Dafny Material to support the

Imperial College Course “Reasoning about Programs”

Sophia Drossopoulou  
3rd June 2013

**Abstract**

In this short document I describe the rationale and the contents of the material which Will Sonnex and I developed to supplement our First Year Undergraduate course: “Reasoning about Programs”.

The development of the Dafny material was financed through a gift by Microsoft Research Connections, and a visit to Microsoft Research. The gift and the visit were organized by Judith Bishop.

**Background** The Undergraduate course: “Reasoning about Programs”

**Target audience of the course “Reasoning about Programs”** The course is taught in the 2nd term of the 1st year. The students have been taught Haskell and Java. They have had a course on First order Logic, and some training in proving, but are not yet very good at it..

**Aims of the course “Reasoning about Programs”:**

1. Write specs for functional code.
2. Detailed**, precise proofs** using induction (various); explicit proof steps, justification, and precision in application of lemmas.
3. “Architecture” of the proof – discover auxiliary lemmas.
4. Develop specs for imperative code, with pre and post conditions. Also defining and using appropriate predicates to specify and prove the code.
5. “Architecture” of the proof: invariants, assertions, specs of auxiliary code. Also, find the lemmas necessary to prove the code, and apply these lemmas.
6. Detailed Hoare-logic proofs, using, essentially the SSA equivalent.
7. Refinement: Develop code from spec: Form spec to invariant, and from invariant to code.
8. Notation (use, develop new). For example, notation like   
    a[0..start)<pivot=a[start..middle]<=a(middle..a.length)

is a convenient shorthand for  
 ∀i:ℕ. 0≤i<start ⇒a[i] < pivot ∧ ∀i:ℕ. start≤i≤middle ⇒a[i] = pivot   
 ∧ ∀i:ℕ. middle<i<a.length ⇒pivot < a[i]

1. No framing *– perhaps this should change*?
2. No concurrency.

**Why use a tool to support the Reasoning Course?**

* Some students like using tools.
* Tools make the area look more relevant; several students asked whether they would be using Dafny in their careers.
* Immediate feedback and soundness of the tool – ie great when proof has been found and accepted.

**Why chose Dafny as a tool to support the Reasoning Course?**

* Dafny supports theorem proving
* Dafny supports both functional and imperative programming. On the other hand, it does not support concurrency, which would have been beyond the scope of a First Years Undergraduate course.
* Dafny is constantly improved and upgraded, and there is a wide range of available exercises.
* Dafny has good support in Visual Studio, and one can “cheaply” start Dafny on rise4fun - even though Visual Studio is much better.

**The Dafny Material**

**“Philosophy” of the Material**

The aim overall is to show to the students how they can use Dafny to check their proofs and their programs, and also how to use it to incrementally develop their proofs/programs. Some if this is shown in an upcoming episode of Verification Corner.

1. Use only a subset of the Dafny features. After all, the material is supposed to support another course, and not teach a new programming language. Luckily, Dafny syntax is easy to pick up when you have Haskell and Jva experience.
2. Explain that Dafny does several proof steps from a hand-written proof implicitly, and show the relation between the hand-written and the Dafny proof. Thus, Dafny cannot substitute the detailed proof writing skills we need, but it can support checking the larger steps in a proof.
3. Use Dafny to explore possible specifications and to find counterexamples.
4. Stress the incremental, exploratory nature of discovery of the “architecture” of proof. Dafny supports that through assume-clauses, and ghost method declarations.
5. Program and proof development through refinement.
6. Show “wrong” code, explain the reason for the error being reported, and show and how to fix.

**Contents of Dafny Material**

All material is available on Skydrive, and is detailed below. Some more material is in preparation. The Lectures are available in one file, called Lectures, while the exercises are on a separate file each. Together with each exercise there is the corresponding Dafny code (in a file called xxx.dfy), and the sample answers (in a file called xxx\_SA.dfy).

1. Lecture 1: Introduction to the tool Dafny in Visual Studio
2. Lecture 2: functional part of Dafny, from Haskell to Dafny, from handwritten proofs to Dafny proofs.
   1. Question 1: Lists   
      develop proofs, specifications and functions, give counterexamples
   2. Question 2: Making Change   
      develop proofs, specifications and functions, give counterexamples
3. Lecture 3: Proof Development – Quicksort Proof developed incrementally, in a top-down manner
   1. Question 3: Tail Recursion  
      Interesting Cases for application of inductive hypothesis, intermediate lemmas, and tail recursion.
   2. Question 4: Tree Encode/Decode  
      Interesting Question, succinct but non-obvious program (Good programmer puzzled about correctness)
4. Lecture 4: Imperative Dafny and Loop Invariants
   1. Question 5: Loops  
      loops, nesting, triple nesting, sequences; only simple arithmetic involved
   2. Question 6: Loops and Arrays:

Finding max or array, Product of array, Sorting an array. Good mix of simpler parts and parts where they need to think, or prove and use Lemmas.

1. Lecture 5: Incremental Development of Imperative code using assume statements – bubble sort.
   1. Question 7: Insertion Sort of an Array. The invariant is more sophisticated than that of bubble sort, and proving preservation of this invariant requires the proof of some useful lemmas.

**Dafny features taught**

Use a restricted, but powerful subset of Dafny, and follow a certain style in code/specifications. Here is the list of features that we found were necessary in order to be able to make an interesting list of examples:

1. For functional programs, define new types, pattern matching, function call
2. For lemmas and specifications use ghost methods and requires/encures clauses, and reads clauses when a function depends on state.
3. Use forall quanttifiers, negation, implication.
4. For functional datatypes, pretty much everything can be encoded through lists. No need to introduce sequences yet, but of course sequences allow for more succinct notation.
5. For imperative programs, use variables, assignment, arrays, reads clauses, requires/encures clauses.
6. array<nat>, seq<nat> do not work. However perhaps this will be improved. At the moment I am using array<int> and seq<int>, and supply extra properties in **requires** clauses.
7. Use ghost variables to record intermediate state, use auxiliary lemmas in the proofs.
8. In imperative code use sequences to record the contents of an array upon entry to a method (eg old(a[..]).

**Style of writing Dafny code and assertions**

Use a restricted, but powerful subset of Dafny, and follow a certain style in code/specifications.

1. For functional programs, give their specification in ghost methods, rather than through ensures clauses – sometimes we need to break this, eg when we need the properties of a function to prove termination of another (might change in future versions).
2. Avoid writing specs which involve symbolic multiplication, as Z3 often cannot prove the relevant properties. Also, sometimes avoid writing terms involving addition on the natural numbers – similar reasons. Instead, you can define natural numbers recursively and define addition, and then Z3 will be able to find the proofs. Alternatively, if the exact operations do not matter for what the exercise is trying to achieve, you could define tour own datastructures and the operations on it.
3. Ghost method which state lemmas which involve universal quantification are expressed through parameters, rather than forall. This makes the lemma more re-usable. For example, we define

**ghost** **method** Lemma(xs: list<int>)

**ensures** reverse(reverse(xs))==xs;

rather than

**ghost** method Lemma( )

**ensures** **forall** xs :: reverse(reverse(xs))==xs;

1. The style we adopted allows us to apply the lemmas with concrete arguments, and thus better guide the proofs.
2. Use **calc** rather than call ghost methods, as this expresses what the intermediate result is, rather than how to get there.
3. Tension between most succinct vs more “readable” proof. I decided to show both. After all, mathematicians use different styles too.
4. Use **assert** statements, even when they are superfluous, for the same reason as those given above. Eg say

Lemma(zs);  
**assert** P(zs);

(In further work, if would be nice if Dafny supported something like:

**assert** P(xs) **by** Lemma(zs);

1. Note that the order of assertions matters. For example, when writing conjunctions which contain subparts which validity depend of each other the order matters,  
   For example, the following works:

**assume** 0 <= k < a.Length && a[k]>0;

while the following does not work:

**assume** a[k]>0 && 0 <= k < a.Length;

1. Avoid putting assertions under quantifiers when not necessary.  
   For example, the following assertion could be proven as an invariant in Pascal’s triangle:

**assume** **forall** i:: 0<i<max 🡺 a[i].Length =i;  
**assume** **forall** i:: **forall** j:: 0<i<max && 0<j< I 🡺 a[i][j]>0;

while the following does not work:

**assume** **forall** i:: **forall** j:: 0<i<max && 0<j< I 🡺 a[i].Length[i]=i && a[i][j]>0;

1. Quantify over tuples, rather than use nested quantifiers.  
   For example the following is usually more “provable”

**assume** **forall** I, j :: 0<=i<a.Length && 0<=j< i 🡺 a[i][j]>0;

than the following:

**assume** **forall** i:: **forall** j:: 0<=i<a.Length && 0<j< I 🡺 a[i][j]>0;

**Conclusions**

**How well does Dafny and the material developed support the Reasoning course?**

1. Write specs for functional code. *YES*
2. Detailed**, precise proofs** using induction (various); explicit proof steps, justification, and precision in application of lemmas. *Proof steps in Dafny are coarser than in had-written proofs. But on the other hand, Dafny supports writing of larger proofs, which require several inrermediate lemmas – see 3 below.*
3. “Architecture” of the proof – discover auxiliary lemmas. *Good, see 2 above, and also Verification Corner episode.*
4. Develop specs for imperative code, with pre and post conditions. Also defining and using appropriate predicates to specify and prove the code. *Good*
5. “Architecture” of the proof: invariants, assertions, specs of auxiliary code. Also, find the lemmas necessary to prove the code, and apply these lemmas. *Dafny more coarse-grained. Only when auxiliary lemmas need to be used..*
6. Detailed Hoare-logic proofs, using, essentially the SSA equivalent. *Very little. In some cases the annotated Dafny code checks without need for proof of any further properties, as opposed to what we teach in the course. But we have ebough exercises where the Dafny proff requires proving and using auxiliary lemmas*
7. Refinement: Develop code from spec: Form spec to invariant, and from invariant to code. *Excellent support through assume-clauses, and ghost methods without bodies.*
8. Notation (use, develop new). … *Not much there, and no possibility to develop*
9. Framing *– Dafny supports, but not in the course*
10. No concurrency. *– no Dafny support, and not in the course*

Overall, the Dafny specifications and proofs do not correspond one-to-one with the way they would be written in the course, but are close enough, and the correspondence is clear. The Dafny material complements the course material by allowing more-coarse grained proofs, supporting the discovery of the architecture of the proof. The fact that proofs are machine checked will greatly enhance students’ confidence in their work. I expect that this material will help us to teach our course more effectively.