TLA$^+$ specification of *PCR* parallel programming pattern

Work in Progress

José E. Solsona[1] Sergio Yovine[2]

Universidad ORT Uruguay

September 2020

[1] solsona@fi365.ort.edu.uy
[2] yovine@fi365.ort.edu.uy

Agenda

1. Goals
2. *PCR*: Produce-Consume-Reduce pattern
   - High level description
   - *PCR* elements: sintax & semantics
   - Example: the Fibonacci Prime counter v1
   - Composition
   - Example: the Fibonacci Prime counter v2
3. TLA+ specification of *PCR*
   - High-level overview
   - Contexts and Contexts mappings
   - Concrete *PCR* modules and the main spec
   - *PCR* elements with basic functions
   - *PCR* elements with nesting
   - Spec properties and verification
4. Current state and further work

Agenda

1. Goals
2. *PCR*: Produce-Consume-Reduce pattern
3. TLA+ specification of *PCR*
4. Current state and further work

Goals

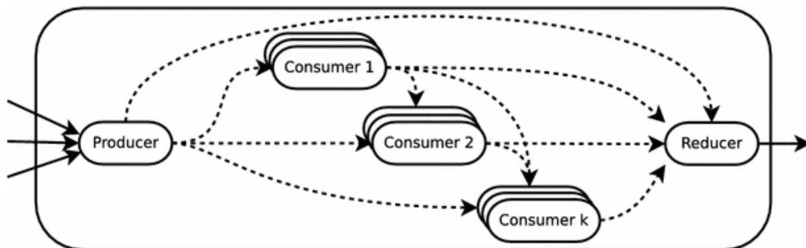
Our research goal is to formalize the semantics of a parallel programming pattern called *PCR* in terms of TLA^+. In this way, we can leverage TLA^+ related tools to prove temporal properties of *PCR* programs. Besides correctness and termination, we are particularly interested in proving *refinement*. Moreover, we envisage to develop a translator from *PCR* into TLA^+ to make the integration seamless.

Agenda

1. Goals
2. *PCR*: Produce-Consume-Reduce pattern
   - High level description
   - *PCR* elements: sintax & semantics
   - Example: the Fibonacci Prime counter v1
   - Composition
   - Example: the Fibonacci Prime counter v2
3. TLA+ specification of *PCR*
4. Current state and further work

High level description

The PCR pattern aims at expressing computations consisting of a *producer* consuming input data items and generating, for each one of them, a data set to be consumed by several *consumers* working in parallel. Their outputs are finally aggregated back into a single result by a *reducer*.



PCRs emphasize the independence between different computations in order to expose all parallelization opportunities.

High level description

Data flow inside a PCR is as follows:

1. For each input data item, the producer component generates a set of output values; each one being immediately available for reading.
2. Consumers read values from the outer scope and from the private data channels to perform their computations.
3. A reducer combines values from one or more data sources coming from the producer and one or more consumers, generating a single output item for every input item processed by the producer.

High level description

Some remarks:

- Reads in data channels are nondestructive, i.e., the same value can be read multiple times by any consumer and by the reducer.
- No input is ignored, i.e., every item is handled by some component—all dashed arrows carry the same number of data items to be read.
- Producer, consumers, and reducer work in parallel subject to data dependencies: all input items must be available for a consumer/reducer instance in order to perform its calculation.

PCR elements: sintax & semantics

We refer as *basic functions*, to user provided functions implemented in the host language. These are iterated by the **produce**, **consume** and **reduce** elements of the PCR pattern.

Syntax of the principal PCR elements (simplified version):

p = **produce**[**Seq**] f x	where f is a basic function or another PCR, and x is PCR input variable
c = **consume** f x p	where f is a basic function or another PCR, x is PCR input variable and p is producer output variable.
r = **reduce** $\oplus$ v_0 c	where $\oplus$ is a commutative and associative operation, v_0 is an initial neutral value and c is consumer output variable.

PCR elements: sintax & semantics

Output variables p and c describes full history of assignments for producers and consumers respectively. This is achieved by dynamic and automatic indexing of each computed value. We denote by p_i the i-th produced value to be consumed at instance i for which corresponding result is c_i.

This property is leveraged into a syntactic mechanism which allows stream operations *look-ahead*/*look-behind* to be used on variables (subject to some restrictions to be discussed) by indexing.

For example, to produce p_i as the i-th Fibonacci number, two previous indexes are accessed to compute the sum: $p_{i-1} + p_{i-2}$.

PCR execution starts with the producer iterating f which produces values p_i for indexes i in some domain.
The domain of i is determined by an *iteration space* prescribed to f which is also provided by the user.

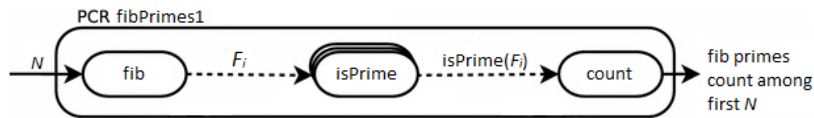
Definition of the *iteration space* consist on:

lbnd $f = \lambda x.\ e$	where $\lambda x.\ e$ is the lower bound expressed as a function of input variable x
ubnd $f = \lambda x.\ e$	where $\lambda x.\ e$ is the upper bound expressed as a function of input variable x
step $f = \lambda i.\ e$	where $\lambda i.\ e$ is a step function

Example: the Fibonacci Prime counter v1

We illustrate previous concepts by two alternative but equivalent PCR specifications of an algorithm that counts primes among the first N Fibonacci numbers. The first PCR is called `fibPrimes1`, it works as follows:

1. The producer `fib` generates the sequence F_0, F_1,..., F_N of Fibonacci numbers.
2. Each instance $i \in [0, N]$ of the `isPrime` consumer checks, in parallel, the primality of F_i, resulting in the unordered output of indexed boolean values $\texttt{isPrime}(F_i)$.
3. The reducer `count` counts the number of those outputs which are true.

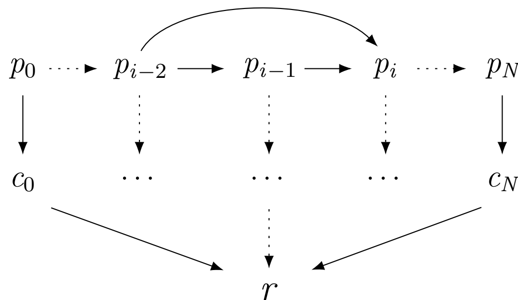


Example: the Fibonacci Prime Counter v1

Sintax

```
1   // Basic functions
2   fun fib(N,p,i) =
3     if i < 2
4     then 1
5     else p_{i-1} + p_{i-2}
6
7   fun isPrime(N,p,i) = ...
8
9   fun count(a,b) =
10    a + (if b then 1 else 0)
11
12  // Iteration space
13  lbnd fib = λ x. 0
14  ubnd fib = λ x. x
15  step fib = λ i. i + 1
16
17  // PCR definition
18  PCR fibPrimes1(N)
19    par
20      p = produceSeq fib N
21      forall p
22        c = consume isPrime N p
23      r = reduce count 0 c
```

Semantics



- In this example, for each $i \in [0, N]$ we have $p_i = F_i$ and $c_i = \texttt{isPrime}(F_i)$.
- $r = \sum_{i \in [0,N]}$(if c_i then 1 else 0)

PCR composition

*PCR*s can be composed by hierarchical nesting, this ability allows reusing components and controlling the desired grain of parallelism.

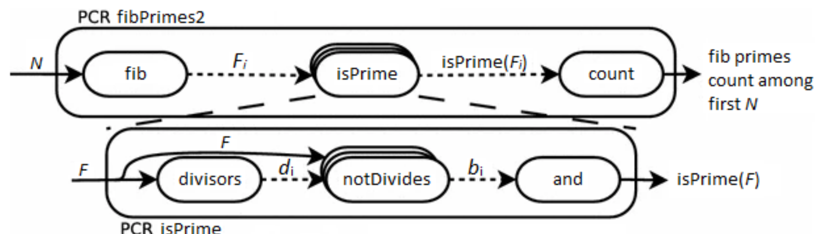
Let I be the index dynamically assigned to a particular execution of a *PCR*. Any child *PCR* inherits the index of the father and extends its dimension by writing in its producer variable, say p, the (I, i)-th value $p_{I,i}$, for every i according to his iteration space.

This multidimensional indexing allows for the concurrent execution of any two instances $I \neq J$ of the producer, each one generating its own set of p values, namely $p_{I,i}$ and $p_{J,j}$.

Example: the Fibonacci Prime counter v2

The second version of our example is the PCR `fibPrimes2`, where `isPrime` is another PCR instead of a basic function and it works as follows:

1. The producer `divisors` generates all the possible divisors of the input number F.
2. Each instance i of the `notDivides` consumer checks, in parallel, the divisibility of F by d_i, resulting in the unordered output of indexed boolean values b_i.
3. The reducer `and` computes the conjunction of those outputs.

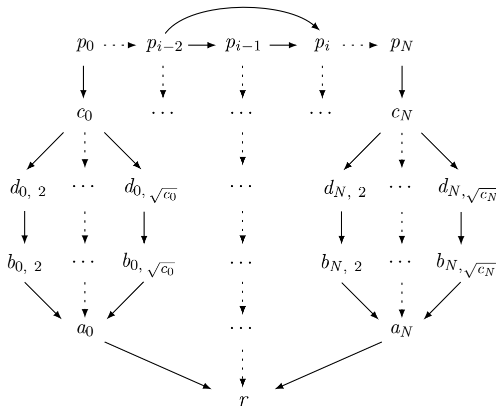


Example: the Fibonacci Prime counter v2

Sintax

```
1   // Basic functions
2   fun divisors(F, d, i) = i
3
4   fun notDivides(F, d, i) =
5     not (F % d_i = 0)
6
7   // Iteration space
8   lbnd divisors = λx. 2
9   ubnd divisors = λx. √x
10  step divisors =
11    λi. if i = 2 then 3 else i + 2
12
13  // PCR definitions
14  PCR fibPrimes2(N)
15    par
16      p = produceSeq fib N
17      forall p
18        c = consume isPrime p
19      r = reduce count 0 c
20
21  PCR isPrime(F)
22    par
23      d = produce divisors F
24      forall d
25        b = consume notDivides F d
26      a = reduce and true b
```

Semantics



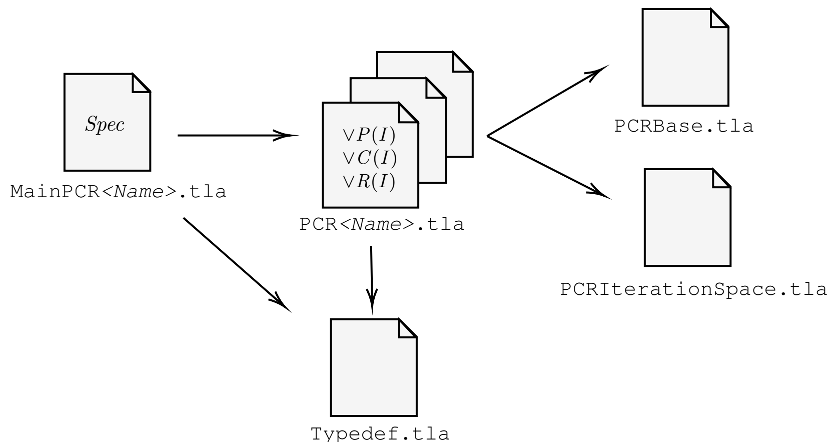
- For each $i \in [0, N]$ and $j \in [2, \sqrt{c_i}]$, $d_{i,j}$ denotes the j-th possible divisor produced for F_i.
- $a_i = \bigwedge_{j \,\in\, [2,\sqrt{c_i}]} b_{i,j}$

Agenda

1. Goals
2. *PCR*: Produce-Consume-Reduce pattern
3. TLA+ specification of *PCR*
   - High-level overview
   - Contexts and Contexts mappings
   - Concrete *PCR* modules and the main spec
   - *PCR* elements with basic functions
   - *PCR* elements with nesting
   - Spec properties and verification
4. Current state and further work

High-level overview

We organize our projects across different TLA^+ modules in a way is convenient for us to handle and hopefully will also be useful for the task of automatic translation from *PCR* syntax to TLA^+.



Contexts and Contexts mappings

In a *PCR*, variables are streams indexed with multidimensional indexes which are automatically generated by the underlying runtime system. We formalize this by *contexts* and *context mappings*.

Let $VarPType$ and $VarCType$ be basic data types. For producer and consumer output variables we define:

$$VarP \triangleq [Nat \rightarrow [v : VarPType \cup \{NULL\}, r : Nat]]$$
$$VarC \triangleq [Nat \rightarrow [v : VarCType \cup \{NULL\}, r : Nat]]$$

Where field v denotes the value of the variable at some i-th assignment, with $i \in Nat$ and $NULL$ meaning it has not occured so far, and field r is used to keep track of the number of times it has been read.

Contexts and Contexts mappings

A *context* represents the PCR state at inner scope:

$$
\begin{array}{lll}
Ctx \triangleq [in & : InType, & \text{input} \\
\quad i_p & : Nat, & \text{iteration index} \\
\quad v_p & : VarP, & \text{producer history} \\
\quad v_c & : VarC, & \text{consumer history} \\
\quad ret & : VarRType, & \text{reducer result} \\
\quad ste & : \{OFF, RUN, END\}] & \text{discrete state}
\end{array}
$$

Multidimensional indexes are modeled by sequences of Nat. A *context mapping* is a partial function from indexes to contexts:

$$CtxMap \triangleq [Seq(Nat) \rightarrow Ctx \cup \{NULL\}]$$

Any *PCR* have its own context mappping $map \in CtxMap$, so $map[I]$ is the *PCR* context at some index I, or in other words, the I-th *PCR* instance.

Contexts and Contexts mappings

For convenience, we give names to context elements:

$$\begin{array}{lcl} in(I) & \triangleq & map[I].in \\ i_p(I) & \triangleq & map[I].i_p \\ v_p(I) & \triangleq & map[I].v_p \\ v_c(I) & \triangleq & map[I].v_c \\ out(I) & \triangleq & map[I].ret \\ state(I) & \triangleq & map[I].ste \end{array}$$

We read $v_p(I)[i].v$ as the i-th produced value at index I, that is, informally $p_{I,i}$.

Also, some useful predicates are defined for output variables:

$$\begin{array}{lcl} written(var,\ i) & \triangleq & var[i].v \neq NULL \\ read(var,\ i) & \triangleq & var[i].r > 0 \end{array}$$

Iteration space

A *PCR* has associated an iteration space which defines the indexes generated by the *PCR*. We define an $iterator(_)$ operator which describes a valid range in terms of higher-order operators $step(_)$, $lowerBnd(_)$ and $upperBnd(_)$.

$$
\begin{array}{l}
range(start, end, step(_)) \triangleq \\
\quad \text{LET } f[i \in Nat] \triangleq \text{IF } i \leq end \\
\qquad\qquad\qquad\qquad\quad \text{THEN } \{i\} \cup f[step(i)] \\
\qquad\qquad\qquad\qquad\quad \text{ELSE } \{\} \\
\quad \text{IN } \quad f[start]
\end{array}
$$

$$
iterator(I) \triangleq range(lowerBnd(in(I)), upperBnd(in(I)), step)
$$

Concrete *PCR* modules and the main spec

Every concrete PCR module describes its initial conditions by means of operator $initCtx$:

$$
\begin{aligned}
initCtx(x) \triangleq [&in \mapsto x,\\
&i_p \mapsto lowerBnd(x),\\
&v_p \mapsto [n \in Nat \mapsto [v \mapsto NULL,\ r \mapsto 0]],\\
&v_c \mapsto [n \in Nat \mapsto [v \mapsto NULL,\ r \mapsto 0]],\\
&ret \mapsto \text{initial neutral value},\\
&ste \mapsto OFF]
\end{aligned}
$$

And also describes what is the possible $Next$ step as a disjunction of actions:

$$
\begin{aligned}
Next(I) \triangleq\ &\lor \land state(I) = OFF\\
&\quad \land Start(I)\\
&\lor \land state(I) = RUN\\
&\quad \land \lor P(I)\\
&\quad\quad \lor C(I)\\
&\quad\quad \lor R(I)\\
&\quad\quad \lor Quit(I)
\end{aligned}
$$

Concrete *PCR* modules and the main spec

Main PCR module instantiates the root PCR and any other PCR involved. Here the main system specification is defined. Let $PCR1$ and $PCR2$ be two instances of some PCR modules, then main module will have roughly this structure:

$$
\begin{array}{lcl}
Init & \triangleq & \land\; x \in InType1 \\
 & & \land\; map1 = [I \in Seq(Nat) \mapsto \text{IF } I = \langle 0 \rangle \\
 & & \qquad\qquad\qquad\qquad\qquad\quad \text{THEN } PCR1!InitCtx(x) \\
 & & \qquad\qquad\qquad\qquad\qquad\quad \text{ELSE } NULL] \\
 & & \land\; map2 = [I \in Seq(Nat) \mapsto NULL]
\end{array}
$$

$$
\begin{array}{lcl}
Done & \triangleq & \land\; \forall I \in Seq(Nat) : PCR1!Finished(I) \\
 & & \land\; \forall I \in Seq(Nat) : PCR2!Finished(I) \\
 & & \land\; vars' = vars
\end{array}
$$

$$
\begin{array}{lcl}
Next & \triangleq & \lor\; \exists I \in Seq(Nat) : PCR1!Next(I) \\
 & & \lor\; \exists I \in Seq(Nat) : PCR2!Next(I) \\
 & & \lor\; Done
\end{array}
$$

$$
\begin{array}{lcl}
Fair & \triangleq & \land\; \forall I \in Seq(Nat) : \text{WF}_{vars}(PCR1!Next(I)) \\
 & & \land\; \forall I \in Seq(Nat) : \text{WF}_{vars}(PCR2!Next(I))
\end{array}
$$

$$
Spec \triangleq Init \land \Box[Next]_{vars} \land Fair
$$

PCR elements with basic functions

We further illustrate our specification using the PCR `fibPrimes1` example.

Producer specification:

$p \; = \; \mathbf{produceSeq} \; \texttt{fib} \; N$

$$
\begin{aligned}
&P(I) \triangleq \\
&\quad \land\, i_p(I) \in iterator(I) \\
&\quad \land\, map1' = [map1 \text{ EXCEPT} \\
&\qquad\quad ![I].v_p[i_p(I)] = [v \mapsto \texttt{fib}(in(I),\, v_p(I),\, i_p(I)), \\
&\qquad\qquad\qquad\qquad\qquad r \mapsto 0], \\
&\qquad\quad ![I].i_p = step(@)]
\end{aligned}
$$

Consumer specification:

$c \; = \; \mathbf{consume} \; \texttt{isPrime} \; N \; p$

$$
\begin{aligned}
&C(I) \triangleq \\
&\quad \exists\, i \in iterator(I) : \\
&\qquad \land\, written(v_p(I),\, i) \\
&\qquad \land\, \neg read(v_p(I),\, i) \\
&\qquad \land\, \neg written(v_c(I),\, i) \\
&\qquad \land\, map1' = [map1 \text{ EXCEPT} \\
&\qquad\quad ![I].v_p[i].r = 1, \\
&\qquad\quad ![I].v_c[i] = [v \mapsto \texttt{isPrime}(in(I), v_p(I), i), \\
&\qquad\qquad\qquad\qquad r \mapsto 0]\;]
\end{aligned}
$$

PCR elements with basic functions

We further illustrate our specification using the PCR `fibPrimes1` example.

Reducer specification:

```
r = reduce count 0 c
```

$$
\begin{array}{l}
R(I) \triangleq \\
\quad \exists\, i \in \mathit{iterator}(I) : \\
\qquad \land\ \mathit{written}(v_c(I),\ i) \\
\qquad \land\ \neg \mathit{read}(v_c(I),\ i) \\
\qquad \land\ \mathit{map1}' = [\mathit{map1}\ \text{EXCEPT} \\
\qquad\qquad !\,[I].\mathit{ret} \quad = \mathtt{count}(@,\ v_c(I)[i].v), \\
\qquad\qquad !\,[I].v_c[i].r \quad = @ + 1, \\
\qquad\qquad !\,[I].\mathit{ste} \quad = \text{IF}\ \mathit{cDone}(I,\ i) \\
\qquad\qquad\qquad\qquad \text{THEN}\ \mathit{END} \\
\qquad\qquad\qquad\qquad \text{ELSE}\ \ @]
\end{array}
$$

Where $cDone$ is a predicate that holds true if every consumer variable on index other than i has been read.

PCR elements with nesting

PCR `fibPrimes2` uses PCR `isPrime` as a consumer.

First, we show the producer of the PCR `isPrime` as it is not a sequential one.

Producer specification:

d = **`produce`** `divisors` F

$$
\begin{array}{l}
P(I) \triangleq \\
\quad \exists\, i \in iterator(I) : \\
\quad\quad \land \neg written(v_p(I),\, i) \\
\quad\quad \land\, map3' = [map3 \text{ EXCEPT} \\
\quad\quad\quad\quad ![I].v_p[i] = [v \mapsto \texttt{divisors}(in(I), v_p(I), i), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad r \mapsto 0]]
\end{array}
$$

Consumer specification of PCR fibPrimes2:

```
c = consume isPrime p
```

$$C(I) \triangleq C_call(I) \lor C_ret(I)$$

$$\begin{array}{l} C_call(I) \triangleq \\ \quad \exists\, i \in iterator(I) : \\ \qquad \land\, written(v_p(I),\, i) \\ \qquad \land\, \neg read(v_p(I),\, i) \\ \qquad \land\, map2' = [map2 \text{ EXCEPT } ![I].v_p[i].r = 1] \\ \qquad \land\, map3' = [map3 \text{ EXCEPT } \\ \qquad\qquad ![I \circ \langle i \rangle] = isPrime!initCtx(v_p(I)[i].v)] \end{array}$$

$$\begin{array}{l} C_ret(I) \triangleq \\ \quad \exists\, i \in iterator(I) : \\ \qquad \land\, read(v_p(I),\, i) \\ \qquad \land\, \neg written(v_c(I),\, i) \\ \qquad \land\, isPrime!finished(I \circ \langle i \rangle) \\ \qquad \land\, map2' = [map2 \text{ EXCEPT } \\ \qquad\qquad ![I].v_c[i] = [v \mapsto isPrime!out(I \circ \langle i \rangle), \\ \qquad\qquad\qquad\qquad r \mapsto 0]\ ] \end{array}$$

Where $isPrime$ is an instance of module `PCRIsPrime`.

Spec properties and verification

Correctness and termination properties are specified in the main module. Our two previous PCR examples, `fibPrimes1` and `fibPrimes2`, have the same properties:

$$\begin{aligned}
solution(x) \;&\triangleq\; \text{LET } allFibonacci \triangleq \{fibonacci[n] : n \in 0 \,..\, x\} \\
&\qquad \text{IN } \; Cardinality(\{k \in allFibonacci : isPrime(k)\}) \\
Correctness \;&\triangleq\; \Box(fibPrimes\,!\,finished(\langle 0 \rangle) \Rightarrow fibPrimes\,!\,out(\langle 0 \rangle) = solution(N)) \\
Termination \;&\triangleq\; \Diamond fibPrimes\,!\,finished(\langle 0 \rangle)
\end{aligned}$$

Where $fibPrimes$ stands for an instance of either module `PCRFibPrimes1` or `PCRFibPrimes2` and $N \in Nat$ is the input variable.

Spec properties and verification

We can relate `fibPrimes1` and `fibPrimes2` by proving the latter is an implementation of the former, or in other words, the former is an abstraction of the latter, under an appropriate refinement mapping.

In module `MainPCRFibPrimes1` there is a context mapping $map1 \in \mathit{fibPrimes}1!\mathit{CtxMap}$. This is the high-level spec.

In module `MainPCRFibPrimes2` there are two context mappings, namely $map2 \in \mathit{fibPrimes}2!\mathit{CtxMap}$ and $map3 \in \mathit{isPrime}!\mathit{CtxMap}$. This is the low-level spec. Here, we instantiate `MainPCRFibPrimes1` with $map1$ substituted for an expression in terms of $map2$ and $map3$.

Spec properties and verification

This refinement works by contracting time between actions C_call and C_ret.

$$
\begin{array}{l}
subst \triangleq [I \in \text{DOMAIN } map2 \mapsto \\
\quad \text{IF } map2[I] \neq NULL \\
\quad \text{THEN } [map2[I] \text{ EXCEPT} \\
\quad\quad !.v_p = [i \in \text{DOMAIN } @ \mapsto \\
\quad\quad\quad \text{IF } \land fibPrimes2!read(map2[I].v_p,\ i) \\
\quad\quad\quad\quad \land \neg isPrime!finished(I \circ \langle i \rangle) \\
\quad\quad\quad \text{THEN } [v \mapsto @[i].v, \\
\quad\quad\quad\quad\quad r \mapsto 0] \\
\quad\quad\quad \text{ELSE } @[i] \\
\quad\quad], \\
\quad\quad !.v_c = [i \in \text{DOMAIN } @ \mapsto \\
\quad\quad\quad \text{IF } \land fibPrimes2!read(map2[I].v_p,\ i) \\
\quad\quad\quad\quad \land isPrime!finished(I \circ \langle i \rangle) \\
\quad\quad\quad \text{THEN } [v \mapsto isPrime!out(I \circ \langle i \rangle), \\
\quad\quad\quad\quad\quad r \mapsto @[i].r] \\
\quad\quad\quad \text{ELSE } @[i] \\
\quad\quad] \\
\quad] \\
\quad \text{ELSE } NULL]
\end{array}
$$

$$PCRFibPrimes1 = \text{INSTANCE } MainPCRFibPrimes1 \text{ WITH } map1 \leftarrow subst$$

Spec properties and verification

Model checking and Theorem proving:

- We try to make the specifications to be TLC friendly and also TLAPS friendly. Best of both worlds.
- Currently we can model check properties like correctness, termination and refinements on relatively small models.
- For very large state spaces simulation can be useful.
- Till now, we have only used TLAPS to prove type invariance.

Agenda

1. Goals
2. *PCR*: Produce-Consume-Reduce pattern
3. TLA+ specification of *PCR*
4. Current state and further work

Current state and further work

- Currently we have applied the presented approach to other known problems like Count Words, MergeSort and NQueens. More examples are on the way.
- Other *PCR* syntax blocks are missing: **`iterate`**, **`feedbackloop`**.
- Handle early termination to support eureka computations.
- Translate *PCR* to TLA^+.
- Formalize an abstract model of a target execution runtime and prove refinements.

Thanks!