A Beginner's Course on Reasoning About Imperative Programs

Kung-Kiu Lau

Department of Computer Science, University of Manchester
Manchester M13 9PL, United Kingdom
kung-kiu@cs.man.ac.uk

Abstract. Formal Methods teaching at undergraduate level has been going on at Manchester for a good number of years. We have introduced various courses based on different approaches. We have experienced the usual problems. To combat these problems, our approaches and our course contents have evolved accordingly over the years. In this paper we briefly trace this evolution, and describe the latest course, on reasoning about simple imperative programs, for first-year students who are half-way through our introductory programming course.

1 Introduction

Formal Methods teaching at undergraduate level has been going on at Manchester for a good number of years. We have run an introductory programming course which taught students how to read, and program from, VDM-like specifications. We have also had various courses on reasoning about programs which taught first-order logic, program specification and verification, and theorem proving techniques. On these courses, we have experienced the usual problems: the perception of Formal Methods as being too mathematical, too complex and completely impractical, coupled with an aversion to all things mathematical. To combat these problems, our approaches and our course contents have evolved accordingly over the years. In this paper we briefly trace this evolution, and describe the latest course, on reasoning about simple imperative programs, for first-year students who are half-way through our introductory programming course. We briefly trace its evolution, evaluate its success, and discuss the lessons learnt.

Apart from sharing our experience and the contents of our current course, we wish to convey the message that with suitable motivation and tool support, we believe it is possible to make the rudiments of Formal Methods accessible to beginner programmers. Moreover, it is highly desirable for such novices to be shown that testing alone is not sufficient for ensuring program correctness, and that Formal Methods are required for accomplishing the task properly.

2 Previous Courses

The current course has evolved from earlier courses on reasoning about programs. The latter courses started after a previous course on introductory programming using Pascal and VDM dropped out of the syllabus. In this section, we briefly describe all these courses.

C.N. Dean and R.T. Boute (Eds.): TFM 2004, LNCS 3294, pp. 1–16, 2004.
© Springer-Verlag Berlin Heidelberg 2004

2.1 Introductory Programming with Pascal and VDM

At first we had an introductory programming course for first-year students, where students were taught how to read specifications written in a notation similar to VDM [8], and how to write Pascal programs from these specifications. This was a compulsory course that ran for the full year. As an introductory programming course it had to teach beginner programmers the Pascal programming language from scratch. At the same time, the students were taught to read specifications written in a notation based on VDM. This notation was rather informal and not proper VDM at all. Even so, the students were not required to write any specifications, they only had to read specifications given to them. The course thus combined teaching Pascal programming with teaching the programming process of going from requirements, expressed as specifications, to program design and coding. To achieve this combination, special course material was developed that integrated the VDM-like notation into a programming process using Pascal. This material was published as a textbook [9] (shortly before the course was dropped from the syllabus! See below).

The course profile was as follows:

Lectures	72 lectures (3 a week)
Labs	10 lab sessions, 8 exercises
Tools	None
Exam	3 hours, answer 4 questions

This course was dropped eventually when the syllabus for the whole first-year in our department was revised, replacing Pascal with SML as the first programming language. Although various versions of the course appeared subsequently as optional (half-year or one-semester) units in the third-year and second-year syllabi, eventually they disappeared altogether.

2.2 Reasoning About Programs I

In the new first-year syllabus, we introduced an optional half-year course on reasoning about programs. This consisted of 2 parts: (i) program verification; (ii) logic. Part (i) covered the specification and verification of a simple imperative language (with assignments, sequential composition, conditional commands, and while loops) and a simple functional language. For imperative programs, the semantics were based on predicate transformers: program specifications were expressed as assertions that represented pre- and post-conditions, and loop invariants were expressed by suitably defined assertions.

The emphasis was on proving correctness. For imperative programs, proof rules were given for the various constructs, and for functional programs, proof by induction was taught. There were no labs, however, so all verification exercises were done on paper only.

Part (ii) was not a continuation of Part (i). Instead it was a separate section that covered the basic concepts of first-order logic: formulae and their truth tables, normal forms, logical consequence, equivalence, validity, etc. The teaching was supported by labs based on Tarski's World (see Section 4.1) for first-order logic.

The course profile was as follows:

Lectures	24 lectures (2 a week)
Labs	5 lab sessions, 4 exercises in logic, none in program verification
Tools	Tarski's World for logic
	No tool for program verification
Exam	2 hours, answer 3 questions

2.3 Reasoning About Programs II

With the switch from Pascal to SML as the first programming language, it was felt that the imperative language part should be taken out of the Reasoning about Programs course. This duly happened, and to replace this part, theorem proving was introduced. However, the course profile remained unchanged.

At first the theorem proving technique that was taught was natural deduction. Later, this was changed to the tableau method. There were no labs for theorem proving, however, and all verification exercises remained pen-and-paper only.

2.4 Reasoning About Programs III

When the whole world went Java-mad, our first-year syllabus was revised again, adopting Java as the introductory programming language. Consequently, the Reasoning about Programs course had to re-focus on imperative languages. However, it was felt that the material on functional programs, in particular proof by induction, was generally useful, and so it was retained. First-order logic was also retained, but theorem proving was dropped, to make way for a more substantial section on imperative programs, based om material from [5]. So the course contents became:

1. First-order Logic.
2. Reasoning about Functional Programs.
3. Reasoning about Imperative Programs.

The course profile was as follows:

Lectures	22 lectures (2 a week)
Labs	4 lab sessions, 4 exercises in first-order logic, none in reasoning about functional programs, none in reasoning about imperative programs
Tools	Tarski's World for first-order logic
	No tool for reasoning about functional programs
	No tool for reasoning about imperative programs
Exam	2 hours, answer 3 questions

3 The Current Course: Reasoning About Programs IV

Now we describe the current course on Reasoning about Programs. In this section, we trace its evolution from its predecessor, state its aims and objectives, and outline the

course contents. In subsequent sections, we will elaborate on the material for different parts of the course.

In the previous versions of the course, the lack of tools for reasoning about imperative programs had always been felt to be unsatisfactory, especially from the students' point of view. Finding a suitable tool that would fit into the scope of the Reasoning about Programs course was no easy task, either. However, recently we came across SPARK [2] and its tools, and decided to adopt it for the course.

This meant that there would be no room for functional programs, but in fact this suited us very well. With Java firmly ensconced by now as the introductory programming language, the material on functional programs looked increasingly irrelevant, so we welcomed the opportunity to lose it.

With some additional material on SPARK, the course contents are now as follows:

1. First-order Logic.
2. Reasoning about Imperative Programs (Part1: Principles).
3. Reasoning about Imperative Programs (Part 2: Practice).

The course profile is now as follows:

Lectures	14 lectures (2 a week)
Labs	4 lab sessions, 2 exercises in first-order logic, 2 in reasoning about imperative programs
Tools	Tarski's World for first-order logic SPARK tools for reasoning about imperative programs
Exam	$1\frac{1}{2}$ hours, answer 2 questions
Assessment	Exam 70%, labs 30%

Now that we have tool support for the entire course, we have made the course more lab-based than before. So labs now contribute 30% of the assessment, and the exam's contribution has been reduced from 100% (the norm) to 70%. Accordingly, the exam in now shorter (by $\frac{1}{2}$ hour), and the number of lectures has also been reduced by 30%. This development is in line with the move in our department to re-appraise the traditional approach of using lectures as the only means of course material delivery.

3.1 Aims and Objectives

The aim of the course is stated in the published course description as follows:

> The aim of this module is to introduce students to the principles and practice of reasoning about simple imperative programs. The principles will be based on first-order logic, and the practice will be provided by the imperative language SPARK and its tools.

The objectives are stated as the following learning outcomes:

1. Students should have a basic knowledge of first-order logic and should be able to understand sentences in first-order logic.
2. Students should be able to write sentences in first-order logic and translate them to and from English sentences.

3. Students should be able to understand first-order logic specification of simple properties of imperative programs.
4. Students should be able to reason informally about simple imperative program properties specified in first-order logic.
5. Students should be able to reason about SPARK programs.
6. Students should be able to use Tarski's World in the lab to create universes and reason about them in first-order logic.
7. Students should be able to use SPARK tools in the lab to write SPARK programs and reason about them.

3.2 Course Contents

The published syllabus is the following:

- Introduction
  - Introductory Example
  - Proving versus Testing
  - Introduction to First-order Logic
- Reasoning about Imperative Programs (Part 1: Principles)
  - Pre-post-condition Specifications
  - Commands as Predicate Transformers
    - Assignments
    - Sequential Composition
    - Conditional Commands
    - Iterative Commands
  - Weakest Pre-Conditions
- Reasoning about Imperative Programs (Part 2: Practice)
  - SPARK: An Introduction
  - The SPARK Tools
  - Path Functions
  - Verification Conditions
  - Using SPARK Tools in the Lab

In the introductory part, the students are introduced to first-order logic. They learn (in the lab) how to write and reason about first-order logic sentences. This knowledge will enable them to write pre-post-condition specifications later.

For reasoning about programs, there are two parts: principles and practice. In the principles part, the course uses a simple made-up imperative language (with assignments, sequential composition, conditional commands, and iterative commands) as a running example to introduce and illustrate predicate transformer semantics for imperative languages. The students learn the principles of how to reason about the correctness of given programs with pre-post-condition specifications.

In the practice part, the course introduces SPARK and its tools, and shows how the principles of predicate transformer semantics can be applied in practice to reasoning about SPARK programs. The SPARK tools provide automated help for correctness verification, so the students not only learn how to write SPARK programs (with pre-post-condition specifications) but also how to use the tools (in the lab) to prove the correctness of these programs.

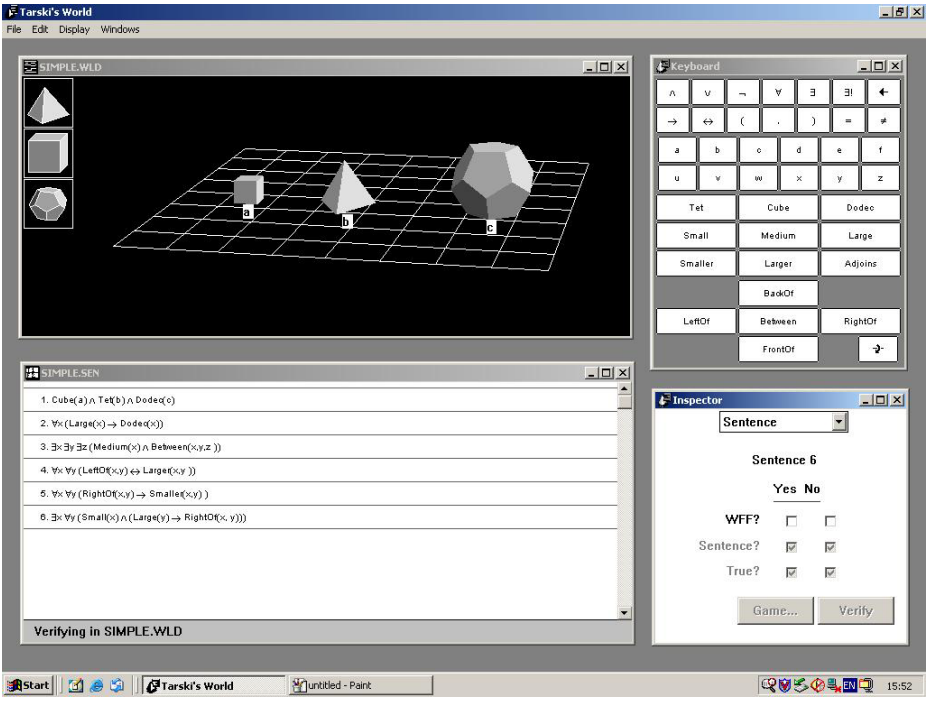


Fig. 1. Tarski's World: the GUI

4 First-Order Logic

The first part of the course gives an introduction to first-order logic. The students are introduced to Tarski's World right from the start, to learn the basics of first-oder logic. In this section, we describe Tarski's World and how our students use it to learn first-order logic.

4.1 Tarski's World

Tarski's World is an award-winning package for teaching beginners first-order logic. The package consists of a text book [3] and a graphical tool, and is supported by a web site [14]. The graphical tool, in particular, provides a very accessible way of learning the basics of first-order logic. Our students rely almost solely on this tool to learn how to write and reason about simple first-order logic sentences.

The GUI of Tarski's World is shown in Fig. 1. It consists of four sub-windows:

1. the World window (top left);
2. the Keyboard (top right);
3. the Sentence window (bottom left);
4. the Inspector (bottom right).

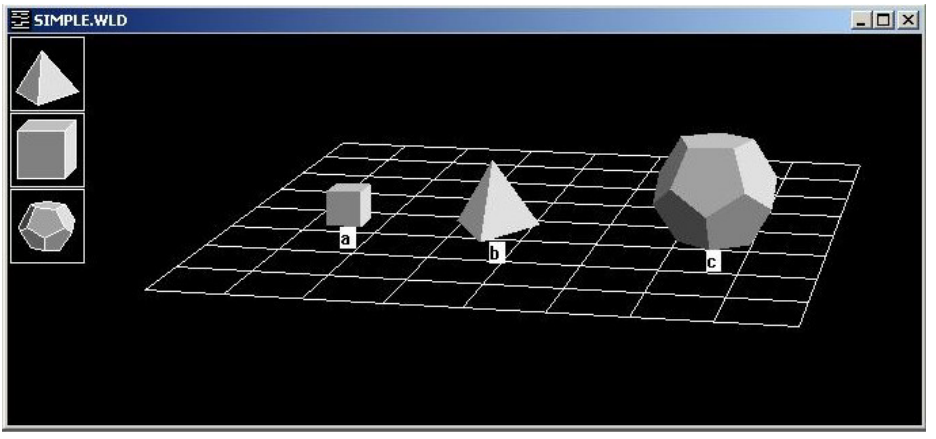


Fig. 2. Tarski's World: the World window

The World window, displays a *world* of objects called *blocks* that are simple geometric shapes, viz. tetrahedrons (tet), cubes and dodecahedron (dodec). Blocks can be large, medium or small in size, and can be labelled. A *world* consists of a 2-D grid (of fixed size) with a set of blocks placed on it. The World window allows a world to be constructed interactively, saved to a file, or reloaded from a saved file. For example, the SIMPLE world (in the file SIMPLE.WLD) in Fig. 2 consists of a small cube a, a medium tetrahedron b and a large dodecahedron c, placed in a line on the grid, as shown.

The Keyboard, Fig. 3, provides keys for logical symbols. These are laid out as follows, going from top to bottom: quantifiers and connectives, constant symbols (a,b,c, d,e,f), variable symbols (u,v,w,x,y,z), and predicate symbols (Tet, Cube, Dodec, …, BackOf, …FrontOf).[1] The Keyboard is used for constructing sentences in the Sentence window.

The Sentence window shows first-order logic sentences about the blocks in the chosen world, i.e. the world displayed in the World window. It allows sentences to be constructed interactively (using the keys provided by the Keyboard), saved to a file, and reloaded from a saved file. For example, Fig. 4 shows an example set of sentences (in the file SIMPLE.SEN) for the SIMPLE world. These sentences are:

1. $\mathsf{Cube} \wedge \mathsf{Tet(b)} \wedge \mathsf{Dodec(c)}$
2. $\forall\, \mathsf{x}\ (\mathsf{Large(x)} \rightarrow \mathsf{Dodec(x)})$
3. $\exists\, \mathsf{x}\, \exists\, \mathsf{y}\, \exists\, \mathsf{z}\ (\mathsf{Medium(x)} \wedge \mathsf{Between(x,y,z)})$
4. $\forall\, \mathsf{x}\, \forall\, \mathsf{y}\ (\mathsf{LeftOf(x,y)} \leftrightarrow \mathsf{Large(x,y)})$
5. $\forall\, \mathsf{x}\, \forall\, \mathsf{y}\ (\mathsf{RightOf(x,y)} \rightarrow \mathsf{Smaller(x,y)})$
6. $\exists\, \mathsf{x}\, \forall\, \mathsf{y}\ (\mathsf{Small(x)} \wedge (\mathsf{Large(y)} \rightarrow \mathsf{RightOf(x,y)}))$

The Inspector is used for inspecting the world in the World window and the associated sentences in the Sentence window. Using the Inspector, it is possible to add a label to a block, or change the label, and to change the size of a block. The Inspector can also be used to check if a formula being constructed in the Sentence window is well-formed

[1] There are no function symbols in Tarski's World.

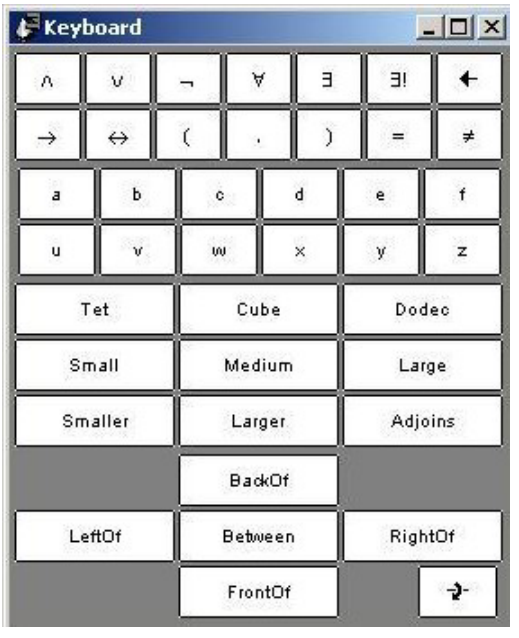


Fig. 3. Tarski's World: Keyboard

and therefore is indeed a sentence. The main purpose of the Inspector, however, is to verify the truth or falsehood of sentences in the Sentence world. For a chosen sentence, Tarski's World performs the verification automatically, and displays the resulting truth value against the sentence in the Sentence window. For example, in Fig. 5, part (b) shows the result of using the Inspector to verify sentence 6 in SIMPLE.SEN. Part (a) shows the truth values of all the sentences in the Sentence window which have been verified. It is obvious that sentences 1 to 3 are true and sentences 4 to 6 are false, so the Sentence window in Part (a) shows the following:

T 1. $\text{Cube} \wedge \text{Tet(b)} \wedge \text{Dodec(c)}$
T 2. $\forall\, x\, (\text{Large(x)} \rightarrow \text{Dodec(x)})$
T 3. $\exists\, x\, \exists\, y\, \exists\, z\, (\text{Medium(x)} \wedge \text{Between(x,y,z)})$
F 4. $\forall\, x\, \forall\, y\, (\text{LeftOf(x,y)} \leftrightarrow \text{Large(x,y)})$
F 5. $\forall\, x\, \forall\, y\, (\text{RightOf(x,y)} \rightarrow \text{Smaller(x,y)})$
F 6. $\exists\, x\, \forall\, y\, (\text{Small(x)} \wedge (\text{Large(y)} \rightarrow \text{RightOf(x,y)}))$

Lab Exercises. In the lab, the students do exercises in writing first-order logic sentences and reasoning about them in worlds, i.e. writing and verifying sentences in worlds that they have been given or have created themselves. These exercises can be classified into the following categories:

1. write well-formed formulas that are correct translations of given English sentences, with no reference to any world;

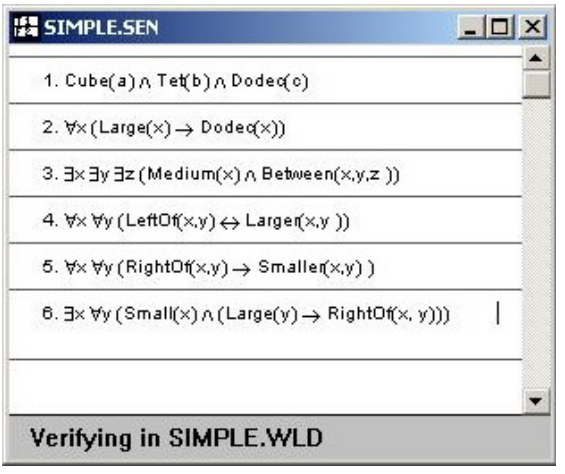


Fig. 4. Tarski's World: the Sentence window

2. write well-formed formulas that are correct translations of given English sentences, and verify them in a given world;
3. create a world such that a given set of sentences (in English or first-order logic) has specified truth values.
4. edit an existing world such that a given set of sentences (in English or first-order logic) has specified truth values.

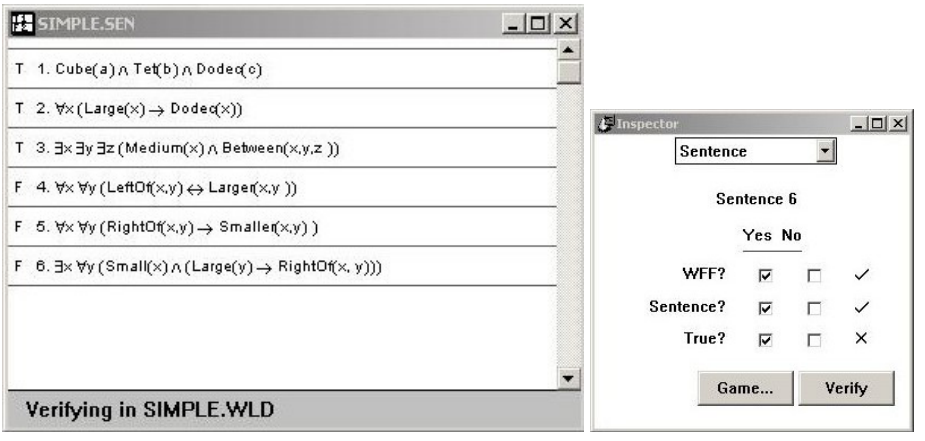
5 Reasoning About Imperative Programs

Reasoning about imperative programs is split into two parts: principles and practice. The first part gives an introduction to predicate transformer semantics of imperative commands and programs, and hence their specifications by pre- and post-conditions. In particular weakest pre-conditions are explained and used. The second part uses SPARK and its tools as an example of doing reasoning on real imperative programs. In this section, we outline these two parts.

5.1 Principles: Predicate Transformer Semantics

The material for the principles of predicate transformer semantics is based on the standard text [5]. Pre-post-condition specifications, and programs as predicate transformers are first explained. Then a simple but representative imperative language is used as a running example. The language contains the following constructs:

- Assignment
- Sequential Composition
- Conditional Command
- Iterative Command.



(a) (b)

Fig. 5. Tarski's World: the Inspector

Each of these commands is defined as a predicate transformer. For iterative commands, loop invariants are defined and explained. Weakest pre-conditions ar then introduced, and the semantics of the commands of the simple language are re-stated in terms of weakest preconditions.

All this is of course well-known, standard material, so we will not elaborate on it any further here.

5.2 Practice: SPARK and Its Tools

To put the principles of reasoning about imperative programs using predicate transformer semantics into practice, we then introduce a real imperative language SPARK and its tools. SPARK has a textbook [2] and a website [12].

SPARK is a subset of Ada, but on our course, we use a simple subset of SPARK which our students can pick up straightaway, since it is really no different from a similar subset of Java. What our students have to learn is the set of SPARK tools used for proving correctness, and the principles behind them.

SPARK: The Language. In SPARK a basic program unit is a *package*, which could be a *procedure* or a *function*. A package has a *specification* part and an *implementation* part, the package *body*. SPARK has various *annotations* (denoted by `--#`). In particular, these include *pre-conditions* (`--# pre`) and *post-conditions* (`--# post`) that can be put in the specification part of a package, and *assertions* (`--# assert`) that can be inserted anywhere in the body of a package. For instance, *loop invariants* in the body are represented by suitable assertions.

Example 1. In the package Div, the procedure Divide divides an integer X by another integer Y, yielding a quotient Q and a remainder R.

```
 1  package Div
 2  is
 3  procedure Divide(X, Y : in Integer;
 4                   Q, R : out Integer);
 5     --# derives Q, R from X,Y;
 6     --# pre (X >= 0) and (Y > 0);
 7     --# post (X = Q * Y + R) and (R < Y) and (R >= 0);
 8  end Div;
 9  package body Div
10  is
11  procedure Divide(X, Y : in Integer; Q, R : out Integer)
12     is
13     begin
14        R := X;
15        Q := 0;
16        loop
17           --# assert (X = Q * Y + R) and (R >= 0);
18           exit when R < Y;
19           R := R - Y;
20           Q := Q + 1;
21        end loop;
22     end Divide;
23  end Div;
```

The SPARK Examiner. SPARK has a tool, the SPARK Examiner, for generating *path functions* and their corresponding *verification conditions*.[2] For path analysis, a program is cut into sections without loops and sections that are just loops. For every path found, its pre- and post-conditions are generated from the annotations in the program. For example, the pre-condition of the first path is just the pre-condition of the program, and the post-condition of the last path is the post-condition of the program. A loop is regarded as one path, with its loop invariant as both its pre- and post-conditions.

For each path, with pre-condition p and post-condition q, the SPARK Examiner generates a verification condition. This is obtained by first generating the weakest pre-condition w for q, by hoisting q backwards through the commands in the path to the beginning of the path. The verification condition for this path is then $p \rightarrow w$. The Examiner outputs this in the form *Hypothesis* $\rightarrow$ *Conclusion*, or $H \rightarrow C$.

For example, for Example 1, the Examiner will generate the following paths and their verification conditions:

```
procedure Div.Divide

For path(s) from start to assertion of line 17:

procedure_divide_1.
H1:    x >= 0 .
H2:    y > 0 .
```

[2] The Examiner first performs syntax checks and flow analysis, but these are not of interest here.

```
         ->
C1:      x = 0 * y + x .
C2:      x >= 0 .

For path(s) from assertion of line 17 to assertion of line 17:

procedure_divide_2.
H1:      x = q * y + r .
H2:      r >= 0 .
H3:      not (r < y) .
         ->
C1:      x = (q + 1) * y + (r - y) .
C2:      r - y >= 0 .

For path(s) from assertion of line 17 to finish:

procedure_divide_3.
H1:      x = q * y + r .
H2:      r >= 0 .
H3:      r < y .
         ->
C1:      x = q * y + r .
C2:      r < y .
C3:      r >= 0 .
```

The SPARK Simplifier. SPARK also has a tool, the SPARK Simplifier, for simplifying or reducing verification conditions. For example, for Example 1, the Simplifier will reduce all verification conditions to true, thus (trivially) completing the correctness proof completely automatically!

```
procedure Div.Divide

For path(s) from start to assertion of line 17:

procedure_divide_1.
*** true .          /* all conclusions proved */

For path(s) from assertion of line 17 to assertion of line 17:

procedure_divide_2.
*** true .          /* all conclusions proved */

For path(s) from assertion of line 17 to finish:

procedure_divide_3.
*** true .          /* all conclusions proved */
```

In general, of course the result of applying the Simplifier to a set of verification conditions is just another set of verification conditions, albeit reduced or simplified in complexity. This set of verification conditions has then to be verified using a theorem

prover. SPARK also provides such a tool. Our students do not have to do any theorem proving, so they do not use this tool (see below).

Lab Exercises. In the lab, our students have to run the SPARK Examiner and Simplifier on given programs, and then write and prove their own program using these tools. The program they have to write is described in English in the lab manual, so the students have to write annotations (pre- and post-conditions) that faithfully capture this specification. The program also requires a loop invariant, so the students have to define this and insert it as an assertion in a suitable place in the loop.

The students are not required to do any theorem proving, however. To ensure this, the exercises are chosen so that they are simple enough for the Simplifier to be able to reduce all verification conditions to true completely automatically, as in Example 1.

To emphasise the importance of proving correctness, as opposed to testing, no compiler for SPARK is provided in the lab. So the students cannot resort to running the program to show correctness by testing. Of course, this does not stop them from writing, running and testing Java programs which they believe to be equivalent to the SPARK program, and then translating the former, after testing, to the latter. However, they still have to prove the SPARK program correct.

6 Motivation

On the previous courses, at the beginning of the course, it had been hard to explain the motivation for the course material. This was principally due to the difficulty of showing the relevance of Formal Methods to beginner programmers, which is what our students are. This is compounded by the fact that we did not use a real programming language.

So on the current course, apart from using a real programming language, we provide motivation along the following lines:

Safety-critical Applications. It is hard to motivate the use of Formal Methods for general applications, so on this course, we motivate the use of Formal Methods for safety-critical applications only. Further motivation is provided by citing well-known examples of expensive and disastrous software failures in safety-critical applications, e.g. Ariane 5.

Hardware Verification. We point out that verification has been accepted as necessary by the hardware industry and is done routinely nowadays. This suggests that sooner or later software verification will (have to) follow suit.

Proving versus Testing. We make the point that to show program correctness, testing alone is not complete, and it is necessary to use proving. Of course at this stage, our students only know about testing, and they use it in the labs for all courses to demonstrate that their programs work correctly. So to be told that correctness needs to be proved does make an impact.

The Praxis Prize. So far, the best motivation we have come up with (we hope) is an annual prize for the best SPARK program written and proved by the students in the lab. This is kindly donated by Praxis Critical Systems. At the time of writing, this prize is so new that it has not even been awarded, so it is too early to assess its impact just yet.

7 Evaluation

The current format of the course came into being in 2003. The course is an optional course in the second semester of our first-year programme. Take-up is good, with around 130 students.

On the whole, the current course has been a qualified success. Both tools, Tarski's World and the SPARK tools, are well received, and students have no problems doing the lab work using them. Tarski's World has a nice GUI, and is accessible even to the maths phobic students. SPARK tools have a less fancy user interface, but do not look strange to the students because they have similar 'look and feel' to the Java development environment that they use. Both Tarski's World and the SPARK tools are also stable and reliable, and cause no problems in the running of the labs.

The SPARK language is simple enough not to need any teaching at all, so the students are not put off by having to learn another language just for this course. The principles behind the SPARK tools do need teaching, but they are not too difficult to learn. Undoubtedly, using the tools in the lab really helps the students to understand this material.

Reducing the number of lectures, and making the course more lab-based have also worked well. Students are not forced to sit through an unnecessary number of 'dense' lectures on stuff that they will not apply in the lab.

The effect of the prize is not known at the time of writing, but I expect in the long term it will be a big plus.

On the down side, the course does not teach a whole Formal Method, even though it may be debatable whether it is advisable to do so. Students only learn to write simple logic specifications for simple SPARK programs, and to prove the programs correct using the SPARK tools. For beginners, this may be the right level to aim for, but it barely scratches the surface of proper Formal Methods like VDM [8], Z [13] or B [1].

8 Lessons Learnt

The lessons we have learnt are probably the obvious ones:

1. Students like tools.
2. Students like lab work.
3. Students do not like theory.
4. Integration with other, in particular core, courses, e.g. introductory programming or software engineering courses, is important.
5. Relevance to the real world is important.

These lessons are probably universal, since we believe our students are no different from students elsewhere. However, it should be pointed out that the lessons are only impressions based purely on observation and subjective feelings, rather than any objective study or analysis.

In order to motivate students to learn Formal Methods, demonstrating relevance to the real world is probably the biggest challenge, and integration with core courses in a crowded study programme is at best problematic (see the Conclusion for our own experience). Computer science students' dislike for maths or theory in general is a

hurdle that seems increasingly insurmountable, with most computer science degrees not including maths in their admissions requirements.

To make a Formal Methods course accessible, clearly it is good to make the course lab-based. In general, lectures are no longer the best way, and definitely should not be the only way, to deliver course material. For less popular topics, such as Formal Methods, lecture-only courses would be a non-starter. Lab-based courses with good tools would be a much more sensible alternative with a better chance of success.

9 Conclusion

We have described a course that teaches beginner programmers the rudiments of Formal Methods. We have experienced the usual problems faced by Formal Methods teaching, but we believe our course represents a reasonable effort to overcome these problems, chiefly by making the course more lab-based and by using good tools in the lab to support learning.

Integration with core courses remains a problem for us. Our course, which runs in the first year, is not linked to, or followed up, by any of our core courses in the second year. Our course is run by the Formal Methods group in the department. In the second year, our Software Engineering group teaches Z [13] as part of their software engineering course, which is a core course. Unfortunately, however, from the Formal Methods point of view, Z specifications are not used in the development process. The Unified Software Development Process [6] with UML [11] is used instead. So the material of our Reasoning about Programs course is not used by this, or indeed any other, course.

It would be reasonable to expect our course to be better integrated with the core introductory programming course if we used Java instead of SPARK. However, in our view, this is not feasible since formal reasoning about object-oriented programs in general is the subject of on-going research, and is far more complex than formal reasoning about non-object-oriented imperative programs. In particular, Java reasoning tools, e.g. JML [7] and ESC Java [4], are not yet mature enough. Even if they were, we suspect that for novices in both programming and Formal Methods, they would be too difficult to master.

With Formal Methods perceived as a fringe rather than a core subject, it remains hard to get an opportunity to teach a whole Formal Method, especially in an undergraduate computer science programme that has hardly enough room for competing core topics. Overcoming this fringe status is, we feel, the biggest stumbling block facing Formal Methods teaching.

Finally, should you be interested in finding out more about our Reasoning about Programs course, the material for this course can be found on the web page [10].

Acknowledgements. I am indebted to my colleagues Peter Aczel and John Latham at Manchester for providing information about the previous courses. I would also like to publicly thank Praxis Critical Systems Limited for their generosity in donating a prize for this course.

References

1. J.R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.
3. J. Barwise and J. Etchemendy. *The Language of First Order Logic*. CSLI, 3rd edition, 1993.
4. Extended Static Checking for Java Home Page. `http://research.compaq.com/SRC/esc/`.
5. D. Gries. *The Science of Programming*. Springer-Verlag, 1981.
6. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
7. The Java Modeling Language (JML) Home Page. `http://www.cs.iastate.edu/~leavens/JML.html`.
8. C.B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, second edition, 1990.
9. J.T. Latham, V.J. Bush, and I.D. Cottam. *The Programming Process: An Introduction using VDM and Pascal*. Addison-Wesley, 1990.
10. K.-K. Lau. CS1112: Reasoning about Programs. `http://www.cs.man.ac.uk/~kung-kiu/cs1112/`.
11. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
12. SPARKAda, Praxis Critical Systems Limited. `http://www.praxis-cs.co.uk/sparkada/`.
13. J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.
14. Tarski's World. `http://wwwcsli.stanford.edu/hp/Tarski1.html`.