A Short Introduction to $^+$CAL

Michał Kapałka

EPFL, LPD

STiDC'06, 31.X 2006

Why $^{+}$CAL

- $^{+}$CAL $=$ algorithm language $\neq$ programming language
- Simple, clear syntax (Pascal- or C-like) $\Rightarrow$ easy to use and understand
- Powerful tool for model-checking of algorithms
- Expressive: many mathematical constructs available
- Efficiency/implementation details a non-issue

+CAL in Short

- No types, objects, pointers, etc. – pure simplicity
- Standard constructs available: if, while, print, goto, procedures, macros and others
- Multi-threaded programs explicitly divided into (atomic) steps (by labels) $\Rightarrow$ no ambiguity
- Non-deterministic behaviour can be easily expressed: range variables, either/or statements
- +CAL algorithms are translated to TLA$^+$
- TLA$^+$ expressions can be used inside +CAL – ideas can be expressed quickly, although not all can be executed as programs
- Some concepts take time to get used to: arrays as functions, no return values of procedures, labels, ...

Today

1. Short tour over $^{+}$CAL by examples
2. Details: in the $^{+}$CAL manual (see the course web page this evening)
3. Two exercises about $^{+}$CAL

Simple Example

Check whether *n* is a prime number (the simplest algorithm possible):

```
--algorithm IsPrimeNumber
  variables n ∈ 1..N, k = 2, m, answer = TRUE;
  begin
    while (k < n) ∧ answer do
      m := 2;
      while (m ≤ k) ∧ answer do
        if m * k = n then answer := FALSE; end if;
        m := m + 1;
      end while;
      k := k + 1;
    end while;
    print ⟨ n, answer ⟩;
  end algorithm
```

A TLA$^+$ Module for Our Example

File `IsPrime.tla`:

```
----- MODULE IsPrime -----
EXTENDS Naturals, TLC
CONSTANT N

(* --algorithm IsPrimeNumber
...
*)

\* BEGIN TRANSLATION
\* END TRANSLATION

=====
```

Automatic Model-Checking

How to check if the algorithm is correct?

- Using `print` $\Rightarrow$ tedious
- Assertions – check that something should be true at some point
- Invariants – check that something should always be true

Assertions in Our Example

Instead of `print ⟨ n, answer ⟩`:

- We put `assert answer = isprime(n)`
- `isprime` is a TLA$^+$ operator, which needs to be defined just after the algorithm (before the `BEGIN TRANSLATION` line)
- For example:

$$isprime(n) \triangleq \neg \exists\, k, m \in 2..n : k * m = n$$

A Configuration File for Our Example

File `IsPrime.cfg`:

```
SPECIFICATION Spec
\* Add statements after this line.
CONSTANT N = 200
```

A Quick How-To

1. Write a `.tla` file with an algorithm (module name = file base name!)
2. Set the `CLASSPATH` to the $^+$CAL/TLA$^+$ directory
3. Translate to TLA$^+$: `java pcal.trans Algorithm`
4. Edit the configuration file `Algorithm.cfg`
5. To run: `java tlc.TLC -simulate Algorithm`
6. To check all possible executions: `java tlc.TLC Algorithm`

Multi-Threaded Example Algorithm

We will try to implement binary consensus in +CAL:

- Each process may propose 0 or 1 (input value)
- The process then decides on (returns) a single value, 0 or 1
- Agreement: no two processes decide differently
- Validity: the value decided is one of the values proposed
- Wait-freedom: every correct process that proposes a value eventually decides

We will use write-once registers (only the first operation writes its value)

Multi-Threaded Example Algorithm (contd.)

```
EXTENDS Naturals, TLC, Sequences
CONSTANT N

NONE == CHOOSE n : n ∉ 0, 1

(* --algorithm ConsAlg
   variables pvalues = [i ∈ 1..N ↦ NONE],
             values = [i ∈ 1..N ↦ NONE],
             R = NONE;

   macro WRITE(val)
   begin
     if R = NONE then R := val end if;
   end macro
```

Multi-Threaded Example Algorithm (contd.)

```
procedure propose(pval)
begin
  write:   WRITE(pval);
  decide:  values[self] := R;
  ret:     return;
end procedure

process Proc ∈ 1..N
begin
  pval:   either pvalues[self] := 0
              or pvalues[self] := 1 end either;
  prop:   call propose(pvalues[self]);
end process
end algorithm *)
```

Consensus Properties

We can make $^+$CAL check the consensus properties for all possible executions of N processes:

$$\begin{aligned} \mathit{Agreement} \triangleq\ & \forall\, i, k \in 1..N : \\ & (\mathit{values}[i] = \mathit{NONE}) \lor \\ & (\mathit{values}[k] = \mathit{NONE}) \lor \\ & (\mathit{values}[i] = \mathit{values}[k]) \end{aligned} \tag{1}$$

$$\begin{aligned} \mathit{Validity} \triangleq\ & \forall\, i \in 1..N : \\ & (\mathit{values}[i] \neq \mathit{NONE}) \Rightarrow \\ & (\exists\, k \in 1..N : (\mathit{values}[i] = \mathit{pvalues}[k])) \end{aligned} \tag{2}$$

(We put them after `END TRANSLATION`)

Configuration File for the Multi-Threaded Example

File `Cons.cfg`:

```
SPECIFICATION Spec
\* Add statements after this line.
CONSTANT N = 2
         NONE = 2
INVARIANT Agreement Validity
```

Exercise for Today

1. Write a $^+$CAL algorithm that implements consensus using a queue (see the previous lecture).
2. Write one of the register transformations presented today using $^+$CAL. Note: if you use safe or regular registers (to implement other ones), then try to define them in $^+$CAL such that they are really safe/regular, not atomic.

To get a bonus point, please:

1. Translate and try your algorithms using the $^+$CAL/TLA$^+$ toolkit.
2. Send me the `.tla` files by e-mail before the next exercises. Put in comments your name and a short information about what and how the algorithm does.
3. Prepare a short (printed) report with the algorithms (you can use, e.g., TLATEX) and bring it to the next exercises (alternatively, you can leave it in my office).

Hints for Exercise 1

- Define first a queue using a macro (only a dequeue operation)
- A sequence in $^{+}$CAL is defined as $\langle e_1, e_2, \ldots, e_k \rangle$, where $e_1, \ldots, e_k$ are its elements
- Two useful operations on sequences: *Head(seq)* and *Tail(seq)* (e.g., $\mathit{Head}(\langle 1, 2 \rangle) = 1$ and $\mathit{Tail}(\langle 1, 2, 3 \rangle) = \langle 2, 3 \rangle$
- Strings are in double-quotes, e.g. `"winner"`

Hints for Exercise 2

- Macros are always executed in a single step $\Rightarrow$ this will define an atomic register:

  ```
  macro WRITE(value) R := value; end macro
  ```

- To define a safe/regular register you have to split an operation into ivocation and response, so that something can happen inbetween:

  ```
  macro RW_INIT() ...end macro
  macro WRITE(value) ...end macro
  ```

  Clearly, when using the safe/regular register you should first invoke `RW_INIT` and then `READ` or `WRITE` (with separate labels).