FreePascal changes: user documentation

Jochem Berndsen

February 2007

Table of Contents

1 Introduction

Comments are used in programming languages to explain the programmer's intent to the programmer self and for other programmers. These are necessary to maintain the program, but they are typically ignored by the compiler of interpreter for the programming language.

There are programming languages in which it is possible to provide annotation that is parsed and used by the compiler for that language (for example, Eiffel). On the other hand, there are efforts to extend existing programming languages with possibilities to provide the compiler with information in formal annotation (for example, ESC/Java).

There exists two possible routes for transforming the formal annotation in actual results: static and dynamic checking. Static checking is what happens in ESC/Java, in that case the compiler generates a number of proof obligations, which can be checked by a theorem prover. If this theorem prover succeeds, the annotation is correct, and this is verified in compile-time. (ESC/Java is imperfect in this respect) Dynamic checking, on the other hand,

means that formal annotation is translated into run-time checks. In the Eiffel programming language, the amount of run-time checks generated can be controlled by switches.

We implemented a form of dynamic and static checking in the FreePascal 2.0 compiler in order to see how difficult such a task would be. This document describes how to use the changes made.

It is possible to insert assertions before, between or after any statement. These are compiled just like regular Assert()-statements. Furthermore, pre- and postconditions of functions, procedures and methods can be given. Class invariants are supported in a limited way. Before loops the programmer can insert an invariant and a variant function. Finally, there is support for propositions and definitions, that act as abbreviations.

Missing are the abilities to use quantified expressions like 'for all', inheritance of class invariants, and giving pre- and postconditions of procedures and functions in the unit header instead of the unit implementation.

2 Accepted syntax

Regular annotation in the Pascal programming language comes in three flavors:

1. between curly braces { this is a comment },
2. between parentheses and asterisks: (* this is a comment *)
3. from double slash to the end of the line // this is a comment

FreePascal by default allows nested comments.

If parsing of formal annotation is enabled, comments that start with {@ are treated as formal annotation. Annotation that starts with {@ inside another comment is ignored. (This makes it easy to 'comment out' formal annotation)

There are three extensions made to the regular Pascal grammar as implemented by FreePascal 2.0:

1. In the declaration part of a procedure, program, unit or function (where normally variables, constants, labels, ... are declared), formal annotation can be used.
2. Before or after any statements, formal annotation can be given.
3. In the definition of a class, formal annotation can be inserted to specify class invariants.

(Note: all grammars are in EBNF, Extended Backus-Naur Form)

Declarations

declarations :=

    (label_decs | const_decs | type_decs | var_decs | ... | formal_decs)*

formal_decs :=

    '{@' formal_dec (';' formal_decs)? '}'

formal_dec :=

('DEF' definitions) |

('SPECVAR' specvars) |

('PRE' '=' expression) |

('POST' '=' expression) |

('RET' '=' expression)

definitions :=

identifier (':' type)? = expression (',' definitions)?

specvars :=

identifier (':' type)? = expression (',' specvars)?

The declarations of a procedure, program, unit or function can now include definitions, specification variables ('specvars'), preconditions, postconditions and return values.

As the default is in Pascal, the literals such as 'DEF' are case insensitive.

Statements

What used to be a statement, is now called a 'pure_statement'. The new production rule for statements is:

statement :=

(assertion* pure_statement assertion*) |

((assertion | invariant | bound)* (while_statement | repeat_statement) assertion*)

assertion :=

'{@' expression? '}'

invariant :=

'{@' 'INV' : expression '}'

bound :=

'{@' 'BND' : expression '}'

There may be exactly zero or one invariant before each while or repeat statement, and exactly zero or one bound before each while or repeat statement.

Again, 'INV' and 'BND' are case insensitive.

Class invariants

In the declaration of a class, instead of a field or method, a class invariant may be given, with the following syntax:

'{@' ('INV' identifier = expression)? '}'.

3 Semantics

Definitions, specification variables

Definitions act a lot like macros, but they obey scoping and typing rules. In a definition, other definitions, specification variables, variables, constants, i.e. any valid expression may occur. Their main use is to abbreviate often-used expressions.

Example:

```
procedure definition_example(const x: Integer); // must be called with positive integer x
var y : Integer;
{@ def P = (x > 0) and (y < 3) }
begin
        y := 0
        {@ P and (y < 2) }
end;
```

Definitions may not be used outside formal annotation. The value of P is not evaluated at the start of the function, but only where P is used. Typing ((x > 0) and (y < 3)) instead of P in the above example has the same effect.

Specification variables exist to 'remember' old values of variables, for example the value of the parameters.

Example:

```
procedure square(var x: Integer);
{@ specvar oldx = x }
begin
        x := x * x;
end;
```

With specification variables, the value of the expression on the right hand side is evaluated and assigned to the specification variable on the left hand side before the function body is executed.

Definitions and specification variables can have any legal type. If no type is given, their type is inferred by the compiler.

Pre- and postconditions

Preconditions and postconditions specify the behavior of the function or procedure.

Specifying a precondition P for a function/procedure means that P should hold before the execution of the body, but after the evaluation of the specification variables.

Specifying a postcondition Q for a function/procedure means that Q should hold after the

execution of the body (even if execution of the body is terminated prematurely by means of the 'exit' statement).

Their type is always 'boolean', and hence the expression on the right hand side must be boolean.

Example:

```
procedure prepost_example(var x : Integer);
{@ pre = x > 0; post = x < 0 }
begin
        x := - x;
end;
```

If more than one precondition is given, the conjunction of the given precondition is used as precondition for the function/procedure. The same holds for the postcondition.

There can be a 'return value' given. This is only allowed with functions (not procedures, since they don't return a value). The type of expression on the the right hand side must be convertible to the result type of the function. {@ ret = E } is semantically the same as {@ post = (Result = E) }

Assertions

Before or after a statement, an expression between {@ and } means that this expression should evaluate to 'true' at that point in the execution of the program. The expression must be of type boolean.

Example:

```
procedure assertion_example;
begin
        x := 0;
        {@ x = 0 }
        x := 1;
end;
```

Loop invariants, bounds

Before a while or repeat loop, a loop invariant and/or variant function (bound function) can be specified. The loop invariant must be of type boolean, and the variant function must be of type integer.

The loop invariant should evaluate to true prior to each execution of the repetition body, and after the execution of the loop. The expression given as variant function should evaluate to a nonnegative integer prior to each execution of the repetition body, and should decrease after each execution of the repetition body, to ensure termination of the loop.

```
procedure loop_example(const N: Integer);
```

```
var i : Integer;
begin
        i := 0;
        {@ inv : i <= N } {@ bnd : N – i }
        while i < N do begin
                i := i + 1;
        end;
end;
```

Class invariants

The meaning of a class invariant C is that prior to each method invocation, after each method invocation, after the constructor terminates, and before the destructor is invoked, C should hold.

Example:

```
class TT =
        FNatural : Integer;
        {@ inv I0 : FNatural >= 0 }
end;
```

The methods, constructor and destructor can specify additional pre- and postconditions.

4 Invoking the FreePascal compiler

By default, formal annotation parsing is switched off. This is to maintain backwards compatibility with programs that have comments that start with {@. The FreePascal compiler itself has at least one place where a comment starts with {@.

To control the behavior of the compiler with respect to formal annotation parsing, two options are made available: compiler directives and command-line arguments.

The command-line argument -Sf switches formal annotation parsing on.

The compiler directive {$FORMAL+} switches formal annotation parsing on, {$FORMAL-} switches it off.

The directives take precedence.

(Please note that some formal annotation constructs make the compiler generate run-time assertion checks. These are not compiled into actual code if assert()-statement compilation is switched off.)

If the argument -Aproofobl is given on the command-line, then proof obligations for procedures and functions with pre- and postconditions are generated (if feasible).

5 Generated output

As explained in the introduction, there exists a notion of static checking, and of dynamic checking.

Dynamic checking

Dynamic checking of loop invariants, bound functions, assertions, pre- and postconditions and class invariants is enabled if assert()-statement compilation is switched on. If a check of any of these conditions failed, the same result happens as if an assert()-statement failed (i.e., a run-time error occurs and the program is terminated with an appropriate error message).

Static checking

The input for the static check is a partially annotated program. The output is, if feasible, a fully annotated program with verification conditions.

Then the following steps are done for each function, procedure and method:

1. A feasibility check is done. It is checked if a pre- and postcondition are given, and if all loops are accompanied with an invariant and a variant function.
2. The partially annotated program's annotation is completed: gaps are filled (only for assignment and selection) according to weakest precondition calculus.
3. All verification conditions are generated.

Supported language constructs

The following language constructs are currently supported:

- assignment,
- selection (if-statement),
- repetition (while-statement),
- sequential execution (;).

Missing are for example function calls, arrays and the possibility of aliasing is a problem. So it is only useful for very small programs such as the ones created in the beginning of an introductory programming course.

Completing annotation

Assignment

x := E { R }

is transformed into:

{ P } x := E { R }

where $P \equiv P_k \wedge P_{k+1}$, where $P_k, P_k + 1$ are introduced for readability, and $P_k \equiv def.E$, $P_{k+1} \equiv R(x := E)$

Selection

The selection comes in two different flavors: with and without an 'else' part.

Without an 'else' part:

if B then S

{ R }

transforms (recursively on S) into:

{ P }

if B then { Q } S { R }

where $P \equiv P_k \wedge P_{k+1} \wedge P_{k+2}$, where $P_k \equiv def.B$, $P_{k+1} \equiv B \rightarrow Q$ and $P_{k+2} \equiv \neg B \rightarrow R$. The predicate Q may be either synthesized or given.

With an 'else' part:

if B then S0 else S1

{ R }

transforms (recursively on S0 and S1) into:

{ P }

if B then { Q0 } S0 { R } else { Q1 } S1 { R }

where $P \equiv P_k \wedge P_{k+1} \wedge P_{k+2}$, where $P_k \equiv def.B$, $P_{k+1} \equiv B \rightarrow Q0$ and $P_{k+2} \equiv \neg B \rightarrow Q1$. The predicates Q0 and Q1 may be either synthesized or given.

If a precondition for the if-statement is given, like this:

{ P }

if B then S0 else S1

{ R }

then this statement could be transformed into:

{ P }

if B

then { P and B } { Q0 } S0 { R }

else { P and (not B) } { Q1 } S1 { R }

where Q0 and Q1 could be synthesized or given. This could be done in order to improve readability, but this is not done in the implementation. (A verification condition must be added in that case: [$P \rightarrow def.B$].)

Repetition

For the repetition no weakest precondition is calculated; only if an invariant and variant function are supplied, the process can continue.

{ inv: P } { bnd: F }

while B do S

{ R }

transforms, recursively on S, into:

{ P }

{ inv: P } { bnd: F }

while B do { P and B } { Q } S { P and (F < bound) }

{ P and (not B) }
{ R }

where bound is the value of F at the start of that particular execution of the repetition body.

Catenation

S ; T { R }

is transformed, recursively on T, into:

S ; { Q } T { R }

even if the sequence was annotated. For example:

S ; { Q } T { R }

transforms into:

S ; { Q } { Q' } T { R }

Generation of verification conditions

Empty statement

If two assertions are placed in a row, without a statement in between, the first assertion must imply the second.

This is a verification condition that must be checked.

{ P } { Q }

yields

{ P } { Q, Note X }

where Note X means $[P \rightarrow Q]$

While statement

{ inv: P } { bnd: F }

while B do S

{ R }

yields

{ inv: P } { bnd: F }

while { Note X } B do S

{ R }

where Note X means $[(P \rightarrow def.B) \wedge ((P \wedge B) \rightarrow (F \geq 0))]$

Format of output file

The output file is a LaTeX file. The file has three parts:

- a human-readable representation of the fully annotated program;
- the definitions used in the verification conditions and annotated program (in order to abbreviate);
- the verification conditions themselves.

(Note that the human-readable fully annotated program has some duplications, it seems that there are unnecessary assertions created by empty statements. This is because of the generated assert()-statements.)

Wherever in the fully annotated program a Note occurs, this refers to a verification condition.

In the definitions and verification conditions formatting and content has been separated. This means that a format.tex file must be used to control the formatting of the output. A default format.tex file is given.

6 Example

Problem definition

Let x be an integer constant, and r be an integer variable. The problem 'integer square root' is specified by:

$\{0 \leq x\}$

isqrt

$\{(0 \leq r) \wedge (r^2 \leq x < (r+1)^2)\}$

Naive solution

This program admits a naive solution:

```
{ 0 <= x }
r := 0;
{ inv: r*r <= x } { bnd: x – r }
while not (x<(r+1)*(r+1)) do begin
        r := r + 1
end;
```

Faster solution

A more intelligent solution is:

```
{ 0 <= x }
r := 0;
s := x + 1;
{ inv: (0 <= r) and (r < s) and (r*r <= x) and (x < s*s) } { bnd: s – r }
while r + 1 <> s do begin
        h := (r + s) div 2;
        if h * h <= x then begin
                r := h;
        end else begin
                s := h;
        end;
end;
```

Output

In an appendix, the output of compiling both programs is included.