

Using SPARK for a Beginner’s Course on Reasoning about Imperative Programs

Kung-Kiu Lau
School of Computer Science, The University of Manchester
Oxford Road, Manchester M13 9PL, United Kingdom
kung-kiu@cs.man.ac.uk

ABSTRACT

Teaching beginners predicate transformer semantics for imperative languages is not a trivial task. For Computer Science majors, the teaching of the theoretical material must be supported by suitable practical course work. For this, we need a suitable language with appropriate tool support. In this paper, we describe our experience of using SPARK and its tools for this purpose. Our experience has been a very positive one.

Categories and Subject Descriptors

K.3.2 [Computers and Education]: Computer and Information Science Education—*Computer Science Education*

General Terms

Experimentation

Keywords

Imperative Programming, Predicate Transformer Semantics

1. INTRODUCTION

Formal reasoning about imperative programs is not an easy subject to teach, even for languages that are not object-oriented. Teaching it to beginners is certainly a challenge. The theoretical foundations are well established (e.g. [3]), but it would be unsatisfactory to teach Computer Science majors just the theory. There must be practical course work that gives the students the opportunity to put the theory into practice, and thereby helping and consolidating their understanding of the basic principles. However, such practical work is hampered by the lack of suitable tools. For a beginner’s course, such tools must be not only stable (tried and tested) but also simple and not too daunting to use (see e.g. [4, 2]).

In this paper, we describe how SPARK and its tools have been used successfully for a beginner’s course on reasoning about imperative programs. The course is given to first-year students in the second semester, who have just completed a beginner’s course in Java programming in the first semester. On the course, SPARK is used to illustrate predicate transformer semantics [3] for impera-

tive languages, and SPARK tools are used by the students to do practical work.

2. COURSE CONTENTS

The course consists of two main parts, outlined as follows:

- Part 1: Principles
 - Assertions
 - Pre/post-condition Specifications
 - Commands as Predicate Transformers
 - * Assignments
 - * Sequential Composition
 - * Conditional Commands
 - * Iterative Commands
 - Weakest Pre-Conditions
- Part 2: Practice
 - SPARK: An Introduction
 - The SPARK Tools
 - Path Functions
 - Verification Conditions
 - Using SPARK Tools in the Lab

Part 1 presents the basic principles of predicate transformer semantics, using material based on [3]. It gives an introduction to predicate transformer semantics of imperative commands and programs, and hence their specifications by pre- and post-conditions, including weakest pre-conditions. A simple made-up imperative language from [3] is used as a running example to illustrate the predicate transformer semantics of assignment, sequential composition, the conditional command, and the iterative command. Each of these commands is defined as a predicate transformer. For iterative commands, loop invariants are defined and explained. Weakest pre-conditions are then introduced, and the semantics of the commands of the simple language are re-stated in terms of weakest pre-conditions.

The students also learn the principles of how to reason about the correctness of programs with given pre/post-condition specifications.

Part 2 gives the students a chance to put the principles into practice. It shows how the principles of predicate transformer semantics can be applied in practice to reasoning about imperative programs in a real language, namely SPARK, using material based on [1].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGAda’07, November 4–9, 2007, Fairfax, Virginia, USA.

Copyright 2007 ACM 978-1-59593-876-3/07/0011 ...\$5.00.

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.

A more detailed account of the course can be found in [4].

3. SPARK AND ITS TOOLS

The simple made-up language used in Part 1 is an artificial language, with no compiler. So it can only be used for defining and explaining predicate transformer semantics on paper. The students could not use it to do real programming. For this purpose, we introduce SPARK and its tools, which provide not only a means to gain practical experience of reasoning about programs in a real language, but also a clear illustration of predicate transformer semantics for a real language.

3.1 Pre/Post-Conditions

In Part 1 of the course, pre/post-conditions are defined as assertions, and the specification of a program consists of a pair of pre- and post-conditions. The meaning of such a specification is that the specified program should be a predicate transformer that transforms the assertion that is the pre-condition into the assertion that is the post-condition. In Part 2 of the course, SPARK illustrates this clearly.

In SPARK, a basic program unit, called a *package*, can be specified by a pair of pre- and post-conditions. A package could be a procedure or a function. It 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. This is illustrated by the specification of the following package (Div) containing a procedure (Divide):

```
package Div
is
procedure Divide(X, Y : in Integer;
                Q, R : out Integer);
--# derives Q, R from X, Y;
--# pre (X>=0) and (Y>0);
--# post (X=Q*Y+R) and (R<Y) and (R>=0);
end Div;
```

The specification of the Divide procedure is given by its pre/post-conditions. The procedure divides a nonnegative integer X ($X \geq 0$ in the pre-condition) by a positive integer Y ($Y > 0$ in the pre-condition) and returns the resulting quotient Q and remainder R ($(X=Q*Y+R)$ and $(R<Y)$ and $(R \geq 0)$ in the post-condition).

This example shows the use of pre/post conditions for specifying SPARK programs as predicate transformers.

3.2 Assertions

In Part 1 of the course, assertions are just any predicates that assert some logical properties that hold at some program points (pre/post conditions are particular examples that assert properties at input and output points). In Part 2 of the course, assertions are illustrated by SPARK. The `--# assert` annotations in SPARK are just assertions.

In the body of a SPARK package, assertions which are not the pre/post conditions of a procedure can be inserted anywhere. For example, in the following body of the Div package, the assertion $(X=Q*Y+R)$ and $(R \geq 0)$ is used as a loop invariant in the Divide procedure:

```
package body Div
is
procedure Divide(X, Y : in Integer;
                Q, R : out Integer)
is
```

```
begin
  R := X;
  Q := 0;
  loop
    --# assert (X=Q*Y+R) and (R>=0);
    exit when R < Y;
    R := R - Y;
    Q := Q + 1;
  end loop;
end Divide;
end Div;
```

As a loop invariant, this assertion is supposed to hold at this program point for every iteration of the loop.

3.3 Commands as Predicate Transformers

In Part 1 of the course, the predicate transformer semantics of the made-up language is defined in terms of the predicate transformer semantics of its commands. This is illustrated in Part 2 of the course by SPARK and its tools.

Although the syntax of SPARK is not identical to that of the made-up language, the predicate transformer semantics of its commands is basically the same as the corresponding commands of the former. To give an idea of the predicate transformer semantics of SPARK, here (for simplicity) we give the semantics of the commands of the made-up language.

The semantics of the assignment command is defined by:

$$\{R_e^x\} x := e \{R\}$$

where e is an expression, R is a predicate, and R_e^x is the predicate obtained by substituting e for all free occurrences of x in R ; or alternatively by

$$wp(x := e, R) = R_e^x \quad (1)$$

where wp denotes the weakest pre-condition.

The semantics for sequential composition is defined by:

$$\{Q\} C_1; C_2 \{R\} \leftrightarrow \{Q\} C_1 \{S\} \wedge \{S\} C_2 \{R\}$$

where C_1 and C_2 are commands, and Q, S, R are predicates; or using weakest pre-conditions,

$$wp(C_1; C_2, R) = wp(C_1, wp(C_2, R)) \quad (2)$$

The semantics of the conditional command is defined by:

$$\{Q\} \text{if } B \text{ then } C_1 \text{ else } C_2 \{R\} \leftrightarrow B \rightarrow \{Q\} C_1 \{R\} \wedge \neg B \rightarrow \{Q\} C_2 \{R\}$$

where B is a boolean condition, and Q and R are predicates; or using weakest pre-conditions,

$$wp(\text{if } B \text{ then } C_1 \text{ else } C_2, R) = B \rightarrow wp(C_1, R) \wedge \neg B \rightarrow wp(C_2, R) \quad (3)$$

The semantics for the iterative command expressed as a *while* loop is defined by:¹

$$wp(\text{while } B \text{ do } C, R) = \exists k, k \geq 0 (H_k(R)) \quad (4)$$

where B is a Boolean condition, C is a composite command (e.g. sequential composition), and R is a predicate,

$$H_k(R) = H_0(R) \vee wp(\text{if } B \text{ then } C, H_{k-1}(R)) \quad (5)$$

¹There is more than one form of the iterative command in SPARK. See the body of the Div package in the previous section for an example. Here for simplicity we consider just the generic form defined in the made-up language.

and

$$H_0(R) = \neg B \wedge R$$

$H_k(0)$ is the predicate that ‘the loop terminates in k or fewer iterations, with R true; and $H_0(R)$ is the predicate that ‘the loop terminates in 0 iterations’ (because the loop condition B is initially false), with R true. Note that the weakest pre-condition of the iterative command is defined (in (4)) in terms of the weakest pre-condition of a conditional command (in (5)).

SPARK tools illustrate this predicate transformer semantics, when they are used to analyse and reason about SPARK programs. The SPARK Examiner is used to analyse programs and generate verification conditions, while the SPARK Simplifier is used to simplify verification conditions, and possibly thereby proving them.

For a given procedure (with pre-condition Q and post-condition R), the SPARK Examiner automatically identifies all possible execution paths, as well as their pre/post-conditions. Any path with traversal condition T that starts from the beginning of the procedure has pre-condition $Q \wedge T$, and any path that terminates at the end of procedure has the same post-condition R as the procedure. Paths that start and/or end in the middle of the procedure need to have assertions to act as their pre- and/or post-conditions.

For example, in the Div package body, there are three paths in the Divide procedure: (i) Path 1: from the start of the procedure to the `--# assert` annotation; (ii) Path 2: from the `--# assert` annotation to itself; (iii) Path 3: from the `--# assert` annotation to the end of the procedure.

The pre-condition of Path 1 is the same as the pre-condition of the Divide procedure since its traversal condition is *true*; its post-condition is the assertion in the `--# assert` annotation. The pre-condition of Path 2 is the assertion in the `--# assert` annotation; and so is its post-condition. Path 2 is a loop. The pre-condition of Path 3 is the assertion in the `--# assert` annotation; its post-condition is that of the Divide procedure.

The predicate transformer semantics of SPARK commands is evident in the way verification conditions are generated for each path, using weakest pre-conditions, as we will now show.

3.4 Weakest Pre-Conditions

In Part 1 of the course, weakest pre-conditions are used to define the predicate transformer semantics of commands. In Part 2 of the course, SPARK illustrates this since its commands are also defined in terms of weakest pre-conditions, as we have seen in the previous section. In addition, the *hoisting* process performed by the SPARK Examiner illustrates the meaning and use of weakest pre-conditions. Hoisting is the process of moving a post-condition backwards through a command. During the process, the predicate that is the post-condition gets ‘backward’ transformed by the operation carried out by the command, and the resulting predicate is the weakest pre-condition.

For the assignment command

$$x := e$$

hoisting a post-condition R backwards through $x := e$ transforms R ‘backwards’ into R_e^x , which is the weakest pre-condition for $x := e$, as in (1).

For the sequential composition

$$C_1; C_2$$

hoisting a post-condition R backwards through C_2 gives the weakest pre-condition for C_2 , and hoisting this backwards through C_1 gives the weakest pre-condition for the composite $C_1; C_2$, as in (2).

For the conditional command

$$\text{if } B \text{ then } C_1 \text{ else } C_2$$

hoisting is done for two paths, with traversal conditions B and $\neg B$ respectively. Hoisting a post-condition R backwards through C_1 and C_2 gives the weakest pre-conditions for the two paths respectively, as in (3).

For the iterative command,

$$\text{while } B \text{ do } C$$

hoisting is done for two paths. This is because SPARK treats an iterative command as a conditional command, as suggested by (5).

The Examiner uses hoisting to work out the verification condition of a given path. For a path with pre-condition p and post-condition q , the SPARK Examiner first generates 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 the three paths in the Div package, the Examiner will generate the following verification conditions:

Path 1:

```
H1:    x >= 0 .
H2:    y > 0 .
      ->
C1:    x = 0 * y + x .
C2:    x >= 0 .
```

Path 2:

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

Path 3:

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

Thus whilst in Part 1 of the course, the predicate transformer semantics of a language is defined in the abstract, in Part 2 of the course, the working out of the verification conditions in SPARK shows clearly the predicate transformer semantics of a real language in action, during the hoisting process. It also shows clearly how weakest pre-conditions are derived and used for reasoning about real programs.

Using SPARK to teach predicate transformer semantics is a huge improvement on using only a made-up language with no tool support. The tool support provides a hands-on opportunity for students to be actively engaged in the learning experience. It is well-known that students learn better from this kind of experience.

3.5 The SPARK Examiner

Using the SPARK Examiner gives the students practical experience of writing SPARK programs with annotations. For the students, having to write annotations (in addition to code) is a new experience, and they like the novelty. Initially most students find

it quite difficult to come up with the correct annotations, in particular loop invariants. However, the automatic tool support makes the task much less daunting, since the students could easily try out alternative annotations and check their effects by running the Examiner. The students have to understand how verification conditions are generated, but they can rely on the Examiner to do the generation automatically. The files that are output by the Examiner are clearly formatted, making it easy for the students to understand their contents.

Using the Examiner also gives the students practical experience of seeing predicate transformer semantics in action for a real language. This is very beneficial for their understanding of predicate transformer semantics (as we saw in the previous section), and for convincing them of the practical value of the semantics for real languages.

3.6 The SPARK Simplifier

To prove the verification conditions generated by the Examiner, the students rely on the SPARK Simplifier, which simplifies or reduces verification conditions. The students do not have to understand how the Simplifier works, but they do have to understand the result of running the Simplifier on the verification conditions generated by the Examiner. To make this possible, we chose suitably simple examples where the Simplifier can reduce all verification conditions to true, and thereby discharging or proving them automatically.

For example, for the Div package, the Simplifier will reduce all verification conditions to true, thus (trivially) completing the correctness proof completely automatically, as shown by the following output:

```
procedure Div.Divide

For path(s) from start to assertion:

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

For path(s) from assertion to assertion:

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

For path(s) from assertion 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.

Even though they rely on the Simplifier for all the proofs, the students like the fact that with SPARK they can claim that they have proved their programs correct once and for all, which they can never do with testing, or with languages without annotations and proof support. Programming in SPARK thus gives them the edge.

3.7 Laboratory Exercises

In the laboratory, 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 laboratory 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 the Div package.

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.

As an extra incentive, a prize is offered to the best SPARK program written by students in the laboratory. This prize is kindly donated by Praxis High-Integrity Systems Ltd. It is only awarded if there is a deserving winner. So far, it has been awarded every year.

4. CONCLUSION

Teaching predicate transformer semantics to beginners is a non-trivial challenge. Teaching the semantics only in the abstract will not be satisfactory. Practical work is required in order to illustrate the semantics and to see it in action for a real programming language. We believe SPARK is a suitable language for this purpose. As we have demonstrated in this paper, for our course, SPARK illustrates all the aspects of predicate transformer semantics that we introduce in the abstract in the first part of the course. SPARK tools support practical work in reasoning with predicate transformer semantics.

Compared to more popular object-oriented languages like Java, SPARK is not only simpler, but it also has tools that are more mature. The tools are also more suitable for our course because they are relatively simple and easy to master. Our course used to be taught purely in the abstract, but it has been much more successful with the students ever since we adopted SPARK.

Acknowledgements

I would like to publicly thank Praxis High-Integrity Systems Limited for their generosity in donating a prize for this course.

5. REFERENCES

- [1] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.
- [2] I. Dony and B. Le Charlier. A tool for helping teach a programming method. In *ITICSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, pages 212–216. ACM Press, 2006.
- [3] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.
- [4] K.-K. Lau. A beginner's course on reasoning about imperative programs. In C. Dean and R. Boute, editors, *Proceedings of Symposium on Teaching Formal Methods 2004, Lecture Notes in Computer Science 3294*, pages 1–16. Springer-Verlag, 2004.