Contract-based Programming: a Route to Finding Bugs Earlier

Jacob Sparre Andersen

JSA Research & Innovation

February 2018
Contract-based Programming

A software development technique, used to find programming errors earlier in the development process.

In its strictest form, the contracts are checked as a part of the compilation process, and only a program which can be proven to conform with the contracts will compile\(^1\).

In a less strict form, it is more similar to “preventive debugging”, where the contracts are inserted as run-time checks, which makes it more likely to identify errors during testing.

In this presentation I will focus on preventive debugging, i.e. how you can insert assertions efficiently in your source text.

\(^1\)This is what SPARK 2014 does. Stay for the following presentation, if you find it interesting.
You insert **assertions** in your source text to tell all readers of the source text – both humans, compilers and other tools – something concrete about the execution state of the software.

Notice that assertions are **different from comments**, since the compiler understands them, and can both check that they are correct, and use them for optimising the generated code.
It is unfortunately common to disable run-time checking of **unproven assertions** in production code, and only have the run-time checking enabled during testing.

In my view that is like bringing along the life-vests during testing of a ship, but removing them before going to sea for real, so...

**Don’t do that!**

If run-time checking of a specific assertion is too costly for the timing requirements of your application, **prove** that the assertion is correct. Once the assertion has been proven true, it is safe to disable checking of it at run-time.
The typical view of contract-based programming is that its core is pre- and post-conditions of subprograms (functions, procedures, etc.)

Here is an example:

```ada
procedure Increment (Counter : in out Natural;
                     Step     : in     Positive)
with Pre  => (Counter < Natural'Last) and
            (Step <= Natural'Last - Counter),
    Post   => (Counter > 0);
```
You could achieve the same run-time effect with plain old-fashioned assertions in the body of the subprogram:

```
procedure Increment (Counter : in out Integer;
                     Step    : in     Integer) is
begin
  pragma Assert (0 <= Counter);  -- Natural
  pragma Assert (1 <= Step);     -- Positive
  pragma Assert ((Counter < Integer'Last) and
                  (Step <= Natural'Last - Counter));

  Counter := Counter + Step;

  pragma Assert (Counter > 0);
end Increment;
```
Subprogram Contracts: Pre- and Post-conditions

Having the **assertions in the specification** does give us some benefits:

- You can write them when you specify the subprogram. – This is a benefit for us **humans**.
- They are visible to the user of the subprogram. – This is a benefit for us **humans**.
- They are explicitly a part of the interface between the subprogram and its users. – This is a benefit for static analysis **tools**.

But none of these benefits can easily be shown to **really** scale to a significant difference in development effort.
Type Contracts

When you put contracts on your types, you get a nice **scaling benefit**, as you write the contract once, but the compiler automatically inserts checks of the contract (assertions) everywhere you modify variables of the type in a way that might break the contract.

If your compiler is good, it will try to prove these automatically inserted assertions at compile-time, and only keep those it can’t prove.
Type Contracts: Ranges

The simplest kind of type contracts in Ada is the range:

```ada
subtype Natural is Integer range 0 .. Integer'Last;
subtype Positive is Integer range 1 .. Integer'Last;
```

You can declare ranges of numeric types and enumeration types:

```ada
subtype Non_Negative_Float is Float
  range 0.0 .. Float'Last;

type Months is (Jan, Feb, Mar, Apr, May, Jun,
  Jul, Aug, Sep, Oct, Nov, Dec);
subtype Winter is Months range May .. Oct;
```
The next step up in complexity is the **static predicate**:

```plaintext
subtype Summer is Months
  with Static_Predicate => Summer in Nov .. Dec | Jan .. Apr;
```

This allows you to put constraints formulated as static set conditions on your subtypes.
Type Contracts: Dynamic Predicates

The most advanced form of type contract in Ada exists in two forms. One is the public **dynamic predicate**:

```ada
subtype Prime is Integer range 2 .. Integer'Last
with Dynamic_Predicate
   => (for all N in 2 .. Prime - 1
       => Prime mod N /= 0);
```

Any kind of Boolean expression is allowed in a dynamic predicate. You can even use one which changes with time:

```ada
subtype Past_Time is Ada.Calendar.Time
with Dynamic_Predicate => Past_Time < Clock;

subtype Last_Hour is Past_Time
with Dynamic_Predicate => Clock - 3600.0 <= Last_Hour;
```
Type Contracts: Type Invariants

For private types with internal constraints, you use a type invariant:

```ada
package Places is
  type Disc_Point is private;
  -- various operations on disc points
private
  type Disc_Point is
      record
          X, Y : Float range -1.0 .. +1.0;
      end record
  with Type_Invariant => Disc_Point.X ** 2 + Disc_Point.Y ** 2 <= 1.0;
end Places;
```

Adapted from the Ada 2012 Rationale.
I advise that you work on your contracts in this order:

1. Specify **types**.
2. Specify **subprograms**.
3. Adapt the subprogram specifications based on use cases for your **package/library**.
Types

Make sure your type declarations are as detailed as possible.

- Declaring a new type or a subtype depends on what level of inter-type compatibility you want – and of course if there is a type to derive from.
- Put an appropriate constraint on the range of values the (sub)type can have.
- Add any extra constraints as predicates (non-private types) or type invariants (private types).

The simpler a kind of contract you use to declare the type, the more things you can use it for.
Types: Refining a Contract

Primes are integers:

```haskell
subtype Prime is Integer;
```

... larger than 1:

```haskell
subtype Prime is Integer range 2 .. Integer'Last;
```

... and have no other factors than 1 and the prime itself:

```haskell
subtype Prime is Integer range 2 .. Integer'Last
with Dynamic_Predicate
=> (for all N in 2 .. Prime - 1
   => Prime mod N /= 0);
```
Subtypes of discrete types declared with ranges can be used as **array indices**, while those declared with predicates or type invariants can’t.

So when we declare the subtype `Positive` like this:

```ada
subtype Positive is Integer range 1 .. Integer’Last;
```

... then we can declare the array type `String` like this:

```ada
type String is array (Positive range <>) of Character;
```
Subtypes declared with ranges or static predicates can be used in **case statements**, while those declared with dynamic predicates or type invariants can’t.

So when we declare the seasons like this:

``` ADA
subtype Spring is Months range Mar .. May;
subtype Summer is Months range Jun .. Aug;
subtype Autumn is Months range Sep .. Nov;
subtype Winter is Months
  with Static_Predicate => Winter in Dec | Jan | Feb;
```
... then we can use the seasons in a case statement like this:

```vhdl
case Input is
  when Spring =>
    Put_Line ("Light and warmer weather.");
  when Summer =>
    Put_Line ("Vacation and strawberries.");
  when Autumn =>
    Put_Line ("Wind and falling leaves.");
  when Winter =>
    Put_Line ("Snow - we hope.");
end case;
```
Make sure that you declare the arguments for your subprograms as **specifically** as possible.

- Select the proper direction ("in", "out" or "in out") for each of the arguments to a subprogram.
- Select as specific a (sub)type as possible for each of the arguments to a subprogram.
- Use pre-conditions (post-conditions) to declare stronger constraints on the input (output) values than those implied by the selected subtypes.
We want to be able to increment a counter by arbitrary steps. We use ("in") the value of both the counter and the step size to generate ("out") a new value for the counter:

```plaintext
procedure Increment (Counter : in out Integer;
                     Step     : in Integer);
```

We count from zero and up (natural numbers). An increment is by one or more (positive numbers):

```plaintext
procedure Increment (Counter : in out Natural;
                     Step     : in Positive);
```
Subprograms: Refining a Specification

There is an upper limit (Natural’Last) to how far we can count with our selected type:

```plaintext
procedure Increment (Counter : in out Natural;
                      Step    : in    Positive)
   with Pre => (Counter < Natural’Last) and

Once Increment returns Counter has changed:

procedure Increment (Counter : in out Natural;
                      Step    : in    Positive)
   with Pre => (Counter < Natural’Last) and
             (Step <= Natural’Last - Counter),
   Post => (Counter > 0);
```
Subprograms: Refining a Specification

What are the requirements of your subprograms?

- Do some of your subprograms have some special requirements, which should be met before they can be called?
- Can a subprogram only be called once?
- Can a subprogram only be called when the system is in a specific state?

This can be documented with appropriately formulated pre-conditions to the subprograms.
If we want to write to a file, it should be open and writable:

```pascal
procedure Put (File : in   File_Type;
               Item : in   String)
with Pre => (Is_Open (File)) and then
         (Mode (File) in Out_File | Append_File);
```

Initialise only once:

```pascal
procedure Initialise
with Pre => State = Not_Initialised;
```
In my view, the ideal pre- and post-conditions are simply “A_Formal_Parameter in A_Subtype”, but there are cases – such as the example on the preceding slide – where the contracts necessarily have to be more complex than that.
Ada doesn’t allow you to write contracts packages as such. It still make sense to take a broader view of the all the contracts in a package.

If one specifies contracts one subprogram at a time, one may miss contract details on one subprogram, which would be helpful for another subprogram.

The following slides contain a few guidelines for ensuring consistent pre- and post-conditions for entire packages.
Packages: Aligning Pre- and Post-conditions

Do post- and pre-conditions match for likely sequences of calls to the declared subprograms?

1. Identify **use cases** for the package (sequences of subprogram calls).

2. For each call in a use case:
   a. Verify that the **documented state** of the input data matches constraints and pre-conditions for the called subprogram.
   b. If there is a mismatch: Attempt to narrow down the documented, possible output values of the source of the input data (by changing constraints and post-conditions).
   c. Identify the documented state of the modified parameters after the call.
Packages: Aligning Pre- and Post-conditions

We look at a simple Text I/O package with some contracts added:

```plaintext
procedure Open (File : in out File_Type;
          Mode : in File_Mode;
          Name : in String);

procedure Close (File : in out File_Type);

procedure Put_Line (File : in File_Type;
              Item : in String)
    with Pre => (Is_Open (File)) and then
              (Mode (File) in Out_File | Append_File),
    Post => (Line (File) = Line (File)’Old + 1);
```

Jacob Sparre Andersen  
Contract-based Programming: a Route to Finding Bugs Earlier
Packages: Aligning Pre- and Post-conditions

A use case:

1. Open (File => Target, Name => "output.txt", Mode => Out_File);
   Put_Line (File => Target, Item => "Hello.");
   Close (File => Target);

2. Open:
   a. File, Name and Mode all OK. No pre-conditions. (no mismatch)
   b. Target can have any valid File_Type value.
Packages: Aligning Pre- and Post-conditions

2. **Put_Line:**
   
   a. Pre-conditions on File not matched by the documented constraints on Target. Item OK.
   
   b. Target was last modified by Open, so we add some appropriate post-conditions there:

   ```hs
   procedure Open (File : in out File_Type;
       Mode : in     File_Mode;
       Name : in     String)
   with Post => (Is_Open (File) and
                 Text_IO.Mode (File) = Mode);
   ```

   c. We now know that Target is open and has the mode Out_File.
Packages: Aligning Pre- and Post-conditions

2. Close:

- **a** Close has no pre-conditions, so Target matches the **documented** requirements for the formal parameter File.
- **b** (no mismatch)
- **c** We know that Target has been changed, so it can have any valid File_Type value.

As some of you may have noticed, I have omitted to document that it is an error to open a file which already is open, or to close one which already is closed. – This is left as an exercise.
Don’t disable unproven assertions.

It is possible to write your assertions centralised, and then have the compiler insert them where it can’t prove that they are not violated.

Don’t use more advanced contract notations than required by your problem.

Use use cases for your packages to check if your contracts are complete.
Jacob Sparre Andersen
JSA Research & Innovation
jacob@jacob-sparre.dk
http://www.jacob-sparre.dk/

Examples:
http://www.jacob-sparre.dk/programming/contracts/fosdem-2018-examples.zip