CS-Note* 3

Bill J Ellis
September 20, 2001

Diving Into The Spark Approach

1 Introduction

The Spark Approach is a method for producing software that improves confidence that a given program meets its specification. This confidence is gained by using a non-ambiguous programming language and subjecting the produced code to formal analysis.

There is a great deal of user manuals describing the many aspects of the Spark Approach. There is even a published book presenting the essence of the user manuals in an accessible form [1]. However, for what is such a practical method, it was felt that the best way to learn about the Spark Approach was to try it out first hand, referring to the documentation as problems were encountered.

Thus this is an account of the initial problems and observations encountered while diving into the Spark Approach for the first time.

2 Installing Spark

Before anything can be done with the Spark tools, the binaries must first be installed. A copy of the software tools and user manuals have been supplied on CD for Windows NT and Solaris. The intention is to install the Windows NT version on a Windows 98 computer. Although installing Windows software is usually a straight forward operation, a few problems were still encountered.

2.1 Path Names in Windows 98

At the end of the installation the user is prompted to add a list of new paths to the PATH environment variable. Only after some experimentation was it discovered that the use of long file names and in particular names with spaces (C:\praxis\spade core) were causing the new PATH line to flounder. Reverting to the DOS 8.3 name format (C:\praxis\spadec~1) fixed this.

2.2 File is Not Path

Probably fixated with the path problem, the environment variable LM_LICENSE_FILE was set to the path where the licence file should be located, rather than the file itself [1].

2.3 Licence

After the software is installed the documentation directs the use of a program called the Installation Verification Procedure (ivp.exe). This program uses the Spark EXAMINER to evaluate a number of program files, many of which have intentional errors in them. The resulting output is then compared character for character with the correct output generated on a perfect installation.

On executing ivp.exe the program reported the rather disheartening message:

```
*****************************************************************************
* SPARK Examiner Release 4.0
* Licence has expired.
* SPARK Examiner will not run.
*****************************************************************************
```

Setting the system clock back to any day before the licence was expected to start (31-AUG-2001) was sufficient to get the program to run. Interestingly, the program will not function at years less than 1995, as the Ada95 checker conducts a sanity check on the date. On executing ivp.exe now it completes successfully, reporting the message:

*Notes in this series are for ϵ-baked ideas, for $1 \geq \epsilon \geq 0$. Only exceptionally should they be cited or distributed outwith the CLAM-SPARK project (EPSRC GR/R24081).

[1]This was soon fixed, with a soft uttering of 'doh..'.

```
Checking the Examiner output files
SPARK Examiner is correctly installed
```

3 Crash Course in the Spark Approach

The Spark Approach is supported by a programming language and tools which analyse programs written in this language. To ensure the broader appeal of the Spark Approach, the various aspects of Spark can be considered as individual packages.

3.1 Spark Programming Language

The Spark programming language is made up of two different kinds of commands. There are the regular commands that make up the code and there are annotation commands that express the specification of the code.

The regular commands available in Spark are a carefully selected subset of Ada. By operating in this subset, Spark is able to remove various aspects of Ada that make formal analysis more difficult, if not impossible [2]. As Spark is a true subset of Ada it can be compiled using existing tried and tested Ada compilers.

Although Spark is a sufficient language for many tasks, there may be some special features required that Spark does not cater for. In these circumstances a special `hide` directive can be used to gain access to full Ada. The existence of this feature adds support to the claim that the Spark Approach is applicable to a wide rage of programs.

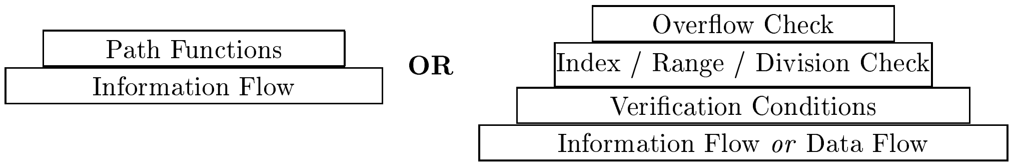
The annotation commands are used to specify the correct behaviour of Spark programs in a concise form. There are two different categories of these commands. The first are called the core annotations and cater for flow analysis and visibility control. The second are called the proof annotations and cater for formal proof.

The regular commands of Spark plus the core annotations are seen as the essential parts of Spark. The proof annotations and tools that operate with these are optional features that may be drawn on as required.

3.2 Spark Tools

The Spark Approach is supported by three separate programs. These may interact with each other by communicating with common files. This separation gives the flexibility to adopt different levels of analysis for different parts of their programs.

- **Spark** EXAMINER **-** This examines programs written in Spark, employing different analysis techniques depending on user configuration. These techniques can be illustrated as stacked below, showing how the choice of flow analysis determines the additional analysis available, and how adding more features will always entail doing those below.

  Some of these analysis techniques will require proof annotations to be present, allowing the creation of conjectures called Verification Conditions (VCs). Proving these VCs will offer stronger evidence that the program meets its specification.

- **Spark** SIMPLIFIER **-** The rigorous nature under which VCs are generated usually means that several will be produced even for a small program. Fortunately, many of these will be extremely trivial to solve. The SIMPLIFIER seeks to automatically prove these trivial VCs, allowing the user to focus on the more difficult ones.

- SPADE **Proof Checker -** Where a VC can not be proved automatically by the SIMPLIFIER it must be solved by the user. This might be successfully conducted using pen and paper, however this process is prone to error. Thus SPADE aims to provide the user with an environment where they can interact with a proof attempt in a strictly formal manner. In this way it is guaranteed that each step in a proof is logically consistent and hence the final answer correct.

[2] In Ada the compiler can choose whether to pass array parameters by value or reference. This can be used to write ambiguous code.

3.3 Presenting Examples to the EXAMINER

Spark does not specify the use of a particular Ada compiler. This leads to an interesting problem where the different conventions of Ada compilers can affect the practical use of Spark. The compiler being used on these examples is GNAT [3]. Most significantly, it strongly encourages that package specification and body files are separated. Fortunately, the EXAMINER accommodates this by accepting a list of each file on the command line. Where this becomes impractical for large projects a special meta file can be created.

3.3.1 Fixing the Listing Files

As the EXAMINER analyses each source file it creates a listing file, documenting any problems encountered. As default these listing files are named after each source file, replacing the source file extension with `lst`. Unfortunately, GNAT requires that package specification and body files share the same file name, using the different extensions `ads` and `adb` respectively. Thus in default mode the EXAMINER overwrites the listing file for package specifications with the listing file for the corresponding body.

In reaction to this problem the EXAMINER provides the special directive `listing_extension=VALUE`. This directive will use `VALUE` as the extension for listing files. Thus as default `listing_extension=lst`. Instead of only using regular characters, the special wild-card _ can be used to refer to the corresponding character in the extension of the file being analysed. Thus `listing_extension=ls_` will create listing files with extensions `lss` for package specification files and `lsb` for body files.

3.3.2 In Practice

Analyse the program unit exchange and generate VCs in Windows NT :
`spark /v /listing_extension=ls_ exchange.ads,exchange.adb`

Analyse the program unit trisort, that makes use of program unit exchange, and generate VCs in Windows NT:
`spark /v /listing_extension=ls_ exchange.ads,exchange.adb,trisort.ads,trisort.adb`

Do the same as above, but for Solaris:
`spark -v -listing_extension=ls_ exchange.ads exchange.adb trisort.ads trisort.adb`

4 Examples

4.1 Example 1 - exchangeInteger

To begin using the system a small existing example was sought. A suitable example was found in [1] and modified slightly as a package before being presented to the EXAMINER. The procedure takes two integer parameters and swaps their values.

```
package exchange is
    procedure exchangeInteger(X,Y: in out Integer);
    --#derives X from Y &
    --#        Y from X;
    --#post X=Y~ and Y=X~;
end exchange;
```

Figure 1: exchange package specification

The specification of the `exchange` package (Figure 1) is shown above. The comments starting `--#` are used to introduce the annotations for the procedure. As `--` marks an Ada comment the Ada compilers will ignore the added syntax.

The `derives` command is part of the core annotations, specifying the dependencies between variables. By conducting flow analysis on the code the actual dependencies can be determined and automatically compared to this specification. Here it shows that X is derived from Y and Y from X. Note that X is derived from Y does not mean that X equals Y, it only captures that the value of X is somehow dependent on the value of Y. For example, in X:=10*Y, X is still dependent on the value of Y.

The `post` command is part of the proof annotations, specifying the post-condition, what should be true if the procedure terminates [3]. The ~ is placed after a variable to mean the value when the procedure is called. Thus X=Y~ means that the value of X coming out should equal the value of Y that came in.

The body of the `exchange` package (Figure 11) implements the familiar method of swapping two variables using an intermediate temporary location. Perhaps not surprisingly, the VCs generated for this procedure were all proven automatically by the SIMPLIFIER.

4.1.1 Testing Derives

There are many cases where the variable Y can occur on the right hand side of an assignment without actually affecting the vale assigned. Consider the following assignments:

```
X:=0*Y;
X:=Y-Y;
X:=(Y div Y)-1;
```

In each case X will always be assigned zero, regardless of the value of Y. Replacing the assignment `X:=Y;` in `exchangeInteger` with these did not result in any errors from the flow analysis. However, using the assignment `X:=0;` brought many warnings detailing that X is not derived from Y, as had been specified in the `derives` command.

This behaviour was not unexpected, as the reasoning required to identify such special cases would be out-with the scope of the EXAMINER. Nevertheless, investigating these anomalies helps to get a better appreciation of what `derives` actually means to Spark.

4.2 Example 2 - trisortInteger

This procedure takes three integer variables (A,B,C) and returns them in ascending order. To experiment with packages this procedure will be implemented making use of the `exchange` package discussed above.

```
with exchange;
--#inherit exchange;

package trisort is
    procedure trisortInteger(A,B,C: in out Integer);
    --#derives A from A,B,C &
    --#        B from A,B,C &
    --#        C from A,B,C;
    --# post ((A=A~ and ((B=B~ and C=C~) or (B=C~ and C=B~)))  or
    --#       (A=B~ and ((B=A~ and C=C~) or (B=C~ and C=A~)))  or
    --#       (A=C~ and ((B=A~ and C=B~) or (B=B~ and C=A~)))) and
    --#       (A<=B and A<=C and B<=C);
end trisort;
```

Figure 2: trisort package specification

In the specification of the `trisort` package (Figure 2) the `inherit` command is used to specify that the annotation in the `exchange` package should be available when analysing the body of this package. The post-condition of `exchangeInteger` can now be queried to find the true impact of calling this procedure from within the `trisortInteger` procedure.

Unlike the first example this procedure has conditional statements. This makes the conceptual interpretation of `derives` more difficult, as different paths may be taken through the code and consequently variables may be derived from different variables. The solution is to specify every variable that that a variable might be derived from. To explore this, on one test of this procedure the derives line was changed to:

```
--#derives A from A,B &
--#        B from A,B,C &
--#        C from A,B,C;
```

[3]Note that termination of the procedure is not guaranteed. The Spark Approach does not provide a formal method to prove termination.

This produced the following warning, correctly extrapolating from the code that A *may* be derived from C:

```
??? Warning :601: The imported value of C may be used in the derivation of A.
```

The first three lines of the `post` command shows that the exit values of each variable must be some combination of the input values. The last line of the post statement specifies that the values of variables from A to C must be ascending.

The body of the `trisort` package (Figure 12) tests each variable against every other, exchanging values where necessary. This can be done a little more efficiently, but the straight forward solution was preferred. Note how the `exchangeInteger` procedure is preceded with its package name. This notation is mandatory in Spark, as the Ada `use` keyword [4] is not allowed. Instead, to use an external procedure Y in package X, the notation X.Y() must be used. This ensures greater clarity in which packages are being used and where.

As with the first example the SIMPLIFIER successfully proved all of the VCs. To determine if the SIMPLIFIER really was operating as expected, the example was repeated after introducing a rather major bug. An assignment statement `C:=3;` was inserted inside the second conditional statement. The resulting VCs could not be solved by the SIMPLIFIER, and on inspection inside SPADE it appeared that the VCs correctly reflected this change:

```
H1:    a <= integer__last .
H2:    c >= integer__first .
H3:    a > b .
H4:    b > c .
H5:    a > b .
       ->
C1:    3 = a or (3 = b or 3 = c) .
C2:    3 <= b .
C3:    3 <= a .
```

4.2.1 Caution is Due in Fully Capturing Specifications

The complex expression for the post-condition was not expected. It evolved from a quite simple expression as the true nature of the procedure was considered. Originally the post-condition had been based on the idea that each output value will equal one the input values. This inspired the post-condition:

```
--#post (A=A~ or A=B~ or A=C~) and
--#     (B=A~ or B=B~ or B=C~) and
--#     (C=A~ or C=B~ or C=C~) ...
```

However, what this fails to capture is that the output values are a combination of the input values. This idea spawned from the observation that the above would be true if the procedure simply set every output variable to the value of A. Thus the post-condition must exhaustively consider every valid combination of input variables, resulting in the large expression.

4.2.2 Avoid Any Assumptions

The last part of the derives statement specifying the ascending order of the output variables was originally specified as:

```
--#     (A<B and A<C and B<C);
```

While at first glance this might appear correct, it hides an assumption that A,B and C are all different values. If any two of these variables are the same, one can not be less than the other. Only by trying to prove the resulting VCs manually [5] did this error present itself.

4.3 Example 3 - multAsLoop

From experience, analysing straight through code is often much easier than where a loop is involved. Thus an example with iteration was sought. This function multiplies two variables, using repeated addition inside a loop.

[4] To open a package into the current name space, much like the `import package` command in Java or the `local open package in end` command in SML.

[5] They can not be proved at all - let alone by the SIMPLIFIER!

```
package mult is
    function multAsLoop(X,Y:integer) return Integer;
    --#pre X>=0;
    --#return X*Y;
end mult;
```

Figure 3: mult package specification

In the specification of the `mult` package (Figure 3) the `pre` command specifies the pre-conditions, what must be true when the function is entered. For implementation reasons, if the value of X is less than zero the function will not terminate. Thus the pre-conditions are that X is not less than zero.

As this is a function there is no post-condition. Instead a `return` command is used to specify the nature of the value that the function returns. This can be a complex expression, but in this case it is quite concise.

The body of the `mult` package (Figure 13) operates by adding the value of Y to an initially zero Result location X times. Note how instead of using `while`, a more general `loop` is used, specifying the exit point. This is encouraged in Spark as it more clearly demonstrates when the `assert` command will be encountered.

Key to generating the VCs for a loop is the `assert` command. This is used to specify what is true inside the loop, the loop invariant. This allows VCs to be generated for entering the loop, iterating round the loop and finally exiting the loop.

Once again the SIMPLIFIER successfully proved all of the resulting VCs.

4.3.1 Manual Proof

To get a feel for SPADE a manual proof was conducted on the VCs generated for this example. This first VC describes entering the loop. As there is no conditions to be met for entering the loop, this must be true:

```
H1:  x >= 0
H2:  x >= integer__first
H3:  x <= integer__last
H4:  y >= integer__first
H5:  y <= integer__last
-->
  C1:  true
```

SPADE can be asked to perform a limited analysis of a VC to determine its truth by issuing the command `done`. In this case such an action is sufficient to prove the VC [6]:

```
CHECK|: done.
```

The second VC describes an iteration around the loop. The hypothesis H1 is taken from the assertion on entering the loop. The hypothesis H2 is based on the notion that to make an iteration of the loop, the exit condition must be false. Finally the conclusion C1 is derived from the changes impacted on the assertion during the execution of the loop body:

```
H1:  result = count * y
H2:  not count = x
-->
  C1:  result + y = (count + 1) * y
```

This VC can be proved by first replacing the occurrence of `result` in C1 with `count * y`. Both sides of the resulting `count * y + y = (count + 1) * y` can then be put in a standard from using `standerise` which is then obviously true. The proof is concluded as above with `done`:

```
CHECK|: replace c#1: result by count * y using eq(1).
CHECK|: standerise c#1.
CHECK|: done.
```

[6]Whereupon SPADE congratulates the user with the kind message: `*** VC PROVED -- Well done!`

As an alternative to using the ambiguous command `standerise` the entire proof of the second VC could have been conduced using `replace` alone:

```
CHECK|: replace c#1: result by count * y using eq(1).
CHECK|: replace c#1: (count + 1) * y by y * count + y * 1 using distrib(2).
CHECK|: replace c#1: y * count by count * y using commut(2).
CHECK|: replace c#1: y * 1 by y using arith(1).
CHECK|: done.
```

The third and final VC describes exiting the loop. The hypothesis H1 is taken from the assertion on leaving the loop. The hypothesis H2 is based on the notion that to exit the loop, the exit condition must be true. Finally the conclusion C1 is derived from the post-condition, specified earlier as a Spark `return` statement:

```
H1:  result = count * y
H2:  count = x
-->
  C1:  result = x * y
```

The occurrence of `result` in H1 can be replaced by `x` to give `result = x * y` and then the occurrence of `result` in C1 can be replaced with `x * y`. The resulting expression is now trivially correct, and can be proved using `standerise` and finalised with `done`:

```
CHECK|: replace h#1: count by x using eq(1).
CHECK|: replace c#1: result by x * y using eq(1).
CHECK|: done.
```

4.3.2 Correct Does Not Imply Solvable

In C, it is customary to write the test condition in loops in terms of a less-than sign. This stems from wanting to repeat the value used to declare arrays in the test:

```
int myArray[10];
while(i<10){ myArray[i]=0; }
```

This same rule of practice was used in the implementation of the `multAsLoop` function [7]. This gave the test:

```
while (Count<X)
```

However, the code produced a VC that the SIMPLIFIER could not tackle. The problem lies in that where the loop terminates the strongest statement that can be said about `count` is that it must equal or be greater than X:

```
H1:    result = count * y .
H2:    not (count < x) .
        ->
C1:    result = x * y .
```

Here the hypothesis H2 is not strong enough to allow replacing `count` in H1 by `x`, and the VC can not be proved. Thus, although the code is correct, it was necessary to use an equality test in the loop to ensure the VCs could be proved. This is quite significant, as the need for provable VCs directly influenced the nature of the algorithm.

4.3.3 Spark Correct Does Not Imply Total Correct

Removing the precondition that X is not less than zero has no effect on being able to prove the VCs for the function inside the SIMPLIFIER.

As negative values of X prevent the function from terminating they do not affect the proof of VCs which are only concerned with showing that the function is partially correct. Thus although the pre-condition is essential to ensure that the function always terminates, this is not required to show the function correct following the Spark Approach.

[7] This was originally written using the Spark `while` command, before discovering the preferred `loop` syntax.

4.4 Example 4 - compareInteger

In this example a small function has been constructed to demonstrate path functions. It returns the appropriate attribute from an enumeration type to describe the relationship between two integers.

```
package compare is
    type ResultOfCompare is (NotPossible,LessThan,GreaterThan,EqualTo);
    function compareInteger(X,Y: Integer) return ResultOfCompare;
    --#return M => (X<Y -> M=LessThan) and
    --#             (X>Y -> M=GreaterThan) and
    --#             (X=Y -> M=EqualTo);
end compare;
```

Figure 4: compare package specification

In the specification of the `compare` package (Figure 5) a new type `ResultOfCompare` is introduced containing a set of possible values. As this is a function there is a `return` statement rather than a `post` statement. Note the flexible notation for specifying different return values based on the input values.

The body of the `compare` package (Figure 14) is implemented using nested `if` commands. Note how `Result` is initialised outside any conditional statement, to avoid EXAMINER warnings, even though this logically need not be present:

```
22  return Result;
                  ^
??? Warning :501: Expression contains reference(s) to variable Result, which
                  may be undefined.

23      end compareInteger;

??? Warning :602: The undefined initial value of Result may be used in the
                  derivation of the function value.
```

The EXAMINER successfully generated the path functions for this code. Unlike verification conditions the path functions are primarily intended for human inspection rather than some formal analysis. This is mainly because there is no formal specification of how the path functions should operate. Consider the following small extract:

```
...
    Path  3
      Traversal condition:
 1:    not (x = y) .
 2:    not (x < y) .
 3:    x > y .
      Action:
        compareinteger' := greaterthan &
        result' := greaterthan .

    Path  4
      Traversal condition:
 1:    not (x = y) .
 2:    not (x < y) .
 3:    not (x > y) .
 4:    x = y .
      Action:
        compareinteger' := notpossible &
        result' := notpossible .
...
```

Although not complicated, even for this very straight forward function, it takes a moment to determine that Path 3 is sound and Path 4 will never be taken. Fortunately, the SIMPLIFIER accepts these path functions and offers assistance in simplifying the traversal conditions. In this particular case it reduces the path functions sufficiently such that Path 3 is much easier to read and correctly identifies that Path 4 can not be traversed:

```
...
    Path  3
      Traversal condition:
    1:  x > y .
      Action:
        compareinteger' := greaterthan &
        result' := greaterthan .

    Path  4
          Path eliminated.  (Contradictory traversal condition)
...
```

4.4.1 Using Formal Proof

Using the same code the EXAMINER was used to generate VCs. These were all discharged successfully using the SIMPLIFIER, proving that the function is correct.

Note that this form of analysis did not highlight the section of dead code in Path 4. Certainly identifying and removing Path 4 would be a sensible and worthwhile task, but its relative importance is much smaller than code which breaks the system [8].

4.4.2 A Note of Caution

There are a number of disadvantages associated with the use of path functions [9]. Most significantly, path functions can only be created and inspected once the code has been written. This does not sit well with the Spark Approach of fully specifying the system before writing any code. Furthermore, the available alternative of generating VCs is generally much more attractive.

However, as shown above, there are circumstances where path function analysis may bring some additional insight into a program.

4.5 Example 5 - runtimeFunction

In this example a rather ridiculous function is used to experiment with the generation of VCs to prove the absence of runtime errors. The function takes an index and an array with know bounds. Where the supplied index is within the bounds, and the value at this index is not zero, the function returns the result of dividing 7718 by this value. In all other cases the function returns zero.

```
with types_runtime;
--#inherit types_runtime;

package runtime is
    function runtimeFunction(MyIndex: Integer;
                             MyArray: types_runtime.Vector) return Integer;
    --#return M => ((MyIndex >= types_runtime.Bottom and
    --#              MyIndex <= types_runtime.Top) ->
    --#                ((MyArray(MyIndex) /= 0 -> M=7718/MyArray(MyIndex)) and
    --#                 (MyArray(MyIndex) = 0 -> M=0))) and
    --#            ((MyIndex < types_runtime.Bottom or
    --#              MyIndex > types_runtime.Top) -> M=0);
end runtime;
```

Figure 5: runtime package specification

[8] It may however be the case that a system must meet certain response times. And executing such dead code could prevent this from happening.

[9] 3 of the 11 pages in the user manual for path functions outline their disadvantages!

The specification of the `runtime` package (Figure 5) is shown above. This refers to the `types_runtime` package (Figure 15) which holds the types and constant values for this function. The more complex form of `return` command is used to specify the function as described above. The body of the `runtime` package (Figure 16) closely mirrors the logic of this `return` command.

As this function will be subjected to a proof of the absence of runtime errors, it is important to know information specific to the target machine. In particular, at what point numbers will overflow. This information is supplied to the EXAMINER using a special target file (Figure 6).

```
integer'first = -32768
integer'last = 32767
```

Figure 6: runtime target file

The target file shows the upper and lower bounds of integers on the target platform. Currently this is the only information that can be provided by the target file, but this may grow in the future.

The SIMPLIFIER is able to solve all but two of the VCs generated. Thus these remaining VCs were tackled inside SPADE. Note that to describe arrays in VCs two special functions are used. `element(X,[Y])` returns the element in array X at index Y, and `update(X,[Y],Z)` returns the array X with the element at index Y replaced with the value Z.

The first VC is concerned with showing that the integer elements stored in the array are within the valid range.

```
H1:    for_all(i___1 : integer,element(myarray,[i___1]) >= - 32768 and element(
          myarray,[i___1]) <= 32767) .
H2:    myindex >= 51 .
H3:    myindex <= 773 .
       ->
C1:    element(myarray,[myindex]) >= - 32768 .
C2:    element(myarray,[myindex]) <= 32767 .
```

The key to tackling this VC is to make use of the universally quantified hypothesis H1. This is achieved by using `unwrap` to strip off the quantifier and then use `instantiate` to create a ground expression.

```
CHECK|: unwrap h#1.
CHECK|: instantiate int_I___1_1 with myindex.
```

This gives the new hypothesis H4:

```
H4:  element(myarray,[myindex]) >= - 32768 and element(myarray,[myindex]) <= 32767
```

It can be seen that this is a conjunction of the conclusions C1 and C2. By using `simplify` H4 will be replaced by two hypothesis representing its two conjuncts. Now there are hypothesis that match the conclusions, thus the VC is obviously true and can be finalised with `done`.

```
CHECK|: simplify.
CHECK|: done.
```

The second VC is concerned with showing that `7718 div X` always has a result, and that this result lies within the valid range for integers.

```
H1:    for_all(i___1 : integer,element(myarray,[i___1]) >= - 32768 and element(
          myarray,[i___1]) <= 32767) .
H2:    myindex >= 51 .
H3:    myindex <= 773 .
H4:    element(myarray,[myindex]) >= - 32768 .
H5:    element(myarray,[myindex]) <= 32767 .
H6:    element(myarray,[myindex]) <> 0 .
       ->
C1:    7718 div element(myarray,[myindex]) >= - 32768 .
C2:    7718 div element(myarray,[myindex]) <= 32767 .
```

As the denominator of the div expression can not be zero (H6), the division should always return a result. As div is integer division the value of the denominator will never be a fraction. From this it seems clear that the result will always lie in the range -7718 to 7718. However, despite a great deal of effort, a formal proof of this could not be found [10].

4.5.1 Mostly Harmless

An interesting little discovery was made while rummaging through the user manuals for references to integer division. Consider the program fragment:

```
Result:Integer;
Result:=Result div -1;
```

Assuming that the range of Integer on the target machine is specified as above, this can produce a run time error. The trick lies in the extreme value -32768. Dividing this by -1 will result in 32768, which is just outside the maximum integer value of 32767. This highlights how even the most harmless looking code can hide a possible source of error.

4.6 Example 6 - lift

An important feature of Spark is the use of proof functions. This trivial example explores proof functions by implementing a tiny part of a lift package, where the only operation available is going up one floor at a time.

```
with increment;
--#inherit increment;

package lift is
    procedure liftUp(Floor: in out Integer);
        --#derives Floor from Floor;
    --#post Floor=Floor~+1;
    end lift;
```

Figure 7: lift package specification

The `lift` package specification (Figure 7) is shown above. The body of this package (Figure 19) is implemented making use of an `increment` package (Figure 17 and 18). The important thing to note is that the return statement of the `liftUp` function is a call to the function `inc` in the `increment` package. This raises the question as to how the return value of the function `liftUp` should be used in describing the VCs for this function.

It may be possible to examine the `inc` function and use its post-condition. However, `inc` need not necessarily have a post-condition [11]. In these cases there will be an unacceptable gap in the generated VCs. Furthermore, even if `inc` does have a post-condition it may in turn refer to more functions. Continuing to expand these nested post-conditions is likely to result in large cryptic VCs.

It is clear from these observations that an alternative solution is required to deal with functions that are implemented in terms of other functions. The solution is to make use of proof functions. These provide a concise mechanism to fill the otherwise potential gap in generated VCs. Proof functions can be introduced by the user or automatically generated by the EXAMINER. In both cases a proof function is tied to a particular package.

A proof function is a full mathematical function. Such functions must make all arguments to the function explicit. Thus a full function with zero arguments must always return the same value. This differs from Spark functions as these may make use of global variables. A Spark function with zero arguments could return different values by accessing a global variable.

A proof function is automatically created for every Spark function. To ensure that these proof functions are full, any global variables referred to inside the Spark function are added as extra arguments to the proof function. These generated proof functions can then be used to overcome the problem seen above where functions are defined in terms of other functions. The single VC generated for the increment function is:

[10] Although there are many rules available to manipulate statements with `div`, there does not appear to be an actual definition of `div`. This is probably the sticking point.

[11] Different parts of Spark programs can be subjected to different levels of analysis. Thus, it may be the case that `inc` has no proof annotations.

```
H1:    floor >= integer__first .
H2:    floor <= integer__last .
       ->
C1:    increment__inc(floor) = floor + 1 .
```

Here it can be seen how the automatically generated proof function `increment__inc(N)` is used to avoid inspecting the details of the function `inc` in the package `increment`.

It might be expected that the EXAMINER would, where possible, automatically generate rules for these proof functions based on known post-conditions of the function. However, in all cases, the rules for manipulating proof functions must be provided by the user. This is achieved by creating a special user rule file. This declares a family of true statements describing the proof functions. For example the following rule file (Figure 8) might be used to define the proof function `increment__inc(N)` and hence allow solving of the VC.

```
rule_family inc: increment__inc(A) requires [A: any].

inc(1):  increment__inc(A) may_be_replaced_by
              A+1.
```

Figure 8: increment rule file

4.7 Example 7 - trisortarrayInteger

In this example a procedure using arrays in some manner is considered. The procedure implements the trisort function as seen before, except in this case the values (A,B,C) are indices (1,2,3) in an array.

```
with types_trisortarray,exchangearray;
--#inherit types_trisortarray,exchangearray;

package trisortarray is
    --# function ordered(A: types_trisortarray.Triple) return Boolean;
    --# function permutation(A, B: types_trisortarray.Triple) return Boolean;
    procedure trisortarrayInteger(MyTriple: in out types_trisortarray.Triple);
    --#derives MyTriple from MyTriple;
    --#post permutation(MyTriple,MyTriple~) and
    --#     ordered(MyTriple);
end trisortarray;
```

Figure 9: trisortarray package specification

The `trisortarray` package specification (Figure 9) is shown above. The `types_trisortarray` package (Figure 20) declares the three element array Triple used for this function. The `function` annotation commands introduce proof functions. These may then be used in specifying the proof annotations. Here the `post` command is specified much more concisely than in the original `trisort` package, by making use of proof functions.

Where proof functions are used in proof annotations they will also occur in the VCs for the procedure. Thus the following rule file (Figure 10) is used to provide the formal definition of these functions.

The body of the `trisortarray` package (Figure 23) is implemented in the same fashion as the first `trisort` package. However, the previous `exchange` package is no longer applicable as the items to swap are now indices in an array rather than variables. To this end the new package `exchangearray` was created (Figure 21 and 22). The VCs for this were all proved by the SIMPLIFIER.

None of the VCs for the `trisortarray` package can be solved by the SIMPLIFIER. This is a result of using proof functions, which the SIMPLIFIER can not reason about.

Aside from the fairly trivial first VC, all of the VCs have the same general form. It is conjectured that being able to solve the second VC would be sufficient knowledge to know how to solve the others, thus only this is considered:

```
H1:  for_all(i___1 : integer, element(mytriple, [i___1]) >= integer__first
        and element(mytriple, [i___1]) <= integer__last)
```

```
rule_family ord: ordered(A) requires [A: any].

ord(1):  ordered(A) may_be_replaced_by
                for_all(o: integer, 1 <= o and o < 3 ->
                        element(A, [o]) <= element(A, [o + 1])).

rule_family perm: permutation(X, Y) requires [X: any, Y: any].

perm(1):  permutation(update(update(A, [I], X), [J], Y), B)
                may_be_deduced_from
                        [ permutation(update(update(A, [J], X), [I], Y), B) ].

perm(2):  permutation(A, A) may_be_deduced.

perm(3):  permutation(A, B) may_be_replaced_by permutation(B, A).
```

Figure 10: trisortarray rule file

```
H2:  element(mytriple, [1]) <= element(mytriple, [2])
H3:  element(mytriple, [1]) > element(mytriple, [3])
H4:  element(mytriple, [2]) > element(mytriple, [1])
-->
  C1:  permutation(update(update(update(mytriple, [1], element(mytriple,
        [3])), [2], element(mytriple, [1])), [3], element(mytriple, [2])),
        mytriple)
  C2:  ordered(update(update(update(mytriple, [1], element(mytriple, [3])),
        [2], element(mytriple, [1])), [3], element(mytriple, [2])))
```

To begin, the rule file shown above is loaded into the system:

```
CHECK|: consult 'trisortarray.rul'.
```

The first conclusion C1 can be proved using the first property of `permutation` and the built-in rule for removing array updates that have no effect:

```
array(2):  update(A,I,element(A,I)) may_be_replaced_by A.
```

A possible proof script is listed below. Note how `infer` rather than `replace` must be used to make use of the `perm(1)` rule, as it is declared as an infer rule (...`may_be_deduced`...) rather than as a rewrite rule (...`may_be_replaced_by`...):

```
CHECK|: infer c#1 using perm(1).
CHECK|: replace c#1:
        update(   update(   update(   mytriple,
                                      [1],
                                      element(mytriple, [3])
                                  ),
                            [3],
                            element(mytriple, [1])
                        ),
                  [2],
                  element(mytriple, [2])
              )
        by
        update(   update(   mytriple,
                            [1],
                            element(mytriple, [3])
                        ),
```

```
                [3],
                element(mytriple, [1])
            )
        using array(2).
CHECK|: infer c#1 using perm(1).
CHECK|: infer c#1 using perm(2).
```

The second conclusion C2 is more problematic. Using the following proof fragment it can be fairly intuitively reduced to a single sticking conjecture:

```
CHECK|: replace c#2:
        ordered(   update(   update(   update(   mytriple,
                                                 [1],
                                                 element(mytriple, [3])
                                             ),
                                       [2],
                                       element(mytriple, [1])
                                   ),
                             [3],
                             element(mytriple, [2])
                         )
                )
        by
        for_all(o : integer, 1 <= o and o < 3 ->
                element(   update(   update(   update(   mytriple,
                                                         [1],
                                                         element(mytriple, [3])
                                                     ),
                                               [2],
                                               element(mytriple, [1])
                                           ),
                                     [3],
                                     element(mytriple, [2])
                                 ),
                           [o]
                       ) <=
                element(   update(   update(   update(   mytriple,
                                                         [1],
                                                         element(mytriple, [3])
                                                     ),
                                               [2],
                                               element(mytriple, [1])
                                           ),
                                     [3],
                                     element(mytriple, [2])),
                           [o + 1])
                )
          using ord(1).
CHECK|: unwrap c#2.
CHECK|: prove c#1 by implication.
CHECK|: prove c#1 by cases on (int_o_1=1 or int_o_1=2 or int_o_1=3).
CHECK|: done.
CHECK|: done.
```

At this point it must be shown that the case `int_o_1=3` will not actually occur. This is achieved through a proof by contradiction:

```
CHECK|: prove c#1 by contradiction.
CHECK|: infer not int_o_1 = 3 using inference(1).
CHECK|: deduce false from [9,11].
CHECK|: done.
```

This final proof only begins to hint at the complex kinds of theorem proving tasks that SPADE is used to tackle.

4.7.1 Matching Rules to Proof Functions

The relationship between Spark functions and proof functions is significant. Spark constrains the programmer to refer to a function Y() in package X, outside of X, using the notation X.Y(). However, accessing Y() within X does not require any dot notation [12].

Thus if Y() were a proof function it will be be assigned different names in the VCs depending on whether it is referenced inside X or outside X. This is significant as the user will need to refer to this proof function by name to declare rules. If Y() is used both inside and outside X then two different rule files will be required.

A possible solution is to always declare proof functions in the same package that uses them. This may not be practical if the same proof function is used in more than one package. Alternatively, all proof functions could be declared in some special package Z that never actually uses the proof functions. Thus all reference to the proof functions must be of the form Z.Y(). The minor downside of this latter solution is that the names of rules will need to contain the name of package Z.

5 Conclusion

The Spark Approach does not specify an exact method that must be pursued in writing Spark programs. Instead it prescribes rules of good practice and provides a flexible architecture within which these rules can be followed.

Of most interest to our project is SPADE. However, it is important to remember that SPADE is just one component within a larger system.

References

[1] J. G. P. Barnes. *High Integrity Ada The SPARK Approach.* Addison-Wesley, 1997.

[2] J. G. P. Barnes. *Programming in Ada 95.* Addison-Wesley, 1997.

[3] At Ada Core Technologies. In *http://www.gnat.com/.*

[12] In fact X.*anything* is illegal inside X.

A Appendix

```
package body exchange is
    procedure exchangeInteger(X,Y: in out Integer) is
        Temp: Integer;
    begin
        Temp:=X;
        X:=Y;
        Y:=Temp;
    end exchangeInteger;
end exchange;
```

Figure 11: exchange package body

```
package body trisort is
    procedure trisortInteger(A,B,C: in out Integer) is
    begin
        if A>B then
            exchange.exchangeInteger(A,B);
        end if;
        if A>C then
            exchange.exchangeInteger(A,C);
        end if;
        if B>C then
            exchange.exchangeInteger(B,C);
        end if;
    end trisortInteger;
end trisort;
```

Figure 12: trisort package body

```
package body mult is
    function multAsLoop(X,Y:Integer) return Integer is
        Count,Result:Integer;
    begin
        Count:=0;
        Result:=0;
        loop
        --#assert Result=Count*Y;
            exit when Count=X;
            Result:=Result+Y;
            Count:=Count+1;
        end loop;
        return Result;
    end multAsLoop;
end mult;
```

Figure 13: mult package body

```
package body compare is
    function compareInteger(X,Y: Integer) return ResultOfCompare is
        Result: ResultOfCompare;
    begin
        Result := NotPossible;
        if X=Y then
            Result := EqualTo;
        else
            if X<Y then
                Result := LessThan;
            else
                if X>Y then
                    Result := GreaterThan;
                else
                    --Already tried this check above, and it must have been
                    --false to have reached here.
                    if X=Y then
                        Result := NotPossible;
                    end if;
                end if;
            end if;
        end if;
        return Result;
    end compareInteger;
end compare;
```

Figure 14: compare package body

```
package types_runtime is
    Bottom: constant := 51;
    Top: constant := 773;
    type Vector is array (Integer) of Integer;
end types_runtime;
```

Figure 15: types_runtime package specification

```
package body runtime is
    function runtimeFunction(MyIndex: Integer;
                             MyArray: types_runtime.Vector) return Integer is
        Temp: Integer;
        Result: Integer;
    begin
        Temp:=0;
        Result:=0;
        if MyIndex>=types_runtime.Bottom then
            if MyIndex<=types_runtime.Top then
                Temp:=MyArray(MyIndex);
            end if;
        end if;
        if Temp /= 0 then
            Result:= 7718 / Temp;
        end if;
        return Result;
    end runtimeFunction;
end runtime;
```

Figure 16: runtime package body

```
package increment is
    function inc(Value: Integer) return Integer;
    --#return Value+1;
end increment;
```

Figure 17: increment package specification

```
package body increment is
    function inc(Value: Integer) return Integer is
    begin
        return Value+1;
    end inc;
end increment;
```

Figure 18: increment package body

```
package body lift is
    procedure liftUp(Floor: in out Integer) is
    begin
        Floor:=increment.inc(Floor);
    end liftUp;
end lift;
```

Figure 19: lift package body

```
package types_trisortarray is
    type TripleRange is range 1 .. 3;
    type Triple is array (TripleRange) of Integer;
end types_trisortarray;
```

Figure 20: types_trisortarray package specification

```
with types_trisortarray;
--#inherit types_trisortarray;

package exchangearray is
    procedure exchangeAtIndex(MyTriple: in out types_trisortarray.Triple;
                              IndexX,IndexY:in types_trisortarray.TripleRange);
    --#derives MyTriple from MyTriple,IndexX,IndexY;
    --#post MyTriple=MyTriple~[IndexX => MyTriple~(IndexY);
    --#                        IndexY => MyTriple~(IndexX)];
end exchangearray;
```

Figure 21: exchangearray package specification

```
package body exchangearray is
    procedure exchangeAtIndex(MyTriple: in out types_trisortarray.Triple;
                              IndexX,IndexY:types_trisortarray.TripleRange) is
        Temp: Integer;
    begin
        Temp:=MyTriple(IndexX);
        MyTriple(IndexX):=MyTriple(IndexY);
        MyTriple(IndexY):=Temp;
    end exchangeAtIndex;
end exchangearray;
```

Figure 22: exchangearray package body

```
package body trisortarray is
    procedure trisortarrayInteger(MyTriple: in out types_trisortarray.Triple) is
    begin
        if MyTriple(1)>MyTriple(2) then
            exchangearray.exchangeAtIndex(MyTriple,1,2);
        end if;
        if MyTriple(1)>MyTriple(3) then
            exchangearray.exchangeAtIndex(MyTriple,1,3);
        end if;
        if MyTriple(2)>MyTriple(3) then
            exchangearray.exchangeAtIndex(MyTriple,2,3);
        end if;
    end trisortarrayInteger;
end trisortarray;
```

Figure 23: trisortarray package body