Dafny’s shortcomings

set a | a in this.mymap :: this.mymap[a]

Error: more than one declaration of variable name: a#5

requires myobject != null;  
reads myobject.member;

This is not entirely unexpected as Dafny is still under development.

requires myobject != null;  
reads myobject != null && myobject.member;

This implies that the clauses are completely independent of each other, and unfortunately introduces redundant code and by experience, verbosity reduces clarity and overview.

method tinker()  
 modifies this;  
{}

method test()  
 requires this.member != null;  
{  
 tinker();  
 assert this.member != null;  
}  
 reads myobject.member;

The Visual Studio plugin provides syntax-highlighting and verification while typing. The Dafny verifier is on by default, but may be turned off. This is useful when your projects get large enough that verification takes 30 seconds or more to complete. Given that the call to the verifier blocks the thread responsible for Visual Studio’s coding interface, delay between every minor change makes for a less than pristine development environment.

method test()   
 modifies this;   
{  
 //do stuff not involving member  
}

method somewhere\_else()  
{  
 this.member := null;  
 this.test();  
 assert this.member == null;  
}

method test()  
 requires this.member == null || this.member.valid();  
 modifies this;  
 ensures this.member == null || this.member.valid();  
{  
 //do stuff not involving member  
}

method somewhere\_else()  
{  
 this.member := null;  
 this.test();  
 assert this.member == null;  
}

This assertion fails, but it is not entirely obvious why that is. Our intention in this code is to prove that member is unchanged after passing through test(). This should be a trivial task, in my opinion, an implicit one, even for most programming languages not intended for verification, as member is obviously never even involved in the code. Dafny however, simply discards all accumulated knowledge about all objects found in the modifies clause, leaving us having to explicitly tell what to regain. You would think the two equal requires and ensures clauses would be capable of that, but the problem is that we accept more than one state, and Dafny cannot see that there is no switching between them.

Our task becomes one of isolating the states. Knowing that the different clauses are independent of each other, and that the single expression allowed in each one is not capable of saving information to be used in the other clauses, this, rather modest verification, could provide a significant challenge.

One naïve solution would be to make one copy of the method for each state, and test the state at runtime to figure out which one to call. Another one is creating a ghost variable copy of the object you are verifying before the call and assert their equality afterwards. Dafny has no deep-copy feature, however, making us having to provide one at the instantiation of the original, and mirroring every interaction with it throughout the object’s life. Both of these solutions I consider horrible options and something I would be ashamed of showing people.

Fortunately, you won’t need a complete copy of the object (ghost or not). In fact, you only need an integer that can be associated with each possible state. Make sure that the integer is not referred to by any of the objects in the modifies clause, as this will cause its information to be discarded as well.

The parentheses are required. And and Or does not play nice together as in other languages. Also, remember that when one of two booleans fail on the left hand side of an Or, neither’s state is determined on the right hand side, thus me checking for not null before calling valid(). Of course since we in this example were working with no more than two states, we could have just as well used a Boolean variable instead of an integer. This is more scalable, however. I have yet to create a method where one would not need to know the state beforehand that is capable of this proof. Furthermore, since we cannot pass ghost variables as arguments to methods, our state becomes unnecessary baggage at runtime.

method test(state: int)  
 requires (state == 0 && this.member == null)  
 || (state == 1 && this.member != null  
 && this.member.valid());  
 modifies this;  
 ensures (state == 0 && this.member == null)  
 || (state == 1 && this.member != null  
 && this.member.valid());  
{  
 //do stuff not involving member  
}

method somewhere\_else()  
{  
 var state := 0;  
 this.member := null;  
 this.test(state);  
 assert this.member == null;  
}

**Paxos explained again**

I will introduce concepts as they are required and not unnecessarily modularize a system where a holistic perspective is crucial to its understanding. This is to reduce the time between when a concept is introduced and it no longer being relevant. Also, to stay somewhat aligned with Dafny’s validation syntax which we will explore later, I will explain in bullet-points of restrictions and conclusions.

Paxos is a protocol for reaching consensus amongst a group of agents. In the Paxos algorithm, an agent performs one or more of these three roles;  
**Proposer:** Responsible for initiating an instance where consensus is sought.  
**Acceptor:** Decides which proposals to accept  
**Learner:** Detects consensus when a majority of acceptors agree.

In communication amongst each agent, two variables are exchanged; Round and value. Round is a unique integer that can be considered an identification of a specific proposal and value is any type or object that we would like to achieve consensus about. Our first important restriction becomes…

* All proposers starts with unique rounds

Because a unique round means that we know its corresponding value comes from the same proposer, we can guarantee that…

* Where we find equal rounds we also have equal values

Starting an instance of Paxos, a proposer picks a round and the value it intends to issue. The round might be the agents unique ID or a previously agreed upon starting number. The value might be a reference to a task to perform from a queue. Then, with that data attached…

**Proposer broadcasts a “Prepare” message to all Acceptors**

Acceptors are completely independent of each other. The receiving acceptor’s job is now to inform the proposer of a previously accepted round and value, if any, and promise to ignore any future messages with round less than that provided by the proposer. Let’s phrase that as a restriction.

* An acceptor will ignore a message where provided round is less than promised round

Do note that a promise does not imply acceptance, and they should not be confused with one another. We will be asked to accept a round and value at a later point. If no earlier promise of a higher round has been made…

**Acceptor answers Proposer with a “Promise” message**

If, however, an acceptor has promised a higher round, we will never hear from it. We can on the basis of our above restriction thus say for the proposer…

* All encountered accepted rounds, if any, will be less than our issued round

They cannot be equal since we are using a unique number. At this point we need to introduce another restriction.

* A proposer will only ask to accept if a majority of acceptors have promised/are prepared

Any acceptor having accepted a round then implies that the majority of all acceptors were prepared for that round and the proposer issuing that round received those promises. Assuming our proposer have received promises from a majority of acceptors, we know that we would have at least one of those acceptors in common with any previous majority groups. That is a mathematical inevitability. We also know that our proposed round is greater than any previously accepted rounds. If not, we would never hear from the majority and thus never achieve one of our own. We can now, for all acceptors, promised or not, guarantee the following statement is true.

* No acceptor has accepted any round equal to or larger than our proposed round

Step 1 is finished. A majority of acceptors are receptive to our proposal, even if some acceptors never get a message or ignores it. We have blocked the possibility of a lesser round being accepted while we are proposing ours.

Optional optimization: For reduced bandwidth use, several instances of Paxos may share this first step’s message exchange. The proposer may send a single prepare message with a list of values. To allow for multiple instances, one needs to include a unique and static identifier for each instance. This is called a *slot*. Rounds need only be unique within one slot, so the list may share a single round. The receiving acceptors will now treat each slot independently yet answer with a single promise message.

Step 2.

**Proposer broadcasts an “Accept” message to all Acceptors**

If the acceptors have not made a new promise with, and/or accepted a higher round to another proposer since our request, they will accept our round and value. Any answers to subsequent requests will contain that accepted information. Upon accepting a new round…

**Acceptor broadcasts a “Learn” message to all Learners**

The learner will keep track of which acceptors have accepted the value of the highest round. Note that none other than the highest accepted round will ever reach a consensus. Once a majority is in agreement, the value is marked as learned and a consensus has been attained. There is still a flaw that must be addressed with another restriction however.

* A proposer must re-issue the highest encountered accepted round’s corresponding value, if any.

We need our proposer to remember the response of the acceptor with the highest accepted round. Consider that a consensus might be reached through a lower round than ours before we receive a majority of promises. Then that accepted round and value will surely be provided by at least one of the acceptors we have in common. If we were to issue our own value at this point, the higher round would override the previous accepted value and the consensus would be lost. We can, considering this restriction, then assert that…

* Once a consensus is reached, the learned value cannot change.

That’s it.

Dafny

Dafny builds on Boogie

The main goal is to replace the time spent debugging and patching with time spent expressing validation schemes.

The tutorials and documentation provided is incomplete. I will not assume the reader has read the rise4fun guide, but I do not intend to replicate it. I will go through some of the basics, where appropriate, just enough to understand the material that’s not found in the official documentation.

Dafny -  
does support generics

does not have any standard libraries.  
does not interact with the OS in other ways than print.  
does not accept arguments to Main.  
does not support concurrent programming.  
does not support inheritance/subclassing or nested classes.

Types are either one of bool, int, nat, array<\_>, set<\_>, seq<\_>, map<\_,\_>  
(nat is an unsigned int)

Objects conversely, are instances of classes.

Methods have 4 possible annotations: requires, ensures, modifies and decreases. The requires and ensures clauses are Boolean expressions, while modifies takes a set of objects and decreases takes a numerical expression useful to guarantee bounds within loop constructs. It is not used in this project and you may refer to rise4fun for further explanation.

Quote from rise4fun “Dafny lifts the burden of writing bug-free code into that of writing bug-free annotations. This is often easier than writing the code, because annotations are shorter and more direct”. In fact, having spent about a week’s time on the actual code and several months on writing annotations, I tend not to agree with this being easier. Nor is it shorter, considering the annotations also takes more space than the code, even after all reusable expressions are refactored out.

I like comparing the difference of methodology in Dafny to the difference between pure functional and object oriented programming languages, but I must confess, having a highly dynamic OOP language like python being my first real programming language, it might not be entirely fair to the more experienced programmer to make this comparison. Even more so when I am neither proficient in C++ nor Haskell, which I consider the most iconic samples of the two paradigms.

Even with examples provided and explained, when the foundational concepts are still alien, I found it extremely hard to generalize and port the necessary lessons to benefit my own cases.

Almost all validation annotations are redundant! Due to artificial restrictions by Dafny developers, likely intended to reduce validation-time, the scope of validation does not extend beyond a single method or function. This increases the amount of annotations needed by orders of magnitude, perhaps comparable to the time saved. They are there first and foremost to specify *what* to prove, not *how* to do it. With the exception of assumptions made about data from outside the system, no information provided in annotations is crucial to the validation of any statement. The code’s behavior is independent of its validation, and by inference, all statements about the code also is. The only useful part of this explicit validation, as I honestly see it, which is ensuring the programmer that a complicated system works “as intended”, is vastly overshadowed by the sheer amount of annotations needed to manually and meticulously set the stage for it to work. The majority of your time goes into validating things that could be implicitly derived.

If you want to validate your code, you need to describe how it is supposed to behave. Is that not the point of the code in the first place? Is not the need for a validating statement an admittance of a lack of clarity in either your code or the language itself? If you could describe the functionality in a simpler way, should not that be the more optimal way to program?

If you wanted to assert something it should be quick to add an annotation, let the validation process cascade through the syntax tree, see the results and move on. Subsequent annotations may build on previous conclusions and finish their evaluation faster.

Imagine hovering your cursor over a variable and seeing things like type(s), value distribution, read/write access ratio, and other helpful statistical data at that point in the code.

Unhandled exceptions cannot happen,

I would be completely fine with several hours of validation time for a system such as the one I have built! I will probably be unable to program faster than the computer can validate anyway. The problem lies in the fact that Dafny evaluates the entire system statelessly in one go after compiling. Repeatedly so, in case of Visual Studio’s language extension. This does not allow for any coding to be done in the meantime. Many applications today still exhibits this archaic behavior of starting with arguments, process it, spit out a result and then terminate, when a potentially continuous service model would better fit the task. The naïve approach of repeatedly parsing, compiling and validating the entire source is something I did not expect to see from professionals in a project not on a strict deadline. This is unfortunately mainly because all validation is done on the compiled product. Although I can imagine a two-step concurrent compilation system that would severely cut the time of continuous background compilation, the fastest option is to validate the parser’s abstract syntax tree (AST) or a derived higher order abstraction.

The benefit of end product validation is that it is independent of language and compiler.

The conditions for an expression to be evaluated true could be localized such that no redundant cascading reevaluation is performed unless the edited code fall outside of these local constraints. If one …

I would expect any moderately clever IDE to restrict its focus to only the code block being edited if it had any features that require parsing code while the user is typing. Once the behavior of that block has been abstracted, any cascading consequences may change the state of the outer closures. It may not, of course, change any conclusions reached about other code blocks in the same closure. This allows for concurrency, restricts the domain affected by change and eliminates redundant parsing and validation. That is something syntax-highlighting, validation, compilation and surely plenty other features all can benefit from.

**Validation for existing languages**

If you access something through a pointer or object without checking for null it is obvious yet implicit that not null is an assumption that must be satisfied in that block of code. This is largely independent of language and may be checked in either the IDE or the compiler.

A more powerful unreachable code detector. Suppose, in an if statement, you compare two separate objects with == (comparison of identity) when you should have used .equals() or something equivalently content sensitive. Reaching the conclusion that the expression is redundant, and always evaluates to false, does not require a very elaborate analysis. Conversely, a redundant expression evaluating to true highlights the same mistake just as clearly, even if no code is unreachable, and yet, this degree of insight is still uncommon in IDE’s. With more advanced validation this would be possible even when it is not immediately clear that the two objects or pointers cannot be the same.

Dafny’s methods have a scope of influence determined by the modifies clause

**-obsolete-.dfy**

Dafny code is compiled as a library and used by a ‘client application’

My first attempt incorporated a hierarchical architecture, DummyNetwork at the top, Interface, Group and then Proposers, Acceptors and Learners.

Because the validation of the Paxos algorithm depends on the interaction between various agents, I sought a way to cheat Dafny into believing the entire system was self-contained in one process, while allowing the client program to utilize parts of the library independently.

DummyNetwork, the overarching singleton, is supposed to pass along messages to the correct Interface based on dest\_ID, the first parameter in all its methods. The class is useless during runtime, as a real Ethernet connection is doing this job, but it serves as a way for Dafny to see that a message gets where it is supposed to within the system. An implementation of random failures was considered, but Dafny has no way to get random values. Those would need to be provided through the client application, in which case we wouldn’t really have to do anything as we don’t use the class at all. I can assume Dafny would cry about the scary unknown value it could get, and not enough time has been spent on my part to figure out how to validate non-deterministic code. Dafny has no concept of probability, so it might even be impossible.

Interface has two sides, one inside for receiving calls from Paxos roles and one outside for receiving from the network. An interface represents one agent. By default, the object simply bridges calls between the DummyNetwork and the agents’ Paxos objects. The client application should create a subclass of Interface, substitute DummyNetwork with a real network and choose a fitting protocol.

Group represents an isolated collection of agents from one agents perspective. It includes arrays of all participants’ IDs for each Paxos role. These are used as a list of recipients when broadcasting to all agents performing a given role. A single agent’s ID may be mentioned in all three arrays if all roles are performed. Group also maps slot\_IDs to local instances of the Paxos algorithm. Also here, one map for each role. No optimization is implemented for multiple slots.

The Proposer, Acceptor and Learner classes will be covered in my next version.

Since valid() is a member of grp we can’t include a check for not null inside it because we depend on the condition to call valid() in the first place.

requires grp != null && grp.valid()  
 && grp.interface.valid()  
 && forall i :: i in grp.interface.net.interfaces  
 ==> grp.interface.net.interfaces[i] != null;

reads if 0 in this.myMap && this.myMap[0] != null then this.myMap[0].valid.reads() else {};  
  
reads set x | forall y :: y in this.myMap && this.myMap[y] != null && x in this.mymap[y].valid.reads();

If a constant is asserted as a key in the map, the value’s members are accessed successfully. If, however, we generalize the expression to account for all the elements in the map with a forall loop, Dafny seems to fail with the error: insufficient reads clause to invoke function. In English this becomes: For each object in myMap, add all the valid function’s reads to a set and finally return that set. So in other words. Since we require all these mapped object to be valid, we must also read everything that the valid function reads for all objects.

My approach was to write the program so that is works, then validate it. That was a very bad idea. The program will only finish compiling successfully if the program is validated. When the validation process started, the program was already big enough to loose ones overview working with it. Having to further add more than twice the code’s worth of annotations, partly due to Dafny’s severely limited scope of validation, this environment was the worst kind to learn in.

**proposer.dfy**

asd

**acceptor.dfy**

asd

**learner.dfy**

asd

Compiler bugs and issues due to language features are quite often indistinguishable to the novice. The former tends to exhibit a weak heisenbug syndrome, appearing sporadically and not disappearing before one has painstakingly worked around the problem. Changing unrelated code seems to remove most of them, so when an issue has been circumvented, one might want to revisit the code after some time if one suspects it was a bug.

“Error: assignment may update an object not in the enclosing context's modifies clause” Well how do you know that if you need me to tell you what is being modified?

Library interface

Methods do not return values. They modify out-pointers given as arguments. Anonymous constructors are named \_ctor().

Other notable work

Raft, Verve, vgo