $R(x)$ is a square root of x.

This is quite something but if we are paying a lot of money for somebody to implement R, we might want to get a little bit more for our money. First, we might want R to always terminate, or, at least, to terminate for values of x in a suitable range. Second, we might want to make sure that R always delivers a positive root. Nothing in our specification so far prevents R to deliver -1 for 1 and 2 for 4!

If we demand more from R, we will typically have to pay more as the implementor will face a more difficult task. The *strength* of a specification is a crucial trait of the contract between the client of a program, say P, and the developer of P. The latter is always free to deliver a program that meets stronger requirements S' than those agreed on in the specification S

$$S'(P) \Rightarrow S(P) \tag{2}$$

Symmetrically, the client is free to accept programs that deliver less than agreed on:

$$S''(P) \Leftarrow S(P) \tag{3}$$

But delivering less than what has been agreed on in the specification (or requiring more) is a potential source of conflicts and perhaps court disputes.

Another source of potential misunderstanding are *impossible* specifications. Often, the client simply demands too much. In the case of the square root function, for instance, demanding $R(x) * R(x)$ to be exactly equal to x is too much if R has to terminate and can thus only compute *approximations* of irrational roots. Impossible specifications are another potential source of misunderstandings

[2]With a slight oversimplification, there are two distinct families of programming languages: imperative and functional. Examples of imperative languages are FORTRAN, C, C++, Java. Examples of functional languages are Haskell, Agda, Coq (The Coq Development Team, 2020), Idris. Imperative programming is a method of specifying what a computing machine shall do in terms of *instructions* and *execution procedures*. In functional programming, one specifies what a computing machine shall do in terms of *functions* and their *application* and *composition*, with an emphasis on inductive definitions and algebraic structure. In functional languages, the expression `a = b` has the same meaning as in mathematics. In imperative programming this is not the case and instructions like `a = 1; a = 3` are valid in spite of the fact that 1 is not equal to 3.

and should be avoided. Program that fulfill impossible specifications are often called *miracles*, see (Morgan, 1990).

The discussion above should have made clear that verified programming is also (perhaps mainly) about being precise about the computations that a program shall perform. As a first step, this is done by putting forward mathematical specifications. In turn, these assign precise meanings to programs.

Indeed, one of the first papers on verified programming was Floyd's 1967 "Assigning Meaning to Programs" paper (Floyd, 1967). It is one of the seminal papers in computer science that still inform modern program verification.

Today, all programs that somehow matter – program that control medical equipment, financial transactions, weapons, access to critical data, power plants, air control systems, etc. – rely on some form of formal verification.

But who does actually need verified programming in science? What does it have to do with scientific computing? Do numerical analysts need to verify their programs? What about climate scientists?

It is probably fair to say that, as long as scientists are operating in a purely academic environment, they do not need to care about program verification: no university teacher is likely to loose her job because of a programming error.

Still, there are prominent examples of scientific claims that have been founded on programming errors (no citations here!). And in absence of clear, unambiguous specifications, even careful physical experiments and testing can easily lead to severe, regrettable consequences (Wikipedia, 2020b), (Lions and others, 1996), (Wikipedia, 2020a).

So do we all need verified programming? The answer is yes, but at different dosages.

As long as we are working on problems that are very well understood, program verification does probably not need to be our major concern. Precise specifications could still save us a lot of tedious work and time but, as long as we are implementing a new discretization for the Navier-Stokes equation, perhaps one that accounts for some insights from asymptotic analysis, we can rely on a whole body of knowledge and theoretical understanding of the problem at stake. In these cases, we indeed rely on very precise, albeit in most cases implicit, specifications. The same holds when our program is meant to deal with stiff ordinary differential equations or when we are implementing multi-grid methods for solving elliptic partial differential equations.

Things start to become different when we move from numerical methods for, e.g., the Euler equations to numerical methods for weather prediction or, even worse, global circulation models (GCM).

Here, the air starts to become thinner and our safety network less reliable. We can try to compensate for the lack of general results with careful testing. But tests can only show the presence of errors, not their absence: in front of unexpected results and without verified programs, we cannot know whether we are confronted with model deficiencies or with errors in the implementations of the models. In this situation, model validation becomes impossible.

Things get worse when we move from GCMs to intermediate complexity models and at the latest when we get to integrated assessment models or, even worse, non-deterministic or stochastic agent-based models or models for decision making, program verification becomes mandatory.

But is it not enough to test our models? Cannot we simply test our square root program R on a sample of inputs that is representative of the values for which we want to compute square roots? We answer this question in the next section.

Testing vs. proving

There are basically two methodologies for assessing that a program behaves according to a specification: testing and proving (Ionescu and Jansson, 2013).

In testing, a program is required to pass a finite number of tests in order to be positively verified. In proving, the program has to be shown to fulfill a formal specification.

In engineering and industrial applications, testing is well supported (Claessen and Hughes, 2000). It has a strong historical record of successes interspersed with a few dramatic failures (Wikipedia, 2020b), (Lions and others, 1996), (Wikipedia, 2020a).

Testing and proving are complementary methods. Testing can show the presence of errors, proving can show their absence. When can we test, when do we have to prove?

Discussing these questions goes beyond the scope of this note, but notice another crucial difference between testing and proving: testing a program P requires running P. By contrast, proving that P fulfills a specification does not require running P.

Thus, when available, proving is the method of choice when running a program takes a lot of time or is very expensive or dangerous. The other way round, when running a program is cheap and safe, testing is a viable choice.

It is also worth noticing that there are cases in which testing and proving are equivalent and thus, testing can indeed ascertain the absence of errors. Can you see when this is the case?

No matter whether we are trying to assess the correctness of a program by tests or formal proofs, we always need a specification. For our square root program R, for instance, we need something like (1). In absence of specifications, we do not know how to test R and also we do not know how to prove that R is correct.

This note is about verified programming and we are not insisting on testing here. However, Michel mentioned design-by-contract as a testing methodology in his talk and thus we are going to say a few words about testing before moving to verified programming.

In a nutshell, design-by-contract is a method for encoding program specifications in run-time tests. The methodology has been popularized in the late eighties, mainly through the work of Bertrand Meyer (Meyer, 1986), (Meyer, 1997), (Meyer, 1992), see also (Meyer, 2020). It allows programmers to specify and document programming tasks and to detect failures to comply with the specification at run time.

If you write programs in, among others, D, Eiffel, Fortress, Scala or Clojure, you can rely on native support for design-by-contract patterns. If you code in C, C++, Java or Python, check language-specific libraries for contract, e.g., (Caminiti, 2020) for C++. For other programming languages, see examples of design-by-contract patterns implemented via assertions on (RosettaCode, 2020).

Design-by-contract cannot guarantee that programs are correct. But it can signal the presence of errors and this has lead to significantly faster and more understandable software development.

A further step toward building programs from verified components has been achieved through Quickcheck (Claessen and Hughes, 2000). Quickcheck is a *combinator* library. It has been designed to assist program testing and is available in most programming languages, see (Wikipedia, 2020c).

Proving: verified programming

How do we actually verify that a program fulfills a specification? This is typically done within a *specification language* and using computer-assisted formal methods.

Until about two decades ago, programs were written and specified in different languages: programming languages and specification languages like Z (Bowen, 1996), VDM (Jones, 1990), B (Abrial, 1996) or Maude (Clavel et al., 2007).

Today, we can rely on a unified framework for program specification and program implementation, one that is mature (several decades old), with solid implementations (NuPRL (Allen et al., 2006), Coq (The Coq Development Team, 2020), Agda (Norell, 2007), Idris (Brady, 2017), Lean (de Moura et al., 2015)), and impeccable mathematical credentials: *Dependent Type Theory*.

In short, Dependent Type Theory is a pure functional programming language with a static type system. It is similar to Haskell (Kees Doets and Eijck, 2004), (Bird, 2014), and stands in roughly the same relation to it as predicate logic to propositional logic (Moschovakis, 2018). Dependent Type Theory was developed by the Swedish mathematician and philosopher Per Martin-Löf (Martin-Löf, 1984), who intended it to have the same foundational role for intuitionistic mathematics that set theory expressed in predicate logic had for classical mathematics.

(In the following we use "Type Theory" for brevity, but understand by it "Dependent Type Theory".)

This is not the place for a presentation of Type Theory, for a particularly accessible one, see (Altenkirch, 2017). What we want to do here is to provide an intuition for why Type Theory provides an environment for both program specification and program implementation and for how this environment is used in program verification.

We start by recalling that set theory derives its foundational role in classical mathematics from its ability to represent properties in several different (equivalent) ways, within a first-order language. For example, given a property P over a set A, expressed as a formula in the first-order language of sets, we can view it as a

- set $P = \{a \mid P\ a\}$, $a \in P$ iff a has the property P
- Boolean-valued function: $P : A \to Bool$, $P\ a = True$ iff a has the property P
- set-valued function: $P : A \to \{\{\}, \{*\}\}$, $P\ a = \{*\}$ iff a has the property P

All these allow us to talk about the property **within** the theory: it becomes an element of the universe of discourse. If we take types in programming languages to be the analogues of sets in set theory, we can see that the available means for their construction are more restricted. In common with other functional programming languages, Type Theory allows the construction of inductive types. For example

$$\frac{}{Z : Nat} \qquad \frac{n : Nat}{S\ n : Nat}$$

and

```
data Nat : Type where
  Z : Nat
  S : Nat -> Nat
```

are two equivalent ways of expressing the familiar rules for the inductive construction of the natural numbers: zero (Z, `Z`) is a natural number; if `n` is a natural number then the successor of `n` ($S\ n$, `S n`) is a natural number, etc.

The first definition is written using *inference rules* - this is how the rules of logical proof systems are commonly presented (Plato, 2018); the other one is written in the style of Haskell, Agda, or Idris. In most programming languages, we can represent properties as Boolean-valued predicates. For example in Haskell:

```
isEven : Nat       ->  Bool
isEven Z           =   True
isEven (S Z)       =   False
isEven (S (S m))   =   isEven m
```

In most cases, however, we cannot represent the associated set as a datatype or as a type-valued function. Therefore, if a function requires its argument to be even, then the best we can do is to guard the call of the function with a run-time test. This leads to expressing requirements or specifications as tests, as in *test-driven development* methods or, as discussed above, design-by-contract.

In contrast, in Type Theory, we have the *additional* possibility of representing a property by a type-valued function (a type *family*), which corresponds to the set-valued version in set theory. For example

```
data Even : Nat -> Type where
  MkEven : (k : Nat) -> Even (2 * k)
```

is a way of expressing the type-valued function version of `isEven`. For every natural number `n`, `Even n` is a type. If `n` is not even, then the type will be empty. Otherwise, the type will have one element, namely `MkEven (n / 2)`.

If a function requires its argument to be even, we can now formulate this requirement at the level of its type, for instance

```
f : (n : Nat) -> Even n -> X
```

In order to call `f` with an argument `n`, we have to supply another argument of type `Even n`. We can only do that if `n` is `Even`, since otherwise `Even n` would be empty. This additional argument must be reducible to the form `MkEven k`, where `k = n / 2`, and this can be checked at *compile time* (or, rather, at “type-checking time”). This ensures that `f` will never give rise to a run-time error, a much stronger guarantee than we can enforce by means of tests.

The ability to define inductive data types and type families lends Type Theory a surprisingly strong expressive power, equal to that of classical higher-order logic. In particular, we can formulate all the notions in current mathematics. Note, however, that the only formulas we can *prove* are those of constructive mathematics: the logic of Type Theory is *intuitionistic* which means that we cannot rely on classical axioms such as *excluded middle* ($A \vee \neg A$) or *double negation elimination* ($\neg\neg A \Rightarrow A$).

When it comes to specifications of programs, this is not a bug, but rather a feature. The requirements on a program can be expressed at the level of types, for example

```
f : (x : X) -> Pre x -> Sigma (y : Y) (Post x y)
```

is the type of a function that takes as input elements of a type `X` having the property `Pre`, and delivers elements of a type `Y` which are in the relation `Post` with the input. The `Sigma` in the return type of `f` represents a dependent pair: this consists of a value `y : Y` and of a value of type `Post x y`, depending both on `x` and on `y`.

After a long detour on Type Theory, we can finally answer the question stated at the beginning of this paragraph: “How do we actually verify that a program fulfills a specification?” This is done by implementing another program. For example, a verified implementation of our square-root program R would consist of the function itself

```
R : (x : Double) -> 0 <= x -> Double
```

and of another function

```
sqrtR : (x : Double) -> 0 <= x -> R(x) * R(x) = x
```

Notice that the type of `sqrtR` (`R` is a square-root function) encodes the logical proposition (1) with $\mathbb{R}$ replaced by `Double`. If an implementation of `sqrtR` can be type-checked to be total (to terminate for every input `x : Double` and for every evidence that `0 <= x`) it is in every respect a proof that `R` is a square-root function and the equivalence between implementing `sqrtR` and proving the corresponding logical proposition has been established rigorously (Wadler, 2015).

As already discussed, implementing `sqrtR` in this form is impossible and the equality `R(x) * R(x) = x` has to be weakened to equality up to a suitable tolerance.

The example shows that even expressing specifications for (let apart verifying) functions that perform floating point operations is not a trivial task. Indeed, as of yet we cannot rely on an easy-to-use form of validated numerics (Tucker, 2011) - this is still an area of on-going research (see e.g. (Boldo and Melquiond, 2017) to get an idea of the current state of the art).

Perhaps not surprisingly, the approach to program specification and verification based on Type Theory works extremely well for all what is not directly based on floating point computations.

It has been successfully applied in e.g., producing a verified C compiler, CompCert (Leroy, 2009); developing database access libraries which statically guarantee that queries are consistent with the schema of the underlying database (Oury and Swierstra, 2008); implementing secure distributed programming (Swamy et al., 2011); implementing resource-safe programs (Morgenstern and Licata, 2010), (Brady and Hammond, 2012); and many others.

In TiPES WP6, we apply verified methods to provide decision makers with policies that are machine-checked to be optimal. We discuss why in the next section.

Formal methods as a surrogate for empirical evidences

We have argued that as we move away from well understood problems to climate models, integrated assessment models, agent-based models and, more generally, methods for climate policy advice, program verification becomes mandatory.

But, in science and engineering, we have plenty of examples of programs that are not verified and yet are successfully applied to inform policy advice. For instance, deep neural networks are routinely applied for decision making in routing problems, gaming, and medical screening.

Cannot we provide accountable policy advice without having to care about program verification? This is a very legitimate question that needs to be addressed with some care.

Ideally, we would like climate policy advice to be based on empirical evidences. This is not only because our understanding of the climate system is far from being perfect.

Most importantly, we know that optimal decisions in matters of climate policy necessarily depend on uncertainties that we can hardly estimate: how likely is it that decisions about emission reductions taken, say, by the EU, are actually going to be implemented over the next decade? What about decisions taken by China or by the USA?

We all know too well that facts do not always follow decisions and that, more than often, taken decisions are not implemented or are implemented with delays. Legislations have large inertia and governments do not always manage to comply with their own decisions.

We know that these uncertainties, but also uncertainties on the consequences of trespassing critical climate thresholds or on the collateral effects of geo-engineering approaches towards mitigating the impacts of climate change, do have an impact on optimal emission policies.

In other words, we know (for sure because we have obtained these assessments by applying verified methods) that decisions on emissions that are optimal when we assume these uncertainties to be zero become sub-optimal when we account for these uncertainties properly (Botta et al., 2018).

Thus, we would like climate policy advice to be based on empirical evidences. But gathering empirical evidences in matters of climate policy is nearly impossible!

Not even global players like China or the USA can afford to perform large scale, carefully designed social experiments in order to assess the effectiveness of, say, carbon taxation schemes. We cannot test two or three carbon taxation schemes on a couple of EU countries to find out which one would be best to adopt on a larger scale.

In other words, we have to advise decision makers without being able to rely on empirical evidences. This is unfortunate but hardly avoidable. In this situation, the only guarantees that we can provide to decision makers come from verified methods. This is not a peculiarity of policy advice in matters of climate: advising governments on how to auction radio frequencies or internet domains (Caminati et al., 2015) faces similar problems, and similar problems are also encountered in policy advice on matters on epidemics, financial markets and taxations and security. Not surprisingly, these are application domains in which formal methods are routinely applied to provide some form of accountability in absence of empirical evidences. They are not ideal but certainly better than nothing.

The bottom line

The main purpose of this note was to discuss why verified programming and formal methods are at the core of the WP6 approach towards "Understanding and communicating the impacts of Tipping Point uncertainties on accountable policies".

We have pointed out the roles of program specification, testing and verification in program development and argued that Type Theory is a useful approach for verified programming.

But most of what we have discussed remains true if we replace the word *program* with the word *problem*[3]!

Thus, this note also points to the fact that Type Theory can be applied as a methodology for understanding and formulating problems and as a vehicle for communication between computer scientists and scientists from other disciplines.

It hopefully also points to the fact that the main role of computer science is not confined to the execution of arithmetical operations or sending data over networks, but is rather to be found in the formulation of concepts, identification and resolution of ambiguities, and, above all, in making our ideas clear.

References

Abrial, J.-R.: The B-Book: Assigning Programs to Meanings, Cambridge University Press., 1996.

Allen, S., Bickford, M., Constable, R., Eaton, R., Kreitz, C., Lorigo, L. and Moran, E.: Innovations in computational type theory using NuPRL, Journal of Applied Logic, 4(4), 428–469, 2006.

Altenkirch, T.: Naive Type Theory, [online] Available from: http://www.cs.nott.ac.uk/~psztxa/mgs-17/notes-mgs17.pdf, 2017.

Bird, R.: Thinking Functionally with Haskell, Cambridge University Press., 2014.

Boldo, S. and Melquiond, G.: Computer Arithmetic and Formal Proofs, ISTE Press - Elsevier., 2017.

Botta, N., Jansson, P. and Ionescu, C.: Contributions to a computational theory of policy advice and avoidability, J. Funct. Program., (27, e23), 2017.

Botta, N., Jansson, P. and Ionescu, C.: The impact of uncertainty on optimal emission policies, Earth System Dynamics, 9(2), 525–542, doi:10.5194/esd-9-525-2018, 2018.

Botta, N., Brede, N., Crucifix, M., Ionescu, C., Jansson, P. and Martínez, M.: Climate science and climate policy, 2020.

Bowen, J.: Formal Specification and Documentation using Z: A Case Study Approach, International Thomson Computer Press., 1996.

Brady, E.: Type-Driven Development in Idris, Manning Publications Co., 2017.

Brady, E. and Hammond, K.: Resource-safe systems programming with embedded domain specific languages, in International Symposium on Practical Aspects of Declarative Languages, Springer, Berlin,

[3]Indeed, as early as 1932, Kolmogorov showed in a short paper (Kolmogoroff, 1932) (written in German!) that the rules of intuitionistic logic – the logic of Type Theory – can be interpreted as rules of "problem computation" ("Aufgabenrechnung" in the original paper).

Heidelberg., 2012.

Caminati, M., Kerber, M., Lange-Bever, C. and Rowat, C.: Sound auction specification and implementation, in EC '15 Proceedings of the 16th ACM Conference on Economics and Computation, edited by T. Roughgarden, M. Feldman, and M. Schwarz, pp. 547–564, Association for Computing Machinery., 2015.

Caminiti, L.: Boost C++ libraries Version 1.73.0, [online] Available from: https://www.boost.org/doc/libs/develop/libs/contract/doc/html/index.html, 2020.

Claessen, K. and Hughes, J.: QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs, in Proceedings of the 5th International Conference on Functional Programming, pp. 268–279., 2000.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J. and Talcott, C.: All About Maude - A High-Performance Logical Framework, Springer-Verlag., 2007.

de Moura, L., Kong, S., Avigad, J., Doorn, F. van and Raumer, J. von: The Lean Theorem Prover (System Description), in Automated Deduction - CADE-25, pp. 378–388, Springer International Publishing, Cham., 2015.

Floyd, R. W.: Assigning Meaning to Programs, Mathematical Aspects of Computer Science, 19–32, 1967.

Ionescu, C. and Jansson, P.: Testing versus proving in climate impact research, in 18th International Workshop on Types for Proofs and Programs (TYPES 2011), vol. 19, pp. 41–54, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany. [online] Available from: http://drops.dagstuhl.de/opus/volltexte/2013/3899, 2013.

Ionescu, C., Jansson, P. and Botta, N.: Type Theory as a Framework for Modelling and Programming, in Proceedings of the 8th International Symposium, ISoLA 2018, part I, vol. 11244, pp. 119–133, Springer., 2018.

Jones, C. B.: Systematic Software Development Using VDM, Second., Prentice-Hall., 1990.

Kees Doets and Eijck, J. van: The Haskell Road to Logic, Math and Programming, College Publications., 2004.

Kolmogoroff, A. N.: Zur Deutung der intuitionistischen Logik, Mathematische Zeitschrift [online] Available from: https://doi.org/10.1007/BF01186549, 1932.

Leroy, X.: Formal verification of a realistic compiler, Communications of the ACM, 52(7), 107–115, doi:10.1145/1538788.1538814, 2009.

Lions, J.-L. and others: Ariane 5 Flight 501 Failure, Report by the Inquiry Board. [online] Available from: https://esamultimedia.esa.int/docs/esa-x-1819eng.pdf, 1996.

Martin-Löf, P.: Intuitionistic Type Theory, Bibliopolis, Napoli., 1984.

Meyer, B.: Design by Contract, Technical Report TR-EI-12/CO, Interactive Software Engineering Inc., 1986.

Meyer, B.: Applying 'design by contract', Computer, 25(10), 40–51, doi:10.1109/2.161279, 1992.

Meyer, B.: Object-Oriented Software Construction (2nd Ed.), Prentice-Hall, Inc., USA., 1997.

Meyer, B.: Getting a program right, in nine episodes, [online] Available from: https://bertrandmeyer.com/category/design-by-contract/, 2020.

Morgan, C.: Programming from specifications, Second., Prentice-Hall., 1990.

Morgenstern, J. and Licata, D.: Security-Typed Programming within Dependently Typed Programming, in Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP, vol. 45, pp. 169–180., 2010.

Moschovakis, J.: Intuitionistic Logic, in The Stanford Encyclopedia of Philosophy, edited by E. N. Zalta, https://plato.stanford.edu/archives/win2018/entries/logic-intuitionistic/; Metaphysics Research Lab, Stanford University., 2018.

Norell, U.: Towards a practical programming language based on dependent type theory, PhD thesis, Chalmers University of Technology; Citeseer. [online] Available from: https://research.chalmers.se/en/publication/46311, 2007.

Oury, N. and Swierstra, W.: The power of Pi, in Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming., 2008.

Plato, J. von: The Development of Proof Theory, in The Stanford Encyclopedia of Philosophy, edited by E. N. Zalta, https://plato.stanford.edu/archives/win2018/entries/proof-theory-development/; Metaphysics Research Lab, Stanford University., 2018.

RosettaCode: Assertions in design by contract, [online] Available from: https://rosettacode.org/wiki/Assertions_in_design_by_contract, 2020.

Swamy, N., Chen, J., Fournet, C., Strub, P.-Y., Bhargavan, K. and Yang, J.: Secure distributed programming with value-dependent types, in Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming., 2011.

The Coq Development Team: The Coq Proof Assistant, version 8.11.0, doi:10.5281/ZENODO.1003420, 2020.

Tucker, W.: Validated Numerics: A Short Introduction to Rigorous Computations, Princeton University Press. [online] Available from: http://www.jstor.org/stable/j.ctvcm4g18, 2011.

Wadler, P.: Propositions as Types, Commun. ACM, 58(12), 75–84, doi:10.1145/2699407, 2015.

Webster, M. D.: The Curious Role of "Learning" in Climate Policy: Should We Wait for More Data?, MIT Joint Program on the Science; Policy of Global Change, Report No. 67., 2000.

Webster, M. D.: Incorporating Path Dependency into Decision-Analytic Methods: An Application to Global Climate-Change Policy, Decision Analysis, 5(2), 60–75, 2008.

Wikipedia: Boeing 737 MAX groundings, [online] Available from: https://en.wikipedia.org/wiki/Boeing_737_MAX_groundings, 2020a.

Wikipedia: Cold fusion, [online] Available from: https://en.wikipedia.org/wiki/Cold_fusion, 2020b.

Wikipedia: QuickCheck - from Wikipedia, the free encyclopedia, [online] Available from: https://en.wikipedia.org/wiki/QuickCheck, 2020c.