此為已歸檔的貼文。您不能於此貼文評分或留言。

Leslie Lamport on real-world use of TLA+

19

(youtube.com)

pron98 於 2 年前 發表

16 留言 分享 儲存 隱藏 贈送金幣 檢舉 crosspost

所有 16 則留言

排序依據: 最佳

[–] **the_evergrowing_fool** 1 指標 2 年前*

> Everything is a object, everything is a function, everything is that...

No, everything is a SM.

Edit: removing a F :)

永久連結 embed 儲存 贈送金幣

[–] **phalp** 2 指標 2 年前

Verified by His Noodly Appendage.

永久連結 embed 儲存 上層留言 贈送金幣

[–] **pron98** [S] 1 指標 2 年前*

Not *F*SMs but state machines, and all computations really are mathematically (abstract) state machines. FSM? State machine. Turing machine? State machines. Lambda calculus? State machine. Even combinator calculus describes a state machine. If you have a computation you have a state machine. Even if everything is an object or everything is a function, computations are still all state machines; everything else is just notation and abstraction. It is stating a

programming

訂閱 1,263,153 讀者

4,387 人在這裡

在這個看板上顯示我的使用者標籤。標籤顯示為: hengxin

/r/programming is a reddit for discussion and news about computer programming

Guidelines

- Please keep submissions on topic and of high quality.
- Just because it has a computer in it doesn't make it programming. If there is no code in your link, it probably doesn't belong here.
- Direct links to app demos (unrelated to programming) will be removed.
- No surveys.
- Please follow proper reddiquette.

Info

- Do you have a question? Check out /r/learnprogramming, /r/cscareerquestions, or Stack Overflow.
- Do you have something funny to share with fellow programmers? Please take it to /r/ProgrammerHumor/.
- For posting job listings, please visit /r/forhire or /r/jobbit.
- Check out our faq. It could use some updating.
- Are you interested in promoting your own content? STOP! Read this first.

Related reddits

- /r/technology
- /r/ProgrammerTIL
- /r/learnprogramming
- /r/askprogramming
- /r/coding
- /r/compsci
- /r/dailyprogrammer
- /r/netsec
- /r/webdev
- /r/web_design
- /r/gamedev
- /r/cscareerquestions
- /r/reverseengineering
- /r/startups
- /r/techsupport

Specific languages

mathematical fact rather than suggesting a specific design.

永久連結 embed 儲存 上層留言 贈送金幣

[–] the_evergrowing_fool 1 指標 2 年前

Ok, lets remove a F to make you agree with me, shall we?

永久連結 embed 儲存 上層留言 贈送金幣

[–] pron98 [S] 1 指標 2 年前

Hmm, nice try, but we're not there yet. You'll have to get rid of the "No", too, because everything (as in every computation) is a state machine even if everything is an object or a function.

永久連結 embed 儲存 上層留言 贈送金幣

[–] the_evergrowing_fool 1 指標 2 年前*

What if I put everything in a union type?

Edit: Oh I see, I think you miss a comma while reading it. So we are pretty much in agreement.

永久連結 embed 儲存 上層留言 贈送金幣

[–] kamatsu 1 指標 2 年前*

State machines are just one of many, many equivalent ways to describe computations. You could just as easily say that lambda calculus or recursive functions are more fundamental because the usual semantics ignore operational differences that don't affect the denotation.

永久連結 embed 儲存 上層留言 贈送金幣

[–] pron98 [S] 1 指標 2 年前*

Let me put it this way: Google's latest achievement beating a Go champion shouldn't be celebrated as we've had algorithms that *denotationally* play Go for decades :)

There's no point in discussing which is more fundamental, but any precise computational model (i.e. one that does not approximately identify computations with the functions they compute) can be expressed as a state machine. Lambda expressions combined with an evaluation strategy do describe a state machine. OTOH, a lambda expression alone does not uniquely define a computation (i.e. different evaluation strategies create computations that can be differentiated by an external observer; you can define an equivalence relation on them, but it is not precise in the physical sense), so I am not certain that every computation can be precisely and uniquely defined as a lambda expression (unless you "embed" state machines within LC, and declare a monad to denote a "step").

BTW, I don't think this is surprising or controversial. Regardless of which is more "fundamental", LC alone (without additional information) is less "powerful" than TMs as a *direct* representation (it lacks introspection required for things like parallel or) relying on encodings and simulation and/or additional definitions to precisely describe computations. And abstract state machines are more "powerful" than TMs, by *directly* supporting non-determinism and by easily specifying any cost model (i.e. what a step does is both arbitrarily complex and directly stated rather than declared externally).

I do not know, however, about definitional equality and whether it also uniquely defines a computation. Perhaps you can enlighten me on that. What is the relationship between definitional equality and bisimilarity?

EDIT:

I think that the fact that the same lambda expression can describe multiple executions, some of which would be considered correct by a programmer and some considered buggy and require fixing, is a sign that a more precise model of computation can have practical consequences. This is philosophical, but AFAIK, denotation has never been viewed as a precise equivalence since the dawn of CS. Turing spoke of functions that machines compute, but it's been clear right from the beginning that those functions are a property of the computations, but most certainly do not define it. This has certainly been the case at least since the '60s with Rabin's (Church's student) computational complexity and possibly even earlier, in Gödel's early ("remarkably modern", according to Sipser) discussion of the issue.

But the question is "precision with respect to what"? And I think that the answer is some physical process. I think it is also clear that at least Turing and Gödel thought that the physicality of a computation is essential (Gödel spoke of computations in terms of effort and time, which is why he was probably the first, or certainly among the first, to think of such things as "brute force" and P vs. NP). Obviously, Lamport is in that group, too, considering that special relativity is what inspired him to come up with notions of time in distributed systems (which later got him the Turing award). I think this philosophy is subtly but distinctly felt in the ideas behind TLA, and in particular the notion of invariance under stuttering, which basically means that unless you explicitly specify a time duration for each step, an observer cannot differentiate between two steps with the same state (a step is "that which can be observed"), and therefore a specification of a machine can be stretched or contracted in time.

[–] **notfancy** 1 指標 2 年前

> FSM? State machine. Turing machine? State machines. Lambda calculus? State machine. Even combinator calculus describes a state machine.

Sorry, this makes no mathematical sense to me at all. First of all, what exactly is a "state machine"? A finite semigroup with a group action? A Chu space? Some kind of Galois connection à la Backhouse?

Second, each of these notions of computation is defined by a different mathematical abstraction, so much so that in some cases there is no obvious nor useful way to couch the abstraction in terms of states and transitions. For instance, in the case of the calculi, what you want is to quotient (identify) terms related by reduction, so in a fundamental sense reduction is a static, not dynamic, view of the evaluation process.

Third, anyone can play the reductivist game: you could say that what these *really* are is computable functions and it would be as illuminating as your bare-faced assertion.

If what you are after is trying to emphasize the dynamic view of computation-as-a-process (machine) instead of the static view of computation-as-a-relation (function) between inputs and outputs, by all means be explicit. But please don't lose sight of the fact that if Church-Turing means anything, it means that those two views are equivalent.

[–] **pron98** [S] 1 指標 2 年前*

> Sorry, this makes no mathematical sense to me at all. First of all, what exactly is a "state machine"? A finite semigroup with a group action? A Chu space? Some kind of Galois connection à la Backhouse?

I don't understand. Are automata not well-known, well-studied, mathematical objects? In any event, a state machine is a (possibly infinite) set of states along with a transition relation. How

each state is defined depends on the model, which is why I've said they're all *abstract* state machine. A state could be labeled with a name, as in FSMs, a set of logical propositions as in Kripke structures, an expression as in lambda calculations etc.. The most general description is that of ASMs which simply say that a state is identified by some *structure*.

In any event, you're free to specify your algorithm as a lambda expression in TLA+, but you'll also need to specify your evaluation strategy to get a fully specified state machine.

> For instance, in the case of the calculi, what you want is to quotient (identify) terms related by reduction, so in a fundamental sense reduction is a static, not dynamic, view of the evaluation process.

That may be so, but it is the state machine doing the evaluation that's determining the computation (which is then a particular *behavior* of the machine). The calculus alone is not a unique description of a computation, but a description of a set of possible computations. For example, in the lambda calculus, call-by-name and call-by-value are two different state machines performing *different* computations. How do we know they're different? Because an outside observer can observe the difference in computational complexity.

Of course, you may define any equivalence relation that you like to achieve certain goals, but in the end, if an external observer can physically observe a difference in the behavior of two computations that are equivalent by your notion, then your equivalence may be convenient but it is not precise. In the end, a computation *is* a dynamic process, which is uniquely described as a state machine behavior.

In order to define a computation uniquely, you need to add more constructs, such as an evaluation strategy, a cost model etc., but once you have everything, you'd be describing a particular state machine.

> you could say that what these really are is computable functions and it would be as illuminating as your bare-faced assertion.

But that's just not true. Computations are not functions; they are processes that compute functions. Again, this is easy to prove as an external observer could easily differentiate between two different computations computing the same function. So identifying a computation with a function is imposing an equivalence that may be useful for some purposes, but is not precise. You could make it precise by adding additional semantics, such as declaring that a certain monad describes the steps of your process, but then you'd be describing a state machine...

[–] **notfancy** 4 指標 2 年前

> I don't understand. Are automata not well-known mathematical objects?

NFAs, DFAs, PDAs, TMs have all rather different mathematical formalisms (quadruples, quintuples, septuples, what have you.) Saying "everything is an automaton" (paraphrasing) is imprecise; saying "everything is an automaton, that's a mathematical fact" (again, paraphrasing) seems to me unjustifiable. Again, the more abstract formalisms vary somewhat wildly, with each author pushing his or her preferred one (the Wells with Chu spaces, Backhouse with Galois connections and relational programming, etc.) as the "true one" formalism.

> which is then a particular *behavior* of the machine

Again, what you call "behavior" someone else calls "semantics" and you're still mired in the dynamic/static divide:

> For example, in the lambda calculus, call-by-name and call-by-value are two *different* state machines

I do think you're committing a category error here by conflating semantics, evaluation and implementation. Some of these views are static (a reduction diagram is given once and for

all in its complete form even if it's really big or actually infinite), some are dynamic (an abstract reduction machine does actually transition between well defined states in a well defined temporal sequence.) CBN and CBV don't escape this dualism: you have the relevant theorems, the algorithms and the actual implementations.

> performing *different* computations.

There's a subtlety here in the case of CBN and CBV in that while the *operational* semantics are rather different, the *denotational* semantics are not so much: CBV computes a strict subset of the computable functions that CBN computes, and if CBV computes a value it is the same value CBN would compute.

All this goes to reinforce my impression that you're painting with too broad a brush and in the process you cover over some differences that not only I think are worth making from a theoretical standpoint, I also think matter in practice:

> In the end, a computation *is* a dynamic process, which is uniquely described as a state machine behavior.

Reasonable people differ on this issue. I don't see how it can be adjudicated one way or another by fiat; I think that believing otherwise is unnecessarily dogmatic.

To go back to the topic of the post, I find Lamport has very set opinions on what is worth doing and what is not, and I don't necessarily agree with him, seeing as reasonable people don't either, even if I acknowledge and respect his expertise. He feels more at home with a dynamic treatment of semantics, good for him! In the end he is peddling his TLA+ over the competition, so I would take his word on the issue with a grain of salt.

> Computations are not functions

If you've disproved Church-Turing I'm sure the world would want to hear all about it. Sorry for the snark, but you do have to address the issue of why you think C-T is false and/or irrelevant if you want to go further with your argument.

> as an external observer could easily differentiate between two different computations computing the same function

In the purely theoretical ("mathematical") sense this is false, because it reduces to halting (proof sketch: disequality is semi-decidable because while you can eventually observe state divergence you can never observe full agreement for non-terminating computations and/or infinite domains.) If you're talking about physical realizations ("CPUs and assembler") I can't see it as anything else but a non-sequitur: your original point is entirely theoretical.

> a certain monad describes the steps of your process, but then you'd be describing a state machine...

Monads don't "describe state machines"! At best you can say that Moggi-style "notions of computation" describe small-step reduction semantics for ad-hoc interpreters with a more-or-less modular framework, but again, I think it is wrong to equate small-step reduction with a purely dynamical process. If the formalism were "just" a fancy state machine I would think it very hard for it to be useful for anything besides "running" it; while in fact from small-step semantics you can derive a host of static properties (soundness, completeness, etc) by purely syntactical means (structural recursion.) Of course "monads run" (hey, Haskellers even call the evaluation function `runMonad`) but I don't think it's the most interesting, let alone only, aspect of monadic semantics. Plus you have the MacLane-style "adjunctions give rise to triples" of monad use that is more about the meaning of the data than of the operations on them.

In the end I would urge you to articulate clearly and transparently the underlying issue that compels you to favor the dynamical view over the static one. I know from your past interventions that you're deeply skeptical of the relevance of the "static" approach to program verification, including but not limited to Hindley-Milner typing, and I suspect this is

at play here too. I applaud your attempts at denouncing "lambda zealotry" (to build a particularly ungainly straw man; I hope this serves) but please don't be tempted to fall into the opposite flavor of dogmatism.

永久連結 **embed** 儲存 上層留言 贈送金幣

[–] **pron98** [S] 2 指標 2 年前*

> NFAs, DFAs, PDAs, TMs have all rather different mathematical formalisms (quadruples, quintuples, septuples, what have you.) Saying "everything is an automaton" (paraphrasing) is imprecise; saying "everything is an automaton, that's a mathematical fact" (again, paraphrasing) seems to me unjustifiable. Again, the more abstract formalisms vary somewhat wildly, with each author pushing his or her preferred one (the Wells with Chu spaces, Backhouse with Galois connections and relational programming, etc.) as the "true one" formalism.

I happened to run across Chu spaces and I remembered hearing the name somewhere...

I think there's a bit more to this. First, all automaton models (and all rewriting systems) are *trivially* special cases of abstract state machines. An abstract state machine is <S, R>, where S is a possibly infinite, possibly even uncountable set of states, R is a next-state relation on S (multiple next states indicate nondeterminism), and a state is any mathematical structure in a sufficiently rich foundation (ZFC, HOL or whatever). The model of such a program in TLA the set of all its possible behaviors, namely a set of (possibly infinite) sequences of states.

Now, obviously there isn't "one true" formalization of anything. If two formalizations are equally descriptive, it's meaningless to say which of them is "true", but that is not to say that all formalizations are equal. First, there's the objective question of power and descriptiveness. A "baseline" formulation must be *precise*, namely it must be able to distinguish between any two things that we consider as different algorithms. Secondly, there is an imprecise, vague and very human-centric notion of simplicity. It is possible that some FP denotational semantics are capable of distinguishing between any two different algorithms, but that requires many more, very advanced, mathematical concepts. Finally, there's another human-centric notion of "naturalness", which is the question of how easy it is to apply a given formulation in as many cases as people are interested in, and how necessary it is. You can think of various differential equations as Hilbert operators, but appealing to operator theory -- while a useful proof technique in some cases -- is not necessary in most practical circumstances. I think Galois connections are a proof technique (which I believe can be applied directly in TLA+, BTW), and from what little I understand of Chu spaces, I believe they fall in the same category.

So I'm not saying that TLA+ is the one true formalization of programs. It is not the only general way to specify programs nor is it always the most convenient or natural way. But I think that 1/ it is precise, sufficiently simple and sufficiently natural to serve as a good *baseline* formalization (useful proof techniques notwithstanding) and 2/ I am not aware of any *other* formalization that scores higher on these points; certainly not one that has been put to the test of significant real-world use.

永久連結 **embed** 儲存 上層留言 贈送金幣

[–] **pron98** [S] 1 指標 2 年前

Thinking about it some more, it is not clear to me what you see as the *essential* difference between what you call a static view vs. a dynamic view, and why you think abstract state machines can "just" be run but monads can be "reasoned about". To me, this seems like a (very, very slight, nearly inconsequential) difference in notation. TLA+ is exactly as referentially transparent -- and just as amenable to logical reasoning -- as

any "static" description. As I said, I just personally believe that it's easier to teach developers FOL set theory and a bit of temporal logic than type theory, and that is a huge *practical* difference that matters to developers a lot, but it's not an essential one. The two amount to the same thing as far as reasoning about programs is concerned.

永久連結 **embed** 儲存 上層留言 贈送金幣

[–] **notfancy** 1 指標 2 年前

I don't see the static and dynamic views "essentially" different, whatever that might entail. It is clear that state machines can be reasoned about formally, as there are plenty of theorems about them and through them in the various settings in which they are the formalism of choice.

Since I don't know the first thing about the theoretical underpinnings of TLA+ and its power, I will take your word for it.

永久連結 **embed** 儲存 上層留言 贈送金幣

[–] **pron98** [S] 1 指標 2 年前*

Well, they're the formalism of choice in the vast majority of software verification techniques in academia as well as the industry... TLA+ is just a very rigorous, *extremely* simple, logically complete (except for probabilistic models) formalization of any abstract nondeterministic state machine.

In any event, TLA+ is quite simple. TLA+ = TLA + ZFC, and TLA = linear temporal logic + invariance under stuttering (i.e. there is no transition if change cannot be observed) + state hiding (called temporal existential quantification) + action formulas. The latter are just predicates relating the current state with possibly multiple next states[1]. So, e.g., you can write `x' = x + 1`, but you can just also describe that as `x' ∈ Int ∧ x' - x = 1`. Nondeterminism can be introduced as `x' ∈ {1, 2, 3}` or `∃ t ∈ {1, 2, 3} : x' = t` or even `x' ∈ Int ∧ x' + y' = 5`. Because the state machine is completely specified in this simple logic, simulation becomes logical implication (under some refinement mapping), so A refines B is simply `A ⇒ B`.

What I like about TLA+ so much is that it's been refined over a few decades by both researchers (Leslie Lamport, Martin Abadi, Ernie Cohen, Jim Horning and Stephan Merz) and engineers, and has been designed in such a way that engineers can relatively easily use it to specify very large real-world systems. This assumption has been tested -- quite successfully -- numerous times. The theory continues the work of Pnueli and Manna, with a strong emphasis on practical usability. In other words, TLA+ has grown at the very core of software verification, but whereas most of the field has been about tools (model checking, abstract interpretation etc.), TLA+ is mostly about specification. It is a language that is at once so elegant, so clear and easy to pick up, yet so rigorous, that it makes other formalisms look clunky in comparison (I'm not saying the syntax is particularly beautiful, but it's more than OK).

In case you're interested, this is a very short historical and philosophical background to the design of TLA+ (Lamport), and this is a (long) in depth discussion of the logic (Merz); this is an earlier discussion of the TLA logic (Abadi + Merz). The only rigorous, mathematical languages that have been as successful as TLA+ in the industry are quite similar conceptually, but less minimally elegant.

[1]: Although, as I've shown in the example I added to my last comment, TLA+ allows recursive operators that can describe any algorithm, completely

"statically", i.e. without a state machine using recursive functions.

[–] **pron98** [S] 1 指標 2 年前*

> NFAs, DFAs, PDAs, TMs have all rather different mathematical formalisms

Sure, but they can all be trivially expressed as *abstract* (nondeterministic) state machines (where every state is an arbitrary mathematical structure). In any event, TLA+ does not limit you to one kind of state machine (and it trivially supports nondeterminism, so all nondeterministic state machines are also easy to directly specify). If you want your state machine to encode beta reductions, that is trivially expressed; hell, you can specify your machine to be cellular automata (but probabilistic machines are indeed not TLA+'s forte; they're very easy to specify but pretty much impossible to to reason about, although people have had success specifying and checking them, too, using dirty tricks).

In fact, if you want you can specify your entire computation as a function (i.e. no steps are taken). I don't think it's the recommended approach, and if you choose to model-check the specification rather than deductively prove it, the checker's output might not be as helpful, but there is nothing in TLA+ that says that you must ever take a step. Functions can be arbitrarily complex. Actually, from the logic's perspective, this is the default. A plain TLA+ expression just denotes a function. You must explicitly ask (using temporal operators) to take a step (in which case the expression specifying the step must be an *action predicate*).

Now, I don't know whether there is a "one true formalism", but I don't think you can precisely describe a computation (and by that I mean that an equivalent computation according to your semantics cannot be told apart by an observer) without *some* notion of a state machine: in other words, there are many ways to describe a computation, but if they do not specify a state machine, then they don't *uniquely* describe a computation. Maybe type-theory's definitional equality has this property, too, but I don't know enough about it to tell. Maybe it's even too strong, namely that two computations that are *not* definitionally equal also cannot be told apart by an observer; don't know.

> the denotational semantics are not so much

Right, but this brings us back to equivalence relations on computations. There are many useful equivalence relations, but only a few (or perhaps just one -- bisimilarity/trace-equivalence) that cannot be told apart by an observer, which makes it the most precise, or, more correctly, arbitrarily precise. In TLA+ you can "care" about denotational semantics alone, small-step or big-step. You choose, and all of them are very easy. It's just a matter of how fine (or crude) you choose to make each step, or even not take any step at all, and opt just for denotational semantics. It's all good[1].

And, as I said in another comment, denotationally Go has been a solved problem for a very long time, yet it is "operationally" solving it that is cause for celebration...

> In the purely theoretical ("mathematical") sense this is false, because it reduces to halting

I am talking about something much more basic. Regardless of the form of execution of the computation, an observer can trivially tell if you're running (in a black box) a bubble-sort or a merge-sort even though they compute the same function. They might even be able to tell if you're doing CBN or CBV.

> Reasonable people differ on this issue. I don't see how it can be adjudicated one way or another by fiat; I think that believing otherwise is unnecessarily dogmatic.

Sorry, I didn't mean to say that computation can be precisely specified *only* by state machines, only that every computation can be precisely and uniquely specified by a

state machine. And while it is true that any essential property a computation has (e.g. computational complexity -- every computation has it) can be encoded in its denotation, I don't think there are reasonable people who disagree that complexity is essentially there.

> To go back to the topic of the post, I find Lamport has very set opinions on what is worth doing and what is not, and I don't necessarily agree with him, seeing as reasonable people don't either, even if I acknowledge and respect his expertise.

Well, regardless of the theoretical merit of that approach, there's no denying that formally specifying computations with temporal logic is by far the most widely used formal approach in the industry. And that is why it appeals to me: it is as rigorous and general as you like, yet requires no more than high-school (or first-year college) math, and is very easily learned and used by plain developers like myself. Also, it is not *just* Lamport's approach, but that of the majority of the verification community -- probably in large part due to its simplicity and practicality.

> If you've disproved Church-Turing I'm sure the world would want to hear all about it. Sorry for the snark, but you do have to address the issue of why you think C-T is false and/or irrelevant if you want to go further with your argument.

On the contrary. Church and Turing were very explicit that they're talking about processes that *compute* functions (computations); not functions. A computation is something that possesses a positive (perhaps infinite) computational complexity. I don't know how to assign complexity to a function.

> At best you can say that Moggi-style "notions of computation" describe small-step reduction semantics

They can describe small-step or large step. Basically any abstract state machine. TLA+ also does not limit what can be done in each step. In TLA+ you can specify a computation where sorting is a mathematical function (takes 0 time), or a computation (with complexity). It's completely general.

> In the end I would urge you to articulate clearly and transparently the underlying issue that compels you to favor the dynamical view over the static one.

What compels me to favor the dynamical view is that I write algorithms, and their complexity is as important to me as their denotation; complexity is very easily expressed and verified with the dynamical view (I know it's possible with the "static" approach; it's just harder).

But more than that, as I've said, I am drawn to the simplicity of the approach, and the confidence that any programmer can pick it up in a few days (perhaps not the art of what to specify, but the theory of the model). In short, I am so fond of TLA+ because it lets me specify any algorithm, at any level of detail, and it does so with only a couple of new mathematical ideas that I didn't learn in high-school (basically, basic temporal logic); it gets the job done and its easy. I am sure other approaches get the job done, too, and perhaps they have their advantages, but they are all *much* harder (in requiring new math) than TLA+. Also, as I don't have the months or years required for semi-manual deductive proof of very complex software, I really need the model checker. Model checking in the more "static" approaches is still in its infancy (liquid types are interesting, but I'm not sure they're powerful enough to get the job done). Although TBF, TLA+'s bundled model-checker (you can use one or two others) is a limitation. It is an explicit-state model checker, while modern ones are much more advanced. I think people are working on a more modern model-checker for TLA+.

EDIT:

[1]:

To make this concrete, let me reuse Lamport's example of Euclid's algorithm. The "machine" specification looks so:

```
CONSTANTS M, N
VARIABLES x, y

Init ≜ x = M ∧ y = N
Next ≜ ∨ x > y ∧ x' = x - y ∧ y' = y
       ∨ x < y ∧ x' = x ∧ y' = y - x
Euclid ≜ Init ∧ □[Next]
```

But you can describe the same algorithm "statically" so:

```
RECURSIVE Euclid(_, _)
Euclid(x, y) ≜
    CASE x = y → x
       □ x > y → Euclid(x - y, y)
       □ x < y → Euclid(x, y - x)
```

The latter eschews the state machine completely (and is "executed" in zero time), which you can note by the lack of temporal operators (perhaps confusingly, the `CASE` expression uses the same symbol □ as the temporal operator □, but that's just syntax)
.

Of course, the semantics can be specified completely declaratively, i.e. with no algorithm at all:

```
Divides(p, n) ≜ ∃ q ∈ Int : n = q * p
DivisorsOf(n) ≜ { p ∈ Int : Divides(p, n) }
SetMax(S) ≜ CHOOSE i ∈ S : ∀ j ∈ S : i ≥ j
GCD(m, n) ≜ SetMax(DivisorsOf(m) ∩ DivisorsOf(n))
```

I hope that these examples convey both the simplicity (high school math + a couple of temporal operators) and the flexibility of TLA+. It really can be completely learned (though not mastered) in a day (except maybe refinement mappings, that specify how a finer spec implements a cruder one).

Deductive proofs in TLA+ are another matter, but I assume most people don't use them anyway -- certainly not at first -- as they're just too much work for the large, real-world systems that TLA+ specs usually specify.

BTW, if you prefer imperative programming rather than logic expressions, you can write the exact same thing as the first specification in PlusCal (it will be translated inline by the IDE to something almost identical to the first spec):

```
--fair algorithm Euclid {
  variables x = M, y = N; {
    while (x ≠ y) {
       if (x < y) { y := y - x }
             else { x := x - y }
    }
  }
}
```

π