Universidad ORT Uruguay

Facultad de Ingeniería

On the specification and verification of the PCR parallel programming pattern in TLA^{+}

Entregado como requisito para la obtención del título de

Master en Ingeniería

José E. Solsona - 182285

Tutores: Álvaro Tasistro, Sergio Yovine

2021

Declaración de autoría

Yo, José Eduardo Solsona, declaro que el trabajo que se presenta en esta obra es de mi propia mano. Puedo asegurar que:

- La obra fue producida en su totalidad mientras realizaba el Proyecto;
- Cuando he consultado el trabajo publicado por otros, lo he atribuido con claridad;
- Cuando he citado obras de otros, he indicado las fuentes. Con excepción de estas citas, la obra es enteramente mía;
- En la obra, he acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, he explicado claramente qué fue contribuido por otros, y qué fue contribuido por mi;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

[illegible]

José E. Solsona
4/2/2021

A mi familia, amigos, y mi compañera Isis.

Agradecimientos

Primero que nada quiero agradecer a mis tutores, Dr. Álvaro Tasistro y Dr. Sergio Yovine, por haberme dado la oportunidad de trabajar estos dos últimos años bajo su tutoría en un proyecto que me motivó a querer continuar mis estudios más allá de la carrera de grado. Tengo la esperanza de que ellos hayan notado que yo he dejado todo de mi parte durante este proyecto, y ése es mi verdadero agradecimiento hacia ellos.

También le agradezco a la Dra. Nora Szasz, quien me ayudó a continuar mis estudios y conseguir una beca.

Le agradezco a mis padres por haberme apoyado todo este tiempo, aunque a la distancia. No ha sido fácil para ellos ni para mí, especialmente en medio de la pandemia.

Agradezco al grupo de discusión de TLA$^+$, fue un recurso muy valioso durante mi aprendizaje. Pero, más específicamente, debo agradecer a Stephan Merz, quién respondió a la mayoría de mis preguntas en dicho grupo y además tuvo la cortesía de hacer un encuentro virtual con nosotros para conversar sobre nuestro trabajo.

Finalmente, agradezco el financiamiento otorgado por ANII para la realización de este proyecto.

Resumen

Limitaciones físicas en el diseño de procesadores han hecho que la industria informática desde 2005 pasara de mejorar la velocidad de un solo procesador a aumentar el número de unidades de proceso. Pero el diseño de software que explote la potencia de procesamiento paralelo de forma correcta y efectiva es una tarea desafiante que requiere un alto grado de experiencia. En 2017, Pérez y Yovine propusieron una herramienta basada en patrones de diseño para facilitar el desarrollo de software paralelo. En particular, la herramienta está basada en un patrón de programación paralela, agnóstico de la plataforma, denominado PCR, que describe las computaciones realizadas en forma concurrente por Productores, Consumidores y Reductores que se comunican entre si. Este combina en un único patrón componible varios conceptos como operaciones colectivas, programación basada en flujos, iteración no acotada y recursividad.

En esta tesis, formalizamos la semántica del patrón PCR en términos de TLA$^+$. De esta manera, podemos aprovechar las herramientas asociadas a TLA$^+$ para demostrar propiedades de diseños de PCR de alto nivel, tales como su corrección funcional y refinamientos entre diferentes diseños de PCR. TLA$^+$ es un lenguaje de especificación formal para sistemas concurrentes que se está utilizando en lugares como Intel, Amazon y Microsoft. Contribuimos así al estado del arte en los refinamientos de programas paralelos a partir de modelos abstractos, especialmente utilizando una caracterización alternativa del patrón PCR general y el *framework* TLA$^+$.

Una presentación de trabajo en progreso para esta tesis fue parte del TLA$^+$ *Community Event* que se llevó a cabo (virtualmente) en octubre de 2020 como satélite de la Conferencia DISC 2020.

Todo el trabajo realizado en esta tesis se puede encontrar en: https://github.com/josedusol/PCR.

La investigación que da origen a los resultados presentados en la presente publicación recibió fondos de la Agencia Nacional de Investigación e Innovación bajo el código POS_NAC_2018_1_152201.

Palabras clave: Algoritmos paralelos, Patrones de diseño, Métodos formales, TLA+

Abstract

Physical limitations in processor design have made computer industry since 2005 shift from improving the speed of a single processor to increasing the number of processing core units. But the design of software to exploit parallel processing power in a correct and cost-effective way is a challenging task requiring a high degree of expertise. In 2017, Pérez and Yovine proposed a pattern-based formally grounded tool that eases writing parallel code. In particular, the tool is based on a platform-agnostic parallel programming pattern called PCR, which describes computations performed concurrently by communicating Producers, Consumers and Reducers. It combines in a single and composable pattern several concepts like collective operations, stream programming, unbounded iteration and recursion.

In this thesis, we formalize the semantics of the PCR pattern in terms of TLA$^+$. In this way, we can leverage TLA$^+$ related tools to prove properties about high level PCR designs such as their functional correctness and refinements between different PCR designs. TLA$^+$ is a formal specification language for concurrent systems that is being used at places such as Intel, Amazon and Microsoft. We thus contribute to the state of the art in formal refinement of parallel programs from abstract models, especially starting off from an alternative characterization of the general PCR pattern, and utilizing the TLA$^+$ framework.

A presentation of the work in progress for this thesis was part of the TLA$^+$ Community Event that was held (virtually) in October 2020 as a satellite of the DISC 2020 Conference.

All the work done in this thesis can be found at: https://github.com/josedusol/PCR.

The research that gives rise to the results presented in this publication received funds from the National Research and Innovation Agency under the code POS_NAC_2018_1_152201.

Keywords: Parallel algorithms, Design patterns, Formal Methods, TLA+

Index

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The numbers of transistors incorporated in a chip will approximately double every two years.

GORDON MOORE (1975)

The industry of general purpose electronic computers has made great advances in the past half century. As a testimony, we often hear that "we now have more computing power in our pocket than the computer aboard Apollo 11", the Apollo Guidance Computer (AGC), which landed man on the moon 50 years ago. Indeed, today's smartphones have roughly one million times the memory and one hundred thousand times the processing power of the AGC. And this is available at the modest cost of a few hundred dollars.

But advances don't always come easy. Figure 1.1 reflects quantitatively the period 1975-2010 in the evolution of microprocessors with respect to the number of transistors, number of cores, clock frequency, power consumption and serial performance. In particular, we can observe that there is an inflection point around year 2005 correlating a slowdown on frequency increase with the rise of core count on processors. This is accompanied by a proportional slowdown for serial computing performance and power consumption. As a notable example, in early 2003 Intel had announced the Tejas and Jayhawk microprocessors to be released around 2004-2005. These were the code names for what would be the successors of the latest Pentium 4 with the Prescott core (indicated in the figure). However, Intel abruptly canceled development on May of 2004, announcing they will move away from single core chips in favor of chips with two cores per die, without giving further technical explanation. That announcement, coming from a big player like Intel, is considered the end of the frequency scaling era. But, what happened to suddenly motivate this change and how are the trends on the graph related?.

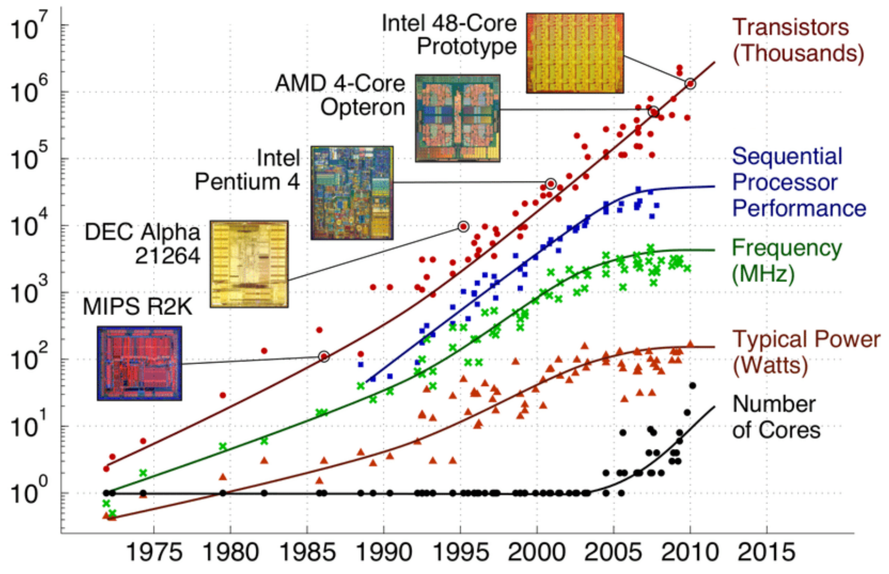


Figure 1.1: Evolution of transistor/cores count, clock frequencies, power consumption and performance in the processor industry from 1975 to 2010. Note the logarithmic scale. (source: [1]).

Historically, advances in semiconductor fabrication allowed to reduce the size (i.e. the *gate lenth*) of transistors of integrated circuits. For example, the first commercially produced microprocessor, the Intel 4004 (1971), had a size of 10 μm, whereas size is predicted to shrunk to 4 nm on 2022 according to the International Technology Roadmap for Semiconductors. Clearly, transistor size reduction allows to package more transistors on the same chip area. A prediction made in 1965 by Gordon Moore, co-founder of Intel, stated that the number of transistors on integrated circuits will double every year. In 1975, when proved correct, he revised his prediction to double roughly every two years, what now is popularly known as *Moore's Law*. It can be appreciated as the red line in figure 1.1. Performance gains are usually attributed to *Moore's Law*, which is understandable because on smaller transistors electrons need to travel less distance and one expects the transistors to switch at higher speeds. However, that attribution is not quite accurate, as we can observe in figure 1.1 how transistor count grows steady throughout the period whereas processor performance slowed down around 2005. Robert Dennard noted in 1974 that MOS transistors had very convenient scaling properties [5]. To put it simply: voltage and current is proportional to transistor size, thus as transistors shrunk, also did necessary voltage and current. The scaling properties given on the cited paper, known as *Dennard scaling* rule, motivated manufacturers to increase clock frequency from one generation to the next, gaining serial performance without significantly increasing overall circuit power consumption until the year 2004, when the rule broke down. It is known that semiconductors require a certain threshold voltage to function, so eventually voltage could not be decreased any further. Also, Dennard ignored the *leakage current*, a quantum phenomenon where current leaks through transistors even when they are turned off. This is another source of power dissipation in microprocessors that constitutes *static power*, whereas the power considered originally by Dennard is instead *dynamic power* which is actually the power used to repeatedly switch the transistors. Manufacturers found that, for sizes below 65 nm, there is an exponential growth of leakage current, so at this scale *static power* begins to dominate power consumption, hence representing a new challenge to ordinary microprocessor design [6].[1] Associated with power consumption is heat dissipation, which becomes an important economic factor due to the need of implementing appropriate cooling solutions. In the post *Dennard scaling* era, the term "Power Wall" was coined to refer to the current inability of scaling down designs without exceeding practical affordable levels of power consumption. The situation is exacerbated on mobile platforms like smartphones. In contrast, while *Dennard scaling* was already considered dead after 2004, various technological advances were devised to keep Moore's Law still alive. An example is the high-k metal gate technology used by Intel for 45 nm and 32 nm fabrication processes circa 2010 [7]. However, it is generally accepted that this miniaturization trend cannot go on forever, and Moore's Law is currently slowing down.[2] What can be concluded from all this, is that Moore's prediction does not always translated into performance gains and that it was specifically Dennard scaling breakdown what prompted a greater focus on multicore processors from 2005 onwards.

[1]Overall power consumption is modeled by equation $P = \alpha\, C\, V^2 f + V\, I_{leak}$ where the first term is the dynamic power lost from transistor switching and the second term is the static power lost due to leakage current. When dynamic power is the dominant source of power consumption, which is the case above 65 nm, the equation can be conveniently approximated by the first term, i.e. $P \approx \alpha\, C\, V^2 f$.

[2]Although, it is certainly possibly to still add transistors in the third dimension, this are called 3D silicon Circuits.

1.1 Motivation

The reasonable move for the (desktop) computer industry was to embrace parallelism through multicore processors. Before this, the increases in clock rates allowed to implement optimizations at the processor level such as instruction-level parallelism, out-of-order execution or Hyper Threading, so that single-threaded programs executed faster on newer processors with no modification needed from the programmer. But now, programmers are faced with the task to exploit *explicit* parallelism. Research in auto parallelizing compilers tried to alleviate this.[3] A compiler can try to extract the required parallelism from a program by applying loop transformations, but this only can be done to a very limited extent. Also, the parallel version of a sequential algorithm can be a very different algorithm, as we will see later.

This paradigm shift puts software engineering in front of the challenging task of providing appropriate tools for effectively building software that correctly and efficiently exploits parallel processing power. Besides the well known pitfalls of concurrent programming, such as deadlocks, livelocks and data races, which are the cause of numerous bugs, developing software for multicore hardware demands taking care of different execution models, be aware of certain hardware characteristics and integrating legacy code which cannot always be easily or simply rewritten from scratch. This complexity makes engineering correct and efficient parallel software to require a high degree of expertise.

In [8], Pérez and Yovine argued that a high-level platform agnostic approach based on design patterns (a.k.a. algorithmic skeletons) would help to develop correct and efficient parallel software in a cost-effective and productive way. However, they also noted that most existent approaches to pattern based design lack formal semantics, which undermines the correctness aspect. To that end, they proposed the PCR parallel programming pattern, for which they gave a semantics based on the FXML formalism [9] and showed how it could be implemented in the more concrete execution model CnC [10]. They were able to support through empirical evidence that it was possible to generate performant

[3]Actually, research in this area precedes the multicore era from at least three decades.

parallel code based on the mentioned approach. Regarding correctness, they first noted that PCRs behaved as functions in the FXML semantics and then they proved the CnC implementation preserves the functional behaviour of PCRs. So, they concentrated in proving the transformations made by a prototype tool are correct, but they were not concerned with proving the correctness of the high level PCR itself.

In this thesis, we seek to provide a formal framework to

- Express high level PCR designs and prove their functional correctness in the sense that their parallel computation computes a given mathematical function.
- Be able to formally relate different PCR designs. In particular, that a PCR with more parallelism implements another PCR with less parallelism, i.e. the former is a refinement of the latter.

For practical reasons, we wish to use off the shelf tools for mechanical verification of the mentioned properties. In this sense, it is especially desired to have some form of automated verification available.

1.2 Structure of this thesis

The present document is structured in six chapters, including the present one. The other chapters are as follows:

- In chapter 2, we introduce some of the fundamentals of parallel algorithm design by way of the pattern-based approach. This will include, in particular, an informal introduction to the PCR pattern which will be our main concern thereafter.
- In chapter 3, we present an abstract model for the PCR pattern which will serve as a rigorous conceptual basis to analyze PCRs. Various concrete PCR examples will be presented and discussed. One of the main motivations here is that if PCRs are intended to behave as functions, then we want to know exactly which are those functions, thus having a mathematical reference for the correctness of the PCR.

- In chapter 4, with the goal of formalization in the horizon, we present a historical and theoretical introduction to the TLA$^+$ formal specification language. We will emphasize how the formalism allows to describe the dynamic and static aspects of the system being described. The available tools for TLA$^+$ will be also discussed.

- In chapter 5, we put the theory and tools of chapter 4 to practice in formalizing the abstract models proposed in chapter 3, this includes both their operational concurrent semantics and the functions associated to them. Within the TLA$^+$ framework, mechanical verification will assists us to prove that the operational semantics is correct with respect to the expected mathematical functions.

- In chapter 6, we present some general conclusions of our work.

Chapter 2

Parallel algorithms fundamentals

Likewise, when a long series of identical computations is to be performed, such as those required for the formation of numerical tables, the machine can be brought into play so as to give several results at the same time, which will greatly abridge the whole amount of the processes.

General L. F. Manabrea (1842)

In the introduction we observed how around 2005 the limitations related to the physics of processor design have made hardware industry shift from improving the speed of a single processor to increasing the number of processing core units. As a result, in the past decade programmers were required to turn their attention to parallelism. But we want to start noting that parallelism itself is not something new. According to Snyder [11], the epigraph in this chapter where General Manabrea refers to a design option for Babbage's Analytical Engine, is the earliest known reference to parallelism in computer design. He also conjectures that Manabrea understood what he calls the *Fundamental law of Paralell Computation*:

> A parallel solution utilizing p processors can improve the best sequential solution by at most a factor of p.

which follows from the observation that a speedup greater than a factor of p would imply the existence of a better sequential solution. He then goes to note how this upper limit implies that parallel computation offers only a *modest potential* benefit on common problems of scientific interest that are typically superlinear (e.g. in the range $O(n^2)$ to $O(n^4)$), and what is even more concerning, to achieve that "modest" benefit is not easy in practice. Yet, parallel computation is considered one of the most promising ways to improve computer performance.

Parallelism was already "hot" decades before 2005, but in the area of high performance computing. And a lot of that knowledge has been transferred to the more ample general purpose computing. In fact, the benefits of parallelism was controversial in its inception. The controversy started with an observation of Gene Amdhal [12]:

> "For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers in such a manner as to permit cooperative solution. ... A fairly obvious conclusion which can be drawn at this point is that the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude."

His argument was later condensed in what is called Amdhal's Law, which states that as every task can be splitted in one part that can be parallelized and another part that is inherently serial, then the theoretical *speedup* (S) is always limited by the part of the task that cannot benefit from the parallel improvement. That is:[1]

$$S(p) \;\leq\; \frac{1}{\sigma + \frac{1-\sigma}{p}}$$

where p is the number of processing units and σ is the serial fraction of the total work. So, if we have infinite processing units, the theoretical *speedup* is limited by the inverse of the serial fraction:

$$S(\infty) \;\leq\; \lim_{p\to\infty} S(p) \;=\; \frac{1}{\sigma}$$

For roughly two decades, Amdhal's Law was unquestioned being the only well known criterion to understand parallel systems performance, until 1988 when Gustafson et al published their own results questioning Amdhal's assumptions [14]. They noted that Amdhal presupposed a fixed size problem, but if we let the problem size to grow and with this the the serial fraction diminishes, then speedup grows indefinitely. So, from their point of view, Amdhal is pessimistic. They argue:

[1]According to our research, what is possibly the first mathematical formulation of the law can be found in [13].

> "One does not take a fixed-sized problem and run it on various numbers of processors except when doing academic research; in practice, the problem size scales with the number of processors. ... speedup should be measured by scaling the problem to the number of processors, not by fixing problem size."

Then, what is known as Gustafson's Law, is formulated as:

$$S(p) \;\leq\; \sigma + p \cdot (1-\sigma) \;=\; p + \sigma \cdot (1-p)$$

For infinite processing units, we have:

$$S(\infty) \;\leq\; \lim_{p \to \infty} S(p) \;=\; \infty$$

What Gustafson et al are interested to measure is more appropriately called scaleup/-sizeup, instead of speedup. So, is important to consider here that Amdhal focused on *how faster we can solve a problem with the same size*, whereas Gustafson et al focused on *how bigger is the problem we can solve in the same time*. The former is called *strong scalability* and the latter is called *weak scalability*. Both are valid points of view.

In this thesis, we will sometimes made use of the Work-Span Model, as is introduced in textbook [2], to give a theoretical measure of speedup. This is defined as

$$S \;=\; \frac{T_1}{T_\infty}$$

where

- T_1 is *work*: the total time that a serialization of the algorithm require.
- T_∞ is *span*: the total time a parallel algorithm would take on an ideal machine with an infinite number of processing units. This can be thought as the longest path of serial computation, a.k.a. the *critical* path.

In particular, we will express work and span in terms of asymptotic complexities.[2] Speedup in this form is usually interpreted as the *degree of parallelism*, and one is typically interested in maximizing it. There are useful results that allow in practice to estimate the speedup for a particular number of processing units, under some reasonable assumptions, from T_1 and T_∞, but in this thesis we will always assume the idealized view. Also, we are not concerned with low level performance optimization.

[2]More specifically, *big Theta* notation $\Theta(f(N))$ denotes the set of all functions $g(N)$ such that there

2.1 Parallel Design patterns

In general, patterns provide a common vocabulary to solve problems and allows reuse of best practices. In this section we carry out a brief overview for some of the most common parallel design patters —in particular those at the foundation of the PCR pattern, which will be introduced in section 2.2. Primary sources for this material are textbooks [2] and [15].

2.1.1 Map

The *map* pattern applies a function to each element of a collection of elements. It is assumed that the function used in a map does not have side effects (i.e. it is a so called *pure function*) which allows all instances of the map to be executed in any order. This independence offers maximum concurrency.

In imperative languages, a map is usually expressed as a loop (`for` or similar) where all iterations are independent (illustrated by the left hand side of Listing 2.1). In functional languages, it is more directly expressed as the familiar higher order function `map` (illustrated by the right hand side of Listing 2.1). Some other common names of map are: *mapcar*, *lapply*, *transform*, *for-each*. On parallel programming platforms, such as Cilk [16] and others, the map pattern is often supported in the form of a special `parallel for` construct.

```
for $i$ = 1 to $n$ do
  $y_i$ := $f(x_i)$
```

```
map $f$ [$x_1, x_2, \ldots, x_n$] = [$f(x_1), f(x_2), \ldots, f(x_n)$]
```

Listing 2.1: Map over collection $x_1, x_2, ..., x_n$ with function f, expressed in imperative style (left) and functional/declarative style (right) pseudocode.

The ordinary `for` loop or the `map` function are expected to have a serial implementation (illustrated by the left hand side of Figure 2.1)[3]. For n elements, assuming the applied

exist positive constants c_1, c_2, and N_0 with $c_1 \cdot f(N) \leq g(N) \leq c_2 \cdot f(N)$ for $N \geq N_0$.

function f takes $\Theta(1)$ time on any element, the serial map takes $\Theta(n)$ time. In sharp contrast, the parallel implementation (illustrated by the right hand side of Figure 2.1) takes just $\Theta(1)$ time, given enough processing units. Thus, speedup is $\frac{\Theta(n)}{\Theta(1)} = \Theta(n)$ which is theoretically optimal.

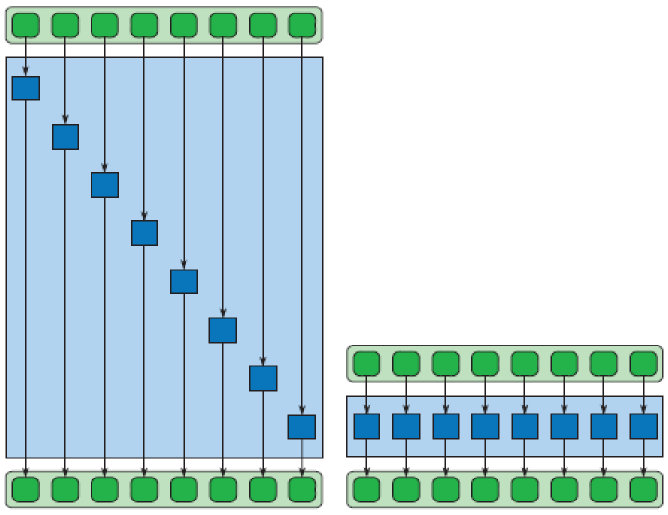


Figure 2.1: Serial and parallel implementations of the map pattern on 8 elements (source [2]).

The map pattern is commonly used to solve *embarrassingly parallel* problems: those problems that can be easily decomposed into independent subtasks requiring no communication/synchronization between the subtasks (except possibly at the end). Indeed, the optimal speedup mentioned before confirms this. Although map is a simple pattern, it is found at the foundation of many important applications.

This pattern can be combined with other parallel design patterns. As an important example, map combined with reduction (see 2.1.2) gives the *map-reduce* pattern, which became very popular with Google's implementation circa 2004 [18] and has since then constituted a key ingredient for efficiently processing enormous quantities of data. Also, other important design patterns arise as natural generalizations or extensions of map, such as *gather*, *scatter*, and *stencil*.

[3]It should be noted that even a not auto-parallellizing but decent compiler like GCC will try to take advantage of *instruction-level parallelism* (ILP) [17] to let the CPU operate on multiple elements at a time. But we are not concerned with that kind of parallelism.

2.1.2 Reduce

In the *reduce* pattern, a combiner function $f(x, y) = x \otimes y$ is used to combine a collection of elements to create a summary value. It is typically assumed that pairs of elements can be combined in *any* order, which allows for a variety of implementations.

In imperative languages, it is usually expressed as a loop (a `for` or similar) where every iteration accumulates a new result combining the previous result and the current element (illustrated by the left hand side of Listing 2.2). In functional languages, it is more directly expressed as the familiar higher order function `fold` (illustrated by the right hand side of Listing 2.2). Some other common names are: *reduce*, *accumulate*, *aggregate*.

```
$r$ := $r_0$
for $i$ = 1 to $n$ do
    $r$ := $r \otimes x_i$
```

```
fold $\otimes$ $r_0$ $[x_1, x_2, \ldots, x_n]$ = $((r_0 \otimes x_1) \otimes x_2) \otimes \ldots \otimes x_n$
```

Listing 2.2: Reduction (from left to right) over collection $x_1, x_2, ..., x_n$ with operation $\otimes$ and initial value r_0, expressed in imperative style (left) and functional/declarative style (right) pseudocode.

A reduction can be performed from left to right (as it is the case in Listing 2.2), from right to left, or in any other possible order of the elements. But the result is not necessarily the same —that depends on the combiner operation. Besides, the initial value can be omitted for reducing over non-empty collections.

The following algebraic properties of the combiner operation are desirable:

1. **Associativity**: $(x \otimes y) \otimes z \;=\; x \otimes (y \otimes z)$

   It allows to use any order of pairwise combinations as long as adjacent elements are intermediate sequences, that is, any *grouping* of the elements.
   Examples: addition and multiplication over integers, logical disjunction and conjunction, addition and multiplication over matrices, list concatenation.

2. **Commutativity**: $x \otimes y \;=\; y \otimes x$

It allows to combine any rearrangement of the elements.

Examples: all the previous associative examples except matrix multiplication and list concatenation.

3. **Identity**: $\texttt{id}_\otimes \otimes x \;=\; x \otimes \texttt{id}_\otimes \;=\; x$

   Existence of an identity element $\texttt{id}_\otimes$ is often useful to make the reduction of an empty collection meaningful.

   Examples: 0 and 1 are (integer) addition identity and multiplication identity, respectively. *False* and *True* are logical disjunction identity and conjunction identity, respectively.

Next, we review the relevance of these properties in more depth.

2.1.2.1 Associativity

Strictly speaking, only *associativity* is required for parallelization of reduce. Consider the reduction of the 8-element collection $x_0, x_1, ..., x_7$ and let r be the resulting value. Then observe that just by taking advantage of associativity we have:

$$\begin{aligned} r &= x_0 \otimes x_1 \otimes x_2 \otimes x_3 \otimes x_4 \otimes x_5 \otimes x_6 \otimes x_7 \\ &= (((((((x_0 \otimes x_1) \otimes x_2) \otimes x_3) \otimes x_4) \otimes x_5) \otimes x_6) \otimes x_7) \qquad (2.1) \\ &= (((x_0 \otimes x_1) \otimes (x_2 \otimes x_3)) \otimes ((x_4 \otimes x_5) \otimes (x_6 \otimes x_7))) \qquad (2.2) \end{aligned}$$

Grouping 2.1 corresponds to a serial reduction from left to right (illustrated by the left hand side of Figure 2.2), while grouping 2.2 corresponds to a parallel binary tree reduction (illustrated by the right hand side of Figure 2.2). The serial version obviously requires 7 steps of computation to reduce the 8 elements. For the parallel version, note that four processing units can be used first to compute partial values $r_1 = x_0 \otimes x_1$, $r_2 = x_2 \otimes x_3$, $r_3 = x_4 \otimes x_5$ and $r_4 = x_6 \otimes x_7$. Then, two processing units can be used to compute partial values $r_5 = r_1 \otimes r_2$ and $r_6 = r_3 \otimes r_4$. Finally, one processing unit can be used to compute the final result $r = r_7 = r_5 \otimes r_6$. Thus, four processing units can be used to compute the result in just $\log_2 8 = 3$ steps instead of the 7 steps required for the serial version.

In general, for n elements and assuming the combiner operation takes constant time, serial reduction takes $\Theta(n)$ time and parallel reduction takes $\Theta(\log_2 n)$ time. Thus, speedup is $\frac{\Theta(n)}{\Theta(\log_2 n)}$ which is very good. Interestingly, note that both implementations perform the same amount of *work*. In our example, the serial version performs 7 operations, and the parallel version also 7 operations overall (although some of them in parallel).

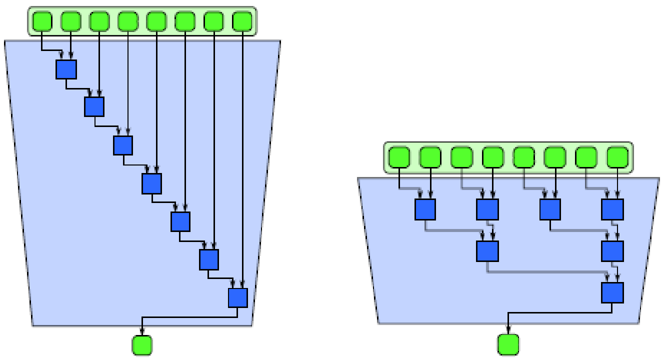


Figure 2.2: Serial and parallel implementations of the reduce pattern on 8 elements (source [2]).

However, it should be noted that parallel reduction requires $\Theta(n)$ space for the intermediate partial values, while serial reduction requires just $\Theta(1)$ space. For this reason, in practice, parallel reduction is not always preferable.

2.1.2.2 Commutativity

As we saw previously, *associativity* allows to work on any *grouping* and this suffices for parallel reduction. But then, what about *commutativity*? Any *grouping* does not mean any *ordering*, for which commutativity is required as it is this property that allows to work on any rearrangement of the elements, thus giving more freedom for implementing reduction. For example, to *vectorize* reduction on a processor with two-way SIMD [19] instructions a reordering could be needed, this technique enables the processor to inherently exploit the locality of data leading to a potential performance impact[4].

[4] As of 2016 most commodity CPUs implement architectures that feature SIMD instructions for vector processing on multiple (vectorized) data sets. In case of Intel x86's, these are the MMX, SSE and AVX instructions.

In fact, most justifications on why commutativity is useful are concerned with low level platform details as in the aforementioned example about *vectorization*. For a more higher level view, consider the reduction $x_0 \otimes x_1 \otimes x_2 \otimes x_3 \otimes x_4$. By associativity, it is possible for two processing units to compute partial values $r_1 = x_0 \otimes x_1$ and $r_2 = x_2 \otimes x_3$ in parallel, but if r_1 finishes before r_2 it would be desirable to compute the partial value $r_3 = r_1 \otimes x_4$ instead of waiting for r_2. If we proceed in this way, the final result is actually the regrouping and rearrangement $((x_0 \otimes x_1) \otimes x_4) \otimes (x_2 \otimes x_3)$, which would possibly be incorrect if $\otimes$ is not commutative.

So, some optimization techniques take advantage of commutativity and, clearly, users would like parallel platforms (compilers, frameworks, libraries, API's, etc.) to produce/implement highly optimized code. As a consequence, it is common for implementers of parallel platforms to actually *assume* commutativity on the combiner operation. For example, Listing 2.3 shows the scalar/dot product of two compatible vectors v and u in OpenMP [20], an API tailored for shared memory multiprocessing applications. This computation involves a sum reduction.

```
#pragma omp parallel for reduction(+: sum)
for (size_t i = 0; i < N; i++)
  sum += v[i]*u[i];
```

Listing 2.3: Scalar/dot product in OpenMP (C++).

The `reduction` clause in OpenMP assumes the operation is associative and also commutative, which in this case is obvious for the pre-defined operation +. Clearly, there are operations known to be not commutative (e.g. matrix multiplication), but most importantly, for custom user-defined operations is not always obvious if commutativity holds or not. Also, the user cannot expect the compiler to issue a warning if it is not, as the reducer *commutativity problem*[5] in general should be undecidable due to Rice's theorem (although for specific cases this negative claim could be overcome [21]). So, in this context, the user should be cautious because for a non-commutative operation the final result is not guaranteed to be deterministic and this is rarely what the user expects. This is a pervasive issue; research in [22], with the suggestive title of *Nondeterminism in MapReduce Considered Harmful?*, conducted an empirical study to determine how common are non-commutative reducers in real-world Map-Reduce programs, if non-commutative reducers are always harmful, and even if they should be marked as bugs.

In contrast, the OpenMPI library [23], an implementation of the Message Passing Interface (MPI) for distributed computing applications, allows the user to define a custom combiner operation (assumed at least associative) with a flag to explicitly indicate if it is commutative or not [24]. If it is set as False, then order of operands is fixed in a specific order, but is still possible for the library to take advantage of the associativity of the operation.

2.1.2.3 Identity

As briefly commented earlier, the identity element $\mathtt{id}_{\otimes}$ for a combiner operation $\otimes$ is often useful to make the reduction of an empty collection meaningful, which also facilitates dealing with boundary conditions in algorithms.

It is often convenient to interpret the identity value also as the initial value of the reduction, which is the result if the collection is empty. For example, in the *stream* aggregate operators of Java SDK 8, documentation for the `Stream.reduce` method states: "The identity element is both the initial value of the reduction and the default result if there are no elements in the stream" [25]. However, it is conceivable to have a different implementation that accepts an arbitrary value, possible different to the identity, to be returned whenever the collection is empty, which is otherwise not used at all in the reduction. Moreover, when the intention is to work only on non-empty collections there is clearly no need for identity.

Also, it should be noted that an identity element does not always exist. For example, consider the binary operation

$$\max(x, y) = \begin{cases} y & , \ x \leq y \\ x & , \ x > y \end{cases}$$

[5]The *commutativity problem* of reducers asks if the output of a reducer is independent of the order of its inputs. This problem, in the context of the Map-Reduce model, is addressed in [21].

It is associative and even commutative over $\mathbb{Z}$. Now, for a fixed size representation of integers, say 32-bit *two's complement*, the representation range is $-(2^{32-1})$ to $2^{32-1}-1$, so the identity should be $-(2^{32-1})$ as clearly for any integer x in this range we have:

$$\max(-(2^{32-1}), x) \;=\; \max(x, -(2^{32-1})) \;=\; x$$

But over arbitrary large integers, there is no identity, for suppose there is an $e \in \mathbb{Z}$ so that $\max(x, e) = x$ for any $x \in \mathbb{Z}$, then $\max(e-1, e) = e$ contradicting our assumption. It just happens that there is no integer element which is smaller than each possible integer.

In this cases, with a little care, the domain of interest can be augmented with a *fictitious* identity. In the case of $\mathbb{Z}$, we can add a special element $-\infty$ and require

$$\max(-\infty, x) \;=\; \max(x, -\infty) \;=\; x$$

to hold for any $x \in \mathbb{Z} \cup \{-\infty\}$. An analogous situation occurs for the operation $\min(x, y)$ and special element ∞. In fact, this technique is always applicable, even when an identity already exists in the domain of interest.

2.2 A primer on the PCR Pattern: Producer, Consumer and Reducer

Stepping on map and reduce patterns, Pérez and Yovine [8] (also [26]), propose a platform-agnostic parallel programming pattern, called PCR, which describes computations performed concurrently by communicating *Producers*, *Consumers* and *Reducers*, each one being either a basic function (business logic) or a nested PCR.

We present in this section an informal explanation of the PCR pattern and illustrate it with a first example. Then, chapter 3 will be devoted to provide a more rigorous basis for the pattern, which in turn will be later formalized in the TLA$^+$ specification language.

2.2.1 High level overview

The PCR pattern aims at expressing computations consisting of a *producer* consuming input data items and generating, for each one of them, possibly many outputs which are consumed by several *consumers* working in parallel. Their outputs are finally aggregated into a single result by a *reducer*. The goal of the pattern is to emphasize the independence between the different computations in order to expose all parallelization opportunities.

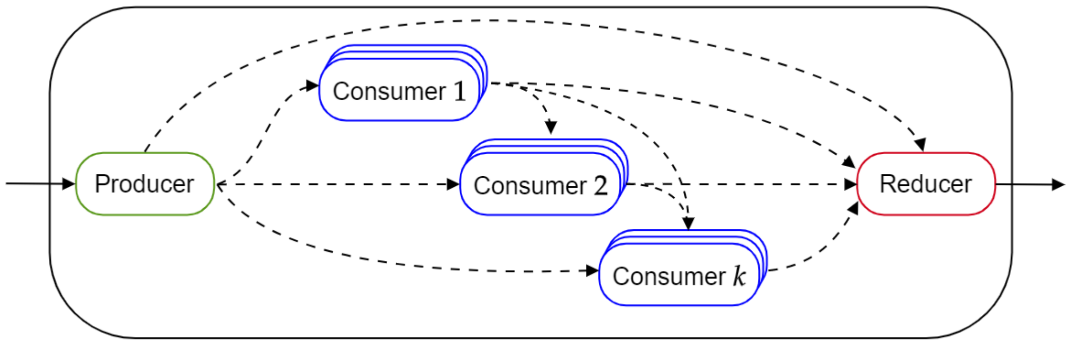


Figure 2.3: Pictorial view of the PCR pattern.

Figure 2.3 depicts the general form of a PCR. Arrows represent data connections in a PCR. Full ones model the external input source and the output channel to the external environment. The external input is available to any inner component. Dashed arrows denote internal data channels. Data cycles between internal components are not allowed: the network is itself a directed acyclic graph (DAG) of which any topological sorting has the producer and the reducer as as the first and the last items, respectively.

PCRs are to be understood as functions, something we will further study in the next chapter on a more rigorous basis.

2.2.1.1 Data flow

Information 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. *Consumer* components read values from the outer scope and from the private data channels to perform their computations.

3. At the end, a *reducer* component combines values from one or more data sources coming from the producer and one or more consumers, generating a single output item for every external input item processed by the producer.

Reads in data channels are non destructive; the same value can be read by any consumer and by the reducer.

2.2.1.2 Concurrency

Producer, consumers, and reducer work in parallel subject to the existing data dependencies: all input items must be available for a producer, consumer or reducer instance in order to perform its calculation. Each producer, consumer and reducer can potentially spawn as many parallel execution instances as necessary for any specific workload. Both the nature of an execution instance (local and/or remote thread or process) and the scheduling policy are assumed to be defined by each PCR underlying implementation.

2.2.1.3 Relation with other patterns

PCRs combine several parallel programming concepts.

- Both the PCR as a whole and each internal consumer as a unit can be regarded as a *parallel map* operation transforming the input stream.
- Putting several consumers in sequence conforms a *parallel pipeline* with each consumer as a worker.
- Depending on which implementation is chosen and on the nature of the computation, the reducer could perform a *parallel reduce* operation.

- The sequence of first the producer output triggering the spawning of consumer instances, and then the aggregation of consumers' outputs by the reducer can be regarded as an instance of the *Fork/Join* model, having the reducer as the synchronization point.

- PCRs combine *collective* operations into a single composable high-level pattern. The distribution of the producer output to instances of the same consumer is a *scatter* operation; availability of the same produced item to all consumers is analogous to a *broadcast*; the combination of all inputs by the reducer is a *gather* operation.

- The reducer could finish its computation before all its input items are processed or even before the producer or the consumer instances finish their work, enabling for so called *eureka* computations.

2.2.2 Example: counting the first prime Fibonacci numbers

Let us consider the problem of counting the prime numbers among the first N Fibonacci numbers. Figure 2.4 illustrates pictorially how two PCRs may cooperate in a compositional fashion to resolve this problem.

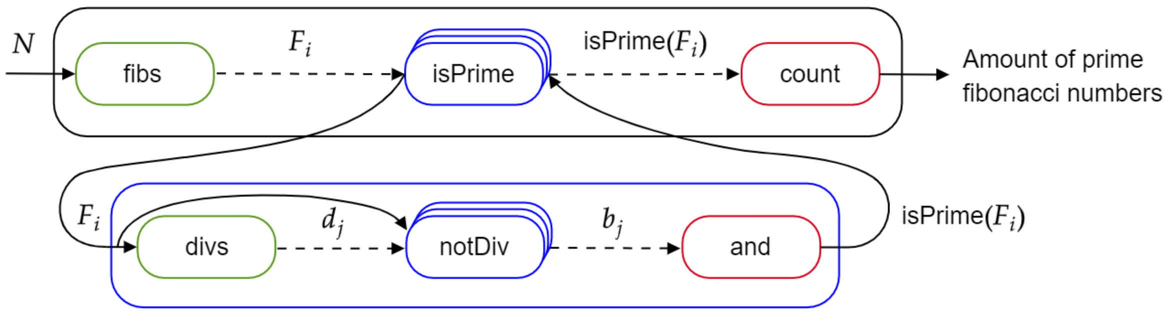


Figure 2.4: PCR solution for counting Fibonacci primes.

Following the structure given in the figure, the PCR solution is intended to work as follows:

1. At the *outer* PCR, the producer `fibs` generates the sequence F_1, F_2, ..., F_N of

Fibonacci numbers.

2. For each F_i, an independent instance of consumer `isPrime` can check, in parallel, its primality generating a boolean result `isPrime(`F_i`)`. This constitutes what Pérez calls a *nesting opportunity*, in which another *inner* PCR may paralellize the sub-task of primality testing as follows:

   2.1. The producer `divs` generates all possible divisors d_j of F_i.

   2.2. For each d_j, an independent instance of consumer `notDiv` can check, in parallel, the (non-)divisibility of F_i by d_j generating a boolean result b_j. This is an example of a consumer reading the producer output and the PCR input as well.

   2.3. The reducer `and` computes the result `isPrime(`F_i`)` as the conjunction of all the consumer outputs b_j.

3. The reducer `count` counts the number of consumer outputs `isPrime(`F_i`)` which are true.

This solution admits parallel execution at several levels. Many instances of `isPrime` could be executed simultaneously as allowed by the available processing units and the F_i production rate. Besides, each instance may be computed by an inner PCR with its own parallel capabilities. Since the `count` and `and` reduce operations are associative (and commutative), they can also be parallelized.

Taking this example further, the consumer `notDiv` can also be considered as a nesting opportunity where another PCR may check divisibility in parallel, although performance-wise this would possibly only make sense for very large numbers.

Chapter 3

Abstract model for the PCR pattern

My hypothesis is that we can solve the software crisis in parallel computing, but only if we work from the algorithm down to the hardware — not the traditional hardware first mentality.

Tim Mattson

Syntax and semantics for the PCR pattern were given by Pérez and Yovine in [8] using FXML, a theoretical formal language that uses partial orders for interpreting parallel computations [9]. Now, FXML does not at the moment possess any associated tools for mechanical verification that can be applied to check e.g. refinements of parallel schemata. Starting from this chapter, we wish to build up from a simpler conceptual foundation and then formalize the PCR pattern using a well established formal tool with support for mechanical verification, namely TLA$^+$. To this end, we introduce an alternative abstract model for PCR computations, as well as an associated syntax. This model should accomplish the following:

- Be faithful to the informal description of the PCR pattern.
- Serve as a rigorous basis (albeit still informal) to talk about PCRs.
- Be simple enough to abstract from details like platform architecture, interleaving vs no-interleaving, task scheduling, or reducer implementation strategies.

The model will be introduced following a case by case basis in the form of general *schemes* serving as algorithmic skeletons. The main focus in this chapter is to conduct a study on the functional behaviour (i.e. the "What") of PCRs. More precisely, PCRs will be identified with certain functions which will serve as a correctness criterion. Later, in chapter 5, the model shall be formalized in the formal specification language TLA$^+$, which will yield a kind of operational semantics for the parallel behaviour of PCRs (i.e. their "How"). There, within the TLA$^+$ framework, it will be formally proven that the "How" matches the "What" as a correctness (and refinement) property.

3.1 The basic PCR

In our model, the PCR computation is assumed to be governed by an *iteration space*, a (possibly input dependent) set of indices (natural numbers) on which the internal components (the producer, consumers and reducer) should act. More precisely, the iteration space indexes the data produced and consumed by the components of the PCR. Let us for the sake of exemplification, consider again the PCR depicted in the precedent chapter. Its input is a number N, and the number of primes among the first N Fibonacci numbers is to be returned. Then we use as iteration space the numbers between 1 and N.

The result of each internal component operation at every index is written (assigned) to an associated *output variable*. More specifically, output variables for the producer and consumers describe the *full history* of assignments over the iteration space, modeling the internal data channels of the PCR pattern. We thus think of these variables as *streams* of values mathematically represented by partial functions $\mathbb{N} \to T$, abbr. $\overrightarrow{T}$, for some range type T. If v is a stream variable, we denote the i-th element by super-script notation v^i and $wrt(v^i)$ denotes the condition that v^i has been *written*. Writing to a stream variable may happen in any order. The stream-like nature of variables can be leveraged into a syntactic mechanism which allows stream operations *look-ahead/look-behind* to be used on variables by indexing. For example, to compute the i-th Fibonacci number it is useful (performance-wise) to have access to the previous computed values at $i-1$ and $i-2$. The syntactic matters are discussed in the next sections.

In what follows, we present a general scheme for the *basic* PCR, where the behaviour of each internal component is assumed to be given by a *basic function*, i.e. user provided functions implemented in some host language.

Definition 3.1 (Basic PCR scheme)**.** Let $\mathcal{A}$ be a basic PCR with k consumers. The computation of $\mathcal{A}$ over input x consists in the following operations (assignments) for each $i \in I_x$:

$$\begin{array}{rrcl}
\textbf{Producer}: & p^i & := & f_p(x, p, i) \\
\textbf{Consumer } 1: & c_1^i & := & f_{c_1}(x, p, i) \\
\textbf{Consumer } 2: & c_2^i & := & f_{c_2}(x, p, c_1, i) \\
\vdots & & \vdots & \\
\textbf{Consumer } k: & c_k^i & := & f_{c_k}(x, p, c_1, c_2, \ldots, c_{k-1}, i) \\
\textbf{Reducer}: & r & := & r \otimes f_r(x, p, c_1, \ldots, c_k, i)
\end{array}$$

where:

- $I_x \subseteq \mathbb{N}$ is an index set representing the *iteration space*. In general, it depends on input x and has the form:

  $$I_x \;=\; \{i \in \mathit{lBnd}(x)..\mathit{uBnd}(x) : \mathit{prop}(i)\}$$

  where *lBnd* and *uBnd* denote its lower bound and the upper bound respectively, and *prop* is a truth condition acting as filter.

- p, c_1, ..., c_{k-1} and c_k are variables representing the *histories* of values computed by producer and consumers. These are the output of producer and consumers.

- f_p, f_{c_1}, ..., f_{c_k} and f_r are *basic functions* associated with producer, consumers and reducer.

- $\otimes$ is a binary combiner operation. r is a variable representing the (partial) combined value, the output of the reducer. The initial value is given by $r_0(x)$, and sometimes we refer to it as r_0.

Except when explicitly stated otherwise, we assume the following:

1. I_x is finite.

2. The combiner $\otimes$ is an associative and commutative operation.

3. $\otimes$ has an identity element, namely $\mathtt{id}_\otimes$, so that $r_0(x) = \mathtt{id}_\otimes$ and this is also the default value in case the iteration space is empty (i.e. $I_x = \varnothing$).

Remark 3.1.

1. All assignments, basic functions and reduce operations are assumed to be atomic.

2. Producer and consumers are considered optional, which can be seen as though they are completely absent or as they are trivial identity functions just passing the values. The presence of the reducer is mandatory.

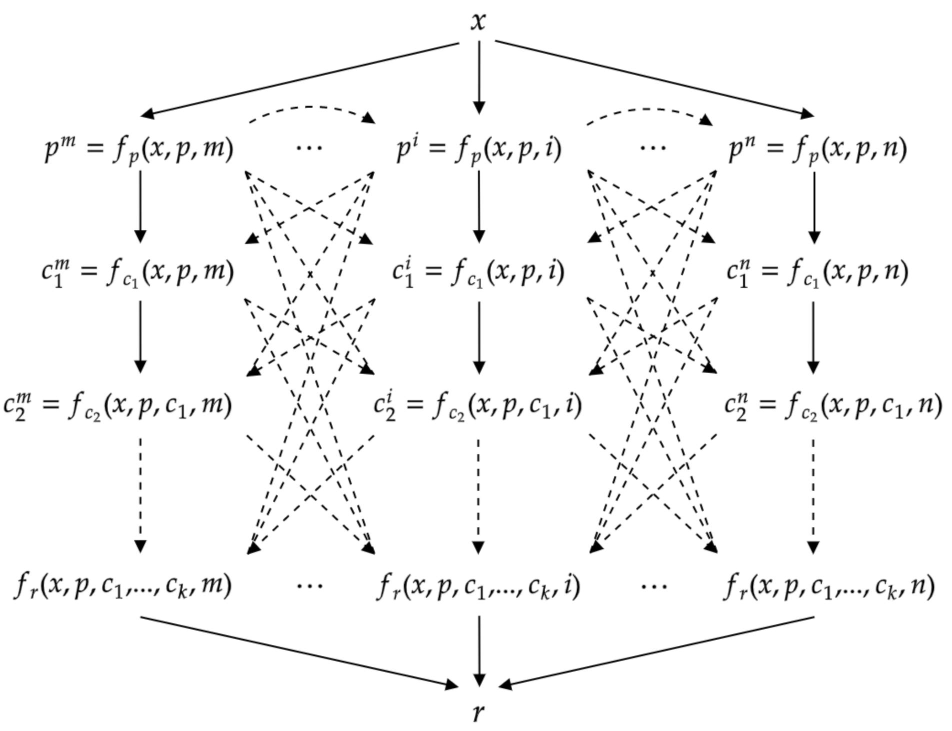
3. Notice that, unlike the other components output variables, r is a scalar, i.e. not a stream variable, and the reduction order with respect to I_x is unspecified.

4. Basic function f_r is used as a last (generally simple) transformation prior to the combiner work done at the reducer. It is not essential, in the sense that it could be separated to a previous consumer as follows:

$$\begin{aligned} c^i_{k+1} &:= f_r(x, p, c_1, c_2, \ldots, c_k, i) \\ r &:= r \otimes c^i_{k+1} \end{aligned}$$

In general, every basic function may consume values from previous output variables, which naturally induces a (strict) partial order representing *data dependencies* between PCR operations. It is convenient to visualize this order as a dependence graph (more precisely a DAG) with input x as source and reducer output variable r as sink, as generically illustrated in figure 3.1. These graphs will be useful tools to reason about PCRs. Here we think of input x as a fixed constant but later, when discussing composition between PCRs, it will be more generally treated as another stream-like variable. Note that the bottom arrows flowing into the sink, in fact, do not represent data dependencies, as f_r is just a transformation made as part of the reducer operation. In upcoming dependence graphs, we will omit f_r if convenient.

By definition 3.1, a basic PCR with k consumers should perform a total of $(k+2)\cdot \mid I_x \mid$ operations. Intuitively speaking, independent operations can effectively be performed in parallel (assuming enough processing units), unlike dependent operations that are normally considered to be performed in series. We denote by $red(i)$ the condition that the reduce operation has been performed for the value of i in question.



x
$p^m = f_p(x,p,m)$
$\cdots$
$p^i = f_p(x,p,i)$
$\cdots$
$p^n = f_p(x,p,n)$
$c_1^m = f_{c_1}(x,p,m)$
$c_1^i = f_{c_1}(x,p,i)$
$c_1^n = f_{c_1}(x,p,n)$
$c_2^m = f_{c_2}(x,p,c_1,m)$
$c_2^i = f_{c_2}(x,p,c_1,i)$
$c_2^n = f_{c_2}(x,p,c_1,n)$
$f_r(x,p,c_1,\ldots,c_k,m)$
$\cdots$
$f_r(x,p,c_1,\ldots,c_k,i)$
$\cdots$
$f_r(x,p,c_1,\ldots,c_k,n)$
r

Figure 3.1: Dependence graph for a basic PCR where $m = min(I_x)$ and $n = max(I_x)$.

Definition 3.2 (Basic PCR Termination)**.** A basic PCR $\mathcal{A}$ *terminates* at input x if $red(i)$ holds for all $i \in I_x$ (i.e. a total of $\mid I_x \mid$ reductions have been made). We shall write $end_{\mathcal{A}}$ for this condition.

From the given definitions, it should be clear that the PCR $\mathcal{A}$'s *output* is the value of r when $end_{\mathcal{A}}$ holds.

3.1.1 Basic syntax: *produce*, *consume* and *reduce*

It will be convenient to describe PCRs, especially the concrete ones, in a more user-friendly style by means of pseudocode[1]. There are three principal primitives: **`produce`**, **`consume`** and **`reduce`**, presented in table 3.1, in direct correspondence with the PCR operations according to our model. This closely follows the PCR specification language presented by Pérez and Yovine in [8] but with the small addendum of explicit syntax for the specification of the iteration space. In particular, we allow here a lower bound and a filter predicate for defining the iteration space. Another primitive called **iterate** shall be presented later in section 3.4 as an extension of the PCR model.

Type	**Syntax**	
Basic function	**fun** $f(x_1, x_2, \ldots, x_n)$	
Iteration space	**lbnd** $\mathcal{A}$ = $\lambda x.\ e$ // default: 0 **ubnd** $\mathcal{A}$ = $\lambda x.\ e$ // default: ∞ **prop** $\mathcal{A}$ = $\lambda i.\ e$ // default: *True*	where $\mathcal{A}$ is the PCR's name and **lbnd**, **ubnd**, **prop** are the lower bound, upper bound and filter functions respectively. Some default values are assumed in case of absence of the corresponding declaration.
Producer	p = **produce** f_p x p	where f_p is a basic function, and x is the PCR's input variable
Consumer	c_e = **consume** f_{c_e} x p c_1 ... c_{e-1}	where f_{c_e} is a basic function ($1 \leq e \leq k$), x is the PCR's input variable, p is the producer's output variable, and c_1, ..., c_{e-1} are the previous consumers' output variables.
Reducer	r = **reduce** $\otimes$ $(r_0\ x)$ $(f_r\ x\ p\ c_1\ \ldots\ c_{k-1})$	where $\otimes$ is the combiner operation (with initial value r_0) and f_r is a basic function on PCR input and producer/consumer outputs.

Table 3.1: Basic PCR syntax.

Listing 3.1 presents the basic PCR scheme of definition 3.1 but in syntactic form according to table 3.1.

```
fun f_p(x,p,i) = ...
fun f_c1(x,p,i) = ...
fun f_c2(x,p,c1,i) = ...
 ⋮
fun f_ck(x,p,c1,c2,...,c_{k-1},i) = ...
fun f_r(x,p,c1,...,ck,i) = ...
fun r0(x) = ...

lbnd A = λx. ...
ubnd A = λx. ...
prop A = λi. ...

PCR A(x)
```

[1]In this thesis, we do not bother with a precise BNF definition because it is a very simple syntax directly representing the abstract model, and we are not assuming any particular host language where the basic functions are implemented.

```
par
  p  = produce f_p x p
  c_1 = consume f_c1 x p
  c_2 = consume f_c2 x p c_1
    ⋮
  c_k = consume f_ck x p c_1 c_2 ... c_{k-1}
  r  = reduce ⊗ (r_0 x) (f_r x p c_1 ... c_k)
```

Listing 3.1: Basic PCR scheme in syntactic form.

In PCR syntax, for any producer/consumer stream variable `v`, `v[0]` refers to the value at the current iteration index i (i.e. v^i) and this can be more conveniently written just as `v` [2]. Look behind/ahead syntax is presented in the next section. Nevertheless, basic functions have optional access to index parameter i if they need to act upon its value. In this chapter we hope to use intuitively clear programming constructions when presenting basic functions as code, having more preference for a functional/declarative style. For most of our discussions we will not dwell on the host language's syntax/semantic specifics.

The combiner operation can be regarded as a non-binary *basic function* but with restricted form in the following manner:

```
fun op(r,x,p,c_1,...,c_k,i) = r ⊗ f_r(x,p,c_1,...,c_k,i)
 ⋮
r = reduce op (r_0 x) x p c_1 ... c_k
```

In the present work we only consider *finite* iteration spaces, so we always require at least the upper bound specification.

Example 3.1.1. IsPrime1 is a *basic* PCR with a single consumer. For an input number N, IsPrime1 returns a boolean value indicating if N is a prime number. It works as follows:

1. The producer `divs` generates all the possible divisors of the input number N.[3] It is a fact that for any composite number N *some* of its divisors different from 1 must be less or equal than $\sqrt{N}$. So, it is enough to test divisibility in the range $2..\lfloor\sqrt{N}\rfloor$, since if we had a divisor greater than $\sqrt{N}$ it would have a smaller one that we would

[2]Whenever needed, syntax `v[0]` can be used to make it clear we are specifically referring to the value v^i, as opposed to the complete history of values v.

have already verified.

2. Each instance i of the `notDiv` consumer checks, in parallel, the divisibility of N by p^i, resulting in the output of indexed boolean values c^i.

3. The reducer `and` computes the conjunction of those outputs.

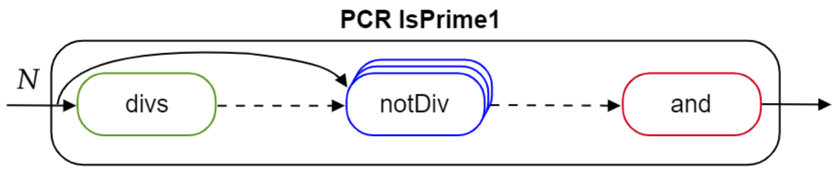


Figure 3.2: Pictorial view of PCR IsPrime1.

```
fun divs(i) = i
fun notDiv(N,p) = not (N % p == 0)

lbnd IsPrime1 = 2
ubnd IsPrime1 = λN.⌊√N⌋

PCR IsPrime1(N)
  par
    p = produce divs
    c = consume notDiv N p
    r = reduce and (N > 1) c
```

Listing 3.2: PCR IsPrime1 code.

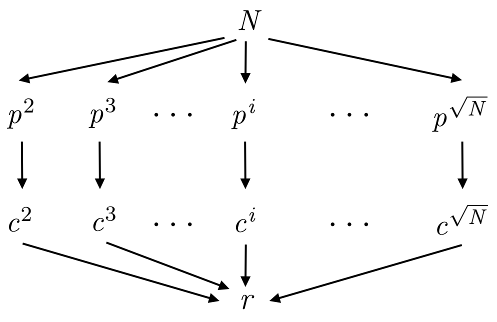


Figure 3.3: Generic dependence graph for PCR IsPrime1.

For PCR IsPrime1, we take the liberty of returning something different from the combiner's identity (i.e. *True*) when the iteration space is empty. Notice that, when $\lfloor\sqrt{N}\rfloor < 2$ we have $r = N > 1 = False$ vacuously. This happens for $N \leq 1$, and makes sense for

[3]In fact, producer `divs` is trivial here, it's just the identity function on the iteration space. In practice, there would be no reason to write the producer in these cases.

this PCR because numbers 0 and 1 are trivially not prime. This decision may seem a bit awkward, but helps to keep this particular introductory example simpler. □

3.1.2 Data dependency syntax

We introduce explicit description of data dependency by means of the **dep** keyword. In general, for variables v_1, v_2 and indices i, j:

$$\textbf{dep} \quad v_1(i) \rightarrow v_2(j)$$

means the value of v_2 at j (i.e. v_2^j) depends on the value of v_1 at i (i.e. v_1^i).

Definition 3.3 (Linear/Non-Linear PCR)**.** A PCR $\mathcal{A}$ is *linear* if its components obey only dependencies of the form (i, i), that is:[4]

$$\textbf{dep} \quad p(i) \rightarrow c_1(i)$$

$$\textbf{dep} \quad c_{e-1}(i) \rightarrow c_e(i), \quad \text{for each } e \in 2..k$$

$$\textbf{dep} \quad c_k(i) \rightarrow r(i).$$

In presence of any other form of data dependency, $\mathcal{A}$ is *non-linear*.

We will not write *linear* dependencies and always assume them implicit. For example, our previous PCR IsPrime1 (3.1.1) is *linear*, which is evident from looking at its dependence graph. Linear PCRs can be thought as representatives of *embarrassing parallelism*, because they evidence that little effort is needed to separate the workload.

Other forms of data dependency (i.e. *non-linear*) relate to more restricted forms of parallelism where the workload cannot be separated in a completely independent way. Table 3.2 presents some simple forms of data dependencies. There, v_1 and v_2 are PCR variables, and we assume that v_1 appears before v_2 if $v_1 \neq v_2$,

We said before that `v[0]` (or more simply `v`) refers to v^i. In the presence of *non-linear* dependencies it is generally necessary to look behind/ahead on these variables. To read

[4]We use *linear* for a lack of better name, and acknowledge that it is a very overloaded term.

Type	Syntax		Meaning
Linear	**dep** $v_1(i) \rightarrow v_2(i)$	where $v_1 \neq v_2$	Value at v_2 depends on value of v_1.
Past	**dep** $v_1(i-k) \rightarrow v_2(i)$	where $k > 0$	Value at v_2 depends on some past value of v_1.
	dep $v_1(..i) \rightarrow v_2(i)$		Value at v_2 depends on all past values of v_1.
Future	**dep** $v_1(i+k) \rightarrow v_2(i)$	where $v_1 \neq v_2$, $k > 0$	Value at v_2 depends on some future value of v_1.
	dep $v_1(i..) \rightarrow v_2(i)$	where $v_1 \neq v_2$	Value at v_2 depends on all future values of v_1.

Table 3.2: Data dependency syntax.

some previous value of v at k (i.e. v^{i-k}) the syntax is `v[-k]`, and to read some posterior value of v at k (i.e. v^{i+k}) the syntax is `v[+k]`.

Remark 3.2.

1. No value can depend on present or future values of the *same* variable, as this situation may introduce deadlock. That is why the $v_1 \neq v_2$ restriction in some rows of table 3.2.

2. Dependencies falling outside iteration space are ignored. For example, if $I_x = 1..N$ then **dep** $v_1(i-1) \rightarrow v_2(i)$ has no effect when $i = 1$ and similarly **dep** $v_1(i+1) \rightarrow v_2(i)$ has no effect when $i = N$ since $v_1(i-1)$ and $v_1(i+1)$ would be undefined in their respective cases.

3. Notice that dependencies on the same consumer (e.g. **dep** $c(i-1) \rightarrow c(i)$) do not have any functional effect, as by definition the function associated to any consumer can only refer to previous consumers and/or the producer but not to itself. In other words, instances of the same consumer are independent, by definition.

4. Although the reducer variable is not treated as a stream, reductions occur along the indexes of the iteration space as with any producer or consumer operation. So, dependencies between reductions can be given a meaningful purpose, for example to fix reduction order. This use is considered later in section 3.2.1.

Example 3.1.2. FibPrimes1 is a basic PCR with a single consumer and a producer that looks behind on previous values to compute. For an input number N, FibPrimes1 returns the number of primes among the first N Fibonacci numbers. It works as follows:

1. The producer `fib` generates the sequence F_1, F_2, ..., F_N of Fibonacci numbers.
2. Each instance $i \in 1..N$ of the `isPrime` consumer checks, in parallel, the primality of F_i, resulting in the output of indexed boolean values c^i.[5]
3. The reducer `count` counts the number of those outputs which are true.

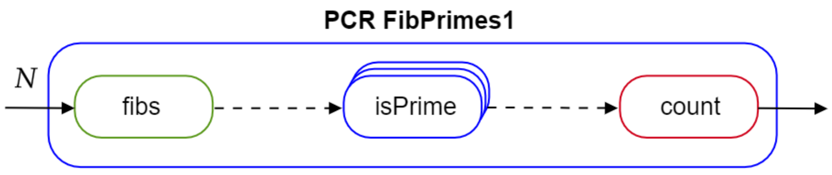


Figure 3.4: Pictorial view of PCR FibPrimes1.

```
fun fibs(p,i)   = if i <= 2 then 1 else p[-1] + p[-2]
fun isPrime(p)  = p > 1 and not (some (λm. divides(m,p)) [2..p-1])
fun count(r,c)  = r + if c then 1 else 0

dep p(i-1) → p(i)
dep p(i-2) → p(i)

lbnd FibPrimes1 = 1
ubnd FibPrimes1 = λN.N

PCR FibPrimes1(N)
  par
    p = produce fibs p
    c = consume isPrime p
    r = reduce count 0 c
```

Listing 3.3: PCR FibPrimes1 code.[6]

Notice when $N < 1$, vacuously: $r = 0$, the identity of $+$. □

Example 3.1.3. RedBlack is a basic PCR with two consumers which presents various *non-linear* dependencies. This implements an *stencil* pattern, which is like a *map* in which each output depends on a "neighborhood" of inputs.

[5]Here, `isPrime` is a basic function, not a PCR. Later we will discuss PCR nesting.
[6]Alternatively, the reducer may be written as: `reduce + 0 (λc. if c then 1 else 0)`

$$\begin{tikzcd}
& & & N \arrow[dlll] \arrow[dl] \arrow[d] \arrow[dr] \arrow[drrr] & & & \\
p^1 \arrow[d] & \cdots & p^{i-2} \arrow[r] \arrow[rr, bend left] \arrow[d] & p^{i-1} \arrow[r] \arrow[d] & p^i \arrow[d] & \cdots & p^N \arrow[d] \\
c^1 \arrow[d] & \cdots & c^{i-2} \arrow[d] & c^{i-1} \arrow[d] & c^i \arrow[d] & \cdots & c^N \arrow[d] \\
f_r(c,1) \arrow[drrr] & \cdots & f_r(c,i-2) \arrow[dr] & f_r(c,i-1) \arrow[d] & f_r(c,i) \arrow[dl] & \cdots & f_r(c,N) \arrow[dlll] \\
& & & r & & &
\end{tikzcd}$$

Figure 3.5: Generic dependence graph for PCR FibPrimes1.

Stencils are used in iterative methods for many applications such as image or signal processing and solving linear systems. In particular, a typical solver for the the method of Finite Differences (used to approximate partial differential equations) iteratively performs stencil computations which, in general terms, consists of a weighted accumulation of the contribution of neighbor points over a a discrete grid. To paralellize the work at each iteration, the red-black ordering treats the grid as a checkerboard with red and black points. Then, each iteration of the algorithm is split into a red step and a black step such that they compute the red and black points respectively.

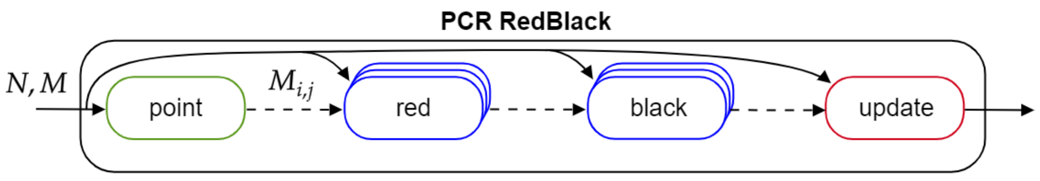


Figure 3.6: Pictorial view of PCR RedBlack.

In our example, the RedBlack PCR represents an iteration step over a grid, represented in turn by a matrix, using a 5-point stencil which, for coordinates x and y, are (x, y), $(x-1, y)$, $(x+1, y)$, $(x, y-1)$ and $(x, y+1)$. This is an example of a 2D stencil; however elements are accessed following a linear indexing scheme. Let M be a $N \times N$ matrix. The vectorization of M, say V, is based on following index relations:

$$V[k] = M[\lfloor (k-1)/N \rfloor + 1][((k-1)\%N) + 1] \qquad ,1 \leq k \leq N^2$$
$$M[i][j] = V[(i-1)N + j] \qquad ,1 \leq i,j \leq N$$

```
fun isRed(i,j)   = (i+j)%2 == 0
fun isBlack(i,j) = (i+j)%2 != 0
fun onBorder(N,i,j) = i == 1 or i == N or j == 1 or j == N
fun point(N,M,k) = M[⌊(k-1)/N⌋+1][((k-1)%N)+1]
fun red(N,p,k) = let i = ⌊(k-1)/N⌋+1
                     j = ((k-1)%N)+1
                 in if isRed(i,j) and not onBorder(N,i,j)
                    then (p+p[-1]+p[+1]+p[-N]+p[+N])/5
                    else p
fun black(N,c1,k) = let i = ⌊(k-1)/N⌋+1
                        j = ((k-1)%N)+1
                    in if isBlack(i,j) and not onBorder(N,i,j)
                       then (c1+c1[-1]+c1[+1]+c1[-N]+c1[+N])/5
                       else c1
fun update(r,N,c2,k) = add(r, L(N,c2,k))
fun zero(N) = ...     // NxN zero matrix
fun L(N,c2,k) = ...   // NxN single entry matrix with c2 at (i,j)
fun add(m1,m2) = ...  // matrix addition

dep p(k-1) → c1(k); dep c1(k-1) → c2(k)
dep p(k+1) → c1(k); dep c1(k+1) → c2(k)
dep p(k-N) → c1(k); dep c1(k-N) → c2(k)
dep p(k+N) → c1(k); dep c1(k+N) → c2(k)

lbnd RedBlack = 1
ubnd RedBlack = λN M.N*N

PCR RedBlack(N,M)  // M is a matrix of size NxN
  par
    p  = produce point N M
    c1 = consume red N p
    c2 = consume black N c1
    r  = reduce update (zero N) N c2
```

Listing 3.4: PCR RedBlack code.

It could be argued that the reducer operation, which is conformed by matrix operations *add* and L, is computationally expensive and defeats the potential benefits of parallelism. Indeed, just *add* requires N^2 additions. However, the intended effect of the reducer here is just to update positions in the resulting matrix, and an efficient implementation for this should be conceivable. But we are not going to extend ourselves on this any further. □

$$\begin{tikzcd}
 & & & & & & & N, M \arrow[dlllllll] \arrow[dlllll] \arrow[dllll] \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] \arrow[drrrr] \arrow[drrrrr] \arrow[drrrrrrr] & & & & & & & \\
p^1 \arrow[d] & \cdots & p^{k-N} \arrow[d] \arrow[drrrrr] & p^{k-N+1} \arrow[d] & \cdots & p^{k-2} \arrow[d] & p^{k-1} \arrow[d] \arrow[dr] & p^k \arrow[d] & p^{k+1} \arrow[d] \arrow[dl] & p^{k+2} \arrow[d] & \cdots & p^{k+N} \arrow[d] \arrow[dllll] & p^{k+N+1} \arrow[d] & \cdots & p^{N^2} \arrow[d] \\
c_1^1 \arrow[d] & \cdots & c_1^{k-N} \arrow[d] & c_1^{k-N+1} \arrow[d] \arrow[drrrrr] & \cdots & c_1^{k-2} \arrow[d] & c_1^{k-1} \arrow[d] & c_1^k \arrow[d] \arrow[dr] & c_1^{k+1} \arrow[d] & c_1^{k+2} \arrow[d] \arrow[dl] & \cdots & c_1^{k+N} \arrow[d] & c_1^{k+N+1} \arrow[d] \arrow[dllll] & \cdots & c_1^{N^2} \arrow[d] \\
c_2^1 \arrow[drrrrrrr] & \cdots & c_2^{k-N} \arrow[drrrrr] & c_2^{k-N+1} \arrow[drrrr] & \cdots & c_2^{k-2} \arrow[drr] & c_2^{k-1} \arrow[dr] & c_2^k \arrow[d] & c_2^{k+1} \arrow[dl] & c_2^{k+2} \arrow[dll] & \cdots & c_2^{k+N} \arrow[dllll] & c_2^{k+N+1} \arrow[dlllll] & \cdots & c_2^{N^2} \arrow[dlllllll] \\
 & & & & & & & r & & & & & & &
\end{tikzcd}$$

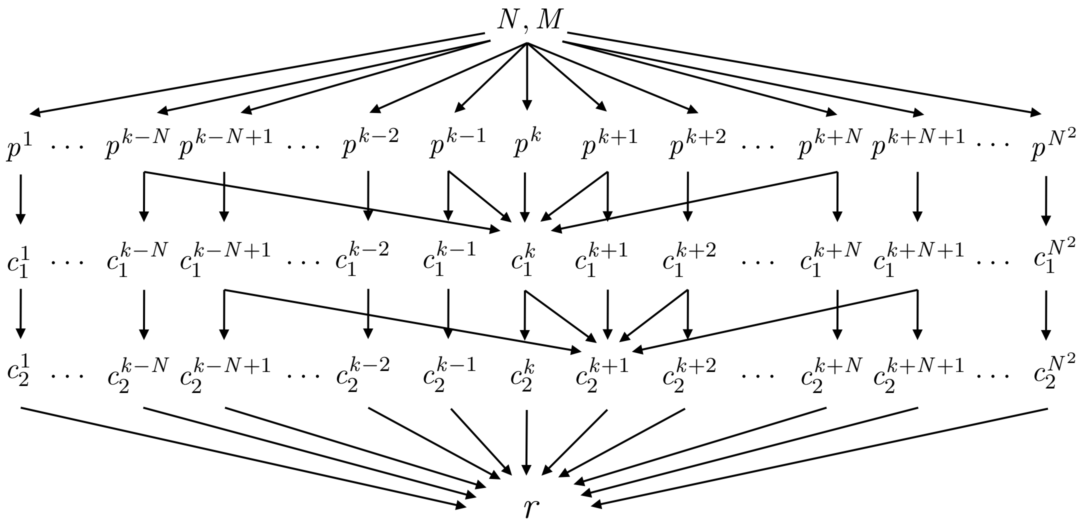


Figure 3.7: Generic dependence graph for PCR RedBlack. Non-linear dependency only shown for c_1^k and c_2^{k+1}. Last transformation L not shown.

3.2 What does a PCR compute?

In this section we investigate what a PCR computes. We are interested in functional behaviour and shall look for an explicit form comprising the basic functions that constitute the PCR.

First, let us make a general observation. All PCR operations, except reductions, are assignments of the form

$$v^i \; := \; f_v(x, u_1, u_2, \ldots, u_k, i) \tag{3.1}$$

where $x \in T$, $v \in \overrightarrow{T}_v$, $u_j \in \overrightarrow{T}_j$ for $1 \leq j \leq k$, $i \in \mathbb{N}$, $f_v : T \times \overrightarrow{T}_1 \times \cdots \times \overrightarrow{T}_k \times \mathbb{N} \to T_v$, and $T_1, \ldots, T_k$, T_v are the (range) types of the intervening variables.

So, variables v and u_j's are streams while f is a function on streams (as well as on input x and index i). Intuitively, we think of the left hand side of 3.1 (i.e. v^i) as an operational aspect of the PCR computation, as with every operation at some assignment i the stream v evolves taking values from f_v evaluated at i. On the other hand, the right hand side (i.e. f_v) can be regarded as a more purely functional aspect of the PCR computation provided the required dependencies over the u_j's streams are met. Moreover, it will be handy to treat it also as a stream. For this, we define $\vec{f}_v : T \times \vec{T}_1 \times \cdots \times \vec{T}_k \to \vec{T}_v$ (essentially a curried version of f_v) as:

$$\vec{f}_v(x, u_1, u_2, \ldots, u_k) \;=\; i \in \mathbb{N} \mapsto f_v(x, u_1, u_2, \ldots, u_k, i) \tag{3.2}$$

so that $v^i = \vec{f}_v^{\,i}(x, u_1, u_2, \ldots, u_k)$ holds for any *written* assignment i, and we do similarly for the u_j's associated functions. This will allow us to express the effect of PCR operations as a functional stream composition originated from basic functions in a *point-free* style:

$$\vec{f}_v(x,\ \vec{f}_{u_1},\ \vec{f}_{u_2},\ \ldots,\ \vec{f}_{u_k}) \tag{3.3}$$

Next, we apply this observation and further discuss related matters carrying out a revision of our previous examples.

Example 3.2.1 (Example 3.1.1 revisited)**.** Consider PCR IsPrime1. Define the stream versions of the basic functions *divs* and *notDiv*:

$$\begin{aligned} \overrightarrow{divs} \;&=\; i \in \mathbb{N} \mapsto divs(i) \\ \overrightarrow{notDiv}(N, p) \;&=\; i \in \mathbb{N} \mapsto notDiv(N, p, i) \end{aligned}$$

Let $N = 19$. Then we have $p^i = \overrightarrow{divs}^{\,i}$ and $c^i = \overrightarrow{notDiv}^{\,i}(19, p)$ for each $i \in 2..19$. The concrete dependence graph is shown in figure 3.8.

Remark 3.3. In this concrete example and others to come, when expanding the stream definitions we will express the basic functions as only acting on the relevant parameters according to the function definition and its associated dependencies. For example, $\overrightarrow{notDiv}^{\,i}(19, \overrightarrow{divs})$ expands to $notDiv(19, divs(i))$ because *notDiv* at i only depends on *divs* at the same i, and also i is not directly used in *notDiv*'s definition. This little piece of informality will help us keeping some parts of the presentation simpler and more readable.

Let us now write $\wedge$ for the combiner **and**. Following the graph, the result should be:

$$\begin{aligned} r \;&=\; \bigwedge_{i=2}^{4} \overrightarrow{notDiv}^{\,i}(19, \overrightarrow{divs}) \\ &=\; True \wedge notDiv(19, divs(2)) \wedge notDiv(19, divs(3)) \wedge notDiv(19, divs(4)) \\ &=\; True \wedge notDiv(19, 2) \wedge notDiv(19, 3) \wedge notDiv(19, 4) \\ &=\; True \wedge True \wedge True \wedge True \\ &=\; True \end{aligned}$$

$$\begin{tikzcd}
 & N = 19 \arrow[dl] \arrow[d] \arrow[dr] & \\
p^2 = \overrightarrow{divs}^2 \arrow[d] & p^3 = \overrightarrow{divs}^3 \arrow[d] & p^4 = \overrightarrow{divs}^4 \arrow[d] \\
c^2 = \overrightarrow{notDiv}^2(N, p) \arrow[dr] & c^3 = \overrightarrow{notDiv}^3(N, p) \arrow[d] & c^4 = \overrightarrow{notDiv}^4(N, p) \arrow[dl] \\
 & r &
\end{tikzcd}$$

Figure 3.8: Dependence graph for PCR IsPrime1 when $N = 19$.

as expected, because 19 is indeed a prime number. So, we could write $\mathit{IsPrime}1(19) = \mathit{True}$.

Recall we decided to not to use the combiner identity when $N \leq 1$.[7] So, in general, we have a piecewise function for $N \in \mathbb{N}$:

$$\mathit{IsPrime}1(N) \;=\; \begin{cases} \mathit{False} & , N \leq 1 \\ \bigwedge_{i=2}^{\lfloor\sqrt{N}\rfloor} \overrightarrow{\mathit{notDiv}}^{i}(N, \overrightarrow{\mathit{divs}}) & , \text{otherwise} \end{cases} \tag{3.4}$$

Or perhaps more conveniently, using McCarthy conditional form:

$$\begin{aligned} \mathit{IsPrime}1(N) \;&=\; \left(N \leq 1 \;\rightarrow\; \mathit{False},\; \bigwedge_{i=2}^{\lfloor\sqrt{N}\rfloor} \overrightarrow{\mathit{notDiv}}^{i}(N, \overrightarrow{\mathit{divs}})\right) \\ &=\; N > 1 \;\wedge\; \bigwedge_{i=2}^{\lfloor\sqrt{N}\rfloor} \overrightarrow{\mathit{notDiv}}^{i}(N, \overrightarrow{\mathit{divs}}) \end{aligned} \tag{3.5}$$

□

Example 3.2.2 (Example 3.1.2 revisited)**.** Consider PCR FibPrimes1. The producer basic function *fibs* is defined as:

$$\mathit{fibs}(p, i) \;=\; \textbf{if } i \leq 2 \textbf{ then } 1 \textbf{ else } p^{i-1} + p^{i-2}$$

Unlike the previous example, the producer depends on p^{i-1} and p^{i-2} to compute the i-th value, which is safe due to the presence of appropriate dependencies. Here, we propose to

[7]Later we will show an alternative PCR solution without this input domain separation and with additional optimization.

understand *fibs* as either:

1. The recurrence
$$\mathit{fibs}(i) \;=\; \begin{cases} 1 & , i \leq 2 \\ \mathit{fibs}(i-1) + \mathit{fibs}(i-2) \end{cases}$$
or

2. The closed form of the recurrence (in this particular case known as Binet's formula):
$$\mathit{fibs}(i) \;=\; \frac{1}{\sqrt{5}}(\varphi^{i} - (-\varphi)^{-i}) \quad \text{where } \varphi \text{ is golden ratio.}$$

In the latter case we abstract away the *non-linear* and self-referential dependencies imposed on the producer. Of course, closed forms are not always known or may not even exist, so the second option is not always viable. It should be noted that the computation of recurrences is often seen as an absolutely sequential endeavour, because the dependence on previous values. For this reason, we normally say this is a *sequential producer*. [8]

Remark 3.4. In general, for any producer function f_p possibly depending on values of the form p^{i-k} for some $k > 0$, we define g_p equal to f_p except replacing every occurrence of the form p^{i-k} by the recursive call $g_p(x, i-k)$. The difference between f_p and g_p is essentially that the former is a *memoized* version of the latter. For the moment, we assume they are equivalent. For the purpose of deductive formal verification, rewriting the producer function as a pure recurrence seems more convenient, because eventually the new function will need to be proved equivalent to the original function and this task could be arbitrarily hard for closed forms. Nevertheless, for automated verification methods like model checking, both options can work fine although the closed form is likely to perform better.

Now, define the stream versions of the basic functions *fibs* and *isPrime*, as well for the

[8]However, it is known from decades ago that there are fast parallel algorithms for a general class of recurrences [27].

transformation before reduction: [9]

$$\begin{aligned}
\overrightarrow{\mathit{fibs}} &= i \in \mathbb{N} \mapsto \mathit{fibs}(i) \\
\overrightarrow{\mathit{isPrime}}(p) &= i \in \mathbb{N} \mapsto \mathit{isPrime}(p, i) \\
[c] &= i \in \mathbb{N} \mapsto [c^i]
\end{aligned}$$

Let $N = 7$, then we have $p^i = \overrightarrow{\mathit{fibs}}^i$ and $c^i = \overrightarrow{\mathit{isPrime}}^i(p)$ for each $i \in 1..7$. The concrete dependence graph is shown in figure 3.9.

$$\begin{tikzcd}
& N = 7 \arrow[dl] \arrow[d] \arrow[drr] & & \\
p^1 = \overrightarrow{\mathit{fibs}}^1 \arrow[r] \arrow[d] & p^2 = \overrightarrow{\mathit{fibs}}^2 \arrow[d] & \cdots & p^7 = \overrightarrow{\mathit{fibs}}^7 \arrow[d] \\
c^1 = \overrightarrow{\mathit{isPrime}}^1(p) \arrow[d] & c^2 = \overrightarrow{\mathit{isPrime}}^2(p) \arrow[d] & \cdots & c^7 = \overrightarrow{\mathit{isPrime}}^7(p) \arrow[d] \\
{[c]^1} \arrow[dr] & {[c]^2} \arrow[d] & \cdots & {[c]^7} \arrow[dll] \\
& r & &
\end{tikzcd}$$

Figure 3.9: Dependence graph for PCR FibPrimes1 when $N = 7$.

Following the graph, the result should be:

$$\begin{aligned}
r &= \sum_{i=1}^{7} \left[\overrightarrow{\mathit{isPrime}}(\overrightarrow{\mathit{fibs}})\right]^i \\
&= 0 + [\mathit{isPrime}(\mathit{fibs}(1))] + [\mathit{isPrime}(\mathit{fibs}(2))] + [\mathit{isPrime}(\mathit{fibs}(3))] + [\mathit{isPrime}(\mathit{fibs}(4))] \\
&\quad + [\mathit{isPrime}(\mathit{fibs}(5))] + [\mathit{isPrime}(\mathit{fibs}(6))] + [\mathit{isPrime}(\mathit{fibs}(7))] \\
&= 0 + [\mathit{isPrime}(1)] + [\mathit{isPrime}(1)] + [\mathit{isPrime}(2)] + [\mathit{isPrime}(3)] \\
&\quad + [\mathit{isPrime}(5)] + [\mathit{isPrime}(8)] + [\mathit{isPrime}(13)] \\
&= 0 + [\mathit{False}] + [\mathit{False}] + [\mathit{True}] + [\mathit{True}] + [\mathit{True}] + [\mathit{False}] + [\mathit{True}] \\
&= 0 + 0 + 1 + 1 + 1 + 0 + 1 \\
&= 4
\end{aligned}$$

as expected. So, we may put $\mathit{FibPrimes1}(7) = 4$.

[9] We use Iverson's bracket notation ($[P]$ where P is a predicate) for the characteristic function which appears as the last transformation before reduction.

In general, for any $N \in \mathbb{N}$:

$$FibPrimes1(N) \;=\; \sum_{i=1}^{N} \left[\overrightarrow{isPrime}(\overrightarrow{fibs})\right]^{i} \tag{3.6}$$

□

Example 3.2.3 (Example 3.1.3 revisited)**.** Consider PCR RedBlack. The producer does not depend on past values as in PCR FibPrimes1, but neither is trivial as in PCR IsPrime1. Unlike those two previous examples, consumer functions *red*, *black* : $\mathbb{N} \times \overrightarrow{\mathbb{R}} \times \mathbb{N} \to \mathbb{R}$ do have *non-linear* dependencies.

Define the stream versions of the basic functions *point*, *red*, and *black*, as well for the transformation L before reduction:

$$\begin{aligned}
\overrightarrow{point}(N, M) &= k \in \mathbb{N} \mapsto point(N, M, k)\\
\overrightarrow{red}(N, p) &= k \in \mathbb{N} \mapsto red(N, p, k)\\
\overrightarrow{black}(N, c_1) &= k \in \mathbb{N} \mapsto black(N, c_1, k)\\
\overrightarrow{L}(N, c_2) &= k \in \mathbb{N} \mapsto L(N, c_2, k)
\end{aligned}$$

Let $N = 4$ and

$$M = \begin{pmatrix} 0 & 0 & 0 & 0\\ 0 & 3 & 9 & 0\\ 0 & 5 & 6 & 0\\ 0 & 0 & 0 & 0 \end{pmatrix}$$

where M is represented as $[[0, 0, 0, 0], [0, 3, 9, 0], [0, 5, 6, 0], [0, 0, 0, 0]]$. It is easy to see that matrix M in vectorized form is just the flattened version:

$$[0, 0, 0, 0, 0, 3, 9, 0, 0, 5, 6, 0, 0, 0, 0, 0]$$

Then, we have $p^k = \overrightarrow{point}^{\,k}(4, M)$, $c_1^k = \overrightarrow{red}^{\,k}(4, p)$ and $c_2^k = \overrightarrow{black}^{\,k}(4, c_1)$ for each $i \in 1..16$. Concrete dependence graph for this input is shown in figure 3.10.

Let's write $\oplus$ for matrix addition, 0_4 for the zero 4×4 matrix and $L_{ij}(x)$ for the single element matrix with value x at position (i, j) where $i = \lfloor (k-1)/N \rfloor + 1$ and $j = ((k-1)\%N) + 1$ (i.e. $L_{ij}(x) = L(N, x, k)$). Following the graph, the result should be: [10]

[10] Anticipating that boundary terms result in trivial zero 4×4 matrices, we hide this 12 terms in ellipsis.

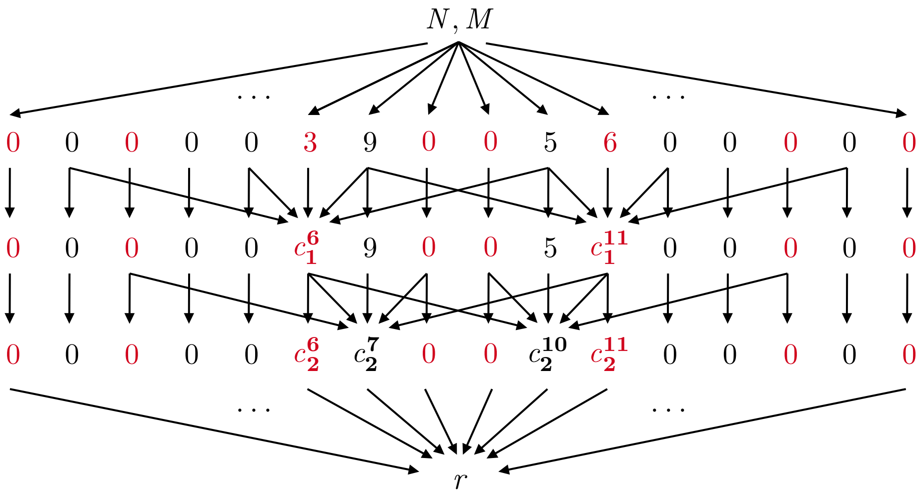


Figure 3.10: Dependence graph for PCR BlackRed when $N = 4$ and $M = [[0,0,0,0],$ $[0,3,9,0],[0,5,6,0],[0,0,0,0]]$. Non-linear dependences only shown for c_1^6, c_1^{11}, c_2^7 and c_2^{10}. Last transformation L not shown.

$$\begin{aligned}
r &= \bigoplus_{k=1}^{16} \overrightarrow{L}^{k}(4, \overrightarrow{black}(4, \overrightarrow{red}(4, \overrightarrow{point}(4, M)))) \\
&= 0_4 \oplus \cdots \oplus L_{2,2}(black(4,\ \overrightarrow{red}^{6}(4,p))) \\
&\qquad\qquad \oplus L_{2,3}(black(4,\ \overrightarrow{red}^{3}(4,p),\ \overrightarrow{red}^{6}(4,p),\ \overrightarrow{red}^{7}(4,p),\ \overrightarrow{red}^{8}(4,p),\ \overrightarrow{red}^{11}(4,p))) \\
&\qquad \oplus \cdots \oplus L_{3,2}(black(4,\ \overrightarrow{red}^{6}(4,p),\ \overrightarrow{red}^{9}(4,p),\ \overrightarrow{red}^{10}(4,p),\ \overrightarrow{red}^{11}(4,p),\ \overrightarrow{red}^{14}(4,p))) \\
&\qquad\qquad \oplus L_{3,3}(black(4,\ \overrightarrow{red}^{11}(4,p))) \oplus \ldots \\
&= 0_4 \oplus \cdots \oplus L_{2,2}(black(4,\ red(4,p^2,p^5,p^6,p^7,p^{10}))) \\
&\qquad\qquad \oplus L_{2,3}(black(4,\ p^3,\ red(4,p^2,p^5,p^6,p^7,p^{10}),\ p^7,\ p^8,\ red(4,p^7,p^{10},p^{11},p^{12},p^{15}))) \\
&\qquad \oplus \cdots \oplus L_{3,2}(black(4,\ red(4,p^2,p^5,p^6,p^7,p^{10}),\ p^9,\ p^{10},\ red(4,p^7,p^{10},p^{11},p^{12},p^{15}),\ p^{14})) \\
&\qquad\qquad \oplus L_{3,3}(black(4,\ red(4,p^7,p^{10},p^{11},p^{12},p^{15}))) \oplus \ldots \\
&= 0_4 \oplus \cdots \oplus L_{2,2}(black(4,\ 3.4)) \oplus L_{2,3}(black(4,\ 0,\ 3.4,\ 9,\ 0,\ 4)) \\
&\qquad \oplus \cdots \oplus L_{3,2}(black(4,\ 3.4,\ 0,\ 5,\ 4,\ 0)) \oplus L_{3,3}(black(4,\ 4)) \oplus \ldots \\
&= 0_4 \oplus \cdots \oplus \underbrace{L_{2,2}(3.4)}_{c_2^6} \oplus \underbrace{L_{2,3}(3.3)}_{c_2^7} \oplus\ \cdots \oplus \underbrace{L_{3,2}(2.5)}_{c_2^{10}} \oplus \underbrace{L_{3,3}(4)}_{c_2^{11}} \oplus\ \ldots \\
&= [[0,0,0,0],[0,3.4,3.3,0],[0,2.5,4,0],[0,0,0,0]]
\end{aligned}$$

In general, for any $N \in \mathbb{N}$ and $M \in \mathcal{M}(\mathbb{R})_{N\times N}$:

$$\mathit{RedBlack}(N, M) \;=\; \bigoplus_{k=1}^{N^2} \overrightarrow{L}^{k}(N, \overrightarrow{\mathit{black}}(N, \overrightarrow{\mathit{red}}(N, \overrightarrow{\mathit{point}}(N, M)))) \tag{3.7}$$

$\square$

In what follows, we generalize what we did on previous examples for arbitrary basic PCRs.

Proposition 3.1. Let $\mathcal{A}$ be a basic PCR with input parameter of type T and whose reducer variable is of type D. Then, for all $x \in T$, the output of $\mathcal{A}$ at x, to be written $\mathcal{A}(x)$, is determined if $\mathcal{A}$ terminates at input x, and it holds: $\mathcal{A}(x) \;=\; \bigotimes_{i\,\in\, I_x} \overrightarrow{f}_{\mathcal{A}}^{\,i}$ for some function $\overrightarrow{f}_{\mathcal{A}} \in \overrightarrow{D}$.

Proof. Let $\mathcal{A}$ consist of k consumers and the following basic functions

$$\begin{aligned}
f_p &: T \times \overrightarrow{T}_p \times \mathbb{N} \to T_p \\
f_{c_h} &: T \times \overrightarrow{T}_p \times \overrightarrow{T}_{c_1} \times \cdots \times \overrightarrow{T}_{c_{h-1}} \times \mathbb{N} \to T_{c_h} \quad \text{for each } h \in 1..k \\
f_r &: T \times \overrightarrow{T}_p \times \overrightarrow{T}_{c_1} \times \cdots \times \overrightarrow{T}_{c_k} \times \mathbb{N} \to D.
\end{aligned}$$

Recall $g_p : T \times \mathbb{N} \to T_p$ is equivalent to f_p but dropping the stream parameter (remark 3.4). Now, define the stream versions of the basic functions g_p, f_{c_1}, ..., f_{c_k} and f_r:

$$\begin{aligned}
\overrightarrow{g}_p(x) &= i \in \mathbb{N} \mapsto g_p(x, i) \\
\overrightarrow{f}_{c_h}(x, p, c_1, \ldots, c_{h-1}) &= i \in \mathbb{N} \mapsto f_{c_h}(x, p, c_1, \ldots, c_{h-1}, i) \quad \text{for each } h \in 1..k \\
\overrightarrow{f}_r(x, p, c_1, \ldots, c_k) &= i \in \mathbb{N} \mapsto f_r(x, p, c_1, \ldots, c_k, i).
\end{aligned}$$

The combiner $\otimes$ is an associative and commutative operation over domain D with identity $\mathtt{id}_\otimes$, i.e. in algebraic terms, the structure $(D, \mathtt{id}_\otimes, \otimes)$ is an abelian monoid. Let $m = \mathit{min}(I_x)$, $n = \mathit{max}(I_x)$, and $\overrightarrow{f}_{\mathcal{A}} \in \overrightarrow{D}$ so that

$$\overrightarrow{f}_{\mathcal{A}} \;=\; \overrightarrow{f}_r(x,\ \overrightarrow{g}_p,\ \overrightarrow{f}_{c_1},\ \ldots,\ \overrightarrow{f}_{c_k}).$$

Then, the output of $\mathcal{A}$ at input x is given by combiner's work on all indices of the iteration space I_x in any order, starting from identity value

$$r \;=\; \mathtt{id}_\otimes \otimes \overrightarrow{f}_{\mathcal{A}}^{\,m} \otimes \overrightarrow{f}_{\mathcal{A}}^{\,m'} \otimes \cdots \otimes \overrightarrow{f}_{\mathcal{A}}^{\,n}$$

which we will write as

$$\mathcal{A}(x) \;=\; \bigotimes_{i \in I_x} \overrightarrow{f}^{\,i}_{\mathcal{A}} \tag{3.8}$$

$\square$

In particular, for a single consumer PCR $\mathcal{A}$, equation 3.8 expands to:

$$\mathcal{A}(x) \;=\; \bigotimes_{i \in I_x} \overrightarrow{f}^{\,i}_{r}\big(x,\ \overrightarrow{g}_p(x),\ \overrightarrow{f}_c(x,\ \overrightarrow{g}_p(x))\big) \tag{3.9}$$

Equations 3.5, 3.6 and 3.7 derived in previous examples are special cases of 3.8, and more specifically 3.5 and 3.6 are special cases of 3.9 as they have only one consumer. Further clarification is needed for 3.5, as this assertion strictly holds only for input $N > 1$, because when $N \leq 1$ we have $r = \mathit{False}$ which is not the combiner's identity.

3.2.1 Non-commutativity

As we discussed earlier in section 2.1.2.2, there are reasons why it is convenient for reduction to assume a *commutative* combiner operation. By default, we adhere to that view in our PCR model. However, there are non-commutative useful and fairly common operations and is generally not safe to treat them as they where commutative since a non-deterministic result is likely to be obtained, thus leading to a non-functional behaviour.

Next, we illustrate by a simple example how to accommodate a non-commutative reduction in the PCR model by imposing a fixed order with respect to the iteration space through *non-linear* dependences. The same idea will be used again in more interesting examples later.

Example 3.2.4. Binary operation `++` (concatenation) over lists[11] is associative and it has an identity element (denoted $[\,]$) but is not commutative.

PCR ListId (see listing 3.5) should behave as the identity function on lists. For this to hold, reduction order is fixed (in particular, from left to right) by using the following

[11] We assume 1-indexed finite lists.

dependency over reducer output variable

$$\textbf{dep} \;\; r(i-1) \rightarrow r(i)$$

So, for the list $[a_1,\ a_2,\ \ldots,\ a_n]$ of length n, reduction has the form

$$r \;=\; (([\,]\ \texttt{++}\ [a_1])\ \texttt{++}\ [a_2])\ \texttt{++}\ \cdots\ \texttt{++}\ [a_n]$$

Of course, for this kind of use case one could envision a specific PCR reducer syntax (e.g. **`reduceLeft`**) with the intended semantics or as syntactic sugar for the previous dependency. But we are fine without that.

What would happen in this example without a fixed order?. There would be $n!$ possible output lists, that is, any permutation of the original input list (assuming no repeated elements).

```
fun elem(L,i) = [L[i]]

dep r(i-1) → r(i)

lbnd ListId = 1
ubnd ListId = λL.len(L)

PCR ListId(L)
  r = reduce (++) [] (elem L)
```

Listing 3.5: PCR ListId code.

□

3.3 Composition

The form of composition we are interested here is *nesting*. PCRs can be composed by hierarchical nesting, which allows reusing components and controlling the desired grain of parallelism. It can also be helpful in making more manageable the task of correctness verification by either automatic or semi-automatic methods.

In [8], Pérez and Yovine considered PCR nesting at the consumer component, but for the present work we also explored nesting at the other components. In general, the composition can be made at three different places: (1) the producer, (2) any of the consumers or (3) the reducer. However, in what follows we will discuss only cases (2) and (3) because (1) is not much different from (2) and we don't have yet an interesting enough use case to illustrate it. For each case, an appropriate PCR scheme will be given and discussed. Besides, a PCR scheme for the parallel *divide and conquer* strategy will be presented as a form of recursive composition at the consumer place. [12]

As we showed previously, a basic PCR behaves as a function of the form given by equation 3.8, thus we intuitively expect nested PCRs to behave as a composition of functions. The definition of termination given before extends naturally to composed PCRs.

3.3.1 Composition through consumer

We prefer to start with a motivating example involving two of our previous PCRs to introduce the basic ideas and then generalize. This is the same example we presented before, but informally, in section 2.2.2.

3.3.1.1 Example: Putting FibPrimes1 and IsPrime1 together

Consider PCR FibPrimes1. Instead of using the `isPrime` basic function at the consumer we can use our PCR IsPrime1 to the same effect but taking advantage of parallel primality testing.[13] This idea is implemented in a new PCR FibPrimes2 illustrated by figures 3.11, 3.12 and listing 3.6.

```
PCR FibPrimes2(N)                 PCR IsPrime1(F)
  par                               par
    p = produce fibs p                d = produce divs
    c = consume IsPrime1 p            b = consume notDiv F d
    r = reduce count 0 c              a = reduce and (F > 1) b
```

Listing 3.6: PCR FibPrimes2 code, which includes PCR IsPrime1.

At the surface, by looking at listing 3.6, it seems like FibPrimes2 works pretty much the

[12] In [26], *divide and conquer* was introduced as a PCR extension.
[13] Here we are assuming that the basic function and the PCR are functionally equivalent.

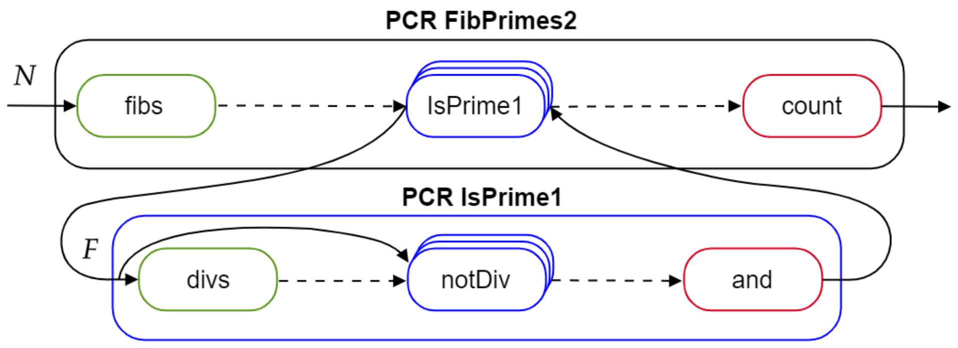


Figure 3.11: Pictorial view of PCR FibPrimes2.

same as FibPrimes1, and indeed that is the intention. Also, IsPrime1 is exactly the same as introduced before (modulo a renaming of variables). However, under the hood there are important differences.

Let N be any input for FibPrimes2. For each assignment $i \in 1..N$ there is the i-ith instance of IsPrime1, so there are in general N instances (and N inputs/outputs) of IsPrime1. Consequently, the producer and consumer variables of the inner PCR IsPrime1 are doubly indexed, i.e. by the outer scope assignment $i \in 1..N$ (the father instance) and the inner scope assignment $j \in 2..\lfloor\sqrt{F^i}\rfloor$ where F^i is the i-th Fibonacci number. We therefore denote by $d^{i,j}$ the j-th produced divisor at instance i of IsPrime1 and do similarly for the boolean variable b, as can be appreciated in figure 3.12. Comparing with FibPrimes1 (see figure 3.5) where the consumer is a basic function, we have that each dependence $p^i \to c^i$ is now expanded to the DAG of the i-ith instance of PCR IsPrime1. The present situation suggests that, in general, PCR indices should be sequences of assignments that grow on nesting. This approach will be developed in the next section.

Now, *what* does the PCR FibPrimes2 compute?. We perform an analysis analogous to the one in example 3.2.2 for FibPrimes1. There is a stream version of PCR IsPrime1 just like for any basic function:

$$\overrightarrow{IsPrime1}(p) \;=\; i \in \mathbb{N} \mapsto IsPrime1(p, i)$$

where $IsPrime1$ is defined according to equation 3.5 and with explicit *father index* pa-

$$\begin{tikzcd}
 & & & & & N \arrow[dllll] \arrow[dll] \arrow[d] \arrow[drr] \arrow[drrrr] & & & & & \\
 & p^1 \arrow[d] & \cdots & p^{i-1} \arrow[rr] \arrow[rrrr, bend left] \arrow[d] & & p^i \arrow[rr] \arrow[d] & & p^{i+1} \arrow[d] & \cdots & p^N \arrow[d] & \\
 & F^1 \arrow[dl] \arrow[dr] & & \vdots & & F^i \arrow[dl] \arrow[dr] & & \vdots & & F^N \arrow[dl] \arrow[dr] & \\
d^{1,2} \arrow[d] & \cdots & d^{1,\sqrt{F^1}} \arrow[d] & \vdots & d^{i,2} \arrow[d] & \cdots & d^{i,\sqrt{F^i}} \arrow[d] & \vdots & d^{N,2} \arrow[d] & \cdots & d^{N,\sqrt{F^N}} \arrow[d] \\
b^{1,2} \arrow[dr] & \cdots & b^{1,\sqrt{F^1}} \arrow[dl] & & b^{i,2} \arrow[dr] & \cdots & b^{i,\sqrt{F^i}} \arrow[dl] & & b^{N,2} \arrow[dr] & \cdots & b^{N,\sqrt{F^N}} \arrow[dl] \\
 & a^1 \arrow[d] & & \vdots \arrow[d] & & a^i \arrow[d] & & \vdots \arrow[d] & & a^N \arrow[d] & \\
 & c^1 \arrow[d] & \cdots & c^{i-1} \arrow[d] & & c^i \arrow[d] & & c^{i+1} \arrow[d] & \cdots & c^N \arrow[d] & \\
 & f_r(c,1) \arrow[drrrr] & \cdots & f_r(c,i-1) \arrow[drr] & & f_r(c,i) \arrow[d] & & f_r(c,i+1) \arrow[dll] & \cdots & f_r(c,N) \arrow[dllll] & \\
 & & & & & r & & & & &
\end{tikzcd}$$

Figure 3.12: Generic dependence graph for PCR FibPrimes2.

rameter:

$$IsPrime1(p,i) \;=\; p^i > 1 \;\wedge\; \bigwedge_{j=2}^{\lfloor\sqrt{p^i}\rfloor} \overrightarrow{notDiv}^{\,j}(p^i, \overrightarrow{divs})$$

Let $N = 7$. The corresponding concrete dependence graph is shown in figure 3.13. Following the graph, the result should be:

$$\begin{aligned}
r &= \sum_{i=1}^{7} \left[\overrightarrow{IsPrime1}(\overrightarrow{fibs})\right]^{i} \\
&= 0 + [fibs(1) > 1] + [notDiv(fibs(2), divs(2))] + [notDiv(fibs(3), divs(2))] \\
&\quad + [notDiv(fibs(4), divs(2))] + [notDiv(fibs(5), divs(2))] + [notDiv(fibs(6), divs(2))] \\
&\quad + [notDiv(fibs(7), divs(2)) \wedge notDiv(fibs(7), divs(3))] \\
&= 0 + [1 > 1] + [notDiv(1, 2)] + [notDiv(2, 2)] \\
&\quad + [notDiv(3, 2)] + [notDiv(5, 2)] + [notDiv(8, 2)] \\
&\quad + [notDiv(13, 2) \wedge notDiv(13, 3)] \\
&= 0 + [False] + [False] + [True] + [True] + [True] + [False] + [True] \\
&= 0 + 0 + 1 + 1 + 1 + 0 + 1 \\
&= 4
\end{aligned}$$

as expected. So, we have $FibPrimes2(7) = 4$.

In general, for any $N \in \mathbb{N}$:

$$FibPrimes2(N) = \sum_{i=1}^{N} \left[i \in \mathbb{N} \mapsto \overrightarrow{fibs}^{i} > 1 \wedge \bigwedge_{j=2}^{\left\lfloor \sqrt{\overrightarrow{fibs}^{i}} \right\rfloor} \overrightarrow{notDiv}^{j}(\overrightarrow{fibs}^{i}, \overrightarrow{divs}) \right]^{i} \qquad (3.10)$$

3.3.1.2 General scheme

In general, indices are sequences of natural numbers. Let I be the index of a particular execution of a PCR. Any child PCR inherits the index of the father and extends its dimension by writing to its producer variable, say p, the (I, i)-th value $p^{I,i}$, for every assignment i according to its iteration space. This multidimensional indexing allows for the concurrent execution of any two instances $I \neq J$ of the child PCR, each one generating its own set of p values, namely $p^{I,i}$ and $p^{J,j}$. The root index of execution is taken to be the empty sequence. [14]

[14]More formally, we should write $p^{I \circ \langle i \rangle}$ where $\circ$ concatenates sequence I with singleton $\langle i \rangle$. But the formal treatment will come later.

$$\begin{tikzcd}
 & & N = 7 \arrow[dll] \arrow[dl] \arrow[drr] & & & \\
p^1 = \overrightarrow{fibs}^{1} \arrow[r] \arrow[d] & p^2 = \overrightarrow{fibs}^{2} \arrow[d] & \cdots & & p^7 = \overrightarrow{fibs}^{7} \arrow[d] & \\
F^1 = p^1 \arrow[ddd] & F^2 = p^2 \arrow[d] & & & F^7 = p^7 \arrow[dl] \arrow[dr] & \\
 & d^{2,2} = \overrightarrow{divs}^{2} \arrow[d] & & d^{7,2} = \overrightarrow{divs}^{2} \arrow[d] & & d^{7,3} = \overrightarrow{divs}^{3} \arrow[d] \\
 & b^{2,2} = \overrightarrow{notDiv}^{2}\left(F^2, d^2\right) \arrow[d] & \cdots & b^{7,2} = \overrightarrow{notDiv}^{2}\left(F^7, d^7\right) \arrow[dr] & & b^{7,3} = \overrightarrow{notDiv}^{3}\left(F^7, d^7\right) \arrow[dl] \\
a^1 = \mathit{False} \arrow[d] & a^2 \arrow[d] & & & a^7 \arrow[d] & \\
c^1 = a^1 \arrow[d] & c^2 = a^2 \arrow[d] & & & c^7 = a^7 \arrow[d] & \\
{[c]^1} \arrow[drr] & {[c]^2} \arrow[dr] & \cdots & & {[c]^7} \arrow[dll] & \\
 & & r & & &
\end{tikzcd}$$

Figure 3.13: Dependence graph for PCR FibPrimes2 when $N = 7$.

Next, we introduce the scheme for PCR composition through consumer. Elements of distinct PCRs will be identified by sub-script notation.

Definition 3.4 (Consumer composition scheme)**.** Let $\mathcal{A}$ and $\mathcal{B}$ be PCRs with k_1 and k_2 consumers respectively. The computation of $\mathcal{A}$ composed with $\mathcal{B}$ through consumer $e \in 1..k_1$, on input x_1, consists in the following operations for each $i \in I_{x_1}$ and $j \in J_{x_2^i}$:

PCR $\mathcal{A}$:

$$\begin{aligned}
p_1^i &:= f_{p_1}(x_1, p_1, i) \\
c_{1,1}^i &:= f_{c_{1,1}}(x_1, p_1, i) \\
c_{1,2}^i &:= f_{c_{1,2}}(x_1, p_1, c_{1,1}, i) \\
&\vdots \\
c_{1,e}^i &:= r_2^i \ \text{ if } \ end_{\mathcal{B}^i} \\
&\vdots \\
c_{1,k_1}^i &:= f_{c_{1,k_1}}(x_1, p_1, c_{1,1}, \ldots, c_{1,k_1-1}, i) \\
r_1 &:= r_1 \otimes f_{r_1}(x_1, p_1, c_{1,1}, \ldots, c_{1,k_1}, i)
\end{aligned}$$

PCR $\mathcal{B}$:

$$\begin{aligned}
x_2^i &:= (x_1, p_1, c_{1,1}, \ldots, c_{1,e-1}, i) \\
p_2^{i,j} &:= f_{p_2}(x_2^i, p_2^i, j) \\
c_{2,1}^{i,j} &:= f_{c_{2,1}}(x_2^i, p_2^i, j) \\
c_{2,2}^{i,j} &:= f_{c_{2,2}}(x_2^i, p_2^i, c_{2,1}^i, j) \\
&\vdots \\
c_{2,k_2}^{i,j} &:= f_{c_{2,k_2}}(x_2^i, p_2^i, c_{2,1}^i, \ldots, c_{2,k_2-1}^i, j) \\
r_2^i &:= r_2^i \oplus f_{r_2}(x_2^i, p_2^i, c_{2,1}^i, \ldots, c_{2,k_2}^i, j)
\end{aligned}$$

where:

- $I_{x_1} = \{i \in lBnd_1(x_1)..uBnd_1(x_1) : prop_1(i)\}$
  $J_{x_2^i} = \{j \in lBnd_2(x_2^i)..uBnd_2(x_2^i) : prop_2(j)\}$ for any $i \in I_{x_1}$
- p_1, $c_{1,1}$, ..., c_{1,k_1}, r_1 are the variables of PCR $\mathcal{A}$, and
  f_{p_1}, $f_{c_{1,1}}$, ..., $f_{c_{1,k_1}}$, f_{r_1} are the *basic functions* associated to these variables.
- p_2, $c_{2,1}$, ..., c_{2,k_2}, r_2 are the variables of PCR $\mathcal{B}$, and
  f_{p_2}, $f_{c_{2,1}}$, ..., $f_{c_{2,k_2}}$, f_{r_2} are the *basic functions* associated to these variables.
- $\otimes$ and $\oplus$ are the combiner operations of PCR $\mathcal{A}$ and $\mathcal{B}$ respectively.

Remark 3.5.

1. There is not a basic function $f_{c_{1,e}}$ in PCR $\mathcal{A}$. The value of $c_{1,e}$ at each assignment i comes from the output of the i-th instance of PCR $\mathcal{B}$ (i.e. $\mathcal{B}^i$), which is the *final* value of r_2^i.
2. $\mathcal{B}$ is assumed to be a basic PCR, whose input is written by $\mathcal{A}$ in variable x_2. In general, any x_2^i is a tuple of the form $(x_1, p_1, c_{1,1}, \ldots, c_{1,e-1}, i)$. This means $\mathcal{B}$ have access to father's input, all father's variables till $c_{1,e-1}$ and outer assignment i. So, any basic function of $\mathcal{B}$ (including the lower/upper bound functions) can look behind/ahead on the father stream variables subject to appropriate dependencies.
3. The present scheme is assuming that the PCR $\mathcal{A}$ is the root of the composition hierarchy, so its index is the empty sequence. However, more generally, if $\mathcal{A}$ is not at the root then it could be indexed at any index I, in which case the scheme can be accommodated prepending all super-scripts with I (e.g. x_1^I, $p_1^{I,i}$, $x_2^{I,i}$, $p_2^{I,i,j}$, etc.).

```
// basic functions $f_{p_1}$, $f_{c_{1,1}}$, ..., $f_{c_{1,k_1}}$, $f_{r_1}$, $r_{1,0}$ for $\mathcal{A}$
// lbnd, ubnd and prop for $\mathcal{A}$

PCR $\mathcal{A}$($x_1$)
  par
    $p_1$     = produce $f_{p_1}$ $x_1$ $p_1$
    $c_{1,1}$   = consume $f_{c_{1,1}}$ $x_1$ $p_1$
    $c_{1,2}$   = consume $f_{c_{1,2}}$ $x_1$ $p_1$ $c_{1,1}$
              ⋮
    $c_{1,e}$   = consume $\mathcal{B}$ $x_1$ $p_1$ $c_{1,1}$ ... $c_{1,e-1}$     // PCR $\mathcal{B}$ call
              ⋮
    $c_{1,k_1}$  = consume $f_{c_{1,k_1}}$ $x_1$ $p_1$ $c_{1,1}$ $c_{1,2}$ ... $c_{1,k_1-1}$
```

```
      $r_1$    = reduce $\otimes$ ($r_{1,0}$ $x_1$) ($f_{r_1}$ $x_1$ $p_1$ $c_{1,1}$ ... $c_{1,k_1}$)

// basic functions $f_{p_2}$, $f_{c_{2,1}}$, ..., $f_{c_{2,k_2}}$, $f_{r_2}$, $r_{2,0}$ for $\mathcal{B}$
// lbnd, ubnd and prop for $\mathcal{B}$

PCR $\mathcal{B}$($x_2$)
  par
    $p_2$      = produce $f_{p_2}$ $x_2$ $p_2$
    $c_{2,1}$    = consume $f_{c_{2,1}}$ $x_2$ $p_2$
    $c_{2,2}$    = consume $f_{c_{2,2}}$ $x_2$ $p_2$ $c_{2,1}$
             $\vdots$
    $c_{2,k_2}$   = consume $f_{c_{2,k_2}}$ $x_2$ $p_2$ $c_{2,1}$ $c_{2,2}$ ... $c_{2,k_2-1}$
    $r_2$      = reduce $\oplus$ ($r_{2,0}$ $x_2$) ($f_{r_2}$ $x_2$ $p_2$ $c_{2,1}$ ... $c_{2,k_2}$)
```

Listing 3.7: Consumer composition scheme in syntactic form. In general, $\mathcal{B}(x_2)$ expands to $\mathcal{B}(x_1, p_1, c_{1,1}, \ldots, c_{1,e-1}, i)$.

Now, just as we did previously for basic PCRs, we establish the composition scheme behaviour as a function.

Proposition 3.2. Let $\mathcal{A}$ and $\mathcal{B}$ be PCRs with k_1 and k_2 consumers respectively, so that $\mathcal{A}$ with input parameter type T and reducer variable of type D_1 is composed with $\mathcal{B}$ through consumer $e \in 1..k_1$. Then, for all $x_1 \in T$, the output of $\mathcal{A}$ at x_1, to be written $\mathcal{A}(x_1)$, is determined if $\mathcal{A}$ terminates at input x_1 and it holds: $\mathcal{A}(x_1) = \bigotimes_{i \in I_{x_1}} \vec{f}^{\,i}_{\mathcal{A}}$ for some function $\vec{f}_{\mathcal{A}} \in \vec{D}_1$.

Proof. Let $x_1 \in T$. In general, we have that $\mathcal{B}$ is a basic PCR with input type (abbr. T_2)

$$T \times \vec{T}_{p_1} \times \vec{T}_{c_{1,1}} \times \cdots \times \vec{T}_{c_{1,e-1}} \times \mathbb{N}$$

According to proposition 3.1, for any $x_2^i \in T_2$ where $i \in I_{x_1}$:

$$\mathcal{B}(x_2^i) \;=\; \bigoplus_{j \in J_{x_2^i}} \vec{f}^{\,j}_{\mathcal{B}} \quad \text{for some} \quad \vec{f}_{\mathcal{B}} \in \vec{D}_2$$

Now, proceed exactly like in proposition 3.1 except that in this case the stream function for consumer $e \in 1..k_1$ of $\mathcal{A}$ is defined as:

$$\vec{f}_{c_{1,e}}(x_1, p_1, c_{1,1}, \ldots, c_{1,e-1}) \;=\; i \in \mathbb{N} \mapsto \mathcal{B}(x_2^i)$$

where $x_2^i = (x_1, p_1, c_{1,1}, \ldots, c_{1,e-1}, i)$ and then obtain

$$\mathcal{A}(x_1) \;=\; \bigotimes_{i \,\in\, I_{x_1}} \vec{f}^{\,i}_{\mathcal{A}} \tag{3.11}$$

where $\vec{f}_{\mathcal{A}} = \vec{f}_{r_1}(x_1,\, \vec{g}_{p_1}, \vec{f}_{c_{1,1}}, \ldots, \vec{f}_{c_{1,e}}, \ldots, \vec{f}_{c_{1,k_1}})$. $\square$

In particular, for two single consumer PCRs $\mathcal{A}$ and $\mathcal{B}$, equation 3.11 expands to:

$$\mathcal{A}(x_1) \;=\; \bigotimes_{i \,\in\, I_{x_1}} \vec{f}^{\,i}_{r_1}\Big(x_1,\; \vec{g}_{p_1}(x_1),\; i \in \mathbb{N} \mapsto \bigoplus_{j \,\in\, J_{x_2^i}} \vec{f}^{\,j}_{r_2}(x_2^i,\; \vec{g}_{p_2}(x_2^i),\; \vec{f}_{c_2}(x_2^i,\; \vec{g}_{p_2}(x_2^i)))\Big) \tag{3.12}$$

where $x_2^i = (x_1,\; \vec{g}_{p_1}(x_1),\; i)$.

Coming back to our previous example FibPrimes2, equation 3.10 is *almost* a special case of 3.12.[15] Also, we have that

$$\mathit{FibPrimes2}(N) \;=\; \mathit{FibPrimes1}(N) \quad \text{for any } N \in \mathbb{N}$$

holds if

$$\mathit{IsPrime1}(F_i) \;=\; \mathit{isPrime}(F_i) \quad \text{for all Fibonacci numbers } F_i \text{ with } i \in 1..N.$$

In other words, if in the PCR FibPrimes1 we replace the basic function isPrime with the PCR IsPrime1 and both behave the same under the relevant inputs generated by FibPrimes1's producer, we obtain a new PCR which is functionally equivalent to FibPrimes1. And, of course, the same observation holds for any pair of PCRs conforming to definition 3.4.

3.3.2 Divide and Conquer

Property 3.1 shows that a basic PCR behaves like a function. This allows calling a PCR from any *basic function* in a blocking way, where the caller holds until the call returns. Of course, even if the caller is blocked, the parallelism inside the callee is preserved. Calling a PCR as a function enables recursive parallelism. A prominent use case is *divide and conquer*, which in turn is an special case of the *Fork/Join* pattern.

[15] The qualification "almost" is because, as was noted earlier, PCR IsPrime1 conforms to proposition 3.1 strictly for input $N > 1$, so there is a small deviation.

Divide and conquer is an algorithmic technique consisting in partitioning a complex instance of a problem into several smaller subproblems, solving each one independently, and combining their solutions in order to calculate the final result. Each subproblem can be solved directly if it is simple enough. Otherwise divide and conquer can be recursively applied. In general the following functions are used:

- `div`: partitions a problem into subproblems.
- `isBase`: checks whether a problem is a base case.
- `base`: computes the solution for a base case.
- `conquer`: describes how to combine solutions.

A PCR scheme for this technique is presented in listing 3.8. The producer partitions the original problem into subproblems using the `iterDiv` function. Consumers process each subproblem, either using `base` or recursively calling PCR $\mathcal{DC}$, depending on the result of `isBase`. The reducer uses `conquer` to combine all the subproblems solutions. Here, r_0 is expected to compute the empty subproblem.

```
fun div(x) = ...
fun isBase(x,p,i) = ...
fun base(x,p,i) = ...
fun f_r(x,p,c,i) = ...
fun r_0(x) = ...

fun iterDiv(x,i) = div(x)[i]
fun subproblem(x,p,i) = if isBase(x,p,i)
                        then base(x,p,i)
                        else DC(p)     // recursive PCR DC call
fun conquer(r,x,p,c,i) = r ⊗ f_r(x,p,c,i)

lbnd DC = λx. 1
ubnd DC = λx. len(div(x))

PCR DC(x)
  par
    p = produce iterDiv x
    c = consume subproblem x p
    r = reduce conquer (r_0 x) x p c
```

Listing 3.8: Divide and conquer scheme in syntactic form.

Now, we state without proof:

Proposition 3.3. Let $\mathcal{DC}$ be a divide and conquer PCR with input parameter of type T and reducer variable of type D. Then, for all $x \in T$:

$$\mathcal{DC}(x) \;=\; \bigotimes_{i=1}^{len(div(x))} \overrightarrow{f}_r^{\,i}(x,\ \overrightarrow{div}(x),\ \overrightarrow{subproblem}(x,\ \overrightarrow{div}(x))) \tag{3.13}$$

where

$$\begin{aligned}
\overrightarrow{div}(x) &= i \in \mathbb{N} \mapsto div(x)[i] \\
\overrightarrow{subproblem}(x, p) &= i \in \mathbb{N} \mapsto \big(isBase(x, p, i) \;\rightarrow\; base(x, p, i),\ \mathcal{DC}(p^i)\big) \\
\overrightarrow{f}_r(x, p, c) &= i \in \mathbb{N} \mapsto f_r(x, p, c, i)
\end{aligned}$$

Remark 3.6.

1. The nature of the recursive problem decomposition process of *divide and conquer* is captured in proposition 3.3 using conditional notation mimicking the *subproblem* definition.

2. PCR $\mathcal{DC}$ composes with itself through the consumer depending on the result of *isBase*. However, the child PCR does not have access to the father's stream variables as in the ordinary composition scheme presented before —only the value p^i is passed in the call.

3. As we have $iterDiv(x, i) = div(x)[i]$ by definition, it is more convenient to work directly with *div*.

4. In this scheme, the producer component has no dependencies on itself, so all its assignments can be executed in parallel like the consumer.

In what follows, we discuss some concrete applications of the divide and conquer PCR scheme.

Example 3.3.1. MergeSort1 is a divide and conquer PCR implementing the very well known *Merge Sort* algorithm[16] for efficiently sorting lists. Indeed, this is usually considered the prototypical divide and conquer problem, so we introduce it now as the first example of this kind in listing 3.9 and figure 3.14. It works as follows:

[16]Firstly discovered by hungarian mathematician John von Neumann circa 1945.

1. The producer generates two halves, say L_1 and L_2, of the input list L.
2. Instances of the consumer work, in parallel, on each sub-list L_i. If $len(L_i) \leq 1$ then L_i is trivially sorted and is ready to be reduced (deemed a base case), otherwise the procedure starts again on input L_i.
3. The reducer merges the sorted halves of the input list.

```
fun div(L)     = let m = ⌊len(L)÷2⌋
                 in [L[1..m], L[m+1..len(L)]]
fun isBase(p)  = len(p) <= 1
fun base(p)    = p
fun ⊎(x,y)     = case x == []       → y
                      y == []       → x
                      x[1] <= y[1]  → x[1] ++ ⊎(tail(x),y)
                      x[1]  > y[1]  → y[1] ++ ⊎(x,tail(y))
PCR MergeSort1(L)
  par
    p = produce iterDiv L
    c = consume subproblem L p
    r = reduce ⊎ [] c
```

Listing 3.9: PCR MergeSort1 code. ⊎ is the binary merge operation on lists.

$$\begin{tikzcd}
 & & & L \arrow[dll] \arrow[drr] & & & \\
 & p^1 \arrow[d] & & & & p^2 \arrow[d] & \\
 & L^1 \arrow[dl] \arrow[dr] & & & & L^2 \arrow[dl] \arrow[dr] & \\
p^{1,1} \arrow[d] & & p^{1,2} \arrow[d] & & p^{2,1} \arrow[d] & & p^{2,2} \arrow[d] \\
L^{1,1} & & L^{1,2} & & L^{2,1} & & L^{2,2} \\
\vdots & & \vdots & & \vdots & & \vdots \\
r^{1,1} \arrow[d] & & r^{1,2} \arrow[d] & & r^{2,1} \arrow[d] & & r^{2,2} \arrow[d] \\
c^{1,1} \arrow[dr] & & c^{1,2} \arrow[dl] & & c^{2,1} \arrow[dr] & & c^{2,2} \arrow[dl] \\
 & r^1 \arrow[d] & & & & r^2 \arrow[d] & \\
 & c^1 \arrow[drr] & & & & c^2 \arrow[dll] & \\
 & & & r & & &
\end{tikzcd}$$

Figure 3.14: Generic dependence graph for PCR MergeSort1.

Let $L = [5, 4, 3, 2, 1]$. Following the graph, the result should be

$$\begin{aligned}
r &= \biguplus_{i=1}^{2} \overrightarrow{subproblem}^{i}(\overrightarrow{div}([5,4,3,2,1])) \\
&= [\,] \uplus \biguplus_{i=1}^{2} \overrightarrow{subproblem}^{i}(\overrightarrow{div}([5,4])) \uplus \biguplus_{i=1}^{2} \overrightarrow{subproblem}^{i}(\overrightarrow{div}([3,2,1])) \\
&= [\,] \uplus r_1 \uplus r_2 \\
&= [1,2,3,4,5]
\end{aligned}$$

where:

$$\begin{aligned}
r_1 &= [\,] \uplus \biguplus_{i=1}^{2} \overrightarrow{subproblem}^{i}(\overrightarrow{div}([5])) \uplus \biguplus_{i=1}^{2} \overrightarrow{subproblem}^{i}(\overrightarrow{div}([4])) \\
&= [\,] \uplus base(div([4])[1]) \uplus base(div([5])[2]) \\
&= [4,5] \\
r_2 &= [\,] \uplus \biguplus_{i=1}^{2} \overrightarrow{subproblem}^{i}(\overrightarrow{div}([3])) \uplus \biguplus_{i=1}^{2} \overrightarrow{subproblem}^{i}(\overrightarrow{div}([2,1])) \\
&= [\,] \uplus base(div([3])[1]) \uplus r_{2,2} \\
&= [1,2,3] \\
r_{2,2} &= [\,] \uplus \biguplus_{i=1}^{2} \overrightarrow{subproblem}^{i}(\overrightarrow{div}([2])) \uplus \biguplus_{i=1}^{2} \overrightarrow{subproblem}^{i}(\overrightarrow{div}([1])) \\
&= [\,] \uplus base(div([2])[1]) \uplus base(div([1])[2]) \\
&= [1,2]
\end{aligned}$$

MergeSort1 has a very regular behaviour, for any non base case input there are exactly two subproblems produced. In general, for any list L:

$$\begin{aligned}
MergeSort1(L) &= \biguplus_{i=1}^{2} \overrightarrow{subproblem}^{i}(\overrightarrow{div}(L)) \\
&= \biguplus_{i=1}^{2} \big(isBase(\overrightarrow{div}(L), i) \rightarrow base(\overrightarrow{div}(L), i), \\
&\qquad\qquad MergeSort1(\overrightarrow{div}^{i}(L))\big)
\end{aligned} \tag{3.14}$$

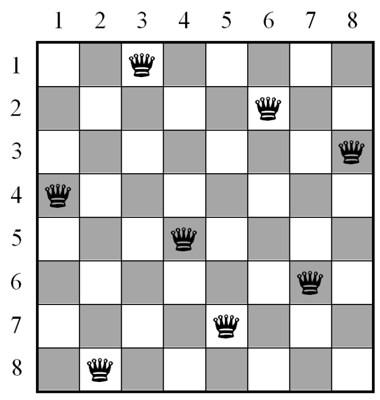
□

Example 3.3.2. NQueensDC is a divide and conquer PCR which resolves the N-Queens problem. This is the problem of placing N chess queens on an $N \times N$ chessboard so that no two queens attack each other as depicted in figure 3.15, which is to say no two queens share the same row, column, or diagonal.

A brute force approach may be computationally prohibitive. For a standard 8×8 chessboard, the number of ways we can choose positions for 8 (identical) queens from 64 squares amounts to

$$\binom{64}{8} = \frac{64!}{!8 \cdot !(64-8)} = 4426165368$$

which is huge considering there are only 92 possible solutions.[17] For this kind of problem, an useful technique is to do a systematic search, normally called *backtracking*, where one incrementally builds candidates to the solution/s and discards partial candidates (backtracks) as soon as it is determined (by some heuristic rule) they cannot possibly be valid solutions, therefore reducing the number of possibilities. On a idealized machine, the backtracking could be done all in parallel, but this is not the case in practice where there are finite resources.



$$\begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ [3, & 6, & 8, & 1, & 4, & 7, & 5, & 2] \end{array}$$

Figure 3.15: One possible solution for a chessboard of size 8×8: (Left) graphical representation, (Right) linear representation as a list.

A chessboard configuration of size $N \times N$ is linearly represented as a list C of length N where index $k \in 1..N$ indicates the row and value $C[k]$ indicates the column of a queen position (see figure 3.15). Value 0 is reserved to denote an empty row. The condition for validly placing a queen at row i and column j on configuration C might be formally

[17] A formula for the exact number of solutions, or even for its asymptotic behaviour, is still unknown. [28]

stated as:

$$\begin{array}{ll} \quad C[i] = 0 & \text{Not in same row} \\ \wedge\ \forall\, k \in 1..N : C[k] \neq j & \text{Not in same column} \\ \wedge\ \forall\, k \in 1..N : C[k] \neq 0 \ \Rightarrow\ |C[k] - j| \neq |k - i| & \text{Not in same diagonal} \end{array}$$

Our PCR solution is illustrated in listing 3.10. It is a divide and conquer approach where not viable candidates are marked as (empty) base cases in order to search no further on those paths. For example, consider configuration $[0, 2, 0, 1]$ where first and third row are empty, is possible to place a queen in the third row but is not possible for the first row, thus the configuration can be discarded. Initially, input is assumed to be the zero configuration $[0_1, \ldots, 0_N]$. It works as follows:

1. The producer generates from input configuration C new candidate configurations, say C_i, for each row of C where is possible to place a queen in the first valid column.

2. Instances of the consumer work, in parallel, on each candidate configuration C_i. If C_i is complete (all the rows have a queen) or it is not possible to add more queens in the empty rows without conflict then C_i is considered a base case, otherwise the procedure starts again on input C_i.

3. The reducer joins the sets of solutions found.

```
fun validPos(C,i,j) = ...
fun addQInRow(C,i) = ...
fun canAddQInRow(C,i) = ...
fun canAddQueens(C) = ...

fun complete(C) = all (λj. j != 0) C
fun div(C) = [addQInRow(C,i) | 1 <= i <= len(C), canAddQInRow(C,i)]
fun isBase(p) = complete(p) or not canAddQueens(p)
fun base(p) = if complete(p) then {p} else {}

PCR NQueensDC(C)
  par
    p = produce iterDiv C
    c = consume subproblem C p
    r = reduce ∪ {} c
```

Listing 3.10: PCR NQueensDC code.

In general, for any zero configuration C:

$$\begin{aligned} \mathit{NQueensDC}(C) &= \bigcup_{i=1}^{len(div(C))} \overrightarrow{\mathit{subproblem}}^{\,i}(\overrightarrow{div}(C)) \qquad (3.15)\\ &= \bigcup_{i=1}^{len(div(C))} \big(\mathit{isBase}(\overrightarrow{div}(C), i) \rightarrow \mathit{base}(\overrightarrow{div}(C), i),\\ &\qquad\qquad \mathit{NQueensDC}(\overrightarrow{div}^{\,i}(C))\big) \end{aligned}$$

□

3.3.3 Composition through reducer

In this section we introduce the scheme for PCR composition through reducer. This form of composition is somewhat more restrictive than in the consumer case considered before, because here the inner PCR adopts the role of the combiner for the reducer operation of the outer PCR.

Definition 3.5 (Reducer composition scheme)**.** Let $\mathcal{A}$ and $\mathcal{B}$ be PCRs with k_1 and k_2 consumers respectively. The computation of $\mathcal{A}$ composed with $\mathcal{B}$ through reducer, on input x_1, consists in the following operations for each $i \in I_{x_1}$ and $j \in J_{x_2^i}$:

PCR $\mathcal{A}$:

$$\begin{aligned} p_1^i &:= f_{p_1}(x_1, p_1, i)\\ c_{1,1}^i &:= f_{c_{1,1}}(x_1, p_1, i)\\ c_{1,2}^i &:= f_{c_{1,2}}(x_1, p_1, c_{1,1}, i)\\ &\vdots\\ c_{1,k_1}^i &:= f_{c_{1,k_1}}(x_1, p_1, c_{1,1}, \ldots, c_{1,k_1-1}, i)\\ r_1 &:= r_2^i \ \text{ if } \ \mathit{end}_{\mathcal{B}^i} \end{aligned}$$

PCR $\mathcal{B}$:

$$\begin{aligned} x_2^i &:= \big(r_1, f_{r_1}(x_1, p_1, c_{1,1}, \ldots, c_{1,k_1}, i)\big)\\ &\quad \text{if } \neg\,\exists\, k \in I_{x_1} : i \neq k \wedge \mathit{wrt}(x_2^k) \wedge \neg \mathit{red}(k)\\ p_2^{i,j} &:= f_{p_2}(x_2^i, p_2^i, j)\\ c_{2,1}^{i,j} &:= f_{c_{2,1}}(x_2^i, p_2^i, j)\\ c_{2,2}^{i,j} &:= f_{c_{2,2}}(x_2^i, p_2^i, c_{2,1}^i, j)\\ &\vdots\\ c_{2,k_2}^{i,j} &:= f_{c_{2,k_2}}(x_2^i, p_2^i, c_{2,1}^i, \ldots, c_{2,k_2-1}^i, j)\\ r_2^i &:= r_2^i \oplus f_{r_2}(x_2^i, p_2^i, c_{2,1}^i, \ldots, c_{2,k_2}^i, j) \end{aligned}$$

where:

- $I_{x_1} = \{i \in \mathit{lBnd}_1(x_1)..\mathit{uBnd}_1(x_1) : \mathit{prop}_1(i)\}$
  $J_{x_2^i} = \{j \in \mathit{lBnd}_2(x_2^i)..\mathit{uBnd}_2(x_2^i) : \mathit{prop}_2(j)\}$ for any $i \in I_{x_1}$

- p_1, $c_{1,1}$, ..., c_{1,k_1}, r_1 are the variables of PCR $\mathcal{A}$, and
  f_{p_1}, $f_{c_{1,1}}$, ..., $f_{c_{1,k_1}}$, f_{r_1} are the *basic functions* associated to these variables.
- p_2, $c_{2,1}$, ..., c_{2,k_2}, r_2 are the variables of PCR $\mathcal{B}$, and
  f_{p_2}, $f_{c_{2,1}}$, ..., $f_{c_{2,k_2}}$, f_{r_2} are the *basic functions* associated to these variables.
- $\oplus$ is the combiner operation of PCR $\mathcal{B}$ and $x \otimes y = \mathcal{B}(x, y)$ is the combiner operation of PCR $\mathcal{A}$.

Remark 3.7.

1. The combiner operation of $\mathcal{A}$ is the nested PCR $\mathcal{B}$ acting on binary inputs. The value of r_1 at reduction i comes from the output of the i-th instance of PCR $\mathcal{B}$ (i.e. $\mathcal{B}^i$) which is the *final* value of r_2^i.
2. $\mathcal{B}$ is assumed to be a basic PCR, whose input is written by $\mathcal{A}$ in variable x_2. In general, any x_2^i is a pair of the form $\big(r_1, f_{r_1}(x_1, p_1, c_{1,1}, \ldots, c_{1,k_1}, i)\big)$, thus $\mathcal{B}$ is treated as a binary operation so that $x \otimes y = \mathcal{B}(x, y)$.
3. The algebraic properties of combiner $\otimes$ *depend* on the constituting elements of $\mathcal{B}$. This is considered in proposition 3.4.

Note that reduction in $\mathcal{A}$ is not an *atomic* operation, because combiner $\otimes$ (acting on partial value r_1) is implemented by the nested PCR $\mathcal{B}$ itself. For this reason, to avoid *race conditions* on r_1, it is enough to write $\mathcal{B}$ inputs on variable x_2 when *there is not another written input for which $\mathcal{B}$ has not terminated.* This condition, which works like a *lock* mechanism, is more formally stated as follows:

$$\neg\, \exists\, k \in I_{x_1} \,:\, i \neq k \;\wedge\; wrt(x_2^k) \;\wedge\; \neg red(k) \tag{3.16}$$

It should be noted that because this lock is very *coarse-grained*, the reducer is restricted to operate necessarily in serial mode, as there could not be different instances of $\mathcal{B}$ working in parallel. However, if the father $\mathcal{A}$ is a *divide and conquer* PCR, the different spawned instances of $\mathcal{A}$ would have their own *local* reducers, which implies that reducers would be necessarily serial only at the local level.

```
// basic functions $f_{p_1}$, $f_{c_{1,1}}$, ..., $f_{c_{1,k_1}}$, $f_{r_1}$, $r_{1,0}$ for $\mathcal{A}$
// lbnd, ubnd and prop for $\mathcal{A}$

PCR $\mathcal{A}$($x_1$)
  par
    $p_1$     = produce $f_{p_1}$ $x_1$ $p_1$
    $c_{1,1}$ = consume $f_{c_{1,1}}$ $x_1$ $p_1$
    $c_{1,2}$ = consume $f_{c_{1,2}}$ $x_1$ $p_1$ $c_{1,1}$
              ⋮
    $c_{1,k_1}$ = consume $f_{c_{1,k_1}}$ $x$ $p_1$ $c_{1,1}$ $c_{1,2}$ ... $c_{1,k_1-1}$
    $r_1$     = reduce $\mathcal{B}$ ($r_{1,0}$ $x_1$) ($f_{r_1}$ $x_1$ $p_1$ $c_{1,1}$ ... $c_{1,k_1}$)  // PCR $\mathcal{B}$ call

// basic functions $f_{p_2}$, $f_{c_{2,1}}$, ..., $f_{c_{2,k_2}}$, $f_{r_2}$, $r_{2,0}$ for $\mathcal{B}$
// lbnd, ubnd and prop for $\mathcal{B}$

PCR $\mathcal{B}$($x_2$)
  par
    $p_2$     = produce $f_{p_2}$ $x_2$ $p_2$
    $c_{2,1}$ = consume $f_{c_{2,1}}$ $x_2$ $p_2$
    $c_{2,2}$ = consume $f_{c_{2,2}}$ $x_2$ $p_2$ $c_{2,1}$
              ⋮
    $c_{2,k_2}$ = consume $f_{c_{2,k_2}}$ $x_2$ $p_2$ $c_{2,1}$ $c_{2,2}$ ... $c_{2,k_2-1}$
    $r_2$     = reduce $\oplus$ ($r_{2,0}$ $x_2$) ($f_{r_2}$ $x_2$ $p_2$ $c_{2,1}$ ... $c_{2,k_2}$)
```

Listing 3.11: Reducer composition scheme in syntactic form. $\mathcal{B}(x_2)$ expands to $\mathcal{B}(x, y)$ where $x, y \in D$.

Proposition 3.4. Let $\mathcal{A}$ and $\mathcal{B}$ be PCRs with k_1 and k_2 consumers, respectively, so that $\mathcal{A}$ with input parameter of type T and reducer variable of type D is composed with $\mathcal{B}$ through reducer. Then, for all $x_1 \in T$, the output of $\mathcal{A}$ at input x_1, to be written $\mathcal{A}(x_1)$, is determined if $\mathcal{A}$ terminates at input x_1, and it holds: $\mathcal{A}(x_1) = \bigotimes_{i \in I_{x_1}} \vec{f}^{\,i}_{\mathcal{A}}$ for some function $\vec{f}_{\mathcal{A}} \in \vec{D}$.

Proof. Let $x_1 \in T$. In general, we have that $\mathcal{B}$ is a basic PCR with input type $D \times D$. It is also the case that $\mathcal{B}$ output type must be D. According to proposition 3.1, for any $x_2^i \in D \times D$ where $i \in I_{x_1}$:

$$\mathcal{B}(x_2^i) = \bigoplus_{j \in J_{x_2^i}} \vec{f}^{\,j}_{\mathcal{B}} \quad \text{for some} \quad \vec{f}_{\mathcal{B}} \in \vec{D}$$

Recall that $x \otimes y = \mathcal{B}(x, y)$. Now, is $(D, \otimes)$ an abelian monoid? It does not need to. For example, take $\vec{f}_{\mathcal{B}}(x, y) = y$, then even assuming $J_{(x,y)} = J_{(y,x)}$ we would have $x \otimes y \neq y \otimes x$ for different x and y.

To ensure associativity and commutativity of $\otimes$ it is sufficient to assume a constant iteration space J (i.e. one that does not depend on the input) and the following properties for $\vec{f}_{\mathcal{B}}$:

$$\vec{f}_{\mathcal{B}}(\vec{f}_{\mathcal{B}}(x,y),z) \;=\; \vec{f}_{\mathcal{B}}(x,\vec{f}_{\mathcal{B}}(y,z)) \tag{3.17}$$

$$\vec{f}_{\mathcal{B}}(x,y) \;=\; \vec{f}_{\mathcal{B}}(y,x) \tag{3.18}$$

$$\vec{f}_{\mathcal{B}}(x \oplus y,z) \;=\; \vec{f}_{\mathcal{B}}(x,z) \oplus \vec{f}_{\mathcal{B}}(y,z) \tag{3.19}$$

So, we have for any $x, y, z \in D$:

$$x \otimes y \;=\; \bigoplus_{j\,\in\,J} \vec{f}^{\,j}_{\mathcal{B}}(x,y) \;\overset{(3.18)}{=}\; \bigoplus_{j\,\in\,J} \vec{f}^{\,j}_{\mathcal{B}}(y,x) \;=\; y \otimes x$$

and

$$\begin{aligned}
(x \otimes y) \otimes z \;&=\; \bigoplus_{j\,\in\,J} \vec{f}^{\,j}_{\mathcal{B}}\Big(\bigoplus_{k\,\in\,J} \vec{f}^{\,k}_{\mathcal{B}}(x,y),\ z\Big) \\
&=\; \bigoplus_{j\,\in\,J}\bigoplus_{k\,\in\,J} \vec{f}^{\,j}_{\mathcal{B}}\big(\vec{f}^{\,k}_{\mathcal{B}}(x,y),\ z\big) \qquad \text{by 3.19} \\
&=\; \bigoplus_{j\,\in\,J}\bigoplus_{k\,\in\,J} \vec{f}^{\,j}_{\mathcal{B}}\big(x,\ \vec{f}^{\,k}_{\mathcal{B}}(y,z)\big) \qquad \text{by 3.17} \\
&=\; \bigoplus_{j\,\in\,J} \vec{f}^{\,j}_{\mathcal{B}}\big(x,\ \bigoplus_{k\,\in\,J} \vec{f}^{\,k}_{\mathcal{B}}(y,z)\big) \qquad \text{by 3.18 and 3.19} \\
&=\; x \otimes (y \otimes z)
\end{aligned}$$

For the identity of $\otimes$, we just assume is the same identity of $\oplus$.

Now, proceed exactly like in proposition 3.1 to obtain

$$\mathcal{A}(x_1) \;=\; \bigotimes_{i\,\in\,I_{x_1}} \vec{f}^{\,i}_{\mathcal{A}} \tag{3.20}$$

where $\vec{f}_{\mathcal{A}} \;=\; \vec{f}_{r_1}(x_1, \vec{g}_{p_1}, \vec{f}_{c_{1,1}}, \ldots, \vec{f}_{c_{1,k_1}})$. $\square$

In particular, for two single consumer PCRs $\mathcal{A}$ and $\mathcal{B}$, equation 3.20 expands to:

$$\mathcal{A}(x_1) \;=\; \bigotimes_{i\,\in\,I_{x_1}} \vec{f}^{\,i}_{r_1}\big(x_1,\ \vec{g}_{p_1}(x_1),\ \vec{f}_{c_1}(x_1,\ \vec{g}_{p_1}(x_1))\big) \tag{3.21}$$

where $x \otimes y \;=\; \mathcal{B}(x,y) \;=\; \bigoplus_{j\,\in\,J} \vec{f}^{\,j}_{r_2}\big(x, y,\ \vec{g}_{p_2}(x,y),\ \vec{f}_{c_2}(x,y,\ \vec{g}_{p_2}(x,y))\big)$.

3.3.3.1 Example: MergeSort with parallel Merge

In general, the most straightforward way to parallelize *divide-and-conquer* based algorithms is to run the recursive calls in parallel. Indeed, we previously presented MergeSort1 (example 3.3.1), a divide and conquer PCR implementation of the merge sort algorithm which operates in the aforementioned fashion. But, how good is MergeSort1 compared to its serial counterpart?. For an input list of size n, the total *work* to be done at any instance of MergeSort1 is dominated by the time spent on two subproblems (of roughly same size, i.e. $n/2$) plus the time for two merge operations (linear on n) [18]. This is expressed as the recurrence

$$MS1_1(n) \;=\; 2 \cdot MS1_1(\tfrac{n}{2}) + 2 \cdot \Theta(n) \;=\; \Theta(n \cdot log(n))$$

Since the two subproblems are (logically) solved in parallel, the *span* should be

$$MS1_\infty(n) \;=\; MS1_\infty(\tfrac{n}{2}) + 2 \cdot \Theta(n) \;=\; \Theta(n)$$

Therefore, speedup is

$$\frac{MS1_1(n)}{MS1_\infty(n)} \;=\; \frac{\Theta(n \cdot log(n))}{\Theta(n)} \;=\; \Theta(log(n)) \tag{3.22}$$

which is a slow growing improvement. Can we do better?. In a *divide and conquer* algorithm, either the *divide* or *conquer* step can sometimes become a bottleneck if performed serially. For MergeSort1, the divide step is trivial while the conquer step is a merge operation between two lists. The strategy for merge is traditionally taught as an (imperative or recursive) $\Theta(n)$ algorithm, essentially the one employed in MergeSort1, moreover, it doesn't easily lends to parallelism (one could say it is *inherently* sequential). Contrary to common practice, this is one of that cases in which it is better to design a new parallel algorithm from scratch instead of trying to parallelize an existing serial one.

As an alternative we introduce PCR Merge, a divide and conquer PCR solution for the merge problem, adapted from [2] and [29]. Here we assume the serial merge function and the PCR are functionally equivalent. The key idea in this approach is to generate

[18] Actually, one of these two merges (reductions) is always trivially $[\,] \uplus r$ for some list r, having a constant cost. Thus we could countabilize a single merge operation like in the ordinary known Merge Sort algorithm, but this doesn't affect asymptotic behaviour.

independent sub-merge problems to be solved and then joined. It works as follows:

1. The producer generates two sub-merge pairs, say $(L_{1,1}, L_{1,2})$ and $(L_{2,1}, L_{2,2})$, of the input lists L_1 and L_2 according to the following strategy. Without loss of generality, assume that list L_1 is at least as long as sequence L_2, then:
   (a) Split L_1 into two sub-lists $L_{1,1}$ and $L_{1,2}$ of approximately equal length.
   (b) Let e be the first key of $L_{1,2}$. Use *binary search* on L_2 to find the point where e could be inserted into L_2.
   (c) Split L_2 at that point into two sub-lists $L_{2,1}$ and $L_{2,2}$.
2. Instances of the consumer work, in parallel, on each sub-merge pair $(L_{i,1}, L_{i,2})$. If $\mathit{len}(L_{i,1}) \leq 1$ and $\mathit{len}(L_{i,2}) \leq 1$ then produce a base case *merging* $L_{i,1}$ with $L_{i,2}$, otherwise the procedure starts again on input pair $L_{i,1}$ and $L_{i,2}$.
3. The reducer concatenates the resulting parts of the sub-merges.

Listing 3.12 presents PCR MergeSort2, an adaptation of MergeSort1 that composes with PCR Merge through the reducer. This is a divide-and-conquer PCR nested in another divide-and-conquer PCR. Both PCRs always generate two sub-problems, and both share same identity element for their combiner operations. A small but important detail is that PCR Merge uses a fixed order reducer (recall section 3.2.1) because its combiner `++` is not commutative, which is interesting considering that this PCR computes a commutative binary operation.

```
fun div1(L)      = let m = ⌊len(L)÷2⌋
                   in [L[1..m], L[m+1..len(L)]]
fun isBase1(p) = len(p) <= 1
fun base1(p)     = p

PCR MergeSort2(L)
  par
    p1 = produce iterDiv1 L
    c1 = consume subproblem1 L p1
    r1 = reduce Merge [] c1

fun binarySearch(L,e) = ...
fun div2(L1,L2) = [(L11,L21), (L12,L22)]  // ensure len(L1) >= len(L2)
                    where m   = ⌊len(L)÷2⌋
                          L11 = L1[1..m]
                          L12 = L1[m+1..len(L1)]
```

```
                           k   = binarySearch(L2, L12[1])
                           L21 = L2[1..k]
                           L22 = L2[k+1..len(L2)]
fun isBase2(p) = len(p[1]) <= 1 and len(p[2]) <= 1
fun base2(p)   = ⊎(p[1],p[2])  // trivial merge

dep r2(i-1) → r2(i)

PCR Merge(L1,L2)
  par
    p2 = produce iterDiv2 L1 L2
    c2 = consume subproblem2 L1 L2 p2
    r2 = reduce (++) [] c2
```

Listing 3.12: PCR MergeSort2 code.

But, how better is MergeSort2?. The change is that MergeSort2 uses a parallel merge, so we need to analyze this new PCR component first. The details are a bit more involved in comparison to what we did for MergeSort1 previously and we refer the reader to details (mutatis mutandis) in [29]. A crucial assumption we make here is that combiner `++` works in constant time[19], so the binary search (plus the work on sub-problems) dominates the total work at any instance of PCR Merge. Let n be the total length of both input lists, the *work* and *span* for parallel merge are given by:

$$M_1(n) = M_1(\alpha \cdot n) + M_1((1-\alpha)\cdot n) + \Theta(log(n)) = \Theta(n) \quad , \tfrac{1}{4} \leq \alpha \leq \tfrac{3}{4} \tag{3.23}$$

$$M_\infty(n) \leq M_\infty(\tfrac{3}{4}n) + \Theta(log(n)) = \Theta(log^2(n)) \tag{3.24}$$

For the *span*, note that in the worst case, half of L_1 must be merged with all of L_2, that is at most $\frac{3}{4}n$ elements.

Now we analyze MergeSort2. The work of inner PCR Merge is asymptotically the same as the ordinary serial merge, thus total work is the same as in MergeSort1:

$$MS2_1(n) = MS1_1(n) = \Theta(n \cdot log(n))$$

[19]One possibility to satisfy this tight assumption is to have a mutable list data structure with constant-time access to the tip of the tail. As a downside, this might be a cache-unfriendly option.

But the span gets a benefit:

$$\begin{aligned} MS2_{\infty}(n) &= MS2_{\infty}(\tfrac{n}{2}) + 2 \cdot M_{\infty}(n) \\ &= MS2_{\infty}(\tfrac{n}{2}) + 2 \cdot \Theta(log^2(n)) \\ &= \Theta(log^3(n)) \end{aligned}$$

Therefore, speedup now is

$$\frac{MS2_1(n)}{MS2_{\infty}(n)} = \frac{\Theta(n \cdot log(n))}{\Theta(log^3(n))} = \Theta\left(\frac{n}{log^2(n)}\right) \tag{3.25}$$

which is a bigger improvement than 3.22. Empirical assessment of this analysis should account for parallel scheduling overhead, memory bandwdith and cache behaviour.

3.3.3.2 The indexing mechanism revisited

In the divide and conquer PCR scheme we presented, recursion is modeled by repeated composition through the consumer with itself. It is evident that each recursive call does not create or use different variables for each new instance[20] —there is *only one* set of variables for the PCR. Even so, there is no ambiguity on their content among the different recursive instances. This is because of how indexing works, as the index of each new recursive instance grows along the recursion depth, allowing to clearly distinguishing each instance.

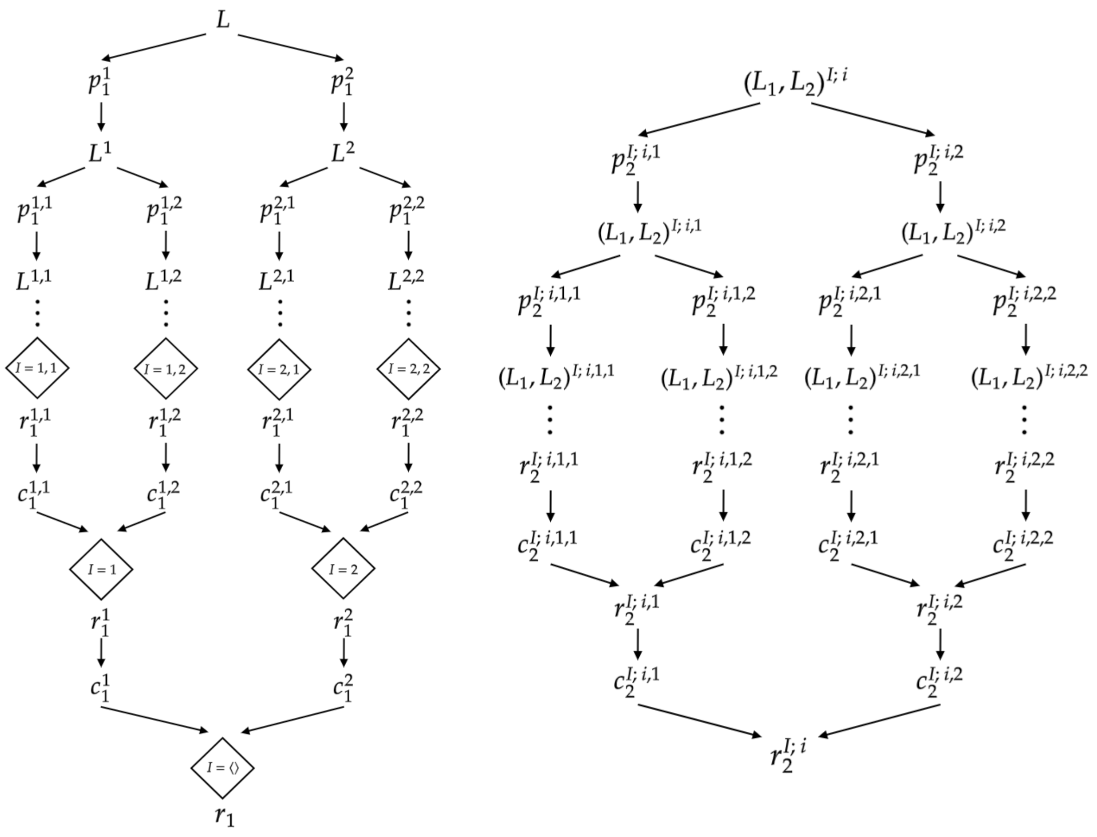
However, a DC scheme nested inside another DC scheme, like our previous PCR Merge-Sort2, poses a problem for indexing as we have been handling it so far. In this setting, recursion of the inner DC (Merge) may happen multiple times at different depths of the outer DC recursion, allowing for different recursions of the inner DC to use the same sequences of indices, thus incurring in ambiguity.[21]

To cope with the aforementioned situation, it is enough for the inner PCR indexing to also keep track of the depth of the outer recursion where the inner recursion takes place. A simple way to do this is to index the inner PCR by pairs of sequences $I; J$ where I is

[20] Surely, this would be the case in a concrete implementation of the PCR pattern.

[21] In fact, we detected this issue after formalizing in TLA$^+$ and doing model checking on large enough inputs.

the father index (serving as a depth mark) and J is the proper index of the inner PCR which will grow along with his recursion as normal. This also generalizes in the obvious way using tuples of sequences. The approach can be appreciated in figure 3.16.



L
p_1^1 p_1^2
L^1 L^2
$p_1^{1,1}$ $p_1^{1,2}$ $p_1^{2,1}$ $p_1^{2,2}$
$L^{1,1}$ $L^{1,2}$ $L^{2,1}$ $L^{2,2}$
$I = 1,1$ $I = 1,2$ $I = 2,1$ $I = 2,2$
$r_1^{1,1}$ $r_1^{1,2}$ $r_1^{2,1}$ $r_1^{2,2}$
$c_1^{1,1}$ $c_1^{1,2}$ $c_1^{2,1}$ $c_1^{2,2}$
$I = 1$ $I = 2$
r_1^1 r_1^2
c_1^1 c_1^2
$I = \langle\rangle$
r_1
$(L_1, L_2)^{I;\, i}$
$p_2^{I;\, i,1}$ $p_2^{I;\, i,2}$
$(L_1, L_2)^{I;\, i,1}$ $(L_1, L_2)^{I;\, i,2}$
$p_2^{I;\, i,1,1}$ $p_2^{I;\, i,1,2}$ $p_2^{I;\, i,2,1}$ $p_2^{I;\, i,2,2}$
$(L_1, L_2)^{I;\, i,1,1}$ $(L_1, L_2)^{I;\, i,1,2}$ $(L_1, L_2)^{I;\, i,2,1}$ $(L_1, L_2)^{I;\, i,2,2}$
$r_2^{I;\, i,1,1}$ $r_2^{I;\, i,1,2}$ $r_2^{I;\, i,2,1}$ $r_2^{I;\, i,2,2}$
$c_2^{I;\, i,1,1}$ $c_2^{I;\, i,1,2}$ $c_2^{I;\, i,2,1}$ $c_2^{I;\, i,2,2}$
$r_2^{I;\, i,1}$ $r_2^{I;\, i,2}$
$c_2^{I;\, i,1}$ $c_2^{I;\, i,2}$
$r_2^{I;\, i}$

Figure 3.16: Generic dependence graph for PCR MergeSort2: (Left) outer PCR, (Right) inner PCR Merge. Each $\diamond$ in the outer PCR expands to the right DAG instantiated with an appropriate pair denoted by $I; i$ where I is the father index and i is the current assignment at outer scope.[22]

3.4 PCR extension: *iterate*

In this section we consider an extension to the basic PCR elements described initially in table 3.1. The **`iterate`** construct, presented in table 3.3, is regarded as a special kind of consumer which provides a topology that is appropriate for parallel algorithms where a

[22]Actually, as there are always two merge operations because the first of them trivially combines with the identity, each $\diamond$ expand to exactly two DAGs of PCR Merge arranged in sequence. The first DAG will be shallow, because recursion is not needed to combine with the identity, is just a base case.

manager PCR iterates a function (a *basic function* or another PCR), say f, starting from some initial value v_0

$$v_0 \xrightarrow{1} f(v_0) \xrightarrow{2} f(f(v_0)) \xrightarrow{3} f(f(f(v_0))) \xrightarrow{4} \cdots$$

until some convergence condition is met. Each iteration depends on the result of the previous one, thus a new iteration can't start until the whole previous iteration has finished. A common use case is given by any of the various iterative methods known to solve linear systems for engineering applications. For example, a PCR could use **`iterate`** on the PCR RedBlack (example 3.1.3) for an appropriate convergence condition.

Type	**Syntax**	
Iterative consumer	c_e = **`iterate`** $cnd\ f_{c_e}\ (v_0\ x)\ x\ p\ c_1\ \ldots\ c_{e-1}$	where f_{c_e} is some iterable function with initial value v_0 and termination condition cnd.

Table 3.3: PCR syntax extension for the iterate construct.

In each iteration, it is possible for the condition function cnd to look behind to decide on previous values. But like **`produce`**, it is restricted to look-behind operations, as look-ahead would generate a deadlock. The initial value of iteration is given by $v_0(x)$, and sometimes we refer to it as v_0.

It should be noted that **`iterate`** was originally introduced in [8] with the intention to be used as a standalone PCR primitive (in other words, a standalone special consumer). In the present work, we allow **`iterate`** to be used in place of any consumer alongside other components in a PCR, then the standalone version can be obtained as a special case. Next, we present an abstract model for a PCR with an iterative consumer.

Definition 3.6 (Iterative PCR scheme)**.** Let $\mathcal{A}$ be a PCR with k consumers and iterable function f_{c_e} in consumer $e \in 1..k$. The computation of $\mathcal{A}$ on input x consists in the following operations for each $i \in I_x$ and $j \in \mathbb{N}^+$:

PCR $\mathcal{A}$:

$$\begin{aligned}
p^i &:= f_p(x, p, i)\\
c_1^i &:= f_{c_1}(x, p, i)\\
c_2^i &:= f_{c_2}(x, p, c_1, i)\\
&\vdots\\
c_e^i &:= s^{i,j} \quad \text{if} \quad cnd(s^i, j)\\
&\vdots\\
c_k^i &:= f_{c_k}(x, p, c_1, \ldots, c_{k-1}, i)\\
r &:= r \otimes f_r(x, p_1, c_1, \ldots, c_k, i)
\end{aligned}$$

If f_{c_e} is a basic function:

$$\begin{aligned}
s^{i,1} &:= v_0(x)\\
s^{i,j} &:= f_{c_e}(s^{i,j-1}, x, p, i) \quad \text{if } j > 1 \wedge \neg cnd(s^i, j-1)
\end{aligned}$$

If f_{c_e} is the function computed by some PCR $\mathcal{B}$ with input x_2 and output r_2:

$$\begin{aligned}
s^{i,1} &:= v_0(x)\\
x_2^{i,j} &:= (s^{i,j-1}, x, p, i) \quad \text{if } j > 1 \wedge \neg cnd(s^i, j-1)\\
s^{i,j} &:= r_2^{i,j} \quad \text{if } j > 1 \wedge end_{\mathcal{B}^{i,j}}
\end{aligned}$$

where:

- $I_x = \{i \in lBnd(x)..uBnd(x) : prop(i)\}$
- p, c_1, ..., c_k, r are the variables of PCR $\mathcal{A}$, and f_p, f_{c_1}, ..., f_{c_k}, f_r (except f_{c_e}) are the *basic functions* associated to these variables.
- s is an auxiliary variable tracking the history of iteration values for each assignment, i.e. a stream for each assignment.
- cnd is a convergence condition on stream s^i and the current iterator index j (not to be confused with the iteration space index assignment i).
- $\otimes$ is the combiner operation of PCR $\mathcal{A}$.

```
// basic functions $f_p$, $f_{c_1}$, ..., $f_{c_k}$, $f_r$, $r_0$ for $\mathcal{A}$
// lbnd, ubnd and prop for $\mathcal{A}$

fun $v_0$($x$) = ...
fun $cnd$($s$,$j$) = ...

PCR $\mathcal{A}$($x$)
  par
    $p$   = produce $f_p$ $x$ $p$
    $c_1$ = consume $f_{c_1}$ $x$ $p$
    $c_2$ = consume $f_{c_2}$ $x$ $p$ $c_1$
          $\vdots$
    $c_e$ = iterate $cnd$ $f_{c_e}$ ($v_0$ $x$) $x$ $p$ $c_1$ ... $c_{e-1}$  // iteration over $f_{c_e}$
          $\vdots$
    $c_k$ = consume $f_{c_k}$ $x$ $p$ $c_1$ $c_2$ ... $c_{k-1}$
    $r$   = reduce $\otimes$ ($r_0$ $x$) ($f_r$ $x$ $p$ $c_1$ ... $c_k$)
```

Listing 3.13: Iterative PCR scheme in syntactic form.

Remark 3.8.

1. An iterative consumer introduces its own iteration space, because for each PCR assignment i there is an iterator indexed by $j \in \mathbb{N}^+$ evolving incrementally and independently from the other assignments. Consequently, s is a multidimensional indexed stream variable.

2. The iterable function f_{c_e} is used to iterate over its first argument, starting from v_0, but it also has access to input x, previous output variables and current assignment i if it needs them in any iteration.

3. In case f_{c_e} is another PCR, say $\mathcal{B}$:
   (a) Remark (1) applies similarly for variables in $\mathcal{B}$.
   (b) The type of first input parameter of $\mathcal{B}$ and his output type should coincide with that of initial value v_0.
   (c) This situation might be seen as a *composition through consumer* occurring, for each father PCR assignment, an arbitrary number of times.

Now, we re-state proposition 3.1 but considering our new addition.

Proposition 3.5. Let $\mathcal{A}$ be a PCR with k consumers and iterable function f_{c_e} in consumer $e \in 1..k$. Assume input parameter of type T, reducer variable of type D and $v_0 \in Z$. Then, for all $x \in T$, if $\mathcal{A}$ terminates at input x then the output of $\mathcal{A}$ at x, to be written $\mathcal{A}(x)$, is determined and it holds: $\mathcal{A}(x) = \bigotimes_{i \in I_x} \overrightarrow{f}^{\,i}_{\mathcal{A}}$ for some function $\overrightarrow{f}_{\mathcal{A}} \in \overrightarrow{D}$.

Proof. First, note that the only difference between this and proposition 3.1 for a basic PCR is that we now allow an iterable function f_{c_e} for consumer c_e. This can be a basic function or another PCR but as we already saw on proposition 3.2 both cases can be treated seamlessly. In general, f_{c_e} is a function of type

$$Z \times T \times \overrightarrow{T}_p \times \overrightarrow{T}_{c_1} \times \cdots \times \overrightarrow{T}_{c_{e-1}} \times \mathbb{N} \to Z$$

For simplicity, let's fix all but the first parameter and write $f_{c_e} : Z \to Z$.

The iteration over $f_{c_e} : Z \to Z$ can be expressed as the following recursive definition:

$$\mathit{iter}(s) \;=\; \begin{cases} s & , \mathit{cnd}(s, \mathit{len}(s)) \\ \mathit{iter}(s \circ \langle f_{c_e}(\mathit{last}(s))\rangle) & , \text{otherwise} \end{cases}$$

so that for any initial value v_0 it unfolds into the sequence

$$\mathit{iter}(\langle v_0\rangle) \;=\; \langle v_0,\ f_{c_e}(v_0),\ f_{c_e}(f_{c_e}(v_0)),\ f_{c_e}(f_{c_e}(v_0)),\ f_{c_e}(f_{c_e}(f_{c_e}(v_0))),\ \ldots\rangle$$

of arbitrary length depending on condition *cnd*, which we assume to hold at some point. This sequence is an explicit representation of the stream s, from the second step and onwards is safe for *cnd* to look behind on previous values. The *final* result should be the last value in the sequence, i.e. $\mathit{last}(\mathit{iter}(\langle v_0\rangle))$.

Now, proceed exactly like in proposition 3.1 except that in this case the stream function for consumer $e \in 1..k$ is defined as:

$$\vec{f}_{c_e}(x, p, c_1, \ldots, c_{e-1}) \;=\; i \in \mathbb{N} \mapsto \mathit{last}(\mathit{iter}(\langle v_0\rangle, x, p, c_1, \ldots, c_{e-1}, i))$$

and then obtain

$$\mathcal{A}(x) \;=\; \bigotimes_{i \in I_x} \vec{f}^{\,i}_{\mathcal{A}} \tag{3.26}$$

where $\vec{f}_{\mathcal{A}} = \vec{f}_r(x, \vec{g}_p, \vec{f}_{c_1}, \ldots, \vec{f}_{c_e}, \ldots, \vec{f}_{c_k})$. □

In particular, for a single consumer PCR $\mathcal{A}$, equation 3.26 expands to:

$$\mathcal{A}(x) \;=\; \bigotimes_{i \in I_x} \vec{f}^{\,i}_r\big(x,\ \vec{g}_p(x),\ i \in \mathbb{N} \mapsto \mathit{last}(\mathit{iter}(\langle v_0\rangle, x, \vec{g}_p(x), i))\big) \tag{3.27}$$

Let us now consider a standalone version for the **`iterate`** construct that we will put to use in upcoming example 3.4.1. Suppose we just want to iterate some PCR $\mathcal{B}$ starting from initial value $v_0 \in Z$ so that the intended result should be the final value computed by $\mathcal{B}$. This means $\mathcal{B}$ behaves as a function of type $Z \to Z$. Computation can be started from a PCR $\mathcal{A}$ defined on a *singleton* iteration space $I = \{0\}$ without producer and consumers other than a iterative consumer over $\mathcal{B}$, but recall we still need a final reducer (remark 3.1). The right projection operation (i.e. $a \otimes b = b$) will do fine here in the reducer, and the choice for initial reduction value $r_0 \in Z$ is irrelevant because the output of $\mathcal{A}$ is given by

$$r \;=\; r_0 \otimes \vec{f}^{\,0}_{c_e}(x) \;=\; \vec{f}^{\,0}_{c_e}(x) \;=\; \mathit{last}(\mathit{iter}(\langle v_0 \rangle))$$

so we give up the usual algebraic properties expected for the combiner as they are not important for $\mathcal{A}$.[23] This approach is illustrated in listing 3.14 where the left side can be seen as syntactic sugar for the right side. Of course, if v_0 is known to be a fixed constant then there is no need for input parameter x in $\mathcal{A}$.

```
fun v₀(x) = ...
fun cnd(s,j) = ...

PCR 𝒜(x)
  c = iterate cnd ℬ (v₀ x)

PCR ℬ(x₂)
  ...
```

```
fun v₀(x) = ...
fun cnd(s,j) = ...
fun ⊗(r,c) = c

ubnd 𝒜 = λx.0

PCR 𝒜(x)
  par
    c = iterate cnd ℬ (v₀ x)
    r = reduce ⊗ (r₀ x) c

PCR ℬ(x₂)
  ...
```

Listing 3.14: The *standalone* iterative PCR (left) is implemented as the *complete* PCR on the right.

Example 3.4.1. We saw previously in example 3.3.2 a *divide and conquer* PCR solution for the N-Queens problem. Now, we present in listing 3.15 and figure 3.17 an alternative solution employing the **`iterate`** construct. Nevertheless the overall idea is pretty much the same. [24]

Our main PCR NQueensIT is a standalone iterative PCR that is assumed to receive as input a zero configuration $[0_1, \ldots, 0_N]$ and iterates the subsidiary PCR NQueensITstep starting from singleton set $\{\,[0_1, \ldots, 0_N]\,\}$. At any iteration, NQueensITstep works as follows:

[23]In fact, right projection is associative over any domain.

[24]As a minor difference, they differ for $N = 0$. The DC solution returns the empty set, whereas the iterative solution returns the singleton set with the zero configuration.

1. The *elem* producer enumerates an input set CS of configurations, distributing each configuration to a corresponding consumer.

2. Instances of the *extend* consumer work, in parallel, on each candidate configuration p^i. If p^i is complete (all the rows have a queen) then the result is just $\{p^i\}$, otherwise it generates a set containing further possible candidate configurations (for which it relies on essentially the same *div* function from the *divide and conquer* version, except that here we use a set instead of a list).

3. The reducer joins the sets of candidates generated at the consumers.

Iteration of NQueensITstep finishes when the fix point condition $s^j = s^{j-1}$ (for $j > 1$) is reached, which means there are no more solutions.

```
fun cnd(s,j) = j > 1 and s == s[-1]

PCR NQueensIT(C)
  c1 = iterate cnd NQueensITstep {C}

fun validPos(C,i,j) = ...
fun addQInRow(C,i) = ...
fun canAddQInRow(C,i) = ...
fun complete(C) = all (λj. j != 0) C

fun elem(CS,i) = enum(CS)[i]
fun div(p) = {addQInRow(p,i) | 1 <= i <= len(p), canAddQInRow(p,i)}
fun extend(p) = if complete(p) then {p} else div(p)

lbnd NQueensITstep = λCS. 1
ubnd NQueensITstep = λCS. #(CS)

PCR NQueensITstep(CS)
  par
    p  = produce elem CS
    c2 = consume extend p
    r  = reduce ∪ {} c2
```

Listing 3.15: PCR NQueensIT code. # is the set cardinality operator, and *enum* is a (deterministic) enumeration of a set into a list.

□

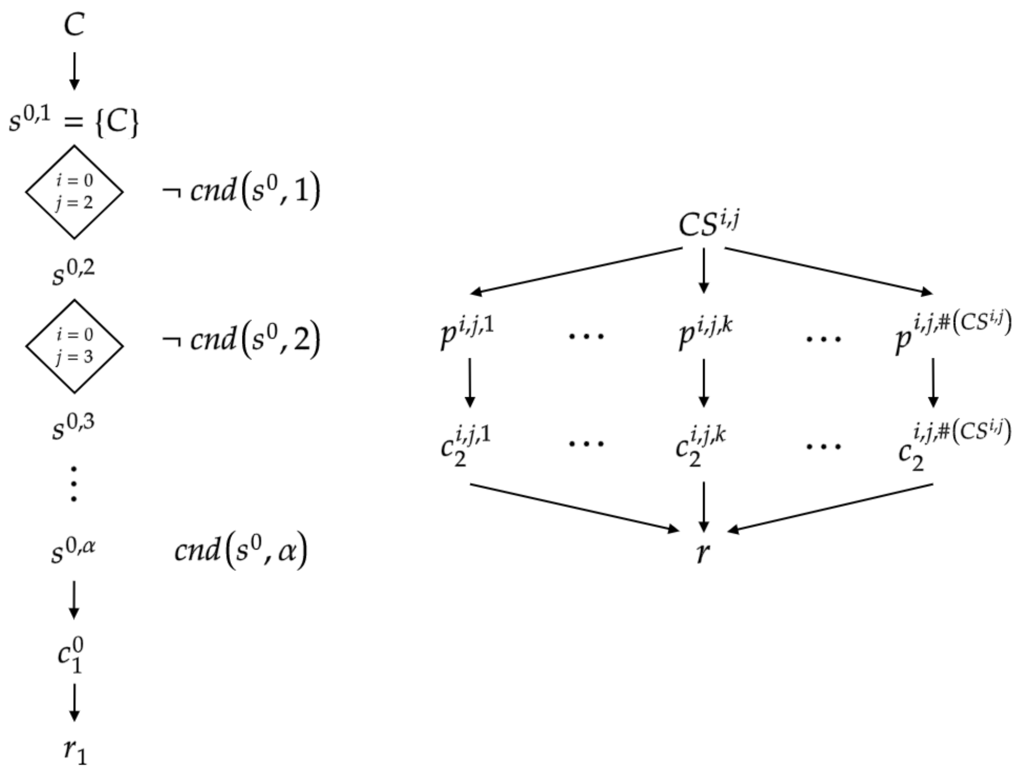


Figure 3.17: Generic dependence graph for PCR NQueensIT: (Left) outer PCR, (Right) inner PCR NQueensITStep. Each $\diamond$ in the outer PCR expand to the right DAG instantiated with an appropriate index i, j where i is the current assignment at outer scope and j is the current iteration at i. As the iteration space of NQueensIT is the singleton set $\{0\}$, we have $i = 0$ always.

Chapter 4

The TLA$^+$ specification language

Thinking doesn't guarantee that we won't make mistakes. But not thinking guarantees that we will.

Leslie Lamport

In this chapter we provide a historical and theoretical background for Lamport's TLA$^+$ specification language. This is the formal tool that we will use to specify and reason about the PCR pattern in the next chapter. Our presentation is mostly based on the textbook *Specifying Systems* [30] and also [31, 32, 33].

TLA$^+$ is a formal specification language that can be analyzed into two fragments: (1) The Temporal Logic of Actions (TLA) [34], a variant of Pnueli's original temporal logic [35] that makes it practical to write a specification as a single formula, and (2) a first order theory based on Zermelo–Fraenkel set theory with the axiom of Choice (ZFC) which Lamport likes to call ZFM [36], and which is the "+" in TLA$^+$. So, to put it simply: TLA$^+$ = TLA + ZFM. In the framework of TLA$^+$, computation is understood as the discrete evolution of *state* described by a temporal logic formula, where the state is formed by logical structures described with ZFM. Consequently, in our discussion we will distinguish between two aspects of TLA$^+$ that we call the *dynamics* and the *statics*.

TLA$^+$ possesses various relatively mature tools. In particular, it supports automated verification (on finite models) by model checking with the TLC tool [37] and semi-automated verification by theorem proving using the TLAPS tool [38].[1] Although most of this tools are standalone and can be used more or less comfortably from the command line, there

[1]A more recent model checking alternative is the APALACHE model checker which is based on symbolic techniques instead of the classic explicit enumeration of states. [39]

is also an IDE [40] available that is very well integrated with the tools and it is generally considered as the preferred way to work with TLA$^+$. Good tool support is a very desirable feature for a formal language (specially for industrial use) so we will also dig a bit on the workings of the aforementioned tools.

4.1 Specification and verification of computer systems: from Turing to Lamport

One of the biggest challenges in computer science and software engineering is to devise appropriate techniques to reduce the number of errors[2] in computer and/or software based systems. This is becoming increasingly important as these systems become omnipresent in our lives. The earliest reference we have in the literature for this kind of activity goes back (once again!) to the work of Charles Babbage in the 1800s who wrote about the "Verification of the Formulae Placed on the [Operation] Cards" of his analytical engine [41]. Now, the analytical engine is not considered a "computer" in the modern sense of the word, so let us move forward. It is commonly agreed that modern computation starts with the work of Alan Turing on the Halting Problem circa 1936, so we will start chronologically from there till Lamport's work on TLA$^+$, which will be subsequently developed in the rest of the chapter. As a disclaimer, we do not pretend to give a complete review of the field here. Rather, our focus is on what are generally known as *state systems*, concurrency phenomena and their related methods. Even so it is impossible to do justice to every contribution, and it should therefore be expected that a lot of important work will result largely omitted. This section in particular is inspired by [41] which is instead focused on sequential and imperative programs.

[2]More colloquially known as "bugs" for some funny historical reason. In 1947, an error in the Mark II (an electromechanical computer) was traced to a moth trapped in a relay, and that is considered the first recorded instance of an actual bug.

4.1.1 Turing: the birth of computing

In 1928, mathematicians David Hilbert and Wilhelm Ackermann posed the Entscheidungsproblem (German for "decision problem") as a challenge [42]. The problem asks for a general *decision procedure* (nowadays we say an "algorithm" or a "program") that determines whether a formula in a first-order logic is *universally valid.* To settle the problem, first a sensible notion of what an *algorithm* is had to be formally defined. In 1936, the seminal work of Alan Turing [43] introduced the idea of a Turing machine, a simple hypothetical machine capable of executing arbitrarily complex procedures and whose expressive power is still today unsurpassed. With this notion of algorithm, he then addressed the *halting problem*: the existence of a general procedure to decide whether an arbitrary algorithm runs forever or eventually halts, on given input. Turing proved such a procedure couldn't exists, so the halting problem turned out to be undecidable and this also proved, via a reduction argument, the Entscheidungsproblem is undecidable, thus giving a negative answer to Hilbert. [3] [4]

There is another implication here that concerns us more directly: a general verification method for algorithms cannot exist because for any such method there would be certain propositions about the algorithm that the method cannot prove or refute. Impossibility results are a very important part of our fundamental knowledge: they do have a negative significance but they are not a dead end —instead they serve as the starting point to further explore what is realistically achievable in practice. This means that different verification methods have to make trade-offs on their goals and for this reason they are normally targeted for particular kind of systems or properties. We accept as a fundamental consequence of the undecidability that the *more general* these methods try to be, the *less*

[3]Working independently and roughly at the same time, Alonzo Church arrived at the same result based on his λ-calculus which turned out to be equivalent to Turing machines as a model of computation [44].

[4]As late as 1930, Hilbert believed that there would be no such thing as an unsolvable problem. Together with Godel's incompleteness theorems in 1931, the undecidability was another big hit to the so called Hilbert's *formalist programme*: an attempt to provide a secure foundations for mathematics in response to the foundational crisis of mathematics in the early 20th century. In fact, the incompleteness and undecidability results are closely related, for example, the first incompleteness theorem (either Godel's or Rosser's stronger version) can be proved from the halting problem without messing with Godel numbers [45].

automatic they will be, thus requiring more effort and expertise. Intuitively speaking, all verification methods lie on a spectrum of the effort they require and the confidence they provide. Typically, a fairly vague distinction is made between *lightweight methods* and *heavyweight methods*, in the sense that *lightweight* tends to the the low end of the spectrum and *heavyweight* tends to the upper end.

Turing himself published work on algorithmic correctness in 1949, suggesting an *assertive method* where the correctness of the whole follows from the correctness of assertions on the parts, and he represented graphically this idea using boxed flow diagrams.[5] A similar idea was elaborated almost two decades later by Floyd who was unaware of Turing's work.

4.1.2 Floyd and Hoare: Axiomatic semantics

Floyd introduces in his 1967 landmark paper [3] the first systematic methodology for imperative program verification based on inductive assertions, although his main motivation was the idea that the semantics of a programming language may be defined independently of processors of the language, much in the style of McCarthy's earlier work on recursive functions.

Floyd's method is based on annotating a flow chart (a directed graph of commands that constitutes the algorithm) with assertions (propositions expressed in a logical language) which relate values of variables in the algorithm. As a simple example, figure 4.1 shows the flowchart for an algorithm that computes the sum of the series a_1, a_2, . . . , a_n and stores it in variable S. The assertion at the entry point (see node START) is called the *precondition*, and the assertion at the exit point is called the *postcondition* (see node HALT). The basic notion of correctness is that whenever the initial values of the program variables satisfy the precondition, the final values on completion will satisfy the postcondition.[6]

In general, verification in this method works roughly as follows. Annotate each command

[5]In fact, this is preceded by work of Goldstine and von Neumann in 1947 [46].
[6]Floyd did not used this *pre/post* terminology, it was introduced later by Hoare.

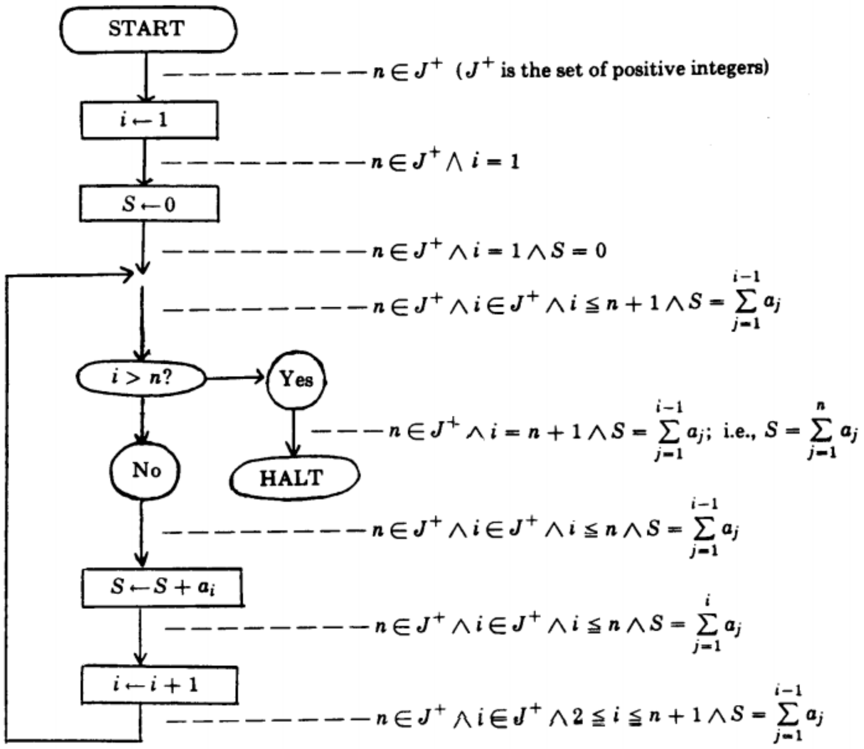

START
$n \in J^+$ (J^+ is the set of positive integers)
$i \leftarrow 1$
$n \in J^+ \wedge i = 1$
$S \leftarrow 0$
$n \in J^+ \wedge i = 1 \wedge S = 0$
$n \in J^+ \wedge i \in J^+ \wedge i \leqq n + 1 \wedge S = \sum_{j=1}^{i-1} a_j$
$i > n?$
Yes
$n \in J^+ \wedge i = n + 1 \wedge S = \sum_{j=1}^{i-1} a_j$; i.e., $S = \sum_{j=1}^{n} a_j$
HALT
No
$n \in J^+ \wedge i \in J^+ \wedge i \leqq n \wedge S = \sum_{j=1}^{i-1} a_j$
$S \leftarrow S + a_i$
$n \in J^+ \wedge i \in J^+ \wedge i \leqq n \wedge S = \sum_{j=1}^{i} a_j$
$i \leftarrow i + 1$
$n \in J^+ \wedge i \in J^+ \wedge 2 \leqq i \leqq n + 1 \wedge S = \sum_{j=1}^{i-1} a_j$

Figure 4.1: Flowchart of an algorithm that computes $S = \sum_{j=1}^{n} a_j$, taken directly from Floyd's original paper [3]. We write $\mathbb{N}^+$ instead of J^+ for the set of positive integers.

c_i with a proposition P_i such that any pair of successive annotations are *inductive*: the execution of c_i in a state satisfying P_i guarantees at completion P_{i+1}. Then, by transitivity, correctness follows. In particular, for the algorithm in figure 4.1 the main goal is to prove that on input $n \in \mathbb{N}^+$ the program guarantees at completion $S = \sum_{j=1}^{n} a_j$. Floyd recognized his inductive method as the basis for proofs of relations between input and output, but there is nothing guaranteeing the exit will ever be reached —instead *termination* is simply assumed. For this reason, this form of correctness would later be more precisely known as *partial* correctness. Nevertheless, in the same paper he also considered termination proofs by the method of well founded sets. Partial correctness plus termination is known as *total* correctness.

Subsequent work by Hoare in 1969, heavily influenced by Floyd's, introduced an axiomatic approach for inferring and proving properties of sequential algorithms based on *triples* instead of flowcharts. Hoare *triples* are annotated segments of programs (more formally, theorems) of the form $\{P\}Q\{R\}$ where Q is a piece of program and P and R are logical propositions respectively known as the *pre-condition* and *post-condition* [7]. For example, the triple

$$\{x = r + y \times (1 + q)\}\ q := 1 + q\ \{x = r + y \times q\}$$

can be read as stating that if the assignment statement $q := 1 + q$ is executed in any state where the values are such that $x = r + y \times (1 + q)$ then, assuming execution terminates, the state after the execution of the assignment will be such that its values make the assertion $x = r + y \times q$ true. The sum example of figure 4.1 can be expressed as the the following triple:

```
{ $n \in \mathbb{N}^+$ }
$i$  := 1
$S$  := 0
while $i \leq n$ do
   $S$  := $S + a_i$
   $i$  := $i + 1$
{ $S = \sum_{j=1}^{n} a_j$ }
```

Hoare provided several axioms and rules to reason about programs. For example, the following iteration rule (D3) is used to prove the correctness of a loop segment:

$$\text{D3}\ \frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \{\neg B \wedge P\}}$$

In D3, the formula P is called the *loop invariant*: a property of the loop that is true before (and after) each iteration. Intuitively speaking, it captures the "essence" of the loop, and it accomplishes the same role as the induction hypothesis in inductive proofs. Note that termination is not proved by this rule, i.e. this is a partial correctness rule.

Hoare triples are considered an improvement in mathematical notation for proofs of program properties. Triples are also more compositional than flowcharts. Mostly, the work of Hoare is a clearer exposition of Floyd's ideas. Indeed, Hoare's logic is usually called by the name Floyd-Hoare logic. Now, we point to what we consider the two main drawbacks of this early developments:

1. Doing proofs in this framework is a task that requires human ingenuity, usually

[7] Original notation was $P\{Q\}R$, but $\{P\}Q\{R\}$ gained more use with time.

resulting in long very detailed proofs. For this reason, the scalability of the approach for industrial software was questioned. Subsequent developments aimed to mitigate this difficulty. Maybe as the most prominent example Dijkstra proposed the idea of *program derivation* which means to "develop proof and program hand in hand" [47]. For this, he introduced a reformulation of the Floyd-Hoare logic that allowed a great deal of automation in proof development. As it turns out, the loop invariant is (in general) the only part that needs to be provided by the user and that cannot be automatically deduced.

2. The classical point of view is that an algorithm computes a (partial) function. Now, it was observed that a lot of software does not seem to fit into such paradigm in a natural way, for example an OS does not need to terminate (rather, termination would be abnormal). The concurrency phenomena, which in general encompasses parallel computing and distributed and/or reactive systems, was lacking semantics and appropriate methods to reason about errors like deadlocks, livelocks, etc. that commonly plague concurrent systems. To cope with this, some of the subsequent developments tried to adapt the Floyd-Hoare logic (e.g. the Owiki-Gries method [48]) while still working within the realm of classical logic. But most notoriously others introduced new formalisms like *process algebras* (e.g. CSP [49]) and *temporal logics* (our next section).

4.1.3 Pnueli: Computer Science meets Temporal Logic

Classical logic is appropriate to reason about mathematical truth. When a theorem is considered proved its truth is established, and we do not expect that to change later unless an error in the proof is found. For example, the truth of the statement $2 + 2 = 4$ does not change with time. Consider the following statements about rain events: $A =$ "It's raining" and $B =$ "It's going to rain tomorrow". They are indistinguishable as propositions for classical logic, however B explicitly refers to the *future*. Philosopher Arthur Prior developed *Tense Logic* (nowadays known as *temporal logic*) as a linguistic tool to study the use of modality and intentionality in natural languages [50]. Temporal logic is actually an umbrella term, in particular it can be regarded as a kind of modal logic intended to reason about expressions with *tense*, that is, expressions with qualifications of time (e.g. statement B). Prior was interested in philosophical problems concerning *determinism* and *free-will*, which can be traced back to the problem of Future Contingents that is exemplified by Aristotle in his treatise *De interpretatione* as follows: let $P =$ "There will be a sea battle tomorrow". By the *principle of bivalence* then either P is true or P is false. If P is predicted to be true (the other case is analogous) then P was true today and also at any time in the past, which means it is true by *necessity* leaving no room for chance (i.e. future is already determined, and this is unlikely to be compatible with the usual notion of *free-will*). Some authors adhere to the view that this kind of statements cannot be assigned a well-defined truth value, thus making the classical *principle of bivalence* fail.

In researching the problem of systems verification, computer scientist Amir Pnueli came across the work of Prior and he realized that this kind of non-classical logic was a perfect fit for computer science where the objects under study are systems that evolve in (discrete) time and the truth of whose properties may vary with time. His findings first appeared in a landmark 1977 paper: *The temporal logic of programs* [35]. Pnueli summarized his work as an unifying framework that allows verification of both sequential and concurrent programs. In this framework, any kind of program is represented as a dynamic discrete system defined by a structure (S, R, s_0) where S is the (possibly infinite) set of system states, R is a transition relation between the states and its possible successors, and s_0 is the initial state. Then, an execution σ of the system (a.k.a. *trace* or *run*) is a (possibly infinite) sequence of states

$$\sigma \;=\; s_0\ , s_1,\ s_2,\ \ldots$$

such that $(s_i, s_{i+1}) \in R$ for any $i \geq 0$.[8] To represent a sequential program, the state is further structured as a pair (π, u) where π is the control component indicating current processor location (typically identified by labels l_0, l_1, etc.) for the program and u is the data component with processor's state (i.e. the memory). To represent a concurrent

[8]This approach bears a strong connection to the work of Keller on *labeled transition systems* (1976) and the *Kripke structures* used by Kripke (1960s) to give semantics to modal logic.

program the same idea is extended: state is $(\pi_1, ..., \pi_n; u)$ where each π_i is the control for processor i, and u is the state shared among all processors. So, the framework admits n programs to be concurrently run by n processors. At each step of the whole system, one processor is selected, say i, and the statement at the location pointed to by π_i is executed atomically.

Pnueli reduced the notion of *correctness* to two main notions: [9]

1. **Invariance**: a property holding continuously throughout the execution of a program. This covers the already mentioned concept of *partial correctness* of sequential programs but also concepts like *mutual exclusion* and *deadlock freedom* for concurrent programs.

2. **Eventuality**: a dependence in time in the behaviour of a program. This covers the already mentioned concept of *total correctness* of sequential programs but also more general concepts like *responsiveness* for reactive systems.

To put these notions more formally, first define the set X of reachable states of a system (S, R, s_0) as

$$X \ = \ \{s \ : \ R^*(s_0, s)\} \ \subseteq \ S$$

where R^* is the transitive closure of R. Let α be a predicate on states. Then α is an *invariant property* of the system if $\alpha(s)$ holds for all $s \in X$. In particular, partial correctness can be stated as the invariant property:

$$\pi = l_{end} \ \Rightarrow \ (P \ \Rightarrow \ R)$$

where l_{end} is the label for program exit point, P is the pre-condition and R is the post-condition. Eventualities are expressed in terms of the temporal operator $\leadsto$ (leads to)[10] that involves two time variables and is defined as follows:

$$\alpha \leadsto \beta \ = \ \forall\, i : \exists\, j \geq i : \ \alpha(s_i) \ \Rightarrow \ \beta(s_j)$$

In particular, total correctness (which only makes sense for programs that are supposed

[9]These notions were introduced a bit earlier by Lamport as *Safety* and *Liveness* respectively.

[10]Pnueli's original symbol to denote this operator is not easy to reproduce here, so we prefer to use symbol $\leadsto$ which is used in TLA$^+$ for the same purpose.

to halt) can be stated as the eventuality:

$$(\pi = l_0 \wedge P) \leadsto (\pi = l_{end} \wedge R)$$

where l_0 is the label for program entry point.

Pnueli shows how to prove *invariance* and *eventuality* properties applying already known proof methods from other authors. More interestingly, he presents a set of axioms and rules to develop deductive proofs. However, the deductive approach for temporal reasoning was better elaborated in his subsequent work [51] where he introduces a full fledged logic called Linear Temporal Logic (LTL) in which the formulas of the logic are interpreted over sequences of states. LTL possesses the following primitive temporal operators[11]

- $\Box\, \alpha$ (always): α is *always* true in the future.
- $\Diamond\, \alpha$ (eventually): α is *eventually* true in the future.
- $\bigcirc\, \alpha$ (next): α is true in the *next* instant.

Then, the binary eventuality operator $\leadsto$ commented earlier can be defined in terms of the primitive unary operators as

$$\alpha \leadsto \beta \;=\; \Box(\alpha \;\Rightarrow\; \Diamond\beta)$$

Some authors introduced other binary operators in the logic. For example, $\alpha\, \mathcal{U}\, \beta$ (until) means that α has to hold *at least until* β becomes true. It is known that $\mathcal{U}$ cannot be expressed in terms of the unary operators like we did for $\leadsto$.[12]

The point of view stressed by Pnueli was that Temporal Logic is an approach to both semantics and verification that provides a formalism for proving temporal properties of systems (of any kind) based on their temporal semantics. However, he admitted the same first drawback we pointed earlier for the Floyd-Hoare approach: proofs may require much effort. In this regard, he hoped that systematic experience could facilitate the task.

[11]Actually, this is not the original symbolism, but it is the most common nowadays. Most of this symbols where adopted first in modal logic.

[12]Hans Kamp was the first to introduce the binary operators $\mathcal{U}$ (until) and $\mathcal{S}$ (since), and proved a remarkable result concerning their expressive power: every temporal operator on a class of continuous, strict linear orderings that is definable in first-order logic is expressible in terms of $\mathcal{U}$ and $\mathcal{S}$. [52]

The introduction of LTL triggered a lot of research into possible variants of this logic, connections with *automata theory* and connections with predicate first-order (and second-order) theories. Indeed, today we have a big zoo of temporal logics (as well for other kind of logics). Instead of thinking there is a single one true logic to be discovered, a pluralist point of view is much accepted either by philosophical or pragmatic reasons.

After LTL, special mention is deserved for Computational Tree Logic (CTL). In 1981, Clarke and Emerson [53] (also independently [54]) introduced CTL as an alternative temporal logic where time forms a *branching structure* (trees) instead of a *linear structure* (sequences) like in LTL. This logic adds new temporal operators that allow to quantify over the possible future paths of execution, something not possible in LTL.[13] But more importantly, in the same work they proposed the verification method of *model-checking*: both the system and its properties are formulated in some logic, then to check that the system satisfies the desired properties means, in terms of formal logic, to check whether a *structure* (representing the system) satisfies a formula (representing the desired property). This general concept applies to many kinds of logic, and allows for an automated verification method of properties over (typically finite) systems. It does not need the same kind of effort and expertise required for deductive proofs —however matters of decidability and complexity can be a limiting factor for their effective applicability. Also, it should be noted that this method is usually applied over an abstraction of the system and not an implementation of the system itself (harder but possible) leaving a *formal gap* between the real system and what is actually verified.[14]

4.1.4 Lamport: the search for a practical formalism

Leslie Lamport is best known by his pioneering work in concurrent and distributed systems.[15] But perhaps less known is all his work on methods and formalisms for rigorously

[13]As it turns out, LTL and CTL are not comparable, i.e. neither is more expressive than the other. The combination of LTL and CTL capabilities is called CTL*.

[14]Interestingly, if we compare this situation with the classic engineering branches, the "gap" is also there. Any system of equations can only describe an abstraction of the physical object, not the object itself. Software can also be an abstract mathematical object in the mind of the designer, but its ultimate execution is a physical phenomena outside the realm of formal logic.

[15]And, of course, the LaTeX document preparation system

establishing the correctness of algorithms. His most important contribution in this regard is TLA and the later TLA$^+$.

Initially, Lamport found the temporal logic introduced by Pnueli appealing because it was appropriate to express both the notions of safety and liveness and, in principle, it allows a concurrent system to be described by a single formula.[16] But later he became disillusioned with its practicality when he saw his colleagues:

> "...spending days trying to specify a simple FIFO queue — arguing over whether the properties they listed were sufficient. I realized that, despite its aesthetic appeal, writing a specification as a conjunction of temporal properties just didn't work in practice."

Lamport became convinced that the only practical way to specify non-trivial systems is to describe them as a kind of abstract state machines. This led him to the invention of TLA in the late 80s, a mathematical foundation for describing concurrent systems that adopts some operators of ordinary temporal logic but relies mostly on the concept of *action*: a function over a pair of states (i.e. a transition). A TLA formula is used to describe the possible executions of the system as a state machine. Some of the most salient characteristics of TLA are the following:

1. TLA is designed to restrict the need for temporal reasoning to a bare minimum. Lamport argues that temporal formulas tend to be harder to understand than formulas of ordinary first-order logic, and temporal logic reasoning is more complicated than ordinary mathematical reasoning.[17] Consequently, in practice the 95% of a TLA specification usually consists of ordinary non-temporal mathematics.

2. Unlike other frameworks like the Floyd-Hoare logic where the system and its properties are written in a different language, TLA is an universal mathematical notation in which both the system and its claimed properties are formulas in the same logic. Consequently, system S *satisfies* property P if and only if $S \Rightarrow P$ is a valid formula

[16]In other specification languages, for example Z, there is no single formula or object that mathematically constitutes the specification.

[17]For example, the deduction theorem of classical logic fails in temporal logic. The proof rule $\alpha \vdash \Box\alpha$ is valid, but $\alpha \Rightarrow \Box\alpha$ is *not* a theorem. For this and other reasons, Lamport claims temporal logic is "evil", although he consider it a "necessary evil".

of the logic. Moreover, system S_2 is a *refinement* of system S_1 if and only if $S_2 \Rightarrow S_1$ is a valid formula of the logic.

3. The $\bigcirc$ (next) and $\mathcal{U}$ (until) temporal operators that are commonly seen in ordinary temporal logic are absent in TLA. The former is incompatible with *stuttering invariance* (a characteristic to be discussed in the next section) and the later can be, at least in principle, defined but it is discouraged as temporal reasoning is deliberately restricted.

However, TLA is not by itself a full specification language, since it does not e.g. fix the interpretation of elementary function and predicate symbols such as $+$ and $\in$. Lamport needed an underlying language to describe the data manipulated by TLA and he wanted it to be as simple as possible. For him, this meant that such language should be based on ordinary mathematics and be as far away as possible of a programming language. During a visit to Oxford in 1991, he explored the possibility of adding TLA to the Z specification language [55], the result would have been called TLZ but it never saw the light of day. He writes about it:

> "Tony Hoare was at Oxford, and concurrency at Oxford meant CSP. The Z community was interested only in combining Z with CSP — which is about as natural as combining predicate logic with C++."

Later, around 1993, he completed a full formalism called TLA$^+$ that is based on a variant of Zermelo-Fraenkel set theory with choice (ZFC) for describing the data. ZFC is widely accepted by mathematicians as the basis for formalizing mathematical theories. Other formal languages like Z, (Event-)B [56] and Mizar [57] are also based on some axiomatization of set theory[18]; however, these languages impose a typing discipline on set theory, whereas TLA$^+$ is untyped following common mathematical practice. As a consequence, type correctness needs to be asserted (and proved) as an invariant, i.e. like any other invariant property, and is not a syntactic requirement. This is possibly the most controversial design choice. Lamport argued that imposing a decidable type system on a specification language leads to unacceptable restrictions of the expressiveness of that

[18]Not necessarily the well known and more standard ZFC. For example, Mizar is based on Tarski–Grothendieck set theory, a non conservative extension of ZFC. In fact, Mizar had in 2007 the largest library of formalized mathematics. [58]

language. In a joint paper with Larry Paulson titled *Should your specification language be typed?* [59] they discuss the advantages/disadvantages of typed/typeless formalisms. They write:

> "Some computer scientists are so used to thinking in terms of types that they find untyped set theory completely unnatural... They feel that we should not be allowed to write a nonsensical formula like $2 \cap \mathbb{N}$... There are mathematicians and computer scientists who find untyped set theory to be completely natural. To them, not being allowed to write $2 \cap \mathbb{N}$ is a confusion of syntax with semantics—like trying to redefine the grammar of English so that 'Rocks are carnivores' is not a well-formed sentence."

It is out of scope to give our own opinion on this subject, but one thing is certain, there is not other formal language in existence that resembles better the feeling of the ordinary pen-and-paper (set theoretic based) mathematics that most people is used to. And this is in part for being untyped.

TLA$^+$ was not originally designed with any form of mechanical verification in mind. This is because "the fundamental goal of TLA$^+$ is not to provide tools for finding bugs, it's to teach people a better way to think about systems", but also because TLA$^+$ is a very expressive language. However, to Lamport's surprise, a model checker, called TLC, was developed in 1999 [37]. Later, a theorem prover, called TLAPS, started development in 2008 [38].[19] We think that cases of industrial success of TLA$^+$ like the one reported by Amazon engineers in [60] would not have been possible without mechanical tool support, even if they themselves acknowledge that specification by itself is helpful.

4.2 TLA$^+$ dynamics: Temporal Logic of Actions

In this section we make emphasis in the dynamic part of TLA$^+$, that is, the temporal logic that is used to describe and reason about the evolution of the state. We believe it is instructive to describe first what is actually called "raw" TLA (rTLA), a simplified version

[19]It should be noted that these tools are not without limitation. None of them supports TLA$^+$ fully, however it can be argued that for most practical purposes it is not really necessary.

of TLA, and then motivate by example the introduction of TLA as a syntactic restriction of rTLA that is needed to enforce the important notion of *stuttering invariance*. On the process, we will discuss the concepts of refinement, hiding and composition, among other things.

4.2.1 Basic concepts

Computer systems differ from the systems traditionally studied by scientists because we can pretend that its state changes in discrete steps. In this sense, computation is a discrete dynamical process, but this is an abstraction. As a model for such abstractions, Lamport defines an *abstract system* to be "a collection of behaviors, each representing a possible execution of the system, where a behavior is a *sequence of states* and a state is an assignment of values to variables". Events are transitions between consecutive states in a behavior (a.k.a. *steps*), and behaviours are assumed to be *infinite* no matter what kind of system we are considering. This view is called the *standard model*, and it is reminiscent of Pnueli's framework.

Transitions are described by *action* formulas in TLA, but "action" is not a primitive notion, it is rather just a name for formulas referring to the current and next state. In general, TLA formulas are interpreted over behaviours, so the semantics of a formula is the collection of behaviours satisfying it. Systems, algorithms, programs, their environment and their properties, are all mathematically represented as collections of behaviors described by appropriate formulas in the same logic. TLA does not presuppose any underlying execution/concurrent/communication model such as shared-variable or message-passing, synchronous or asynchronous, interleaving or non interleaving, etc. It is the designer's responsibility to represent the system at the appropriate level of abstraction. This may seem too liberal to be of practical use, but there are some standardized ways to define TLA specifications that facilitate its use.

Properties of systems are classified into two classes that were first introduced by Lamport in 1977 [61]. They are:

- **Safety**: a property specifying what must *never* happen, or more colloquially, "something bad will never happen". For example, the statement "x is always less than 10", in symbols $\Box(x < 10)$, asserts a safety property that is violated if $x \geq 10$ holds in some state of a behaviour. Note that the violation can be observed in a finite *prefix* of the behaviour, that is, the evidence of the violation can be found after a finite amount of states.

- **Liveness**: a property specifying what must *eventually* happen, or more colloquially, "something good will eventually happen". For example, the statement "x will eventually be at least equal to 10", in symbols $\Diamond(x \geq 10)$, asserts a liveness property that is violated if $x \geq 10$ never occurs in the future. Note that, unlike the safety case, the violation can't be observed on a finite *prefix* of the behaviour, because at any time the "good" event might still occur later (i.e. no finite counterexample).

The concepts of safety and liveness where formalized by Alpern and Schneider in 1987 [62]. They characterize the universe of behaviors as a topological space where safety properties are closed sets and liveness properties are dense sets. As every set in a topological space is equal to the intersection of a closed set and a dense set, it follows that any property (or any system) is an intersection of a safety property and a liveness property. This theoretical result is embodied in the *canonical form* for state machine specifications we will see later.

Currently, there is ongoing research on properties of systems that are not just collections of behaviours as the classic safety and liveness. The so called *hyperproperties* generalize ordinary properties by expressing relations among multiple executions of a system, and allow to express security policies, such as secure information flow [63].

4.2.2 Syntax and semantics of formulas

TLA formulas are built from two kind of variables:

- ***Rigid* variables**, i.e. those whose values are fixed throughout a behaviour. However, they may not be the same across different behaviours. We will refer to these more simply as *constants*. They are declared with the CONSTANT/S keyword.

- ***Flexible* variables**, i.e. those whose values are not fixed throughout a behaviour. They constitute the *state*. We will refer to these more simply as *variables*. They are declared with the VARIABLE/S keyword.

The syntax of TLA formulas is defined as a linear hierarchy of three levels:[20]

1. **State formulas**: standard terms and formulas of first-order logic. For example: $x^2 + y - 3$ is a state function and $\exists\, x : x \in u \land x \in v$ is a state predicate. Semantically, they are interpreted over individual states.

2. **Transition formulas (actions)**: first-order terms and formulas that may contain both normal (unprimed) variables (e.g. v) and primed variables (e.g. v'). For example: $v' = v + 1$ and $u' \in v$ are actions.[21] Semantically, they are interpreted over pairs (s_i, s_{i+1}) of consecutive states, with unprimed variables being interpreted in the current state s_i and primed variables in the next state s_{i+1}.

3. **Temporal formulas**: they are built from state and transition formulas by applying operators of temporal logic ($\Box$, $\Diamond$, $\leadsto$). Semantically, they are interpreted over behaviours.

Given any state formula e, the transition formula e' is obtained by replacing all (free) occurrences of variables by their primed counterparts. For example $(\exists\, x : x \in u \land x \in v)'$ is $\exists\, x : x \in u' \land x \in v'$ assuming u and v are variables. Semantically, e' denotes the value of e at the second state of the pair of states at which e is evaluated. The action UNCHANGED e is shorthand for $e' = e$, which means the value of the variables in e don't change. For example, to assert variables $v_1, \ldots, v_n$ are unchanged we can write UNCHANGED $\langle v_1, \ldots, v_n \rangle$ as a shorthand for $\langle v_1', \ldots, v_n' \rangle = \langle v_1, \ldots, v_n \rangle$ which is useful if we have many variables. Note that tuple notation in TLA uses angular brackets.

Formulas at all three levels are closed under the propositional operators ($\neg$, $\land$, $\lor$, $\Rightarrow$, $\equiv$) and first order quantifiers ($\forall$, $\exists$) of classical logic with their usual semantics. For example,

[20]Four levels if we also distinguish the *state formulas* into those which only uses constants (level 0) and those which also use variables (level 1). Some presentations adopt that view.

[21]The LTL equivalent to TLA action $u' \in v$ would be $\exists\, x : x \in v \land \bigcirc(u = x)$.

if α is a formula and v is a variable then $\exists\, t : \alpha \Rightarrow \Diamond\Box(v = t)$ is a legal formula. However, different levels can't be arbitrarily mixed, for example $A \Rightarrow F$, where A is an action and F is a temporal formula, is not legal.

The operators $\Box$ and $\Diamond$ are the familiar *always* and *eventually* operators of LTL. Let α be a formula, then

- $\Box\,\alpha$ is true in a behaviour σ iff α is true of every prefix of σ, or more formally,

$$\sigma \models \Box\,\alpha \;\text{ iff }\; \sigma^n \models \alpha \;\text{ for all } n \in \mathbb{N}$$

- $\Diamond\alpha$ is true in a behaviour σ iff α holds for some prefix of σ, or more formally

$$\sigma \models \Diamond\,\alpha \;\text{ iff }\; \sigma^n \models \alpha \;\text{ for some } n \in \mathbb{N}$$

The following are some well known temporal identities:

Duality:	$\Box\,\alpha \equiv \neg\Diamond\neg\alpha$	$\Diamond\,\alpha \equiv \neg\Box\neg\alpha$
Distributivity 1:	$\Box(\alpha \wedge \beta) \equiv \Box\alpha \wedge \Box\beta$	$\Diamond(\alpha \vee \beta) \equiv \Diamond\alpha \vee \Diamond\beta$
Distributivity 2:	$\Diamond\Box(\alpha \wedge \beta) \equiv \Diamond\Box\alpha \wedge \Diamond\Box\beta$	$\Box\Diamond(\alpha \vee \beta) \equiv \Box\Diamond\alpha \vee \Box\Diamond\beta$

The binary operator $\alpha \leadsto \beta$ (α *leads to* β) is a shorthand for $\Box(\alpha \Rightarrow \Diamond\,\beta)$. Other useful syntax constructs will be introduced on the go.

4.2.3 Systems as state machines

First, we suggest how a system, say S, might be defined as a state machine. Let *Init* be a *state formula* that describes the possibly initial states of S, and *Next* an *action* that describes how state might change at any step of S. Then, the (specification of) system S is defined as follows

$$\mathit{Spec} \;\triangleq\; \mathit{Init} \wedge \Box \mathit{Next} \tag{4.1}$$

where TLA symbol $\triangleq$ denotes *is defined as*. Definitions are the main building block in TLA (and TLA$^+$). Next, we introduce two concrete specifications based on this model that will be used as running examples. We will quickly see that there is a problem.

Example 4.2.1 (Hour clock)**.** Let's consider an hour clock system (HC). If we let variable h represent the hour, the possible values should be integers between 1 and 12. The clock is not intended to stop (i.e. terminate).

A possible definition for HC following the form of 4.1 is

$$
\begin{array}{lcl}
\textsc{variable}\ h & & \\
HCini & \triangleq & h \in 1\,..\,12 \\
HCnxt & \triangleq & h' = \textsc{if}\ \ h \neq 12\ \textsc{then}\ \ h+1\ \ \textsc{else}\ \ 1 \\
HC & \triangleq & HCini \land \Box HCnxt
\end{array}
$$

The TLA expression $\textsc{if}\ p\ \textsc{then}\ e_1\ \textsc{else}\ e_2$ is just a shorthand for $(p \Rightarrow e_1) \land (\neg p \Rightarrow e_1)$. The initial formula $HCini$ allows starting from any hour value, which is an example of input non-determinism.[22] The action $HCnxt$ controls the evolution of h at each step, h is updated incrementally unless $h = 12$ in which case it goes back again to $h = 1$.[23]

From the semantics point of view, $[\![HC]\!]$ denotes the collection of all behaviours σ such that $\sigma \vDash HC$. They may be represented as:

$$
\begin{array}{c}
h = 1\ \rightarrow\ h = 2\ \rightarrow\ h = 3\ \rightarrow\ \cdots\ \rightarrow\ h = 1\ \rightarrow\ \cdots \\
h = 2\ \rightarrow\ h = 3\ \rightarrow\ h = 4\ \rightarrow\ \cdots\ \rightarrow\ h = 2\ \rightarrow\ \cdots \\
\vdots \\
h = 12\ \rightarrow\ h = 1\ \rightarrow\ h = 2\ \rightarrow\ \cdots\ \rightarrow\ h = 12\ \rightarrow\ \cdots
\end{array}
$$

Or to be more succinct, let h_0 be the initial value of h:

$$h = h_0\ \rightarrow\ h = (h_0 + 1)\%12\ \rightarrow\ h = (h_0 + 2)\%12\ \rightarrow\ \cdots\ \rightarrow\ h = h_0\ \rightarrow\ \cdots$$

If the clock is behaving properly, then its display should be an integer from 1 through 12. So, h should be an integer from 1 through 12 in every state of any behavior satisfying the clock's specification. The following implication asserts type correctness of HC:

$$HC\ \Rightarrow\ \Box(h \in 1..12)$$

[22]For a model checker based on explicit enumeration (like TLC), this means to enumerate the state space starting from twelve different roots.

[23]Of course, another way to define action $HCnxt$ is to take advantage of modular arithmetic: $HCnxt \triangleq h' = (h + 1)\,\%\,12$

$\square$

Example 4.2.2 (Hour-minute clock)**.** Based on our previous example, let us now consider an hour-minute clock system (*HMC*). We use a variable h just like in *HC*, but also a variable m to represent minutes whose possible values should be integers between 0 and 60. A possible definition for *HMC* following the form of 4.1 is

$$\begin{array}{lcl}
\multicolumn{3}{l}{\textsc{variables}\ h,\ m} \\
HMCini & \triangleq & HCini \land m \in 0\,..\,59 \\
Min & \triangleq & m' = \textsc{if}\ m \neq 59\ \textsc{then}\ m+1\ \textsc{else}\ 0 \\
Hour & \triangleq & (m = 59 \land HCnxt) \lor (m < 59 \land h' = h) \\
HMCnxt & \triangleq & Min \land Hour \\
HMC & \triangleq & HMCini \land \square HMCnxt
\end{array}$$

The initial formula *HMCini* conjoints the initial condition of system clock *HC* with the possibility of any initial minute value. The action *HMCnxt* controls the evolution of h and m at each step with two sub-actions *Hour* and *Min* respectively. m is updated similarly to h in system clock *HC*, and h is updated if $m = 59$, otherwise it remains unchanged.

The following implication assert type correctness of *HMC*:

$$HMC \Rightarrow \square(h \in 1..12 \land m \in 0..59)$$

$\square$

Intuitively speaking, an hour-minute clock is a special case of an hour clock, because if we hide the minute in the hour-minute clock then it behaves just like the more simple hour clock. In TLA terms, *HMC* should also be a specification for *HC* in the sense that any behaviour of *HMC* is also a behaviour of *HC*. This is the notion of *refinement* between specifications, i.e. we can say *HMC* is a *refinement* (or implementation) of *HC*, and conversely, that *HC* is an *abstraction* of *HMC*. Refinement allows to carry out system development as a process running through several levels of detail forming a chain where each new refinement preserves the properties of previous specifications. In principle, one could start with a very abstract specification of the system and then add enough detail to obtain an executable specification (i.e. a program), although it is very hard to reach that point in practice.

Semantically, that *HMC* is a refinement of *HC* means $[\![HMC]\!] \subseteq [\![HC]\!]$, or alternatively

$$\text{if } \sigma \vDash HMC \text{ then } \sigma \vDash HC \quad \text{for all } \sigma$$

which is the same as the validity of an implication formula in the logic

$$\vDash \; HMC \;\Rightarrow\; HC \tag{4.2}$$

But there is a problem. Our intuition does not match the formalism here, because it is easy to see that 4.2 does not hold: in any behaviour of *HC*, the value of h changes at every step, but in the behaviours of *HMC* the value of h changes only every 59 steps.

TLA embraces the notion of *stuttering invariance* which fixes this problem and makes refinement (and composition) possible.

4.2.3.1 Enforcing stuttering invariance

First, we introduce some technicalities. For any behaviour σ, its *stutter free* form, denoted $\natural\sigma$, is obtained by replacing any finite sub-sequence of consecutively repeating states with just a single instance of the state (e.g. $\natural(a, b, b, b, c, d, d, \ldots, d, \ldots) = a, b, c, d, d, \ldots, d, \ldots$). For any pair of behaviours σ and τ, they are *stuttering equivalent*, denoted $\sigma \simeq \tau$, iff $\natural\sigma = \natural\tau$. If Σ is a collection of behaviours, its closure is defined by $\Sigma_{\simeq} = \{\sigma' \,:\, \sigma \simeq \sigma'\}$. So, if $\Sigma = \Sigma_{\simeq}$, we say Σ is *closed under stuttering*.

Now, we revise the syntax of (raw) TLA presented earlier. A formula α is *stuttering invariant* if its semantics is closed under stuttering, that is $[\![\alpha]\!] = [\![\alpha]\!]_{\simeq}$. Let us consider a formula that describes a monotonically increasing value:

$$F \;\triangleq\; x = 1 \wedge \Box(x' = x + 1) \tag{4.3}$$

The formula F is not stuttering invariant because it *distinguishes* (for example) the following behaviours

$$\sigma_1 : \; x = 1 \rightarrow x = 2 \rightarrow h = 3 \rightarrow \cdots$$
$$\sigma_2 : \; x = 1 \rightarrow x = 1 \rightarrow x = 2 \rightarrow h = 3 \rightarrow \cdots$$

as $\sigma_1 \in [\![F]\!]$ but $\sigma_2 \notin [\![F]\!]$, however $\sigma_1 \simeq \sigma_2$. In order to not distinguish them, the formula should instead be

$$G \triangleq x = 1 \land \Box(x' = x + 1 \lor x' = x) \tag{4.4}$$

as $\sigma_1, \sigma_2 \in [\![F]\!]$, and G is stuttering invariant.

TLA is designed to be stutter insensitive. It should be noted that *state formulas* are no threat to this aim, because they are interpreted over individual states, thus they are trivially stutter invariant. However, *state transitions* (i.e. actions) have the potential to be stutter sensitive as our previous example 4.3 shows. In TLA, the $\circ$ (next) operator of ordinary temporal logic is absent because is incompatible with stuttering invariance. Instead the simpler postfix $'$ operator is available at the action level to refer to the value of variables at the next state. In order to enforce all the formulas to be stutter invariant, TLA generalizes what we did in 4.4 introducing the following special abbreviations as part of the syntax. Let A be an action formula and f a state formula:

$$[A]_f \triangleq A \lor (f' = f) \tag{4.5}$$

$$\langle A \rangle_f \triangleq A \land (f' \neq f) \tag{4.6}$$

Then, with this notation, TLA restricts the formation of temporal formulas in the following way:

- When $\Box$ is immediately followed by an action A, the action must be of the form $[A]_f$.
- When $\Diamond$ is immediately followed by an action A, the action must be of the form $\langle A \rangle_f$.

Consequently, our previous formulas HC, HMC and F are *not* legal TLA formulas. Also, although G is stuttering invariant it will be rejected by the parser, the action must be rewritten as $[x' = x + 1]_x$ for G to be accepted.

TLA is a proper subset of rTLA, and this restriction is enough to ensure any formula is stutter invariant. Formally, a fundamental theorem asserts that TLA is not expressive enough to distinguish between stuttering equivalent behaviours.

Theorem 4.1 (Stuttering invariance)**.** Assume α is a TLA formula and that σ, τ are behaviours such that $\sigma \simeq \tau$. Then $[\![\alpha]\!] = [\![\alpha]\!]_{\simeq}$

Proof. See [32]. □

From now on, we will work with proper TLA (and not rTLA). Accordingly, our clock systems HC and HMC should be:

$$\begin{aligned} HC &\triangleq HCini \wedge \Box[HCnxt]_h \\ HMC &\triangleq HMCini \wedge \Box[HMCnxt]_{\langle h,m\rangle} \end{aligned}$$

Recall that earlier we where interested in proving the following about HMC and HC

$$\models HMC \Rightarrow HC$$

Now we can argue (informally) why this holds. In any behaviour of HMC, there are sub-sequences of steps where $h' = h$ and $m' \neq m$. These behaviours are admitted by HC because HC does not know about m (i.e. m is irrelevant for HC) and allows h to stutter when only m is updated by HMC (see figure 4.2). Therefore, HMC is a refinement of HC.

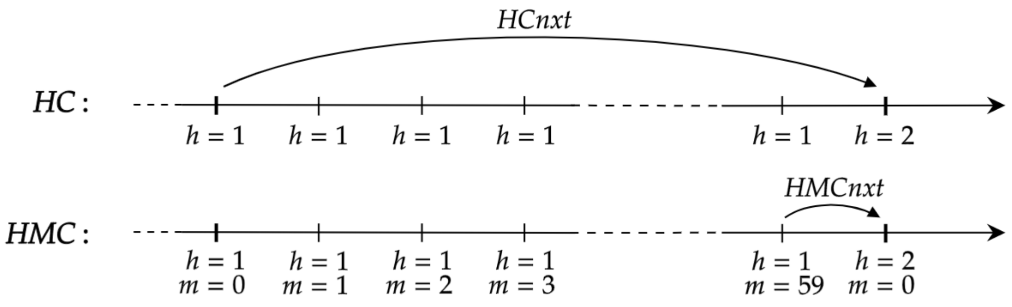


Figure 4.2: Illustration of the passage of time between hour one and hour two in both HC and HMC. This is an example of a contractive refinement, multiple steps in the low level spec are mapped to a single step of the high level spec. Only the last low level step correspond with an observable high level change.

4.2.3.2 Considering fairness

As a consequence of enforcing stuttering invariance, the following class of behaviours is now possible for HC:

$$h = h_0 \to h = h_0 \to h = h_0 \to \cdots$$

That is, suppose $h = h_0$ for some $h_0 \in 1..12$ in the initial state, then action $[HCnxt]_h$ admits variable h to stutter indefinitely, maybe forever. Recall that (by definition) $[HCnxt]_h = (HCnxt \lor h' = h)$, i.e. there is nothing guaranteeing $HCnxt$ to hold. A similar situation occurs for system HMC.

Formulas like HC and HMC specify *unfair* state machines, which allow infinite stuttering. Behaviors exhibiting infinite stuttering can be interpreted as that the system is stuck or is not progressing, which is commonly known as a *deadlock*. In particular, for the clock example, the clock was not intended to stop at any moment, but an infinite stuttering says otherwise. We say this is a situation of *bad* termination (*intended* termination is discussed in 4.2.7). To exclude this kind of undesirable behavior, *fairness conditions* should be added as part of the system specification.

The notion of fairness is not exclusive to TLA, it is ubiquitous in the theory of concurrency, being used as a criterion for a reasonable execution. In fact, fairness and liveness are related notions: fairness properties are a special case of liveness properties, however, they are not properties to be verified, rather conditions assumed to be enforced (e.g. in the scheduler) which in turn can be used to prove other liveness properties (e.g. termination). Informally, a fairness condition asserts that some action, say A, occurs *eventually* provided it is *sufficiently often* enabled. The most common interpretations of what *sufficiently often* means leads to the following notions of fairness (with respect to A):

- **Weak fairness (WF)**: A occurs eventually if it is *permanently* enabled after some point.
- **Strong fairness (SF)**: A occurs eventually if it is *infinitely often* enabled after some point.

Strong fairness is (as its name implies) stronger than weak fairness because an action can be *infinitely often* enabled without being *permanently* enabled. To identify adequate fairness conditions for a system is often a non-trivial task.

Now, we will see how fairness conditions are formalized in TLA. If A is an action, ENABLED A is a state formula stating there exists some next state that satisfies A. Formally, it may be defined as follows:

$$\text{ENABLED } A \ \triangleq\ \exists\, v_1', \ldots, v_n' : A \tag{4.7}$$

where $v_1', \ldots, v_n'$ are all the free primed variables in A. For example, let $A \triangleq x > 0 \land x' = x - 1$, action A can be read as "if $x > 0$ in the current state, then x is decremented by one in the next state". So A specifies a next state only if $x > 0$, thus ENABLED $A \equiv x > 0$. The two notions of fairness are formalized as follows for actions of the form $\langle A \rangle_f$:

$$\begin{aligned} \text{WF}_f(A) &\triangleq \Box(\Box\text{ENABLED } \langle A\rangle_f \Rightarrow \Diamond\langle A\rangle_f) \\ &\equiv \Diamond\Box\text{ENABLED } \langle A\rangle_f \Rightarrow \Box\Diamond\langle A\rangle_f \end{aligned} \tag{4.8}$$

$$\begin{aligned} \text{SF}_f(A) &\triangleq \Box(\Box\Diamond\text{ENABLED } \langle A\rangle_f \Rightarrow \Diamond\langle A\rangle_f) \\ &\equiv \Box\Diamond\text{ENABLED } \langle A\rangle_f \Rightarrow \Box\Diamond\langle A\rangle_f \end{aligned} \tag{4.9}$$

Since $\Diamond\Box\alpha$ implies $\Box\Diamond\alpha$ for any formula α, $\text{SF}_f(A)$ implies $\text{WF}_f(A)$ for any action A.

4.2.3.3 The canonical form

We are now ready to state the *canonical (or standard) form* to define the specification of a system as a (fair) state machine:

$$\mathit{Spec} \ \triangleq\ \mathit{Init} \land \Box[\mathit{Next}]_{vs} \land F \tag{4.10}$$

where

- State formula *Init* specifies the possible initial states.
- Action formula *Next* specifies the next-state relation, typically is a disjunction of sub-actions that describe atomic transitions of the system or of its environment.

- vs is the tuple of all variables used in the specification.
- F is typically a conjunction of fairness conditions (either $\textsc{wf}$ or $\textsc{sf}$) on (some of) the sub-actions of $Next$.

Note that $Spec$ is the conjunction of a safety formula (i.e. $Init \land \Box[Next]_{vs}$) and a liveness formula (i.e. F), thus following the topological characterization given by Alpern and Schneider. In practice, TLA specifications are written in the canonical form. Of course, one is not forced to do so, but this form represents a well-behaved subset of TLA formulas and, in principle, any system can be described as a single formula in this form. Also, as a matter of tool support, the model checker TLC only accepts formulas in this form, although, on the other hand, the theorem prover TLAPS is not restricted to it.

One may also want to use arbitrary liveness properties (e.g. formulas of the form $\Diamond\alpha$) in place of F, but this can lead to subtle issues. The "safety" component $Init \land \Box[Next]_{vs}$ only specifies a safety property and cannot imply a liveness property as it can stutter forever, but an arbitrary liveness formula *can* imply a safety property, in which case it can restrict the expected behaviour specified by the safety component in surprising ways. In general, is undesirable to have a liveness property that interferes with the safety component, although there may be exceptions. Specifications without this issue are called *machine closed*, and the canonical form 4.10 is guaranteed to be *machine closed.* A thorough treatment of this topic can be seen in [30, Chap. 8].

Coming back to our running example, a reasonable fairness condition for system HC is to assert weak fairness on the $HCnxt$ action, thus the clock is always guaranteed to eventually "tick" without getting stuck forever, and we can do similarly for system HMC. So, we can define the fair versions of HC and HMC according to form 4.10 as follows:

$$FairHC \triangleq HCini \land \Box[HCnxt]_h \land \textsc{wf}_h(HCnxt)$$

$$FairHMC \triangleq HMCini \land \Box[HMCnxt]_{\langle h,m\rangle} \land \textsc{wf}_{\langle h,m\rangle}(HCnxt)$$

4.2.4 Refinement

We already introduced the notion of refinement with the clock example and discussed how stuttering invariance plays an important role thereof. In general, a low level spec R may introduce more detail in a high level spec S, where "more detail" is represented by new variables. Newly introduced actions in R that only modify these new variables correspond to stuttering steps at the level of S and, by stuttering invariance, cannot invalidate S (exactly what happens in the clock example). However, actions that modify the variables also present in S must do so in ways that are allowed by (the next-state relation of) S, which is more complex and may require the concept of *hiding* discussed in next section. Roughly speaking, S must be able to "simulate" every R action.

We say R is a *refinement* of S if and only if any behaviour of R is also a behaviour of S, that is, the implication $R \Rightarrow S$ is valid. Assuming both specifications are in the canonical form 4.10, this implication can be decomposed in the following sub-goals

i. R satisfies the initial conditions of S.

$$Init^R \;\Rightarrow\; Init^S$$

ii. Any step of R is "simulated" by a step of S.

$$[Next^R]_{vs^R} \;\Rightarrow\; [Next^S]_{vs^S}$$

iii. R preserves the fairness conditions of S.

$$\Box[Next^R]_{vs^R} \wedge F^R \;\Rightarrow\; F^S$$

The implication can be mechanically verified in (at least) two ways: (1) By semantics means (i.e. $\models\; R \Rightarrow S$) using the TLC model checker. This is only for finite models, but is easier. (2) By deductive means (i.e. $\vdash\; R \Rightarrow S$) using the TLAPS theorem prover. This is more general, but needs more work.

For the deductive approach, sub-goals may be strengthened with invariant properties of R. Usually, at least a type invariant is needed to make the proof go through. For example, for the clock refinement discussed earlier, formally we need to know that h is *always* a number in the range 1..12 and not an arbitrary value of the universe, otherwise we cannot reason about h in the proof. Note that sub-goals i. and ii. do not involve temporal logic reasoning, thus, if fairness is ignored, the proof just needs the tools of classical logic.

4.2.5 Hiding of internal state

The canonical form 4.10 of specifications is useful for describing a system as a state machine, but it does not distinguish between variables that are visible at the interface and those that represent the internal state of the machine. For example, suppose we have the specification of a FIFO queue, say *FIFOSpec*, with variables $\langle in, out, q\rangle$ where q is a buffer of messages [30, Chap. 4]. Variables *in* and *out* are *external* variables, but q is an *internal* variable, and we may want to expose only an interface of the spec (not *FIFOSpec* itself) with q hidden. In TLA, this is done with the *existential temporal quantifier* $\boldsymbol{\exists}$ (not to be confused with $\exists$) i.e. we may define the interface of the spec as

$$FIFOSpecI \ \triangleq\ \boldsymbol{\exists}\, q : FIFOSpec$$

The meaning and use of $\boldsymbol{\exists}$ is analogous to those of $\exists$. Consider the ordinary mathematical expression $x \times y + b = 0$. Its truth depends on free vars x and y, but the abstraction $\exists\, x : x \times y + b = 0$ only depends on y. In a similar way, we can say the temporal expression $\boldsymbol{\exists}\, q : FIFOSpec$ only depends on its free variables reflecting the external visible behaviour (i.e. *in* and *out*).

In general, the formula $\boldsymbol{\exists}\, x : \alpha$ asserts that there exists some sequence of values, one in each state of the behavior, that can be assigned to the variable x that will make formula α true. Semantically, the definition of $\boldsymbol{\exists}$ is more complex than that of $\exists$, however, deductively it follows the same rules:

$$(\boldsymbol{\exists}\,\mathrm{i})\ \ \alpha[v := t] \Rightarrow \boldsymbol{\exists}\, v : \alpha \qquad\qquad (\boldsymbol{\exists}\,\mathrm{e})\ \ \frac{\alpha \Rightarrow \beta}{(\boldsymbol{\exists}\, v : \alpha) \Rightarrow \beta}\ \ v \text{ not free in } \beta$$

where v is a (flexible) variable and t is a state function. And we also have $(\boldsymbol{\exists}\, x : \alpha) \equiv \alpha$ if x not free in α.[24] Hiding the internals of the specification feels like the "correct" way

[24]The universal dual $\boldsymbol{\forall}$ is sometimes reported in the TLA literature, but as far as we know is it not

to do it, and this naturally leads to a more general canonical form:

$$Spec \triangleq \exists\, v_1, \ldots, v_n : Init \land \Box[Next]_{vs} \land F \tag{4.11}$$

where $v_1, \ldots, v_n$ are the *internal* variables of the specification.

But the concept of hiding and the use of $\exists$ is not just about aesthetics, is also related to refinement in an important way. Suppose we have a different specification of a FIFO, say *FIFOSpec*2, that does not use a buffer q, and we want to prove it is a refinement of *FIFOSpec*. Surely, we would not be able to prove the implication $FIFOSpec2 \Rightarrow FIFOSpec$ because they may differ in how they store messages internally (i.e. different data structures), but we can prove $FIFOSpec2 \Rightarrow \exists\, q : FIFOSpec$ where the internal representation q is hidden. This would be a more complex kind of refinement than the one in the clock example.

At this moment, none of the TLA related tools supports $\exists$. From the point of view of model checking, the complexity involved is believed to be too high (co-NP-complete in the number of states) to be worth while. From the point of view of theorem proving the complexity is not a problem, TLAPS will provably have support some day, but it is not a priority to implement it because, in fact, it is seldom necessary in practice. The kind of refinement that requires hiding can be proved by finding an appropriate substitution over variables (called a *refinement mapping*) without manipulating $\exists$ directly.[25]

4.2.5.1 Refinement mappings and their (non)existence

Let $Spec_1$ be a specification with internal variables $v_1, \ldots, v_n$ (abbrev. $\vec{v}$) and external variables $x_1, \ldots, x_m$ (abbrev. $\vec{x}$), and $Spec_2$ a specification with internal variables $u_1, \ldots, u_k$ (abbrev. $\vec{u}$) and the same external variables that $Spec_1$. As we noted previously, to compare different specifications we may need to hide the internal variables to abstract away from the internal implementation details. We can attempt to do this by working with $\exists\, \vec{v} : Spec_1$ and $\exists\, \vec{u} : Spec_2$ instead of $Spec_1$ and $Spec_2$.

very useful.

[25]According to Lamport, $\exists$ is like the stone in the (TLA$^+$) soup, "one eats everything but the stone". It was originally important for TLA, but the introduction of TLA$^+$ made it less relevant.

We may want to prove they are equivalent, that is:

$$(\exists\, \vec{v} : \mathit{Spec}_1) \;\equiv\; (\exists\, \vec{u} : \mathit{Spec}_2) \tag{4.12}$$

which amounts to proving two implications (i.e. two refinements)

$$(\exists\, \vec{v} : \mathit{Spec}_1) \;\Rightarrow\; (\exists\, \vec{u} : \mathit{Spec}_2) \tag{4.13}$$

$$(\exists\, \vec{u} : \mathit{Spec}_2) \;\Rightarrow\; (\exists\, \vec{v} : \mathit{Spec}_1) \tag{4.14}$$

Let us consider the first direction, as the other is analogous. We have that 4.13 is equivalent to

$$\mathit{Spec}_1 \;\Rightarrow\; (\exists\, \vec{u} : \mathit{Spec}_2) \tag{4.15}$$

provided variables $v_1, \ldots, v_n$ do not occur free in Spec_2. The proviso holds because we are assuming the only shared variables are $x_1, \ldots, x_m$.

Now, to prove 4.15 it suffices to show that for any behaviour σ satisfying Spec_1 there is at each state an assignment of variables $\vec{u}$ such that the resulting behaviour satisfies Spec_2. We do this syntactically. For each u_i we find an appropriate expression $\bar{u}_i$ in terms of variables $\vec{x}$ and $\vec{v}$ such that substituting each $\bar{u}_i$ for u_i we can prove

$$\mathit{Spec}_1 \;\Rightarrow\; \mathit{Spec}_2[u_1 := \bar{u}_1,\ \ldots,\ u_k := \bar{u}_k] \tag{4.16}$$

The substitution is called a *refinement mapping*. If 4.16 holds, we say Spec_1 refines Spec_2 under the given *refinement mapping*. Consequently, in practice we don't need to deal with $\exists$, but we need to find an appropriate refinement mapping. This partly explains why there is no support for $\exists$ yet. Considering again the two reciprocal goals 4.13 and 4.14, the refinement mapping does not need to be the same for both.

Note that in formula 4.16 we are just using informal substitution notation $\_[\_ := \_]$ like is customary in logic treatments. But this is not part of TLA syntax, so this is not a legal formula. In fact, TLA does not include a syntax for substitution. However, TLA$^+$ introduces syntax for modules and *module instantiation* which semantically behaves as a substitution but at the module level. Then, in practice, specifications are organized in modules, and from one module one can instantiate other module substituting the module's declared symbols. So, assume Spec_1 and Spec_2 are in modules $M1$ and $M2$ respectively, then we can instantiate $M2$ from $M1$ to prove the assertion 4.16 that in TLA$^+$ syntax looks as follows:

$$M2 \triangleq \textsc{instance } M2 \textsc{ with } u_1 \leftarrow \bar{u}_1,\ \ldots,\ u_k \leftarrow \bar{u}_k$$

$$\textsc{theorem}\ \ Spec_1 \Rightarrow M2!Spec_2$$

where THEOREM is TLA$^+$ syntax for the deduction symbol $\vdash$ (turnstyle), The theorem can be proved using TLAPS. Alternatively, the refinement can be verified by the model checker TLC (assuming specifications in the canonical form and finite bounds on the data types), after all is just an implication. A more detailed discussion of module syntax and the proof language is carried out in section 4.3. We talk about TLC in section 4.4.

The (non)existence of refinement mappings

Unfortunately, refinement mappings do not always exist. This is a more subtle issue, but hopefully we can illustrate the situation with our clock example. First, recall that we intuitively suggested that the hour-minute clock HMC is a special case of the hour clock HC. Then, thanks to stuttering invariance, we argued the implication $HMC \Rightarrow HC$ holds, confirming the intuition. There was no mention of a "refinement mapping", which is because it was not necessary to hide anything about variable h as this evolves with the same values in both systems (but at different paces). Alternatively it can be said that there is a trivial identity mapping implicitly involved (obviously $HC = HC[h := h]$).

Now, the reader may wonder about the converse question, that is, could HC be a special case of HMC?. Intuitively, yes, i.e. we claim that if we hide the minute in HMC the specifications must, in fact, be equivalent. But we cannot prove $HC \Rightarrow \exists\, m : HMC$, i.e. in this case stuttering alone is not enough. For, to begin with, HC does not control a minute variable like HMC and it runs "faster" than HMC. Intuitively speaking, when we proved $HMC \Rightarrow HC$, we proved something "slower" implements something "faster", the slow behaviours of HMC are no problem for HC because the later can wait to increment h doing 59 (or more) *stuttering* steps. But for the converse, we are trying to prove that something "faster" implements something "slower". There is *no* function mapping each state of HC (which changes only once an hour) to multiple states of HMC representing the same hour (at least 60 states for each hour), so there is no refinement mapping possible to prove $HC \Rightarrow \exists\, m : HMC$.

However, there is a solution. We can add to HC an *auxiliary variable* s to obtain a new specification, say HC^s, so that before each step where h changes it adds 59 steps where *only* s changes. Variable s is called an *stuttering variable* because it makes the internal variable h to stutter, it is just an artifice with the purpose of "slowing" the internal behaviour but having no effect on external visible behaviour. The expected relation between the original HC and the new HC^s should be $HC \equiv (\exists\, s : HC^s)$. Then we can prove $(\exists\, s : HC^s) \Rightarrow (\exists\, m : HMC)$ under a suitable refinement mapping and finally conclude $HC \Rightarrow (\exists\, m : HMC)$ as wanted. This technique is easily generalized for arbitrary specifications where we need to prove something "faster" implements something "slower".

But how to be sure that $HC \equiv (\exists\, s : HC^s)$ holds?, or more in general, how to be sure that adding "auxiliary variables" to a specification is a sound approach?. In [64], Abadi and Lamport provide what they call a "completeness" result. They exhibit a general classification of cases where a refinement mapping do not exist, and for each case they show that adding certain *auxiliary variables* and assuming some *reasonable conditions* there is a *refinement mapping* to prove refinement. Three kinds of auxiliary variables are proposed: *history*, *prophecy* and *stuttering*. For each case, there are some conditions to be checked that guarantee the correctness of the approach.

4.2.6 Composition

Let $Spec_1$ and $Spec_2$ be the specifications of two systems that are intended to work in parallel. The formula $Spec_1 \wedge Spec_2$ represents the parallel composition of both systems. Semantically, the composition is the intersection of their behaviours. Transitions that only modify the variables from one specification are trivially allowed as stuttering steps for the other and vice versa, but shared variables synchronize transitions between the specifications. In particular, if both specifications don't share variables then they are completely independent. So, stuttering invariance is also important for composition.

Assuming both specifications have the canonical form 4.10, their conjunction $Spec_1 \wedge Spec_2$ is not in the canonical form. We mentioned earlier that TLC will only accept canonical formulas. However, the conjunction can be transformed to canonical form using laws of classical logic and the following TLA equivalence for any action formulas A and B:

$$\Box[A]_{vs} \ \wedge \ \Box[B]_{us} \ \equiv \ \Box\big[[A]_{vs} \ \wedge \ [B]_{us}\big]_{\langle vs, us\rangle} \tag{4.17}$$

Example 4.2.3 (Composition of two clocks)**.** Consider the following two hour clock systems on variables x and y respectively:

$$HC1 \ \triangleq \ x \in 1..12 \ \wedge \ \Box[Tick(x)]_x$$

$$HC2 \ \triangleq \ y \in 1..12 \ \wedge \ \Box[Tick(y)]_y$$

where $Tick$ is the parameterized next-state action defined as

$$Tick(h) \ \triangleq \ h' = \text{IF } h \neq 12 \text{ THEN } h+1 \text{ ELSE } 1$$

Their conjoint operation is described by

$$\begin{aligned} TwoClocks \ &\triangleq \ HC1 \ \wedge \ HC2 \\ &\equiv \ \wedge \ x \in 1..12 \ \wedge \ y \in 1..12 \qquad\qquad \text{by } 4.17 \\ &\quad\ \wedge \ \Box\big[\vee \ Tick(x) \wedge Tick(y) \\ &\qquad\quad\ \vee \ Tick(x) \wedge (y' = y) \\ &\qquad\quad\ \vee \ Tick(y) \wedge (x' = x)\big]_{\langle x, y\rangle} \end{aligned}$$

The next state action of $TwoClocks$ is a disjuntion of three possibilities: (1) both clocks ticks simultaneously, (2) the first clock ticks but the second does nothing and (3) the second clock ticks but the first does nothing.

If we wanted to work according to an *interleaving* semantics, where each step represents an operation of only one component, the first possibility should be removed. This can be enforced with an additional condition on the composition asserting that always either x or y must be unchanged:

$$\begin{aligned} \mathit{TwoClocks} &\triangleq \mathit{HC}1 \wedge \mathit{HC}2 \wedge \Box[x' = x \vee y' = y]_{\langle x,y\rangle} \\ &\equiv \wedge\ x \in 1..12 \wedge y \in 1..12 \qquad\qquad \text{by 4.17} \\ &\quad \wedge\ \Box\big[\vee\ \mathit{Tick}(x) \wedge (y' = y) \\ &\qquad\quad \vee\ \mathit{Tick}(y) \wedge (x' = x)\big]_{\langle x,y\rangle} \end{aligned}$$

A generalization for an arbitrary number of clocks is possible [30, Chap. 10]. □

4.2.7 Handling termination

As Lamport explains in [34] (emphasis ours):

> "The observation that a single behavior can represent an execution of two or more noninteracting programs [e.g. the *TwoClocks* example] explains why we represent terminating as well as nonterminating executions by infinite behaviors. *Termination of a program means that it has stopped; it does not mean that the entire universe has come to a halt.*"

Formally, (good) termination is an intended deadlock where all the variables stop changing. Usually, an special action is used to detect termination, as we illustrate in the next example.

Example 4.2.4 (A terminating hour clock)**.** The following hour clock *HCD* ticks for a day (i.e. two cycles of twelve hours) and then stops. An auxiliary counter variable c is used to track the passage of time.

$$\begin{aligned} &\textsc{variables}\ c,\ h \\ &\mathit{HCDini} \triangleq c = 0 \wedge h \in 1\,..\,12 \\ &\mathit{HCDnxt} \triangleq \wedge\ c < 24 \\ &\qquad\qquad \wedge\ h' = \textsc{if}\ h \neq 12\ \textsc{then}\ h+1\ \textsc{else}\ 1 \\ &\qquad\qquad \wedge\ c' = c + 1 \\ &\mathit{Done} \quad\ \triangleq c = 24 \ \wedge \textsc{unchanged}\ \langle c,\ h\rangle \\ &\mathit{HCD} \quad\ \ \triangleq \mathit{HCDini} \wedge \Box[\mathit{HCDnxt} \vee \mathit{Done}]_{\langle c,\,h\rangle} \wedge \mathrm{WF}_{\langle c,\,h\rangle}(\mathit{HCDnxt}) \end{aligned}$$

When action *Done* occurs, *HCD* deadlocks indicating (good) termination. Note that

HCD is fair on the *HCDnxt* action, otherwise it would be possible to get deadlocked at any moment and that would be undesirable. The following implication asserts termination of *HCD*:

$$HCD \;\Rightarrow\; \Diamond(c = 24)$$

$\square$

4.2.8 The rules of TLA

A set of axioms and rules for TLA is provided by Lamport in [34]. Those rules, alongside ordinary first-order logic reasoning, form a relatively complete proof system for reasoning about systems in TLA. Other useful verification rules can be derived from those. In what follows we present some derived rules to deal with common safety properties. Their names are for our own reference and may differ with other presentations. Rules for liveness properties will not be discussed.

The most basic kind of safety property is *invariance*, which asserts that some *state formula* I is always true in every behavior satisfying the specification. If we prove the implication

$$Spec \;\Rightarrow\; \Box I$$

we say I is an invariant of *Spec*. Invariants characterize the set of reachable states of the system. Simple examples are the type correctness assertions given for the clock systems *HC* (example 4.2.1) and *HMC* (example 4.2.2). Other more interesting example is *partial correctness*, which in general may be expressed in the form $end \Rightarrow R$, where *end* is some state predicate indicating termination and R is the desired post-condition, so that the implication to prove is

$$Spec \;\Rightarrow\; \Box(end \Rightarrow R)$$

To prove *total correctness*, we also need to prove *end* eventually holds

$$Spec \;\Rightarrow\; \Diamond end$$

but this is instead a liveness property, which needs appropriate fairness conditions in the specification in order to be proved. An example of this was given for the clock *HCD* (example 4.2.4), where *end* would be $c = 24$. Fairness conditions are irrelevant for invariance, so they are omitted in the rules for invariance discussed below. There are also other kinds of invariance properties that can be considered like action invariants (i.e. a *transition formula* should always be true in any behaviour) or prefix invariants (e.g. asserting something must be true *at most once* in any behaviour) but those will not be discussed.

Rules for invariants

The basic rule for proving invariants is

$$\text{INV1}\ \frac{\mathit{Init} \Rightarrow I \qquad I \wedge [\mathit{Next}]_{vs} \Rightarrow I'}{\mathit{Init} \wedge \Box[\mathit{Next}]_{vs} \Rightarrow \Box I}$$

It express an induction principle: in order to prove I is true in all states of a behavior, it suffices to prove (1) the initial formula *Init* is true in the initial state, and (2) if I is true in any state of the behavior, then it is true in the next state of the behavior. A state formula I satisfying the hypothesis of this rule is an *inductive* invariant. It is clear form the hypothesis that the proof does not need temporal logic, it is classical reasoning over state formulas and actions. The same goes for the following rules.

Now, not all invariants are *inductive*, they may not satisfy hypothesis $I \wedge [\mathit{Next}]_{vs} \Rightarrow I'$. To prove an invariant property that is *not* inductive, say P, one needs first to find an inductive invariant I such that $I \Rightarrow P$ holds. So if I is true for all states in a behavior then P is true for all states in that behavior, hence $\mathit{Spec} \Rightarrow \Box I$ implies $\mathit{Spec} \Rightarrow \Box P$.[26] This is embodied in the following rule

$$\text{INV2}\ \frac{\mathit{Init} \Rightarrow I \qquad I \wedge [\mathit{Next}]_{vs} \Rightarrow I' \qquad I \Rightarrow P}{\mathit{Init} \wedge \Box[\mathit{Next}]_{vs} \Rightarrow \Box P}$$

It should be noted that to find an inductive invariant requires creativity, just like the loop invariant in the Hoare rules. The model checker can be helpful here to gain confidence, as it is much easier to prove something when we have confidence on it's truth. Almost always

[26]Formally, the justification is based on temporal rule $\alpha \Rightarrow \beta \vdash \Box\alpha \Rightarrow \Box\beta$.

an inductive invariant asserts type correctness of the variables, thus one approach is to start with an inductive invariant I_0 asserting something very basic like type correctness and then successively conjoin other conditions I_1, I_2, ..., etc. (preserving inductiveness) to obtain $I_k = I_0 \land I_1 \land ... \land I_{k-1}$ such that $I_k \Rightarrow P$ holds. Of course, this is easier said than done.

In general, we can use invariants we have already proved to prove others. The first of them needs to be proved with the first rule INV1. As a generalization of this rule let us use the invariance of I to help proving the invariance of P:

$$\text{INV3}\quad \frac{\mathit{Init} \land \Box[\mathit{Next}]_{vs} \Rightarrow \Box I \qquad \mathit{Init} \land I \Rightarrow P \qquad I \land I' \land P \land [\mathit{Next}]_{vs} \Rightarrow P'}{\mathit{Init} \land \Box[\mathit{Next}]_{vs} \Rightarrow \Box P}$$

If I is also an inductive invariant, the formula I' is not needed as it follows from $I \land \mathit{Next}$.

A rule for refinement

We discussed in section 4.2.4 how to prove that a low-level specification R refines a high-level specification S. Leaving fairness conditions aside, the refinement of the safety part can be justified by the following rule where I is an invariant of R strong enough to prove that R can be "simulated" by S at any step:

$$\text{REF}\quad \frac{\mathit{Init}^R \land \Box[\mathit{Next}^R]_{vs^R} \Rightarrow \Box I \qquad \mathit{Init}^R \land I \Rightarrow \mathit{Init}^S \qquad I \land I' \land [\mathit{Next}^R]_{vs^R} \Rightarrow [\mathit{Next}^S]_{vs^S}}{\mathit{Init}^R \land \Box[\mathit{Next}^R]_{vs^R} \Rightarrow \mathit{Init}^S \land \Box[\mathit{Next}^S]_{vs^S}}$$

4.3 TLA$^+$ statics: ordinary mathematics

In the previous section we presented some basic examples of specifications manipulating very simple data. However, in practice one usually needs to work with more complex data types. In this section we make emphasis in the static part of TLA$^+$ (the "+"), which allows to define arbitrarily complex data structures, organize specifications into modules and also provides a proof language to carry out formal proofs with TLAPS.

In fact, there are two versions of TLA$^+$: the original TLA$^+$ (released around 2000) and

TLA$^+$ version 2 (released around 2006) [65]. By TLA$^+$ we assume the latest version.

4.3.1 Modules

In section 4.2.5.1 we briefly mentioned modules and the module instantiation mechanism as the only form of substitution available in TLA$^+$, which in particular allows to carry out refinement proofs. A module serves as a (possibly parameterized) name space for the definitions inside. Modules can be extended and instantiated. Module instantiation is required to provide a substitution for the module's declared symbols (variables and constants). In general, to work with the tools TLC and TLAPS the specifications need to be organized into modules.

Figure 4.3 presents our earlier clock examples HC and HMC in their respective modules. Let us make some general observations:

- Modules can be extended using keyword EXTENDS. For example, both modules HC and HMC need to talk about natural numbers, so they extend the standard TLA$^+$ module *Naturals*.
- Each module declares the variables and constants that are relevant to its operation. In the case of HC and HMC, there are no constants involved.
- It is customary to use the same "Spec/Init/Next" naming nomenclature for the specification formulas in all modules because we can differentiate the main formulas by the notation $HC!Spec$ and $HMC!Spec$, and similarly for the other sub-formulas.
- Both modules HC and HMC define a type invariant formula and asserts a corresponding theorem using the THEOREM keyword. Theorems can be conveniently named. The proof language is discussed in section 4.3.4.
- Module HMC instantiates module HC using the INSTANCE keyword. The identity substitution $h \leftarrow h$ provided is redundant because it is assumed by default, but we are being explicit here. A refinement theorem is asserted referencing the specification of HC as $HC!Spec$.

```
──────────────────── MODULE HC ────────────────────
An hour clock system.
EXTENDS Naturals
VARIABLE h
───────────────────────────────────────────────────
Specification of HC
Init  ≜ h ∈ 1 .. 12
Next  ≜ h′ = IF h ≠ 12 THEN h + 1 ELSE 1
Spec  ≜ Init ∧ □[Next]_h ∧ WF_h(Next)
───────────────────────────────────────────────────
Type correctness of HC
TypeInv ≜ h ∈ 1 .. 12
THEOREM Thm_TypeInv ≜ Spec ⇒ □TypeInv
───────────────────────────────────────────────────
```

```
──────────────────── MODULE HMC ────────────────────
An hour-minute clock system.
EXTENDS Naturals
VARIABLES h, m
────────────────────────────────────────────────────
Specification of HMC
Init  ≜ h ∈ 1 .. 12 ∧ m ∈ 0 .. 59
Min   ≜ m′ = IF m ≠ 59 THEN m + 1 ELSE 0
Hour  ≜ ∨ m = 59 ∧ h′ = IF h ≠ 12 THEN h + 1 ELSE 1
        ∨ m < 59 ∧ h′ = h
Next  ≜ Min ∧ Hour
Spec  ≜ Init ∧ □[Next]_⟨h, m⟩ ∧ WF_⟨h, m⟩(Next)
────────────────────────────────────────────────────
Type correctness of HMC
TypeInv ≜ h ∈ 1 .. 12 ∧ m ∈ 0 .. 59
THEOREM Thm_TypeInv ≜ Spec ⇒ □TypeInv
HMC is a refinement of HC
HC ≜ INSTANCE HC WITH h ← h
THEOREM Thm_Ref ≜ Spec ⇒ HC!Spec
────────────────────────────────────────────────────
```

Figure 4.3: Complete TLA$^+$ specifications for the clocks *HC* and *HMC*.

If we also want to prove the converse refinement, i.e. that *HC* refines *HMC*, we need to create another module to apply the technique of auxiliary variables discussed in section 4.2.5.1 and then instantiate *HMC* appropriately. But we will not do it here.

4.3.2 Zermelo-Fraenkel set theory with choice

As we noted earlier, TLA does not fix the interpretation of elementary function and predicate symbols. But TLA$^+$ instantiates TLA with a specific first order language based on first-order predicate logic with equality, namely a variant of the Zermelo-Fraenkel set theory with choice axiom (ZFC) which Lamport calls ZFM and considers as a "straightforward formalization of everyday mathematics" [66], thus fixing a standard set-theoretic interpretation in which every TLA$^+$ value is a set.

The *choose* operator

One important difference with the usual presentation of ZFC is the inclusion of Hilbert's ε operator, named CHOOSE in TLA$^+$ syntax, which can be used to formalize common claims like "pick an arbitrary x such that property P holds" that are usually found in algorithm descriptions or proofs.[27] The expression $\textsc{choose}\ x : P(x)$ denotes an arbitrary value x that satisfies $P(x)$ if one exists, otherwise it denotes an arbitrary value of the set universe. In particular, to pick an arbitrary element from a set we may write an operation like:

$$Any(S) \triangleq \textsc{choose}\ x : x \in S$$

Parameterized definitions such as *Any* are called *operators*. Even if S is empty, we have that $Any(S)$ denotes some fixed but arbitrary value. Thus, TLA$^+$ avoids undefinedness by the way of underspecification.

The $\textsc{choose}$ operator is governed by the following laws:[28]

$$(\exists\, x : P(x)) \quad\equiv\quad P(\textsc{choose}\ x : P(x)) \tag{4.18}$$

$$(\forall\, x : (P(x) \equiv Q(x))) \quad\Rightarrow\quad (\textsc{choose}\ x : P(x)) = (\textsc{choose}\ x : Q(x)) \tag{4.19}$$

The first law shows how the quantifier $\exists$ (and therefore $\forall$) can be expressed in terms of $\textsc{choose}$, the second law (typically named *extensionality*) asserts that the operator assigns the same witness (although unknown) to equivalent formulae P and Q. In the context of the temporal logic behind TLA$^+$, the choice is always the same (i.e. deterministic) inside each behaviour —and we do not know (or care) which is— but it is possibly different across different behaviours.[29]

The bounded version of the quantifiers, namely $\exists\, x \in S : P$ and $\forall\, x \in S : P$, are short-hand's for $\exists\, x : x \in S \wedge P$ and $\forall\, x : x \in S \Rightarrow P$ respectively. Also, the bounded choice $\textsc{choose}\ x \in S : P$ is a shorthand for $\textsc{choose}\ x : (x \in S) \wedge P$. These shorthands extend to multiple variables (e.g. $\exists\, x, y \in S : P$) as well in a similar manner.

All the TLA$^+$ constructs, except $\in$ (membership), $=$ (equality) and the propositional

[27] Historically, Hilbert introduced the ε operator in his ε-calculus, a reformulation of first-order predicate logic. The ε operator allows to eliminate quantifiers and the ι-terms (formed by the ι-rule) introduced by Russell to formally represent *definite* descriptions (e.g. the expression "the positive square root of 4" uniquely denotes 2 without using a name). The ε-terms have a more liberal interpretation than the ι-terms, they denote instead *indefinite* descriptions allowing to choose nondeterministically an unnamed individual satisfying the property.

[28] In the ε-calculus, the axiom $P(y) \Rightarrow P(\varepsilon\, x.P(x))$ is postulated, from which the first law can be derived as a theorem.

[29] Some newbies get confused thinking that the choice is a "random" element, which is not the case.

connectives, can be defined in terms of CHOOSE. For example, the following are some of the basic set-theoretic constructs [32]:[30]

$$
\begin{array}{rrcl}
\textbf{Enumeration:} & \{e_1, \ldots, e_n\} & \triangleq & \text{CHOOSE } S : \forall\, x : (x \in S \equiv x = e_1 \lor \cdots \lor x = e_n) \\
\textbf{Union:} & \text{UNION } S & \triangleq & \text{CHOOSE } M : \forall\, x : (x \in M \equiv \exists\, T \in S : x \in T) \\
\textbf{Binary union:} & S \cup T & \triangleq & \text{UNION } \{S, T\} \\
\textbf{Powerset:} & \text{SUBSET } S & \triangleq & \text{CHOOSE } M : \forall\, x : (x \in M \equiv x \subseteq S) \\
\textbf{Comprehension 1:} & \{x \in S : P\} & \triangleq & \text{CHOOSE } M : \forall\, x : (x \in M \equiv x \in S \land P) \\
\textbf{Comprehension 2:} & \{t : x \in S\} & \triangleq & \text{CHOOSE } M : \forall\, y : (y \in M \equiv \exists\, x \in S : y = t)
\end{array}
$$

The comprehension 1 construct can be thought as a set filter, for example $\{x \in 0..4 : x \,\%\, 2 = 0\} = \{0, 2, 4\}$. The comprehension 2 construct can be thought as a set map[31], for example $\{x * x : x \in 0..4\} = \{0, 1, 4, 9, 16\}$.

In general, for values defined in terms of CHOOSE the existence of some set satisfying the characteristic predicate should be proven. For example, consider a definition like:

$$c \triangleq \text{CHOOSE } x \in S : P(x)$$

Then, in order to prove a property $Q(c)$, we need to prove:

$$\begin{array}{l} \exists\, x \in S : P(x) \\ \forall\, x \in S : P(x) \Rightarrow Q(x) \end{array}$$

Standard sets and set equality

In TLA$^+$ all values are encoded as some set, so the number 2 is a set although we don't know what its elements are. An important built-in constant is the set of boolean values:

$$BOOLEAN \triangleq \{\text{FALSE}, \text{TRUE}\}$$

There is no distinction between the elements of *BOOLEAN* and the logical truth values. This identification is convenient as for any predicate P we have $P \equiv (P = \text{TRUE})$. How boolean operators are interpreted and the subtleties involved are discussed at length in [30, Chap. 16]. Another built-in constant is *STRING*, which stands for the set of all

[30] The constructs UNION S and SUBSET S formalizes the more ordinary notation $\bigcup_{i \in S} S_i$ and 2^S (or $\mathcal{P}(S)$) respectively.

[31] Although, of course, repeated elements may collapse due to set idempotence.

finite character strings (e.g. "hello" $\in$ *STRING*).

The usual sets of numbers *Nat* ($\mathbb{N}$), *Int* ($\mathbb{Z}$), *Real* ($\mathbb{R}$) and operators on them (e.g. $(+), (*), (-), (\div), (/)$, etc.) are defined in the three standard modules *Naturals*, *Integers* and *Reals* respectively. They satisfy the expected inclusion hierarchy $Nat \subset Int \subset Real$. There is no specific module for the rationals. Operator $(\div)$ denotes integer division whereas $(/)$ denotes ordinary division. Currently, there is no mechanical tool supporting the *Reals* module. The convenient and special syntax $a..b$ defines an integer interval set

$$a..b \triangleq \{n \in Int : a \leq n \land n \leq b\}$$

Because TLA$^+$ is an untyped language, an expression like $1 =$ "one" that may be rejected in a strongly typed formalism is syntactically legal. The value of the equality $1 =$ "one" is unknown to us because the encoding of integers and strings are *opaque*, however the formula $1 =$ "one" $\in$ *BOOLEAN* is true. So, the value of $1 =$ "one" its not undefined but we cannot determine it.[32] Similarly, we don't know if the set $\{1,$ "one"$\}$ has two elements or just one element, but we have that $\{1,$ "one"$\} \subseteq Nat \cup STRING$. It should be noted that the model checker will refuse to check equality between elements that are incomparable and by extension forbids sets with incomparable elements.

Other example of a legal, but nonsensical, expression is $1/0$. The answer for *what is the value of* $1/0$*?* is simply that *we don't know and we don't care*, and the same goes for $1 \in 2$.[33]

Defining something *undefined*

For some use cases, it may be useful to explicitly distinguish something as *undefined*. A common idiom in TLA$^+$ specifications is to define a constant symbol to denote some special value outside the domain of interest (i.e. a bottom, undefined or null value). For example, to work with possibly undefined natural numbers, we may define:

$$\bot \triangleq \text{CHOOSE } x : x \notin Nat$$

The laws of set theory ensure that no set can contain all values (i.e. there is no universal

[32]In a dynamically typed programming language it would be determinedly false.
[33]In some prover assistants based on dependent type theory, e.g. Lean and Coq, we have $1/0 = 0$.

set), hence there exists an element (here named $\bot$) that is not in Nat, and we can be sure that the set $\{1, \bot\}$ has two elements. This technique can also be useful to define partial functions augmenting the range with an special value, because functions in TLA$^+$ are total as we explain in the next section.

4.3.3 Functions and operators

Another departure from the usual presentation of ZFC is that in TLA$^+$ functions are not defined as sets of pairs but as primitives with certain properties. They are also values in the set universe, but with opaque encoding, like numbers or strings. Functions are understood in the mathematical sense (sans specific set encoding). They are always *total* and must not be confused with computations.

Any function, say f, has a domain denoted by $\textsc{domain}\ f$. Then, $[A \rightarrow B]$ denotes the set of all functions f such that $\textsc{domain}\ f = A$ and $f[x] \in B$ for all $x \in A$ (note that function application uses square brackets).[34] Two functions f and g are equal (in the extensional sense) if the following conditions hold:

$$\textsc{domain}\ f = \textsc{domain}\ g \tag{4.20}$$

$$\forall\, x \in \textsc{domain}\ f\ :\ f[x] = g[x] \tag{4.21}$$

A function can be explicitly described by the notation $[x \in S \mapsto e]$, which is reminiscent of the informal mapping notation (i.e. $x \mapsto e$) and the λ-calculus abstraction notation (i.e. $\lambda x.\, e$). For example, $[n \in Nat \mapsto 2 * x]$ is the doubling function, and we can also give it a name:

$$double \triangleq [n \in Nat \mapsto 2 * x]$$

Fortunately, TLA$^+$ allows us to do the same in a more convenient style:

$$double[n \in Nat] \triangleq 2 * x$$

But we could also define this doubling function as the following operator, which is just a

[34]Function application may be defined by the operator $Apply(f, x) \triangleq \textsc{choose}\ y : \langle x, y \rangle \in f$, so that $f(x)$ would be the expected abbreviation for $Apply(f, x)$. However, TLA$^+$ chooses $f[x]$ to distinguish function application from operator application.

parameterized definition:

$$Double(x) \triangleq 2 * x$$

Both, the operator *Double* and the function *double* are, at least intuitively, the same doubling function $2 * x$, i.e. we expect $double[x] = Double(x)$ for all $x \in Nat$. But they are different kind of objects. Functions and operators differ in some important ways:

- A function like *double* is by itself a complete expression denoting a value, but an operator like *Double* is not. For example, it makes perfect sense to write $double \in [Nat \rightarrow Nat]$ (i.e. the type of the function) or $double[2] \in Nat$, and both are indeed true. But, while $Double(2) \in Nat$ is also legal and true, the expression $Double \in ...$ is not even syntactically correct, i.e. it is meaningless just like writing $+ > 0$.
- Unlike operators, function must have a domain, which is a set. Then, a fundamental limitation is that the domain can't be any arbitrary collection, as there are collections "too large to be a set". These collections, called proper classes, do not exist inside the set universe. For example, the $Cardinality(\_)$ operator on finite sets can't be defined as a function.

These differences suggest that one may need to choose between a function or an operator depending on what is more appropriate for the problem at hand.

Multiple parameters and higher order

Functions may have multiple parameters. For example, we may define the function that adds two numbers as either:

$$add \triangleq [m \in Nat, n \in Nat \mapsto m + n] \tag{4.22}$$
$$add[m \in Nat, n \in Nat] \triangleq m + n$$

Here, multiple parameters are actually represented as tuples $\langle m, n\rangle \in Nat \times Nat$ so that $add \in Nat \times Nat \rightarrow Nat$ and, for instance, $add[1, 3] = 4$. This means the previous definitions are syntactic sugar for

$$\begin{aligned} add &\triangleq [\langle m, n\rangle \in Nat \times Nat \mapsto m + n] \\ add[\langle m, n\rangle \in Nat \times Nat] &\triangleq m + n \end{aligned} \tag{4.23}$$

In general, for sets A and B, their Cartesian product is $A \times B = \{\langle a, b\rangle : a \in A, b \in B\}$. For a third set C, their product is $A \times B \times C = \{\langle a, b, c\rangle : a \in A, b \in B, c \in C\}$, and so on.[35] Tuples are, in fact, sequences and these in turn are functions. We talk a bit of sequences later.

Alternatively, we may dispense the tuples and represent the same adding function in curried form:

$$\begin{aligned} add2 &\triangleq [m \in Nat \mapsto [n \in Nat \mapsto m + n]] \\ add2[m \in Nat] &\triangleq [n \in Nat \mapsto m + n] \end{aligned} \tag{4.24}$$

so that $add2 \in [Nat \to [Nat \to Nat]]$ and $add2[1][3] = add[1, 3] = 4$.

Operators can receive other operators as arguments, for example:[36]

$$Twice(op(\_), x) \triangleq op(op(x)) \tag{4.25}$$

so that $Twice(Double(1)) = 4$. The argument operator can be specified anonymously using the LAMBDA operator notation as $Twice(\text{LAMBDA } x : 2 * x,\ 1) = 4$.[37]

Recursive definitions

TLA$^+$ allows to define recursive functions. For example, the following function is the factorial of a non-negative integer n:

$$fact[n \in Nat] \triangleq \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * fact[n - 1] \tag{4.26}$$

Actually, this is just syntactic sugar for a fixed-point definition in terms of the choice

[35]Note that the Cartesian product is not associative, we have $A \times B \times C \neq (A \times B) \times C \neq A \times (B \times C)$ because $\langle a, b, c\rangle \neq \langle\langle a, b\rangle, c\rangle \neq \langle a, \langle b, c\rangle\rangle$.

[36]It is a deliberate decision that they can be at most second order.

[37]The LAMBDA notation is not a function in the λ-calculus sense, it's an operator. The notation for actual functions in TLA$^+$ is closer to the λ-calculus notion of function than the LAMBDA notation.

operator:

$$fact \triangleq \text{CHOOSE } f : f = [n \in Nat \mapsto \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * f[n-1]] \qquad (4.27)$$

Then, to deduce that *fact* is a function of domain *Nat* we must prove that there exists a function f that equals the given definition, otherwise, from the standpoint of theorem proving, we can't use *fact*. The standard TLAPS library provides some basic theorems to prove that recursive definitions like this are well defined. Not all recursive definitions define functions, if there is no function matching the definition then it defines some unknown value. For example, consider $f[n \in Nat] \triangleq 1 + f[n]$.

It is also possible to define recursive operators (since the second version of TLA$^+$). The factorial example as an operator is:

$$\begin{aligned} &\text{RECURSIVE } fact(\_) \\ &fact(n) \triangleq \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * fact(n-1) \end{aligned} \qquad (4.28)$$

The first line makes evident that this is not an an ordinary definition. The semantics of recursive operators was given by Lamport and George Gonthier in [67]. However, recursive operators are not allowed to be higher order, and it should be noted that the theorem prover TLAPS does not handle recursive operators yet.

To define higher order recursive functions, that can also be handled by the current mechanical tools, a combination of operator and recursive function definition can be used. We will see how to do it for our use case in chapter 5.

The EXCEPT construct

In order to "change" the value at some point of a function there is a special notation. If f is a function, then $[f \text{ EXCEPT } ![e_1] \mapsto e_2]$ is another function $\hat{f}$ that is the same as f *except* with $\hat{f}[e_1] = e_2$. Formally, this is just:

$$[f \text{ EXCEPT } ![e_1] \mapsto e_2] \triangleq [x \in \text{DOMAIN } f \mapsto \text{IF } x = e_1 \text{ THEN } e_2 \text{ ELSE } f[x]] \qquad (4.29)$$

So, for example, we can represent the null vector (array or list) $[0, 0, 0]$ of length 3 as the function $v_1 \triangleq [x \in 1..3 \mapsto 0]$. Then we can increment its last position as $v_2 \triangleq [v_1 \text{ EXCEPT } ![3] \mapsto v_1[3] + 1]$. So, v_2 is the same as v_1 except with $v_2[3] = v_1[3] + 1 =$

$0 + 1 = 1$. Also, we can use the symbol @ to refer to the previous value, so instead of $v_1[3] + 1$ we can more conveniently write $@ + 1$.

Extending the previous example, we can represent the null matrix (array of arrays or list of lists) $[[0, 0, 0], [0, 0, 0], [0, 0, 0]]$ of size 3×3 as the function (of functions) $m_1 \triangleq [x \in 1..3 \mapsto [y \in 1..3 \mapsto 0]]$. Then we can increment its last position in the diagonal as $m_2 \triangleq [m_1 \text{ EXCEPT } ![3][3] \mapsto @ + 1]$ where @ stands for $m_1[3][3]$.

Sequences and Records

Some of the most common data structures are sequences and records. They are just special cases of functions but with a bit of special syntax support. The EXCEPT construct applies to them just as to any function.

In particular, finite sequences are just functions defined on some prefix of $Nat \setminus \{0\}$, i.e. they are indexed from 1. As they are functions, indexes are accessed by square brackets. The standard module *Sequences* provides a bunch of common operators to handle finite sequences like $\_ \circ \_$ (binary concatenation), $Len(\_)$, $Append(\_, \_)$, etc. The set of all finite sequences over some set S, called $Seq(S)$, is defined as:

$$Seq(S) \triangleq \text{UNION } \{ [1..n \mapsto S] : n \in Nat \}$$

We already saw that we can write tuples using angular brackets, e.g. the triple $t \triangleq \langle \text{“a”}, \text{“b”}, \text{“c”} \rangle$ such that $t \in STRING \times STRING \times STRING$. In TLA$^+$, tuples are just sequences, the tuple notation used for t is syntactic sugar for:

$$\begin{aligned}[i \in 1..3 \mapsto \text{CASE } & i = 1 \rightarrow \text{“a”} \\ \Box\ & i = 2 \rightarrow \text{“b”} \\ \Box\ & i = 3 \rightarrow \text{“c”}]\end{aligned}$$

where the CASE syntax is a generalization of IF-THEN-ELSE (the squares are the case separators). So, t is also a sequence of strings with length 3, i.e. $t \in Seq(STRING)$ and $Len(t) = 3$. Quantifiers can be used on tuples/sequences, e.g. we can write $\exists \langle a, b \rangle \in Nat \times Nat : a = b$ instead of the more cumbersome $\exists p \in Nat \times Nat : p[1] = p[2]$.

Records are convenient structures to store different kinds of data. For example, suppose we wanted to associate object names with coordinates in 3D, the record $r \triangleq [name \mapsto$ "A", $coord \mapsto \langle 0, 0, 0 \rangle]$ may represent that an object named A is at the origin. This notation is syntactic sugar for:

$$[i \in \{\text{"name"}, \text{"coord"}\} \mapsto \text{CASE } i = \text{"name"} \rightarrow \text{"A"} \\ \Box\ i = \text{"coord"} \rightarrow \langle 0, 0, 0 \rangle]$$

Also, $[name : STRING, coord : Int \times Int \times Int]$ denotes the set of all records with the fields *name* and *coord* where the former is a string and the later is a triple of integers. So, we have that $r \in [name : STRING, coord : Int \times Int \times Int]$.

4.3.4 The proof language and theorem proving with TLAPS

TLA$^+$ includes a declarative and hierarchical proof language designed primarily to produce proofs that are easily readable and maintainable by the users. TLAPS (TLA Proof System) is a proof system for checking TLA$^+$ proofs [38]. In what follows, we carry out an overview of how these work together.

Declaring theorems

Consider the famous Goldbach conjecture. We can assert it is a theorem (i.e. $\vdash$ *Goldbach*) and left it unproven as follows:

$\text{THEOREM } Goldbach \triangleq$
$\quad \forall n \in Nat : n > 2 \land Even(n) \Rightarrow \exists p, q \in Nat : IsPrime(p) \land IsPrime(q) \land n = p + q$
OMITTED

Here, *Goldbach* is a name for the asserted formula. We do not know how to prove it (and apparently, nobody does), so we use the OMITTED clause to indicate we are deliberately asserting its validity without providing a proof. We are not really forced to use the OMITTED clause but it may be more informative. Or instead, we may assert that it is provable from the Peano axioms:

$$\textsc{theorem}\ \mathit{Goldbach} \triangleq$$
$$\forall n \in \mathit{Nat} : n > 2 \land \mathit{Even}(n) \Rightarrow \exists p, q \in \mathit{Nat} : \mathit{IsPrime}(p) \land \mathit{IsPrime}(q) \land n = p + q$$
$$\textsc{proof by}\ \mathit{PeanoAxioms}$$

Although, even assuming it is a theorem provable from Peano axioms and that these axioms can be captured in the formula *PeanoAxioms* [38], it is unlikely that a mechanical tool could check it without further user assistance. We will talk about proofs in a moment. The keywords LEMMA, PROPOSITION and COROLLARY are convenient synonyms of THEOREM.

In TLA$^+$, a theorem can assert either a formula, like the previous Goldbach example, or a sequent of the form ASSUME / PROVE. For example, the universal generalization rule can be expressed as follows: [39]

$$\begin{array}{lll} \textsc{theorem}\ \mathit{ForallGen} \triangleq & \textsc{assume new}\ P(\_),\ \textsc{new}\ S, & \\ & \quad \textsc{assume new}\ x \in S & \\ & \quad \textsc{prove}\ \ P(x) & \\ & \textsc{prove}\ \ \forall x \in S : P(x) & \\ \textsc{proof obvious} & & \end{array}$$

The clause OBVIOUS means this follows directly from the known facts (built-in axioms or inference rules) and available definitions. In practice, interesting facts are unlikely to be directly proved in this way. The keyword NEW introduces a new *constant* symbol (is a shorthand for NEW CONSTANT) which stands for any constant formula. There are other keywords for different levels of formulas. The following express a standard temporal logic rule:[40]

$$\begin{array}{ll} \textsc{theorem}\ \mathit{BoxDistrOverAnd} \triangleq & \textsc{assume temporal}\ F,\ \textsc{temporal}\ G, \\ & \textsc{prove}\ \ \Box(F \land G) \equiv \Box F \land \Box G \\ \textsc{proof obvious} & \end{array}$$

Because TLA$^+$ is a first order language, we cannot express these sequents as formulas.

[38] In fact, as TLA$^+$ is a first-order language, the Peano axioms can't be finitely expressed in a formula.
[39] Informally: *if* $\vdash P(x)$ choosing a new $x \in S$ *then* $\vdash \forall x \in S : P(x)$.
[40] Informally: $\vdash \Box(F \land G) \equiv \Box F \land \Box G$.

Sequents allow to write more general second-order theorems.

Declaring general assumptions

We can also assert general assumptions at the module level. This is common practice, although care must be taken as it could introduce inconsistency in the logic. For example, to work with a constant symbol S that is supposed to be a finite set we can write:

$$\begin{array}{l} \textsc{constant}\ S \\ \textsc{assume}\ \mathit{Finiteness} \triangleq \mathit{IsFiniteSet}(S) \end{array}$$

The keywords ASSUMPTION and AXIOM are convenient synonyms of ASSUME at the module level. But there is some minor difference, ASSUME and ASSUMPTION will be by default model checked when running TLC, whereas AXIOM will be ignored. This can be useful if for some reason we do not want some assumption to be model checked (e.g. it may be hard or impossible to model check). It should be noted that this kind of assertion works only for formulas. To assert a second-order assumption it would be necessary to write a sequent and left it unproven treating it like an axiom.

If there are assumptions and theorems present in the module, the theorems assume by default the assumptions as axioms. In general, suppose module M has declared ASSUME A and THEOREM T. We may want to instantiate M from other module (possibly with substitutions) like in

$$M \triangleq \textsc{instance}\ M\ \textsc{with}\ u_1 \leftarrow \bar{u}_1,\ \ldots,\ u_k \leftarrow \bar{u}_k$$

Then we can access the imported theorem T with $M!T$. But $M!T$ is now in fact the sequent

$$\begin{array}{ll} \textsc{assume} & \overline{A} \\ \textsc{prove} & \overline{T} \end{array}$$

where $\overline{A}$ and $\overline{T}$ denote that there may be substituted symbols in the original statements A and T, so that the theorem $\overline{T}$ depends on the assumption $\overline{A}$. To conclude $\overline{T}$ (i.e. to be able to use it) we need first to prove (or assume again) $\overline{A}$.

Writing proofs checked by TLAPS

A theorem may be followed by a *proof*, and proofs are written in an assertional and hierarchical style. A proof is either (1) a terminal proof (justified with BY or left unproved with OMMIT) or (2) a sequence of steps, some of which may also have proofs (i.e. sub-proofs), thus forming a tree-like structure. Every step is labeled in the format ⟨*depth*⟩ *name* where *depth* is the depth in the tree and *name* is an optional label (number or letter) to identify the step. Usually, steps are labeled so that can be referenced below in the BY clauses. Every sequence of proof steps must end with a QED step. The following illustrates how the proof of a theorem may look like:

```
THEOREM ...
⟨1⟩1. ...
  ⟨2⟩1. ... OBVIOUS
  ⟨2⟩2. ... BY ...
  ⟨2⟩ QED BY ...
⟨1⟩2. ...
  ⟨2⟩1. ... OMITTED    Comment: Maybe some day :)
  ⟨2⟩2. ... BY ...
    ⟨3⟩A. ... BY ...
    ⟨3⟩B. ... BY ...
    ⟨3⟩ QED BY ...
  ⟨2⟩ QED BY ...
⟨1⟩ QED BY ...
```

The proof language is prover agnostic. A proof step is checkable if there is some backend verifier that accepts it. The high level architecture of the theorem prover TLAPS is depicted in figure 4.4. The main component is the *proof manager* which interprets the proof language. At any point in the proof there is a current *obligation* that is to be proved. The obligation contains a *context* of known facts and definitions, and a *goal*. To check that the goal is effectively entailed by the context, the default behaviour of the proof manager is to send the obligations to three backend solvers in order:

1. A baseline SMT solver. By default is Z3 [68], but can be changed to CVC4 [69]. It can be explicitly invoked with the clause BY *SMT* or more specifically with BY *Z*3.

2. Zenon: a tableaux-based prover for first-order logic [70]. It can be explicitly invoked with the clause BY *Zenon*.

3. Isabelle/TLA$^+$: encodes the core of TLA$^+$ and instantiates some of the main semi-automatic proof methods of Isabelle. By default is invoked with tactic *auto*. It can be explicitly invoked with the clause BY *Isa* or more specifically with BY *IsaM*("tactic") to change the tactic.[41]

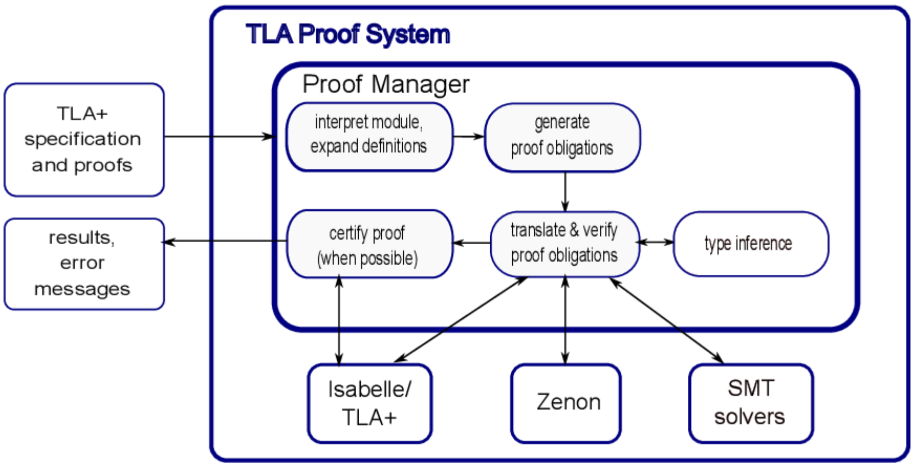

TLA Proof System
Proof Manager
TLA+ specification and proofs
results, error messages
interpret module, expand definitions
generate proof obligations
certify proof (when possible)
translate & verify proof obligations
type inference
Isabelle/ TLA+
Zenon
SMT solvers

Figure 4.4: General architecture of TLAPS. (source: [4]).

In general, the SMT solver is faster than Zenon which in turns is faster than Isabelle/TLA$^+$. Goals involving schematic variables require second-order unification and this can only be done by Isabelle/TLA$^+$ (e.g. when using induction rules).[42] If none of the three backends solvers find a proof, TLAPS reports a failure on the obligation. The trusted component of TLAPS is Isabelle/TLA$^+$, the idea is that the other backend provers should output a proof trace that can be certified with respect to Isabelle/TLA+. This certification mechanism is currently implemented only for Zenon.

We provide a concrete and simple example that cannot be proved in just one step. Let $a, b, c \in Nat$. If a a divides b and a divides c then a divides any linear combination of b and c, i.e. $(b * x) + (c * y)$ for any $x, y \in Nat$.[43] First, let us define m *divides* n, denoted by $m \mid n$, as:

$$m \mid n \triangleq \exists k \in Nat : n = m * k \tag{4.30}$$

[41]We have found that sometimes *IsaM*("blast") is needed to make the proof go through. The *blast* tactic, unlike *auto*, omits rewriting which can be counterproductive in some cases.

[42]Automation of higher-order steps is poor, typically requiring the user to break down proofs into smaller steps. Some recent research addresses this shortcoming with the introduction of a superposition-based automatic theorem prover [71].

[43]In fact, it is a theorem on the integers.

Informally, the proof is as follows. We have $a \mid b$ and $a \mid c$. To prove the conclusion it suffices to prove $a \mid (b * x) + (c * y)$ assuming x and y are arbitrary natural numbers. By the definition of divides, we have that there exists $k1$ and $k2$ such that $b = a * k1$ and $c = a * k2$. Now, we observe that:

$$(b * x) + (c * y) \;=\; (a * k1 * x) + (a * k2 * y) \;=\; a * (k1 * x + k2 * y) \tag{4.31}$$

So, we found there is a $k = k1 * x + k2 * y$ such that $(b * x) + (c * y) = a * k$. Therefore a divides the (arbitrary) linear combination QED.

The formalized version of the theorem and the proof are as follows:

```
THEOREM $DividesLinearCombination \triangleq$
    ASSUME NEW $a \in Nat$, NEW $b \in Nat$, NEW $c \in Nat$,
                $a \mid b,\ a \mid c$
    PROVE  $\forall x, y \in Nat : a \mid ((b * x) + (c * y))$
$\langle 1 \rangle$ SUFFICES ASSUME NEW $x \in Nat$, NEW $y \in Nat$
     PROVE  $a \mid ((b * x) + (c * y))$                      OBVIOUS     predicate logic
$\langle 1 \rangle$ PICK $k1 \in Nat : b = a * k1$                     BY  DEF $\mid$
$\langle 1 \rangle$ PICK $k2 \in Nat : c = a * k2$                     BY  DEF $\mid$
$\langle 1 \rangle 1.$ $(b * x) + (c * y) = ((a * k1) * x) + ((a * k2) * y)$   OBVIOUS     b = a * k1, c = a * k2
$\langle 1 \rangle 2.$              @ $= a * (k1 * x + k2 * y)$         OBVIOUS     arithmetic
$\langle 1 \rangle 3.$ $\exists k \in Nat : (b * x) + (c * y) = a * k$         BY $\langle 1 \rangle 1$, $\langle 1 \rangle 2$
$\langle 1 \rangle$ QED                                          BY $\langle 1 \rangle 3$  DEF $\mid$
```

The formal proof makes the steps more explicit, as expected. The first step is obviously justified by universal generalization (i.e. predicate logic). The PICK keyword, for existential instantiation, allows to use existential assumptions of the form $\exists x : P(x)$ picking a fresh z for which $P(z)$ is assumed, here it is used to pick some constants named $k1$ and $k2$. The equational reasoning 4.31 is formally represented by lines $\langle 1 \rangle 1$ and $\langle 1 \rangle 2$. There are some basic laws of arithmetic involved in this equalities, but fortunately they are automatically inferred by the backend solvers when justified by OBVIOUS. This allows to reduce the burden of proof to the higher level aspects that need human ingenuity. The penultimate step $\langle 1 \rangle 3$ introduces an existencial abstraction, then the last step match $\langle 1 \rangle 3$ with the goal using the definition of *divides*.[44]

[44]Note that definitions need to be expanded when they are required using the DEF clause, they are not globally visible by default as this could be counterproductive for the backend solvers, especially when there are too many definitions. Besides, the last two steps can be collapsed into one, but we usually prefer to be more explicit.

There is also another backend prover, called LS4, for temporal logic. This must be invoked explicitly with clause BY PTL (Propositional Temporal Logic) when needed. Figure 4.5 presents the rules of TLA we discussed earlier as theorems in the proof language. All of them require the temporal prover to check the main goal follows from the assumptions because there is some temporal logic involved. However, as we already noted before, the sub-goals in these rules do not need temporal logic.

THEOREM $INV1 \triangleq Init \land [Next]_{vs} \Rightarrow \Box I$
$\langle 1 \rangle 1.\ Init \Rightarrow I$
$\langle 1 \rangle 2.\ I \land [Next]_{vs} \Rightarrow I'$
$\langle 1 \rangle$ QED
  BY $\langle 1 \rangle 1,\ \langle 1 \rangle 2,\ PTL$

THEOREM $INV2 \triangleq Init \land [Next]_{vs} \Rightarrow \Box P$
$\langle 1 \rangle 1.\ Init \Rightarrow I$
$\langle 1 \rangle 2.\ I \land [Next]_{vs} \Rightarrow I'$
$\langle 1 \rangle 3.\ I \Rightarrow P$
$\langle 1 \rangle$ QED
  BY $\langle 1 \rangle 1,\ \langle 1 \rangle 2,\ \langle 1 \rangle 3,\ PTL$

THEOREM $INV3 \triangleq Init \land [Next]_{vs} \Rightarrow \Box P$
$\langle 1 \rangle 1.\ Init \land [Next]_{vs} \Rightarrow \Box I$
$\langle 1 \rangle 2.\ Init \land I \Rightarrow P$
$\langle 1 \rangle 3.\ I \land I' \land P \land [Next]_{vs} \Rightarrow P'$
$\langle 1 \rangle$ QED
  BY $\langle 1 \rangle 1,\ \langle 1 \rangle 2,\ \langle 1 \rangle 3,\ PTL$

THEOREM $REF \triangleq R!Spec \Rightarrow S!Spec$
$\langle 1 \rangle 1.\ R!Init \land [R!Next]_{R!vs} \Rightarrow \Box I$
$\langle 1 \rangle 2.\ R!Init \land I \Rightarrow S!Init$
$\langle 1 \rangle 3.\ I \land I' \land [R!Next]_{R!vs} \Rightarrow [S!Next]_{S!vs}$
$\langle 1 \rangle$ QED
  BY $\langle 1 \rangle 1,\ \langle 1 \rangle 2,\ \langle 1 \rangle 3,\ PTL$

Figure 4.5: The rules INV1, INV2, INV3 and REF embodied as theorems in the proof language.

The following is the formal proof of the type correctness invariant asserted for the clock system HMC (module in figure 4.3). It can be proved by rule INV1, so it is in particular an inductive invariant (as is normally the case for type correctness). The proof is trivial, in fact the step $\langle 1 \rangle 2$ could be proved in just one line, but here we prefer to make explicit the two cases, namely $\langle 2 \rangle A$ and $\langle 2 \rangle B$, involved in the proof: either the clock ticks or it stutters.

```
THEOREM $Thm\_TypeInv \triangleq Spec \Rightarrow \Box TypeInv$
⟨1⟩1. $Init \Rightarrow TypeInv$
  BY  DEF $Init$, $TypeInv$
⟨1⟩2. $TypeInv \land [Next]_{\langle h, m \rangle} \Rightarrow TypeInv'$
  ⟨2⟩ SUFFICES ASSUME $TypeInv$, $[Next]_{\langle h, m \rangle}$
               PROVE  $TypeInv'$
    OBVIOUS
  ⟨2⟩A. CASE $Next$                                  The clock ticks
    BY ⟨2⟩A  DEF $TypeInv$, $Next$, $Min$, $Hour$
  ⟨2⟩B. CASE UNCHANGED $\langle h, m \rangle$        The clock stutters
    BY ⟨2⟩B  DEF $TypeInv$
  ⟨2⟩ QED
    BY ⟨2⟩A, ⟨2⟩B  DEF $Next$
⟨1⟩ QED
  BY ⟨1⟩1, ⟨1⟩2, $PTL$ DEF $Spec$
```

Because much of the proof structure can be considered boilerplate, the Toolbox IDE provides a graphical UI that allows the user to automatically generate part of the proof structure.

Current limitations of TLAPS

TLAPS does not yet fully support TLA$^+$. Some of the current limitations are:

- Recursive operator definitions are not supported.
- There is no support for real numbers (the standard *Reals* module).
- The ENABLED operator is not supported. This means is not possible to do liveness proofs, as fairness conditions (WF and SF) are defined in terms of this operator.
- Many temporal operators (e.g. $\exists$) are not supported.

In the process of writing this thesis, a postdoctoral fellow is working in the implementation of the ENABLED operator. There are already some publications about this new feature and liveness proofs [72].

4.4 Model checking with TLC

TLC is a model checker for specifications written in TLA$^+$. More precisely, it accepts a subclass of TLA$^+$ specifications that should include most reasonable descriptions of real system designs [37]. It allows to check safety properties and also some simple forms of liveness properties. When it finds a violation, it reports an error trace where it is possible to explore the values of expressions at each step of the trace. TLC can be used in at least three different ways:

1. **Model checking:** tries to find the graph of all reachable states using breadth-first search (or optionally a depth-first search).

2. **Simulation:** checks an unending series of behaviors, each of which it constructs by starting from a randomly choosen initial state and repeatedly making a random choice of a possible next state.

3. **No behaviour:** this is used to evaluate constant expressions in a way somewhat akin to a REPL console. For example, if one writes $\{x \in 0..4 : x \,\%\, 2 = 0\}$ the result should be $\{0, 2, 4\}$.

The specification to be checked must be written in the canonical form 4.10 already discussed. Additionally, all the present data types must be bounded, which means that the verification is over a finite-state model of the specification. For example, a formula like $x \in Nat$ can't be checked as Nat is obviously an infinite set, instead one should write $x \in 0..N$ for some appropriate upper bound N. This also implies TLC does not accept formulas with general quantification like $\forall\, x : P$, it only accept bounded ones like $\forall\, x \in S : P$ where S is a bounded set. Fortunately, the Toolbox IDE allows the user to write the general specification (e.g. containing $x \in Nat$) and to separately manage finite models of the specification where the symbol Nat can be overridden with a finite set (e.g. $1..N$ for some N). This enables a clear separation between the general specification and its finite checkable models, so that theorem proving can be applied on the general specification if a general verification is desired.

TLC is developed in Java (like most TLA$^+$ tools) and it relies on its own implementation of the TLA$^+$ standard modules to evaluate specifications. For example, the *Naturals* module defines the natural numbers as usual by the Peano axioms, but it is not possible for TLC to use the Peano representation of numbers as it would be terribly slow. Besides, TLC allows the user to provide his own Java implementation to override defined operators, which is convenient and can be justified for performance reasons.

TLC works with an explicit representation of states, not a symbolic one like BDD. This decision was partly to avoid additional restrictions on the class of specifications that it could handle. TLC does not (yet) implement partial order reduction techniques. However, it may take advantage of *symmetry sets* to speedup model checking as explained in [73]. Currently, TLC may be run locally making use of multiple processors in a single computer, or may be run in distributed making use of multiple computers. We are not aware of any research related to GPU capability exploitation.

There is another model checker currently under development (still below v1.0), called APALACHE, which is based on symbolic techniques as it translates TLA$^+$ (also bounded) specifications into the logic supported by SMT solvers such as Microsoft Z3 [39]. Reported experiments shows it can outperform TLC for some use cases. We have not tested this alternative yet.

Chapter 5

Specification of the PCR pattern in TLA$^+$

Most engineers are looking for tools that can find bugs automatically without requiring any thought. Such tools are useful, but good systems are not built by removing the bugs from poorly designed ones.

LESLIE LAMPORT

In this chapter we present a formalization of the abstract PCR models and the functions they compute (chapter 3) in TLA$^+$ (chapter 4). There are mainly two themes that need to be discussed:

1. **The functional point of view (aka *What?*)**
   PCR behaviour was previously identified with certain functions under some algebraic assumptions. To deal with this formally, we organize the required mathematical concepts, properties and proofs across a collection of TLA$^+$ modules. To this end, only the *classical* mathematical fragment of the TLA$^+$ formalism (i.e. only the "+") is used, no temporal logic is required.

2. **The operational point of view (aka *How?*)**
   A formal concurrent semantics for PCRs is given in terms of temporal TLA$^+$ formulas. To this end, we carry out a straightforward translation of the PCR operations as *actions* in the TLA$^+$ sense so that a PCR execution can be seen as a (fair) labeled transition system.

Then, for the sake of verification, both points of view will be connected by the notion of (partial) *correctness*, i.e. the concurrent semantics (2) should compute the appropriate function (1). More precisely, the notions of (partial) *correctness* and *termination* will be formally expressed as special cases of *safety* and *liveness* properties, respectively, under the temporal logic framework provided by TLA$^+$. Additionally, specifications of different PCR models will be connected by the notion of *refinement* under appropriate substitutions. We will have the opportunity to use TLC and TLAPS to perform mechanical verification of these developments.

Most of our discussion will still be "abstract" in the sense that our PCR specifications are parameterized with symbols for the *basic functions* and other concrete elements that should be provided by the user under normal circumstances. In this way, an entire class of concrete PCRs (*models* in the formal logic sense) can be represented with a single specification (a *theory*), so that instantiating the specification parameters gives a particular concrete PCR.

5.1 A note about notation

In this chapter we will be working with the formal specification language TLA$^+$, which comes with some nice pretty printing capabilities implemented by the TLATeX tool which translates plain ASCII input into LaTeX. As usual, snippets of TLA$^+$ specs (and also complete specs in Appendix) will be presented in pretty printed style. By default, there is very limited support for two dimensional notation. For example $x\hat{}y$ is parsed as a binary operation and printed as x^y as expected. However this is already taken as exponentiation in the standard modules. There is no general support for subscript notation, x_y is parsed just a name and printed literally, i.e. it does not represent a binary operation on x and y. Consequently, things like the usual Euler style notation $\sum_{i=m}^{n} f(i)$ (and their variants) can only be defined and printed in linear form (e.g. $sum(m, n, f(\_))$).

When presenting fragments of the PCR specification we will mostly use standard TLA$^+$ notation. But when explaining concepts and specially when presenting the properties and proofs of the specification we will be more liberal and still be using some of the informal notation from chapter 3. For example, our notation $v^{I,i}$ will be formalized as a curried function denoted by $v[I][i]$ in TLA$^+$ but we prefer to use the former when possible. Anyway, we will try our best to be consistent in the use of notation and favour clarity of exposition over particular formalism details.

5.2 Initial assumptions

Before actually starting to discuss the specification, we want to fix hereinafter some ground assumptions about PCRs and their execution. These assumptions are made mostly in the interest of simplicity but some of them are also due to limitations on the tooling support that will be briefly mentioned.

5.2.1 Interleaving

We will assume an interleaving execution semantics, which is the most common approach adopted to model concurrency. This means that parallelism is reduced to the non-deterministic choice among their possible sequentializations. We briefly mentioned interleaving in the *TwoClocks* example 4.2.3 when discussing composition in TLA$^+$. In that example there where two components involved. In general, interleaving of n components may be enforced using the following condition over component's variables:

$$Interleave(vs_1, \ldots, vs_n) \triangleq \bigwedge_{i \neq j} \Box[vs_i' = vs_i \lor vs_j' = vs_j]_{\langle vs_i, vs_j \rangle} \tag{5.1}$$

Interleaving specifications are considered a reasonable abstraction, adequate for most practical purposes. Also, without interleaving, we have not been able to find a way for the model checker TLC to scale appropriately. It is a known shortcoming, and has to do with the way TLC enumerates the next states when computing reachable states. In contrast, this is not an issue for TLAPS. [1]

Ironically, interleaving is one of the reasons for the state explosion problem. Today, some

[1]It is true that TLA$^+$ has a linear-trace based semantics, however, as we already explained in previous chapter, at the language level TLA$^+$ does not commit to any specific execution semantics. The disjunction of two actions $A_1 \lor A_2$ naturally encompasses the possibility of simultaneity.

techniques exist to circumvent the problem at least for safety properties. These have originated in research on *truly-concurrent* semantics (e.g. Mazurkiewicz theory of traces used in [74]) where (very roughly) if two actions are *independent* (this must be known somehow) then they don't need to be interleaved which may help to reduce the verification effort.[2] Partial order reduction is one of these techniques, but is not implemented by TLC as was already noted in section 4.4.

5.2.2 Constant initial reduction value

We maintain our assumptions about the initial reducer value, but now it will be a *constant* value, that is, there will be no basic function r_0 to compute it. This means that our previous PCR examples $IsPrime1$ and $RedBlack$ can't be faithfully formalized, but we have in Appendix D another version of $IsPrime1$ called $IsPrime2$ to fill its place. In the case of $RedBlack$, it is out of reach because none of the tools have support for the *Reals* module yet.

This decision makes the formalization simpler because in this way the initial reducer value for any PCR is already known ahead of time, so there is no need to formalize exactly how or when it is actually computed. However, the real reason behind this decision is an error we found in TLAPS when handling parameterized module instantiation with a first order symbol, a feature we would need in order to allow an input dependent initial value. In contrast, TLC had no problem with it.

5.2.3 Simplified schemes

According to the definitions given in chapter 3 we allowed the basic functions to read from all the previous variables. A simplification we make now is to read only from the immediately preceding variable. For example, consider the scheme given for the basic

[2] As a simple example, consider a system of n independent components. In interleaving semantics there are $n!$ possible executions, whereas in truly-concurrency semantics there is only one parallel execution.

PCR (definition 3.1), now it reduces to

$$\begin{array}{rcl} p^i & := & f_p(x,p,i) \\ c_1^i & := & f_{c_1}(x,p,i) \\ c_2^i & := & f_{c_2}(x,c_1,i) \\ & \vdots & \\ c_k^i & := & f_{c_k}(x,c_{k-1},i) \\ r & := & r \otimes f_r(x,c_k,i) \end{array}$$

There is no loss of expressiveness, as each basic function could forward the previous values together with its output if it was necessary. Besides, we will work only with single consumer PCRs, so the previous scheme further reduces to

$$\begin{array}{rcl} p^i & := & f_p(x,p,i) \\ c^i & := & f_c(x,p,i) \\ r & := & r \otimes f_r(x,c,i) \end{array}$$

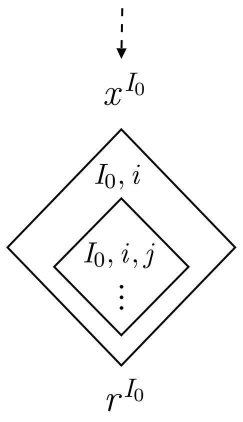
Again, there is no loss of expressiveness, as we can express the computation of multiple consumers with a single one. For example, if there are two consumers c_1 and c_2 such that $c_1^i = f_{c_1}(x,p,i)$ and $c_2^i = f_{c_2}(x,c_1,i)$ for any $i \in I_x$, we can express their effect with a single consumer c such that

$$c^i = f_c(x,p,i) = \big(f_{c_1}(x,p,i),\, f_{c_2}(x, \overrightarrow{f_{c_1}}(x,p), i)\big)$$

i.e. so that the consumer returns a pair with the original two consumer output values making it possible for the reducer to access the output values of c_1 and c_2 at any i, namely as $c^i[1]$ and $c^i[2]$ respectively.

5.2.4 Relativize the execution hierarchy to I_0

For the sake of uniformity in our specifications, we prefer not to assume that the main PCR being specified is located at the root of the execution hierarchy. Recall that, in general, indexes are sequences of natural numbers, formally objects of the form $I \circ \langle i \rangle \in \mathit{Seq}(\mathit{Nat})$ (which we informally write as I, i) where I is the index of the instance (also the index of the father's instance) and i is the current assignment at the inner scope. So, the point of view we adopt is that the main PCR being specified operates starting from some base index $I_0 \in \mathit{Seq}(\mathit{Nat})$, which *may* be the empty index $\langle\rangle$ (in which case it could be considered as the root of execution) but otherwise it would mean is located deeper in the execution hierarchy, thus reflecting a bit more general setting. Consequently, the execution hierarchy of what is being specified is relativized to I_0, and the high level dependence graph may be depicted as follows:



The output values of the internal components in $\diamond$ are indexed by sequences of the form I_0, i for the first level, I_0, i, j for the second level (if there is composition inside), and so on. This point of view may be interpreted as that there might be in the system environment other parallel instances of the same PCR operating from different father instances (i.e. $\neq I_0$). In general, this is true. However, we are not interested in those other instances, as their behaviours are independent from whatever happens starting from I_0 and all of them come from the same PCR being specified. For simplicity, in our specifications we assume there is a *single* instance, with index I_0, from which the main PCR operates. All the properties we care about the main PCR will be stated with respect to I_0.

5.3 Formalizing the mathematics underlying the functional point of view

Recall proposition 3.1 states the result of a basic PCR $\mathcal{A}$ on some input x as given by the following expression

$$\mathcal{A}(x) \;=\; \bigotimes_{i \in I_x} \vec{f}_{\mathcal{A}}^{\,i} \;=\; \mathtt{id}_{\otimes} \otimes \vec{f}_{\mathcal{A}}^{\,m} \otimes \vec{f}_{\mathcal{A}}^{\,m'} \otimes \cdots \otimes \vec{f}_{\mathcal{A}}^{\,n} \tag{5.2}$$

for some function $\vec{f}_{\mathcal{A}} : \mathbb{N} \to D$ (built from the PCR basic functions) and assuming $(D, \mathtt{id}_{\otimes}, \otimes)$ is an abelian monoid. The commutativity requirement can be dropped if a fixed reduction order is imposed.

It is evident that equation 5.2 constitutes an *informal* definition, albeit acceptable by ordinary mathematical standards. But, this time we need a suitable *formal* definition in the TLA$^+$ language that should also be manageable by the TLA$^+$ tools. To this end, we formalize the extension of a binary operation $\otimes$ to finite (not necessarily consecutive) intervals of $\mathbb{N}$, assuming certain algebraic properties. We call this generalization the *big operation* and denote it by $\bigotimes$.[3] Currently, the standard library of mathematics offered by TLAPS is not very large, at least not compared with other more mature and general purpose proof tools like e.g. Isabelle/HOL. However, there is (more than) enough support to develop the theory we need. In fact, the Community Repository [75], which is a fairly recent initiative to gather contributions from the TLA$^+$ community, includes some fold-like operators which could fit our needs —but without associated theorems and proofs. This is easily explained because those are relatively new contributions and the community is far more concerned with model checking techniques than with theorem proving.

Our formalization is completely independent of the PCR concept (although we do not claim it to constitute a general purpose library) and is organized in several modules that can be found in PCR/BaseModules (GitHub). Some of the most important are

[3] Also known as the *generalization* or *iterated* binary operation. There is nothing special about symbols $\otimes$ and $\bigotimes$, we may use the pair $\oplus$ and $\bigoplus$ as well. However, it should be noted that $\oplus$ is already taken for the multiset (bag) union operation in the standard TLA$^+$ modules, which is not a language primitive but still requires care in order to avoid conflict.

Module Name	**Description**
AbstractAlgebra	Definitions from elementary abstract algebra.
MonoidBigOp	The big operator assuming a monoid structure.
MonoidBigOpThms	Theorems for *MonoidBigOp*. Proofs are in a separate module at GitHub.
AbelianMonoidBigOp	The big operator assuming an abelian monoid structure.
AbelianMonoidBigOpThms	Theorems for *AbelianMonoidBigOp*. Proofs are in a separate module at GitHub.

Table 5.1: Modules of the TLA$^+$ specification for algebraic concepts and their theorems.

summarized in table 5.1 and can also be found at Appendix A. Induction principles over intervals of $\mathbb{N}$ are heavily used as the main proof method in those modules:

Theorem 5.1 (IntervalNatInduction)**.** Let $m, n \in \mathbb{N}$ and P a predicate:

i. Simple induction:

$$\frac{P(m) \qquad \forall\, i \in (m+1)..n : P(i-1) \Rightarrow P(i)}{\forall\, i \in m..n : P(i)}$$

ii. General induction:

$$\frac{\forall\, i \in m..n : (\forall j \in m..(i-1) : P(j)) \Rightarrow P(i)}{\forall\, i \in m..n : P(i)}$$

Proof. These are analogous to the very well known induction principles over $\mathbb{N}$ in their *simple* and *general* form, but here specifically stated for intervals of the form $m..n \subseteq \mathbb{N}$. We first prove i., then use it to prove ii.:

i. Define $Q(i) = i \in m..n \Rightarrow P(i)$. Then it suffices to prove $\forall\, i \in \mathbb{N} : Q(i)$. Apply *NatInduction* from the standard library invoking the Isabelle solver.

ii. Define $Q(k) = \forall\, i \in m..(k-1) \Rightarrow P(i)$. Then it suffices to prove $Q(m)$ and $\forall\, k \in (m+1)..n : Q(k-1) \Rightarrow Q(k)$. Apply i. invoking the Isabelle solver.

$\square$

In what follows we discuss in more detail some bits of the modules in table 5.1.

5.3.1 Abstract algebra

Module *AbstractAlgebra* defines some general algebraic structures as a hierarchy of concepts. A *magma* structure $(D, \otimes)$ consists of a set equipped D with a single binary operation $\otimes : D \times D \to D$ that must be closed by definition. No other properties are imposed.

$$Magma(D,\ \_ \otimes \_) \triangleq \forall\, x,\ y \in D : x \otimes y \in D$$

A *semigroup* structure $(D, \otimes)$ is a magma where the operation $\otimes : D \times D \to D$ satisfies the associativity law.

$$\begin{aligned} SemiGroup(D,\ \_ \otimes \_) \triangleq\ & \land\ Magma(D,\ \otimes\ ) \\ & \land\ \forall\, x,\ y,\ z \in D : (x \otimes y) \otimes z = x \otimes (y \otimes z) \end{aligned}$$

A *monoid* structure $(D, c, \otimes)$ is a semigroup having an identity element c.

$$\begin{aligned} Monoid(D,\ c,\ \_ \otimes \_) \triangleq\ & \land\ SemiGroup(D,\ \otimes\ ) \\ & \land\ \exists\, e \in D : \forall\, x \in D : \begin{aligned}[t] & \land\, e = c \\ & \land\, x \otimes e = x \\ & \land\, e \otimes x = x \end{aligned} \end{aligned}$$

A well known fact of the identity element in a monoid is its *uniqueness*: if we have two elements c_1 and c_2 such that $Monoid(D, c_1, \otimes)$ and $Monoid(D, c_2, \otimes)$ holds then $c_1 = c_2$. TLAPS can prove this fact through the Z3 solver directly from the *Monoid* definition without further assistance. [4]

An *abelian* (or *commutative*) *monoid* structure $(D, c, \otimes)$ is a *monoid* where the operation $\otimes : D \times D \to D$ satisfies the commutativity law.

$$\begin{aligned} AbelianMonoid(D,\ c,\ \_ \otimes \_) \triangleq\ & \land\ Monoid(D,\ c,\ \otimes\ ) \\ & \land\ \forall\, x,\ y \in D : x \otimes y = y \otimes x \end{aligned}$$

[4]Interestingly, the other alternative SMT CVC4 can't do the same.

5.3.2 Operation on monoid structure

Let $(D, Id, \otimes)$ be a monoid and $\{x\}_{i \in I}$ a family in D with finite index $I = m..n$ for integers $m \leq n$. First, we consider a sensible definition for the extended operation

$$x_m \otimes x_{m+1} \otimes x_{m+2} \otimes \cdots \otimes x_n$$

Let us write the indexed family more formally as a function $f : I \to D$, so that $f(i) = x_i$ for $i \in I$. Following a fairly standard approach, the extended operation $\otimes$ from m to n can be inductively defined and denoted by

$$\bigotimes_{i=m}^{n} f(i) \;=\; \begin{cases} f(m) & ,\; n = m \\ \displaystyle\bigotimes_{i=m}^{n-1} f(i) \otimes f(i) & ,\; n > m \end{cases} \tag{5.3}$$

which we call the *big operator* and corresponds to a left reduction

$$(\cdots(x_m \otimes x_{m+1}) \otimes x_{m+2}) \otimes \cdots \otimes x_n)$$

In fact what we have can serve more generally for semigroup structures. But we have not finished yet, and most importantly we are not formal enough yet. Next, we will see how to formalize 5.3 in TLA$^+$ and how to deal with some missing details.

Module *MonoidBigOp* extends *AsbtractAlgebra* and is parameterized by three symbols for which a monoid structure is assumed:

$$\textsc{constants}\; D,\; Id,\; \_\otimes\_$$

$$\textsc{axiom}\; Algebra \;\triangleq\; Monoid(D,\; Id,\; \otimes\,)$$

Symbol *Id* was noted $\texttt{id}_\otimes$ in chapter 3. To formalize 5.3, we define a recursive function *recDef* local to the higher order operator *bigOp*, which is parameterized by the interval bounds and the indexing function (recall the discussion about recursive definitions in 4.3.3):

$$\begin{array}{l} \mathit{bigOp}(m,\ n,\ f(\_)) \triangleq \textsc{let}\ \mathit{recDef}[i \in m\ ..\ n] \triangleq \\ \qquad\qquad\qquad\qquad \textsc{if}\ i = m\ \textsc{then}\ f(m) \\ \qquad\qquad\qquad\qquad\qquad\qquad \textsc{else}\ \ \mathit{recDef}[i-1] \otimes f(i) \\ \qquad\qquad\qquad\quad \textsc{in}\quad \mathit{recDef} \end{array}$$

The job of *recDef* is to traverse the interval $m..n$ using a standard recursive scheme that we can readily prove to be well defined using results from the standard TLAPS library. Note that the *bigOp* operator is by definition the function *recDef* of type $m..n \to D$.

Now we have the big operator expressed as a function (in the set theoretic sense), but what we really want is the resulting value of the operation over all $m..n$, which means evaluating *bigOp* at n (i.e. $\mathit{bigOp}(m, n, f(\_))[n]$). Besides, if we allow $m > n$ to be possible then $m..n = \varnothing$ and *bigOp* is undefined, so in this case we adopt the monoid identity as the result. Following this ideas, we define a relative of *bigOp* named *BigOp*:

$$\begin{array}{l} \mathit{BigOp}(m,\ n,\ f(\_)) \triangleq \textsc{if}\ m \leq n \\ \qquad\qquad\qquad\qquad\quad \textsc{then}\ \mathit{bigOp}(m,\ n,\ f)[n] \\ \qquad\qquad\qquad\qquad\quad \textsc{else}\ \ \mathit{Id} \end{array}$$

Note that $\mathit{bigOp} \in m..n \to D$ but $\mathit{BigOp} \in D$. We denote *BigOp* informally by $\displaystyle\bigotimes_{i=m}^{n} f(i)$.

We have previously only considered consecutive index sets of the form $m..n$. More general index sets can be represented by $\{i \in m..n : P(i)\}$ for some predicate P (the original $m..n$ interval can be recovered using a tautological predicate). We can cope with this if we "skip the holes" returning the monoid identity. For this we define a new big operator *BigOpP* in terms of *BigOp*:

$$\mathit{BigOpP}(m,\ n,\ P(\_),\ f(\_)) \triangleq \mathit{BigOp}(m,\ n,\ \textsc{lambda}\ i : \textsc{if}\ P(i)\ \textsc{then}\ f(i)\ \textsc{else}\ \ \mathit{Id})$$

As *BigOpP* is in fact just a straightforward abbreviation in terms of *BigOp*, almost all the results related to *BigOp* will have as immediate corollaries an analogous result for *BigOpP*. We denote *BigOpP* informally by $\displaystyle\bigotimes_{\{i \in m..n\, :\, P(i)\}} f(i)$.

5.3.2.1 Theorems

Module *MonoidBigOpThms* extends *MonoidBigOp* and gathers a bunch of useful theorems concerning the big operators. Any other module that needs to use these theorems is required to prove (or just assume again) the postulated monoid laws. The proofs of the theorems are in another module MonoidBigOpThms_proofs at GitHub. Here we present some of the results that will be referenced later with fairly rough proof sketches.

Theorem 5.2 (FunctionEq)**.** Let $f, g : m..n \to D$ such that $f(i) = g(i)$ for all $i \in m..n$. Then:

$$\bigotimes_{i=m}^{n} f(i) \;=\; \bigotimes_{i=m}^{n} g(i)$$

Proof. If $m > n$ the result is trivial, otherwise apply *IntervalNatInduction*. □

Corollary 5.1 (FunctionEqP)**.** Let $P : m..n \to Bool$ and $f, g : I \to D$ where $I = \{i \in m..n : P(i)\}$. If $f(i) = g(i)$ for all $i \in I$ then:

$$\bigotimes_{\{i \in m..n \,:\, P(i)\}} f(i) \;=\; \bigotimes_{\{i \in m..n \,:\, P(i)\}} g(i)$$

Proof. Direct by *FunctionEq*. □

Corollary 5.2 (PredicateEq)**.** Let $P, Q : m..n \to Bool$ and $f : I \to D$ where $I = \{i \in m..n : P(i) \wedge Q(i)\}$. If $P(i) \equiv Q(i)$ for all $i \in m..n$ then:

$$\bigotimes_{\{i \in m..n \,:\, P(i)\}} f(i) \;=\; \bigotimes_{\{i \in m..n \,:\, Q(i)\}} f(i)$$

Proof. Direct by *FunctionEq*. □

Corollary 5.3 (FalsePredicate)**.** Let $P : m..n \to Bool$ and $f : I \to D$ where $I = \{i \in m..n : P(i)\}$. If $\neg P(i)$ for all $i \in m..n$ then:

$$\bigotimes_{\{i \in m..n \,:\, P(i)\}} f(i) \;=\; \mathtt{id}_{\otimes}$$

Proof. Proceed as follows:

$$\bigotimes_{\{i \in m..n \,:\, P(i)\}} f(i) \;=\; \bigotimes_{i=m}^{n} \big(P(i) \to f(i),\, \mathtt{id}_{\otimes}\big) \;=\; \bigotimes_{i=m}^{n} \mathtt{id}_{\otimes} \;=\; \mathtt{id}_{\otimes}$$

□

Corollary 5.4 (TruePredicate)**.** Let $P : m..n \to \mathit{Bool}$ and $f : I \to D$ where $I = \{i \in m..n : P(i)\}$. If $P(i)$ for all $i \in m..n$ then:

$$\bigotimes_{\{i \in m..n \,:\, P(i)\}} f(i) \;=\; \bigotimes_{i=m}^{n} f(i)$$

Proof. Proceed as follows:

$$\bigotimes_{\{i \in m..n \,:\, P(i)\}} f(i) \;=\; \bigotimes_{i=m}^{n} \big(P(i) \to f(i),\, \mathtt{id}_{\otimes}\big) \;=\; \bigotimes_{i=m}^{n} f(i)$$

□

Theorem 5.3 (SplitUp)**.** Let $f : m..n \to D$ and $k \in m..n$ where $m \leq n$. Then we can split the operation at index k:

$$\bigotimes_{i=m}^{n} f(i) \;=\; \bigotimes_{i=m}^{k} f(i) \;\otimes\; \bigotimes_{i=k+1}^{n} f(i)$$

Proof. If $k = n$ the result is trivial, otherwise (i.e. $k \in m..(n-1)$) apply *IntervalNatInduction*.

□

Corollary 5.5 (SplitUpP)**.** Let $P : m..n \to \mathit{Bool}$, $f : I \to D$ and $k \in m..n$ where $I = \{i \in m..n : P(i)\}$ and $m \leq n$. Then we can split the operation at index k:

$$\bigotimes_{\{i \in m..n \,:\, P(i)\}} f(i) \;=\; \bigotimes_{\{i \in m..k \,:\, P(i)\}} f(i) \;\otimes\; \bigotimes_{\{i \in k+1..n \,:\, P(i)\}} f(i)$$

Proof. Direct by *SplitUp*. □

Theorem 5.4 (SplitDown)**.** Let $f : m..n \to D$ and $k \in m..n$ where $m \leq n$. Then we can split the operation at index $k - 1$:

$$\bigotimes_{i=m}^{n} f(i) \;=\; \bigotimes_{i=m}^{k-1} f(i) \;\otimes\; \bigotimes_{i=k}^{n} f(i)$$

Proof. If $k = m$ the result is trivial, otherwise (i.e. $k \in (m+1)..n$) apply *SplitUp*. $\square$

Theorem 5.5 (SplitLast)**.** Let $f : m..n \to D$ and $m \leq n$. Then we can extract the last term:

$$\bigotimes_{i=m}^{n} f(i) \;=\; \bigotimes_{i=m}^{n-1} f(i) \;\otimes\; f(n)$$

Proof. Proceed as follows:

$$\bigotimes_{i=m}^{n} f(i) \;=\; \bigotimes_{i=m}^{n-1} f(i) \;\otimes\; \bigotimes_{i=n}^{n} f(i) \;=\; \bigotimes_{i=m}^{n-1} f(i) \;\otimes\; f(i)$$

$\square$

Corollary 5.6 (SplitLastP)**.** Let $P : m..n \to \mathit{Bool}$ and $f : I \to D$ where $I = \{i \in m..n : P(i)\}$ and $m \leq n$. Then we can extract the last term:

$$\bigotimes_{\{i \in m..n \,:\, P(i)\}} f(i) \;=\; \bigotimes_{\{i \in m..n-1 \,:\, P(i)\}} f(i) \;\otimes\; \big(P(n) \to f(n),\, \mathtt{id}_{\otimes}\big)$$

Proof. Direct by *SplitLast*. $\square$

5.3.3 Operation on abelian monoid structure

The presence of commutativity for $\otimes : D \times D \to D$ enables a more flexible algebraic manipulation as any rearrangement of the elements under consideration is now allowed, i.e. order does not matter. Now, what is an appropriate way to formalize the fact that order does not matter?.

Perhaps a more "traditional" approach would be to note that for any finite set I there is a bijection $\chi : 1..n \to I$ where n is the cardinal of I, so we may define the big operator in terms of our previous definition 5.3 like

$$\bigotimes_{i \in I} f(i) \;=\; \bigotimes_{i=1}^{n} f(\chi(i))$$

Because of commutativity, we can choose an arbitrary bijection and therefore not commit ourselves to any specific order. Considering that TLA$^+$ possesses a choose operator, an approach like this seems like a good fit.

Another approach is to define the big operator as the unique homomorphism from the free abelian monoid (which is in bijection with the multisets). More precisely, let $I*$ be the free abelian monoid generated from I. Given any function $f : I \to D$ there is an unique function $\bigotimes_f : I* \to D$ satisfying the equations

$$\begin{aligned} \bigotimes_f \lbag\rbag &= \mathtt{id}_{\otimes} \\ \bigotimes_f \lbag i \rbag &= f(i) \\ \bigotimes_f (x \uplus y) &= \bigotimes_f x \otimes \bigotimes_f y \end{aligned}$$

where $\lbag\rbag$ is the empty multiset (identity of multiset union), $\lbag i \rbag$ is the singleton multiset, $\uplus$ is multiset union and $x, y \in I*$. This approach was introduced by Richard Bird in the context of constructive functional programming as a central method to define functions in the Bird-Mertens formalism [76].

However, we adopt a simpler and less general approach. As order does not matter, we argue that we can commit to any order as long as our reasoning does not depend on the specific choice. With this in mind, module *AbelianMonoidBigOp* extends *MonoidBigOp* to reuse the same definition as before but now demanding commutativity:

$$\text{AXIOM } \mathit{Commutativity} \triangleq \forall\, x, y \in D : x \otimes y = y \otimes x$$

5.3.3.1 Theorems

Module *AbelianMonoidBigOpThms* extends *AbelianMonoidBigOp* and the theorems in *MonoidBigOpThms* with more theorems concerning the big operators. Any other module that needs to use these theorems is required to prove (or just assume again) the postulated monoid laws and commutativity. The proofs of the new theorems are in another module AbelianMonoidBigOpThms_proofs at GitHub. Here we present some of the results that will be referenced later with fairly rough proof sketches.

Theorem 5.6 (SplitRandom)**.** Let $f : m..n \to D$ and $j \in m..n$ where $m \leq n$. Then we can extract the j-th term:

$$\bigotimes_{i=m}^{n} f(i) \;=\; \bigotimes_{\{i \in m..n \,:\, i \neq j\}} f(i) \;\otimes\; f(j)$$

Proof. Proceed as follows:

$$\begin{array}{rll}
\displaystyle\bigotimes_{i=m}^{n} f(i) & = \displaystyle\bigotimes_{i=m}^{j} f(i) \otimes \bigotimes_{i=j+1}^{n} f(i) & \text{by } \textit{SplitUp} \\
& = \displaystyle\bigotimes_{i=m}^{j} f(i) \otimes \bigotimes_{\{i \in j+1..n \,:\, i \neq j\}} f(i) & j \notin (j+1)..n \\
& = \displaystyle\left(\bigotimes_{i=m}^{j-1} f(i) \otimes f(j)\right) \otimes \bigotimes_{\{i \in j+1..n \,:\, i \neq j\}} f(i) & \text{by } \textit{SplitLast} \\
& = \displaystyle\left(\bigotimes_{\{i \in m..j-1 \,:\, i \neq j\}} f(i) \otimes f(j)\right) \otimes \bigotimes_{\{i \in j+1..n \,:\, i \neq j\}} f(i) & j \notin m..(j-1) \\
& = \displaystyle\left(\bigotimes_{\{i \in m..j-1 \,:\, i \neq j\}} f(i) \otimes \bigotimes_{\{i \in j+1..n \,:\, i \neq j\}} f(i)\right) \otimes f(j) & \text{by Associativity} \\
& & \text{\& Commutativity} \\
& = \displaystyle\left(\left(\bigotimes_{\{i \in m..j-1 \,:\, i \neq j\}} f(i) \otimes \mathtt{id}_{\otimes}\right) \otimes \bigotimes_{\{i \in j+1..n \,:\, i \neq j\}} f(i)\right) \otimes f(j) & \text{by Identity} \\
& = \displaystyle\left(\left(\bigotimes_{\{i \in m..j-1 \,:\, i \neq j\}} f(i) \otimes (j \neq j \to f(j), \mathtt{id}_{\otimes})\right) \otimes \bigotimes_{\{i \in j+1..n \,:\, i \neq j\}} f(i)\right) \otimes f(j) & \text{obviously } j = j \\
& = \displaystyle\left(\bigotimes_{\{i \in m..j \,:\, i \neq j\}} f(i) \otimes \bigotimes_{\{i \in j+1..n \,:\, i \neq j\}} f(i)\right) \otimes f(j) & \text{by } \textit{SplitLastP} \\
& = \displaystyle\bigotimes_{\{i \in m..n \,:\, i \neq j\}} f(i) \otimes f(j) & \text{by } \textit{SplitUpP}
\end{array}$$

□

Corollary 5.7 (SplitRandomP)**.** Let $P : m..n \to Bool$, $f : I \to D$ and $j \in m..n$ where $I = \{i \in m..n : P(i)\}$, $m \leq n$ and $P(j)$. Then we can extract the j-th term:

$$\bigotimes_{\{i \in m..n \,:\, P(i)\}} f(i) \;=\; \bigotimes_{\{i \in m..n \,:\, P(i) \wedge i \neq j\}} f(i) \;\otimes\; f(j)$$

Proof. Direct by *SplitRandom*. □

5.4 Formalizing the abstract PCR models

In this section we present the formalization in TLA$^+$ of the abstract PCR models introduced in chapter 3. Table 5.2 summarizes all the TLA$^+$ modules representing the abstract PCR models. They can be found at PCR/AbstractModels (GitHub) and also in Appendix B. Each module includes the corresponding functional and operational specification. We distinguish the use of an ordinary reducer vs a left reducer by formalizing them in separate modules because the corresponding algebraic assumptions are not the same. For example, module *PCR_A* formalizes an ordinary basic PCR and assumes an abelian monoid, but *PCR_ArLeft* formalizes a basic PCR with left reducer and assumes a monoid. However, we can prove that *PCR_ArLeft* is a refinement of *PCR_A*.

Module Name	**Description**	**Concrete example**
PCR_A	Basic PCR. See def. 3.1.	FibPrimes1, IsPrime2
PCR_ArLeft	Basic PCR with left reducer assuming a consecutive iteration space.	ListId
PCR_A1step	Basic PCR as a one step computation.	
PCR_A_c_B	PCR composed through consumer with basic PCR. See def. 3.4.	FibPrimes2
PCR_A_r_B	PCR composed through reducer with basic PCR. See def. 3.5.	
PCR_DC	Divide-and-conquer PCR. See section 3.3.2.	MergeSort1, NQueensDC
PCR_DCrLeft	Divide-and-conquer PCR with left reducer.	Merge
PCR_DC_r_DCrLeft	Divide-and-conquer PCR composed through reducer with other divide-and-conquer PCR with left reducer.	MergeSort2
PCR_A_it	PCR with iterative consumer over a basic function. See def. 3.6.	
PCR_A_it_B	PCR with iterative consumer over a basic PCR. See def. 3.6.	NQueensIT

Table 5.2: Modules of the TLA$^+$ specification for the abstract PCR models.

All modules are (logically) partitioned in different sections for better clarity. The order of this sections is not completely arbitrary, as the TLA$^+$ parser works in a single pass. In general, the sections are:

- Constants and variables: symbol parameters are declared for each PCR. This includes symbols for the PCR variables, auxiliary variables, basic functions, iteration space, and more.
- General definitions: some basic definitions for operators that are not specific to any PCR. They could possibly be extracted to another module but they are very few so we don't bother.
- PCR definitions and assumptions: the definitions and assumptions needed to characterize each PCR. This includes definitions for the iteration space, dependencies, and more. Some assumptions are declared for the basic functions and others.
- The functional specification: each PCR characterized as a function of streams.
- The operational specification: a temporal formula in the canonical form specifying the conjoint behaviours of the PCRs.
- Properties: the properties, safety or liveness, satisfied for each PCR. This may also include refinement concerning other PCR module.

In what follows, we discuss each module in more detail, starting with the basic PCR which will serve as a baseline to explain the fundamental aspects of our approach. Most of the other modules can be considered extensions or variations of the basic PCR, and consequently shall not be discussed at the same length. Nevertheless, we will make sure to stress the more relevant differences when appropriate. The properties section of the specification shall be presented later in section 5.5 when discussing verification.

5.4.1 Basic PCR

The main reference for this formalization is the definition of a basic PCR given in 3.1. Next we review each section of the corresponding specification.

5.4.1.1 Variables and constants

The constant symbols used in the specification to characterize the PCR elements are the following:

- I_0: the base index of the execution hierarchy (recall assumption 5.2.4).
- $\mathit{pre}(\_)$: a precondition predicate useful for some specifications where the input is expected to have some very specific characteristic.
- T, Tp, Tc, D : the range types for, respectively, the PCR input variable, the producer variable, the consumer variable and the reducer variable (i.e. the PCR output variable).
  These where noted as T, T_p, T_c and D in chapter 3.
- id and $\mathit{Op}(\_,\_)$: the initial reducer value and the binary combiner operation. id is expected to be the identity of the combiner.
  These where noted as $\mathtt{id}_\otimes$ and $\otimes$ in chapter 3.
- $\mathit{lBnd}(\_)$, $\mathit{uBnd}(\_)$ and $\mathit{prop}(\_)$: operators representing the basic functions determining the iteration space.
- $\mathit{fp}(\_,\_,\_)$, $\mathit{fc}(\_,\_,\_)$, $\mathit{fr}(\_,\_,\_)$ and $\mathit{gp}(\_,\_)$: operators representing the basic functions for, respectively, the producer, the consumer, the reducer and the simplified producer (recall remark 3.4).
  These where noted as f_p, f_c, f_r and g_p in chapter 3.
- $\mathit{Dep\_pp}$, $\mathit{Dep\_pc}$ and $\mathit{Dep\_cr}$: pairs of look behind/ahead sets of the form

  $$\langle \{b_1, b_2, \dots\}, \{a_1, a_2, \dots\} \rangle$$

  representing the non-linear dependencies between: the producer with itself, the consumer with respect to producer and the reducer with respect to consumer.

The expected types of the constant symbols are postulated axiomatically in section PCR definitions and assumptions of the specification.

The state of a basic PCR is represented by the following five (actually six) variables. Their expected types are not assumed, they are asserted as a type correctness invariant in section Properties of the specification, but we also briefly mention them here as it is useful to know them in advance.

- X[5] : The input of the PCR. Is a (partial) function from indexes to input type T:

  $$X \in [Seq(Nat) \rightarrow T \cup \{Undef\}]$$

  It is well defined only on index I_0 (recall assumption 5.2.4) with the initial value of in (see next item). The symbol $Undef$ is used to define partial functions, it is defined in section General definitions of the specification.

- in : An artificial variable used to let TLC check the specification on a specified range of input values in one go. Its value is used to define X at I_0, we have $in \in T$. This variable is maintained constant in each behaviour, and is not needed for TLAPS.[6]

- p : The output variable of the producer. It is a function mapping indexes to partial functions from Nat to producer type Tp.

  $$p \in [Seq(Nat) \rightarrow [Nat \rightarrow Tp \cup \{Undef\}]]$$

- c : The output variable of the consumer. It is a function mapping indexes to partial functions from Nat to consumer type Tc.

  $$c \in [Seq(Nat) \rightarrow [Nat \rightarrow Tc \cup \{Undef\}]]$$

- r : The output variable of the reducer (i.e. the output of the PCR). It is a function from indexes to output type D:

  $$r \in [Seq(Nat) \rightarrow D]$$

[5]A minor nitpick: we use X in capitals because if we declare x in TLA$^+$ then the parser would not let us use it again as a parameter name, which would be a bit restrictive considering x is a very common name. In general, in TLA$^+$ there is no way to distinguish "global" variable names and "local" variable names (except when instantiating between different modules). However, x could be used bounded to a quantifier with no problem.

[6]We tried to dispense with this artifice whose only purpose is to make model checking more convenient. However, the alternatives weren't much better. Anyways, for most purposes it can be safely ignored like if was not present.

- rs : An auxiliary variable to track the reductions done at any moment. It is a function mapping indexes to functions from Nat to $BOOLEAN$:

$$rs \in [Seq(Nat) \rightarrow [Nat \rightarrow BOOLEAN]]$$

The variables p, c, r and rs represent the internal behaviour of the PCR, whereas the variables X and r represent the external visible behaviour. Other PCR may want to write on input X or to read from output r. For convenience, the internal variables are curried functions separating the index component $I \in Seq(Nat)$ from the assignment component $i \in Nat$.

Notice that in TLA$^+$ notation all these variables (except in) are accessed with (multiple) square brackets as they are functions (of functions). For internal variables we have, for example, that $p[I]$ denotes the producer stream at index I whereas $p[I][i]$ denotes the value of the i-th assignment of the producer stream at index I. Sometimes we will write them in super-index notation like was customary in chapter 3, e.g. p^I and $p^{I,i}$ respectively. For external variables we have, for example, that $X[I \circ \langle i \rangle]$ denotes the input at index $I \circ \langle i \rangle$ where I is the index of the father PCR and i is the assignment on the scope of the father PCR. Sometimes we may write this informally either as $X^{I,i}$ or $X^{I \circ \langle i \rangle}$ depending on the situation, but in general the distinction should not be a concern.

5.4.1.2 General definitions

The meaning of being "undefined" is defined as something outside PCR main variables range types:

$$Undef \triangleq \text{CHOOSE } x : x \notin \text{UNION } \{T,\ Tp,\ Tc,\ D\}$$

Notice this definition involves an unbounded CHOOSE, but TLC does not accept unbounded quantifiers. Finite models for TLC must override the symbol with a *model value*: an unspecified value that TLC considers to be unequal to any other value.

Testing for "undefinedness" is done with the wrt predicate:

$$wrt(v) \triangleq v \neq Undef$$

For example, we may test with $\mathit{wrt}(p^{I,i})$ if on instance indexed by I the producer assignment at i has already been written. A generalized version is useful to test for multiple assignments:

$$\mathit{wrts}(v, S) \triangleq \forall\, k \in S : \mathit{wrt}(v[k])$$

For example, we may test with $\mathit{wrts}(p^I, S)$ if on instance indexed by I every producer assignment in set S has already been written. If S is the iteration space and the test result true, it means there is no producer assignment left to run, i.e. the producer terminated.

We are also interested in comparing written values from different sources:

$$\mathit{eqs}(v1, v2, S) \triangleq \forall\, k \in S : \mathit{wrt}(v1[k]) \wedge v1[k] = v2[k]$$

For example, we say two streams $v1$ and $v2$ (they can be variables or functions) are *equal* with respect (or relative) to S if $\mathit{eqs}(v1, v2, S)$ holds true.

5.4.1.3 PCR definitions and assumptions

The specification uses the following aliases for indexes (*Index*), assignments (*Assig*) and streams ($\mathit{St}(\_)$):

$$\begin{aligned} \mathit{Index} &\triangleq \mathit{Seq}(\mathit{Nat}) \\ \mathit{Assig} &\triangleq \mathit{Nat} \\ \mathit{St}(R) &\triangleq [\mathit{Assig} \to R \cup \{\mathit{Undef}\}] \end{aligned}$$

The aliases *Index* and *Assig* are used instead of their meanings so that we can override this symbols with appropriate bounded sets when doing model checking. Of course, we can also override *Nat* globally, but that is undesirable as normally there are different uses of *Nat* across the specification with different upper bounds. For theorem proving, these names are just expanded and forgotten. Let R be any type, $\mathit{St}(R)$ abbreviates the type of a stream function with range R, what we noted as $\overrightarrow{R}$ in chapter 3. For example, with this definitions the types of p, $p[I]$ and $p[I][i]$ are written in the specification as $[\mathit{Index} \to \mathit{St}(\mathit{Tp})]$, $\mathit{St}(\mathit{Tp})$ and Tp respectively.

Well defined indexes representing a PCR execution (either ongoing or terminated) are identified as those indexes for which the input has been written:

$$\mathit{WDIndex} \triangleq \{\, I \in \mathit{Index} \;:\; \mathit{wrt}(X[I]) \,\}$$

The iteration space of the PCR is defined exactly like in the informal presentation:[7]

$$\mathit{It}(x) \triangleq \{\, i \in \mathit{lBnd}(x)..\mathit{uBnd}(x) \;:\; \mathit{prop}(i) \,\}$$

The predicates *red* and *end* indicating reduction of an assignment and termination respectively, for an instance indexed by I, are defined as:

$$\begin{aligned} \mathit{red}(I,i) &\triangleq \mathit{rs}[I][i] \\ \mathit{end}(I) &\triangleq \forall\, i \in \mathit{It}(X[I]) : \mathit{red}(I,i) \end{aligned}$$

The set of all dependencies with respect to some assignment i and a pair d of look behind/ahead sets for input x is defined as:[8]

$$\begin{aligned} \mathit{deps}(x,d,i) \triangleq \quad & \{\, i-k \;:\; k \in \{\, k \in d[1] \;:\; i-k \geq \mathit{lBnd}(x) \wedge \mathit{prop}(i-k) \,\}\,\} \\ \cup\; & \{\, i \,\} \\ \cup\; & \{\, i+k \;:\; k \in \{\, k \in d[2] \;:\; i+k \leq \mathit{uBnd}(x) \wedge \mathit{prop}(i+k) \,\}\,\} \end{aligned}$$

For example, suppose $\mathit{Dep\_pc} = \langle\{1\}, \{1\}\rangle$, then at any assignment i of the instance indexed by I, $\mathit{deps}(X^I, \mathit{Dep\_pc}, i)$ is the set of all dependencies the consumer has with respect to the producer so that

$$\{i\} \subseteq \mathit{deps}(X^I, \mathit{Dep\_pc}, i) \subseteq \{i-1, i, i+1\}$$

This set always includes at least i itself.

Now we present the postulated assumptions about the symbol parameters in the specification. Assumption $\mathit{H\_Type}$ asserts the type assumptions for symbols concerning the base index, the iteration space and non-linear dependence pairs.

[7]We prefer in this chapter to use *It* instead of I to avoid confusion with indexes $I \in \mathit{Seq}(\mathit{Nat})$.

[8]Actually, *deps* could be considered a general definition if it where parameterized by the iteration space elements, e.g. $\mathit{deps}(\mathit{lBnd}(\_), \mathit{uBnd}(\_), \mathit{prop}(\_), x, d, i)$.

$$
\begin{array}{l}
\textsc{axiom}\ \mathit{H\_Type}\ \triangleq \\
\quad \land\ I0 \in \mathit{Index} \\
\quad \land\ \forall x \in T : \mathit{lBnd}(x) \in \mathit{Nat} \\
\quad \land\ \forall x \in T : \mathit{uBnd}(x) \in \mathit{Nat} \\
\quad \land\ \forall i \in \mathit{Nat} : \mathit{prop}(i) \in \text{BOOLEAN} \\
\quad \land\ \forall x \in T : \mathit{pre}(x) \in \text{BOOLEAN} \\
\quad \land\ \mathit{Dep\_pp} \in (\textsc{subset}\ (\mathit{Nat} \setminus \{0\})) \times (\textsc{subset}\ \{\}) \\
\quad \land\ \mathit{Dep\_pc} \in (\textsc{subset}\ (\mathit{Nat} \setminus \{0\})) \times (\textsc{subset}\ (\mathit{Nat} \setminus \{0\})) \\
\quad \land\ \mathit{Dep\_cr} \in (\textsc{subset}\ (\mathit{Nat} \setminus \{0\})) \times (\textsc{subset}\ (\mathit{Nat} \setminus \{0\}))
\end{array}
$$

Notice that *Dep_pp*, which represents non-linear dependencies of the producer with itself, can't have look ahead values (recall remark 3.2(1)).

Assumption *H_BFunType* asserts that, in general, the basic functions associated to the PCR components are expected to be partial functions.

$$
\begin{array}{l}
\textsc{axiom}\ \mathit{H\_BFunType}\ \triangleq \\
\quad \forall x \in T,\ i \in \mathit{Assig} : \\
\qquad \land\ \mathit{gp}(x,\ i) \in \mathit{Tp} \cup \{\mathit{Undef}\} \\
\qquad \land\ \forall \mathit{vp} \in \mathit{St}(\mathit{Tp}) : \mathit{fp}(x,\ \mathit{vp},\ i) \in \mathit{Tp} \cup \{\mathit{Undef}\} \\
\qquad \land\ \forall \mathit{vp} \in \mathit{St}(\mathit{Tp}) : \mathit{fc}(x,\ \mathit{vp},\ i) \in \mathit{Tc} \cup \{\mathit{Undef}\} \\
\qquad \land\ \forall \mathit{vc} \in \mathit{St}(\mathit{Tc}) : \mathit{fr}(x,\ \mathit{vc},\ i) \in D \cup \{\mathit{Undef}\}
\end{array}
$$

Assumption *H_BFunWD* asserts that the basic functions associated to the PCR components are well defined *at least* inside the iteration space *when* all their dependencies are meet (i.e. written).

$$
\begin{array}{l}
\textsc{axiom}\ \mathit{H\_BFunWD}\ \triangleq \\
\quad \forall x \in T : \forall i \in \mathit{It}(x) : \\
\qquad \land\ \mathit{gp}(x,\ i) \in \mathit{Tp} \\
\qquad \land\ \forall \mathit{vp} \in \mathit{St}(\mathit{Tp}) : \mathit{wrts}(\mathit{vp},\ \mathit{deps}(x,\ \mathit{Dep\_pp},\ i) \setminus \{i\}) \Rightarrow \mathit{fp}(x,\ \mathit{vp},\ i) \in \mathit{Tp} \\
\qquad \land\ \forall \mathit{vp} \in \mathit{St}(\mathit{Tp}) : \mathit{wrts}(\mathit{vp},\ \mathit{deps}(x,\ \mathit{Dep\_pc},\ i)) \Rightarrow \mathit{fc}(x,\ \mathit{vp},\ i) \in \mathit{Tc} \\
\qquad \land\ \forall \mathit{vc} \in \mathit{St}(\mathit{Tc}) : \mathit{wrts}(\mathit{vc},\ \mathit{deps}(x,\ \mathit{Dep\_cr},\ i)) \Rightarrow \mathit{fr}(x,\ \mathit{vc},\ i) \in D
\end{array}
$$

Notice that for the producer function *fp*, assignment i is discarded from the dependence set in the assumption, as to compute the i-th value it can't depended on itself at i.

We call the following axioms the *relevance axioms*.[9] Their job is to assert that the basic functions associated to the PCR components are insensitive with respect to what they do not depend on. For example, if $vp1$ and $vp2$ are producer streams *equal* with respect to the set of dependencies the consumer has on the producer (see the *eqs* definition), then $fc(x, vp1, i) = fc(x, vp2, i)$ for any input x and assignment i. These axioms are used in deductive proofs of equalities by substitution of equals for equals.

$$\begin{array}{l}
\textsc{axiom}\ \mathit{H\_fpRelevance}\ \triangleq \\
\quad \forall\, x \in T : \forall\, i \in \mathit{It}(x),\ vp1 \in \mathit{St}(Tp),\ vp2 \in \mathit{St}(Tp) : \\
\qquad \mathit{eqs}(vp1,\ vp2,\ \mathit{deps}(x,\ \mathit{Dep\_pp},\ i) \setminus \{i\})\ \Rightarrow\ \mathit{fp}(x,\ vp1,\ i) = \mathit{fp}(x,\ vp2,\ i) \\
\textsc{axiom}\ \mathit{H\_fcRelevance}\ \triangleq \\
\quad \forall\, x \in T : \forall\, i \in \mathit{It}(x),\ vp1 \in \mathit{St}(Tp),\ vp2 \in \mathit{St}(Tp) : \\
\qquad \mathit{eqs}(vp1,\ vp2,\ \mathit{deps}(x,\ \mathit{Dep\_pc},\ i))\ \Rightarrow\ \mathit{fc}(x,\ vp1,\ i) = \mathit{fc}(x,\ vp2,\ i) \\
\textsc{axiom}\ \mathit{H\_frRelevance}\ \triangleq \\
\quad \forall\, x \in T : \forall\, i \in \mathit{It}(x),\ vc1 \in \mathit{St}(Tc),\ vc2 \in \mathit{St}(Tc) : \\
\qquad \mathit{eqs}(vc1,\ vc2,\ \mathit{deps}(x,\ \mathit{Dep\_cr},\ i))\ \Rightarrow\ \mathit{fr}(x,\ vc1,\ i) = \mathit{fr}(x,\ vc2,\ i)
\end{array}$$

Assumption *H_ProdEqInv* is stated as a LEMMA left unproved because it is actually an invariant assumption concerning a temporal variable (i.e. not a constant formula). It asserts the functional equivalence of the producer functions *fp* and *gp* (recall remark 3.4).

$$\begin{array}{l}
\textsc{lemma}\ \mathit{H\_ProdEqInv}\ \triangleq \\
\quad \forall\, x \in T : \forall\, i \in \mathit{It}(x) : \mathit{wrt}(p[I0][i])\ \Rightarrow\ \mathit{fp}(x,\ p[I0],\ i) = \mathit{gp}(x,\ i)
\end{array}$$

Now, it should be added that all these assumptions (except *H_BFunType*) could be proved as theorems when dealing with concrete PCRs. Most of them can also be re-stated as invariants of the specification relative to base index I_0 and model checked at least for small bounds. For example, we did prove them for PCR FibPrimes1.[10] Regarding *H_BFunType*, there is no way to tell what happens in a function outside the relevant iteration space, and the point is that we do not want to know. However, we do need the information of partiality conveyed in the axiom when comparing stream functions in proofs.

[9]They resemble the usual relevance lemmas on formal languages stating the evaluation of the formulas depend only on the values of the variables present in the formula.

[10]On a possible practical setting, we believe most of them don't deserve to be proved by any means. For an implementation in a programming language, that would be in part the job of type checking.

5.4.1.4 The functional specification

The functional specification provides a formula characterizing the mathematical function associated to the basic PCR. This formula is used in the Properties section of the specification to assert correctness, and is also used to represent the PCR as one step computation in module PCR_A1step to prove correctness by way of refinement.

Module *AbelianMonoidBigOp*, explained previously in section 5.3, is instantiated with the signature $(D, id, Op(\_,\_))$. Assumption *H_Algebra* asserts the signature satisfies the laws of an abelian monoid, which gives us access to the corresponding theorems for theorem proving.

$$\begin{array}{l} M \triangleq \text{INSTANCE } AbelianMonoidBigOp \\ \quad \text{WITH } D \leftarrow D,\ Id \leftarrow id,\ \otimes \leftarrow Op \end{array}$$

$$\text{AXIOM } H\_Algebra \triangleq AbelianMonoid(D,\ id,\ Op)$$

Again, it should be added that the assumption *H_Algebra* could be (and should be) proved as a theorem when dealing with concrete PCRs.

The stream versions of the basic functions, noted as $\overrightarrow{g}_p$, $\overrightarrow{f}_c$ and $\overrightarrow{f}_r$ in chapter 3, are defined as follows:

$$\begin{array}{rcl} Gp(x) & \triangleq & [i \in Assig \mapsto gp(x, i)] \\ Fc(x, vp) & \triangleq & [i \in Assig \mapsto fc(x, vp, i)] \\ Fr(x, vc) & \triangleq & [i \in Assig \mapsto fr(x, vc, i)] \end{array}$$

Because the basic functions are partial, we have $Gp \in St(Tp)$, $Fc \in St(Tc)$ and $Fr \in St(D)$. The formalization of the function 3.9 associated to the PCR $\mathcal{A}$ is:

$$A(x) \triangleq M!BigOpP(lBnd(x),\ uBnd(x),\ prop,\ \text{LAMBDA } i : Fr(x, Fc(x, Gp(x)))[i])$$

5.4.1.5 The operational specification

The operational specification provides a temporal formula in the canonical form 4.10 specifying the operational behaviour of the basic PCR.

A main baseline unfair specification, named *Spec* as customary, and its fair version are defined according to the canonical form:

$$\begin{aligned} \mathit{Spec} &\triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\langle \mathit{in},\mathit{vs}\rangle} \\ \mathit{FairSpec} &\triangleq \mathit{Spec} \land \mathit{WF}_{\mathit{vs}}(\mathit{Step}) \end{aligned}$$

where $\mathit{vs} = \langle X, p, c, r, \mathit{rs}\rangle$. The initial conditions are established by the *Init* state predicate as follows:

$$\begin{aligned} \mathit{Init} \triangleq\ &\land\ \mathit{in} \in T \land \mathit{pre}(\mathit{in}) \\ &\land\ X = [I \in \mathit{Index} \mapsto \text{IF } I = I0 \text{ THEN } \mathit{in} \text{ ELSE } \mathit{Undef}] \\ &\land\ p\ = [I \in \mathit{Index} \mapsto [i \in \mathit{Assig} \mapsto \mathit{Undef}]] \\ &\land\ c\ = [I \in \mathit{Index} \mapsto [i \in \mathit{Assig} \mapsto \mathit{Undef}]] \\ &\land\ \mathit{rs} = [I \in \mathit{Index} \mapsto [i \in \mathit{Assig} \mapsto \text{FALSE}]] \\ &\land\ r\ = [I \in \mathit{Index} \mapsto \mathit{id}] \end{aligned}$$

Notice that there is a single well defined input with base index I_0 (recall assumption 5.2.4), having value $\mathit{in} \in T$ which satisfies a possible precondition *pre*. The reducer output is initially by default the monoid identity (recall assumption 5.2.2). Producer and consumer outputs are initially undefined, and the reduction history is initially entirely false.

The *Next* action governing the evolution of the state is the disjunction of two possible and mutually exclusive sub actions:

$$\mathit{Next} \triangleq \mathit{Step} \lor \mathit{Done}$$

Sub action *Done* detects termination when all well defined instances of the PCR have terminated:[11]

$$\begin{aligned} \mathit{Done} \triangleq\ &\land\ \forall I \in \mathit{WDIndex} : \mathit{end}(I) \\ &\land\ \text{UNCHANGED } \langle \mathit{in}, \mathit{vs}\rangle \end{aligned}$$

The sub action *Step* represents any possible operation of the basic PCR components. For this, operations are modeled by atomic actions, namely P for the producer, C for the consumer and R for the reducer, which are parameterized by the instance index I and the current assignment i. So, *Step* is defined as an (double) existential quantifier which introduces non determinism to model the concurrent execution of possible multiple

[11]For the main PCR being specified, this just means termination at index I_0.

instances of the basic PCR and their component actions:[12]

$$
\begin{array}{lll}
Step \triangleq & \exists I \in WDIndex : & \\
& \exists i \in It(X[I]) : \lor P(I, i) & \text{Producer action} \\
& \qquad\qquad\qquad \lor C(I, i) & \text{Consumer action} \\
& \qquad\qquad\qquad \lor R(I, i) & \text{Reducer action}
\end{array}
$$

Table 5.3 presents the PCR operations as syntax primitives and their corresponding formalization as TLA$^+$ actions. Note that all these actions have in common that they are enabled if:

1. They haven't already happened. For example, producer action $P(I, i)$ will be enabled until $wrt(p^I, i)$ holds, after that it will never be enabled again.

2. Their dependencies with respect to a previous component are meet. For example, producer action $P(I, i)$ is enabled if his past dependencies prescribed by Dep_pp (i.e. $deps(X^I, Dep\_pp, i) \setminus \{i\}$) have been written.

PCR Syntax	**TLA$^+$ action**
p = **`produce`** f_p x p	$P(I, i) \triangleq$ $\land\ \neg wrt(p[I][i])$ $\land\ wrts(p[I], deps(X[I], Dep\_pp, i) \setminus \{i\})$ $\land\ p' = [p \text{ EXCEPT } ![I][i] = fp(X[I], p[I], i)]$ $\land$ UNCHANGED $\langle X, c, r, rs \rangle$
c = **`consume`** f_c x p	$C(I, i) \triangleq$ $\land\ \neg wrt(c[I][i])$ $\land\ wrts(p[I], deps(X[I], Dep\_pc, i))$ $\land\ c' = [c \text{ EXCEPT } ![I][i] = fc(X[I], p[I], i)]$ $\land$ UNCHANGED $\langle X, p, r, rs \rangle$
r = **`reduce`** $\otimes$ `id`$_\otimes$ (f_r x c)	$R(I, i) \triangleq$ $\land\ \neg red(I, i)$ $\land\ wrts(c[I], deps(X[I], Dep\_cr, i))$ $\land\ r' = [r \text{ EXCEPT } ![I] = Op(@, fr(X[I], c[I], i))]$ $\land\ rs' = [rs \text{ EXCEPT } ![I][i] = \text{TRUE}]$ $\land$ UNCHANGED $\langle X, p, c \rangle$

Table 5.3: TLA$^+$ specification for the principal PCR primitives: *produce*, *consume* and *reduce*, over basic function names.

[12] We are omitting here that variable *in* is keep unchanged.

For any of the PCR actions, *if* it is enabled *and* is chosen for execution, its effect will be reflected in the corresponding output variable. In particular, an action $R(I, i)$ writes on reducer output variable r but also marks the assignment as reduced in rs. Note that more than one action cannot occur simultaneously because:

1. The EXCEPT construct as is being used here asserts exactly only one instance I and one assignment i is being modified in the transition. So, for example, producers on different PCR instances cannot change variable p at the same step, nor can different assignments on the same instance.

2. The UNCHANGED assertion precludes the variables of the other components to change. So, a producer action cannot occur in the same step with a consumer (or reducer) action.

This means the execution of all actions is interleaved in accordance with our assumption 5.2.1.

Finally, we wish to justify why asserting weak fairness (WF) over the *Step* action, i.e. $\mathrm{WF}_{vs}(\mathit{Step})$, is enough for the fair formula *FairSpec* to guarantee progress and eventual termination of the PCR computation. Fairness conditions (and liveness in general) are a subtle point in system specification. For example, why do we need to assert WF for the three kind of actions and not just one or two of them?. We make the following observations:

- Initially, assuming the iteration space is not empty, at least one producer action should be enabled and $\mathrm{WF}_{vs}(\mathit{Step})$ will make sure this producer action will be eventually chosen. Recall actions are no longer enabled after their execution, so the *Step* disjunction isn't necessarily enabled by the same action at each time. Subsequent producer actions will be made enabled following a domino-like effect as their dependencies are meet. Those could also enable consumer actions, but even if there is producer bias and no consumer action is chosen, eventually all producer actions on the iteration space will occur and at some moment there will be no other option except to choose consumer actions. Similarly, when all consumer actions had been already executed, there will be no other option except to choose reducer actions.

Therefore, eventually *Step* is not true anymore and *Done* is true.

- It should be clear by the previous point, why it is not enough to assert WF only on one or two of the kind of actions. For example, if we assert WF only on producer actions, then there is no guarantee for the occurrence of consumer actions and this in turn have the same consequence for reducer actions. Or, if we assert WF only for consumer actions, then there is no guarantee for the occurrence of producer actions and because the consumer actions depends on the producer actions they will never be permanently enabled to be eligible for execution according to WF.

The complete TLA$^+$ specification is in module PCR_A (Appendix B).

5.4.2 Basic PCR with left reducer

A basic PCR with a left reducer is exactly a basic PCR except its reducer fixes the reduction order as was explained in section 3.2.1. However, for simplicity, we assume here a consecutive iteration space. This can be seen as either there is no filter predicate *prop* or it is trivially TRUE. For the specification we prefer the former:

$$It(x) \triangleq lBnd(x)..uBnd(x)$$

In PCR syntax, the following dependence is imposed on the reducer variable:

$$\textbf{dep} \quad r(i-1) \rightarrow r(i)$$

Table 5.4 presents the left reducer formalization as a TLA$^+$ action. It is just as the ordinary reducer except it demands another enabling condition concerning the occurrence of the previous reduce operation. The condition relies on the history of reductions.

The complete TLA$^+$ specification is in module PCR_ArLeft (Appendix B).

PCR Syntax	**TLA$^+$ action**

`dep` `r(i-1)` $\rightarrow$ `r(i)`
`...`
r = **`reduce`** $\otimes$ $\mathtt{id}_{\otimes}$ $(f_r\ x\ c)$
	$R(I, i) \triangleq$
	$\land \neg red(I, i)$
	$\land wrts(c[I], deps(X[I], Dep\_cr, i))$
	$\land i - 1 \geq lBnd(X[I]) \Rightarrow red(I, i - 1)$
	$\land r' = [r \text{ EXCEPT } ![I] = Op(@, fr(X[I], c[I], i))]$
	$\land rs' = [rs \text{ EXCEPT } ![I][i] = \text{TRUE}]$
	$\land \text{UNCHANGED } \langle X, p, c \rangle$

Table 5.4: TLA$^+$ specification for a fixed order (left) reduction over a basic function name.

5.4.3 Basic PCR as a one step computation

The result of basic PCR computations, as described by the operational specification, is captured as a mathematical function defined in the functional specification. Thus, the computation of the basic PCR can be represented in just one step.

Two variables *in* and *out* are enough to model input and output respectively. Then, the initial predicate and the next-state action are defined as follows:

$$\begin{aligned} Init &\triangleq in \in T \land pre(in) \land out = id \\ Next &\triangleq out' = A(in) \land \text{UNCHANGED } in \end{aligned}$$

This is a short enough specification to be written as a one liner. It can be used as an alternative way to prove correctness of the basic PCR. Instead of proving the operational specification of the basic PCR satisfies a correctness property on formula A, we can prove it is a refinement of this one step specification. However, the proof in both ways involves essentially the very same ideas and amount of work.

The complete TLA$^+$ specification is in module PCR_A1step (Appendix B).

5.4.4 Composition through consumer

The main reference for this formalization is the definition of composition through consumer given in 3.4.

Here we have two PCRs in one specification: the main PCR $\mathcal{A}$ and the nested basic PCR $\mathcal{B}$. There is a set of constants and variables symbols for each one, and also there are basic definitions associated with each one. To distinguish them, we suffix variables and constants names with numbers 1 and 2 for $\mathcal{A}$ and $\mathcal{B}$ respectively (much like in 3.4), and suffix definitions with the corresponding PCR name (e.g. we have *IndexA* and *IndexB*).

The operational and functional specification of $\mathcal{B}$ is essentially the same as was explained previously for a basic PCR, so what we call $\mathcal{B}$ in module *PCR_A_c_B* is what we call $\mathcal{A}$ in module *PCR_A*. The only important thing to note about $\mathcal{B}$ here is that its input is controlled by $\mathcal{A}$ and there can be multiple instances of $\mathcal{B}$. Inputs for $\mathcal{B}$ are triplets of type

$$T2 \triangleq T1 \times StA(Tp1) \times AssigA$$

Initially, there is no input defined for $\mathcal{B}$ and consequently there is no initial instance of $\mathcal{B}$. In a later section, where we discuss a refinement proof, we will have a little more to say about the relevance axioms that are actually needed for $\mathcal{B}$.

Module *AbelianMonoidBigOp* is instantiated twice, namely $M1$ and $M2$, with the signatures $(D1, id1, Op1(\_,\_))$ and $(D2, id2, Op2(\_,\_))$ for $\mathcal{A}$ and $\mathcal{B}$ respectively. The algebraic assumptions of an abelian monoid are asserted for both. Consequently there is an associated function defined for each PCR:[13]

$$\begin{aligned} B(x2) \;&\triangleq\; M2!BigOpP(lBnd2(x2),\, uBnd2(x2),\, prop2,\\ &\qquad\qquad \text{LAMBDA } i : Fr2(x2, Fc2(x2, Gp2(x2)))[i])\\ A(x1) \;&\triangleq\; M1!BigOpP(lBnd1(x1),\, uBnd1(x1),\, prop1,\\ &\qquad\qquad \text{LAMBDA } i : Fr1(x1, Fc1(x1, Gp1(x1)))[i]) \end{aligned}$$

where in particular we have

$$Fc1(x1, vp) = [i \in AssigA \;\mapsto\; B(\langle x1, vp, i\rangle)]$$

so that the function of $\mathcal{A}$ is composed with the function of $\mathcal{B}$. This formalizes the function in equation 3.12.

[13] Alternatively, as we know here that B is used as a function on three parameters, we could define $B(x1, vp, i) \triangleq$ ..., but it get messier and doesn't worthwhile. Sometimes we may write $B(x1, vp, i)$ informally.

The operational specification of $\mathcal{A}$ differs from a basic PCR only in what concerns the consumer component, as his values are computed by the nested PCR $\mathcal{B}$ and not by an atomic basic function. Table 5.5 presents the formalization of the consumer syntax over a PCR name as a pair of TLA$^+$ actions that can be thought as a call and a return with respect to $\mathcal{B}$:

1. $C1ini(I, i)$: writes on $\mathcal{B}$'s input variable $X_2^{I,i}$ *if* have not done yet and the dependencies on the producer are meet.

2. $C1end(I, i)$: reads from $\mathcal{B}$'s output variable $r_2^{I,i}$ *if* have not done yet, there is an input written for $\mathcal{B}$ and it has finished calculating on that entry.

PCR Syntax	**TLA$^+$ actions**
$c_1 \;=\; \texttt{consume}\;\; \mathcal{B}\;\; x_1\;\; p_1$	$C1ini(I,\, i) \triangleq$ $\wedge\; \neg wrt(X2[I \circ \langle i \rangle])$ $\wedge\; wrts(p1[I],\, depsA(X1[I],\, Dep\_pc1,\, i))$ $\wedge\; X2' = [X2 \text{ EXCEPT } ![I \circ \langle i \rangle] = \langle X1[I],\, p1[I],\, i \rangle]$ $\wedge\; \text{UNCHANGED } \langle X1,\, p1,\, c1,\, r1,\, rs1 \rangle$ $C1end(I,\, i) \triangleq$ $\wedge\; \neg wrt(c1[I][i])$ $\wedge\; wrt(X2[I \circ \langle i \rangle])$ $\wedge\; endB(I \circ \langle i \rangle)$ $\wedge\; c1' = [c1 \text{ EXCEPT } ![I][i] = r2[I \circ \langle i \rangle]]$ $\wedge\; \text{UNCHANGED } \langle X1,\, p1,\, r1,\, rs1,\, X2 \rangle$

Table 5.5: TLA$^+$ specification for the *consume* primitive over a PCR named $\mathcal{B}$.

In general, we could have two (fair) specifications for $\mathcal{A}$ and $\mathcal{B}$ defined as:

$$FairSpecA \triangleq InitA \wedge \Box[StepA \vee DoneA]_{\langle in, vs1 \rangle} \wedge WF_{vs1}(StepA)$$
$$FairSpecB \triangleq InitB \wedge \Box[StepB \vee DoneB]_{vs2} \wedge WF_{vs2}(StepB)$$

where $vs1 = \langle X1, p1, c1, r1, rs1, X2 \rangle$ and $vs2 = \langle p2, c2, r2, rs2 \rangle$, so that their conjoint specification is their conjunction taking interleaving into account (recall assumption 5.2.1):

$$FairSpec \triangleq FairSpecA \wedge FairSpecB \wedge \Box[vs1' = vs1 \vee vs2' = vs2]_{\langle vs1, vs2 \rangle}$$

As we already noted, the main PCR $\mathcal{A}$ is not basic, so its *StepA* action must be accommodated to the consumer specification explained previously as follows:

$$\begin{array}{lll} \mathit{StepA} & \triangleq & \exists\, I \in \mathit{WDIndexA} : \\ & & \quad \exists\, i \in \mathit{ItA}(X1[I]) : \lor P1(I,\ i) \qquad \text{Producer action} \\ & & \qquad\qquad\qquad\qquad\quad \lor C1ini(I,\ i) \qquad \text{Consumer call action on B} \\ & & \qquad\qquad\qquad\qquad\quad \lor C1end(I,\ i) \qquad \text{Consumer return action from B} \\ & & \qquad\qquad\qquad\qquad\quad \lor R1(I,\ i) \qquad \text{Reducer action} \end{array}$$

But, in order to work with TLC, we transform the conjoint specification to a single canonical formula using logical equivalences as explained in chapter 4 which produces a single system formula that can be arranged as:

$$\begin{array}{rcl} \mathit{Spec} & \triangleq & \mathit{Init} \land \Box[\mathit{Next}]_{\langle in, vs1, vs2 \rangle} \\ \mathit{FairSpec} & \triangleq & \mathit{Spec} \land \mathit{WF}_{vs1}(\mathit{StepA}) \land \mathit{WF}_{vs2}(\mathit{StepB}) \end{array}$$

where

$$\begin{array}{rcl} \mathit{Init} & = & \mathit{InitA} \land \mathit{InitB} \\ \mathit{Next} & = & \lor\ \mathit{StepA} \quad \land\ \langle in, vs2 \rangle' = \langle in, vs2 \rangle \\ & & \lor\ \mathit{StepB} \quad \land\ \langle in, vs1 \rangle' = \langle in, vs1 \rangle \\ & & \lor\ \mathit{DoneA} \land \mathit{DoneB} \end{array}$$

We add that, in general, for two actions A_1 and A_2:

$$\mathrm{WF}_{vs_1}(A_1) \land \mathrm{WF}_{vs_2}(A_2) \ \not\equiv\ \mathrm{WF}_{\langle vs_1, vs_2 \rangle}(A_1 \lor A_2)$$

This looks like a desirable equivalence to have, because when composing two specifications the fairness conditions of both could be made into one. But it does not hold in general, it depends on the system at hand. In our case, $\mathit{WF}_{\langle vs1, vs2 \rangle}(\mathit{StepA} \lor \mathit{StepB})$ have the same effect that their separate conjunction.

The complete TLA$^+$ specification is in module PCR_A_c_B (Appendix B).

5.4.5 Composition through reducer

The main reference for this formalization is the definition of composition through reducer given in 3.5.

The situation here is very similar to the previous section, except that the composition happens at the reducer component of the main PCR $\mathcal{A}$ and the inputs of $\mathcal{B}$ are pairs of type

$$T2 \triangleq D \times D$$

Module *AbelianMonoidBigOp* is instantiated twice with the signatures $(D, id, Op1(\_,\_))$ and $(D, id, Op2(\_,\_))$. Again, the algebraic assumptions of an abelian monoid are asserted for both and there is an associated function defined for each PCR. In particular, here we have that $Op1(x, y) = B(\langle x, y\rangle)$. This formalizes the function given in equation 3.21.

Table 5.6 presents the formalization of the reducer syntax over a PCR name as a pair of TLA$^+$ actions that can be thought as a call and a return with respect to $\mathcal{B}$:

1. $R1ini(I, i)$: writes on $\mathcal{B}$'s input variable $X_2^{I,i}$ *if* this has not been done yet and the dependencies on the producer are met. It also uses the lock mechanism explained in 3.5 to avoid race conditions on the reducer variable.

2. $R1end(I, i)$: reads from $\mathcal{B}$'s output variable $r_2^{I,i}$ and marks the assignment as reduced *if* this has not been done yet, there is an input written for $\mathcal{B}$ and it has finished calculating on that entry.

The *StepA* action is defined as follows:

$$\begin{array}{lll} StepA \triangleq \exists\, I \in WDIndexA : & & \\ \quad \exists\, i \in ItA(X1[I]) : & \lor\ P1(I,\ i) & \text{Producer action} \\ & \lor\ C1(I,\ i) & \text{Consumer action} \\ & \lor\ R1ini(I,\ i) & \text{Reducer call action on B} \\ & \lor\ R1end(I,\ i) & \text{Reducer return action from B} \end{array}$$

Further treatment is just as we did before for composition through consumer. The complete TLA$^+$ specification is in module PCR_A_r_B (Appendix B).

PCR Syntax	**TLA$^+$ actions**
$r_1 \;=\; \texttt{reduce}\; \mathcal{B}\; \texttt{id}_{\otimes}\; (f_{r_1}\;\; x_1\;\; c_1)$	$R1ini(I,\, i) \triangleq$ $\quad \land\; \neg wrt(X2[I \circ \langle i \rangle])$ $\quad \land\; wrts(c1[I],\, depsA(X1[I],\, Dep\_cr1,\, i))$ $\quad \land\; \neg \exists\, k \in ItA(X1[I]) :\; \land\, k \neq i$ $\qquad\qquad\qquad\qquad\qquad \land\, wrt(X2[I \circ \langle k \rangle])$ $\qquad\qquad\qquad\qquad\qquad \land\, \neg redA(I,\, k)$ $\quad \land\; X2' = [X2 \text{ EXCEPT } ![I \circ \langle i \rangle] =$ $\qquad\qquad\qquad \langle r1[I],\, fr1(X1[I],\, c1[I],\, i)\rangle]$ $\quad \land\; \text{UNCHANGED } \langle X1,\, p1,\, c1,\, r1,\, rs1 \rangle$ $R1end(I,\, i) \triangleq$ $\quad \land\; wrt(X2[I \circ \langle i \rangle])$ $\quad \land\; endB(I \circ \langle i \rangle)$ $\quad \land\; \neg redA(I,\, i)$ $\quad \land\; r1' \;= [r1 \;\text{ EXCEPT } ![I] \;\;\;\; = r2[I \circ \langle i \rangle]]$ $\quad \land\; rs1' = [rs1 \text{ EXCEPT } ![I][i] = \text{TRUE}]$ $\quad \land\; \text{UNCHANGED } \langle X1,\, p1,\, c1,\, X2 \rangle$

Table 5.6: TLA$^+$ specification for the *reduce* primitive over a PCR named $\mathcal{B}$.

5.4.6 Divide and conquer (DC)

As we explained in section 3.3.2, the divide and conquer PCR is a special case of composition through consumer with itself, and this composition can actually occur an arbitrary number of times at execution time depending on the *isBase* condition. In particular, the iteration space is defined in a more specific form:

$$\begin{aligned} uBnd(x) &\triangleq Len(div(x)) \\ It(x) &\triangleq 1..uBnd(x) \end{aligned}$$

where $div(\_)$ is an operator representing the partitioning function. Besides, we have the operators symbols $isBase(\_,\_,\_)$, $base(\_,\_,\_)$ and $fr(\_,\_,\_)$, where *isBase* and *base* represents the basic functions involved in the recursive *subproblem* function associated to the consumer in the DC scheme.

For the functional specification, module *AbelianMonoidBigOp* is instantiated with the signature $(D, id, Op(\_,\_))$. The formalization of the function 3.3 associated to the PCR

$\mathcal{DC}$ can be expressed as the recursive operator:

$$\begin{array}{l} DC(x) \triangleq M!BigOp(1,\ uBnd(x), \\ \qquad \textsc{lambda}\ i : Fr(x, [k \in Assig \mapsto \textsc{if}\ isBase(x, div(x), k) \\ \qquad\qquad \textsc{then}\ base(x, div(x), k) \\ \qquad\qquad \textsc{else}\ DC(div(x)[k])])[i]) \end{array}$$

or as the recursive function:

$$\begin{array}{l} DC[x \in T] \triangleq M!BigOp(1,\ uBnd(x), \\ \qquad \textsc{lambda}\ i : Fr(x, [k \in Assig \mapsto \textsc{if}\ isBase(x, div(x), k) \\ \qquad\qquad \textsc{then}\ base(x, div(x), k) \\ \qquad\qquad \textsc{else}\ DC[div(x)[k]]])[i]) \end{array}$$

Both can be handled by TLC, but only the second can be handled by TLAPS. The inner anonymous TLA$^+$ function is the stream function noted as $\overrightarrow{subproblem}$ in 3.3.

For the operational specification, the consumer component differs from the ordinary consumer composition because here the consumer is writing and reading on the same PCR variables (although in different indexes) and we need to take into account when the *isBase* condition. However, the overall idea is pretty much the same. Table 5.7 presents the formalization of the consumer syntax in a DC PCR in terms of three TLA$^+$ actions:

1. $Cbase(I, i)$: takes the value from the *base* function *if* haven't done yet, the dependencies on the producer are met and condition *isBase* does hold.

2. $Cini(I, i)$: writes on input variable at $I \circ \langle i \rangle$ creating a new instance of the same PCR *if* haven't done yet, the dependencies on the producer are meet and condition *isBase* does not hold.

3. $Cend(I, i)$: reads from output variable at $I \circ \langle i \rangle$ *if* haven't done yet, there is an input written at $I \circ \langle i \rangle$ and the corresponding instance has finished calculating on that entry.

The *Step* action of the divide and conquer PCR is defined as follows:

PCR Syntax	**TLA$^+$ actions**
	$\begin{array}{l} \mathit{Cbase}(I,\ i)\ \triangleq \\ \quad \land\ \neg \mathit{wrt}(c[I][i]) \\ \quad \land\ \mathit{wrts}(p[I],\ \mathit{deps}(X[I],\ \mathit{Dep\_pc},\ i)) \\ \quad \land\ \mathit{isBase}(X[I],\ p[I],\ i) \\ \quad \land\ c' = [c \text{ EXCEPT } ![I][i] = \mathit{base}(X[I],\ p[I],\ i)] \\ \quad \land\ \text{UNCHANGED } \langle X,\ p,\ r,\ rs\rangle \end{array}$
`fun subproblem(x,p,i) = if isBase(x,p,i) then base(x,p,i) else DC(p) ... c = consume subproblem x p`	$\begin{array}{l} \mathit{Cini}(I,\ i)\ \triangleq \\ \quad \land\ \neg \mathit{wrt}(X[I \circ \langle i\rangle]) \\ \quad \land\ \mathit{wrts}(p[I],\ \mathit{deps}(X[I],\ \mathit{Dep\_pc},\ i)) \\ \quad \land\ \neg \mathit{isBase}(X[I],\ p[I],\ i) \\ \quad \land\ X' = [X \text{ EXCEPT } ![I \circ \langle i\rangle] = p[I][i]] \\ \quad \land\ \text{UNCHANGED } \langle p,\ c,\ r,\ rs\rangle \end{array}$
	$\begin{array}{l} \mathit{Cend}(I,\ i)\ \triangleq \\ \quad \land\ \neg \mathit{wrt}(c[I][i]) \\ \quad \land\ \mathit{wrt}(X[I \circ \langle i\rangle]) \\ \quad \land\ \mathit{end}(I \circ \langle i\rangle) \\ \quad \land\ c' = [c \text{ EXCEPT } ![I][i] = r[I \circ \langle i\rangle]] \\ \quad \land\ \text{UNCHANGED } \langle X,\ p,\ r,\ rs\rangle \end{array}$

Table 5.7: TLA$^+$ specification for the *consume* primitive in a divide and conquer PCR named $\mathcal{DC}$.

$$\begin{array}{lll} \mathit{Step}\ \triangleq\ \exists\, I \in \mathit{WDIndex} : & & \\ \quad \exists\, i \in \mathit{It}(X[I]) : & \lor\ P(I,\ i) & \text{Producer action} \\ & \lor\ \mathit{Cbase}(I,\ i) & \text{Consumer base case action} \\ & \lor\ \mathit{Cini}(I,\ i) & \text{Consumer recursive call action} \\ & \lor\ \mathit{Cend}(I,\ i) & \text{Consumer recursive return action} \\ & \lor\ R(I,\ i) & \text{Reducer action} \end{array}$$

The complete TLA$^+$ specification is in module PCR_DC (Appendix B).

5.4.7 DC with left reducer (DCrLeft)

A DC PCR with a left reducer is a combination of what we did previously for the ordinary DC PCR and the basic PCR with left reduction. There is nothing important to note. The complete TLA$^+$ specification is in module PCR_DCrLeft (Appendix B).

5.4.8 DC composed through reducer with a DCrLeft

A DC PCR composed through reducer with a DCrLeft PCR is a combination of what we did previously for the ordinary DC PCR, the basic PCR with left reduction and composition through reducer.

One important thing to note here is that, according to our discussion in section 3.3.3.2, the indexing scheme needs to be slightly generalized to accommodate correctly the concurrent execution of the nested DC PCRs. Indexes for the inner DCrLeft PCR $\mathcal{B}$ are defined as pairs of sequences:

$$\mathit{IndexB} \triangleq \mathit{Seq}(\mathit{Nat}) \times \mathit{Seq}(\mathit{Nat})$$

This means that, in general, when manipulating $\mathcal{B}$ variables the indexing is differs from our previous specifications. For example, we need to write $X2[\langle I, \langle i \rangle \rangle]$ (informally $X_2^{I;i}$) instead of $X2[I \circ \langle i \rangle]$ (informally $X_2^{I,i}/X_2^{I \circ \langle i \rangle}$). But apart from this, there is nothing new.

The complete TLA$^+$ specification is in module PCR_DC_r_DCrLeft (Appendix B).

5.4.9 Iteration over basic function

The main reference for this formalization is the definition of the iterative PCR scheme given in 3.6, in case that the iterable function is a basic function.

Some important constant symbols in this specification are the following:

1. $v0(\_)$: operator representing the initial value of iteration that possibly depends on PCR input.

2. $fc(\_,\_,\_,\_)$: operator representing the iterable basic function. So, $fc(y, x, p, i)$ computes the next value of the iteration which should be of type Tc, where $y \in Tc$ is the previous value.

3. $cnd(\_,\_)$: operator representing the termination condition. So, $cnd(s, k)$ is a truth

condition on iteration sequence s and the iteration number k.

The auxiliary variable s is used to track iterations at any assignment and instance. Iterations are also streams in our sense and starting from index 1, but they evolve incrementally in a well behaved fashion so its convenient to model them as ordinary TLA$^+$ sequences for which a new value is appended at each iteration.[14] Thus, s is a function mapping indexes to streams of sequences.

$$s \in [Seq(Nat) \to [Nat \to Seq(Tc) \cup \{Undef\}]]$$

Like other internal variables representing functions, s is curried for convenience. So, $s[I]$ denotes all the iteration sequences at index I, $s[I][i]$ denotes the i-th iteration sequence at index I, $s[I][i][k]$ denotes the value at iteration k on the i-th iteration sequence of index I, and more specifically $last(s[I][i])$ denotes the last value on the i-th iteration sequence of index I where $last(S) = S[Len(S)]$.

The functional specification, formalizing the function at equation 3.27, is exactly like the one for a basic PCR except that we have the stream function for the consumer defined as

$$Fc(x, vp) \triangleq [i \in Assig \mapsto last(iter(\langle v0(x)\rangle, x, vp, i))]$$

where $iter$ can be expressed as the recursive operator:

$$\begin{aligned} iter(vs, x, vp, i) \triangleq\ & \text{IF } cnd(vs, Len(vs)) \\ & \text{THEN } vs \\ & \text{ELSE } iter(vs \circ \langle fc(last(vs), x, vp, i)\rangle, x, vp, i) \end{aligned}$$

or alternatively as a recursive function like we did for the DC PCR.

For the operational specification, initially all the iteration sequences are undefined, that is the condition

$$s = [I \in Index \mapsto [i \in Assig \mapsto Undef]]$$

Table 5.8 presents the formalization of the iterative consumer syntax over basic functions in terms of three TLA$^+$ actions:

[14]However, it should be noted this sequences are finite, so they aren't faithfully modeling the possible unbounded iterations as real streams do.

1. $Cstart(I, i)$: initializes an iteration sequence using v_0 *if* haven't done yet and the dependencies on the producer are meet.

2. $Cstep(I, i)$: expands an iteration sequence computing f_c on the previous value (and other elements) *if* a sequence is defined and the condition *cnd* does not hold.

3. $Cend(I, i)$: reads the last value of an iteration sequence *if* haven't done yet, a sequence is defined and condition *cnd* does hold.

One interesting difference between action $Cstep(I, i)$ and the other actions is that this action can occur more than once on the same I and i, because each occurrence represents the expansion of the i-th iteration sequence at index I. It could be argued that this is a slight deviation from our abstract model and how we formalized everything else so far, but we found it convenient enough and captures the same intended behaviour. As a possible advantage, it resembles more the functional specification.

PCR Syntax	**TLA$^+$ actions**
c = **`iterate`** $cnd\ f_c\ (v_0\ x)\ x\ p$	$Cstart(I,\ i) \triangleq$ $\quad \land\ \neg wrt(s[I][i])$ $\quad \land\ wrts(p[I],\ deps(X[I],\ Dep\_pc,\ i))$ $\quad \land\ s' = [s \text{ EXCEPT } ![I][i] = \langle v0(X[I])\rangle]$ $\quad \land\ \text{UNCHANGED } \langle X,\ p,\ c,\ r,\ rs\rangle$ $Cstep(I,\ i) \triangleq$ $\quad \land\ wrt(s[I][i])$ $\quad \land\ \neg cnd(s[I][i],\ Len(s[I][i]))$ $\quad \land\ s' = [s \text{ EXCEPT } ![I][i] =$ $\qquad\qquad @ \circ \langle fc(last(s[I][i]),\ X[I],\ p[I],\ i)\rangle]$ $\quad \land\ \text{UNCHANGED } \langle X,\ p,\ c,\ r,\ rs\rangle$ $Cend(I,\ i) \triangleq$ $\quad \land\ \neg wrt(c[I][i])$ $\quad \land\ wrt(s[I][i])$ $\quad \land\ cnd(s[I][i],\ Len(s[I][i]))$ $\quad \land\ c' = [c \text{ EXCEPT } ![I][i] = last(s[I][i])]$ $\quad \land\ \text{UNCHANGED } \langle X,\ p,\ r,\ rs,\ s\rangle$

Table 5.8: TLA$^+$ specification for the *iterate* primitive over a basic function name.

The *Step* action of the iterative PCR is defined as follows:

$$
\begin{array}{lll}
\mathit{Step} \triangleq \exists\, I \in \mathit{WDIndex} : & & \\
\quad \exists\, i \in \mathit{It}(X[I]) : & \lor\, P(I,\, i) & \text{Producer action} \\
 & \lor\, \mathit{Cstart}(I,\, i) & \text{Iteration start action} \\
 & \lor\, \mathit{Cstep}(I,\, i) & \text{Iteration step action} \\
 & \lor\, \mathit{Cend}(I,\, i) & \text{Iteration end action} \\
 & \lor\, R(I,\, i) & \text{Reducer action}
\end{array}
$$

The complete TLA$^+$ specification is in module PCR_A_it (Appendix B).

5.4.10 Iteration over PCR

The main reference for this formalization is the definition of the iterative PCR scheme given in 3.6, in case that the iterable function is a basic PCR. And this is essentially a combination of what we did previously for the iteration over basic functions and composition through the consumer, because the iterate component is used in place of the consumer.

In this setting there is a main PCR $\mathcal{A}$ iterating over a nested basic PCR $\mathcal{B}$. The important thing to note here is that there is no basic function $fc(\_,\_,\_,\_)$. For the functional specification, this means the *iter* function denoting the iterative behaviour of $\mathcal{A}$ should obtain the new values from $B(\langle \mathit{last}(vs), x, vp, i\rangle)$ where B is, of course, the function associated to $\mathcal{B}$. This also means the inputs of $\mathcal{B}$ are quadruples

$$T2 \triangleq D2 \times T \times \mathit{StA}(Tp1) \times \mathit{AssigA}$$

where $D2$ is the type of the reducer variable of $\mathcal{B}$ (i.e. its output type).

For the operational specification, the iteration step action should be further divided into a pair of call and return actions with respect to $\mathcal{B}$. Table 5.9 presents the formalization of the iterative consumer syntax over a PCR in terms of four TLA$^+$ actions. The first and last of them are the same presented previously in 5.8 representing the starting and ending actions of the iteration, the others are the call and return pair on $\mathcal{B}$.

The *StepA* action of the composed iterative PCR is defined as follows:

PCR Syntax	**TLA$^+$ actions**
	$C1start(I,\ i)\ \triangleq$ $\quad\land\ wrts(p1[I],\ depsA(X1[I],\ Dep\_pc1,\ i))$ $\quad\land\ \neg wrt(s[I][i])$ $\quad\land\ s' = [s\ \text{EXCEPT}\ ![I][i] = \langle v0(X1[I])\rangle]$ $\quad\land\ \text{UNCHANGED}\ \langle X1,\ p1,\ c1,\ r1,\ rs1,\ X2\rangle$
	$C1stepIni(I,\ i)\ \triangleq$ $\quad\land\ wrt(s[I][i])$ $\quad\land\ \neg cnd(s[I][i],\ Len(s[I][i]))$ $\quad\land\ \neg wrt(X2[I \circ \langle Len(s[I][i])\rangle])$ $\quad\land\ X2' = [X2\ \text{EXCEPT}\ ![I \circ \langle Len(s[I][i])\rangle] =$ $\qquad\qquad\langle last(s[I][i]),\ X1[I],\ p1[I],\ i\rangle]$ $\quad\land\ \text{UNCHANGED}\ \langle X1,\ p1,\ c1,\ r1,\ rs1,\ s\rangle$
c_1 = **`iterate`** cnd $\mathcal{B}$ (v_0 x_1) x_1 p_1	$C1stepEnd(I,\ i)\ \triangleq$ $\quad\land\ wrt(s[I][i])$ $\quad\land\ wrt(X2[I \circ \langle Len(s[I][i])\rangle])$ $\quad\land\ endB(I \circ \langle Len(s[I][i])\rangle)$ $\quad\land\ s' = [s\ \text{EXCEPT}\ ![I][i] =$ $\qquad\qquad @ \circ \langle r2[I \circ \langle Len(s[I][i])\rangle]\rangle]$ $\quad\land\ \text{UNCHANGED}\ \langle X1,\ p1,\ c1,\ r1,\ rs1,\ X2\rangle$
	$C1end(I,\ i)\ \triangleq$ $\quad\land\ \neg wrt(c1[I][i])$ $\quad\land\ wrt(s[I][i])$ $\quad\land\ cnd(s[I][i],\ Len(s[I][i]))$ $\quad\land\ c1' = [c1\ \text{EXCEPT}\ ![I][i] = last(s[I][i])]$ $\quad\land\ \text{UNCHANGED}\ \langle X1,\ p1,\ r1,\ rs1,\ s,\ X2\rangle$

Table 5.9: TLA$^+$ specification for the *iterate* primitive over a PCR named $\mathcal{B}$.

$$
\begin{aligned}
StepA\ \triangleq\ & \exists\, I \in WDIndexA : \\
& \quad \exists\, i \in ItA(X1[I]) : \ \lor\ P1(I,\ i) && \text{Producer action} \\
& \qquad\qquad\qquad\qquad\quad \lor\ C1start(I,\ i) && \text{Iteration start action} \\
& \qquad\qquad\qquad\qquad\quad \lor\ C1stepIni(I,\ i) && \text{Iteration call action on B} \\
& \qquad\qquad\qquad\qquad\quad \lor\ C1stepEnd(I,\ i) && \text{Iteration return action from B} \\
& \qquad\qquad\qquad\qquad\quad \lor\ C1end(I,\ i) && \text{Iteration end action} \\
& \qquad\qquad\qquad\qquad\quad \lor\ R1(I,\ i) && \text{Reducer action}
\end{aligned}
$$

The complete TLA$^+$ specification is in module PCR_A_it_B (Appendix B).

5.5 Verification of properties

As we explained before when introducing our formalization of the abstract PCR models, each module has a final section that states the safety and liveness properties concerning the PCRs being specified in the module and possibly also refinement properties relating to PCRs specified in other modules. Here, safety properties includes the partial correctness of the PCR computation and other invariant properties like type correctness, whereas liveness properties are, more specifically, termination properties.

Figure 5.1 illustrates the graph of all the refinements that have been established between the ten modules summarised in table 5.2. It should be added that refinements are not always just direct implications between the specification formulas. In general, refinements exists under appropriate substitutions (i.e. refinement mappings). The graph is not connected, because we have not identified yet a refinement between the divide and conquer models and the rest so that it all comes together to the one-step computation model.

$$\begin{tikzcd}
 & \mathit{PCR\_A1step} & \\
\mathit{PCR\_A\_it} \arrow[r] & \mathit{PCR\_A} \arrow[u] & \mathit{PCR\_ArLeft} \arrow[l] \\
\mathit{PCR\_A\_it\_B} \arrow[u] & \mathit{PCR\_A\_c\_B} \arrow[u] & \mathit{PCR\_A\_r\_B} \arrow[ul]
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{PCR\_DC} & \mathit{PCR\_DCrLeft} \arrow[l] \\
\mathit{PCR\_DC\_r\_DCrLeft} \arrow[u] &
\end{tikzcd}$$

Figure 5.1: A hierarchy of refinements between the ten modules formalizing the abstract PCR models.

In the TLA$^+$ framework, the mechanical verification of this properties can be done, subject to some restrictions, in either of the following ways:

- Model checking with TLC. The modules we presented so far as a formalization of the abstract PCR models are specifying general classes of PCRs. For model checking purposes, constant symbol parameters need to be instantiated with concrete definitions that can be evaluated, which result in concrete PCRs. This means that model checking can be used as a method of verification for concrete PCRs over finite data types. TLC allows to verify safety, liveness and refinement.[15]

- Assisted theorem proving with TLAPS. For the deductive approach we don't need to instantiate constant symbol parameters. This means verification can be actually done for the abstract PCR model, i.e. general classes of PCRs without finiteness restrictions. TLAPS currently allows to verify safety properties and the safety component of refinement properties (i.e. without fairness), liveness properties are ruled out.

For this thesis, we have done theorem proving for the following:

- All the safety properties of the basic PCR (PCR_A). They are in module PCR_A_Thms at GitHub. In particular, the correctness proof is also in Appendix C.

- All the safety properties of the basic PCR with left reducer (PCR_ArLeft). They are in module PCR_ArLeft_Thms at GitHub. In particular, the correctness proof is also in Appendix C.

- A basic PCR (PCR_A) is a refinement of a basic PCR expressed as a one step computation (PCR_A1step). It is in module PCR_A_Thms at GitHub and can also be found at Appendix C.

- A basic PCR with left reducer (PCR_ArLeft) is a refinement of a basic PCR (PCR_A). It is in module PCR_ArLeft_Thms at GitHub and can also be found at Appendix C.

- The composition through the consumer ($PCR\_A\_c\_B$) is a refinement of a basic PCR (PCR_A). It is in module PCR_A_c_B_Thms at GitHub and can also be found at Appendix C.

[15]By default, TLC searches for deadlocks. However, the absence of deadlocks does not rule out infinite pathological behaviours like livelock or starvation.

- The assumptions made for the basic PCR where expressed as lemmas/theorems and proved for the concrete PCR FibPrimes1. They are in modules PCR_FibPrimes1_Lems and PCR_FibPrimes1_Thms at GitHub.

All this and everything else has been verified by model checking on concrete PCRs. The complete specification of the concrete PCRs are in PCR/Concrete (GitHub) and Appendix D. They are just modules instantiating the abstract PCR modules we presented previously with the concrete elements that characterize them (e.g. types, data dependencies, basic functions, etc.). We summarize the results of model checking in Appendix E.

In what follows, we discuss properties and deductive proofs for some of the mentioned models. We will present semi-formal versions of the proofs, preserving the formal hierarchical proof structure.

5.5.1 Basic PCR

The most basic property we can assert for a basic PCR (and any TLA$^+$ specification for that matter) is type correctness:

$$
\begin{array}{lll}
\mathit{TypeInv} \triangleq & \land\ \mathit{in} \in T & \\
& \land\ X \in [\mathit{Index} \rightarrow T \cup \{\mathit{Undef}\}] \land X^{I_0} = \mathit{in} & \\
& \land\ p\ \in [\mathit{Index} \rightarrow \mathit{St}(\mathit{Tp})] & \\
& \land\ c\ \in [\mathit{Index} \rightarrow \mathit{St}(\mathit{Tc})] & \\
& \land\ r\ \in [\mathit{Index} \rightarrow D] & \\
& \land\ \mathit{rs} \in [\mathit{Index} \rightarrow [\mathit{Assig} \rightarrow \text{BOOLEAN}]] &
\end{array}
$$

This can be proved directly as an inductive invariant using rule INV1. But instead, we have proved it by rule INV3 relying on the following inductive invariant which states that the only well defined instance of the main PCR have index I_0 (an expected consequence of our assumption 5.2.4):

$$\mathit{IndexInv} \triangleq \mathit{WDIndex} = \{ I_0 \}$$

In general, all the properties of the main PCR are stated relative to index I_0, thus the their proofs relies on *IndexInv*, as well on *TypeInv*. The following invariant *PInv* states the expected consequences of producer actions:

$$\mathit{PInv} \triangleq \forall\, i \in \mathit{It}(X^{I_0}) \,:\, \mathit{wrt}(p^{I_0,i}) \;\Rightarrow\; \begin{array}{l} \wedge\ \mathit{wrts}(p^{I_0}, \mathit{deps}(X^{I_0}, \mathit{Dep\_pp}, i)) \\ \wedge\ p^{I_0,i} = f_p(X^{I_0}, p^{I_0}, i) \end{array}$$

That is, if a producer assignment i has been written, then the past dependencies relative to i where meet and the written value was calculated by the basic function f_p at i. Analogously, a very similar invariant *CInv* states the expected consequences of consumer actions:

$$\mathit{CInv} \triangleq \forall\, i \in \mathit{It}(X^{I_0}) \,:\, \mathit{wrt}(c^{I_0,i}) \;\Rightarrow\; \begin{array}{l} \wedge\ \mathit{wrts}(p^{I_0}, \mathit{deps}(X^{I_0}, \mathit{Dep\_pc}, i)) \\ \wedge\ c^{I_0,i} = f_c(X^{I_0}, p^{I_0}, i) \end{array}$$

As for the reducer actions, two separate invariants *RInv*1 and *RInv*2 state their consequences:

$$\begin{array}{l} \mathit{RInv}1 \triangleq \forall\, i \in \mathit{It}(X^{I_0}) \,:\, \mathit{red}(I_0, i) \;\Rightarrow\; \mathit{wrts}(c^{I_0}, \mathit{deps}(X^{I_0}, \mathit{Dep\_cr}, i)) \\ \mathit{RInv}2 \triangleq r^{I_0} = \displaystyle\bigotimes_{\{i \,\in\, m..n \,:\, Q(i) \,\wedge\, \mathit{red}(I_0,i)\}} f_r(x, c^{I_0}, i) \end{array}$$

where $m = \mathit{lBnd}(X^{I_0})$, $n = \mathit{uBnd}(X^{I_0})$ and $Q(i) = \mathit{prop}(i)$. These two invariants are related, they were separated for convenience. Indeed, the proof of *RInv*2 relies also on *RInv*1. In particular, *RInv*2 characterizes the reducer output value at any time of the PCR computation as the combination (i.e. $\otimes$) of the values computed at the assignments marked as reduced.

All the previous invariants are conjoined into one invariant named *Inv*:

$$\mathit{Inv} \triangleq \mathit{TypeInv} \wedge \mathit{IndexInv} \wedge \mathit{PInv} \wedge \mathit{CInv} \wedge \mathit{RInv}1 \wedge \mathit{RInv}2$$

Now, note that when model checking invariant properties the graph of *reachable* states is generated. But that is not a verification of the inductiveness of the invariant, as not all invariants are *inductive*. It is possible to use the model checker TLC to verify , on finite concrete PCRs, that *Inv* is actually an inductive invariant. This constitutes a particular and interesting use case of TLC. For this, we need to define first an alternative main specification formula for the PCR as follows:

$$\mathit{ISpec} \triangleq \mathit{Inv} \wedge \Box[\mathit{Next}]_{\langle in, vs\rangle}$$

Note that instead of using *Init* as initial state formula, like in the main *Spec*, we use *Inv*. Then, we can instruct TLC to check *ISpec* satisfies the invariant *Inv*. If the verification is positive, we have gained confidence in that *Inv* is inductive and not just an invariant, which can be useful later when doing deductive reasoning. This technique is explained by Lamport as a tutorial in [77].

Partial correctness and termination is stated as follows:

$$\begin{aligned} \mathit{Correctness} &\triangleq \mathit{end}(I_0) \Rightarrow r^{I_0} = \mathcal{A}(X^{I_0}) \\ \mathit{Termination} &\triangleq \Diamond \mathit{end}(I_0) \end{aligned}$$

5.5.1.1 Partial Correctness

Next, we prove partial correctness of the basic PCR relying on the invariant properties presented before.

Theorem 5.7 (*PCR_A_Thms*!*Thm_Correctness*)**.**

$$\mathit{Spec} \Rightarrow \Box(\mathit{end}(I_0) \Rightarrow r^{I_0} = \mathcal{A}(x^{I_0}))$$

Proof. First, let us make the following abbreviations:

$$\begin{aligned} x &= X^{I_0} \\ y &= r^{I_0} \\ m &= \mathit{lBnd}(x) \\ n &= \mathit{uBnd}(x) \\ Q(i) &= \mathit{prop}(i) \\ \overrightarrow{f}_{\mathcal{A}} &= \overrightarrow{f}_r(x, \overrightarrow{f}_c(x, \overrightarrow{g}_p(x))) \end{aligned}$$

By rule INV3 it is enough to prove ⟨1⟩1 and ⟨1⟩2 with the help of invariant *Inv*:

⟨1⟩1. $\mathit{Init} \Rightarrow \mathit{Correctness}$

It suffices to assume *Init* and $\mathit{end}(I_0)$ to prove $y = \mathcal{A}(x)$. We proceed by cases on iteration space $\mathit{It}(x)$:

⟨2⟩A. $\mathit{It}(x) = \varnothing$: For this to hold, there are two possibilities:

i. $m > n$: Then trivially $\mathcal{A}(x) = \mathtt{id}_\otimes$. Besides, $y = \mathtt{id}_\otimes$ by *Init*. Therefore $y = \mathcal{A}(x)$.

ii. $\neg Q(i)$ for all $i \in m..n$: Then $\mathcal{A}(x) = \mathtt{id}_\otimes$ by *FalsePredicate*. Besides, $y = \mathtt{id}_\otimes$ by *Init*. Therefore $y = \mathcal{A}(x)$.

$\langle 2 \rangle$B. $It(x) \neq \varnothing$: We show this case is not possible arriving at a contradiction. By our assumption $end(I_0)$ it should be the case that $red(I_0, j)$ for any $j \in It(x)$. But in fact, is the opposite, by *Init* we have $\neg red(I_0, j)$ for any $j \in It(x)$.

$\langle 1 \rangle$2. $Inv \land Correctness \land [Next]_{\langle in, vs \rangle} \Rightarrow Correctness'$

We have that $\mathcal{A}(x)$ is a constant expression, i.e. $\mathcal{A}(x)' = \mathcal{A}(x)$. Thus, it suffices to assume *Inv*, *Correctness*, $[Next]_{\langle in, vs \rangle}$ and $end(I_0)'$ to prove $y' = \mathcal{A}(x)$. By def. of $[Next]_{\langle in, vs \rangle}$ there are three cases to consider:

$\langle 2 \rangle$A. *Step* : By invariant *IndexInv* it suffices to assume there exists some $i \in It(x)$ such that

$$P(I_0, i) \ \lor \ C(I_0, i) \ \lor \ R(I_0, i)$$

We proceed by cases on these actions:

$\langle 3 \rangle$A. $P(I_0, i)$: We show this case is not possible arriving at a contradiction. By invariants *CInv* and *RInv*1 we have for all $j \in It(x)$

$$(red(I_0, j) \Rightarrow wrt(c^{I_0, j})) \ \land \ (wrt(c^{I_0, j}) \Rightarrow wrt(p^{I_0, j})) \qquad (5.4)$$

from which we can deduce

$$(\neg wrt(c^{I_0, j}) \Rightarrow \neg red(I_0, j) \ \land \ (\neg wrt(p^{I_0, j}) \Rightarrow \neg wrt(c^{I_0, j})) \qquad (5.5)$$

By the producer action $P(I_0, i)$ we have $\neg wrt(p^{I_0, i})$, thus by formula 5.5 it follows that $\neg red(I_0, i)$. Besides, $rs' = rs$, and thus $\neg red(I_0, i)'$ also holds. But the last fact contradicts with our assumption $end(I_0)'$ as this means $red(I_0, j)'$ for all $j \in It(x)$.

$\langle 3 \rangle$B. $C(I_0, i)$: Not possible. Proof analogous to previous case.

$\langle 3 \rangle$C. $R(I_0, i)$: The following holds in the reducer action:

$$\neg \mathit{red}(I_0, i) \tag{5.6}$$
$$\wedge \; \mathit{wrts}(c^{I_0}, \mathit{deps}(x, \mathit{Dep\_cr}, i)) \tag{5.7}$$
$$\wedge \; y' \; = \; y \otimes f_r(x, c^{I_0}, i) \tag{5.8}$$

First, note that by assumption $\mathit{end}(I_0)'$ we have $\mathit{red}(I_0, j)'$ for all $j \in \mathit{It}(x)$. So, by 5.6 the *only* assignment that has *not* been reduced yet is current i. This and 5.7 implies that *all* producer and consumer assignments have been *written*.

Now, we gather some facts.

i. $Q(j) \wedge j \neq i \;\equiv\; Q(j) \wedge \mathit{red}(I_0, j)$ for all $j \in m..n$

Because the *only* assignment that has *not* been reduced yet is the current i.

ii. $\overrightarrow{f}^{\,j}_{\mathcal{A}} \;=\; f_r(x, c^{I_0}, j)$ for all $j \in \mathit{It}(x)$

By def. of $\overrightarrow{f}_{\mathcal{A}}$ and $\overrightarrow{f}_r$, we need to prove:

$$f_r(x, \overrightarrow{f}_c(x, \overrightarrow{g}_p(x)), j) = f_r(x, c^{I_0}, j) \;\text{ for any }\; j \in \mathit{It}(x)$$

Take any $j \in \mathit{It}(x)$. By axiom *H_frRelevance* and def. of $\overrightarrow{f}_c$, it suffices to prove:

$$f_c(x, \overrightarrow{g}_p(x)), k) = c^{I_0,k} \;\text{ for any }\; k \in \mathit{deps}(x, \mathit{Dep\_cr}, j)$$

But, by invariant *CInv*, we can prove instead:

$$f_c(x, \overrightarrow{g}_p(x)), k) = f_c(x, p^{I_0}, k) \;\text{ for any }\; k \in \mathit{deps}(x, \mathit{Dep\_cr}, j)$$

Take any $k \in \mathit{deps}(x, \mathit{Dep\_cr}, j)$. By axiom *H_frRelevance* and def. of $\overrightarrow{g}_p$ it suffices to prove:

$$g_p(x, l) = p^{I_0,l} \;\text{ for any }\; l \in \mathit{deps}(x, \mathit{Dep\_pc}, k)$$

But, by invariant *PInv*, we can prove instead:

$$g_p(x, l) = f_p(x, p^{I_0}, l) \;\text{ for any }\; l \in \mathit{deps}(x, \mathit{Dep\_pc}, k)$$

Which it is true by axiom *H_ProdEqInv*.

Finally, let us calculate:

$$\begin{array}{rll}
\mathcal{A}(x) & = \displaystyle\bigotimes_{\{j \in m..n \,:\, Q(j)\}} \vec{f}^{\,j}_{\mathcal{A}} & \text{by def. } \mathcal{A} \\
& = \displaystyle\bigotimes_{\{j \in m..n \,:\, Q(j) \wedge j \neq i\}} \vec{f}^{\,j}_{\mathcal{A}} \otimes \vec{f}^{\,i}_{\mathcal{A}} & \text{by } \mathit{SplitRandomP} \\
& = \displaystyle\bigotimes_{\{j \in m..n \,:\, Q(j) \wedge \mathit{red}(I_0, j)\}} \vec{f}^{\,j}_{\mathcal{A}} \otimes \vec{f}^{\,i}_{\mathcal{A}} & \text{by i. and } \mathit{PredicateEq} \\
& = \displaystyle\bigotimes_{\{j \in m..n \,:\, Q(j) \wedge \mathit{red}(I_0, j)\}} f_r(x, c^{I_0}, j) \otimes f_r(x, c^{I_0}, i) & \text{by ii. and } \mathit{FunctionEqP} \\
& = y \otimes f_r(x, c^{I_0}, i) & \text{by invariant } \mathit{RInv2} \\
& = y' & \text{by 5.8}
\end{array}$$

⟨2⟩B. *Done* : We have that $end(I_0)$ and $\langle in, vs\rangle' = \langle in, vs\rangle$ holds. So it must be that $y' = y$ because nothing changes. By the *Correctness* assumption and $end(I_0)$ we have $y = \mathcal{A}(x)$. Therefore $y' = y = \mathcal{A}(x)$.

⟨2⟩C. $\langle in, vs\rangle' = \langle in, vs\rangle$: By assumption we have $end(I_0)'$. As nothing changes, it must be that $end(I_0)' = end(I_0)$ and $y' = y$, thus we also have $end(I_0)$. Then, by the *Correctness* assumption and $end(I_0)$ we have $y = \mathcal{A}(x)$. Therefore, $y' = y = \mathcal{A}(x)$.

□

As a general comment about proofs, at some point we realized that we had never used rule INV2, instead we always relied on the more general rule INV3 (and INV1 for the most basic properties). As we mentioned earlier, we gained confidence in that *Inv* is an inductive invariant via model checking verification. Then, it should not be hard to corroborate it formally and apply rule INV2 which then requires to prove

$$\mathit{Inv} \;\Rightarrow\; \mathit{Correctness}$$

in order to conclude $\mathit{Spec} \Rightarrow \Box\mathit{Correctness}$ as we desired. That would have resulted in, perhaps, a more direct proof eliding the cases we justified resorting to their impossibility. However, note that in those cases we did not made use of indirect reasoning, we reasoned "by negation" and not "by contradiction", thus being acceptable from the constructive point of view.

5.5.1.2 Refinement of a one step computation

Next, we prove that the basic PCR is a refinement of the basic PCR expressed as a one step computation. First, we instantiate module PCR_A1step as follows:

$$\begin{aligned} inS &\triangleq X^{I_0} \\ outS &\triangleq \text{IF } end(I_0) \text{ THEN } r^{I_0} \text{ ELSE } \mathtt{id}_{\otimes} \\ A1step &\triangleq \text{INSTANCE } PCR\_A1step \text{ WITH } in \leftarrow inS,\ out \leftarrow outS \end{aligned}$$

Typically, we suffix by S the names of the symbols substituted with new non-trivial expressions (i.e. not the identity) which allow us to hide part of the state. Figure 5.2 presents an intuitive illustration of the refinement. For convenience, we will reuse the

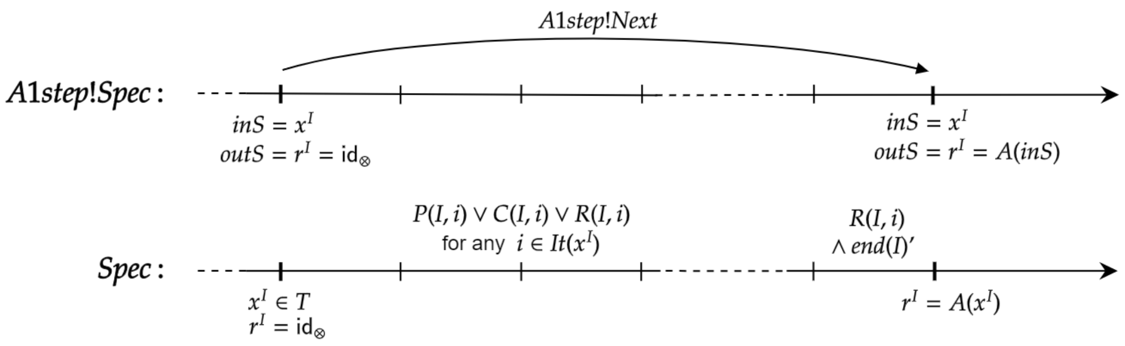


Figure 5.2: $PCR\_A!Spec$ (here $Spec$) refines $PCR\_A1step!Spec$ (here $A1step!Spec$). All the computation in $Spec$ before the last reduction action is simulated by stuttering steps at $A1step!Spec$. What allows $A!Spec$ to stutter is that the substitution sets $outS = \mathtt{id}_{\otimes}$ until termination, thus hiding from $A!Spec$ the intermediate partial values of the reducer output at the low level. The last reduction action corresponds to the single step action at $A1step!Spec$.

previous (partial) correctness result. This can be seen as a bit of cheating as the same formula $\mathcal{A}(x)$ that is used in the correctness property is also used in the one step computation. But we actually do not need correctness, we only need invariant Inv. The long version of the formal proof is in Appendix C.

Theorem 5.8 (*PCR_A_Thms!Thm_Refinement*)**.**

$$Spec \;\Rightarrow\; A1step!Spec$$

Proof. First, let us make the following abbreviations:

$$\begin{aligned} x &= X^{I_0} \\ y &= r^{I_0} \end{aligned}$$

By rule REF it is enough to prove ⟨1⟩1 and ⟨1⟩2 with the help of invariants *Inv* and *Correctness*:

⟨1⟩1. $Init \;\Rightarrow\; A1step!Init$

By def. of *Init* we have $x \in T$ and $y = \mathtt{id}_{\otimes}$, so

$$outS \;=\; \text{IF } end(I_0) \text{ THEN } y \text{ ELSE } \mathtt{id}_{\otimes} \;=\; \mathtt{id}_{\otimes}$$

Therefore *A*1*step*!*Init* holds.

⟨1⟩2. $Inv \wedge Correctness' \wedge [Next]_{\langle in, vs\rangle} \;\Rightarrow\; [A1step!Next]_{\langle inS, outS\rangle}$

It suffices to assume *Inv*, *Correctness*′ and $[Next]_{\langle in, vs\rangle}$ to prove that *A*1*step*!*Spec* either makes a *A*1*step*!*Next* step or stutters, that is:

$$outS = \mathcal{A}(inS) \;\vee\; \langle inS, outS\rangle' = \langle inS, outS\rangle$$

By def. of $[Next]_{\langle in, vs\rangle}$ there are three cases to consider:

⟨2⟩A. *Step* : By invariant *IndexInv* it suffices to assume there exists some $i \in It(x)$ such that

$$P(I_0, i) \;\vee\; C(I_0, i) \;\vee\; R(I_0, i)$$

We proceed by cases on these actions, and further consider for the case $R(I_0, i)$ the mutually exclusive possibilities $end(I_0)'$ and $\neg end(I_0)'$.

⟨3⟩A. $P(I_0, i)$: In a producer action, we have $\neg end(I_0)$ and $\neg end(I_0)'$, thus $outS' \;=\; outS \;=\; \mathtt{id}_{\otimes}$. Also, input *inS* remains constant. Therefore $\langle inS, outS\rangle' = \langle inS, outS\rangle$.

⟨3⟩B. $C(I_0, i)$: Analogous to previous case.

⟨3⟩C. $R(I_0, i) \wedge end(I_0)'$: In this case, this is the last reduction. We have that

$outS' \neq outS$, but input inS remains constant. We prove $outS' = \mathcal{A}(inS)$ as follows:

$$\begin{array}{rcll} outS' & = & y' & \text{by } end(I_0)' \\ & = & \mathcal{A}(inS)' & \text{by } Correctness' \text{ and } end(I_0)' \\ & = & \mathcal{A}(inS) & \text{input is constant} \end{array}$$

⟨3⟩D. $R(I_0, i) \wedge \neg end(I_0)'$: In this case, this is not the last reduction. If $\neg end(I_0)'$ then also $\neg end(I_0)$, thus $outS' = outS = \texttt{id}_{\otimes}$. Besides, input inS remains constant. Therefore $\langle inS, outS\rangle' = \langle inS, outS\rangle$.

⟨2⟩B. *Done* : We have that $\langle in, vs\rangle' = \langle in, vs\rangle$, i.e. *Spec* stutters. Variables inS and $outS$ of $A1step!Spec$ are expressed in terms of vs by our substitution. Therefore, if *Spec* stutters then $A1step!Spec$ also stutters.

⟨2⟩C. $\langle in, vs\rangle' = \langle in, vs\rangle$: Analogous to previous case.

□

5.5.2 Basic PCR with left reducer

A basic PCR with left reducer differs from the ordinary one only in the reducer component specification. However, recall that we assume a consecutive iteration space for simplicity, i.e. there is no filter condition.

Here, the relevant difference is in the invariants concerning the consequences of the reducer actions:

$$\begin{array}{rl} RInv1 \triangleq \forall\, i \in It(X^{I_0}) \,:\, red(I_0, i) \,\Rightarrow & \wedge\, wrts(c^{I_0}, deps(X^{I_0}, Dep\_cr, i)) \\ & \wedge\, \forall\, k \in It(X^{I_0}) \,:\, k < i \,\Rightarrow\, red(I_0, k) \\ RInv2 \triangleq \forall\, i \in It(X^{I_0}) \,:\, \neg red(I_0, i) \,\Rightarrow & r^{I_0} = \displaystyle\bigotimes_{\{j \,\in\, m..i-1 \,:\, red(I_0, j)\}} f_r(x, c^{I_0}, j) \end{array}$$

where $m = lBnd(X^{I_0})$. Note that $RInv1$ now also states that for any assignment i that has been reduced, we can be sure all the previous assignments $k < i$ have also been reduced. Unfortunately, $RInv2$ is a bit more involved than what we saw for the ordinary basic

PCR. Here, it states that for any assignment i that has not been reduced, the reducer output value is the combination (i.e. $\otimes$) of all the values computed at the assignments from m to $i-1$ that are marked as reduced.[16]

5.5.2.1 Partial Correctness

Most of the partial correctness proof for *PCR_ArLeft* is very similar to that for *PCR_A* (theorem 5.7). So, in what follows we concentrate only on what happens at the reducer component. A fundamental difference is that we have less mathematical facts available, because commutativity is not assumed.

Theorem 5.9 (*PCR_ArLeft_Thms*!*Thm_Correctness*)**.**

$$Spec \;\Rightarrow\; \Box(end(I_0) \;\Rightarrow\; r^{I_0} = \mathcal{A}(x^{I_0}))$$

Proof. First, let us make the following abbreviations:

$$\begin{aligned} x &= X^{I_0} \\ y &= r^{I_0} \\ m &= lBnd(x) \\ n &= uBnd(x) \\ \overrightarrow{f}_{\mathcal{A}} &= \overrightarrow{f}_r(x, \overrightarrow{f}_c(x, \overrightarrow{g}_p(x))) \end{aligned}$$

Having as hypothesis invariant *Inv* and condition $end(I_0)'$, now assume reducer action $R(I_0, i)$ for some $i \in It(x) = m..n$. The following holds in this action:

$$\neg red(I_0, i) \tag{5.9}$$
$$\wedge \;\; wrts(c^{I_0}, deps(x, Dep\_cr, i)) \tag{5.10}$$
$$\wedge \;\; i-1 \geq m \;\Rightarrow\; red(I_0, i-1) \tag{5.11}$$
$$\wedge \;\; y' \;=\; y \otimes f_r(x, c^{I_0}, i) \tag{5.12}$$

First, note that by assumption $end(I_0)'$ we have $red(I_0, j)'$ for all $j \in m..n$. So, by 5.9 the *only* assignment that has *not* been reduced yet is current i. This and 5.10 implies that *all* producer and consumer assignments have been *written*.

[16]Actually, we found simpler alternatives for $RInv2$. But the proof would need to be done again.

Now, we gather some facts.

i. $\mathit{red}(I_0, j)$ for all $j \in m..(i-1)$

   Because by invariant $RInv1$ we know that $\mathit{red}(I_0, j)$ holds for all $j < i$ where $j \in m..n$.

ii. $i = n$

   We know that $\neg\mathit{red}(I_0, i)$ by 5.9 where $i \in m..n$, then by i. it must be the case that $i = n$.

iii. $\vec{f}^{\,j}_{\mathcal{A}} = f_r(x, c^{I_0}, j)$ for all $j \in m..n$

   Proceed exactly like we did in point ii. for the basic PCR.

Finally, let's calculate:

$$\begin{aligned}
\mathcal{A}(x) &= \bigotimes_{j=m}^{n} \vec{f}^{\,j}_{\mathcal{A}} && \text{by def. } \mathcal{A}\\
&= \bigotimes_{j=m}^{n-1} \vec{f}^{\,j}_{\mathcal{A}} \otimes \vec{f}^{\,n}_{\mathcal{A}} && \text{by } \mathit{SplitLast}\\
&= \bigotimes_{\{j \in m..n-1 \,:\, \mathit{red}(I_0,j)\}} \vec{f}^{\,j}_{\mathcal{A}} \otimes \vec{f}^{\,n}_{\mathcal{A}} && \text{by i. and } \mathit{TruePredicate}\\
&= \bigotimes_{\{j \in m..n-1 \,:\, \mathit{red}(I_0,j)\}} f_r(x, c^{I_0}, j) \otimes f_r(x, c^{I_0}, n) && \text{by iii. and } \mathit{FunctionEqP}\\
&= y \otimes f_r(x, c^{I_0}, n) && \text{by invariant } RInv2 \text{ and } 5.9\\
&= y' && \text{by } 5.12 \text{ and ii.}
\end{aligned}$$

□

5.5.2.2 Refinement of a basic PCR

The fact that the basic PCR with left reducer is a refinement of the ordinary basic PCR is a very direct result. It is easy to see that any behavior of the former is an admitted behavior on the later, as the former only reduces in a fixed order but the later allows any order. To prove this refinement we do not require any non-trivial substitution when doing module instantiation. The formal proof is in Appendix C.

5.5.3 Composition through consumer

Here we have a main PCR $\mathcal{A}$ and a nested basic PCR $\mathcal{B}$. There are properties stated for both of them separately, so that we group most of them as *InvA* and *InvB* for convenience. But $\mathcal{B}$ is a basic PCR, so its properties are essentially no different from what we saw earlier except that now there can be multiple instances of $\mathcal{B}$ and their indexes are now characterized by the following invariant:

$$\mathit{IndexInvB} \triangleq \mathit{WDIndexB} \subseteq \{I_0 \circ \langle i \rangle : i \in \mathit{AssigA}\}$$

PCR $\mathcal{A}$ is more interesting because the consumer specification is different from a basic PCR. As we explained in the formalization, there is a pair of call and return actions with respect to $\mathcal{B}$. The following invariant $\mathit{CInv}1A$ state the consequences of a call action on $\mathcal{B}$:

$$\begin{array}{l}
\mathit{CInv}1A \triangleq \\
\quad \forall\, i \in \mathit{ItA}(X_1^{I_0}) \,:\, \mathit{wrt}(X_2^{I_0,i}) \;\Rightarrow\; \begin{array}[t]{l} \wedge\, \mathit{wrts}(p_1^{I_0}, \mathit{depsA}(X_1^{I_0}, \mathit{Dep\_pc}1, i)) \\ \wedge\, \exists\, \mathit{vp} \in \overrightarrow{T}_{p_1} \,: \\ \qquad \wedge\; \mathit{eqs}(\mathit{vp}, p_1^{I_0}, \mathit{depsA}(X_1^{I_0}, \mathit{Dep\_pc}1, i)) \\ \qquad \wedge\; X_2^{I_0,i} = \langle X_1^{I_0}, \mathit{vp}, i \rangle \end{array}
\end{array}$$

That is, if an input on $\mathcal{B}$ has been written on assignment i, then the dependencies with respect to the producer and relative to i were met and there exists some stream vp on range type T_{p_1} such that is equal to the current producer stream $p_1^{I_0}$ and whose input for $\mathcal{B}$ is exactly the triplet $\langle X_1^{I_0}, \mathit{vp}, i\rangle$. Something important to note here is that it would be erroneous to assert the input is $\langle X_1^{I_0}, p_1^{I_0}, i\rangle$, because $p_1^{I_0}$ will continue to evolve in time after the call, whereas the input for $\mathcal{B}$ is actually only a "freezed" version of $p_1^{I_0}$ at the instant of the call[17], so the invariant would be surely violated afterwards.

[17]It could be argued that in mathematics everything is passed "by value", using programming terminology. That is exactly what is happening here.

Invariant $\mathit{CInv2A}$ states the consequences of a return action from $\mathcal{B}$:

$$\mathit{CInv2A} \triangleq \forall\, i \in \mathit{ItA}(X_1^{I_0}) \,:\, \mathit{wrt}(c_1^{I_0,i}) \;\Rightarrow\; \begin{array}{l} \wedge\ \mathit{wrt}(X_2^{I_0,i}) \\ \wedge\ \mathit{endB}(I_0, i) \\ \wedge\ c_1^{I0,i} = r_2^{I_0,i} \end{array}$$

That is, if a consumer assignment i has been written, then there is an input written on $\mathcal{B}$ for which the corresponding instance terminated and the written value came from the output variable of that instance of $\mathcal{B}$.

Partial correctness and termination is stated for both PCRs as follows:

$$\begin{aligned} \mathit{CorrectnessA} &\triangleq \mathit{endA}(I_0) \;\Rightarrow\; r_1^{I_0} = \mathcal{A}(X_1^{I_0}) \\ \mathit{TerminationA} &\triangleq \Diamond \mathit{endA}(I_0) \end{aligned}$$

$$\begin{aligned} \mathit{CorrectnessB} &\triangleq \forall\, I \in \mathit{WDIndexB} : \mathit{endB}(I) \;\Rightarrow\; r_2^I = \mathcal{B}(X_2^I) \\ \mathit{TerminationB} &\triangleq \Diamond(\forall\, I \in \mathit{WDIndexB} : \mathit{endB}(I)) \end{aligned}$$

The correctness of the overall specification is what we state for the main PCR $\mathcal{A}$, as always relative to I_0, which in turns depends on the correctness of possibly multiple instances of $\mathcal{B}$. However, the proof of the refinement of a basic PCR, to be discussed next, only relies on the correctness of $\mathcal{B}$ (and some basic invariants of $\mathcal{A}$).

The relation between the termination of both PCRs can be made explicit by the invariant:

$$\mathit{endA}(I_0) \;\Rightarrow\; \forall\, I \in \mathit{WDIndexB} : \mathit{endB}(I)$$

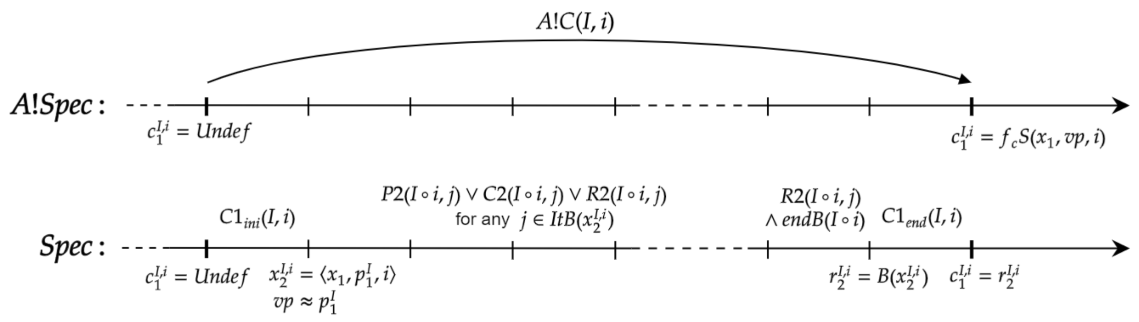
It should be obvious that the converse does not hold.

5.5.3.1 Refinement of a basic PCR

In what follows, we prove that a PCR $\mathcal{A}$ composed through the consumer with a basic PCR $\mathcal{B}$ is a refinement of a basic PCR where the consumer computes the function associated to $\mathcal{B}$. First, we instantiate module *PCR_A* as follows:

$$
\begin{aligned}
inS &\triangleq X^{I_0} \\
f_cS(x, vp, i) &\triangleq \mathcal{B}(\langle x, vp, i \rangle) \\
A &\triangleq \text{INSTANCE } PCR\_A \text{ WITH } in \leftarrow inS,\ f_c \leftarrow f_cS
\end{aligned}
$$

Figure 5.2 presents an intuitive illustration of the refinement. Before the proof, we still



$A!C(I, i)$
$A!Spec:$
$c_1^{I,i} = Undef$
$c_1^{I,i} = f_cS(x_1, vp, i)$
$C1_{ini}(I, i)$
$P2(I \circ i, j) \vee C2(I \circ i, j) \vee R2(I \circ i, j)$
for any $j \in ItB(x_2^{I,i})$
$R2(I \circ i, j)$
$\wedge\, endB(I \circ i)$
$C1_{end}(I, i)$
$Spec:$
$c_1^{I,i} = Undef$
$x_2^{I,i} = \langle x_1, p_1^I, i \rangle$
$vp \approx p_1^I$
$r_2^{I,i} = B(x_2^{I,i})$
$c_1^{I,i} = r_2^{I,i}$

Figure 5.3: *PCR_A_c_B*!*Spec* (here *Spec*) refines *PCR_A*!*Spec* (here *A*!*Spec*). All the computation in *Spec* starting with a call action on $\mathcal{B}$ till the termination of $\mathcal{B}$ is simulated by stuttering steps at *A*!*Spec*. The return action from $\mathcal{B}$ in *Spec* corresponds to the consumer action at *A*!*Spec*. We write $vp \approx p_1^I$ to abbreviate $eqs(vp, p_1^I, depsA(X_1^I, Dep\_pc1, i))$, i.e. they are equivalent streams relative to the dependencies the consumer has on the producer.

need other relevance axioms apart of what is already asserted for $\mathcal{A}$ and $\mathcal{B}$ in the specification. Note that PCR $\mathcal{B}$ has as input the producer stream variable of the father PCR $\mathcal{A}$. Thus, all the basic functions of $\mathcal{B}$ can use not only their own internal streams corresponding to internal variables but also the stream received in the input which corresponds to an external variable (in this case the producer variable of $\mathcal{A}$). So, we also need to know that *all* the basic functions of $\mathcal{B}$ are also insensitive to indexes of that stream which they do not depend on. We have not added all the corresponding relevance axioms in the specification module *PCR_A_c_B* because, at least for the purposes of the refinement proof, we can capture all that information in a single axiom as follows:[18]

$$\begin{array}{l}\textsc{axiom}\ \mathit{H\_fcSRelevance}\ \triangleq \\ \quad \forall\, x \in T\ :\ \forall\, i \in \mathit{ItA}(x),\ \mathit{vp}_1 \in \overrightarrow{T}_{p_1},\ \mathit{vp}_2 \in \overrightarrow{T}_{p_1}\ : \\ \qquad \mathit{eqs}(\mathit{vp}_1, \mathit{vp}_2, \mathit{depsA}(x, \mathit{Dep\_pc}1, i))\ \Rightarrow\ f_c S(x, \mathit{vp}_1, i) = f_c S(x, \mathit{vp}_2, i)\end{array}$$

Without further ado, we present the proof.

Theorem 5.10 (*PCR_A_c_B_Thms*!*Thms_Refinement*)**.**

$$\mathit{Spec}\ \Rightarrow\ A!\mathit{Spec}$$

Proof. First, let us make the following abbreviations:

$$\begin{array}{rcl} x_1 & = & X_1^{I_0} \\ x_2^I & = & X_2^I \end{array}$$

By rule REF it is enough to prove $\langle 1\rangle 1$ and $\langle 1\rangle 2$ with the help of invariants *InvA* and *CorrectnessB*:

$\langle 1\rangle$1. $\mathit{Init}\ \Rightarrow\ A!\mathit{Init}$

We have $\mathit{Init} = \mathit{InitA} \wedge \mathit{InitB}$, and *InitA* by def. asserts the same initial conditions that $A!\mathit{Init}$, therefore $A!\mathit{Init}$ holds.

$\langle 1\rangle$2. $\mathit{InvA}\ \wedge\ \mathit{CorrectnessB}\ \wedge\ [\mathit{Next}]_{\langle in, vs1, vs2\rangle}\ \Rightarrow\ [A!\mathit{Next}]_{\langle inS, A!vs\rangle}$

By our substitution, we have that $A!vs = \langle x_1, p_1, c_1, r_1, rs_1\rangle$ and $inS = x_1$. So, it suffices to assume *InvA*, *CorrectnessB* and $[\mathit{Next}]_{\langle in, vs1, vs2\rangle}$ to prove that $A!\mathit{Spec}$ either makes a $A!\mathit{Next}$ step or stutters on variables $A!vs$, that is:

$$A!\mathit{Next}\ \vee\ A!vs' = A!vs$$

By def. of $[\mathit{Next}]_{\langle in, vs1, vs2\rangle}$ there are four cases to consider:

$\langle 2\rangle$A. *StepA* : This is a step of the outer PCR. By invariant *IndexInvA* it suffices to assume there exists some $i \in \mathit{ItA}(x_1)$ such that

$$P1(I_0, i)\ \vee\ C1_{ini}(I_0, i)\ \vee\ C1_{end}(I_0, i)\ \vee\ R1(I_0, i)$$

[18] There is another consequence here that can be easily missed. The basic functions computing the upper/lower bounds for the iteration space of $\mathcal{B}$ can also depend on the input stream. So, to be more precise, maybe it would be more appropriate to treat them as partial functions just like the other basic functions with some knowledge of what are their dependencies (if any) on the input stream.

We proceed by cases on these actions.

$\langle 3 \rangle$A. $P1(I_0, i)$: A producer action corresponds directly with sub-action $A!P(I_0, i)$ of $A!Next$.

$\langle 3 \rangle$B. $C1_{ini}(I_0, i)$: The consumer of the outer PCR invokes the inner PCR writing on input $x_2^{I_0,i}$. In this case, we have that $A!vs1' = A!vs1$, i.e. $A!Spec$ stutters.

$\langle 3 \rangle$C. $C1_{end}(I_0, i)$: The (I_0, i)-th instance of the inner PCR finished computing on input $x_2^{I_0,i}$. The following holds in this action:

$$\neg wrt(c_1^{I_0,i}) \tag{5.13}$$
$$\wedge \; wrt(x_2^{I_0,i}) \tag{5.14}$$
$$\wedge \; endB(I_0, i) \tag{5.15}$$
$$\wedge \; (c_1^{I_0,i})' = r_2^{I_0,i} \tag{5.16}$$

By 5.14, index I_0, i is a well defined instance of the inner PCR, thus $I_0, i \in WDIndexB$. Then, by invariant $CorrectnessB$, we have:

$$endB(I_0, i) \Rightarrow r_2^{I_0,i} = \mathcal{B}(x_2^{I_0,i}) \tag{5.17}$$

By invariant $CInv1A$ and 5.14 there exists a stream variable vp on range type T_{p_1} such that:

$$eqs(vp, p_1^{I_0}, depsA(x_1, Dep\_pc1, i)) \tag{5.18}$$
$$\wedge \; x_2^{I_0,i} = \langle x_1, vp, i \rangle \tag{5.19}$$

Now, we prove the result computed by the inner PCR coincides with the basic function f_cS:

$$\begin{array}{rcll}
(c_1^{I_0,i})' & = & r_2^{I_0,i} & \text{by } 5.16 \\
 & = & \mathcal{B}(x_2^{I_0,i}) & \text{by } 5.17 \text{ and } 5.15 \\
 & = & \mathcal{B}(\langle x_1, vp, i \rangle) & \text{by } 5.19 \\
 & = & f_cS(x_1, vp, i) & \text{by our substitution} \\
 & = & f_cS(x_1, p_1^{I_0}, i) & \text{by axiom } H\_fcSRelevance \text{ and } 5.18
\end{array}$$

This result proves that action $C1_{end}(I_0, i)$ corresponds to sub-action $A!C(I_0, i)$ of $A!Next$.

$\langle 3 \rangle$D. $R1(I_0, i)$: A reducer action corresponds directly with sub-action $A!R(I_0, i)$ of $A!Next$.

$\langle 2 \rangle$B. $StepB$: This is a step of the inner PCR. We have that $\langle in, vs1 \rangle' = \langle in, vs1 \rangle$, i.e. the outer PCR state does not change. Note that $A!vs$ is included in $vs1$, therefore $A!vs1' = A!vs1$.

$\langle 2 \rangle$C. $Done$: We have that $\langle in, vs1, vs2 \rangle' = \langle in, vs1, vs2 \rangle$, i.e. $Spec$ stutters. Note that $A!vs$ is included in $vs1$, therefore $A!vs1' = A!vs1$.

$\langle 2 \rangle$D. $\langle in, vs1, vs2 \rangle' = \langle in, vs1, vs2 \rangle$: Analogous to previous case.

$\square$

Chapter 6

Conclusions

If debugging is the process of removing software bugs, then programming must be the process of putting them in.

Edsger Dijkstra

We would like to start with a retrospective of our work. When we started with this thesis, our main objective was to build upon the work of Pérez and Yovine [26]. More specifically, as we explained at the introduction, our main concern was the functional correctness of high level PCR designs. Firstly, we were looking for some appropriate formal framework with associated tools on which we could base our work. We experimented a bit of concurrent design with Promela/Spin and TLA$^+$ during a semester, and we were also looking into some of the introductory examples of the K framework. We were particularly attracted by TLA$^+$ philosophy of expressing refinement by logical implication at the language level, which seemed simple and elegant. Of course, we did not have lots of experience with it at the time, so that was not the only selling point. But considering also the availability of tools for model checking and theorem proving, and the industry attention it gained in the last five years, it was enough for us to decide to stick with it during this thesis.

Initially, we started trying to formalize two of the most relevant features of FXML that where used originally in [26] to give semantics to PCRs, namely stream variables and data dependencies, and we were able to prove relations between different data dependencies. Soon, we learned about PlusCal [78], an algorithmic language that can be used to replace pseudo-code for sequential and concurrent algorithms. This tool translates something that looks like pseudo-code (and of which one might think it does not possess any formal meaning) to very structured and understandable TLA$^+$. It was designed by Lamport as a tool to teach rigorous algorithmic thinking, but received a good amount of industrial attention, for example the PGo project aims to produce Go programs from PlusCal [79].[1] Inspired by this, we opted for a more direct approach: instead of relying on FXML, we could just take from it the concepts we needed for the PCR pattern and express them directly in TLA$^+$. So, there was no intention to capture all of FXML semantics in TLA$^+$, as we really did not need all of it. We also saw this as a possible more direct path to generate correct parallel executable code, but out of the scope of the current thesis, of course.

We worked for some time on concrete PCR problems, treating them as study cases. Here we tried to be as systematic as possible, envisioning an automatic translator tool in the spirit of PlusCal.[2] At this stage, we noticed model checking was particularly helpful to validate quickly what we where doing. When model checking is simply not possible because of the state explosion problem, the simulation mode can be used as last resort. We were pleased to have the opportunity to present our work in progress at the TLA$^+$ Community Event that was held (virtually) in October 2020 as a satellite of the DISC 2020 Conference. The abstract can be seen in [80].

Until that moment, we had not written any formal proof of correctness, we relied only on model checking. So we started to put TLAPS to serious use on some concrete examples, and after some work intuition suggested that proofs for different concrete examples would be almost identical, were it not by the fact that the mathematical functions used at the time where chosen in an ad-hoc way. We wanted to seek a more uniform methodology of proof. At this point, we had enough confidence to stop worrying about concrete examples and to try to abstract away of the concrete basic functions. So, we identified, first informally, abstract PCR models along the mathematical functions characterizing their functional behaviour. The idea of representing these functions as a composition of *stream functions* allowed us to capture in a single formula any possible dependency be-

[1]In fact, the Amazon report on TLA$^+$ notes that some of their engineers feels more productive on PlusCal than with raw TLA$^+$.

[2]Actually, we started its development using OCaml, but we realized it was too much work to be sustained in a short time, so it was abandoned.

tween the components without need to mention them explicitly in the formula itself. For the formalization in TLA$^+$, the mental translation from the dependence graphs induced by the data dependencies to the transition systems described by temporal formulas was very intuitive. Finally, with the formalization of the abstract PCR models, operational and functional, we now could carry out general proofs for entire classes of PCRs instead of just concrete PCRs. In particular, we could prove a chain of refinements where a PCR is represented initially as a mathematical function and is further implemented by PCRs adding more parallelism but preserving functional equivalence.

We want to stress that even if theorem proving is the only way to prove general results, like we have done, for a possible practical setting where one is concerned with a concrete PCR for a specific problem, model checking is surely more cost-effective. In particular, note that instead of using the standard correctness property as we had defined it, we may want to check correctness with respect to an alternative custom solution formula (see the *CorrectnessAlt* formulas in the concrete specifications at Appendix D). This is worthwhile for model checking purposes, but for theorem proving it requires its own formal proof.

Now, we make some observations with regard to what has been done and possible future work:

- We have not covered all the PCR features. One of them is the the **`feedbackloop`** extension. In the context of task-based parallelism, this corresponds to the *workpile pattern*, where an instance of a task can generate more instances and add them to a pile of tasks to be done.
  Other interesting feature is the implementation of *eureka* computations. This makes possible early termination, which is useful for search or optimization problems (e.g. return only the first solution on the NQueens problem). An eureka condition, say *cnd*, is made part of to the reducer syntax as follows:

  $$\texttt{\textbf{reduce}}\ \mathit{cnd}\ \otimes\ (v_0\ x)\ (f_r\ x\ p\ c_1\ \ldots\ c_k)$$

  We modeled eureka conditions for some time, as it is relatively easy. The problem was that the concurrent semantics of the PCR is non-deterministic, and if there is an eureka condition present then we cannot assert correctness as usual, because the

PCR output will not necessarily match the output of the mathematical function. For now, the only reasonable idea we have is that in the context of eureka computations the correctness assertion could be relaxed to state a containment condition and not an equality, i.e. that the output computed by the PCR is contained in all the possible results. But we have not investigated how this could be formalized in general.
Finally, in this thesis we limited ourselves to finite iteration spaces, which seems enough for most use cases, but maybe there are compelling use cases for infinite (i.e. unbounded) iteration spaces.[3]

- We have not finished all the formal proofs. In particular, we have not formally dealt with termination, which is a kind of liveness property, because of a limitation of TLAPS. However, we believe the next release of TLAPS will have support for liveness proofs, so it will be possible to deal formally with termination in the near future.
  It should be noted that formal proofs for divide and conquer and the iterative schemes are likely to be harder than the rest. Surely, well founded induction with some additional assumptions are needed to justify the mathematical functions are well defined. We add that at some time we had a formal proof for the partial correctness of the divide and conquer scheme —the reason we never mentioned it is because it was done on a simpler specification than what we presented here. We need to rewrite it for the current version.

- Currently, we have enough confidence to know that a tool can be written to translate high level PCR designs to TLA$^+$. For this, it would be productive if we could reuse parts of the PlusCal translator or maybe even extend it somehow. In this way, we would not need a host language to define the basic functions. About the possibility of generating correct parallel executable code, it seems plausible if we can leverage on a project like PGo. For the best of our knowledge, research in code synthesis from TLA$^+$ specifications is in its infancy.

[3]As an aside, eureka conditions together with infinite iteration spaces allow a basic PCR to express the **`iterate`** construct so that it is no longer a primitive extension. The possibility of an unbounded search is reminiscent of role of the minimization operator μ whose addition to the class of recursive primitive functions results in a turing complete model.

- In this thesis we have made the tacit assumption that one can express the solution of the problem in exactly the mathematical form associated to the abstract PCR models. Of course, it may not be obvious how to put the solution in that form to obtain a PCR. This is an interesting problem in the area of program derivation, something that initiated with Dijkstra [47] in the structured programming paradigm and was then taken to the functional programming paradigm by Bird [76], as in the Bird–Meertens formalism, where both functions and programs live at the same level. Bird showed how one can start with an inefficient functional solution assumed as obviously correct and this can be transformed by equational reasoning to a simple map-reduce expression which have an intuitive parallel interpretation and an algebraic significance (an homomorphism between algebraic structures). At least for basic PCRs, our mathematical functions used in the correctness assertions are just that, a reduce operation (the reducer combiner $\otimes$) over a composition of maps (the producer and the consumers transforming the iteration space) that can be simplified in just one map. But we are oversimplifying a bit here, because the producer is not in general just a map, it can depend on its past values. In general, one may need to start with an obvious mathematical definition and massage it somehow to obtain a functional PCR form.

In summary, we think this thesis contributes to the state of the art in formal refinement of parallel programs from abstract models, especially starting off from an alternative characterization of the general PCR pattern, and utilizing the TLA$^+$ framework. We hope to have shown this as an interesting line of research within the general quest for quality programming.

References

[1] P. Tendulkar, "Mapping and Scheduling on Multi-core Processors using SMT Solvers," Ph.D. dissertation, Universite de Grenoble I - Joseph Fourier, 2014.

[2] M. McCool, A. D. Robinson, and J. Reinders, *Structured Parallel Programming. Patterns for Efficient Computation.* Burlington, MA, USA: Morgan Kaufmann, 2012.

[3] R. W. Floyd, "Assigning meaning to programs," in *Proceedings of Symposium on Applied Mathematics*, vol. 14, 1967, pp. 19–32.

[4] S. Merz and H. Vanzetto, "Automatic Verification of TLA$^+$ Proof Obligations with SMT Solvers," in *LPAR*, 2012, pp. 289–303.

[5] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[6] N. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. Hu, M. Irwin, M. Kandemir, and V. Narayanan, "Leakage current: Moore's law meets static power," *Computer*, vol. 36, no. 12, pp. 68–75, 2003.

[7] Intel, "High-k and Metal Gate Transistor Research," [Online]. Available: https://www.intel.com/pressroom/kits/advancedtech/doodle/ref_HiK-MG/high-k.htm. Accessed on: Aug. 24, 2021.

[8] G. Pérez and S. Yovine, "Formal specification and implementation of an automated pattern-based parallel-code generation framework," *STTT*, vol. 21, no. 2, p. 183–202, 2017.

[9] S. Yovine, I. Assayad, F. Defaut, M. Zanconi, and A. Basu, “Formal approach to derivation of concurrent implementations in software product lines,” *Algebra for Parallel and Distributed Processing*, pp. 359–401, 2008.

[10] Intel® Cnc, “Intel Concurrent Collections for C++,” [Online]. Available: https://icnc.github.io/. Accessed on: Aug. 2, 2021.

[11] L. Snyder, “Type architectures, shared memory, and the corollary of modest potential,” in *Annual Review of Computer Science*, vol. 1, 1986, pp. 289–317.

[12] G. M. Amdahl, “Validity of the single-processor approach to achieving large scale computing capabilities,” *Proceedings of the American Federation of Information Processing Societies Spring Joint Computer Conference*, p. 483–485, 1967.

[13] W. H. Ware, “The ultimate computer,” *IEEE Spectrum*, vol. 9, no. 3, p. 84–91, 1972.

[14] J. L. Gustafson, “Reevaluating Amdahl’s law,” *Communications of the ACM*, vol. 31, p. 532–533, 1988.

[15] B. Schmidt, J. Gonzalez-Dominguez, C. Hundt, and M. Schlarb, *Parallel Programming. Concepts and Practice.* Burlington, MA, USA: Morgan Kaufmann, 2017.

[16] R. D. Blumofe and C. E. Leiserson, “Scheduling multithreaded computations by work stealing,” *Journal of the ACM*, vol. 46, no. 5, pp. 720–748, 1999.

[17] A. Aiken, U. Banerjee, A. Kejariwal, and A. Nicolau, *Instruction Level Parallelism. Professional Computing*, 1st ed. Boston, MA, USA: Springer, 2016.

[18] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” in *OSDI’04: Sixth Symposium on Operating System Design and Implementation*, 2004, pp. 137–150.

[19] J. Hennessy and D. Patterson, *Computer Architecture. A Quantitative Approach*, 6th ed. Burlington, MA, USA: Morgan Kaufmann, 2017.

[20] Y. H. T. G. Mattson and A. E. Koniges, *The OpenMP Common Core. Making OpenMP Simple Again.* Cambridge, MA, USA: MIT Press, 2019.

[21] Y.-F. Chen, "Commutativity of reducers," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2015, pp. 131–146.

[22] T. Xiao, J. Zhang, H. Zhou, Z. Guo, S. McDirmid, W. Lin, W. Chen, and L. Zhou, "Nondeterminism in mapreduce considered harmful? an empirical study on non-commutative aggregators in mapreduce programs," *36th International Conference on Software Engineering, ICSE Companion 2014 - Proceedings*, p. 44–53, 05 2014.

[23] T. S. W. R. L. Graham and J. M.Squyres, "Open MPI: A Flexible High Performance MPI," Parallel Processing and Applied Mathematics. LNCS 3911. Springer, 2006.

[24] MPI_Op_create documentation, [Online]. Available: https://www.open-mpi.org/doc/v4.0/man3/MPI_Op_create.3.php. Accessed on: Feb. 15, 2021.

[25] The Java™ Tutorials, "Aggregate Operations: Reduction," [Online]. Available: https://docs.oracle.com/javase/tutorial/collections/streams/reduction.html. Accessed on: Feb. 3, 2021.

[26] G. Pérez, "Especificación, diseño e implementación de un entorno de programación concurrente basado en patrones." Ph.D. dissertation, Universidad de Buenos Aires, 2018.

[27] P. Kogge and H. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Transactions on Computers*, vol. C-22, no. 8, pp. 786–792, 1973.

[28] Wikipedia, "Eight queens puzzle," [Online]. Available: https://en.wikipedia.org/wiki/Eight_queens_puzzle. Accessed on: Jul. 23, 2021.

[29] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: The MIT Press, 2009.

[30] L. Lamport, *Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers*. Redwood City, CA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[31] S. Merz, "Formal specification and verification," in *Concurrency: the Works of Leslie Lamport*, 2019, vol. 29, pp. 103–129.

[32] S. Merz, "On the Logic of TLA$^+$," *Computers and Informatics*, vol. 22, pp. 351–379, 2003.

[33] TLA$^+$ in Practice and Theory, [Online]. Available: https://pron.github.io/posts/tlaplus_part1. Accessed on: Oct. 1, 2020.

[34] L. Lamport, "The Temporal Logic of actions," *ACM Trans. Program. Lang. Syst.*, p. 872–923, 1994.

[35] A. Pnueli, "The Temporal Logic of programs," *18th Annual Symposium on Foundations of Computer Science*, pp. 46–57, 1977.

[36] L. Lamport, "Specifying Concurrent Systems with TLA$^+$," *Calculational System Design*, pp. 183–247, 1999.

[37] Y. Yu, P. Manolios, and L. Lamport, "Model Checking TLA$^+$ Specifications," in *Correct Hardware Design and Verification Methods*, 1999, pp. 54–66.

[38] D. Cousineau, D. Doligez, L. Lamport, S. Merz, D. Ricketts, and H. Vanzetto, "TLA$^+$ Proofs," in *FM 2012: Formal Methods*, vol. 7436, 2012, pp. 147–154.

[39] I. Konnov, J. Kukovec, and T.-H. Tran, "TLA$^+$ Model Checking made symbolic," *Proc. ACM Program. Lang.*, vol. 3, p. 1–30, 2019.

[40] M. Kuppe, L. Lamport, and D. Ricketts, "The TLA$^+$ Toolbox," *ArXiv*, vol. abs/1912.10633, pp. 50–62, 2019.

[41] C. Jones, "The early search for tractable ways of reasoning about programs," *IEEE Annals of the History of Computing*, vol. 25, no. 2, pp. 26–49, 2003.

[42] D. Hilbert and W. Ackermann, *Grundzüge der Theoretischen Logik.* Boston, MA, USA: Springer, 1928.

[43] A. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. s2-42, no. 1, p. 230–265, 1937.

[44] A. Church, "An unsolvable problem of elementary number theory," *American Journal of Mathematics*, vol. 58, p. 345, 1936.

[45] Shtetl-Optimized, "Rosser's Theorem via Turing machines," [Online]. Available: https://www.scottaaronson.com/blog/?p=710. Accessed on: Jul. 8, 2021.

[46] H. Goldstine and J. Neumann, *Planning and Coding of Problems for an Electronic Computing Instrument*, ser. Report on the mathematical and logical aspects of an electronic computing instrument. Princeton, NJ, USA: Institute for Advanced Study, 1947.

[47] E. W. Dijkstra, "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," *Communications of the ACM*, vol. 18, no. 8, p. 453–457, 1975.

[48] S. Owicki and D. Gries, "Verifying Properties of Parallel Programs: An Axiomatic Approach," *Communications of the ACM*, vol. 19, no. 5, p. 279–285, 1976.

[49] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, p. 666–677, 1978.

[50] B. J. Copeland, "Arthur Prior," in *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed. Metaphysics Research Lab, Stanford University, 2020.

[51] A. Pnueli, "The Temporal Semantics of Concurrent Programs," in *Proceedings of the International Sympoisum on Semantics of Concurrent Computation*, 1979, p. 1–20.

[52] V. Goranko and A. Rumberg, "Temporal Logic," in *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed. Metaphysics Research Lab, Stanford University, 2021.

[53] E. M. Clarke and E. A. Emerson, "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic," in *Logic of Programs, Workshop*, 1981, p. 52–71.

[54] J.-P. Queille and J. Sifakis, “Specification and Verification of Concurrent Systems in CESAR,” in *Proceedings of the 5th Colloquium on International Symposium on Programming*, 1982, p. 337–351.

[55] Wikipedia, “Z notation,” [Online]. Available: https://en.wikipedia.org/wiki/Z_notation. Accessed on: Aug. 2, 2021.

[56] Event-B and the Rodin Platform, [Online]. Available: http://www.event-b.org/. Accessed on: Aug. 2, 2021.

[57] Mizar Home Page, [Online]. Available: http://mizar.org/. Accessed on: Aug. 1, 2021.

[58] Wikipedia, “Mizar system,” [Online]. Available: https://en.wikipedia.org/wiki/Mizar_system. Accessed on: Aug. 2, 2021.

[59] L. Lamport and L. C. Paulson, “Should your specification language be typed,” *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 3, p. 502–526, 1999.

[60] C. Newcombe, “Why Amazon Chose TLA$^+$,” in *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, vol. 8477, 2014, pp. 25–39.

[61] L. Lamport, “Proving the Correctness of Multiprocess Programs,” *IEEE Transactions on Software Engineering*, vol. SE-3, no. 2, pp. 125–143, 1977.

[62] B. Alpern and F. Schneider, “Recognizing Safety and Liveness,” *Distributed Computing*, vol. 2, no. 3, pp. 117–126, 09 1987.

[63] L. Lamport and F. Schneider, “Verifying Hyperproperties with TLA,” [Online]. Available: https://lamport.azurewebsites.net/pubs/hyper2.pdf. Accessed on: Aug. 1, 2021.

[64] M. Abadi and L. Lamport, “The existence of refinement mappings,” in *Proceedings. Third Annual Symposium on Logic in Computer Science*, 1988, pp. 165–175.

[65] L. Lamport, “TLA$^+$ Version 2. A Preliminary Guide,” [Online]. Available: https://lamport.azurewebsites.net/tla/tla2-guide.pdf. Accessed on: Jul. 7, 2021.

[66] L. Lamport, "Types are not harmless," [Online]. Available: http://research.microsoft.com/en-us/um/people/lamport/tla/types.ps.Z. Accessed on: Aug. 1, 2021.

[67] G. Gonthier and L. Lamport, "Recursive Operator Definitions," [Online]. Available: https://lamport.azurewebsites.net/pubs/recursive-ops.pdf. Accessed on: Aug. 1, 2021.

[68] Wikipedia, "Z3 Theorem Prover," [Online]. Available: https://en.wikipedia.org/wiki/Z3_Theorem_Prover. Accessed on: Aug. 2, 2021.

[69] About CVC4, [Online]. Available: https://cvc4.github.io/. Accessed on: Aug. 9, 2021.

[70] The Zenon automatic theorem prover, [Online]. Available: http://zenon-prover.org/. Accessed on: Aug. 2, 2021.

[71] A. Defourné, "Improving Automation for Higher-Order Proof Steps," in *Frontiers of Combining Systems*, vol. 12941, 2021, pp. 139–153.

[72] Youtube, "Reasoning about the TLA$^+$ operator ENABLED within TLAPS," [Online]. Available: https://www.youtube.com/watch?v=ilCrIRUbDHI. Accessed on: Aug. 2, 2021.

[73] Model Values and Symmetry, [Online]. Available: https://tla.msr-inria.inria.fr/tlatoolbox/doc/model/model-values.html. Accessed on: Aug. 15, 2021.

[74] P. Godefroid and P. Wolper, "Using partial orders for the efficient verification of deadlock freedom and safety properties," *Formal Methods in System Design*, vol. 2, pp. 149–164, 1993.

[75] Github, "TLA$^+$ snippets, operators and modules contributed and curated by the TLA$^+$ community," [Online]. Available: https://github.com/tlaplus/CommunityModules. Accessed on: Mar. 12, 2021.

[76] R. Bird, "Lectures on Constructive Functional Programming," *Constructive Methods in Computing Science*, vol. 55, pp. 151–217, 1988.

[77] L. Lamport, “Using TLC to Check Inductive Invariance,” [Online]. Available: https://lamport.azurewebsites.net/tla/inductive-invariant.pdf. Accessed on: Aug. 1, 2021.

[78] L. Lamport, “The pluscal algorithm language,” *Theoretical Aspects of Computing-ICTAC*, vol. 5684, pp. 36–60, January 2009.

[79] GitHub, “PGo is a source to source compiler from Modular PlusCal specs into Go programs,” [Online]. Available: https://github.com/UBC-NSS/pgo. Accessed on: Oct. 3, 2020.

[80] J. E. Solsona and S. Yovine, “TLA$^+$ specification of PCR parallel programming pattern,” [Online]. Available: https://conf.tlapl.us/2020/10-Yovine_and_Solsona-TLA_+_specification_of_PCR_parallel_programming_pattern.pdf. Accessed on: Oct. 2, 2020.

[81] GitHub, “Jupyter kernel for TLA$^+$,” [Online]. Available: https://github.com/kelvich/tlaplus_jupyter. Accessed on: Sept. 20, 2021.

[82] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[83] G. Brightwell and P. Winkler, “Counting Linear Extensions is #P-Complete,” in *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, 1991, p. 175–181.

[84] D. Hansen and M. Leuschel, “Translating TLA$^+$ to B for Validation with ProB,” in *Integrated Formal Methods*, vol. 732, 2012, pp. 24–38.

Appendix A

Specification of algebraic concepts and their theorems

A.1 Abstract algebra

```
──────────────────── MODULE AbstractAlgebra ────────────────────
    This module formally defines some basic algebraic structures.
EXTENDS TLAPS
────────────────────────────────────────────────────────────────
    A Magma consists of a set D equipped with a single binary operation ⊗ that is closed on D .
Magma(D, _ ⊗ _) ≜
  ∀ x, y ∈ D : x ⊗ y ∈ D

    A SemiGroup is a Magma where operation ⊗ : D × D → D satisfies Associativity.
SemiGroup(D, _ ⊗ _) ≜
  ∧ Magma(D, ⊗ )
  ∧ ∀ x, y, z ∈ D : (x ⊗ y) ⊗ z = x ⊗ (y ⊗ z)

    A Monoid is a SemiGroup with identity element for operation ⊗ : D × D → D .
Monoid(D, c, _ ⊗ _) ≜
  ∧ SemiGroup(D, ⊗ )
  ∧ ∃ e ∈ D : ∀ x ∈ D : ∧ e = c
                        ∧ x ⊗ e = x
                        ∧ e ⊗ x = x

Monoid2(D, _ ⊗ _) ≜
  ∧ SemiGroup(D, ⊗ )
  ∧ ∃ e ∈ D : ∀ x ∈ D : ∧ x ⊗ e = x
                        ∧ e ⊗ x = x

    An Abelian Monoid is a Monoid where operation ⊗ : D × D → D satisfies Commutativity.
AbelianMonoid(D, c, _ ⊗ _) ≜
  ∧ Monoid(D, c, ⊗ )
  ∧ ∀ x, y ∈ D : x ⊗ y = y ⊗ x

AbelianMonoid2(D, _ ⊗ _) ≜
  ∧ Monoid2(D, ⊗ )
  ∧ ∀ x, y ∈ D : x ⊗ y = y ⊗ x
────────────────────────────────────────────────────────────────
    The identity element in a Monoid is unique.
THEOREM MonoidUniqueIdentity ≜
  ASSUME NEW D, NEW _ ⊗ _,
         Monoid2(D, ⊗ ),
         NEW e1, NEW e2,
         ∀ x : x ⊗ e1 = x ∧ e1 ⊗ x = x,
         ∀ x : x ⊗ e2 = x ∧ e2 ⊗ x = x
  PROVE  e1 = e2
  BY Z3 DEF Monoid2
────────────────────────────────────────────────────────────────
```

A.2 Operation on monoid structure

```
──────────────────── MODULE MonoidBigOp ────────────────────
```

Let $(D, Id, \otimes)$ be a *Monoid* structure. This module defines the extension of binary operation $\otimes$ over finite intervals.

See relevant theorems in module *MonoidBigOpThms*.

```
EXTENDS AbstractAlgebra

LOCAL INSTANCE Naturals
────────────────────────────────────────────────────────────
```

This module is parameterized by the signature $(D, Id, \otimes)$.

Is taken as an assumption that $(D, Id, \otimes)$ obeys the laws of a *Monoid*.

```
CONSTANTS D,          Domain
          Id,         Special constant symbol
          _ ⊗ _       Binary operator

AXIOM Algebra ≜ Monoid(D, Id, ⊗ )
────────────────────────────────────────────────────────────
```

The operation $\otimes$ over (non empty) interval $m..n$ defined as a recursive function

$$\bigotimes_{m}^{n} f \ : \ m..n \rightarrow D$$

```
bigOp(m, n, f(_)) ≜
  LET recDef[i ∈ m .. n] ≜
         IF i = m
          THEN f(m)
          ELSE recDef[i − 1] ⊗ f(i)
  IN  recDef
```

When $m > n$, and thus interval $m..n$ is empty, it is assumed the result is the monoid identity Id . The result of the operation $\otimes$ over interval $m..n$ is denoted by

$$\bigotimes_{i=m}^{n} f(i) \ \triangleq \ \left(\bigotimes_{m}^{n} f \right) [n]$$

```
BigOp(m, n, f(_)) ≜
  IF m ≤ n
   THEN bigOp(m, n, f)[n]
   ELSE Id
```

A convenient abbreviation to deal with holes in the interval:

$$\bigotimes_{\{i \in m..n \, : \, P(i)\}} f(i) \ \triangleq \ \bigotimes_{i=m}^{n} (P(i) \rightarrow f(i), \ Id)$$

```
BigOpP(m, n, P(_), f(_)) ≜ BigOp(m, n, LAMBDA i : IF P(i) THEN f(i) ELSE Id)
────────────────────────────────────────────────────────────
```

A.3 Theorems for monoid structures

```
──────────────────── MODULE MonoidBigOpThms ────────────────────
    This module states theorems for module MonoidBigOp.
    See proofs in module MonoidBigOpThms_proofs.

EXTENDS MonoidBigOp

LOCAL INSTANCE NaturalsInduction

────────────────────────────────────────────────────────────────

    The following theorems asserts that our definition of bigOp is well defined, that is, there exists a function matching the
    recursive definition.
LEMMA bigOpDefConclusion ≜
  ASSUME NEW m ∈ Nat,
         NEW n ∈ Nat,
         NEW f(_)
  PROVE  FiniteNatInductiveDefConclusion(bigOp(m, n, f), f(m), LAMBDA v, i : f(i) ⊗ v, m, n)

THEOREM bigOpDef ≜
  ASSUME NEW m ∈ Nat,
         NEW n ∈ Nat,
         NEW f(_),
         NEW i ∈ m . . n
  PROVE  bigOp(m, n, f)[i] = IF i = m THEN f(m) ELSE bigOp(m, n, f)[i − 1] ⊗ f(i)

    The bigOp function type.
    $\bigotimes_{m}^{n} f \ : \ m..n \rightarrow D$
THEOREM bigOpType ≜
  ASSUME NEW m ∈ Nat,
         NEW n ∈ Nat,
         NEW f(_), ∀ x ∈ m . . n : f(x) ∈ D,
         m ≤ n
  PROVE  bigOp(m, n, f) ∈ [m . . n → D]

    The base case for bigOp function:
    $\bigotimes_{m}^{n} f\,[i] = f(m)$  when  $i = m$
LEMMA bigOpBaseCase ≜
  ASSUME NEW m ∈ Nat,
         NEW n ∈ Nat,
         NEW i  ∈ m . . n, i = m,
         NEW f(_)
  PROVE  bigOp(m, n, f)[i] = f(m)

    The recursive step for bigOp function:
    $\bigotimes_{m}^{n} f\,[i] \ = \ \bigotimes_{m}^{n} f\,[i-1] \oplus f(i)$  when  $i \neq m$
LEMMA bigOpRecStep ≜
  ASSUME NEW m ∈ Nat,
         NEW n ∈ Nat,
         NEW i  ∈ m . . n, i ≠ m,
         NEW f(_)
  PROVE $bigOp(m, n, f)[i] = bigOp(m, n, f)[i-1] \otimes f(i)$
```

The *BigOp* result type.

$$\bigotimes_{i=m}^{n} f(i) \in D$$

```
THEOREM $BigOpType \triangleq$
 ASSUME NEW $m \in Nat$,
        NEW $n \in Nat$,
        NEW $f(\_)$, $\forall x \in m \,..\, n : f(x) \in D$
  PROVE $BigOp(m, n, f) \in D$
----------------------------------------------------------------
```

The (assumed) monoid laws for binary operator $\otimes$.

```
PROPOSITION $OpClosure \triangleq$
  $\forall x, y \in D : x \otimes y \in D$

PROPOSITION $OpAssociativity \triangleq$
  $\forall x, y, z \in D : (x \otimes y) \otimes z = x \otimes (y \otimes z)$

PROPOSITION $OpIdentity \triangleq$
  $\exists e \in D : \forall x \in D :$ $\land\, e = Id$
                           $\land\, x \otimes Id = x$
                           $\land\, Id \otimes x = x$

THEOREM $OpUniqueIdentity \triangleq$
  ASSUME NEW $e1$, NEW $e2$,
         $\forall x : x \otimes e1 = x \land e1 \otimes x = x$,
         $\forall x : x \otimes e2 = x \land e2 \otimes x = x$
  PROVE  $e1 = e2$

PROPOSITION $OpIdentityLeft \triangleq$
  $\forall x \in D : Id \otimes x = x$
  BY $OpIdentity$

PROPOSITION $OpIdentityRight \triangleq$
  $\forall x \in D : x \otimes Id = x$
  BY $OpIdentity$
----------------------------------------------------------------
```

To evaluate the *bigOp* function on $m..n$ starting from any $i \in m..n$ is the same as evaluating on subinterval $m..i$ starting from i . That is:

$$\bigotimes_{m}^{n} f\,[i] \;=\; \bigotimes_{m}^{i} f\,[i]$$

This fact is used in some proofs. For example, when $i \neq m$ in the recursive step we have:

$$\bigotimes_{m}^{n} f\,[i] \;=\; \bigotimes_{m}^{n} f\,[i-1] \otimes f(i) \;=\; \bigotimes_{m}^{n-1} f\,[i-1] \otimes f(i)$$

```
THEOREM $BasicNotationalEq \triangleq$
  ASSUME NEW $m \in Nat$,
         NEW $n \in Nat$,
         NEW $l \in m \,..\, n$,
         NEW $f(\_)$, $\forall x \in m \,..\, n : f(x) \in D$,
         $m \leq n$
  PROVE  $bigOp(m, n, f)[l] = bigOp(m, l, f)[l]$
```

BigOp over constant identity.

$$\bigotimes_{i=m}^{n} Id = Id$$

```
THEOREM $OpIdentityExt \triangleq$
  ASSUME NEW $m \in Nat$,
         NEW $n \in Nat$
  PROVE  $BigOp(m, n,$ LAMBDA $i : Id) = Id$
```

If $f(i) = g(i)$ for all $i \in m..n$ then:

$$\bigotimes_{i=m}^{n} f(i) \;=\; \bigotimes_{i=m}^{n} g(i)$$

```
THEOREM $FunctionEq \triangleq$
  ASSUME NEW $m \in Nat$,
         NEW $n \in Nat$,
         NEW $f(\_), \forall i \in m \,..\, n : f(i) \in D$,
         NEW $g(\_), \forall i \in m \,..\, n : g(i) \in D$,
         $\forall i \in m \,..\, n : f(i) = g(i)$
  PROVE  $BigOp(m, n, f) = BigOp(m, n, g)$
```

Operation over unitary interval is trivial.

$$\bigotimes_{i=n}^{n} f(i) \;=\; f(n)$$

```
THEOREM $UnitInterval \triangleq$
  ASSUME NEW $n \in Nat$,
         NEW $f(\_)$
  PROVE  $BigOp(n, n, f) = f(n)$
```

For any $k \in m..n$, operation can be split at k .

$$\bigotimes_{i=m}^{n} f(i) \;=\; \bigotimes_{i=m}^{k} f(i) \;\otimes\; \bigotimes_{i=k+1}^{n} f(i)$$

```
THEOREM $SplitUp \triangleq$
  ASSUME NEW $m \in Nat$,
         NEW $n \in Nat$,
         NEW $f(\_), \forall x \in m \,..\, n : f(x) \in D$,
         NEW $k \in m \,..\, n$,
         $m \leq n$
  PROVE  $BigOp(m, n, f) = BigOp(m, k, f) \otimes BigOp(k + 1, n, f)$
```

For any $k \in m..n$, operation can be split at $k - 1$.

$$\bigotimes_{i=m}^{n} f(i) \;=\; \bigotimes_{i=m}^{k-1} f(i) \;\otimes\; \bigotimes_{i=k}^{n} f(i)$$

```
THEOREM $SplitDown \triangleq$
  ASSUME NEW $m \in Nat$,
         NEW $n \in Nat$,
         NEW $f(\_), \forall x \in m \,..\, n : f(x) \in D$,
         NEW $k \in m + 1 \,..\, n$,
         $m \leq n$
  PROVE  $BigOp(m, n, f) = BigOp(m, k - 1, f) \otimes BigOp(k, n, f)$
```

Extraction of the m-indexed (first) term.

$$\bigotimes_{i=m}^{n} f(i) \;=\; f(m) \otimes \bigotimes_{i=m+1}^{n} f(i)$$

THEOREM $\mathit{SplitFirst} \triangleq$
  ASSUME NEW $m \in \mathit{Nat}$,
         NEW $n \in \mathit{Nat}$,
         NEW $f(\_)$, $\forall x \in m \,..\, n : f(x) \in D$,
         $m \leq n$
  PROVE $\mathit{BigOp}(m, n, f) = f(m) \otimes \mathit{BigOp}(m+1, n, f)$

Extraction of the n-indexed (last) term.

$$\bigotimes_{i=m}^{n} f(i) \;=\; \bigotimes_{i=m}^{n-1} f(i) \otimes f(n)$$

THEOREM $\mathit{SplitLast} \triangleq$
  ASSUME NEW $m \in \mathit{Nat}$,
         NEW $n \in \mathit{Nat}$,
         NEW $f(\_)$, $\forall x \in m \,..\, n : f(x) \in D$,
         $m \leq n$
  PROVE $\mathit{BigOp}(m, n, f) = \mathit{BigOp}(m, n-1, f) \otimes f(n)$

When $m > n$, and thus inverval $m..n$ is empty, it is assumed the result is the monoid identity Id .

COROLLARY $\mathit{EmptyIntvAssumpP} \triangleq$
  ASSUME NEW $m \in \mathit{Int}$,
         NEW $n \in \mathit{Int}$,
         NEW $P(\_)$,
         NEW $f(\_)$,
         $n < m$
  PROVE $\mathit{BigOpP}(m, n, P, f) = \mathit{Id}$

The BigOpP result type.

$$\bigotimes_{\{i \in m..n \,:\, P(i)\}} f(i) \;\in D$$

COROLLARY $\mathit{BigOpPType} \triangleq$
  ASSUME NEW $m \in \mathit{Nat}$,
         NEW $n \in \mathit{Nat}$,
         NEW $P(\_)$, $\forall i \in m \,..\, n : P(i) \in$ BOOLEAN ,
         NEW $f(\_)$, $\forall x \in \{i \in m \,..\, n : P(i)\} : f(x) \in D$
  PROVE $\mathit{BigOpP}(m, n, P, f) \in D$

BigOpP over constant identity.

$$\bigotimes_{\{i \in m..n \,:\, P(i)\}} \mathit{Id} \;= \mathit{Id}$$

COROLLARY $\mathit{OpIdentityExtP} \triangleq$
  ASSUME NEW $m \in \mathit{Nat}$,
         NEW $n \in \mathit{Nat}$,
         NEW $P(\_)$, $\forall i \in m \,..\, n : P(i) \in$ BOOLEAN
  PROVE $\mathit{BigOpP}(m, n, P,$ LAMBDA $i : \mathit{Id}) = \mathit{Id}$

If $f(i) = g(i)$ for all $i \in m..n$ then:

$$\bigotimes_{\{i \in m..n \,:\, P(i)\}} f(i) \quad = \quad \bigotimes_{\{i \in m..n \,:\, P(i)\}} g(i)$$

```
COROLLARY FunctionEqP ≜
  ASSUME NEW m ∈ Nat,
         NEW n ∈ Nat,
         NEW P(_), ∀ i ∈ m . . n : P(i) ∈ BOOLEAN ,
         NEW f(_), ∀ i ∈ {j ∈ m . . n : P(j)} : f(i) ∈ D,
         NEW g(_), ∀ i ∈ {j ∈ m . . n : P(j)} : g(i) ∈ D,
         ∀ i ∈ {j ∈ m . . n : P(j)} : f(i) = g(i)
  PROVE BigOpP(m, n, P, f) = BigOpP(m, n, P, g)
```

For always false predicate P the result is identity.

$$\bigotimes_{\{i \in m..n \,:\, P(i)\}} f(i) \quad = Id$$

```
COROLLARY FalsePredicate ≜
 ASSUME NEW m ∈ Nat,
        NEW n ∈ Nat,
        NEW P(_), ∀ i ∈ m . . n : P(i) ∈ BOOLEAN ,
        NEW f(_), ∀ x ∈ {i ∈ m . . n : P(i)} : f(x) ∈ D,
        ∀ i ∈ m . . n : ¬P(i)
  PROVE BigOpP(m, n, P, f) = Id
```

For always true predicate P , collapse to basic definition.

$$\bigotimes_{\{i \in m..n \,:\, P(i)\}} f(i) \quad = \bigotimes_{i=m}^{n} f(i)$$

```
COROLLARY TruePredicate ≜
 ASSUME NEW m ∈ Nat,
        NEW n ∈ Nat,
        NEW P(_), ∀ i ∈ m . . n : P(i) ∈ BOOLEAN ,
        NEW f(_), ∀ x ∈ {i ∈ m . . n : P(i)} : f(x) ∈ D,
        ∀ i ∈ m . . n : P(i)
  PROVE BigOpP(m, n, P, f) = BigOp(m, n, f)
```

If $P(i) \equiv Q(i)$ for all $i \in m..n$ then:

$$\bigotimes_{\{i \in m..n \,:\, P(i)\}} f(i) \quad = \quad \bigotimes_{\{i \in m..n \,:\, Q(i)\}} f(i)$$

```
COROLLARY PredicateEq ≜
  ASSUME NEW m ∈ Nat,
         NEW n ∈ Nat,
         NEW P(_), ∀ i ∈ m . . n : P(i) ∈ BOOLEAN ,
         NEW Q(_), ∀ i ∈ m . . n : Q(i) ∈ BOOLEAN ,
         NEW f(_), ∀ i ∈ {j ∈ m . . n : P(j) ∧ Q(j)} : f(i) ∈ D,
         ∀ i ∈ m . . n : P(i) ≡ Q(i)
  PROVE BigOpP(m, n, P, f) = BigOpP(m, n, Q, f)
```

Operation over unitary interval is trivial.

$$\bigotimes_{\{i \in n..n \,:\, P(i)\}} f(i) \quad = \ (P(n) \rightarrow f(n),\ Id)$$

```
COROLLARY UnitIntervalP ≜
  ASSUME NEW n ∈ Nat,
         NEW P(_),
            NEW $f(\_)$
  PROVE $BigOpP(n, n, P, f) = \text{IF } P(n) \text{ THEN } f(n) \text{ ELSE } Id$
```

For any $k \in m..n$, operation can be split at k .

$$\bigotimes_{\{i \in m..n : P(i)\}} f(i) = \bigotimes_{\{i \in m..k : P(i)\}} f(i) \otimes \bigotimes_{\{i \in k+1..n : P(i)\}} f(i)$$

```
COROLLARY $SplitUpP \triangleq$
  ASSUME NEW $m \in Nat$,
         NEW $n \in Nat$,
         NEW $P(\_), \forall i \in m \,..\, n : P(i) \in \text{BOOLEAN}$ ,
         NEW $f(\_), \forall x \in \{i \in m \,..\, n : P(i)\} : f(x) \in D$,
         NEW $k \in m \,..\, n$,
         $m \leq n$
  PROVE $BigOpP(m, n, P, f) = BigOpP(m, k, P, f) \otimes BigOpP(k+1, n, P, f)$
```

For any $k \in m..n$, operation can be split at $k-1$.

$$\bigotimes_{\{i \in m..n : P(i)\}} f(i) = \bigotimes_{\{i \in m..k-1 : P(i)\}} f(i) \otimes \bigotimes_{\{i \in k..n : P(i)\}} f(i)$$

```
COROLLARY $SplitDownP \triangleq$
  ASSUME NEW $m \in Nat$,
         NEW $n \in Nat$,
         NEW $P(\_), \forall i \in m \,..\, n : P(i) \in \text{BOOLEAN}$ ,
         NEW $f(\_), \forall x \in \{i \in m \,..\, n : P(i)\} : f(x) \in D$,
         NEW $k \in m+1 \,..\, n$,
         $m \leq n$
  PROVE $BigOpP(m, n, P, f) = BigOpP(m, k-1, P, f) \otimes BigOpP(k, n, P, f)$
```

Extraction of the m-indexed (first) term.

$$\bigotimes_{\{i \in m..n : P(i)\}} f(i) = (P(m) \rightarrow f(m),\ Id) \otimes \bigotimes_{\{i \in m+1..n : P(i)\}} f(i)$$

```
COROLLARY $SplitFirstP \triangleq$
  ASSUME NEW $m \in Nat$,
         NEW $n \in Nat$,
         NEW $P(\_), \forall i \in m \,..\, n : P(i) \in \text{BOOLEAN}$ ,
         NEW $f(\_), \forall x \in \{i \in m \,..\, n : P(i)\} : f(x) \in D$,
         $m \leq n$
  PROVE $BigOpP(m, n, P, f) = (\text{IF } P(m) \text{ THEN } f(m) \text{ ELSE } Id) \otimes BigOpP(m+1, n, P, f)$
```

Extraction of the n-indexed (last) term.

$$\bigotimes_{\{i \in m..n : P(i)\}} f(i) = \bigotimes_{\{i \in m..n-1 : P(i)\}} f(i) \otimes (P(n) \rightarrow f(n),\ Id)$$

```
COROLLARY $SplitLastP \triangleq$
  ASSUME NEW $m \in Nat$,
         NEW $n \in Nat$,
         NEW $P(\_), \forall i \in m \,..\, n : P(i) \in \text{BOOLEAN}$ ,
         NEW $f(\_), \forall x \in \{i \in m \,..\, n : P(i)\} : f(x) \in D$,
         $m \leq n$
  PROVE $BigOpP(m, n, P, f) = BigOpP(m, n-1, P, f) \otimes \text{IF } P(n) \text{ THEN } f(n) \text{ ELSE } Id$
```

A.4 Operation on abelian monoid structure

MODULE *AbelianMonoidBigOp*

Let $(D, Id, \otimes)$ be an *Abelian* monoid structure. This module defines the extension of binary operation $\otimes$ over finite intervals.

As order does not matter, definition from module *MonoidBigOp* is reused and commutativity is postulated as an additional assumption.

See relevant theorems in module *AbelianMonoidBigOpThms*.

EXTENDS *AbstractAlgebra*, *MonoidBigOp*

AXIOM $\mathit{OpCommutativity} \triangleq \forall\, x, y \in D : x \otimes y = y \otimes x$

A.5 Theorems for abelian monoid structures

```
──────────────── MODULE AbelianMonoidBigOpThms ────────────────
```

This module states theorems for module *AbelianMonoidBigOp*.

See proofs in module *AbelianMonoidBigOpThms_proofs*.

```
EXTENDS AbelianMonoidBigOp, MonoidBigOpThms

LOCAL INSTANCE Naturals
───────────────────────────────────────────────────────────────
```

Distributivity (aka linearity).

$$\bigotimes_{i=m}^{n} f(i) \;\otimes\; \bigotimes_{i=m}^{n} g(i) \;=\; \bigotimes_{i=m}^{n} (f(i) \otimes g(i))$$

```
THEOREM Linearity ≜
  ASSUME NEW m ∈ Nat,
         NEW n ∈ Nat,
         NEW f(_), ∀ x ∈ m .. n : f(x) ∈ D,
         NEW g(_), ∀ x ∈ m .. n : g(x) ∈ D,
         m ≤ n
  PROVE BigOp(m, n, f) ⊗ BigOp(m, n, g) = BigOp(m, n, LAMBDA i : f(i) ⊗ g(i))
```

Skip index outside operation interval has no effect. That is, if $j \notin m..n$ then:

$$\bigotimes_{\{i \in m..n \,:\, i \neq j\}} f(i) \;=\; \bigotimes_{i=m}^{n} f(i)$$

```
THEOREM SkipOutOfBounds ≜
  ASSUME NEW m ∈ Nat,
         NEW n ∈ Nat,
         NEW f(_), ∀ x ∈ m .. n : f(x) ∈ D,
         NEW j, j ∉ m .. n
  PROVE BigOpP(m, n, LAMBDA i : i ≠ j, f) = BigOp(m, n, f)
```

For any $j \in m..n$, the j-indexed term can be extracted.

$$\bigotimes_{i=m}^{n} f(i) \;=\; \bigotimes_{\{i \in m..n \,:\, i \neq j\}} f(i) \;\otimes\; f(j)$$

```
THEOREM SplitRandom ≜
  ASSUME NEW m ∈ Nat,
         NEW n ∈ Nat,
         NEW f(_), ∀ i ∈ m .. n : f(i) ∈ D,
         NEW j ∈ m .. n,
         m ≤ n
  PROVE BigOp(m, n, f) = BigOpP(m, n, LAMBDA i : i ≠ j, f) ⊗ f(j)
```

For any $j \in m..n$ and predicate P for which $P(j)$ holds, the j-indexed term can be extracted.

$$\bigotimes_{\{i \in m..n \,:\, P(i)\}} f(i) \;=\; \bigotimes_{\{i \in m..n \,:\, P(i) \wedge i \neq j\}} f(i) \;\otimes\; f(j)$$

```
COROLLARY SplitRandomP ≜
  ASSUME NEW m ∈ Nat,
         NEW n ∈ Nat,
         NEW P(_), ∀ i ∈ m .. n : P(i) ∈ BOOLEAN ,
         NEW f(_), ∀ i ∈ {k ∈ m .. n : P(k)} : f(i) ∈ D,
         NEW j ∈ m .. n, P(j),
         m ≤ n
```

```
PROVE $BigOpP(m,\, n,\, P,\, f) = BigOpP(m,\, n,$ LAMBDA $i : P(i) \land i \neq j,\, f) \otimes f(j)$
```

Appendix B

Specification of abstract PCR models

B.1 Basic PCR

MODULE PCR_A

Basic *PCR*.

```
 ----------------------------------------------------------------
 fun fp(x,p,i) = ...              // fp : T x St(Tp) x N -> Tp
 fun fc(x,p,i) = ...              // fc : T x St(Tp) x N -> Tc
 fun fr(x,c,i) = ...              // fr : T x St(Tc) x N -> D

 dep p(i-k) -> p(i)
 dep p(i[+/-]k) -> c(i)
 dep c(i[+/-]k) -> r(i)

 lbnd A = \x. ...
 ubnd A = \x. ...
 prop A = \i. ...

 PCR A(x)                         // x \in T
   par
     p = produce fp x p
     c = consume fc x p
     r = reduce  Op id (fr x c)    // r \in D
----------------------------------------------------------------
```

EXTENDS $AbstractAlgebra$, $Naturals$, $Sequences$, $Bags$, $SeqUtils$, $ArithUtils$, TLC

PCR constants and variables

CONSTANTS $I0$, $pre(\_)$,
T, Tp, Tc, D,
id, $Op(\_, \_)$,
$lBnd(\_)$, $uBnd(\_)$, $prop(\_)$,
$fp(\_, \_, \_)$, $fc(\_, \_, \_)$, $fr(\_, \_, \_)$, $gp(\_, \_)$,
Dep_pp, Dep_pc, Dep_cr

VARIABLES in, X, p, c, r, rs

General definitions

$Undef \triangleq$ CHOOSE $x : x \notin$ UNION $\{T, Tp, Tc, D\}$

$wrt(v) \triangleq v \neq Undef$
$wrts(v, S) \triangleq \forall k \in S : wrt(v[k])$
$eqs(v1, v2, S) \triangleq \forall k \in S : wrt(v1[k]) \wedge v1[k] = v2[k]$

PCR definitions and assumptions

$Index \triangleq Seq(Nat)$
$Assig \triangleq Nat$
$St(R) \triangleq [Assig \rightarrow R \cup \{Undef\}]$
$WDIndex \triangleq \{I \in Index : wrt(X[I])\}$
$It(x) \triangleq \{i \in lBnd(x) \,..\, uBnd(x) : prop(i)\}$
$red(I, i) \triangleq rs[I][i]$
$end(I) \triangleq \forall i \in It(X[I]) : red(I, i)$

$deps(x, d, i) \triangleq$
$\quad \{i - k : k \in \{k \in d[1] : i - k \geq lBnd(x) \wedge prop(i - k)\}\}$
$\cup \quad \{i\}$

```
    $\cup \quad \{i + k : k \in \{k \in d[2] : i + k \leq uBnd(x) \land prop(i + k)\}\}$

AXIOM $H\_Type \triangleq$
    $\land\ I0 \quad \in Index$
    $\land\ \forall x \in T \quad : lBnd(x) \in Nat$
    $\land\ \forall x \in T \quad : uBnd(x) \in Nat$
    $\land\ \forall i \in Nat : prop(i) \in$ BOOLEAN
    $\land\ \forall x \in T \quad : pre(x) \in$ BOOLEAN
    $\land\ Dep\_pp \in (\text{SUBSET } (Nat \setminus \{0\})) \times (\text{SUBSET } \{\})$
    $\land\ Dep\_pc \in (\text{SUBSET } (Nat \setminus \{0\})) \times (\text{SUBSET } (Nat \setminus \{0\}))$
    $\land\ Dep\_cr \in (\text{SUBSET } (Nat \setminus \{0\})) \times (\text{SUBSET } (Nat \setminus \{0\}))$

AXIOM $H\_BFunType \triangleq$
  $\forall x \in T,\ i \in Assig :$
    $\land\ gp(x, i) \in Tp \cup \{Undef\}$
    $\land\ \forall vp \in St(Tp) : fp(x, vp, i) \in Tp \cup \{Undef\}$
    $\land\ \forall vp \in St(Tp) : fc(x, vp, i) \in Tc \cup \{Undef\}$
    $\land\ \forall vc \in St(Tc) : fr(x, vc, i) \in D \ \ \cup \{Undef\}$

AXIOM $H\_BFunWD \triangleq$
  $\forall x \in T : \forall i \in It(x) :$
    $\land\ gp(x, i) \in Tp$
    $\land\ \forall vp \in St(Tp) : wrts(vp, deps(x, Dep\_pp, i) \setminus \{i\}) \Rightarrow fp(x, vp, i) \in Tp$
    $\land\ \forall vp \in St(Tp) : wrts(vp, deps(x, Dep\_pc, i)) \qquad \Rightarrow fc(x, vp, i) \in Tc$
    $\land\ \forall vc \in St(Tc) : wrts(vc, deps(x, Dep\_cr, i)) \qquad \Rightarrow fr(x, vc, i) \in D$

AXIOM $H\_fpRelevance \triangleq$
  $\forall x \in T : \forall i \in It(x),\ vp1 \in St(Tp),\ vp2 \in St(Tp) :$
    $eqs(vp1, vp2, deps(x, Dep\_pp, i) \setminus \{i\}) \Rightarrow fp(x, vp1, i) = fp(x, vp2, i)$

AXIOM $H\_fcRelevance \triangleq$
  $\forall x \in T : \forall i \in It(x),\ vp1 \in St(Tp),\ vp2 \in St(Tp) :$
    $eqs(vp1, vp2, deps(x, Dep\_pc, i)) \Rightarrow fc(x, vp1, i) = fc(x, vp2, i)$

AXIOM $H\_frRelevance \triangleq$
  $\forall x \in T : \forall i \in It(x),\ vc1 \in St(Tc),\ vc2 \in St(Tc) :$
    $eqs(vc1, vc2, deps(x, Dep\_cr, i)) \Rightarrow fr(x, vc1, i) = fr(x, vc2, i)$

LEMMA $H\_ProdEqInv \triangleq$
  $\forall x \in T : \forall i \in It(x) :$
    $wrt(p[I0][i]) \Rightarrow fp(x, p[I0], i) = gp(x, i)$

├──────────────────────────────────────────────────────────┤

    Functional specification
$M \triangleq$ INSTANCE $AbelianMonoidBigOp$
    WITH $D \leftarrow D,\ Id \leftarrow id,\ \otimes \leftarrow Op$

AXIOM $H\_Algebra \triangleq AbelianMonoid(D, id, Op)$

$Gp(x) \qquad \triangleq [i \in Assig \mapsto gp(x, i)]$
$Fc(x, vp) \triangleq [i \in Assig \mapsto fc(x, vp, i)]$
$Fr(x, vc) \triangleq [i \in Assig \mapsto fr(x, vc, i)]$
```

Informal notation:

$$\mathcal{A}(x) \triangleq \bigoplus_{i \in I_x} \vec{f}_r^{\,i}(x, \vec{f}_c(x, \vec{g}_p(x))) \qquad \text{where } \; I_x \triangleq \{i \in lBnd2(x)..uBnd(x) : prop(i)\}$$

$$A(x) \triangleq M!BigOpP(lBnd(x), uBnd(x), prop, \text{LAMBDA } i : Fr(x, Fc(x, Gp(x)))[i])$$

Operational specification

$$vs \triangleq \langle X, p, c, r, rs \rangle$$

$$\begin{aligned}
Init \triangleq\ & \land in \in T \land pre(in)\\
& \land X = [I \in Index \mapsto \text{IF } I = I0 \text{ THEN } in \text{ ELSE } Undef]\\
& \land p\ \ = [I \in Index \mapsto [i \in Assig \mapsto Undef]]\\
& \land c\ \ = [I \in Index \mapsto [i \in Assig \mapsto Undef]]\\
& \land rs = [I \in Index \mapsto [i \in Assig \mapsto \text{FALSE}]]\\
& \land r\ \ = [I \in Index \mapsto id]
\end{aligned}$$

$$\begin{aligned}
&P(I, i) \triangleq\\
&\quad \land \neg wrt(p[I][i])\\
&\quad \land wrts(p[I], deps(X[I], Dep\_pp, i) \setminus \{i\})\\
&\quad \land p' = [p \text{ EXCEPT } ![I][i] = fp(X[I], p[I], i)]\\
&\quad \land \text{UNCHANGED } \langle X, c, r, rs \rangle
\end{aligned}$$

$$\begin{aligned}
&C(I, i) \triangleq\\
&\quad \land \neg wrt(c[I][i])\\
&\quad \land wrts(p[I], deps(X[I], Dep\_pc, i))\\
&\quad \land c' = [c \text{ EXCEPT } ![I][i] = fc(X[I], p[I], i)]\\
&\quad \land \text{UNCHANGED } \langle X, p, r, rs \rangle
\end{aligned}$$

$$\begin{aligned}
&R(I, i) \triangleq\\
&\quad \land \neg red(I, i)\\
&\quad \land wrts(c[I], deps(X[I], Dep\_cr, i))\\
&\quad \land r'\ \ = [r\ \ \text{EXCEPT } ![I]\quad\ = Op(@, fr(X[I], c[I], i))]\\
&\quad \land rs' = [rs \text{ EXCEPT } ![I][i]\ = \text{TRUE}]\\
&\quad \land \text{UNCHANGED } \langle X, p, c \rangle
\end{aligned}$$

$$\begin{aligned}
Done \triangleq\ & \land \forall I \in WDIndex : end(I)\\
& \land \text{UNCHANGED } \langle in, vs \rangle
\end{aligned}$$

$$\begin{aligned}
Step \triangleq\ & \land \exists I \in WDIndex :\\
&\qquad \exists i \in It(X[I]) : \lor P(I, i)\\
&\qquad\qquad\qquad\qquad\quad \lor C(I, i)\\
&\qquad\qquad\qquad\qquad\quad \lor R(I, i)\\
& \land \text{UNCHANGED } in
\end{aligned}$$

$$Next \triangleq Step \lor Done$$

$$Spec \triangleq Init \land \Box[Next]_{\langle in, vs \rangle}$$

$$FairSpec \triangleq Spec \land \text{WF}_{vs}(Step)$$

Properties

```
IndexInv ≜ WDIndex = {I0}

TypeInv ≜
  ∧ in ∈ T
  ∧ X  ∈ [Index → T ∪ {Undef}] ∧ X[I0] = in
  ∧ p  ∈ [Index → St(Tp)]
  ∧ c  ∈ [Index → St(Tc)]
  ∧ r  ∈ [Index → D]
  ∧ rs ∈ [Index → [Assig → BOOLEAN ]]

PInv ≜
  ∀ i ∈ It(X[I0]) :
     wrt(p[I0][i]) ⇒ ∧ wrts(p[I0], deps(X[I0], Dep_pp, i))
                     ∧ p[I0][i] = fp(X[I0], p[I0], i)

CInv ≜
  ∀ i ∈ It(X[I0]) :
     wrt(c[I0][i]) ⇒ ∧ wrts(p[I0], deps(X[I0], Dep_pc, i))
                     ∧ c[I0][i] = fc(X[I0], p[I0], i)

RInv1 ≜
  ∀ i ∈ It(X[I0]) :
     red(I0, i) ⇒ wrts(c[I0], deps(X[I0], Dep_cr, i))

RInv2 ≜
  r[I0] = M!BigOpP(lBnd(X[I0]), uBnd(X[I0]),
                   LAMBDA i : prop(i) ∧ red(I0, i),
                   LAMBDA i : fr(X[I0], c[I0], i))

Inv ≜ ∧ TypeInv
      ∧ IndexInv
      ∧ PInv
      ∧ CInv
      ∧ RInv1
      ∧ RInv2

ISpec ≜ Inv ∧ □[Next]_⟨vs⟩

Correctness ≜ end(I0) ⇒ r[I0] = A(X[I0])

Termination ≜ ◇end(I0)

 Refinement
inS  ≜ X[I0]
outS ≜ IF end(I0) THEN r[I0] ELSE id

A1step ≜ INSTANCE PCR_A1step
  WITH in ← inS, out ← outS,
       T ← T, D ← D,
       id ← id, Op ← Op,
       lBnd ← lBnd, uBnd ← uBnd, prop ← prop,
       fp ← fp, fc ← fc, fr ← fr, gp       ← gp

────────────────────────────────────────────────────────────
```

B.2 Basic PCR with left reducer

MODULE *PCR_ArLeft*

Basic *PCR* with left reducer over a consecutive iteration space.

```
  ---------------------------------------------------------------
  fun fp(x,p,i) = ...               // fp : T x St(Tp) x N -> Tp
  fun fc(x,p,i) = ...               // fc : T x St(Tp) x N -> Tc
  fun fr(x,c,i) = ...               // fr : T x St(Tc) x N -> D

  dep p(i-k) -> p(i)
  dep p(i[+/-]k) -> c(i)
  dep c(i[+/-]k) -> r(i)
  dep r(i-1) -> r(i)

  lbnd A = \x. ...
  ubnd A = \x. ...

  PCR A(x)                          // x \in T
    par
      p = produce fp x p
      c = consume fc x p
      r = reduce Op id (fr x c)     // r \in D
 ---------------------------------------------------------------
```

EXTENDS *AbstractAlgebra*, *Naturals*, *Sequences*, *Bags*, *SeqUtils*, *ArithUtils*, *TLC*

PCR constants and variables

CONSTANTS $I0,\ pre(\_),$
$T,\ Tp,\ Tc,\ D,$
$id,\ Op(\_,\_),$
$lBnd(\_),\ uBnd(\_),$
$fp(\_,\_,\_),\ fc(\_,\_,\_),\ fr(\_,\_,\_),\ gp(\_,\_),$
$Dep\_pp,\ Dep\_pc,\ Dep\_cr$

VARIABLES $in,\ X,\ p,\ c,\ r,\ rs$

General definitions

$Undef \triangleq$ CHOOSE $x : x \notin$ UNION $\{T,\ Tp,\ Tc,\ D\}$

$wrt(v) \triangleq v \neq Undef$
$wrts(v, S) \triangleq \forall k \in S : wrt(v[k])$
$eqs(v1, v2, S) \triangleq \forall k \in S : wrt(v1[k]) \land v1[k] = v2[k]$

PCR definitions and assumptions

$Index \triangleq Seq(Nat)$
$Assig \triangleq Nat$
$It(x) \triangleq lBnd(x) \,..\, uBnd(x)$
$WDIndex \triangleq \{I \in Index : wrt(X[I])\}$
$St(R) \triangleq [Assig \rightarrow R \cup \{Undef\}]$
$red(I, i) \triangleq rs[I][i]$
$end(I) \triangleq \forall i \in It(X[I]) : red(I, i)$

$deps(x, d, i) \triangleq$
$\qquad \{i - k : k \in \{k \in d[1] : i - k \geq lBnd(x)\}\}$
$\quad \cup \quad \{i\}$

```
$\cup \quad \{i + k : k \in \{k \in d[2] : i + k \leq uBnd(x)\}\}$

$\text{AXIOM } H\_Type \triangleq$
    $\land\, I0 \in Index$
    $\land\, \forall x \in T : lBnd(x) \in Nat$
    $\land\, \forall x \in T : uBnd(x) \in Nat$
    $\land\, \forall x \in T : pre(x) \in \text{BOOLEAN}$
    $\land\, Dep\_pp \in (\text{SUBSET } (Nat \setminus \{0\})) \times (\text{SUBSET } \{\})$
    $\land\, Dep\_pc \in (\text{SUBSET } (Nat \setminus \{0\})) \times (\text{SUBSET } (Nat \setminus \{0\}))$
    $\land\, Dep\_cr \in (\text{SUBSET } (Nat \setminus \{0\})) \times (\text{SUBSET } (Nat \setminus \{0\}))$

$\text{AXIOM } H\_BFunType \triangleq$
    $\forall x \in T, i \in Assig :$
        $\land\, gp(x, i) \in Tp \cup \{Undef\}$
        $\land\, \forall vp \in St(Tp) : fp(x, vp, i) \in Tp \cup \{Undef\}$
        $\land\, \forall vp \in St(Tp) : fc(x, vp, i) \in Tc \cup \{Undef\}$
        $\land\, \forall vc \in St(Tc) : fr(x, vc, i) \in D \cup \{Undef\}$

$\text{AXIOM } H\_BFunWD \triangleq$
    $\forall x \in T : \forall i \in It(x) :$
        $\land\, gp(x, i) \in Tp$
        $\land\, \forall vp \in St(Tp) : wrts(vp, deps(x, Dep\_pp, i) \setminus \{i\}) \Rightarrow fp(x, vp, i) \in Tp$
        $\land\, \forall vp \in St(Tp) : wrts(vp, deps(x, Dep\_pc, i)) \quad\quad \Rightarrow fc(x, vp, i) \in Tc$
        $\land\, \forall vc \in St(Tc) : wrts(vc, deps(x, Dep\_cr, i)) \quad\quad \Rightarrow fr(x, vc, i) \in D$

$\text{AXIOM } H\_fpRelevance \triangleq$
    $\forall x \in T : \forall i \in It(x), vp1 \in St(Tp), vp2 \in St(Tp) :$
        $eqs(vp1, vp2, deps(x, Dep\_pp, i) \setminus \{i\}) \Rightarrow fp(x, vp1, i) = fp(x, vp2, i)$

$\text{AXIOM } H\_fcRelevance \triangleq$
    $\forall x \in T : \forall i \in It(x), vp1 \in St(Tp), vp2 \in St(Tp) :$
        $eqs(vp1, vp2, deps(x, Dep\_pc, i)) \Rightarrow fc(x, vp1, i) = fc(x, vp2, i)$

$\text{AXIOM } H\_frRelevance \triangleq$
    $\forall x \in T : \forall i \in It(x), vc1 \in St(Tc), vc2 \in St(Tc) :$
        $eqs(vc1, vc2, deps(x, Dep\_cr, i)) \Rightarrow fr(x, vc1, i) = fr(x, vc2, i)$

$\text{LEMMA } H\_ProdEqInv \triangleq$
    $\forall x \in T : \forall i \in It(x) :$
        $wrt(p[I0][i]) \Rightarrow fp(x, p[I0], i) = gp(x, i)$

|------------------------------------------------------------------------------|
```

Functional specification

```
$M \triangleq \text{INSTANCE } MonoidBigOp$
    $\text{WITH } D \leftarrow D, Id \leftarrow id, \otimes \leftarrow Op$

$\text{AXIOM } H\_Algebra \triangleq Monoid(D, id, Op)$

$Gp(x) \quad\;\; \triangleq [i \in Assig \mapsto gp(x, i)]$
$Fc(x, vp) \triangleq [i \in Assig \mapsto fc(x, vp, i)]$
$Fr(x, vc) \triangleq [i \in Assig \mapsto fr(x, vc, i)]$
```

Informal notation:

$$\mathcal{A}(x) \triangleq \bigotimes_{i \in I_x} \vec{f}_r^{\,i}(x, \vec{f}_c(x, \vec{g}_p(x))) \qquad \text{where} \quad I_x \triangleq \{i \in lBnd(x)..uBnd(x) : prop(i)\}$$

```
A(x) ≜ M!BigOp(lBnd(x), uBnd(x), LAMBDA i : Fr(x, Fc(x, Gp(x)))[i])
├──────────────────────────────────────────────────────────────────┤

  Operational specification
vs ≜ ⟨X, p, c, r, rs⟩

Init ≜ ∧ in ∈ T ∧ pre(in)
       ∧ X = [I ∈ Index ↦ IF I = I0 THEN in ELSE  Undef]
       ∧ p  = [I ∈ Index ↦ [i ∈ Assig ↦ Undef]]
       ∧ c  = [I ∈ Index ↦ [i ∈ Assig ↦ Undef]]
       ∧ rs = [I ∈ Index ↦ [i ∈ Assig ↦ FALSE]]
       ∧ r  = [I ∈ Index ↦ id]

P(I, i) ≜
   ∧ ¬wrt(p[I][i])
   ∧ wrts(p[I], deps(X[I], Dep_pp, i) \ {i})
   ∧ p′ = [p EXCEPT ![I][i] = fp(X[I], p[I], i)]
   ∧ UNCHANGED ⟨X, c, r, rs⟩

C(I, i) ≜
   ∧ ¬wrt(c[I][i])
   ∧ wrts(p[I], deps(X[I], Dep_pc, i))
   ∧ c′ = [c EXCEPT ![I][i] = fc(X[I], p[I], i)]
   ∧ UNCHANGED ⟨X, p, r, rs⟩

R(I, i) ≜
   ∧ ¬red(I, i)
   ∧ wrts(c[I], deps(X[I], Dep_cr, i))
   ∧ i − 1 ≥ lBnd(X[I]) ⇒ red(I, i − 1)      dep r(i − 1) → r(i)
   ∧ r′    = [r  EXCEPT ![I]    = Op(@, fr(X[I], c[I], i))]
   ∧ rs′   = [rs EXCEPT ![I][i] = TRUE]
   ∧ UNCHANGED ⟨X, p, c⟩

Done ≜ ∧ ∀ I ∈ WDIndex : end(I)
       ∧ UNCHANGED ⟨in, vs⟩

Step  ≜ ∧ ∃ I ∈ WDIndex :
             ∃ i ∈ It(X[I]) : ∨ P(I, i)
                              ∨ C(I, i)
                              ∨ R(I, i)
        ∧ UNCHANGED in

Next  ≜ Step ∨ Done

Spec  ≜ Init ∧ □[Next]_⟨in, vs⟩

FairSpec ≜ Spec ∧ WF_vs(Step)
├──────────────────────────────────────────────────────────────────┤

  Properties
IndexInv ≜ WDIndex = {I0}

TypeInv ≜
  ∧ in ∈ T
  ∧ X  ∈ [Index → T ∪ {Undef}] ∧ X[I0] = in
  ∧ p  ∈ [Index → St(Tp)]
  ∧ c  ∈ [Index → St(Tc)]
  ∧ r  ∈ [Index → D]
  ∧ rs ∈ [Index → [Assig → BOOLEAN ]]

PInv ≜
  ∀ i ∈ It(X[I0]) :
     wrt(p[I0][i]) ⇒ ∧ wrts(p[I0], deps(X[I0], Dep_pp, i))
                     ∧ p[I0][i] = fp(X[I0], p[I0], i)

CInv ≜
  ∀ i ∈ It(X[I0]) :
     wrt(c[I0][i]) ⇒ ∧ wrts(p[I0], deps(X[I0], Dep_pc, i))
                     ∧ c[I0][i] = fc(X[I0], p[I0], i)

RInv1 ≜
  ∀ i ∈ It(X[I0]) :
     red(I0, i) ⇒ ∧ wrts(c[I0], deps(X[I0], Dep_cr, i))
                  ∧ ∀ k ∈ It(X[I0]) : k < i ⇒ red(I0, k)

RInv2 ≜
  ∀ i ∈ It(X[I0]) :
     ¬red(I0, i) ⇒ r[I0] = M!BigOpP(lBnd(X[I0]), i − 1,
                                     LAMBDA j : red(I0, j),
                                     LAMBDA j : fr(X[I0], c[I0], j))

Inv ≜ ∧ TypeInv
      ∧ IndexInv
      ∧ PInv
      ∧ CInv
      ∧ RInv1
      ∧ RInv2
```

$ISpec \triangleq Inv \land \Box[Next]_{\langle in,\, vs\rangle}$

```
Correctness ≜ end(I0) ⇒ r[I0] = A(X[I0])

Termination ≜ ◇end(I0)
```

Refinement

```
inS ≜ X[I0]
propS(x) ≜ TRUE

PCR_A ≜ INSTANCE PCR_A
   WITH in ← inS, X ← X, p  ← p, c ← c, r ← r, rs ← rs,
        T ← T, Tp ← Tp, Tc ← Tc, D ← D,
        id ← id, Op ← Op,
        lBnd ← lBnd, uBnd ← uBnd, prop ← propS,
        fp ← fp, fc ← fc, fr ← fr, gp       ← gp,
        Dep_pp ← Dep_pp, Dep_pc ← Dep_pc, Dep_cr ← Dep_cr
```

B.3 Basic PCR: one step computation

MODULE *PCR_A1step*

Basic *PCR* as a one step computation.

```
 ----------------------------------------------------------------
 fun fp(x,p,i) = ...              // fp : T x St(Tp) x N -> Tp
 fun fc(x,p,i) = ...              // fc : T x St(Tp) x N -> Tc
 fun fr(x,c,i) = ...              // fr : T x St(Tc) x N -> D

 dep p(i-k) -> p(i)
 dep p(i[+/-]k) -> c(i)
 dep c(i[+/-]k) -> r(i)

 lbnd A = \x. ...
 ubnd A = \x. ...
 prop A = \i. ...

 PCR A(x)                         // x \in T
   par
     p = produce fp x p
     c = consume fc x p
     r = reduce  Op id (fr x c)    // r \in D
----------------------------------------------------------------
```

EXTENDS *AbstractAlgebra*, *Naturals*, *Sequences*, *Bags*, *SeqUtils*, *ArithUtils*, *TLC*

PCR constants and variables

CONSTANTS $pre(\_)$,
$\quad T, D,$
$\quad id, Op(\_, \_),$
$\quad lBnd(\_), uBnd(\_), prop(\_),$
$\quad fp(\_, \_, \_), fc(\_, \_, \_), fr(\_, \_, \_), gp(\_, \_)$

VARIABLES in, out

PCR definitions

$Assig \triangleq Nat$

$M \triangleq$ INSTANCE *AbelianMonoidBigOp*
$\quad$ WITH $D \leftarrow D, Id \leftarrow id, \otimes \leftarrow Op$

$Gp(x) \triangleq [i \in Assig \mapsto gp(x, i)]$
$Fc(x, vp) \triangleq [i \in Assig \mapsto fc(x, vp, i)]$
$Fr(x, vc) \triangleq [i \in Assig \mapsto fr(x, vc, i)]$

Informal notation:

$$\mathcal{A}(x) \triangleq \bigotimes_{i \in I_x} \vec{f}_r^{\,i}(x, \vec{f}_c(x, \vec{g}_p(x))) \qquad \text{where} \quad I_x \triangleq \{i \in lBnd(x)..uBnd(x) : prop(i)\}$$

$A(x) \triangleq M!BigOpP(lBnd(x), uBnd(x), prop,$ LAMBDA $i : Fr(x, Fc(x, Gp(x)))[i])$

Operational specification

$vs \triangleq \langle in, out\rangle$

$Init \triangleq \land\ in \in T$
$\qquad\quad \land\ pre(in)$

```
                $\land\ out = id$

$Next \triangleq \land\ out' = A(in)$
                 $\land$ UNCHANGED $in$

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

$FairSpec \triangleq Spec \land \mathrm{WF}_{vs}(Next)$

────────────────────────────────────────────────

  Properties
$Termination \triangleq \Diamond(out = A(in))$

────────────────────────────────────────────────
```

B.4 Composition through consumer

MODULE *PCR_A_c_B*

PCR composed through consumer with a basic *PCR*.

```
  ------------------------------------------------------------------
  // PCR A

  fun fp1(x1,p1,i) = ...                  // fp1 : T x St(Tp1) x N -> Tp1
  fun fr1(x1,c1,i) = ...                  // fr1 : T x St(D2) x N -> D1

  dep p1(i-k) -> p1(i)
  dep p1(i[+/-]k) -> c1(i)
  dep c1(i[+/-]k) -> r1(i)

  lbnd A = \x1. ...
  ubnd A = \x1. ...
  prop A = \i. ...

  PCR A(x1)                               // x1 \in T
    par
      p1 = produce fp1 x1 p1
      c1 = consume B x1 p1
      r1 = reduce Op1 id1 (fr1 x1 c1)    // r1 \in D1

  // PCR B
                                          // T2 = T x St(Tp1) x N
  fun fp2(x2,p2,j) = ...                  // fp2 : T2 x St(Tp2) x N -> Tp2
  fun fc2(x2,p2,j) = ...                  // fc2 : T2 x St(Tp2) x N -> Tc2
  fun fr2(x2,c2,j) = ...                  // fr2 : T2 x St(Tc2) x N -> D2

  dep p2(i-k) -> p2(i)
  dep p2(i[+/-]k) -> c2(i)
  dep c2(i[+/-]k) -> r2(i)

  lbnd B = \x2. ...
  ubnd B = \x2. ...
  prop B = \j. ...

  PCR B(x2)                               // x2 \in T2
    par
      p2 = produce fp1 x2 p2
      c2 = consume fc2 x2 p2
      r2 = reduce Op2 id2 (fr2 x2 c2)    // r2 \in D2
------------------------------------------------------------------
```

EXTENDS *AbstractAlgebra*, *Naturals*, *Sequences*, *Bags*, *SeqUtils*, *ArithUtils*, *TLC*

PCR A constants and variables

CONSTANTS $I0$, $pre(\_)$,
T, $Tp1$, $D1$,
$id1$, $Op1(\_, \_)$,
$lBnd1(\_)$, $uBnd1(\_)$, $prop1(\_)$,
$fp1(\_, \_, \_)$, $fr1(\_, \_, \_)$, $gp1(\_, \_)$,
Dep_pp1, Dep_pc1, Dep_cr1

VARIABLES in, $X1$, $p1$, $c1$, $r1$, $rs1$

PCR B constants and variables

CONSTANTS $Tp2$, $Tc2$, $D2$,
$id2$, $Op2(\_, \_)$,
$lBnd2(\_)$, $uBnd2(\_)$, $prop2(\_)$,
$fp2(\_, \_, \_)$, $fc2(\_, \_, \_)$, $fr2(\_, \_, \_)$, $gp2(\_, \_)$,
Dep_pp2, Dep_pc2, Dep_cr2

```
VARIABLES X2, p2, c2, r2, rs2
────────────────────────────────────────────────────────────────

  General definitions
Undef ≜ CHOOSE x : x ∉ UNION {T, Tp1, Tp2, Tc2, D1, D2}

wrt(v)            ≜ v ≠ Undef
wrts(v, S)        ≜ ∀ k ∈ S : wrt(v[k])
eqs(v1, v2, S)    ≜ ∀ k ∈ S : wrt(v1[k]) ∧ v1[k] = v2[k]

────────────────────────────────────────────────────────────────

  PCR A definitions and assumptions
IndexA       ≜ Seq(Nat)
AssigA       ≜ Nat
ItA(x)       ≜ {i ∈ lBnd1(x) .. uBnd1(x) : prop1(i)}
WDIndexA     ≜ {I ∈ IndexA : wrt(X1[I])}
StA(R)       ≜ [AssigA → R ∪ {Undef}]
redA(I, i)   ≜ rs1[I][i]
endA(I)      ≜ ∀ i ∈ ItA(X1[I]) : redA(I, i)

depsA(x, d, i) ≜
            {i − k : k ∈ {k ∈ d[1] : i − k ≥ lBnd1(x) ∧ prop1(i − k)}}
  ∪         {i}
  ∪         {i + k : k ∈ {k ∈ d[2] : i + k ≤ uBnd1(x) ∧ prop1(i + k)}}

AXIOM H_TypeA ≜
  ∧ I0    ∈ IndexA
  ∧ ∀ x   ∈ T   : lBnd1(x)  ∈ Nat
  ∧ ∀ x   ∈ T   : uBnd1(x)  ∈ Nat
  ∧ ∀ i   ∈ Nat : prop1(i)  ∈ BOOLEAN
  ∧ ∀ x   ∈ T   : pre(x) ∈ BOOLEAN
  ∧ Dep_pp1 ∈ (SUBSET (Nat \ {0})) × (SUBSET {})
  ∧ Dep_pc1 ∈ (SUBSET (Nat \ {0})) × (SUBSET (Nat \ {0}))
  ∧ Dep_cr1 ∈ (SUBSET (Nat \ {0})) × (SUBSET (Nat \ {0}))

AXIOM H_BFunTypeA ≜
  ∀ x ∈ T, i     ∈ AssigA :
     ∧ gp1(x, i) ∈ Tp1 ∪ {Undef}
     ∧ ∀ vp ∈ StA(Tp1) : fp1(x, vp, i) ∈ Tp1 ∪ {Undef}
     ∧ ∀ vc ∈ StA(D2)  : fr1(x, vc, i) ∈ D1  ∪ {Undef}

AXIOM H_BFunWDA ≜
  ∀ x ∈ T : ∀ i ∈ ItA(x) :
     ∧ gp1(x, i) ∈ Tp1
     ∧ ∀ vp ∈ StA(Tp1) : wrts(vp, depsA(x, Dep_pp1, i) \ {i}) ⇒ fp1(x, vp, i) ∈ Tp1
     ∧ ∀ vc ∈ StA(D2)  : wrts(vc, depsA(x, Dep_cr1, i))        ⇒ fr1(x, vc, i) ∈ D1

AXIOM H_fpRelevanceA ≜
  ∀ x ∈ T : ∀ i ∈ ItA(x), vp1 ∈ StA(Tp1), vp2 ∈ StA(Tp1) :
     eqs(vp1, vp2, depsA(x, Dep_pp1, i) \ {i}) ⇒ fp1(x, vp1, i) = fp1(x, vp2, i)

AXIOM H_frRelevanceA ≜
  ∀ x ∈ T : ∀ i ∈ ItA(x), vc1 ∈ StA(D2), vc2 ∈ StA(D2) :
    $eqs(vc1, vc2, depsA(x, Dep\_cr1, i)) \Rightarrow fr1(x, vc1, i) = fr1(x, vc2, i)$

LEMMA $H\_ProdEqInvA \triangleq$
  $\forall x \in T : \forall i \in ItA(x) :$
    $wrt(p1[I0][i]) \Rightarrow fp1(x, p1[I0], i) = gp1(x, i)$

$PCR\ B$ definitions and assumptions
$IndexB \triangleq Seq(Nat)$
$AssigB \triangleq Nat$
$ItB(x) \triangleq \{i \in lBnd2(x) \,..\, uBnd2(x) : prop2(i)\}$
$WDIndexB \triangleq \{I \in IndexB : wrt(X2[I])\}$
$StB(R) \triangleq [AssigB \rightarrow R \cup \{Undef\}]$
$redB(I, i) \triangleq rs2[I][i]$
$endB(I) \triangleq \forall i \in ItB(X2[I]) : redB(I, i)$

$depsB(x, d, i) \triangleq$
            $\{i - k : k \in \{k \in d[1] : i - k \geq lBnd2(x) \land prop2(i - k)\}\}$
  $\cup$        $\{i\}$
  $\cup$        $\{i + k : k \in \{k \in d[2] : i + k \leq uBnd2(x) \land prop2(i + k)\}\}$

$T2 \triangleq T \times StA(Tp1) \times AssigA$

AXIOM $H\_TypeB \triangleq$
  $\land \forall x \in T2 : lBnd2(x) \in Nat$
  $\land \forall x \in T2 : uBnd2(x) \in Nat$
  $\land \forall i \in Nat : prop2(i) \in$ BOOLEAN
  $\land Dep\_pp2 \in (\text{SUBSET } (Nat \setminus \{0\})) \times (\text{SUBSET } \{\})$
  $\land Dep\_pc2 \in (\text{SUBSET } (Nat \setminus \{0\})) \times (\text{SUBSET } (Nat \setminus \{0\}))$
  $\land Dep\_cr2 \in (\text{SUBSET } (Nat \setminus \{0\})) \times (\text{SUBSET } (Nat \setminus \{0\}))$

AXIOM $H\_BFunTypeB \triangleq$
  $\forall x \in T2, i \in AssigB :$
    $\land gp2(x, i) \in Tp2 \cup \{Undef\}$
    $\land \forall vp \in StB(Tp2) : fp2(x, vp, i) \in Tp2 \cup \{Undef\}$
    $\land \forall vp \in StB(Tp2) : fc2(x, vp, i) \in Tc2 \cup \{Undef\}$
    $\land \forall vc \in StB(Tc2) : fr2(x, vc, i) \in D2 \cup \{Undef\}$

AXIOM $H\_BFunWDB \triangleq$
  $\forall x \in T2 : \forall i \in ItB(x) :$
    $\land gp2(x, i) \in Tp2$
    $\land \forall vp \in StB(Tp2) : wrts(vp, depsB(x, Dep\_pp2, i) \setminus \{i\}) \Rightarrow fp2(x, vp, i) \in Tp2$
    $\land \forall vp \in StB(Tp2) : wrts(vp, depsB(x, Dep\_pc2, i))$        $\Rightarrow fc2(x, vp, i) \in Tc2$
    $\land \forall vc \in StB(Tc2) : wrts(vc, depsB(x, Dep\_cr2, i))$        $\Rightarrow fr2(x, vc, i) \in D2$

AXIOM $H\_fpRelevanceB \triangleq$
  $\forall x \in T2 : \forall i \in ItB(x), vp1 \in StB(Tp2), vp2 \in StB(Tp2) :$
    $eqs(vp1, vp2, depsB(x, Dep\_pp2, i) \setminus \{i\}) \Rightarrow fp2(x, vp1, i) = fp2(x, vp2, i)$

AXIOM $H\_fcRelevanceB \triangleq$
  $\forall x \in T2 : \forall i \in ItB(x), vp1 \in StB(Tp2), vp2 \in StB(Tp2) :$
    $eqs(vp1, vp2, depsB(x, Dep\_pc2, i)) \Rightarrow fc2(x, vp1, i) = fc2(x, vp2, i)$

AXIOM $H\_frRelevanceB \triangleq$
  $\forall x \in T2 : \forall i \in ItB(x), vc1 \in StB(Tc2), vc2 \in StB(Tc2) :$
    $eqs(vc1, vc2, depsB(x, Dep\_cr2, i)) \Rightarrow fr2(x, vc1, i) = fr2(x, vc2, i)$
```

$\text{LEMMA } H\_ProdEqInvB \triangleq$
$\quad \forall I \in WDIndexB : \forall i \in ItB(X2[I]) :$
$\qquad wrt(p1[I][i]) \Rightarrow fp2(X2[I], p2[I], i) = gp2(X2[I], i)$

Functional specification

$M2 \triangleq \text{INSTANCE } AbelianMonoidBigOp$
$\quad \text{WITH } D \leftarrow D2, Id \leftarrow id2, \otimes \leftarrow Op2$

$\text{AXIOM } H\_AlgebraB \triangleq AbelianMonoid(D2, id2, Op2)$

$Gp2(x) \triangleq [i \in AssigB \mapsto gp2(x, i)]$
$Fc2(x, vc) \triangleq [i \in AssigB \mapsto fc2(x, vc, i)]$
$Fr2(x, vc) \triangleq [i \in AssigB \mapsto fr2(x, vc, i)]$

Informal notation:

$\mathcal{B}(x_2) \triangleq \bigoplus_{j \in J_{x_2}} \vec{f}^{\,j}_{r_2}(x_2, \vec{f}_{c_2}(x_2, \vec{g}_{p_2}(x_2)))$ where $J_{x_2} \triangleq \{j \in lBnd2(x_2)..uBnd2(x_2) : prop2(j)\}$

$B(x2) \triangleq M2!BigOpP(lBnd2(x2), uBnd2(x2), prop2,$
$\qquad\qquad\qquad \text{LAMBDA } j : Fr2(x2, Fc2(x2, Gp2(x2)))[j])$

$M1 \triangleq \text{INSTANCE } AbelianMonoidBigOp$
$\quad \text{WITH } D \leftarrow D1, Id \leftarrow id1, \otimes \leftarrow Op1$

$\text{AXIOM } H\_AlgebraA \triangleq AbelianMonoid(D1, id1, Op1)$

$Gp1(x) \triangleq [i \in AssigA \mapsto gp1(x, i)]$
$Fc1(x1, vp) \triangleq [i \in AssigA \mapsto B(\langle x1, vp, i\rangle)]$
$Fr1(x, vc) \triangleq [i \in AssigA \mapsto fr1(x, vc, i)]$

Informal notation:

$\mathcal{A}(x_1) \triangleq \bigotimes_{i \in I_{x_1}} \vec{f}^{\,i}_{r_1}(x_1, i \in \mathbb{N} \mapsto \mathcal{B}(x_1, \vec{g}_{p_1}(x_1), i))$ where $I_{x_1} \triangleq \{i \in lBnd1(x_1)..uBnd1(x_1) : prop1(i)\}$

$A(x1) \triangleq M1!BigOpP(lBnd1(x1), uBnd1(x1), prop1,$
$\qquad\qquad\qquad \text{LAMBDA } i : Fr1(x1, Fc1(x1, Gp1(x1)))[i])$

Operational specification

$vs1 \triangleq \langle X1, p1, c1, r1, rs1, X2\rangle$
$vs2 \triangleq \langle p2, c2, r2, rs2\rangle$

$InitA \triangleq \land\, in \in T \land pre(in)$
$\qquad\quad \land\, X1 = [I \in IndexA \mapsto \text{IF } I = I0 \text{ THEN } in \text{ ELSE } Undef]$
$\qquad\quad \land\, p1 = [I \in IndexA \mapsto [i \in AssigA \mapsto Undef]]$
$\qquad\quad \land\, c1 = [I \in IndexA \mapsto [i \in AssigA \mapsto Undef]]$
$\qquad\quad \land\, rs1 = [I \in IndexA \mapsto [i \in AssigA \mapsto \text{FALSE}]]$
$\qquad\quad \land\, r1 = [I \in IndexA \mapsto id1]$

$InitB \triangleq \land\, X2 = [I \in IndexB \mapsto Undef]$
$\qquad\quad \land\, p2 = [I \in IndexB \mapsto [i \in AssigB \mapsto Undef]]$
$\qquad\quad \land\, c2 = [I \in IndexB \mapsto [i \in AssigB \mapsto Undef]]$
$\qquad\quad \land\, rs2 = [I \in IndexB \mapsto [i \in AssigB \mapsto \text{FALSE}]]$
$\qquad\quad \land\, r2 = [I \in IndexB \mapsto id2]$

```
$Init \triangleq InitA \land InitB$

$P1(I, i) \triangleq$
    $\land \neg wrt(p1[I][i])$
    $\land wrts(p1[I], depsA(X1[I], Dep\_pp1, i) \setminus \{i\})$
    $\land p1' = [p1 \text{ EXCEPT } ![I][i] = fp1(X1[I], p1[I], i)]$
    $\land \text{UNCHANGED } \langle X1, c1, r1, rs1, X2 \rangle$

$C1ini(I, i) \triangleq$
   $\land \quad \neg wrt(X2[I \circ \langle i \rangle])$
   $\land \quad wrts(p1[I], depsA(X1[I], Dep\_pc1, i))$
   $\land \quad X2' = [X2 \text{ EXCEPT } ![I \circ \langle i \rangle] = \langle X1[I], p1[I], i \rangle]$
   $\land \quad \text{UNCHANGED } \langle X1, p1, c1, r1, rs1 \rangle$

$C1end(I, i) \triangleq$
   $\land \quad \neg wrt(c1[I][i])$
   $\land \quad wrt(X2[I \circ \langle i \rangle])$
   $\land \quad endB(I \circ \langle i \rangle)$
   $\land \quad c1' = [c1 \text{ EXCEPT } ![I][i] = r2[I \circ \langle i \rangle]]$
   $\land \quad \text{UNCHANGED } \langle X1, p1, r1, rs1, X2 \rangle$

$R1(I, i) \triangleq$
    $\land \neg redA(I, i)$
    $\land wrts(c1[I], depsA(X1[I], Dep\_cr1, i))$
    $\land r1' \ = [r1 \quad \text{EXCEPT } ![I] \quad\ = Op1(@, fr1(X1[I], c1[I], i))]$
    $\land rs1' = [rs1 \text{ EXCEPT } ![I][i] = \text{TRUE}]$
    $\land \text{UNCHANGED } \langle X1, p1, c1, X2 \rangle$

$P2(I, i) \triangleq$
    $\land \neg wrt(p2[I][i])$
    $\land wrts(p2[I], depsB(X2[I], Dep\_pp2, i) \setminus \{i\})$
    $\land p2' = [p2 \text{ EXCEPT } ![I][i] = fp2(X2[I], p2[I], i)]$
    $\land \text{UNCHANGED } \langle c2, r2, rs2 \rangle$

$C2(I, i) \triangleq$
    $\land \neg wrt(c2[I][i])$
    $\land wrts(p2[I], depsB(X2[I], Dep\_pc2, i))$
    $\land c2' = [c2 \text{ EXCEPT } ![I][i] = fc2(X2[I], p2[I], i)]$
    $\land \text{UNCHANGED } \langle p2, r2, rs2 \rangle$

$R2(I, i) \triangleq$
    $\land \neg redB(I, i)$
    $\land wrts(c2[I], depsB(X2[I], Dep\_cr2, i))$
    $\land r2' \ = [r2 \quad \text{EXCEPT } ![I] \quad\ = Op2(@, fr2(X2[I], c2[I], i))]$
    $\land rs2' = [rs2 \text{ EXCEPT } ![I][i] \ = \text{TRUE}]$
    $\land \text{UNCHANGED } \langle p2, c2 \rangle$

$Done \triangleq \land \forall I \in WDIndexA : endA(I)$
             $\land \forall I \in WDIndexB : endB(I)$
             $\land \text{UNCHANGED } \langle in, vs1, vs2 \rangle$

$StepA \triangleq \land \exists I \in WDIndexA :$
                 $\exists i \in ItA(X1[I]) : \lor P1(I, i)$
                                   $\lor C1ini(I, i)$
                                   $\lor C1end(I, i)$
```

```
                                          $\lor R1(I, i)$
              $\land$ UNCHANGED $\langle in, vs2 \rangle$

$StepB \triangleq \land \exists I \in WDIndexB :$
                   $\exists i \in ItB(X2[I]) : \lor P2(I, i)$
                                          $\lor C2(I, i)$
                                          $\lor R2(I, i)$
              $\land$ UNCHANGED $\langle in, vs1 \rangle$

$Next \triangleq StepA \lor StepB \lor Done$

$Spec \triangleq Init \land \Box[Next]_{\langle in, vs1, vs2 \rangle}$

$FairSpec \triangleq Spec \land \mathrm{WF}_{vs1}(StepA) \land \mathrm{WF}_{vs2}(StepB)$

--------------------------------------------------------------------------------

  PCR A properties
$IndexInvA \triangleq WDIndexA = \{I0\}$

$TypeInvA \triangleq$
  $\land in \in T$
  $\land X1 \in [IndexA \to T \cup \{Undef\}] \land X1[I0] = in$
  $\land p1 \in [IndexA \to StA(Tp1)]$
  $\land c1 \in [IndexA \to StA(D2)]$
  $\land r1 \in [IndexA \to D1]$
  $\land rs1 \in [IndexA \to [AssigA \to$ BOOLEAN $]]$

$PInvA \triangleq$
  $\forall i \in ItA(X1[I0]) :$
    $wrt(p1[I0][i]) \Rightarrow \land wrts(p1[I0], depsA(X1[I0], Dep\_pp1, i))$
                            $\land p1[I0][i] = gp1(X1[I0], i)$

$CInv1A \triangleq$
  $\forall i \in ItA(X1[I0]) :$
    $wrt(X2[I0 \circ \langle i \rangle]) \Rightarrow \land wrts(p1[I0], depsA(X1[I0], Dep\_pc1, i))$
                               $\land \exists vp \in StA(Tp1) :$
                                    $\land eqs(vp, p1[I0], depsA(X1[I0], Dep\_pc1, i))$
                                    $\land X2[I0 \circ \langle i \rangle] = \langle X1[I0], vp, i \rangle$
$CInv2A \triangleq$
  $\forall i \in ItA(X1[I0]) :$
    $wrt(c1[I0][i]) \Rightarrow \land wrt(X2[I0 \circ \langle i \rangle])$
                            $\land endB(I0 \circ \langle i \rangle)$
                            $\land c1[I0][i] = r2[I0 \circ \langle i \rangle]$

$RInv1A \triangleq$
  $\forall i \in ItA(X1[I0]) :$
    $redA(I0, i) \Rightarrow wrts(c1[I0], depsA(X1[I0], Dep\_cr1, i))$

$RInv2A \triangleq$
  $r1[I0] = M1!BigOpP(lBnd1(X1[I0]), uBnd1(X1[I0]),$
                              LAMBDA $i : prop1(i) \land redA(I0, i),$
                              LAMBDA $i : fr1(X1[I0], c1[I0], i))$

$InvA \triangleq \land IndexInvA$
             $\land TypeInvA$
```

$$\begin{aligned} &\land\ PInvA \\ &\land\ CInv1A \\ &\land\ CInv2A \\ &\land\ RInv1A \\ &\land\ RInv2A \end{aligned}$$

$$CorrectnessA \triangleq endA(I0) \Rightarrow r1[I0] = A(X1[I0])$$

$$TerminationA \triangleq \Diamond endA(I0)$$

$$GTermination \triangleq endA(I0) \Rightarrow \forall\, I \in WDIndexB : endB(I)$$

PCR B properties

$$IndexInvB \triangleq WDIndexB \subseteq \{I0 \circ \langle i \rangle : i \in AssigA\}$$

$$\begin{aligned} &TypeInvB \triangleq \\ &\quad \land\ X2 \ \in [IndexB \rightarrow T2 \cup \{Undef\}] \\ &\quad \land\ p2 \ \in [IndexB \rightarrow StB(Tp2)] \\ &\quad \land\ c2 \ \in [IndexB \rightarrow StB(Tc2)] \\ &\quad \land\ r2 \ \in [IndexB \rightarrow D2] \\ &\quad \land\ rs2 \ \in [IndexB \rightarrow [AssigB \rightarrow \text{BOOLEAN}\ ]] \end{aligned}$$

$$\begin{aligned} &PInvB \triangleq \\ &\quad \forall\, I \in WDIndexB : \forall\, i \in ItB(X2[I]) : \\ &\qquad wrt(p2[I][i]) \Rightarrow \land\ wrts(p2[I],\ depsB(X2[I],\ Dep\_pp2,\ i)) \\ &\qquad\qquad\qquad\qquad\ \land\ p2[I][i] = gp2(X2[I],\ i) \end{aligned}$$

$$\begin{aligned} &CInvB \triangleq \\ &\quad \forall\, I \in WDIndexB : \forall\, i \in ItB(X2[I]) : \\ &\qquad wrt(c2[I][i]) \Rightarrow \land\ wrts(p2[I],\ depsB(X2[I],\ Dep\_pc2,\ i)) \\ &\qquad\qquad\qquad\qquad\ \land\ c2[I][i] = fc2(X2[I],\ p2[I],\ i) \end{aligned}$$

$$\begin{aligned} &RInv1B \triangleq \\ &\quad \forall\, I \in WDIndexB : \forall\, i \in ItB(X2[I]) : \\ &\qquad redB(I,\ i) \Rightarrow wrts(c2[I],\ depsB(X2[I],\ Dep\_cr2,\ i)) \end{aligned}$$

$$\begin{aligned} &RInv2B \triangleq \\ &\quad \forall\, I \in WDIndexB : \\ &\qquad r2[I] = M2!BigOpP(lBnd2(X2[I]),\ uBnd2(X2[I]), \\ &\qquad\qquad\qquad\qquad \text{LAMBDA}\ j : prop2(j) \land redB(I,\ j), \\ &\qquad\qquad\qquad\qquad \text{LAMBDA}\ j : fr2(X2[I],\ c2[I],\ j)) \end{aligned}$$

$$\begin{aligned} InvB \triangleq\ &\land\ IndexInvB \\ &\land\ TypeInvB \\ &\land\ PInvB \\ &\land\ CInvB \\ &\land\ RInv1B \\ &\land\ RInv2B \end{aligned}$$

$$CorrectnessB \triangleq \forall\, I \in WDIndexB : endB(I) \Rightarrow r2[I] = B(X2[I])$$

$$TerminationB \triangleq \Diamond(\forall\, I \in WDIndexB : endB(I))$$

Conjoint properties

```
TypeInv ≜ ∧ TypeInvA
          ∧ TypeInvB

Inv ≜ ∧ TypeInv
      ∧ InvA
      ∧ InvB

Correctness ≜ ∧ CorrectnessA
              ∧ CorrectnessB

Termination ≜ ∧ TerminationA
              ∧ TerminationB

 Refinement
inS ≜ X1[I0]
fcS(x1, vp1, i) ≜ B(⟨x1, vp1, i⟩)

PCR_A ≜ INSTANCE PCR_A
   WITH in ← inS, X ← X1, p  ← p1, c  ← c1, r ← r1, rs ← rs1,
        T ← T, Tp ← Tp1, Tc ← D2, D ← D1,
        id ← id1, Op ← Op1,
        lBnd ← lBnd1, uBnd ← uBnd1, prop ← prop1,
        fp ← fp1, fc ← fcS, fr ← fr1, gp ← gp1,
        Dep_pp ← Dep_pp1, Dep_pc ← Dep_pc1, Dep_cr ← Dep_cr1

AXIOM H_UndefRestrict ≜ PCR_A!Undef = Undef

AXIOM H_fcSRelevance ≜
  ∀ x ∈ T : ∀ i ∈ ItA(x), vp1 ∈ StA(Tp1), vp2 ∈ StA(Tp1) :
    eqs(vp1, vp2, depsA(x, Dep_pc1, i)) ⇒ fcS(x, vp1, i) = fcS(x, vp2, i)

```

B.5 Composition through reducer

MODULE *PCR_A_r_B*

PCR composed through reducer with a basic *PCR*.

```
  ----------------------------------------------------------------
  // PCR A

  fun fp1(x1,p1,i) = ...                // fp1 : T x St(Tp1) x N -> Tp1
  fun fc1(x1,p1,i) = ...                // fc1 : T x St(Tp1) x N -> Tc1
  fun fr1(x1,c1,i) = ...                // fr1 : T x St(Tc1) x N -> D

  dep p1(i-k) -> p1(i)
  dep p1(i[+/-]k) -> c1(i)
  dep c1(i[+/-]k) -> r1(i)

  lbnd A = \x1. ...
  ubnd A = \x1. ...
  prop A = \i. ...

  PCR A(x1) :                           // x1 \in T
    par
      p1 = produce fp1 x1 p1
      c1 = consume fc1 x1 p1
      r1 = reduce B id1 (fr1 x1 c1)     // r1 \in D, Op1(x,y) = B(<x,y>)

  // PCR B
                                        // T2 = D x D
  fun fp2(x2,p2,j) = ...                // fp2 : T2 x St(Tp2) x N -> Tp2
  fun fc2(x2,p2,j) = ...                // fc2 : T2 x St(Tp2) x N -> Tc2
  fun fr2(x2,c2,j) = ...                // fr2 : T2 x St(Tc2) x N -> D

  dep p2(i-k) -> p2(i)
  dep p2(i[+/-]k) -> c2(i)
  dep c2(i[+/-]k) -> r2(i)

  lbnd B = \x2. ...
  ubnd B = \x2. ...
  prop B = \j. ...

  PCR B(x2) :                           // x2 \in T2
    par
      p2 = produce fp1 x2 p2
      c2 = consume fc2 x2 p2
      r2 = reduce Op2 id2 (fr2 x2 c2)   // r2 \in D
 ----------------------------------------------------------------
```

EXTENDS *AbstractAlgebra*, *Naturals*, *Sequences*, *Bags*, *SeqUtils*, *ArithUtils*, *TLC*

PCR A constants and variables

CONSTANTS $I0$, $pre(\_)$,
T, $Tp1$, $Tc1$, D,
id,
$lBnd1(\_)$, $uBnd1(\_)$, $prop1(\_)$,
$fp1(\_, \_, \_)$, $fc1(\_, \_, \_)$, $fr1(\_, \_, \_)$, $gp1(\_, \_)$,
Dep_pp1, Dep_pc1, Dep_cr1

VARIABLES in, $X1$, $p1$, $c1$, $r1$, $rs1$

PCR B constants and variables

CONSTANTS $Tp2$, $Tc2$,
$Op2(\_, \_)$,
$lBnd2(\_)$, $uBnd2(\_)$, $prop2(\_)$,
$fp2(\_, \_, \_)$, $fc2(\_, \_, \_)$, $fr2(\_, \_, \_)$, $gp2(\_, \_)$,
Dep_pp2, Dep_pc2, Dep_cr2

$\textsc{variables}\ X2,\ p2,\ c2,\ r2,\ rs2$

General definitions

$Undef \triangleq \textsc{choose}\ x : x \notin \textsc{union}\ \{T,\ Tp1,\ Tp2,\ Tc1,\ Tc2,\ D\}$

$$\begin{aligned}
&wrt(v) &&\triangleq v \neq Undef\\
&wrts(v,\ S) &&\triangleq \forall\, k \in S : wrt(v[k])\\
&eqs(v1,\ v2,\ S) &&\triangleq \forall\, k \in S : wrt(v1[k]) \land v1[k] = v2[k]
\end{aligned}$$

PCR A definitions and assumptions

$$\begin{aligned}
&IndexA &&\triangleq Seq(Nat)\\
&AssigA &&\triangleq Nat\\
&ItA(x) &&\triangleq \{i \in lBnd1(x)\ ..\ uBnd1(x) : prop1(i)\}\\
&WDIndexA &&\triangleq \{I \in IndexA : wrt(X1[I])\}\\
&StA(R) &&\triangleq [AssigA \rightarrow R \cup \{Undef\}]\\
&redA(I,\ i) &&\triangleq rs1[I][i]\\
&endA(I) &&\triangleq \forall\, i \in ItA(X1[I]) : redA(I,\ i)
\end{aligned}$$

$$\begin{aligned}
&depsA(x,\ d,\ i) \triangleq\\
&\qquad\quad \{i - k : k \in \{k \in d[1] : i - k \geq lBnd1(x) \land prop1(i - k)\}\}\\
&\quad \cup \quad \{i\}\\
&\quad \cup \quad \{i + k : k \in \{k \in d[2] : i + k \leq uBnd1(x) \land prop1(i + k)\}\}
\end{aligned}$$

$$\begin{aligned}
&\textsc{axiom}\ H\_TypeA \triangleq\\
&\quad \land I0 \in IndexA\\
&\quad \land \forall\, x \in T : lBnd1(x) \in Nat\\
&\quad \land \forall\, x \in T : uBnd1(x) \in Nat\\
&\quad \land \forall\, i \in Nat : prop1(i) \in \textsc{boolean}\\
&\quad \land \forall\, x \in T : pre(x) \in \textsc{boolean}\\
&\quad \land Dep\_pp1 \in (\textsc{subset}\ (Nat \setminus \{0\})) \times (\textsc{subset}\ \{\})\\
&\quad \land Dep\_pc1 \in (\textsc{subset}\ (Nat \setminus \{0\})) \times (\textsc{subset}\ (Nat \setminus \{0\}))\\
&\quad \land Dep\_cr1 \in (\textsc{subset}\ (Nat \setminus \{0\})) \times (\textsc{subset}\ (Nat \setminus \{0\}))
\end{aligned}$$

$$\begin{aligned}
&\textsc{axiom}\ H\_BFunTypeA \triangleq\\
&\quad \forall\, x \in T,\ i \in AssigA :\\
&\qquad \land gp1(x,\ i) \in Tp1 \cup \{Undef\}\\
&\qquad \land \forall\, vp \in StA(Tp1) : fp1(x,\ vp,\ i) \in Tp1 \cup \{Undef\}\\
&\qquad \land \forall\, vp \in StA(Tp1) : fc1(x,\ vp,\ i) \in Tc1 \cup \{Undef\}\\
&\qquad \land \forall\, vc \in StA(Tc1) : fr1(x,\ vc,\ i) \in D \cup \{Undef\}
\end{aligned}$$

$$\begin{aligned}
&\textsc{axiom}\ H\_BFunWDA \triangleq\\
&\quad \forall\, x \in T : \forall\, i \in ItA(x) :\\
&\qquad \land gp1(x,\ i) \in Tp1\\
&\qquad \land \forall\, vp \in StA(Tp1) : wrts(vp,\ depsA(x,\ Dep\_pp1,\ i) \setminus \{i\}) \Rightarrow fp1(x,\ vp,\ i) \in Tp1\\
&\qquad \land \forall\, vp \in StA(Tp1) : wrts(vp,\ depsA(x,\ Dep\_pc1,\ i)) \Rightarrow fc1(x,\ vp,\ i) \in Tc1\\
&\qquad \land \forall\, vc \in StA(Tc1) : wrts(vc,\ depsA(x,\ Dep\_cr1,\ i)) \Rightarrow fr1(x,\ vc,\ i) \in D
\end{aligned}$$

$$\begin{aligned}
&\textsc{axiom}\ H\_fpRelevanceA \triangleq\\
&\quad \forall\, x \in T : \forall\, i \in ItA(x),\ vp1 \in StA(Tp1),\ vp2 \in StA(Tp1) :\\
&\qquad eqs(vp1,\ vp2,\ depsA(x,\ Dep\_pp1,\ i) \setminus \{i\}) \Rightarrow fp1(x,\ vp1,\ i) = fp1(x,\ vp2,\ i)
\end{aligned}$$

AXIOM $H\_fcRelevanceA \triangleq$
$\quad \forall x \in T : \forall i \in ItA(x),\ vp1 \in StA(Tp1),\ vp2 \in StA(Tp1) :$
$\qquad eqs(vp1,\ vp2,\ depsA(x,\ Dep\_pc1,\ i)) \Rightarrow fc1(x,\ vp1,\ i) = fc1(x,\ vp2,\ i)$

AXIOM $H\_frRelevanceA \triangleq$
$\quad \forall x \in T : \forall i \in ItA(x),\ vc1 \in StA(Tc1),\ vc2 \in StA(Tc1) :$
$\qquad eqs(vc1,\ vc2,\ depsA(x,\ Dep\_cr1,\ i)) \Rightarrow fr1(x,\ vc1,\ i) = fr1(x,\ vc2,\ i)$

LEMMA $H\_ProdEqInvA \triangleq$
$\quad \forall x \in T : \forall i \in ItA(x) :$
$\qquad wrt(p1[I0][i]) \Rightarrow fp1(x,\ p1[I0],\ i) = gp1(x,\ i)$

PCR B definitions and assumptions

$IndexB \triangleq Seq(Nat)$
$AssigB \triangleq Nat$
$ItB(x) \triangleq \{i \in lBnd2(x) \mathrel{..} uBnd2(x) : prop2(i)\}$
$WDIndexB \triangleq \{I \in IndexB : wrt(X2[I])\}$
$StB(R) \triangleq [AssigB \rightarrow R \cup \{Undef\}]$
$redB(I,\ i) \triangleq rs2[I][i]$
$endB(I) \triangleq \forall i \in ItB(X2[I]) : redB(I,\ i)$

$depsB(x,\ d,\ i) \triangleq$
$\qquad \{i - k : k \in \{k \in d[1] : i - k \geq lBnd2(x) \land prop2(i-k)\}\}$
$\quad \cup \quad \{i\}$
$\quad \cup \quad \{i + k : k \in \{k \in d[2] : i + k \leq uBnd2(x) \land prop2(i+k)\}\}$

$T2 \triangleq D \times D$

AXIOM $H\_TypeB \triangleq$
$\quad \land \forall x \in T2 : lBnd2(x) \in Nat$
$\quad \land \forall x \in T2 : uBnd2(x) \in Nat$
$\quad \land \forall i \in Nat : prop2(i) \in$ BOOLEAN
$\quad \land Dep\_pp2 \in (\text{SUBSET } (Nat \setminus \{0\})) \times (\text{SUBSET } \{\})$
$\quad \land Dep\_pc2 \in (\text{SUBSET } (Nat \setminus \{0\})) \times (\text{SUBSET } (Nat \setminus \{0\}))$
$\quad \land Dep\_cr2 \in (\text{SUBSET } (Nat \setminus \{0\})) \times (\text{SUBSET } (Nat \setminus \{0\}))$

AXIOM $H\_BFunTypeB \triangleq$
$\quad \forall x \in T2,\ i \in AssigB :$
$\qquad \land gp2(x,\ i) \in Tp2 \cup \{Undef\}$
$\qquad \land \forall vp \in StB(Tp2) : fp2(x,\ vp,\ i) \in Tp2 \cup \{Undef\}$
$\qquad \land \forall vp \in StB(Tp2) : fc2(x,\ vp,\ i) \in Tc2 \cup \{Undef\}$
$\qquad \land \forall vc \in StB(Tc2) : fr2(x,\ vc,\ i) \in D \cup \{Undef\}$

AXIOM $H\_BFunWDB \triangleq$
$\quad \forall x \in T2 : \forall i \in ItB(x) :$
$\qquad \land gp2(x,\ i) \in Tp2$
$\qquad \land \forall vp \in StB(Tp2) : wrts(vp,\ depsB(x,\ Dep\_pp2,\ i) \setminus \{i\}) \Rightarrow fp2(x,\ vp,\ i) \in Tp2$
$\qquad \land \forall vp \in StB(Tp2) : wrts(vp,\ depsB(x,\ Dep\_pc2,\ i)) \Rightarrow fc2(x,\ vp,\ i) \in Tc2$
$\qquad \land \forall vc \in StB(Tc2) : wrts(vc,\ depsB(x,\ Dep\_cr2,\ i)) \Rightarrow fr2(x,\ vc,\ i) \in D$

AXIOM $H\_fpRelevanceB \triangleq$
$\quad \forall x \in T2 : \forall i \in ItB(x),\ vp1 \in StB(Tp2),\ vp2 \in StB(Tp2) :$
$\qquad eqs(vp1,\ vp2,\ depsB(x,\ Dep\_pp2,\ i) \setminus \{i\}) \Rightarrow fp2(x,\ vp1,\ i) = fp2(x,\ vp2,\ i)$

AXIOM $H\_fcRelevanceB \triangleq$

$\forall\, x \in T2 : \forall\, i \in ItB(x),\ vp1 \in StB(Tp2),\ vp2 \in StB(Tp2) :$
$\quad eqs(vp1,\ vp2,\ depsB(x,\ Dep\_pc2,\ i)) \Rightarrow fc2(x,\ vp1,\ i) = fc2(x,\ vp2,\ i)$

AXIOM $H\_frRelevanceB \triangleq$
$\forall\, x \in T2 : \forall\, i \in ItB(x),\ vc1 \in StB(Tc2),\ vc2 \in StB(Tc2) :$
$\quad eqs(vc1,\ vc2,\ depsB(x,\ Dep\_cr2,\ i)) \Rightarrow fr2(x,\ vc1,\ i) = fr2(x,\ vc2,\ i)$

LEMMA $H\_ProdEqInvB \triangleq$
$\forall\, I \in WDIndexB : \forall\, i \in ItB(X2[I]) :$
$\quad wrt(p1[I][i]) \Rightarrow fp2(X2[I],\ p2[I],\ i) = gp2(X2[I],\ i)$

Functional specification

$M2 \triangleq$ INSTANCE $AbelianMonoidBigOp$
$\quad$ WITH $D \leftarrow D,\ Id \leftarrow id,\ \otimes \leftarrow Op2$

AXIOM $H\_AlgebraB \triangleq AbelianMonoid(D,\ id,\ Op2)$

$Gp2(x) \triangleq [i \in AssigB \mapsto gp2(x,\ i)]$
$Fc2(x,\ vc) \triangleq [i \in AssigB \mapsto fc2(x,\ vc,\ i)]$
$Fr2(x,\ vc) \triangleq [i \in AssigB \mapsto fr2(x,\ vc,\ i)]$

Informal notation:

$$\mathcal{B}(x_2) \triangleq \bigoplus_{j \,\in\, J_{x_2}} \vec{f}^{\,j}_{r_2}(x_2,\ \vec{f}_{c_2}(x_2,\ \vec{g}_{p_2}(x_2))) \qquad \text{where} \quad J_{x_2} \triangleq \{j \in lBnd2(x_2)..uBnd2(x_2) : prop2(j)\}$$

$B(x2) \triangleq M2!BigOpP(lBnd2(x2),\ uBnd2(x2),\ prop2,$
$\qquad\qquad$ LAMBDA $j : Fr2(x2,\ Fc2(x2,\ Gp2(x2)))[j])$

$Op1(x,\ y) \triangleq B(\langle x,\ y\rangle)$

$M1 \triangleq$ INSTANCE $AbelianMonoidBigOp$
$\quad$ WITH $D \leftarrow D,\ Id \leftarrow id,\ \otimes \leftarrow Op1$

AXIOM $H\_AlgebraA \triangleq AbelianMonoid(D,\ id,\ Op1)$

$Gp1(x) \triangleq [i \in AssigA \mapsto gp1(x,\ i)]$
$Fc1(x,\ vp) \triangleq [i \in AssigA \mapsto fc1(x,\ vp,\ i)]$
$Fr1(x,\ vc) \triangleq [i \in AssigA \mapsto fr1(x,\ vc,\ i)]$

Informal notation:

$$\mathcal{A}(x_1) \triangleq \bigotimes_{i \,\in\, I_{x_1}} \vec{f}^{\,i}_{r_1}(x_1,\ \vec{f}_{c_1}(x_1,\ \vec{g}_{p_1}(x_1)))$$

where $I_{x_1} \triangleq \{i \in lBnd1(x_1)..uBnd1(x_1) : prop1(i)\}$ and $x \otimes y \triangleq \mathcal{B}(x, y)$

$A(x1) \triangleq M1!BigOpP(lBnd1(x1),\ uBnd1(x1),\ prop1,$
$\qquad\qquad$ LAMBDA $i : Fr1(x1,\ Fc1(x1,\ Gp1(x1)))[i])$

Operational specification

$vs1 \triangleq \langle X1,\ p1,\ c1,\ r1,\ rs1,\ X2\rangle$
$vs2 \triangleq \langle p2,\ c2,\ r2,\ rs2\rangle$

$InitA \triangleq \land\ in \in T \land pre(in)$
$\qquad\qquad \land\ X1 = [I \in IndexA \mapsto$ IF $I = I0$ THEN in ELSE $Undef]$

```
             ∧ p1  = [I ∈ IndexA ↦ [i ∈ AssigA ↦ Undef]]
             ∧ c1  = [I ∈ IndexA ↦ [i ∈ AssigA ↦ Undef]]
             ∧ rs1 = [I ∈ IndexA ↦ [i ∈ AssigA ↦ FALSE]]
             ∧ r1  = [I ∈ IndexA ↦ id]

InitB ≜      ∧ X2  = [I ∈ IndexB ↦ Undef]
             ∧ p2  = [I ∈ IndexB ↦ [i ∈ AssigB ↦ Undef]]
             ∧ c2  = [I ∈ IndexB ↦ [i ∈ AssigB ↦ Undef]]
             ∧ rs2 = [I ∈ IndexB ↦ [i ∈ AssigB ↦ FALSE]]
             ∧ r2  = [I ∈ IndexB ↦ id]

Init ≜ InitA ∧ InitB

P1(I, i) ≜
    ∧ ¬wrt(p1[I][i])
    ∧ wrts(p1[I], depsA(X1[I], Dep_pp1, i) \ {i})
    ∧ p1′ = [p1 EXCEPT ![I][i] = fp1(X1[I], p1[I], i)]
    ∧ UNCHANGED ⟨X1, c1, r1, rs1, X2⟩

C1(I, i) ≜
    ∧ ¬wrt(c1[I][i])
    ∧ wrts(p1[I], depsA(X1[I], Dep_pc1, i))
    ∧ c1′ = [c1 EXCEPT ![I][i] = fc1(X1[I], p1[I], i)]
    ∧ UNCHANGED ⟨X1, p1, r1, rs1, X2⟩

R1ini(I, i) ≜
   ∧  ¬wrt(X2[I ∘ ⟨i⟩])
   ∧  wrts(c1[I], depsA(X1[I], Dep_cr1, i))
   ∧  ¬∃ k ∈ ItA(X1[I]) : ∧ k ≠ i
                           ∧ wrt(X2[I ∘ ⟨k⟩])
                           ∧ ¬redA(I, k)
   ∧  X2′ = [X2 EXCEPT ![I ∘ ⟨i⟩] = ⟨r1[I], fr1(X1[I], c1[I], i)⟩]
   ∧  UNCHANGED ⟨X1, p1, c1, r1, rs1⟩

R1end(I, i) ≜
   ∧   wrt(X2[I ∘ ⟨i⟩])
   ∧   endB(I ∘ ⟨i⟩)
   ∧   ¬redA(I, i)
   ∧   r1′  = [r1  EXCEPT ![I] = r2[I ∘ ⟨i⟩]]
   ∧   rs1′ = [rs1 EXCEPT ![I][i] = TRUE]
   ∧   UNCHANGED ⟨X1, p1, c1, X2⟩

P2(I, i) ≜
    ∧ ¬wrt(p2[I][i])
    ∧ wrts(p2[I], depsB(X2[I], Dep_pp2, i) \ {i})
    ∧ p2′ = [p2 EXCEPT ![I][i] = fp2(X2[I], p2[I], i)]
    ∧ UNCHANGED ⟨c2, r2, rs2⟩

C2(I, i) ≜
    ∧ ¬wrt(c2[I][i])
    ∧ wrts(p2[I], depsB(X2[I], Dep_pc2, i))
    ∧ c2′ = [c2 EXCEPT ![I][i] = fc2(X2[I], p2[I], i)]
    ∧ UNCHANGED ⟨p2, r2, rs2⟩

R2(I, i) ≜
```

```
    $\land \neg redB(I,\ i)$
    $\land wrts(c2[I],\ depsB(X2[I],\ Dep\_cr2,\ i))$
    $\land r2' \ = [r2 \quad \text{EXCEPT } ![I] \quad = Op2(@,\ fr2(X2[I],\ c2[I],\ i))]$
    $\land rs2' = [rs2 \text{ EXCEPT } ![I][i] \ = \text{TRUE}]$
    $\land \text{UNCHANGED } \langle p2,\ c2 \rangle$

$Done \triangleq \land \forall I \in WDIndexA : endA(I)$
              $\land \forall I \in WDIndexB : endB(I)$
              $\land \text{UNCHANGED } \langle in,\ vs1,\ vs2 \rangle$

$StepA \triangleq \land \exists I \in WDIndexA :$
                    $\exists i \in ItA(X1[I]) : \lor P1(I,\ i)$
                                          $\lor C1(I,\ i)$
                                          $\lor R1ini(I,\ i)$
                                          $\lor R1end(I,\ i)$
               $\land \text{UNCHANGED } \langle in,\ vs2 \rangle$

$StepB \triangleq \land \exists I \in WDIndexB :$
                    $\exists i \in ItB(X2[I]) : \lor P2(I,\ i)$
                                          $\lor C2(I,\ i)$
                                          $\lor R2(I,\ i)$
               $\land \text{UNCHANGED } \langle in,\ vs1 \rangle$

$Next \triangleq StepA \lor StepB \lor Done$

$Spec \triangleq Init \land \Box[Next]_{\langle in,\ vs1,\ vs2 \rangle}$

$FairSpec \triangleq Spec \land \text{WF}_{vs1}(StepA) \land \text{WF}_{vs2}(StepB)$

----------------------------------------------------------------------------

    $PCR$ A properties
$IndexInvA \triangleq WDIndexA = \{I0\}$

$TypeInvA \triangleq$
  $\land in \ \in T$
  $\land X1 \in [IndexA \rightarrow T \cup \{Undef\}] \land X1[I0] = in$
  $\land p1 \ \in [IndexA \rightarrow StA(Tp1)]$
  $\land c1 \ \in [IndexA \rightarrow StA(Tc1)]$
  $\land r1 \ \in [IndexA \rightarrow D]$
  $\land rs1 \in [IndexA \rightarrow [AssigA \rightarrow \text{BOOLEAN}\ ]]$

$PInvA \triangleq$
  $\forall i \in ItA(X1[I0]) :$
    $wrt(p1[I0][i]) \Rightarrow \land wrts(p1[I0],\ depsA(X1[I0],\ Dep\_pp1,\ i))$
                        $\land p1[I0][i] = gp1(X1[I0],\ i)$

$CInvA \triangleq$
  $\forall i \in ItA(X1[I0]) :$
    $wrt(c1[I0][i]) \Rightarrow \land wrts(p1[I0],\ depsA(X2[I0],\ Dep\_pc2,\ i))$
                        $\land c1[I0][i] = fc1(X1[I0],\ p1[I0],\ i)$

$RInv1A \triangleq$
  $\forall i \in ItA(X1[I0]) :$
    $redA(I0,\ i) \Rightarrow \land wrts(c1[I0],\ depsA(X1[I0],\ Dep\_cr1,\ i))$
                      $\land wrt(X2[I0 \circ \langle i \rangle])$
```

```
                        $\land\ endB(I0 \circ \langle i \rangle)$

$RInv2A\ \triangleq$
  $r1[I0] = M1!BigOpP(lBnd1(X1[I0]),\ uBnd1(X1[I0]),$
                              $\text{LAMBDA}\ i : prop1(i) \land redA(I0,\ i),$
                              $\text{LAMBDA}\ i : fr1(X1[I0],\ c1[I0],\ i))$

$InvA\ \triangleq\ \land\ TypeInvA$
             $\land\ IndexInvA$
             $\land\ PInvA$
             $\land\ CInvA$
             $\land\ RInv1A$
             $\land\ RInv2A$

$CorrectnessA\ \triangleq\ endA(I0) \Rightarrow r1[I0] = A(X1[I0])$

$TerminationA\ \triangleq\ \Diamond endA(I0)$
```

PCR B roperties

```
$IndexInvB\ \triangleq\ WDIndexB \subseteq \{I0 \circ \langle i \rangle : i \in AssigA\}$

$TypeInvB\ \triangleq$
  $\land\ X2\ \in [IndexB \to T2 \cup \{Undef\}]$
  $\land\ p2\ \ \in [IndexB \to StB(Tp2)]$
  $\land\ c2\ \ \in [IndexB \to StB(Tc2)]$
  $\land\ r2\ \ \in [IndexB \to D]$
  $\land\ rs2\ \in [IndexB \to [AssigB \to \text{BOOLEAN}\ ]]$

$PInvB\ \triangleq$
  $\forall I \in WDIndexB : \forall i \in ItB(X2[I]) :$
    $wrt(p2[I][i]) \Rightarrow\ \land\ wrts(p2[I],\ depsB(X2[I],\ Dep\_pp2,\ i))$
                        $\land\ p2[I][i] = gp2(X2[I],\ i)$

$CInvB\ \triangleq$
  $\forall I \in WDIndexB : \forall i \in ItB(X2[I]) :$
    $wrt(c2[I][i]) \Rightarrow\ \land\ wrts(p2[I],\ depsB(X2[I],\ Dep\_pc2,\ i))$
                        $\land\ c2[I][i] = fc2(X2[I],\ p2[I],\ i)$

$RInv1B\ \triangleq$
  $\forall I \in WDIndexB : \forall i \in ItB(X2[I]) :$
    $redB(I0,\ i) \Rightarrow wrts(c2[I],\ depsB(X2[I],\ Dep\_cr2,\ i))$

$RInv2B\ \triangleq$
  $\forall I \in WDIndexB :$
    $r2[I] = M2!BigOpP(lBnd2(X2[I]),\ uBnd2(X2[I]),$
                              $\text{LAMBDA}\ j : prop2(j) \land redB(I0,\ j),$
                              $\text{LAMBDA}\ j : fr2(X2[I],\ c2[I],\ j))$

$InvB\ \triangleq\ \land\ TypeInvB$
             $\land\ IndexInvB$
             $\land\ PInvB$
             $\land\ CInvB$
             $\land\ RInv1B$
             $\land\ RInv2B$
```

$CorrectnessB \triangleq \forall I \in WDIndexB : endB(I) \Rightarrow r2[I] = B(X2[I])$

$TerminationB \triangleq \Diamond(\forall I \in WDIndexB : endB(I))$

Conjoint properties

$$\begin{aligned} TypeInv \triangleq\ & \land\ TypeInvA \\ & \land\ TypeInvB \end{aligned}$$

$$\begin{aligned} Inv \triangleq\ & \land\ TypeInv \\ & \land\ InvA \\ & \land\ InvB \end{aligned}$$

$$\begin{aligned} Correctness \triangleq\ & \land\ CorrectnessA \\ & \land\ CorrectnessB \end{aligned}$$

$$\begin{aligned} Termination \triangleq\ & \land\ TerminationA \\ & \land\ TerminationB \end{aligned}$$

Refinement

$$\begin{aligned} & PCR\_A \triangleq \textsc{instance}\ PCR\_A \\ & \quad \textsc{with}\ X \leftarrow X1,\ p \leftarrow p1,\ c \leftarrow c1,\ r \leftarrow r1,\ rs \leftarrow rs1, \\ & \qquad T \leftarrow T,\ Tp \leftarrow Tp1,\ Tc \leftarrow Tc1,\ D \leftarrow D, \\ & \qquad id \leftarrow id,\ Op \leftarrow Op1, \\ & \qquad lBnd \leftarrow lBnd1,\ uBnd \leftarrow uBnd1,\ prop \leftarrow prop1, \\ & \qquad fp \leftarrow fp1,\ fc \leftarrow fc1,\ fr \leftarrow fr1,\ gp \leftarrow gp1, \\ & \qquad Dep\_pp \leftarrow Dep\_pp1,\ Dep\_pc \leftarrow Dep\_pc1,\ Dep\_cr \leftarrow Dep\_cr1 \end{aligned}$$

B.6 Divide and conquer (DC)

MODULE *PCR_DC*

Divide-and-conquer *PCR*.

```
---------------------------------------------------------------
fun div(x)        = ...                    //    div : T -> Seq(T)
fun isBase(x,p,i) = ...                    // isBase : T x St(T) x N -> Bool
fun base(x,p,i)   = ...                    //   base : T x St(T) x N -> D
fun fr(x,c,i)     = ...                    //     fr : T x St(D) x N -> D

fun iterDiv(x,p,i)    = div(x)[i]
fun subproblem(x,p,i) = if isBase(x,p,i)
                        then base(x,p,i)
                        else DC(p)
fun conquer(r,x,c,i) = Op(r, fr(x,c,i))

dep p(i[+/-]k) -> c(i)
dep c(i[+/-]k) -> r(i)

lbnd DC = \x. 1
ubnd DC = \x. len(div(x))

PCR DC(x)                                   // x \in T
  par
    p = produce iterDiv x p
    c = consume subproblem x p
    r = reduce conquer id x c               // r \in D
---------------------------------------------------------------
```

EXTENDS *AbstractAlgebra*, *Naturals*, *Sequences*, *Bags*, *SeqUtils*, *ArithUtils*, *TLC*

PCR constants and variables

CONSTANTS $I0$, $pre(\_)$,
T, D,
id, $Op(\_, \_)$,
$div(\_)$, $isBase(\_, \_, \_)$, $base(\_, \_, \_)$, $fr(\_, \_, \_)$,
Dep_pc, Dep_cr

VARIABLES in, X, p, c, r, rs

General definitions

$Undef \triangleq$ CHOOSE $x : x \notin T \cup D$

$wrt(v) \triangleq v \neq Undef$
$wrts(v, S) \triangleq \forall k \in S : wrt(v[k])$
$eqs(v1, v2, S) \triangleq \forall k \in S : wrt(v1[k]) \land v1[k] = v2[k]$

PCR definitions and assumptions

$Index \triangleq Seq(Nat)$
$Assig \triangleq Nat$
$uBnd(x) \triangleq Len(div(x))$
$It(x) \triangleq 1 \,..\, uBnd(x)$
$WDIndex \triangleq \{I \in Index : wrt(X[I])\}$
$St(R) \triangleq [Assig \rightarrow R \cup \{Undef\}]$
$red(I, i) \triangleq rs[I][i]$
$end(I) \triangleq \forall i \in It(X[I]) : red(I, i)$

```
$deps(x,\ d,\ i)\ \triangleq$
              $\{i - k : k \in \{k \in d[1] : i - k \geq 1\}\}$
  $\cup$          $\{i\}$
  $\cup$          $\{i + k : k \in \{k \in d[2] : i + k \leq uBnd(x)\}\}$

AXIOM $H\_Type\ \triangleq$
  $\land\ I0\ \ \in Index$
  $\land\ \forall\, x\ \ \in T : uBnd(x) \in Nat$
  $\land\ \forall\, x\ \ \in T : pre(x) \in$ BOOLEAN
  $\land\ Dep\_pc \in ($SUBSET $(Nat \setminus \{0\})) \times ($SUBSET $(Nat \setminus \{0\}))$
  $\land\ Dep\_cr \in ($SUBSET $(Nat \setminus \{0\})) \times ($SUBSET $(Nat \setminus \{0\}))$

AXIOM $H\_BFunType\ \triangleq$
  $\forall\, x \in T,\ i \in Assig :$
    $\land\ div(x) \in Seq(T) \cup \{Undef\}$
    $\land\ \forall\, vp \in St(T) : isBase(x,\ vp,\ i) \in$ BOOLEAN $\cup \{Undef\}$
    $\land\ \forall\, vp \in St(T) : base(x,\ vp,\ i)\ \ \ \in D\ \ \cup \{Undef\}$
    $\land\ \forall\, vc \in St(D) : fr(x,\ vc,\ i)\ \ \ \ \ \ \in D\ \ \cup \{Undef\}$

AXIOM $H\_BFunWD\ \triangleq$
  $\forall\, x \in T : \forall\, i \in It(x) :$
    $\land\ div(x) \in Seq(T)$
    $\land\ \forall\, vp \in St(T) : wrts(vp,\ deps(x,\ Dep\_pc,\ i)) \Rightarrow isBase(x,\ vp,\ i) \in$ BOOLEAN
    $\land\ \forall\, vp \in St(T) : wrts(vp,\ deps(x,\ Dep\_pc,\ i)) \Rightarrow base(x,\ vp,\ i) \in D$
    $\land\ \forall\, vc \in St(D) : wrts(vc,\ deps(x,\ Dep\_cr,\ i)) \Rightarrow fr(x,\ vc,\ i) \in D$

AXIOM $H\_fcRelevance\ \triangleq$
  $\forall\, x \in T : \forall\, i \in It(x),\ vp1 \in St(T),\ vp2 \in St(T) :$
    $eqs(vp1,\ vp2,\ deps(x,\ Dep\_pc,\ i)) \Rightarrow isBase(x,\ vp1,\ i) = isBase(x,\ vp2,\ i)$

AXIOM $H\_baseRelevance\ \triangleq$
  $\forall\, x \in T : \forall\, i \in It(x),\ vp1 \in St(T),\ vp2 \in St(T) :$
    $eqs(vp1,\ vp2,\ deps(x,\ Dep\_pc,\ i)) \Rightarrow base(x,\ vp1,\ i) = base(x,\ vp2,\ i)$

AXIOM $H\_frRelevance\ \triangleq$
  $\forall\, x \in T : \forall\, i \in It(x),\ vc1 \in St(D),\ vc2 \in St(D) :$
    $eqs(vc1,\ vc2,\ deps(x,\ Dep\_cr,\ i)) \Rightarrow fr(x,\ vc1,\ i) = fr(x,\ vc2,\ i)$
```

Functional specification

```
$M\ \triangleq$ INSTANCE $AbelianMonoidBigOp$
   WITH $D \leftarrow D,\ Id \leftarrow id,\ \otimes \leftarrow Op$

AXIOM $H\_Algebra\ \triangleq\ AbelianMonoid(D,\ id,\ Op)$

$Fr(x,\ vc)\ \triangleq\ [i \in Assig \mapsto fr(x,\ vc,\ i)]$
```

Informal notation:

$$\mathcal{DC}(x) \quad \triangleq \quad \bigotimes_{i=1}^{len(div(x))} \vec{f}_r^{\,i}\Big(x,\ i \in \mathbb{N} \mapsto \big(isBase(x, \overrightarrow{div}(x), i) \to base(x, \overrightarrow{div}(x), i),\ \mathcal{DC}(\overrightarrow{div}^{\,i}(x))\big)\Big)$$

```
RECURSIVE $DC(\_)$
$DC(x)\ \triangleq\ M\,!\,BigOp(1,\ uBnd(x),$
                       LAMBDA $i : Fr(x,\ [k \in Assig \mapsto$ IF $isBase(x,\ div(x),\ k)$
                                                     THEN $base(x,\ div(x),\ k)$
```

```
                                                        ELSE  $DC(div(x)[k])])[i])$
```

Alternatively, DC defined as a recursive function:

```
$DC[x \in T] \triangleq$
  $M\,!\,BigOp(1,\, uBnd(x),$
        LAMBDA  $i : Fr(x,\, [k \in Assig \mapsto$ IF  $isBase(x,\, div(x),\, k)$
                                    THEN $base(x,\, div(x),\, k)$
                                    ELSE  $DC[div(x)[k]]])[i])$
```

Operational specification

```
$vs \triangleq \langle X,\, p,\, c,\, r,\, rs \rangle$

$Init \triangleq$ $\wedge\ in \in T \wedge pre(in)$
         $\wedge\ X = [I \in Index \mapsto$ IF $I = I0$ THEN $in$ ELSE $Undef]$
         $\wedge\ p\ \ = [I \in Index \mapsto [i \in Assig \mapsto Undef]]$
         $\wedge\ c\ \ = [I \in Index \mapsto [i \in Assig \mapsto Undef]]$
         $\wedge\ rs = [I \in Index \mapsto [i \in Assig \mapsto$ FALSE$]]$
         $\wedge\ r\ \ = [I \in Index \mapsto id]$

$P(I,\, i) \triangleq$
   $\wedge\ \neg wrt(p[I][i])$
   $\wedge\ p' = [p$ EXCEPT $![I][i] = div(X[I])[i]]$
   $\wedge$ UNCHANGED $\langle X,\, c,\, r,\, rs \rangle$

$Cbase(I,\, i) \triangleq$
   $\wedge\ \ \neg wrt(c[I][i])$
   $\wedge\ \ wrts(p[I],\, deps(X[I],\, Dep\_pc,\, i))$
   $\wedge\ \ isBase(X[I],\, p[I],\, i)$
   $\wedge\ \ c' = [c$ EXCEPT $![I][i] = base(X[I],\, p[I],\, i)]$
   $\wedge$  UNCHANGED $\langle X,\, p,\, r,\, rs \rangle$

$Cini(I,\, i) \triangleq$
   $\wedge\ \neg wrt(X[I \circ \langle i \rangle])$
   $\wedge\ wrts(p[I],\, deps(X[I],\, Dep\_pc,\, i))$
   $\wedge\ \neg isBase(X[I],\, p[I],\, i)$
   $\wedge\ X' = [X$ EXCEPT $![I \circ \langle i \rangle] = p[I][i]]$
   $\wedge$ UNCHANGED $\langle p,\, c,\, r,\, rs \rangle$

$Cend(I,\, i) \triangleq$
   $\wedge\ \neg wrt(c[I][i])$
   $\wedge\ wrt(X[I \circ \langle i \rangle])$
   $\wedge\ end(I \circ \langle i \rangle)$
   $\wedge\ c' = [c$ EXCEPT $![I][i] = r[I \circ \langle i \rangle]]$
   $\wedge$ UNCHANGED $\langle X,\, p,\, r,\, rs \rangle$

$R(I,\, i) \triangleq$
   $\wedge\ \neg red(I,\, i)$
   $\wedge\ wrts(c[I],\, deps(X[I],\, Dep\_cr,\, i))$
   $\wedge\ r'\ \ = [r$  EXCEPT $![I]\qquad = Op(@,\, fr(X[I],\, c[I],\, i))]$
   $\wedge\ rs' = [rs$ EXCEPT $![I][i]\ = $ TRUE$]$
   $\wedge$ UNCHANGED $\langle X,\, p,\, c \rangle$
```

```
Done ≜ ∧ ∀ I ∈ WDIndex : end(I)
       ∧ UNCHANGED ⟨in, vs⟩

Step  ≜ ∧ ∃ I ∈ WDIndex :
            ∃ i ∈ It(X[I]) : ∨ P(I, i)
                             ∨ Cbase(I, i)
                             ∨ Cini(I, i)
                             ∨ Cend(I, i)
                             ∨ R(I, i)
        ∧ UNCHANGED in

Next  ≜ Step ∨ Done

Spec  ≜ Init ∧ □[Next]_⟨in, vs⟩

FairSpec ≜ Spec ∧ WF_vs(Step)

├──────────────────────────────────────────────────────────────┤

Properties
TypeInv ≜
  ∧ in ∈ T
  ∧ X ∈ [Index → T ∪ {Undef}] ∧ X[I0] = in
  ∧ p  ∈ [Index → St(T)]
  ∧ c  ∈ [Index → St(D)]
  ∧ r  ∈ [Index → D]
  ∧ rs ∈ [Index → [Assig → BOOLEAN ]]

PInv ≜
  ∀ I ∈ WDIndex : ∀ i ∈ It(X[I]) :
    wrt(p[I][i]) ⇒ p[I][i] = div(X[I])[i]

CInv1 ≜
  ∀ I ∈ WDIndex : ∀ i ∈ It(X[I]) :
    wrt(c[I][i]) ∧ ¬isBase(X[I], p[I], i)
      ⇒ ∧ wrts(p[I], deps(X[I], Dep_pc, i))
        ∧ wrt(X[I ∘ ⟨i⟩])
        ∧ c[I][i] = r[I ∘ ⟨i⟩]

CInv2 ≜
  ∀ I ∈ WDIndex : ∀ i ∈ It(X[I]) :
    wrt(c[I][i]) ∧ isBase(X[I], p[I], i)
      ⇒ ∧ wrts(p[I], deps(X[I], Dep_pc, i))
        ∧ c[I][i] = base(X[I], p[I], i)

RInv1 ≜
  ∀ I ∈ WDIndex : ∀ i ∈ It(X[I]) :
    red(I, i) ⇒ wrts(c[I], deps(X[I], Dep_cr, i))

RInv2 ≜
  ∀ I ∈ WDIndex :
    r[I] = M!BigOpP(1, uBnd(X[I]),
                    LAMBDA i : red(I, i),
                    LAMBDA i : fr(X[I], c[I], i))

Inv ≜ ∧ TypeInv
```

$\quad\quad \land\ PInv$

$\quad\quad \land\ CInv1$

$\quad\quad \land\ CInv2$

$\quad\quad \land\ RInv1$

$\quad\quad \land\ RInv2$

$Correctness \triangleq \forall\, I \in WDIndex : end(I) \Rightarrow r[I] = DC(X[I])$

$Termination \triangleq \Diamond(\forall\, I \in WDIndex : end(I))$

B.7 DC with left reducer (DCrLeft)

MODULE *PCR_DCrLeft*

Divide-and-conquer *PCR* with left reducer.

```
-----------------------------------------------------------------
fun div(x)         = ...                    //    div : T -> Seq(T)
fun isBase(x,p,i)  = ...                    // isBase : T x St(T) x N -> Bool
fun base(x,p,i)    = ...                    //   base : T x St(T) x N -> D
fun fr(x,c,i)      = ...                    //     fr : T x St(D) x N -> D

fun iterDiv(x,p,i)    = div(x)[i]
fun subproblem(x,p,i) = if isBase(x,p,i)
                        then base(x,p,i)
                        else DC(p)
fun conquer(r,x,c,i) = Op(r, fr(x,c,i))

dep p(i[+/-]k) -> c(i)
dep c(i[+/-]k) -> r(i)
dep r(i-1) -> r(i)

lbnd DC = \x. 1
ubnd DC = \x. len(div(x))

PCR DC(x)                                   // x \in T
  par
    p = produce iterDiv x p
    c = consume subproblem x p
    r = reduce conquer id x c               // r \in D
-----------------------------------------------------------------
```

EXTENDS *AbstractAlgebra*, *Naturals*, *Sequences*, *Bags*, *SeqUtils*, *ArithUtils*, *TLC*

PCR constants and variables

CONSTANTS $I0,\ pre(\_),$
$T,\ D,$
$id,\ Op(\_, \_),$
$div(\_),\ isBase(\_, \_, \_),\ base(\_, \_, \_),\ fr(\_, \_, \_),$
$Dep\_pc,\ Dep\_cr$

VARIABLES $in,\ X,\ p,\ c,\ r,\ rs$

General definitions

$Undef \triangleq \text{CHOOSE } x : x \notin T \cup D$

$wrt(v) \triangleq v \neq Undef$
$wrts(v, S) \triangleq \forall\, k \in S : wrt(v[k])$
$eqs(v1, v2, S) \triangleq \forall\, k \in S : wrt(v1[k]) \land v1[k] = v2[k]$

PCR definitions and assumptions

$Index \triangleq Seq(Nat)$
$Assig \triangleq Nat$
$uBnd(x) \triangleq Len(div(x))$
$It(x) \triangleq 1 \,..\, uBnd(x)$
$WDIndex \triangleq \{I \in Index : wrt(X[I])\}$
$St(R) \triangleq [Assig \rightarrow R \cup \{Undef\}]$
$red(I, i) \triangleq rs[I][i]$

```
$end(I) \triangleq \forall\, i \in It(X[I]) : red(I,\, i)$

$deps(x,\, d,\, i) \triangleq$
        $\{i - k : k \in \{k \in d[1] : i - k \geq 1\}\}$
    $\cup$  $\{i\}$
    $\cup$  $\{i + k : k \in \{k \in d[2] : i + k \leq uBnd(x)\}\}$

AXIOM $H\_Type \triangleq$
    $\land\, I0 \in Index$
    $\land\, \forall\, x \in T : uBnd(x) \in Nat$
    $\land\, \forall\, x \in T : pre(x) \in \text{BOOLEAN}$
    $\land\, Dep\_pc \in (\text{SUBSET } (Nat \setminus \{0\})) \times (\text{SUBSET } (Nat \setminus \{0\}))$
    $\land\, Dep\_cr \in (\text{SUBSET } (Nat \setminus \{0\})) \times (\text{SUBSET } (Nat \setminus \{0\}))$

AXIOM $H\_BFunType \triangleq$
    $\forall\, x \in T,\, i \in Assig :$
        $\land\, div(x) \in Seq(T) \cup \{Undef\}$
        $\land\, \forall\, vp \in St(T) : isBase(x,\, vp,\, i) \in \text{BOOLEAN} \cup \{Undef\}$
        $\land\, \forall\, vp \in St(T) : base(x,\, vp,\, i) \in D \cup \{Undef\}$
        $\land\, \forall\, vc \in St(D) : fr(x,\, vc,\, i) \in D \cup \{Undef\}$

AXIOM $H\_BFunWD \triangleq$
    $\forall\, x \in T : \forall\, i \in It(x) :$
        $\land\, div(x) \in Seq(T)$
        $\land\, \forall\, vp \in St(T) : wrts(vp,\, deps(x,\, Dep\_pc,\, i)) \Rightarrow isBase(x,\, vp,\, i) \in \text{BOOLEAN}$
        $\land\, \forall\, vp \in St(T) : wrts(vp,\, deps(x,\, Dep\_pc,\, i)) \Rightarrow base(x,\, vp,\, i) \in D$
        $\land\, \forall\, vc \in St(D) : wrts(vc,\, deps(x,\, Dep\_cr,\, i)) \Rightarrow fr(x,\, vc,\, i) \in D$

AXIOM $H\_fcRelevance \triangleq$
    $\forall\, x \in T : \forall\, i \in It(x),\, vp1 \in St(T),\, vp2 \in St(T) :$
        $eqs(vp1,\, vp2,\, deps(x,\, Dep\_pc,\, i)) \Rightarrow isBase(x,\, vp1,\, i) = isBase(x,\, vp2,\, i)$

AXIOM $H\_baseRelevance \triangleq$
    $\forall\, x \in T : \forall\, i \in It(x),\, vp1 \in St(T),\, vp2 \in St(T) :$
        $eqs(vp1,\, vp2,\, deps(x,\, Dep\_pc,\, i)) \Rightarrow base(x,\, vp1,\, i) = base(x,\, vp2,\, i)$

AXIOM $H\_frRelevance \triangleq$
    $\forall\, x \in T : \forall\, i \in It(x),\, vc1 \in St(D),\, vc2 \in St(D) :$
        $eqs(vc1,\, vc2,\, deps(x,\, Dep\_cr,\, i)) \Rightarrow fr(x,\, vc1,\, i) = fr(x,\, vc2,\, i)$

|------------------------------------------------------------------------------|

    Functional specification
$M \triangleq$ INSTANCE $MonoidBigOp$
    WITH $D \leftarrow D,\, Id \leftarrow id,\, \otimes \leftarrow Op$

AXIOM $H\_Algebra \triangleq Monoid(D,\, id,\, Op)$

$Fr(x,\, vc) \triangleq [i \in Assig \mapsto fr(x,\, vc,\, i)]$

    Informal notation:
    $\mathcal{DC}(x) \triangleq \bigotimes_{i=1}^{len(div(x))} \vec{f}_r^{\,i}\big(x,\, i \in \mathbb{N} \mapsto \big(isBase(x, \overrightarrow{div}(x), i) \rightarrow base(x, \overrightarrow{div}(x), i),\ \mathcal{DC}(\overrightarrow{div}^{\,i}(x))\big)\big)$
RECURSIVE $DC(\_)$
$DC(x) \triangleq M!BigOp(1,\, uBnd(x),$
                    LAMBDA $i : Fr(x, [k \in Assig \mapsto$ IF $isBase(x, div(x), k)$
                                                           THEN $base(x, div(x), k)$
                                                           ELSE $DC(div(x)[k])])[i])$
```

Alternatively, DC defined as a recursive function:

```
$DC[x \in T] \triangleq$
  $M\,!\,BigOp(1, uBnd(x),$
        LAMBDA $i : Fr(x, [k \in Assig \mapsto$ IF $isBase(x, div(x), k)$
                                   THEN $base(x, div(x), k)$
                                   ELSE $DC[div(x)[k]]])[i])$
```

Operational specification

```
$vs \triangleq \langle X, p, c, r, rs \rangle$

$Init \triangleq \land in \in T \land pre(in)$
         $\land X = [I \in Index \mapsto$ IF $I = I0$ THEN $in$ ELSE $Undef]$
         $\land p\ \ = [I \in Index \mapsto [i \in Assig \mapsto Undef]]$
         $\land c\ \ = [I \in Index \mapsto [i \in Assig \mapsto Undef]]$
         $\land rs = [I \in Index \mapsto [i \in Assig \mapsto$ FALSE$]]$
         $\land r\ \ = [I \in Index \mapsto id]$

$P(I, i) \triangleq$
   $\land \neg wrt(p[I][i])$
   $\land p' = [p$ EXCEPT $![I][i] = div(X[I])[i]]$
   $\land$ UNCHANGED $\langle X, c, r, rs \rangle$

$Cbase(I, i) \triangleq$
  $\land$  $\neg wrt(c[I][i])$
  $\land$  $wrts(p[I], deps(X[I], Dep\_pc, i))$
  $\land$  $isBase(X[I], p[I], i)$
  $\land$  $c' = [c$ EXCEPT $![I][i] = base(X[I], p[I], i)]$
  $\land$  UNCHANGED $\langle X, p, r, rs \rangle$

$Cini(I, i) \triangleq$
  $\land \neg wrt(X[I \circ \langle i \rangle])$
  $\land wrts(p[I], deps(X[I], Dep\_pc, i))$
  $\land \neg isBase(X[I], p[I], i)$
  $\land X' = [X$ EXCEPT $![I \circ \langle i \rangle] = p[I][i]]$
  $\land$ UNCHANGED $\langle p, c, r, rs \rangle$

$Cend(I, i) \triangleq$
  $\land \neg wrt(c[I][i])$
  $\land wrt(X[I \circ \langle i \rangle])$
  $\land end(I \circ \langle i \rangle)$
  $\land c' = [c$ EXCEPT $![I][i] = r[I \circ \langle i \rangle]]$
  $\land$ UNCHANGED $\langle X, p, r, rs \rangle$

$R(I, i) \triangleq$
   $\land \neg red(I, i)$
   $\land wrts(c[I], deps(X[I], Dep\_cr, i))$
   $\land i - 1 \geq 1 \Rightarrow red(I, i - 1)$
   $\land r'\ \ \ = [r$  EXCEPT $![I]\ \ \ = Op(@, fr(X[I], c[I], i))]$
```

```
    ∧ rs′   = [rs EXCEPT ![I][i] = TRUE]
    ∧ UNCHANGED ⟨X, p, c⟩

Done ≜ ∧ ∀ I ∈ WDIndex : end(I)
       ∧ UNCHANGED ⟨in, vs⟩

Step ≜ ∧ ∃ I ∈ WDIndex :
           ∃ i ∈ It(X[I]) : ∨ P(I, i)
                            ∨ Cbase(I, i)
                            ∨ Cini(I, i)
                            ∨ Cend(I, i)
                            ∨ R(I, i)
       ∧ UNCHANGED in

Next ≜ Step ∨ Done

Spec ≜ Init ∧ □[Next]_⟨in, vs⟩

FairSpec ≜ Spec ∧ WF_vs(Step)

├──────────────────────────────────────────────────────────────┤

  Properties
TypeInv ≜
  ∧ in ∈ T
  ∧ X ∈ [Index → T ∪ {Undef}] ∧ X[I0] = in
  ∧ p  ∈ [Index → St(T)]
  ∧ c  ∈ [Index → St(D)]
  ∧ r  ∈ [Index → D]
  ∧ rs ∈ [Index → [Assig → BOOLEAN ]]

PInv ≜
  ∀ I ∈ WDIndex : ∀ i ∈ It(X[I]) :
    wrt(p[I][i]) ⇒ p[I][i] = div(X[I])[i]

CInv1 ≜
  ∀ I ∈ WDIndex : ∀ i ∈ It(X[I]) :
    wrt(c[I][i]) ∧ ¬isBase(X[I], p[I], i)
      ⇒ ∧ wrts(p[I], deps(X[I], Dep_pc, i))
         ∧ wrt(X[I ∘ ⟨i⟩])
         ∧ c[I][i] = r[I ∘ ⟨i⟩]

CInv2 ≜
  ∀ I ∈ WDIndex : ∀ i ∈ It(X[I]) :
    wrt(c[I][i]) ∧ isBase(X[I], p[I], i)
      ⇒ ∧ wrts(p[I], deps(X[I], Dep_pc, i))
         ∧ c[I][i] = base(X[I], p[I], i)

RInv1 ≜
  ∀ I ∈ WDIndex : ∀ i ∈ It(X[I]) :
    red(I, i) ⇒ ∧ wrts(c[I], deps(X[I], Dep_cr, i))
                ∧ ∀ k ∈ It(X[I]) : k < i ⇒ red(I, k)

RInv2 ≜
  ∀ I ∈ WDIndex : ∀ i ∈ It(X[I]) :
    ¬red(I, i) ⇒ r[I] = M ! BigOpP(1, i − 1,
                                        LAMBDA $j : red(I, j)$,
                                        LAMBDA $j : fr(X[I], c[I], j))$

$Inv \triangleq \land TypeInv$
         $\land PInv$
         $\land CInv1$
         $\land CInv2$
         $\land RInv1$
         $\land RInv2$

$Correctness \triangleq \forall I \in WDIndex : end(I) \Rightarrow r[I] = DC(X[I])$

$Termination \triangleq \Diamond \forall I \in WDIndex : end(I)$

Refinement
$PCR\_DC \triangleq$ INSTANCE $PCR\_DC$
   WITH $X \leftarrow X,\ p \leftarrow p,\ c \leftarrow c,\ r \leftarrow r,\ rs \leftarrow rs,$
        $T \leftarrow T,\ D \leftarrow D,$
        $id \leftarrow id,\ Op \leftarrow Op,$
        $div \leftarrow div,\ isBase \leftarrow isBase,\ base \leftarrow base,\ fr \leftarrow fr,$
        $Dep\_pc \leftarrow Dep\_pc,\ Dep\_cr \leftarrow Dep\_cr$
```

B.8 DC composed through reducer with DCrLeft

MODULE *PCR_DC_r_DCrLeft*

Divide-and-conquer *PCR* composed through reducer with a divide-and-conquer *PCR* with left reducer.

```
----------------------------------------------------------------
// PCR A

fun div1(x1)         = ...                      //    div1 : T -> Seq(T)
fun isBase1(x1,p1,i) = ...                      // isBase1 : T x St(T) -> Bool
fun base1(x1,p1,i)   = ...                      //   base1 : T x St(T) -> D
fun fr1(x1,c1,i)     = ...                      //     fr1 : T x St(D) -> D

fun iterDiv1(x1,p1,i)    = div1(x1)[i]
fun subproblem1(x1,p1,i) = if isBase1(x1,p1,i)
                           then base1(x1,p1,i)
                           else A(p1)
fun conquer1(r1,x1,c1,i) = B(r1, fr1(x1,c1,i))

dep p1(i[+/-]k) -> c1(i)
dep c1(i[+/-]k) -> r1(i)

lbnd A = \x1. 1
ubnd A = \x1. len(div1(x1))

PCR A(x1)                                       // x1 \in T
  par
    p1 = produce iterDiv1 x1
    c1 = consume subproblem1 x1 p1
    r1 = reduce conquer1 id x1 c1               // r1 \in D

// PCR B
                                                // T2 = D x D
fun div2(x2)         = ...                      //    div2 : T2 -> Seq(T2)
fun isBase2(x2,p2,i) = ...                      // isBase2 : T2 x St(T2) x N -> Bool
fun base2(x2,p2,i)   = ...                      //   base2 : T2 x St(T2) x N -> D
fun fr2(x2,c2,i)     = ...                      //     fr2 : T2 x St(D) x N -> D

fun iterDiv2(x2,p2,i)    = div2(x2)[i]
fun subproblem2(x2,p2,i) = if isBase2(x2,p2,i)
                           then base2(x2,p2,i)
                           else B(p2)
fun conquer2(r2,x2,c2,i) = Op2(r2, fr2(x2,c2,i))

dep p2(i[+/-]k) -> c2(i)
dep c2(i[+/-]k) -> r2(i)
dep r2(i-1) -> r2(i)

lbnd B = \x2. 1
ubnd B = \x2. len(div2(x2))

PCR B(x2)                                       // x2 \in T2
  par
    p2 = produce iterDiv2 x2
    c2 = consume subproblem2 x2 p2
    r2 = reduce conquer2 id x2 c2               // r2 \in D
----------------------------------------------------------------
```

EXTENDS *AbstractAlgebra*, *Naturals*, *Sequences*, *Bags*, *SeqUtils*, *ArithUtils*, *TLC*

PCR A constants and variables

CONSTANTS $I0$, $pre(\_)$,
T, D,
id,
$div1(\_)$, $isBase1(\_, \_, \_)$, $base1(\_, \_, \_)$, $fr1(\_, \_, \_)$,
Dep_pc1, Dep_cr1

VARIABLES in, $X1$, $p1$, $c1$, $r1$, $rs1$

```
PCR B constants and variables
CONSTANTS $Op2(\_, \_)$,
          $div2(\_), isBase2(\_, \_, \_), base2(\_, \_, \_), fr2(\_, \_, \_)$,
          $Dep\_pc2, Dep\_cr2$

VARIABLES $X2, p2, c2, r2, rs2$

|-----------------------------------------------------------------------|

General definitions
$Undef \triangleq \text{CHOOSE } x : x \notin T \cup D$

$wrt(v)            \triangleq v \neq Undef$
$wrts(v, S)        \triangleq \forall k \in S : wrt(v[k])$
$eqs(v1, v2, S)    \triangleq \forall k \in S : wrt(v1[k]) \land v1[k] = v2[k]$

|-----------------------------------------------------------------------|

PCR A definitions and assumptions
$IndexA     \triangleq Seq(Nat)$
$AssigA     \triangleq Nat$
$uBnd1(x)   \triangleq Len(div1(x))$
$ItA(x)     \triangleq 1 \,..\, uBnd1(x)$
$WDIndexA   \triangleq \{I \in IndexA : wrt(X1[I])\}$
$StA(R)     \triangleq [AssigA \rightarrow R \cup \{Undef\}]$
$redA(I, i) \triangleq rs1[I][i]$
$endA(I)    \triangleq \forall i \in ItA(X1[I]) : redA(I, i)$

$depsA(x, d, i) \triangleq$
          $\{i - k : k \in \{k \in d[1] : i - k \geq 1\}\}$
  $\cup$      $\{i\}$
  $\cup$      $\{i + k : k \in \{k \in d[2] : i + k \leq uBnd1(x)\}\}$

AXIOM $H\_TypeA \triangleq$
  $\land\ I0 \in IndexA$
  $\land\ \forall x \in T : uBnd1(x) \in Nat$
  $\land\ \forall x \in T : pre(x) \in \text{BOOLEAN}$
  $\land\ Dep\_pc1 \in (\text{SUBSET } (Nat \setminus \{0\})) \times (\text{SUBSET } (Nat \setminus \{0\}))$
  $\land\ Dep\_cr1 \in (\text{SUBSET } (Nat \setminus \{0\})) \times (\text{SUBSET } (Nat \setminus \{0\}))$

AXIOM $H\_BFunTypeA \triangleq$
  $\forall x \in T, i \in AssigA :$
    $\land\ div1(x) \in Seq(T) \cup \{Undef\}$
    $\land\ \forall vp \in StA(T) : isBase1(x, vp, i) \in \text{BOOLEAN} \cup \{Undef\}$
    $\land\ \forall vp \in StA(T) : base1(x, vp, i) \in D \cup \{Undef\}$
    $\land\ \forall vc \in StA(D) : fr1(x, vc, i) \in D \cup \{Undef\}$

AXIOM $H\_BFunWDA \triangleq$
  $\forall x \in T : \forall i \in ItA(x) :$
    $\land\ div1(x) \in Seq(T)$
    $\land\ \forall vp \in StA(T) : wrts(vp, depsA(x, Dep\_pc1, i)) \Rightarrow isBase1(x, vp, i) \in \text{BOOLEAN}$
    $\land\ \forall vp \in StA(T) : wrts(vp, depsA(x, Dep\_pc1, i)) \Rightarrow base1(x, vp, i) \in D$
    $\land\ \forall vc \in StA(D) : wrts(vc, depsA(x, Dep\_cr1, i)) \Rightarrow fr1(x, vc, i) \in D$

AXIOM $H\_isBaseRelevanceA \triangleq$
```

```
  $\forall x \in T : \forall i \in ItA(x),\ vp1 \in StA(T),\ vp2 \in StA(T) :$
    $eqs(vp1,\ vp2,\ depsA(x,\ Dep\_pc1,\ i)) \Rightarrow isBase1(x,\ vp1,\ i) = isBase1(x,\ vp2,\ i)$

AXIOM $H\_baseRelevanceA \triangleq$
  $\forall x \in T : \forall i \in ItA(x),\ vp1 \in StA(T),\ vp2 \in StA(T) :$
    $eqs(vp1,\ vp2,\ depsA(x,\ Dep\_pc1,\ i)) \Rightarrow base1(x,\ vp1,\ i) = base1(x,\ vp2,\ i)$

AXIOM $H\_frRelevanceA \triangleq$
  $\forall x \in T : \forall i \in ItA(x),\ vc1 \in StA(D),\ vc2 \qquad \in StA(D) :$
    $eqs(vc1,\ vc2,\ depsA(x,\ Dep\_cr1,\ i)) \Rightarrow fr1(x,\ vc1,\ i) = fr1(x,\ vc2,\ i)$
```

PCR B definitions and assumptions

```
$IndexB \quad\ \ \triangleq Seq(Nat) \times Seq(Nat)$
$AssigB \quad\ \ \triangleq Nat$
$uBnd2(x) \quad \triangleq Len(div2(x))$
$ItB(x) \qquad \triangleq 1 \mathrel{..} uBnd2(x)$
$WDIndexB \triangleq \{I \in IndexB : wrt(X2[I])\}$
$StB(R) \qquad \triangleq [AssigB \rightarrow R \cup \{Undef\}]$
$redB(I,\ i) \quad \triangleq rs2[I][i]$
$endB(I) \qquad \triangleq \forall i \in ItB(X2[I]) : redB(I,\ i)$

$depsB(x,\ d,\ i) \triangleq$
               $\{i - k : k \in \{k \in d[1] : i - k \geq 1\}\}$
   $\cup$          $\{i\}$
   $\cup$          $\{i + k : k \in \{k \in d[2] : i + k \leq uBnd2(x)\}\}$

$T2 \triangleq D \times D$

AXIOM $H\_TypeB \triangleq$
   $\land \forall x \in T2 : uBnd2(x) \in Nat$
   $\land Dep\_pc2 \in (\text{SUBSET } (Nat \setminus \{0\})) \times (\text{SUBSET } (Nat \setminus \{0\}))$
   $\land Dep\_cr2 \in (\text{SUBSET } (Nat \setminus \{0\})) \times (\text{SUBSET } (Nat \setminus \{0\}))$

AXIOM $H\_BFunTypeB \triangleq$
  $\forall x \in T2,\ i \in AssigB :$
     $\land div2(x) \in Seq(T2) \cup \{Undef\}$
     $\land \forall vp \in StB(T2) : isBase2(x,\ vp,\ i) \in \text{BOOLEAN} \cup \{Undef\}$
     $\land \forall vp \in StB(T2) : base2(x,\ vp,\ i) \quad\ \in D \cup \{Undef\}$
     $\land \forall vc \in StB(D) \ \ : fr2(x,\ vc,\ i) \qquad \in D \cup \{Undef\}$

AXIOM $H\_BFunWDB \triangleq$
  $\forall x \in T2 : \forall i \in ItB(x) :$
     $\land div2(x) \in Seq(T2)$
     $\land \forall vp \in StB(T2) : wrts(vp,\ depsB(x,\ Dep\_pc2,\ i)) \Rightarrow isBase2(x,\ vp,\ i) \in \text{BOOLEAN}$
     $\land \forall vp \in StB(T2) : wrts(vp,\ depsB(x,\ Dep\_pc2,\ i)) \Rightarrow base2(x,\ vp,\ i) \in D$
     $\land \forall vc \in StB(D) \ \ : wrts(vc,\ depsB(x,\ Dep\_cr2,\ i)) \Rightarrow fr2(x,\ vc,\ i) \in D$

AXIOM $H\_isBaseRelevanceB \triangleq$
  $\forall x \in T2 : \forall i \in ItB(x),\ vp1 \in StB(T2),\ vp2 \in StB(T2) :$
    $eqs(vp1,\ vp2,\ depsB(x,\ Dep\_pc2,\ i)) \Rightarrow isBase2(x,\ vp1,\ i) = isBase2(x,\ vp2,\ i)$

AXIOM $H\_baseRelevanceB \triangleq$
  $\forall x \in T2 : \forall i \in ItB(x),\ vp1 \in StB(T2),\ vp2 \qquad \in StB(T2) :$
    $eqs(vp1,\ vp2,\ depsB(x,\ Dep\_pc2,\ i)) \Rightarrow base2(x,\ vp1,\ i) = base2(x,\ vp2,\ i)$
```

```
AXIOM $H\_frRelevanceB \triangleq$
  $\forall x \in T2 : \forall i \in ItB(x),\ vc1 \in StB(D),\ vc2 \in StB(D) :$
    $eqs(vc1,\ vc2,\ depsB(x,\ Dep\_cr2,\ i)) \Rightarrow fr2(x,\ vc1,\ i) = fr2(x,\ vc2,\ i)$
```

Functional specification

```
$M2 \triangleq$ INSTANCE $MonoidBigOp$
  WITH $D \leftarrow D,\ Id \leftarrow id,\ \otimes \leftarrow Op2$

AXIOM $H\_AlgebraB \triangleq Monoid(D,\ id,\ Op2)$

$Fr2(x,\ vc) \triangleq [i \in AssigB \mapsto fr2(x,\ vc,\ i)]$
```

Informal notation:

$$\mathcal{B}(x_2) \triangleq \bigoplus_{j=1}^{len(div2(x_2))} \vec{f}^{\,j}_{r2}\Big(x_2,\ j \in \mathbb{N} \mapsto \big(isBase2(x_2, \overrightarrow{div2}(x_2), i) \rightarrow base2(x_2, \overrightarrow{div2}(x_2), j),\ \mathcal{B}(\overrightarrow{div2}^{\,j}(x_2))\big)\Big)$$

```
RECURSIVE $B(\_)$
$B(x) \triangleq M2!BigOp(1,\ uBnd2(x),$
                LAMBDA $i : Fr2(x,\ [k \in AssigB \mapsto$ IF $isBase2(x,\ div2(x),\ k)$
                                                   THEN $base2(x,\ div2(x),\ k)$
                                                   ELSE $B(div2(x)[k])])[i])$
```

Alternatively, B as a recursive function:

```
$B[x \in D \times D] \triangleq$
  $M2!BigOp(1,\ uBnd2(x),$
      LAMBDA $i : Fr2(x,\ [k \in AssigB \mapsto$ IF $isBase2(x,\ div2(x),\ k)$
                                     THEN $base2(x,\ div2(x),\ k)$
                                     ELSE $B[div2(x)[k]]])[i])$
```

```
$Op1(x,\ y) \triangleq B(\langle x,\ y \rangle)$

$M1 \triangleq$ INSTANCE $AbelianMonoidBigOp$
  WITH $D \leftarrow D,\ Id \leftarrow id,\ \otimes \leftarrow Op1$

AXIOM $H\_AlgebraA \triangleq AbelianMonoid(D,\ id,\ Op1)$

$Fr1(x,\ vc) \triangleq [i \in AssigA \mapsto fr1(x,\ vc,\ i)]$
```

Informal notation:

$$\mathcal{A}(x_1) \triangleq \bigotimes_{i=1}^{len(div1(x_1))} \vec{f}^{\,i}_{r1}\Big(x_1,\ i \in \mathbb{N} \mapsto \big(isBase1(x_1, \overrightarrow{div1}(x_1), i) \rightarrow base1(x_1, \overrightarrow{div1}(x_1), i),\ \mathcal{A}(\overrightarrow{div1}^{\,i}(x_1))\big)\Big)$$

where $x \otimes y \triangleq \mathcal{B}(x, y)$

```
RECURSIVE $A(\_)$
$A(x) \triangleq M1!BigOp(1,\ uBnd1(x),$
                LAMBDA $i : Fr1(x,\ [k \in AssigA \mapsto$ IF $isBase1(x,\ div1(x),\ k)$
                                                   THEN $base1(x,\ div1(x),\ k)$
                                                   ELSE $A(div1(x)[k])])[i])$
```

Alternatively, A as a recursive function:

```
$A[x \in T] \triangleq$
  $M1!BigOp(1,\ uBnd1(x),$
      LAMBDA $i : Fr1(x,\ [k \in AssigA \mapsto$ IF $isBase1(x,\ div1(x),\ k)$
```

$\qquad\qquad\qquad\qquad\textsc{then}\ \mathit{base}1(x,\ \mathit{div}1(x),\ k)$

$\qquad\qquad\qquad\qquad\textsc{else}\ \ A[\mathit{div}1(x)[k]]])[i])$

Operational specification

$\mathit{vs}1 \triangleq \langle X1,\ p1,\ c1,\ r1,\ \mathit{rs}1\rangle$

$\mathit{vs}2 \triangleq \langle p2,\ c2,\ r2,\ \mathit{rs}2\rangle$

$$\begin{aligned}
\mathit{InitA} \triangleq\ & \land\ \mathit{in} \in T \land \mathit{pre}(\mathit{in})\\
& \land\ X1 \ = [I \in \mathit{IndexA} \mapsto \textsc{if}\ I = I0\ \textsc{then}\ \mathit{in}\ \textsc{else}\ \ \mathit{Undef}]\\
& \land\ p1 \ = [I \in \mathit{IndexA} \mapsto [i \in \mathit{AssigA} \mapsto \mathit{Undef}]]\\
& \land\ c1 \ = [I \in \mathit{IndexA} \mapsto [i \in \mathit{AssigA} \mapsto \mathit{Undef}]]\\
& \land\ \mathit{rs}1 = [I \in \mathit{IndexA} \mapsto [i \in \mathit{AssigA} \mapsto \textsc{false}]]\\
& \land\ r1 \ = [I \in \mathit{IndexA} \mapsto \mathit{id}]
\end{aligned}$$

$$\begin{aligned}
\mathit{InitB} \triangleq\ & \land\ X2 \ = [I \in \mathit{IndexB} \mapsto \mathit{Undef}]\\
& \land\ p2 \ = [I \in \mathit{IndexB} \mapsto [i \in \mathit{AssigB} \mapsto \mathit{Undef}]]\\
& \land\ c2 \ = [I \in \mathit{IndexB} \mapsto [i \in \mathit{AssigB} \mapsto \mathit{Undef}]]\\
& \land\ \mathit{rs}2 = [I \in \mathit{IndexB} \mapsto [i \in \mathit{AssigB} \mapsto \textsc{false}]]\\
& \land\ r2 \ = [I \in \mathit{IndexB} \mapsto \mathit{id}]
\end{aligned}$$

$\mathit{Init} \triangleq \mathit{InitA} \land \mathit{InitB}$

$$\begin{aligned}
& P1(I,\ i) \triangleq\\
&\quad \land \neg \mathit{wrt}(p1[I][i])\\
&\quad \land p1' = [p1\ \textsc{except}\ ![I][i] = \mathit{div}1(X1[I])[i]]\\
&\quad \land \textsc{unchanged}\ \langle X1,\ c1,\ r1,\ \mathit{rs}1,\ X2\rangle
\end{aligned}$$

$$\begin{aligned}
& C1\mathit{base}(I,\ i) \triangleq\\
&\quad \land \neg \mathit{wrt}(c1[I][i])\\
&\quad \land \mathit{wrts}(p1[I],\ \mathit{depsA}(X1[I],\ \mathit{Dep\_pc}1,\ i))\\
&\quad \land \mathit{isBase}1(X1[I],\ p1[I],\ i)\\
&\quad \land c1' = [c1\ \textsc{except}\ ![I][i] = \mathit{base}1(X1[I],\ p1[I],\ i)]\\
&\quad \land \textsc{unchanged}\ \langle X1,\ p1,\ r1,\ \mathit{rs}1,\ X2\rangle
\end{aligned}$$

$$\begin{aligned}
& C1\mathit{ini}(I,\ i) \triangleq\\
&\quad \land\ \ \neg \mathit{wrt}(X1[I \circ \langle i\rangle])\\
&\quad \land\ \ \mathit{wrts}(p1[I],\ \mathit{depsA}(X1[I],\ \mathit{Dep\_pc}1,\ i))\\
&\quad \land\ \ \neg \mathit{isBase}1(X1[I],\ p1[I],\ i)\\
&\quad \land\ \ X1' = [X1\ \textsc{except}\ ![I \circ \langle i\rangle] = p1[I][i]]\\
&\quad \land\ \ \textsc{unchanged}\ \langle p1,\ c1,\ r1,\ \mathit{rs}1,\ X2\rangle
\end{aligned}$$

$$\begin{aligned}
& C1\mathit{end}(I,\ i) \triangleq\\
&\quad \land\ \ \neg \mathit{wrt}(c1[I][i])\\
&\quad \land\ \ \mathit{wrt}(X1[I \circ \langle i\rangle])\\
&\quad \land\ \ \mathit{endA}(I \circ \langle i\rangle)\\
&\quad \land\ \ c1' = [c1\ \textsc{except}\ ![I][i] = r1[I \circ \langle i\rangle]]\\
&\quad \land\ \ \textsc{unchanged}\ \langle X1,\ p1,\ r1,\ \mathit{rs}1,\ X2\rangle
\end{aligned}$$

$$\begin{aligned}
& R1\mathit{ini}(I,\ i) \triangleq\\
&\quad \land\ \ \neg \mathit{wrt}(X2[\langle I,\ \langle i\rangle\rangle])\\
&\quad \land\ \ \mathit{wrts}(c1[I],\ \mathit{depsA}(X1[I],\ \mathit{Dep\_cr}1,\ i))\\
&\quad \land\ \ \neg\exists\, k \in \mathit{ItA}(X1[I]) :\ \land\ k \neq i\\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \ \land\ \mathit{wrt}(X2[\langle I,\ \langle k\rangle\rangle])
\end{aligned}$$

```
                                   $\land \neg redA(I, k)$
    $\land\ X2' = [X2 \text{ EXCEPT } ![\langle I, \langle i\rangle\rangle] = \langle r1[I], fr1(X1[I], c1[I], i)\rangle]$
    $\land$ UNCHANGED $\langle X1, p1, c1, r1, rs1\rangle$

$R1end(I, i) \triangleq$
    $\land\ \neg redA(I, i)$
    $\land\ wrt(X2[\langle I, \langle i\rangle\rangle])$
    $\land\ endB(\langle I, \langle i\rangle\rangle)$
    $\land\ r1' = [r1 \text{ EXCEPT } ![I] = r2[\langle I, \langle i\rangle\rangle]]$
    $\land\ rs1' = [rs1 \text{ EXCEPT } ![I][i] = \text{TRUE}]$
    $\land$ UNCHANGED $\langle X1, p1, c1, X2\rangle$

PCR B

$P2(I, i) \triangleq$
     $\land \neg wrt(p2[I][i])$
     $\land\ p2' = [p2 \text{ EXCEPT } ![I][i] = div2(X2[I])[i]]$
     $\land$ UNCHANGED $\langle c2, r2, rs2, X2\rangle$

$C2base(I, i) \triangleq$
    $\land \neg wrt(c2[I][i])$
    $\land\ wrts(p2[I], depsB(X2[I], Dep\_pc2, i))$
    $\land\ isBase2(X2[I], p2[I], i)$
    $\land\ c2' = [c2 \text{ EXCEPT } ![I][i] = base2(X2[I], p2[I], i)]$
    $\land$ UNCHANGED $\langle p2, r2, rs2, X2\rangle$

$C2ini(I, i) \triangleq$
    $\land\ \neg wrt(X2[\langle I[1], I[2] \circ \langle i\rangle\rangle])$
    $\land\ wrts(p2[I], depsB(X2[I], Dep\_pc2, i))$
    $\land\ \neg isBase2(X2[I], p2[I], i)$
    $\land\ X2' = [X2 \text{ EXCEPT } ![\langle I[1], I[2] \circ \langle i\rangle\rangle] = p2[I][i]]$
    $\land$ UNCHANGED $\langle p2, c2, r2, rs2\rangle$

$C2end(I, i) \triangleq$
    $\land\ \neg wrt(c2[I][i])$
    $\land\ wrt(X2[\langle I[1], I[2] \circ \langle i\rangle\rangle])$
    $\land\ endB(\langle I[1], I[2] \circ \langle i\rangle\rangle)$
    $\land\ c2' = [c2 \text{ EXCEPT } ![I][i] = r2[\langle I[1], I[2] \circ \langle i\rangle\rangle]]$
    $\land$ UNCHANGED $\langle p2, r2, rs2, X2\rangle$

$R2(I, i) \triangleq$
     $\land \neg redB(I, i)$
     $\land\ wrts(c2[I], depsB(X2[I], Dep\_cr2, i))$
     $\land\ i - 1 \geq 1 \Rightarrow redB(I, i - 1)$
     $\land\ r2' = [r2 \text{ EXCEPT } ![I] = Op2(@, fr2(X2[I], c2[I], i))]$
     $\land\ rs2' = [rs2 \text{ EXCEPT } ![I][i] = \text{TRUE}]$
     $\land$ UNCHANGED $\langle p2, c2, X2\rangle$

$Done \triangleq \land \forall I \in WDIndexA : endA(I)$
              $\land \forall I \in WDIndexB : endB(I)$
              $\land$ UNCHANGED $\langle in, vs1, X2, vs2\rangle$

$StepA \triangleq \land \exists I \in WDIndexA :$
                 $\exists i \in ItA(X1[I]) : \lor P1(I, i)$
                                             $\lor C1base(I, i)$
```

```
                                        $\lor\ C1ini(I,\ i)$
                                        $\lor\ C1end(I,\ i)$
                                        $\lor\ R1ini(I,\ i)$
                                        $\lor\ R1end(I,\ i)$
            $\land$ UNCHANGED $\langle in,\ vs2\rangle$

$StepB\ \triangleq\ \land\ \exists\, I \in WDIndexB :$
                $\exists\, i \in ItB(X2[I]) :$  $\lor\ P2(I,\ i)$
                                        $\lor\ C2base(I,\ i)$
                                        $\lor\ C2ini(I,\ i)$
                                        $\lor\ C2end(I,\ i)$
                                        $\lor\ R2(I,\ i)$
            $\land$ UNCHANGED $\langle in,\ vs1\rangle$

$Next\ \triangleq\ StepA \lor StepB \lor Done$

$Spec\ \triangleq\ Init \land \Box[Next]_{\langle in,\ vs1,\ vs2,\ X2\rangle}$

$FairSpec\ \triangleq\ Spec \land \mathrm{WF}_{\langle vs1,\ X2\rangle}(StepA) \land \mathrm{WF}_{\langle vs2,\ X2\rangle}(StepB)$
```

PCR A properties

```
$TypeInvA\ \triangleq$
  $\land\ in\ \ \in T$
  $\land\ X1 \in [IndexA \to T \cup \{Undef\}] \land X1[I0] = in$
  $\land\ p1\ \ \in [IndexA \to StA(T)]$
  $\land\ c1\ \ \in [IndexA \to StA(D)]$
  $\land\ r1\ \ \in [IndexA \to D]$
  $\land\ rs1 \in [IndexA \to [AssigA \to$ BOOLEAN $]]$

$PInvA\ \triangleq$
  $\forall\, I \in WDIndexA : \forall\, i \in ItA(X1[I]) :$
    $wrt(p1[I][i]) \Rightarrow p1[I][i] = div1(X1[I])[i]$

$CInv1A\ \triangleq$
  $\forall\, I \in WDIndexA : \forall\, i \in ItA(X1[I]) :$
    $wrt(c1[I][i]) \land \neg isBase1(X1[I],\ p1[I],\ i)$
      $\Rightarrow\ \land\ wrts(p1[I],\ depsA(X1[I],\ Dep\_pc1,\ i))$
          $\land\ wrt(X1[I \circ \langle i\rangle])$
          $\land\ endA(I \circ \langle i\rangle)$
          $\land\ c1[I][i] = r1[I \circ \langle i\rangle]$

$CInv2A\ \triangleq$
  $\forall\, I \in WDIndexA : \forall\, i \in ItA(X1[I]) :$
    $wrt(c1[I][i]) \land isBase1(X1[I],\ p1[I],\ i)$
      $\Rightarrow\ \land\ wrts(p1[I],\ depsA(X1[I],\ Dep\_pc1,\ i))$
          $\land\ c1[I][i] = base1(X1[I],\ p1[I],\ i)$

$RInv1A\ \triangleq$
  $\forall\, I \in WDIndexA : \forall\, i \in ItA(X1[I]) :$
    $redA(I,\ i) \Rightarrow\ \land\ wrts(c1[I],\ depsA(X1[I],\ Dep\_cr1,\ i))$
                    $\land\ wrt(X2[\langle I,\ \langle i\rangle\rangle])$
                    $\land\ endB(\langle I,\ \langle i\rangle\rangle)$
```

```
RInv2A ≜
  ∀ I ∈ WDIndexA :
    r1[I] = M1!BigOpP(1, uBnd1(X1[I]),
                         LAMBDA i : redA(I, i),
                         LAMBDA i : fr1(X1[I], c1[I], i))

InvA ≜ ∧ TypeInvA
       ∧ PInvA
       ∧ CInv1A
       ∧ CInv2A
       ∧ RInv1A
       ∧ RInv2A

CorrectnessA ≜ ∀ I ∈ WDIndexA : endA(I) ⇒ r1[I] = A(X1[I])

TerminationA ≜ ◇(∀ I ∈ WDIndexA : endA(I))
```

PCR B properties

```
TypeInvB ≜
  ∧ X2  ∈ [IndexB → T2 ∪ {Undef}]
  ∧ p2  ∈ [IndexB → StB(T2)]
  ∧ c2  ∈ [IndexB → StB(D)]
  ∧ r2  ∈ [IndexB → D]
  ∧ rs2 ∈ [IndexB → [AssigB → BOOLEAN ]]

PInvB ≜
  ∀ I ∈ WDIndexB : ∀ i ∈ ItB(X2[I]) :
    wrt(p2[I][i]) ⇒ ∧ p2[I][i] = div2(X2[I])[i]
                     ∧ wrt(X2[I])

CInv1B ≜
  ∀ I ∈ WDIndexB : ∀ i ∈ ItB(X2[I]) :
    wrt(c2[I][i]) ∧ ¬isBase2(X2[I], p2[I], i)
      ⇒ ∧ wrts(p2[I], depsB(X2[I], Dep_pc2, i))
         ∧ wrt(X2[⟨I[1], I[2] ∘ ⟨i⟩⟩])
         ∧ endB(⟨I[1], I[2] ∘ ⟨i⟩⟩)
         ∧ c2[I][i] = r2[⟨I[1], I[2] ∘ ⟨i⟩⟩]

CInv2B ≜
  ∀ I ∈ WDIndexB : ∀ i ∈ ItB(X2[I]) :
    wrt(c2[I][i]) ∧ isBase2(X2[I], p2[I], i)
      ⇒ ∧ wrts(p2[I], depsB(X2[I], Dep_pc2, i))
         ∧ c2[I][i] = base2(X2[I], p2[I], i)

RInv1B ≜
  ∀ I ∈ WDIndexB : ∀ i ∈ ItB(X2[I]) :
    redB(I, i)  ⇒ ∧ wrts(c2[I], depsB(X2[I], Dep_cr2, i))
                  ∧ ∀ k ∈ ItB(X2[I]) : k < i ⇒ redB(I, k)

RInv2B ≜
  ∀ I ∈ WDIndexB : ∀ i ∈ ItB(X2[I]) :
    ¬redB(I, i) ⇒ r2[I] = M2!BigOpP(1, i − 1,
                                     LAMBDA j : redB(I, j),
                                     LAMBDA j : fr2(X2[I], c2[I], j))
```

```
InvB ≜ ∧ TypeInvB
       ∧ PInvB
       ∧ CInv1B
       ∧ CInv2B
       ∧ RInv1B
       ∧ RInv2B
```

$CorrectnessB \triangleq \forall I \in WDIndexB : endB(I) \Rightarrow r2[I] = B(X2[I])$

$TerminationB \triangleq \Diamond(\forall I \in WDIndexB : endB(I))$

Conjoint properties

```
TypeInv ≜ ∧ TypeInvA
          ∧ TypeInvB

Inv ≜ ∧ TypeInv
      ∧ InvA
      ∧ InvB

Correctness ≜ ∧ CorrectnessA
              ∧ CorrectnessB

Termination ≜ ∧ TerminationA
              ∧ TerminationB
```

Refinement

```
PCR_DC ≜ INSTANCE PCR_DC
   WITH X ← X1, p ← p1, c ← c1, r ← r1, rs ← rs1,
        T ← T, D ← D,
        id ← id, Op ← Op1,
        div ← div1, isBase ← isBase1, base ← base1, fr ← fr1,
        Dep_pc ← Dep_pc1, Dep_cr ← Dep_cr1
```

B.9 Iteration over basic function

MODULE *PCR_A_it*

PCR with iterative consumer over basic function.

```
  ----------------------------------------------------------------
  fun fp(x,p,i) = ...                // fp : T x St(Tp) x N -> Tp
  fun fc(y,x,p,i) = ...              // fc : Tc x T x St(Tp) x N -> Tc
  fun fr(x,c,i) = ...                // fr : T x St(Tc) x N -> D
  fun cnd(s,k) = ...                 // cnd : St(Tc) x N -> Bool

  dep p(i-k) -> p(i)
  dep p(i[+/-]k) -> c(i)
  dep c(i[+/-]k) -> r(i)

  lbnd A = \x. ...
  ubnd A = \x. ...
  prop A = \i. ...

  PCR A(x)                           // x \in T
    par
      p = produce fp x p
      c = iterate cnd fc (v0 x) x p
      r = reduce Op id (fr x c)      // r \in D
----------------------------------------------------------------
```

EXTENDS *AbstractAlgebra*, *Naturals*, *Sequences*, *Bags*, *SeqUtils*, *ArithUtils*, *TLC*

PCR constants and variables

CONSTANTS $I0$, $pre(\_)$,
T, Tp, Tc, D,
id, $Op(\_, \_)$, $v0(\_)$,
$lBnd(\_)$, $uBnd(\_)$, $prop(\_)$,
$fp(\_, \_, \_)$, $fc(\_, \_, \_, \_)$, $fr(\_, \_, \_)$, $gp(\_, \_)$, $cnd(\_, \_)$,
Dep_pp, Dep_pc, Dep_cr

VARIABLES in, X, p, c, r, rs, s

General definitions

$Undef \triangleq$ CHOOSE $x : x \notin$ UNION $\{T, Tp, Tc, D\}$

$last(S) \triangleq S[Len(S)]$
$wrt(v) \triangleq v \neq Undef$
$wrts(v, S) \triangleq \forall k \in S : wrt(v[k])$
$eqs(v1, v2, S) \triangleq \forall k \in S : wrt(v1[k]) \land v1[k] = v2[k]$

PCR A definitions and assumptions

$Index \triangleq Seq(Nat)$
$Assig \triangleq Nat$
$It(x) \triangleq \{i \in lBnd(x) \,..\, uBnd(x) : prop(i)\}$
$WDIndex \triangleq \{I \in Index : wrt(X[I])\}$
$St(R) \triangleq [Assig \rightarrow R \cup \{Undef\}]$
$Iter(Z) \triangleq Seq(Z)$
$red(I, i) \triangleq rs[I][i]$
$end(I) \triangleq \forall i \in It(X[I]) : red(I, i)$

```
$deps(x, d, i) \triangleq$
            $\{i - k : k \in \{k \in d[1] : i - k \geq lBnd(x) \land prop(i - k)\}\}$
   $\cup$   $\{i\}$
   $\cup$   $\{i + k : k \in \{k \in d[2] : i + k \leq uBnd(x) \land prop(i + k)\}\}$

AXIOM $H\_Type \triangleq$
   $\land\ I0 \in Index$
   $\land\ \forall x \in T : lBnd(x) \in Nat$
   $\land\ \forall x \in T : uBnd(x) \in Nat$
   $\land\ \forall i \in Nat : prop(i) \in \text{BOOLEAN}$
   $\land\ \forall x \in T : pre(x) \in \text{BOOLEAN}$
   $\land\ \forall x \in T : v0(x) \in Tc$
   $\land\ Dep\_pp \in (\text{SUBSET } (Nat \setminus \{0\})) \times (\text{SUBSET } \{\})$
   $\land\ Dep\_pc \in (\text{SUBSET } (Nat \setminus \{0\})) \times (\text{SUBSET } (Nat \setminus \{0\}))$
   $\land\ Dep\_cr \in (\text{SUBSET } (Nat \setminus \{0\})) \times (\text{SUBSET } (Nat \setminus \{0\}))$

AXIOM $H\_BFunType \triangleq$
   $\land\ \forall x \in T, i \in Assig :$
         $\land\ gp(x, i) \in Tp \cup \{Undef\}$
         $\land\ \forall vp \in St(Tp) : fp(x, vp, i) \in Tp \cup \{Undef\}$
         $\land\ \forall vc \in St(Tc) : fr(x, vc, i) \in D \cup \{Undef\}$
         $\land\ \forall vp \in St(Tp), y \in Tc : fc(y, x, vp, i) \in Tc \cup \{Undef\}$
   $\land\ \forall vs \in Iter(Tc), k \in Nat : cnd(vs, k) \in \text{BOOLEAN}$

AXIOM $H\_BFunWD \triangleq$
   $\forall x \in T : \forall i \in It(x) :$
      $\land\ gp(x, i) \in Tp$
      $\land\ \forall vp \in St(Tp) : wrts(vp, deps(x, Dep\_pp, i) \setminus \{i\}) \Rightarrow fp(x, vp, i) \in Tp$
      $\land\ \forall vc \in St(Tp) : wrts(vc, deps(x, Dep\_cr, i)) \Rightarrow fr(x, vc, i) \in D$
      $\land\ \forall vp \in St(Tp), y \in Tc :$
            $wrts(vp, deps(x, Dep\_pc, i)) \Rightarrow fc(y, x, vp, i) \in Tc$

AXIOM $H\_fpRelevance \triangleq$
   $\forall x \in T : \forall i \in It(x), vp1 \in St(Tp), vp2 \in St(Tp) :$
      $eqs(vp1, vp2, deps(x, Dep\_pp, i) \setminus \{i\}) \Rightarrow fp(x, vp1, i) = fp(x, vp2, i)$

AXIOM $H\_fcRelevance \triangleq$
   $\forall y \in Tc : \forall x \in T : \forall i \in It(x), vp1 \in St(Tp), vp2 \in St(Tp) :$
      $eqs(vp1, vp2, deps(x, Dep\_pc, i)) \Rightarrow fc(y, x, vp1, i) = fc(y, x, vp2, i)$

AXIOM $H\_frRelevance \triangleq$
   $\forall x \in T : \forall i \in It(x), vc1 \in St(Tp), vc2 \in St(Tp) :$
      $eqs(vc1, vc1, deps(x, Dep\_cr, i)) \Rightarrow fr(x, vc1, i) = fr(x, vc2, i)$

LEMMA $H\_ProdEqInv \triangleq$
   $\forall x \in T : \forall i \in It(x) :$
      $wrt(p[I0][i]) \Rightarrow fp(x, p[I0], i) = gp(x, i)$

|--------------------------------------------------------------------|

    Functional specification
$M \triangleq$ INSTANCE $AbelianMonoidBigOp$
    WITH $D \leftarrow D, Id \leftarrow id, \otimes \leftarrow Op$

AXIOM $H\_Algebra \triangleq AbelianMonoid(D, id, Op)$
```

```
RECURSIVE $iter(\_, \_, \_, \_)$
$iter(vs, x, vp, i) \triangleq$ IF $cnd(vs, Len(vs))$
                               THEN $vs$
                               ELSE $iter(vs \circ \langle fc(last(vs), x, vp, i)\rangle, x, vp, i)$
```

Alternatively, *iter* defined as a recursive function:

```
$iter[vs \in Iter(Tc), x \in T, vp \in St(Tp), i \in Assig] \triangleq$
 IF $cnd(vs, Len(vs))$
  THEN $vs$
  ELSE $iter[vs \circ \langle fc(lst(vs), x, vp, i)\rangle, x, vp, i]$
```

```
$Gp(x) \quad\quad \triangleq [i \in Assig \mapsto gp(x, i)]$
$Fr(x, vc) \triangleq [i \in Assig \mapsto fr(x, vc, i)]$
$Fc(x, vp) \triangleq [i \in Assig \mapsto last(iter(\langle v0(x)\rangle, x, vp, i))]$
```

Informal notation:

$\mathcal{A}(x) \triangleq \bigotimes_{i \in I_x} \overrightarrow{f}^{\,i}_{r}(x, i \in \mathbb{N} \mapsto last(iter(\langle v_0(x)\rangle, x, \overrightarrow{g}_{p}(x), i)))$ where $I_x \triangleq \{i \in lBnd(x)..uBnd(x) : prop(i)\}$

```
$A(x) \triangleq M!BigOpP(lBnd(x), uBnd(x), prop,$ LAMBDA $i : Fr(x, Fc(x, Gp(x)))[i])$
```

Operational specification

```
$vs \triangleq \langle X, p, c, r, rs, s\rangle$

$Init \triangleq \land\, in \in T \land pre(in)$
           $\land\, X = [I \in Index \mapsto$ IF $I = I0$ THEN $in$ ELSE $Undef]$
           $\land\, p\ = [I \in Index \mapsto [i \in Assig \mapsto Undef]]$
           $\land\, c\ = [I \in Index \mapsto [i \in Assig \mapsto Undef]]$
           $\land\, s\ = [I \in Index \mapsto [i \in Assig \mapsto Undef]]$
           $\land\, rs = [I \in Index \mapsto [i \in Assig \mapsto$ FALSE$]]$
           $\land\, r\ = [I \in Index \mapsto id]$

$P(I, i) \triangleq$
    $\land \neg wrt(p[I][i])$
    $\land\, wrts(p[I], deps(X[I], Dep\_pp, i) \setminus \{i\})$
    $\land\, p' = [p$ EXCEPT $![I][i] = fp(X[I], p[I], i)]$
    $\land$ UNCHANGED $\langle X, c, r, rs, s\rangle$

$Cstart(I, i) \triangleq$
   $\land \neg wrt(s[I][i])$
   $\land\, wrts(p[I], deps(X[I], Dep\_pc, i))$
   $\land\, s' = [s$ EXCEPT $![I][i] = \langle v0(X[I])\rangle]$
   $\land$ UNCHANGED $\langle X, p, c, r, rs\rangle$

$Cstep(I, i) \triangleq$
   $\land\ \ wrt(s[I][i])$
   $\land\ \ \neg cnd(s[I][i], Len(s[I][i]))$
   $\land\ \ s' = [s$ EXCEPT $![I][i] = @ \circ \langle fc(last(s[I][i]), X[I], p[I], i)\rangle]$
   $\land\ \ $UNCHANGED $\langle X, p, c, r, rs\rangle$

$Cend(I, i) \triangleq$
   $\land \neg wrt(c[I][i])$
   $\land\, wrt(s[I][i])$
```

```
    ∧ cnd(s[I][i], Len(s[I][i]))
    ∧ c′ = [c EXCEPT ![I][i] = last(s[I][i])]
    ∧ UNCHANGED ⟨X, p, r, rs, s⟩

R(I, i) ≜
    ∧ ¬red(I, i)
    ∧ wrts(c[I], deps(X[I], Dep_cr, i))
    ∧ r′  = [r   EXCEPT ![I]    = Op(@, fr(X[I], c[I], i))]
    ∧ rs′ = [rs EXCEPT ![I][i] = TRUE]
    ∧ UNCHANGED ⟨X, p, c, s⟩

Done ≜ ∧ ∀ I ∈ WDIndex : end(I)
       ∧ UNCHANGED ⟨in, vs⟩

Step ≜ ∧ ∃ I ∈ WDIndex :
             ∃ i ∈ It(X[I]) : ∨ P(I, i)
                              ∨ Cstart(I, i)
                              ∨ Cstep(I, i)
                              ∨ Cend(I, i)
                              ∨ R(I, i)
       ∧ UNCHANGED in

Next ≜ Step ∨ Done

Spec ≜ Init ∧ □[Next]_⟨in, vs⟩

FairSpec ≜ Spec ∧ WF_vs(Step)

├──────────────────────────────────────────────────────────────────────┤

  Properties
IndexInv ≜ WDIndex = {I0}

TypeInv ≜
   ∧ in ∈ T
   ∧ X ∈ [Index → T ∪ {Undef}] ∧ X[I0] = in
   ∧ p  ∈ [Index → St(Tp)]
   ∧ c  ∈ [Index → St(Tc)]
   ∧ s  ∈ [Index → St(Iter(Tc))]
   ∧ r  ∈ [Index → D]
   ∧ rs ∈ [Index → [Assig → BOOLEAN ]]

PInv ≜
  ∀ i ∈ It(X[I0]) :
     wrt(p[I0][i]) ⇒ ∧ wrts(p[I0], deps(X[I0], Dep_pp, i))
                     ∧ p[I0][i] = gp(X[I0], i)

CInv ≜
  ∀ i ∈ It(X[I0]) :
     wrt(c[I0][i]) ⇒ ∧ wrts(p[I0], deps(X[I0], Dep_pc, i))
                     ∧ c[I0][i] = last(iter(⟨v0(X[I0])⟩, X[I0], p[I0], i))

RInv1 ≜
  ∀ i ∈ It(X[I0]) :
     red(I0, i) ⇒ wrts(c[I0], deps(X[I0], Dep_cr, i))
```

```
$RInv2 \triangleq$
  $r[I0] = M\,!\,BigOpP(lBnd(X[I0]),\ uBnd(X[I0]),$
                              $\text{LAMBDA}\ i : prop(i) \land red(I0,\ i),$
                              $\text{LAMBDA}\ i : fr(X[I0],\ c[I0],\ i))$

$Inv \triangleq\ \land TypeInv$
         $\land IndexInv$
         $\land PInv$
         $\land CInv$
         $\land RInv1$
         $\land RInv2$

$Correctness\ \triangleq\ end(I0) \Rightarrow r[I0] = A(X[I0])$

$Termination \triangleq \Diamond end(I0)$
```

Refinement

```
$fcS(x,\ vp,\ i) \triangleq last(iter(\langle v0(x)\rangle,\ x,\ vp,\ i))$

$PCR\_A \triangleq \text{INSTANCE}\ PCR\_A$
  $\text{WITH}\ X \leftarrow X,\ p \leftarrow p,\ c \leftarrow c,\ r \leftarrow r,\ rs \leftarrow rs,$
         $T \leftarrow T,\ Tp \leftarrow Tp,\ Tc \leftarrow Tc,\ D \leftarrow D,$
         $id \leftarrow id,\ Op \leftarrow Op,$
         $lBnd \leftarrow lBnd,\ uBnd \leftarrow uBnd,\ prop \leftarrow prop,$
         $fp \leftarrow fp,\ fc \leftarrow fcS,\ fr \leftarrow fr,\ gp \leftarrow gp,$
         $Dep\_pp \leftarrow Dep\_pp,\ Dep\_pc \leftarrow Dep\_pc,\ Dep\_cr \leftarrow Dep\_cr$
```

B.10 Iteration over PCR

```
──────────────────────── MODULE PCR_A_it_B ────────────────────────
```

PCR with iterative consumer over a basic *PCR*.

```
  -----------------------------------------------------------------
  // PCR A

  fun fp1(x1,p1,i) = ...                 // fp1 : T x St(Tp1) x N -> Tp1
  fun fr1(x1,c1,i) = ...                 // fr1 : T x St(D2) x N -> D1
  fun cnd(s,k)     = ...                 // cnd : St(D2) x N -> Bool

  dep p1(i-k) -> p1(i)
  dep p1(i[+/-]k) -> c1(i)
  dep c1(i[+/-]k) -> r1(i)

  lbnd A = \x1. ...
  ubnd A = \x1. ...
  prop A = \i. ...

  PCR A(x1)                              // x1 \in T
    par
      p1 = produce fp1 x p1
      c1 = iterate cnd B (v0 x) x p1
      r1 = reduce Op1 id1 (fr1 x1 c1)    // r1 \in D1

  // PCR B
                                         // T2 = D2 x T x St(Tp1) x N
  fun fp2(x2,p2,j) = ...                 // fp2 : T2 x St(Tp2) x N -> Tp2
  fun fc2(x2,p2,j) = ...                 // fc2 : T2 x St(Tp2) x N -> Tc2
  fun fr2(x2,c2,j) = ...                 // fr2 : T2 x St(Tc2) x N -> D2

  dep p2(i-k) -> p2(i)
  dep p2(i[+/-]k) -> c2(i)
  dep c2(i[+/-]k) -> r2(i)

  lbnd B = \x2. ...
  ubnd B = \x2. ...
  prop B = \j. ...

  PCR B(x2)                              // x2 \in T2
    par
      p2 = produce fp2 x2 p2
      c2 = consume fc2 x2 p2
      r2 = reduce Op2 id2 (fr2 x2 c2)    // r2 \in D2
-----------------------------------------------------------------
```

EXTENDS *AbstractAlgebra*, *Naturals*, *Sequences*, *Bags*, *SeqUtils*, *ArithUtils*, *SequencesExt*, *TLC*

PCR A constants and variables

```
CONSTANTS I0, pre(_),
          T, Tp1, D1,
          id1, Op1(_, _), v0(_),
          lBnd1(_), uBnd1(_), prop1(_),
          fp1(_, _, _), fr1(_, _, _), gp1(_, _), cnd(_, _),
          Dep_pp1, Dep_pc1, Dep_cr1

VARIABLES in, X1, p1, c1, r1, rs1, s
```

PCR B constants and variables

```
CONSTANTS Tp2, Tc2, D2,
          id2, Op2(_, _),
          lBnd2(_), uBnd2(_), prop2(_),
          fp2(_, _, _), fc2(_, _, _), fr2(_, _, _), gp2(_, _),
          Dep_pp2, Dep_pc2, Dep_cr2
```

```
VARIABLES X2, p2, c2, r2, rs2

├──────────────────────────────────────────────────────────────────────┤

  General definitions
Undef ≜ CHOOSE x : x ∉ UNION {T, Tp1, Tp2, Tc2, D1, D2}

last(S)          ≜ S[Len(S)]
wrt(v)           ≜ v ≠ Undef
wrts(v, S)       ≜ ∀ k ∈ S : wrt(v[k])
eqs(v1, v2, S)   ≜ ∀ k ∈ S : wrt(v1[k]) ∧ v1[k] = v2[k]

├──────────────────────────────────────────────────────────────────────┤

  PCR A definitions and assumptions
IndexA      ≜ Seq(Nat)
AssigA      ≜ Nat
ItA(x)      ≜ {i ∈ lBnd1(x) . . uBnd1(x) : prop1(i)}
WDIndexA    ≜ {I ∈ IndexA : wrt(X1[I])}
StA(R)      ≜ [AssigA → R ∪ {Undef}]
Iter(Z)     ≜ Seq(Z)
redA(I, i)  ≜ rs1[I][i]
endA(I)     ≜ ∀ i ∈ ItA(X1[I]) : redA(I, i)

depsA(x, d, i) ≜
          {i − k : k ∈ {k ∈ d[1] : i − k ≥ lBnd1(x) ∧ prop1(i − k)}}
  ∪       {i}
  ∪       {i + k : k ∈ {k ∈ d[2] : i + k ≤ uBnd1(x) ∧ prop1(i + k)}}

AXIOM H_TypeA ≜
  ∧ I0   ∈ IndexA
  ∧ ∀ x  ∈ T   : lBnd1(x) ∈ Nat
  ∧ ∀ x  ∈ T   : uBnd1(x) ∈ Nat
  ∧ ∀ i  ∈ Nat : prop1(i)  ∈ BOOLEAN
  ∧ ∀ x  ∈ T   : pre(x)    ∈ BOOLEAN
  ∧ ∀ x  ∈ T   : v0(x)     ∈ D2
  ∧ Dep_pp1 ∈ (SUBSET (Nat \ {0})) × (SUBSET {})
  ∧ Dep_pc1 ∈ (SUBSET (Nat \ {0})) × (SUBSET (Nat \ {0}))
  ∧ Dep_cr1 ∈ (SUBSET (Nat \ {0})) × (SUBSET (Nat \ {0}))

AXIOM H_BFunTypeA ≜
  ∧ ∀ x ∈ T, i    ∈ AssigA :
        ∧ gp1(x, i) ∈ Tp1 ∪ {Undef}
        ∧ ∀ vp ∈ StA(Tp1) : fp1(x, vp, i) ∈ Tp1 ∪ {Undef}
        ∧ ∀ vc ∈ StA(D2)  : fr1(x, vc, i) ∈ D1  ∪ {Undef}
  ∧ ∀ vc ∈ Seq(Tp1), k ∈ Nat : cnd(vc, k) ∈ BOOLEAN

AXIOM H_BFunWDA ≜
  ∀ x ∈ T : ∀ i ∈ ItA(x) :
     ∧ gp1(x, i) ∈ Tp1
     ∧ ∀ vp ∈ StA(Tp1) : wrts(vp, depsA(x, Dep_pp1, i) \ {i}) ⇒ fp1(x, vp, i) ∈ Tp1
     ∧ ∀ vc ∈ StA(Tp1) : wrts(vc, depsA(x, Dep_cr1, i))       ⇒ fr1(x, vc, i) ∈ D1

AXIOM H_ProdEqA ≜
  ∀ x ∈ T : ∀ i ∈ ItA(x), vp ∈ StA(Tp1) :
    wrts(vp, depsA(x, Dep_pp1, i) \ {i}) ⇒ fp1(x, vp, i) = gp1(x, i)

AXIOM H_fpRelevanceA ≜
  ∀ x ∈ T : ∀ i ∈ ItA(x), vp1 ∈ StA(Tp1), vp2 ∈ StA(Tp1) :
    eqs(vp1, vp2, depsA(x, Dep_pp1, i) \ {i}) ⇒ fp1(x, vp1, i) = fp1(x, vp2, i)

AXIOM H_frRelevanceA ≜
  ∀ x ∈ T : ∀ i ∈ ItA(x), vc1 ∈ StA(Tp1), vc2 ∈ StA(Tp1) :
    eqs(vc1, vc1, depsA(x, Dep_cr1, i)) ⇒ fr1(x, vc1, i) = fr1(x, vc2, i)

LEMMA H_ProdEqInvA ≜
  ∀ x ∈ T : ∀ i ∈ ItA(x) :
    wrt(p1[I0][i]) ⇒ fp1(x, p1[I0], i) = gp1(x, i)

  PCR B definitions and assumptions
IndexB      ≜ Seq(Nat)
AssigB      ≜ Nat
ItB(x)      ≜ {i ∈ lBnd2(x) .. uBnd2(x) : prop2(i)}
WDIndexB ≜ {I ∈ IndexB : wrt(X2[I])}
StB(R)      ≜ [AssigB → R ∪ {Undef}]
redB(I, i)  ≜ rs2[I][i]
endB(I)     ≜ ∀ i ∈ ItB(X2[I]) : redB(I, i)

depsB(x, d, i) ≜
              {i − k : k ∈ {k ∈ d[1] : i − k ≥ lBnd2(x) ∧ prop2(i − k)}}
  ∪           {i}
  ∪           {i + k : k ∈ {k ∈ d[2] : i + k ≤ uBnd2(x) ∧ prop2(i + k)}}

T2 ≜ D2 × T × StA(Tp1) × AssigA

AXIOM H_TypeB ≜
  ∧ ∀ x ∈ T2  : lBnd2(x)  ∈ Nat
  ∧ ∀ x ∈ T2  : uBnd2(x) ∈ Nat
  ∧ ∀ i  ∈ Nat : prop2(i)   ∈ BOOLEAN
  ∧ Dep_pp2 ∈ (SUBSET (Nat \ {0})) × (SUBSET {})
  ∧ Dep_pc2 ∈ (SUBSET (Nat \ {0})) × (SUBSET (Nat \ {0}))
  ∧ Dep_cr2 ∈ (SUBSET (Nat \ {0})) × (SUBSET (Nat \ {0}))

AXIOM H_BFunTypeB ≜
  ∀ x ∈ T2, i ∈ AssigB :
    ∧ gp2(x, i) ∈ Tp2 ∪ {Undef}
    ∧ ∀ vp ∈ StB(Tp2) : fp2(x, vp, i) ∈ Tp2 ∪ {Undef}
    ∧ ∀ vp ∈ StB(Tp2) : fc2(x, vp, i) ∈ Tc2 ∪ {Undef}
    ∧ ∀ vc ∈ StB(Tc2) : fr2(x, vc, i) ∈ T2  ∪ {Undef}

AXIOM H_BFunWDB ≜
  ∀ x ∈ T2 : ∀ i ∈ ItB(x) :
    ∧ gp2(x, i) ∈ Tp2
    ∧ ∀ vp ∈ StB(Tp2) : wrts(vp, depsB(x, Dep_pp2, i) \ {i}) ⇒ fp2(x, vp, i) ∈ Tp2
    ∧ ∀ vp ∈ StB(Tp2) : wrts(vp, depsB(x, Dep_pc2, i))       ⇒ fc2(x, vp, i) ∈ Tc2
    ∧ ∀ vc ∈ StB(Tc2) : wrts(vc, depsB(x, Dep_cr2, i))       ⇒ fr2(x, vc, i) ∈ T2

AXIOM H_fpRelevanceB ≜
  ∀ x ∈ T2 : ∀ i ∈ ItB(x), vp1 ∈ StB(Tp2), vp2 ∈ StB(Tp2) :
    eqs(vp1, vp2, depsB(x, Dep_pp2, i) \ {i}) ⇒ fp2(x, vp1, i) = fp2(x, vp2, i)
```

```
$\text{AXIOM } H\_fcRelevanceB \triangleq$
  $\forall x \in T2 : \forall i \in ItB(x),\ vp1 \in StB(Tp2),\ vp2 \in StB(Tp2) :$
    $eqs(vp1,\ vp2,\ depsB(x,\ Dep\_pc2,\ i)) \Rightarrow fc2(x,\ vp1,\ i) = fc2(x,\ vp2,\ i)$

$\text{AXIOM } H\_frRelevanceB \triangleq$
  $\forall x \in T2 : \forall i \in ItB(x),\ vc1 \in StB(Tc2),\ vc2 \in StB(Tc2) :$
    $eqs(vc1,\ vc1,\ depsB(x,\ Dep\_cr2,\ i)) \Rightarrow fr2(x,\ vc1,\ i) = fr2(x,\ vc2,\ i)$

$\text{LEMMA } H\_ProdEqInvB \triangleq$
  $\forall I \in WDIndexB : \forall i \in ItB(X2[I]) :$
    $wrt(p1[I][i]) \Rightarrow fp2(X2[I],\ p2[I],\ i) = gp2(X2[I],\ i)$
```

Functional specification

```
$M2 \triangleq \text{INSTANCE } AbelianMonoidBigOp$
  $\text{WITH } D \leftarrow D2,\ Id \leftarrow id2,\ \otimes \leftarrow Op2$

$\text{AXIOM } H\_AlgebraB \triangleq AbelianMonoid(Tp2,\ id2,\ Op2)$

$Gp2(x) \quad\;\; \triangleq [i \in AssigB \mapsto gp2(x,\ i)]$
$Fc2(x,\ vc) \triangleq [i \in AssigB \mapsto fc2(x,\ vc,\ i)]$
$Fr2(x,\ vc) \triangleq [i \in AssigB \mapsto fr2(x,\ vc,\ i)]$
```

Informal notation:

$\mathcal{B}(x_2) \triangleq \bigoplus_{j \in J_{x_2}} \vec{f}^{\,j}_{r2}(x_2,\ \vec{f}_{c2}(x_2,\ \vec{g}_{p2}(x_2)))$ where $J_{x_2} \triangleq \{j \in lBnd2(x_2)..uBnd2(x_2) : prop2(j)\}$

```
$B(x2) \triangleq M2!BigOpP(lBnd2(x2),\ uBnd2(x2),\ prop2,$
                              $\text{LAMBDA } j : Fr2(x2,\ Fc2(x2,\ Gp2(x2)))[j])$

$M1 \triangleq \text{INSTANCE } AbelianMonoidBigOp$
  $\text{WITH } D \leftarrow D1,\ Id \leftarrow id1,\ \otimes \leftarrow Op1$

$\text{AXIOM } H\_AlgebraA \triangleq AbelianMonoid(D1,\ id1,\ Op1)$

$\text{RECURSIVE } iter(\_,\ \_,\ \_,\ \_)$
$iter(vs,\ x,\ vp,\ i) \triangleq \text{IF } cnd(vs,\ Len(vs))$
                          $\text{THEN } vs$
                          $\text{ELSE } iter(vs \circ \langle B(\langle last(vs),\ x,\ vp,\ i\rangle)\rangle,\ x,\ vp,\ i)$
```

Alternatively, *iter* defined as a recursive function:

```
$iter[vs \in Iter(D2),\ x \in T,\ vp \in StA(Tp1),\ i \in AssigA] \triangleq$
  $\text{IF } cnd(vs,\ Len(vs))$
   $\text{THEN } vs$
   $\text{ELSE } iter[vs \circ \langle B(\langle lst(vs),\ x,\ vp,\ i\rangle)\rangle,\ x,\ vp,\ i]$
```

```
$Gp1(x) \quad\;\;\; \triangleq [i \in AssigA \mapsto gp1(x,\ i)]$
$Fc1(x1,\ vp) \triangleq [i \in AssigA \mapsto last(iter(\langle v0(x1)\rangle,\ x1,\ vp,\ i))]$
$Fr1(x,\ vc) \;\;\; \triangleq [i \in AssigA \mapsto fr1(x,\ vc,\ i)]$
```

Informal notation:

$\mathcal{A}(x_1) \triangleq \bigotimes_{i \in I_{x_1}} \vec{f}^{\,i}_{r1}(x_1,\ i \in \mathbb{N} \mapsto last(iter(\langle v_0(x_1)\rangle, x_1,\ \vec{g}_{p1}(x_1), i)))$

where $I_{x_1} \triangleq \{i \in lBnd1(x_1)..uBnd1(x_1) : prop1(i)\}$

$A(x1) \triangleq M1!BigOpP(lBnd1(x1), uBnd1(x1), prop1,$
$\qquad\qquad \text{LAMBDA } i : Fr1(x1, Fc1(x1, Gp1(x1)))[i])$

Operational specification

$vs1 \triangleq \langle X1, p1, c1, r1, rs1, s, X2\rangle$
$vs2 \triangleq \langle p2, c2, r2, rs2\rangle$

$$\begin{aligned}
InitA \triangleq\ & \land\ in \in T \land pre(in)\\
& \land\ X1 \ = [I \in IndexA \mapsto \text{IF } I = I0 \text{ THEN } in \text{ ELSE } Undef]\\
& \land\ p1 \ = [I \in IndexA \mapsto [i \in AssigA \mapsto Undef]]\\
& \land\ c1 \ = [I \in IndexA \mapsto [i \in AssigA \mapsto Undef]]\\
& \land\ s \ = [I \in IndexA \mapsto [i \in AssigA \mapsto Undef]]\\
& \land\ rs1 = [I \in IndexA \mapsto [i \in AssigA \mapsto \text{FALSE}]]\\
& \land\ r1 \ = [I \in IndexA \mapsto id1]
\end{aligned}$$

$$\begin{aligned}
InitB \triangleq\ & \land\ X2 \ = [I \in IndexB \mapsto Undef]\\
& \land\ p2 \ = [I \in IndexB \mapsto [i \in AssigB \mapsto Undef]]\\
& \land\ c2 \ = [I \in IndexB \mapsto [i \in AssigB \mapsto Undef]]\\
& \land\ rs2 = [I \in IndexB \mapsto [i \in AssigB \mapsto \text{FALSE}]]\\
& \land\ r2 \ = [I \in IndexB \mapsto id2]
\end{aligned}$$

$Init \triangleq InitA \land InitB$

$P1(I, i) \triangleq$
$\quad \land \neg wrt(p1[I][i])$
$\quad \land wrts(p1[I], depsA(X1[I], Dep\_pp1, i) \setminus \{i\})$
$\quad \land p1' = [p1 \text{ EXCEPT } ![I][i] = fp1(X1[I], p1[I], i)]$
$\quad \land \text{UNCHANGED } \langle X1, c1, r1, rs1, s, X2\rangle$

$C1start(I, i) \triangleq$
$\quad \land wrts(p1[I], depsA(X1[I], Dep\_pc1, i))$
$\quad \land \neg wrt(s[I][i])$
$\quad \land s' = [s \text{ EXCEPT } ![I][i] = \langle v0(X1[I])\rangle]$
$\quad \land \text{UNCHANGED } \langle X1, p1, c1, r1, rs1, X2\rangle$

$C1stepIni(I, i) \triangleq$
$\quad \land wrt(s[I][i])$
$\quad \land \neg cnd(s[I][i], Len(s[I][i]))$
$\quad \land \neg wrt(X2[I \circ \langle Len(s[I][i])\rangle])$
$\quad \land X2' = [X2 \text{ EXCEPT } ![I \circ \langle Len(s[I][i])\rangle] = \langle last(s[I][i]), X1[I], p1[I], i\rangle]$
$\quad \land \text{UNCHANGED } \langle X1, p1, c1, r1, rs1, s\rangle$

$C1stepEnd(I, i) \triangleq$
$\quad \land wrt(s[I][i])$
$\quad \land wrt(X2[I \circ \langle Len(s[I][i])\rangle])$
$\quad \land endB(I \circ \langle Len(s[I][i])\rangle)$
$\quad \land s' = [s \text{ EXCEPT } ![I][i] = @ \circ \langle r2[I \circ \langle Len(s[I][i])\rangle]\rangle]$
$\quad \land \text{UNCHANGED } \langle X1, p1, c1, r1, rs1, X2\rangle$

$C1end(I, i) \triangleq$
$\quad \land \quad \neg wrt(c1[I][i])$
$\quad \land \quad wrt(s[I][i])$
$\quad \land \quad cnd(s[I][i], Len(s[I][i]))$

```
    ∧    c1′ = [c1 EXCEPT ![I][i] = last(s[I][i])]
    ∧    UNCHANGED ⟨X1, p1, r1, rs1, s, X2⟩

R1(I, i) ≜
    ∧ ¬redA(I, i)
    ∧ wrts(c1[I], depsA(X1[I], Dep_cr1, i))
    ∧ r1′  = [r1  EXCEPT ![I]     = Op1(@, fr1(X1[I], c1[I], i))]
    ∧ rs1′ = [rs1 EXCEPT ![I][i]  = TRUE]
    ∧ UNCHANGED ⟨X1, p1, c1, s, X2⟩

P2(I, i) ≜
    ∧ ¬wrt(p2[I][i])
    ∧ wrts(p2[I], depsB(X2[I], Dep_pp2, i) \ {i})
    ∧ p2′ = [p2 EXCEPT ![I][i] = fp2(X2[I], p2[I], i)]
    ∧ UNCHANGED ⟨c2, r2, rs2⟩

C2(I, i) ≜
    ∧ ¬wrt(c2[I][i])
    ∧ wrts(p2[I], depsB(X2[I], Dep_pc2, i))
    ∧ c2′ = [c2 EXCEPT ![I][i] = fc2(X2[I], p2[I], i)]
    ∧ UNCHANGED ⟨p2, r2, rs2⟩

R2(I, i) ≜
    ∧ ¬redB(I, i)
    ∧ wrts(c2[I], depsB(X2[I], Dep_cr2, i))
    ∧ r2′  = [r2  EXCEPT ![I]     = Op2(@, fr2(X2[I], c2[I], i))]
    ∧ rs2′ = [rs2 EXCEPT ![I][i]  = TRUE]
    ∧ UNCHANGED ⟨p2, c2⟩

Done ≜ ∧ ∀ I ∈ WDIndexA : endA(I)
       ∧ ∀ I ∈ WDIndexB : endB(I)
       ∧ UNCHANGED ⟨in, vs1, vs2⟩

StepA ≜ ∧ ∃ I ∈ WDIndexA :
              ∃ i ∈ ItA(X1[I]) : ∨ P1(I, i)
                                 ∨ C1start(I, i)
                                 ∨ C1stepIni(I, i)
                                 ∨ C1stepEnd(I, i)
                                 ∨ C1end(I, i)
                                 ∨ R1(I, i)
        ∧ UNCHANGED ⟨in, vs2⟩

StepB ≜ ∧ ∃ I ∈ WDIndexB :
              ∃ i ∈ ItB(X2[I]) : ∨ P2(I, i)
                                 ∨ C2(I, i)
                                 ∨ R2(I, i)
        ∧ UNCHANGED ⟨in, vs1⟩

Next ≜ StepA ∨ StepB ∨ Done
```

$Spec \triangleq Init \land \Box[Next]_{\langle in,\, vs1,\, vs2\rangle}$

$FairSpec \triangleq Spec \land \mathrm{WF}_{vs1}(StepA) \land \mathrm{WF}_{vs2}(StepB)$

PCR A properties

$IndexInvA \triangleq WDIndexA = \{I0\}$

$TypeInvA \triangleq$
$\land\, in \in T$
$\land\, X1 \in [IndexA \rightarrow T \cup \{Undef\}] \land X1[I0] = in$
$\land\, p1 \in [IndexA \rightarrow StA(Tp1)]$
$\land\, c1 \in [IndexA \rightarrow StA(D2)]$
$\land\, s \in [IndexA \rightarrow StA(Iter(D2))]$
$\land\, r1 \in [IndexA \rightarrow D1]$
$\land\, rs1 \in [IndexA \rightarrow [AssigA \rightarrow \text{BOOLEAN}]]$

```
PInvA ≜
  ∀ i ∈ ItA(X1[I0]) :
    wrt(p1[I0][i]) ⇒ ∧ wrts(p1[I0], depsA(X1[I0], Dep_pp1, i))
                     ∧ p1[I0][i] = gp1(X1[I0], i)

CInv1A ≜
  ∀ i ∈ ItA(X1[I0]) :
    wrt(c1[I0][i]) ⇒ ∧ wrts(p1[I0], depsA(X1[I0], Dep_pc1, i))
                     ∧ wrt(s[I0][i])
                     ∧ c1[I0][i] = last(s[I0][i])

CInv2A ≜
  ∀ i ∈ ItA(X1[I0]) :
    wrt(c1[I0][i]) ⇒ ∀ k ∈ 1 .. (Len(s[I0][i]) − 1) :
                       ∧ wrt(X2[I0 ∘ ⟨k⟩])
                       ∧ endB(I0 ∘ ⟨k⟩)

RInv1A ≜
  ∀ i ∈ ItA(X1[I0]) :
    redA(I0, i) ⇒ wrts(c1[I0], depsA(X1[I0], Dep_cr1, i))

RInv2A ≜
  r1[I0] = M1!BigOpP(lBnd1(X1[I0]), uBnd1(X1[I0]),
                     LAMBDA i : prop1(i) ∧ redA(I0, i),
                     LAMBDA i : fr1(X1[I0], c1[I0], i))

InvA ≜ ∧ IndexInvA
       ∧ TypeInvA
       ∧ PInvA
       ∧ CInv1A
       ∧ CInv2A
       ∧ RInv1A
       ∧ RInv2A
```

$CorrectnessA \triangleq endA(I0) \Rightarrow r1[I0] = A(X1[I0])$

$TerminationA \triangleq \Diamond endA(I0)$

PCR B properties

$IndexInvB \triangleq WDIndexB \subseteq \{I0 \circ \langle i \rangle : i \in Nat\}$

$TypeInvB \triangleq$
$\land\, X2 \in [IndexB \rightarrow T2 \cup \{Undef\}]$

```
  ∧ p2   ∈ [IndexB → StB(Tp2)]
  ∧ c2   ∈ [IndexB → StB(Tc2)]
  ∧ r2   ∈ [IndexB → D2]
  ∧ rs2  ∈ [IndexB → [AssigB → BOOLEAN ]]

PInvB ≜
  ∀ I ∈ WDIndexB : ∀ i ∈ ItB(X2[I]) :
    wrt(p2[I][i]) ⇒ ∧ wrts(p2[I], depsB(X2[I], Dep_pp2, i))
                    ∧ p2[I][i] = gp2(X2[I], i)

CInvB ≜
  ∀ I ∈ WDIndexB : ∀ i ∈ ItB(X2[I]) :
    wrt(c2[I][i]) ⇒ ∧ wrts(p2[I], depsB(X2[I], Dep_pc2, i))
                    ∧ c2[I][i] = fc2(X2[I], p2[I], i)


RInv1B ≜
  ∀ I ∈ WDIndexB : ∀ i ∈ ItB(X2[I]) :
    redB(I, i) ⇒ wrts(c2[I], depsB(X2[I], Dep_cr2, i))

RInv2B ≜
  ∀ I ∈ WDIndexB :
    r2[I] = M2!BigOpP(lBnd2(X2[I]), uBnd2(X2[I]),
                      LAMBDA j : prop2(j) ∧ redB(I, j),
                      LAMBDA j : fr2(X2[I], c2[I], j))

InvB ≜ ∧ TypeInvB
       ∧ IndexInvB
       ∧ PInvB
       ∧ CInvB
       ∧ RInv1B
       ∧ RInv2B

CorrectnessB  ≜ ∀ I ∈ WDIndexB : endB(I) ⇒ r2[I] = B(X2[I])

TerminationB ≜ ◇(∀ I ∈ WDIndexB : endB(I))

 Conjoint properties
TypeInv ≜ ∧ TypeInvA
          ∧ TypeInvB

Inv ≜ ∧ TypeInv
      ∧ InvA
      ∧ InvB

Correctness  ≜ ∧ CorrectnessA
               ∧ CorrectnessB

Termination ≜ ∧ TerminationA
              ∧ TerminationB

 Refinement
fcS(y, x, vp, i) ≜ B(⟨y, x, vp, i⟩)

PCR_A_it ≜ INSTANCE PCR_A_it
  WITH X ← X1, p ← p1, c ← c1, r ← r1, rs ← rs1, s ← s,
        $T \leftarrow T,\ Tp \leftarrow Tp1,\ Tc \leftarrow D2,\ D \leftarrow D1,$
        $id \leftarrow id1,\ Op \leftarrow Op1,\ v0 \leftarrow v0,$
        $lBnd \leftarrow lBnd1,\ uBnd \leftarrow uBnd1,\ prop \leftarrow prop1,$
        $fp \leftarrow fp1,\ fc \leftarrow fcS,\ fr \leftarrow fr1,\ gp \leftarrow gp1,$
        $Dep\_pp \leftarrow Dep\_pp1,\ Dep\_pc \leftarrow Dep\_pc1,\ Dep\_cr \leftarrow Dep\_cr1$
```

Appendix C

Formal proofs on abstract PCR models

C.1 Basic PCR: Correctness

```
──────────────────── MODULE $PCR\_A\_Thms$ ────────────────────
THEOREM $Thm\_Correctness \triangleq Spec \Rightarrow \Box Correctness$
⟨1⟩ DEFINE $x$    $\triangleq X[I0]$
           $y$    $\triangleq r[I0]$
           $m$    $\triangleq lBnd(x)$
           $n$    $\triangleq uBnd(x)$
           $Q(j)$ $\triangleq prop(j)$
           $f(i)$ $\triangleq Fr(x, Fc(x, Gp(x)))[i]$
⟨1⟩1. $Init \Rightarrow Correctness$
  ⟨2⟩0. SUFFICES ASSUME $Init$,
                        $end(I0)$
                 PROVE  $y = A(x)$
    BY  DEF $Correctness$
  ⟨2⟩1. $\land\ I0 \in Seq(Nat)$
        $\land\ x \in T$
        $\land\ It(x) \subseteq Nat$
     BY ⟨2⟩0, $H\_Type$ DEF $Init$, $It$
  ⟨2⟩A.CASE $It(x) = \{\}$
    ⟨3⟩1. $m \in Nat \land n \in Nat$   BY ⟨2⟩1, $H\_Type$
    ⟨3⟩A.CASE $m > n$
      ⟨4⟩1. $A(x) = id$
        BY ⟨3⟩1, ⟨3⟩A, $H\_AMon$, $H\_MeqMT$,
           $MT!EmptyIntvAssumpP$ DEF $A$
      ⟨4⟩2. $y = id$
        BY ⟨2⟩0, ⟨2⟩1  DEF $Init$
      ⟨4⟩ QED
        BY ⟨4⟩1, ⟨4⟩2
    ⟨3⟩B.CASE $\forall i \in m \,..\, n : \neg Q(i)$
      ⟨4⟩1. $\forall i \in m \,..\, n : Q(i) \in$ BOOLEAN
        BY ⟨3⟩1, $m \,..\, n \subseteq Nat$, $H\_Type$
      ⟨4⟩2. $\forall i \in \{j \in m \,..\, n : Q(j)\} : f(i) \in D$
        BY ⟨2⟩A, ⟨2⟩1, ⟨3⟩1  DEF $f$, $It$
      ⟨4⟩ HIDE  DEF $m$, $n$, $Q$, $f$
      ⟨4⟩3. $M!BigOpP(m, n, Q, f) = id$
        BY ⟨3⟩1, ⟨3⟩B, ⟨4⟩1, ⟨4⟩2, $H\_AMon$, $H\_MeqMT$,
           $MT!FalsePredicate$, $Isa$ DEF $A$
      ⟨4⟩4. $A(x) = id$
        BY ⟨4⟩3  DEF $A$, $f$
      ⟨4⟩5. $y = id$
        BY ⟨2⟩0, ⟨2⟩1  DEF $Init$
      ⟨4⟩ QED
        BY ⟨4⟩4, ⟨4⟩5
    ⟨3⟩ QED
      BY ⟨2⟩A, ⟨3⟩A, ⟨3⟩B  DEF $It$
  ⟨2⟩B.CASE $It(x) \neq \{\}$
    ⟨3⟩1. $\forall i \in It(x) : red(I0, i)$       BY ⟨2⟩0  DEF $end$
    ⟨3⟩2. $\forall i \in Nat : \neg red(I0, i)$     BY ⟨2⟩0, ⟨2⟩1  DEF $Init$
    ⟨3⟩3. FALSE                                      BY ⟨2⟩1, ⟨2⟩B, ⟨3⟩1, ⟨3⟩2
    ⟨3⟩ QED                                          BY ⟨3⟩3
  ⟨2⟩ QED
    BY ⟨2⟩A, ⟨2⟩B
```

```
$\langle 1\rangle 2.\ \land\ Inv$
      $\land\ Correctness$
      $\land\ [Next]_{\langle in,\, vs\rangle}$
      $\Rightarrow\ Correctness'$
  $\langle 2\rangle 0.$ SUFFICES ASSUME $IndexInv,\ TypeInv,\ PInv,$
                              $CInv,\ RInv1,\ RInv2,$
                              $Correctness,$
                              $[Next]_{\langle in,\, vs\rangle}$
                       PROVE  $Correctness'$
    BY DEF $Inv$
  $\langle 2\rangle$A.CASE $Step$
    $\langle 3\rangle 0.$ SUFFICES ASSUME $\exists\, i \in It(x) :\ \lor\ P(I0,\ i)$
                                          $\lor\ C(I0,\ i)$
                                          $\lor\ R(I0,\ i)$
                         PROVE $Correctness'$
      BY $\langle 2\rangle 0,\ \langle 2\rangle A$ DEF $Step,\ IndexInv$
    $\langle 3\rangle 1.$ PICK $i \in It(x) :\ \lor\ P(I0,\ i)$
                         $\lor\ C(I0,\ i)$
                         $\lor\ R(I0,\ i)$
      BY $\langle 3\rangle 0$
    $\langle 3\rangle 2.\ x \in T$
      BY $\langle 2\rangle 0$ DEF $IndexInv,\ TypeInv,\ WDIndex,\ wrt$
    $\langle 3\rangle 3.\ \land\ I0 \in Seq(Nat)$
         $\land\ I0 \in WDIndex$
         $\land\ i \in Nat$
         $\land\ i \in \{k \in m\ ..\ n : Q(k)\}$
         $\land\ It(x) \subseteq Nat$
      BY $\langle 2\rangle 0,\ \langle 3\rangle 2,\ H\_Type$ DEF $IndexInv,\ WDIndex,\ It$
    $\langle 3\rangle 4.\ m \in Nat \land n \in Nat$
      BY $\langle 2\rangle 0,\ \langle 3\rangle 2,\ H\_Type$ DEF $TypeInv,\ m,\ n$
    $\langle 3\rangle 5.\ \forall\, j \in It(x) :$
           $\land\ red(I0,\ j)\ \ \ \ \ \ \Rightarrow wrt(c[I0][j])$
           $\land\ wrt(c[I0][j])\ \ \Rightarrow wrt(p[I0][j])$
      BY $\langle 2\rangle 0$ DEF $CInv,\ RInv1,\ wrts,\ deps$

    $\langle 3\rangle$A.CASE $P(I0,\ i)$
      $\langle 4\rangle 0.$ SUFFICES ASSUME $P(I0,\ i),$
                                $end(I0)'$
                         PROVE  FALSE
        BY $\langle 2\rangle 0,\ \langle 3\rangle A$ DEF $P,\ Correctness$
      $\langle 4\rangle 1.\ \land\ \neg wrt(p[I0][i])$
           $\land\ \langle X,\ rs\rangle' = \langle X,\ rs\rangle$          BY $\langle 4\rangle 0$ DEF $P$
      $\langle 4\rangle 2.\ \neg wrt(p[I0][i]) \Rightarrow \neg red(I0,\ i)$    BY $\langle 3\rangle 5$
      $\langle 4\rangle 3.\ \neg red(I0,\ i)$                       BY $\langle 4\rangle 1,\ \langle 4\rangle 2$
      $\langle 4\rangle 4.\ \neg red(I0,\ i)'$                      BY $\langle 4\rangle 1,\ \langle 4\rangle 3$
      $\langle 4\rangle 5.\ \exists j \in It(x)' : \neg red(I0,\ j)'$    BY $\langle 4\rangle 1,\ \langle 4\rangle 4$ DEF $It$
      $\langle 4\rangle 6.$ FALSE                            BY $\langle 4\rangle 5,\ end(I0)'$ DEF $end$
      $\langle 4\rangle$ QED
        BY $\langle 4\rangle 6$

    $\langle 3\rangle$B.CASE $C(I0,\ i)$
      $\langle 4\rangle 0.$ SUFFICES ASSUME $C(I0,\ i),$
```

```
                              $end(I0)'$
                       PROVE  FALSE
      BY $\langle 2\rangle 0$, $\langle 3\rangle B$  DEF $C$, $Correctness$
    $\langle 4\rangle 1.$ $\land \neg wrt(c[I0][i])$
         $\land \langle X, rs\rangle' = \langle X, rs\rangle$           BY $\langle 4\rangle 0$  DEF $C$
    $\langle 4\rangle 2.$ $\neg wrt(c[I0][i]) \Rightarrow \neg red(I0, i)$  BY $\langle 3\rangle 5$
    $\langle 4\rangle 3.$ $\neg red(I0, i)$                        BY $\langle 4\rangle 1$, $\langle 4\rangle 2$
    $\langle 4\rangle 4.$ $\neg red(I0, i)'$                       BY $\langle 4\rangle 1$, $\langle 4\rangle 3$
    $\langle 4\rangle 5.$ $\exists j \in It(x)' : \neg red(I0, j)'$  BY $\langle 4\rangle 1$, $\langle 4\rangle 4$  DEF $It$
    $\langle 4\rangle 6.$ FALSE                              BY $\langle 4\rangle 5$, $end(I0)'$ DEF $end$
    $\langle 4\rangle$ QED
      BY $\langle 4\rangle 6$

  $\langle 3\rangle C.$CASE $R(I0, i)$
    $\langle 4\rangle 0.$ SUFFICES ASSUME $R(I0, i)$,
                              $end(I0)'$
                       PROVE  $y' = A(x)$
      BY $\langle 2\rangle 0$, $\langle 3\rangle C$  DEF $R$, $Correctness$, $A$, $M!BigOpP$, $M!BigOp$, $M!bigOp$
    $\langle 4\rangle$ DEFINE $g(j) \triangleq fr(x, c[I0], j)$
    $\langle 4\rangle 1.$ $\land \neg red(I0, i)$
         $\land wrts(c[I0], deps(x, Dep\_cr, i))$
         $\land y' \ = Op(y, g(i))$
         $\land rs' = [rs$ EXCEPT $![I0][i] =$ TRUE$]$
         $\land X' = X$
      BY $\langle 2\rangle 0$, $\langle 4\rangle 0$, $\langle 3\rangle 3$  DEF $TypeInv$, $R$, $St$

    $\langle 4\rangle 2.$ $\land \forall j \in It(x) \setminus \{i\} : red(I0, j)$
         $\land \neg red(I0, i)$
      $\langle 5\rangle 1.$ $It(x)' = It(x)$
        BY $\langle 4\rangle 1$  DEF $It$, $m$, $n$
      $\langle 5\rangle 2.$ $\neg red(I0, i)$
        BY $\langle 4\rangle 1$
      $\langle 5\rangle 3.$ $rs' = [rs$ EXCEPT $![I0][i] =$ TRUE$]$
        BY $\langle 4\rangle 1$
      $\langle 5\rangle 4.$ $\forall j \in It(x) : red(I0, j)'$
        BY $\langle 4\rangle 0$, $\langle 5\rangle 1$  DEF $end$
      $\langle 5\rangle$ QED
        BY $\langle 5\rangle 2$, $\langle 5\rangle 3$, $\langle 5\rangle 4$

    $\langle 4\rangle 3.$ $\land \forall j \in It(x) : wrt(c[I0][j])$
         $\land \forall j \in It(x) : wrt(p[I0][j])$
      $\langle 5\rangle 1.$ $\forall j \in It(x) : wrt(c[I0][j])$
        $\langle 6\rangle 1.$ $wrt(c[I0][i])$
          BY $\langle 4\rangle 1$  DEF $wrts$, $deps$
        $\langle 6\rangle 2.$ $\forall j \in It(x) \setminus \{i\} : wrt(c[I0][j])$
          BY $\langle 2\rangle 0$, $\langle 3\rangle 3$, $\langle 3\rangle 5$, $\langle 4\rangle 2$
        $\langle 6\rangle$ QED
          BY $\langle 2\rangle 0$, $\langle 3\rangle 3$, $\langle 6\rangle 1$, $\langle 6\rangle 2$
      $\langle 5\rangle 2.$ $\forall j \in It(x) : wrt(p[I0][j])$
        BY $\langle 3\rangle 5$, $\langle 5\rangle 1$
      $\langle 5\rangle$ QED
        BY $\langle 5\rangle 1$, $\langle 5\rangle 2$
```

```
⟨4⟩4. $\wedge\ \forall j \in It(x) : Gp(x)[j] \quad = p[I0][j]$
      $\wedge\ Gp(x) \in St(Tp) \wedge p[I0] \in St(Tp)$
  ⟨5⟩1. $Gp(x) \in St(Tp)$
    ⟨6⟩1. $\forall j \in Nat : gp(x, j) \in Tp \cup \{Undef\}$
      BY ⟨3⟩2, ⟨3⟩3, $H\_BFunType$
    ⟨6⟩ QED
      BY ⟨3⟩3, ⟨6⟩1  DEF $Gp$, $St$
  ⟨5⟩2. $p[I0] \in St(Tp)$
    BY ⟨2⟩0, ⟨3⟩3  DEF $TypeInv$
  ⟨5⟩3. $\forall j \in It(x) : Gp(x)[j] = p[I0][j]$
    ⟨6⟩0. SUFFICES ASSUME NEW $j \in It(x)$
                   PROVE  $gp(x, j) = fp(x, p[I0], j)$
      BY ⟨2⟩0, ⟨3⟩3, ⟨4⟩3  DEF $PInv$, $Gp$
    ⟨6⟩ QED
      BY ⟨3⟩2, ⟨4⟩3, $H\_ProdEqInv$
  ⟨5⟩ QED
    BY ⟨5⟩1, ⟨5⟩2, ⟨5⟩3  DEF $St$

⟨4⟩5. $\wedge\ \forall j \in It(x) : Fc(x, Gp(x))[j] \quad = c[I0][j]$
      $\wedge\ Fc(x, Gp(x)) \in St(Tc) \wedge c[I0] \in St(Tc)$
  ⟨5⟩1. $Fc(x, Gp(x)) \in St(Tc)$
    ⟨6⟩1. $\forall j \in Nat : fc(x, Gp(x), j) \in Tc \cup \{Undef\}$
      BY ⟨3⟩2, ⟨3⟩3, ⟨4⟩4, $H\_BFunType$
    ⟨6⟩ QED
      BY ⟨3⟩3, ⟨6⟩1  DEF $Fc$, $St$
  ⟨5⟩2. $c[I0] \in St(Tc)$
    BY ⟨2⟩0, ⟨3⟩3  DEF $TypeInv$
  ⟨5⟩3. $\forall j \in It(x) : Fc(x, Gp(x))[j] = c[I0][j]$
    ⟨6⟩0. SUFFICES ASSUME NEW $j \in It(x)$
                   PROVE  $fc(x, Gp(x), j) = fc(x, p[I0], j)$
      BY ⟨2⟩0, ⟨3⟩3, ⟨4⟩3  DEF $CInv$, $Fc$
    ⟨6⟩1. $eqs(Gp(x), p[I0], deps(x, Dep\_pc, j))$
      ⟨7⟩1. $deps(x, Dep\_pc, j) \subseteq It(x)$
        BY ⟨3⟩2, ⟨3⟩3, $H\_Type$ DEF $deps$, $It$
      ⟨7⟩2. $wrts(p[I0], deps(x, Dep\_pc, j))$
        BY ⟨4⟩3, ⟨7⟩1  DEF $wrts$
      ⟨7⟩3. $\forall k \in deps(x, Dep\_pc, j) :$
              $wrt(p[I0][k]) \wedge Gp(x)[k] = p[I0][k]$
        BY ⟨4⟩4, ⟨7⟩1, ⟨7⟩2  DEF $wrts$
      ⟨7⟩ QED
        BY ⟨7⟩3  DEF $eqs$
    ⟨6⟩2. $Gp(x) \in St(Tp) \wedge p[I0] \in St(Tp)$
      BY ⟨4⟩4
    ⟨6⟩3. $fc(x, Gp(x), j) = fc(x, p[I0], j)$
      BY ⟨3⟩2, ⟨3⟩3, ⟨6⟩1, ⟨6⟩2, $H\_fcRelevance$
    ⟨6⟩ QED
      BY ⟨6⟩3
  ⟨5⟩ QED
    BY ⟨5⟩1, ⟨5⟩2, ⟨5⟩3  DEF $St$

⟨4⟩6. $\wedge\ \forall j \in It(x) : f(j) = g(j)$
      $\wedge\ \forall j \in It(x) : f(j) \in D \wedge g(j) \in D$
```

```
⟨5⟩1. $\forall j \in It(x) : f(j) = g(j)$
  ⟨6⟩0. SUFFICES ASSUME NEW $j \in It(x)$
                 PROVE  $fr(x, Fc(x, Gp(x)), j) = fr(x, c[I0], j)$
    BY ⟨3⟩3 DEF $Fr$
  ⟨6⟩1. $eqs(Fc(x, Gp(x)), c[I0], deps(x, Dep\_cr, j))$
    ⟨7⟩1. $deps(x, Dep\_cr, j) \subseteq It(x)$
      BY ⟨3⟩2, ⟨3⟩3, $H\_Type$ DEF $deps$, $It$
    ⟨7⟩2. $wrts(c[I0], deps(x, Dep\_cr, j))$
      BY ⟨4⟩3, ⟨7⟩1 DEF $wrts$
    ⟨7⟩3. $\forall k \in deps(x, Dep\_cr, j) :$
              $wrt(c[I0][k]) \land Fc(x, Gp(x))[k] = c[I0][k]$
      BY ⟨4⟩5, ⟨7⟩1, ⟨7⟩2 DEF $wrts$
    ⟨7⟩ QED
      BY ⟨7⟩3 DEF $eqs$
  ⟨6⟩2. $Fc(x, Gp(x)) \in St(Tc) \land c[I0] \in St(Tc)$
    BY ⟨4⟩5
  ⟨6⟩3. $fr(x, Fc(x, Gp(x)), j) = fr(x, c[I0], j)$
    BY ⟨3⟩2, ⟨3⟩3, ⟨6⟩1, ⟨6⟩2, $H\_frRelevance$
  ⟨6⟩ QED
    BY ⟨6⟩3
⟨5⟩2. $\forall j \in It(x) : f(j) \in D$
  ⟨6⟩1. $\forall j \in It(x) : deps(x, Dep\_cr, j) \subseteq It(x)$
    BY ⟨3⟩2, ⟨3⟩3, $H\_Type$ DEF $deps$, $It$
  ⟨6⟩2. $\forall j \in It(x) : fr(x, Fc(x, Gp(x)), j) \in D$
    BY ⟨3⟩2, ⟨4⟩3, ⟨4⟩5, ⟨6⟩1, $H\_BFunWD$ DEF $wrts$
  ⟨6⟩ QED
    BY ⟨3⟩3, ⟨6⟩2 DEF $Fr$, $St$
⟨5⟩3. $\forall j \in It(x) : g(j) \in D$
  ⟨6⟩1. $\forall j \in It(x) : deps(x, Dep\_cr, j) \subseteq It(x)$
    BY ⟨3⟩2, ⟨3⟩3, $H\_Type$ DEF $deps$, $It$
  ⟨6⟩2. $\forall j \in It(x) : fr(x, c[I0], j) \in D$
    BY ⟨3⟩2, ⟨4⟩3, ⟨4⟩5, ⟨6⟩1, $H\_BFunWD$ DEF $wrts$
  ⟨6⟩ QED
    BY ⟨6⟩2
⟨5⟩ QED
  BY ⟨5⟩1, ⟨5⟩2, ⟨5⟩3

⟨4⟩ DEFINE $Q1(j) \triangleq Q(j) \land j \neq i$
           $Q2(j) \triangleq Q(j) \land red(I0, j)$
⟨4⟩ HIDE DEF $Q$, $Q1$, $Q2$, $f$, $g$, $m$, $n$

⟨4⟩7. $M!BigOpP(m, n, Q, f) = Op(M!BigOpP(m, n, Q1, f), f(i))$
  ⟨5⟩1. $m \in Nat \land n \in Nat \land m \leq n$
    BY ⟨3⟩3, ⟨3⟩4
  ⟨5⟩2. $i \in m \,..\, n \land Q(i)$
    BY ⟨3⟩3 DEF $Q$
  ⟨5⟩3. $\forall j \in m \,..\, n : Q(j) \in$ BOOLEAN
    BY ⟨3⟩3, ⟨3⟩2, $H\_Type$, $m \,..\, n \subseteq Nat$ DEF $It$, $Q$, $m$, $n$
  ⟨5⟩4. $\forall j \in \{k \in m \,..\, n : Q(k)\} : f(j) \in D$
    BY ⟨4⟩6 DEF $It$, $m$, $n$, $Q$
  ⟨5⟩5. $MT!BigOpP(m, n, Q, f) = Op(MT!BigOpP(m, n, Q1, f), f(i))$
    BY ⟨5⟩1, ⟨5⟩2, ⟨5⟩3, ⟨5⟩4, $H\_AMon$, $MT!SplitRandomP$, $Isa$ DEF $Q1$
```

```
      ⟨5⟩ QED
        BY ⟨5⟩5, $H\_MeqMT$

    ⟨4⟩8. $M!BigOpP(m, n, Q1, f) = M!BigOpP(m, n, Q2, f)$
      ⟨5⟩1. $m \in Nat \land n \in Nat$
        BY ⟨3⟩4
      ⟨5⟩2. $\forall j \in \{k \in m..n : Q1(k) \land Q2(k)\} : f(j) \in D$
        BY ⟨4⟩6 DEF $Q, Q1, Q2, It, m, n$
      ⟨5⟩3. $\land \forall j \in m..n : Q1(j) \in$ BOOLEAN
            $\land \forall j \in m..n : Q2(j) \in$ BOOLEAN
        BY DEF $Q1, Q2$
      ⟨5⟩4. $\forall j \in m..n : Q1(j) \equiv Q2(j)$
        BY ⟨3⟩3, ⟨3⟩4, ⟨4⟩2 DEF $It, Q, Q1, Q2, m, n$
      ⟨5⟩5. $MT!BigOpP(m, n, Q1, f) = MT!BigOpP(m, n, Q2, f)$
        BY ⟨5⟩1, ⟨5⟩2, ⟨5⟩3, ⟨5⟩4, $H\_AMon$, $MT!PredicateEq$, $Isa$
      ⟨5⟩ QED
        BY ⟨5⟩5, $H\_MeqMT$

    ⟨4⟩9. $\land M!BigOpP(m, n, Q2, f) = M!BigOpP(m, n, Q2, g)$
          $\land f(i) = g(i)$
      ⟨5⟩1. $m \in Nat \land n \in Nat$
        BY ⟨3⟩4
      ⟨5⟩2. $\forall j \in m..n : Q2(j) \in$ BOOLEAN
        BY DEF $Q2$
      ⟨5⟩3. $\land \forall j \in \{k \in m..n : Q2(k)\} : f(j) \in D$
            $\land \forall j \in \{k \in m..n : Q2(k)\} : g(j) \in D$
        BY ⟨4⟩6 DEF $Q, Q2, It, m, n$
      ⟨5⟩4. $\forall j \in \{k \in m..n : Q2(k)\} : f(j) = g(j)$
        BY ⟨2⟩0, ⟨3⟩3, ⟨4⟩6 DEF $Q, Q2, It, m, n$
      ⟨5⟩5. $MT!BigOpP(m, n, Q2, f) = MT!BigOpP(m, n, Q2, g)$
        BY ⟨5⟩1, ⟨5⟩2, ⟨5⟩3, ⟨5⟩4, $H\_AMon$, $MT!FunctionEqP$, $IsaM$("blast")
      ⟨5⟩6. $f(i) = g(i)$
        BY ⟨4⟩6
      ⟨5⟩ QED
        BY ⟨5⟩5, ⟨5⟩6, $H\_MeqMT$

    ⟨4⟩E1. $A(x) = M!BigOpP(m, n, Q, f)$              BY DEF $A, Q, f, m, n$
    ⟨4⟩E2.    $@ = Op(M!BigOpP(m, n, Q1, f), f(i))$  BY ⟨4⟩7
    ⟨4⟩E3.    $@ = Op(M!BigOpP(m, n, Q2, f), f(i))$  BY ⟨4⟩8
    ⟨4⟩E4.    $@ = Op(M!BigOpP(m, n, Q2, g), g(i))$  BY ⟨4⟩9
    ⟨4⟩E5.    $@ = Op(y, g(i))$                      BY ⟨2⟩0, ⟨3⟩3 DEF $RInv2, Q, Q2, g, m, n$
    ⟨4⟩E6.    $@ = y'$                               BY ⟨4⟩1

    ⟨4⟩ QED
      BY ⟨4⟩E1, ⟨4⟩E2, ⟨4⟩E3, ⟨4⟩E4, ⟨4⟩E5, ⟨4⟩E6

  ⟨3⟩ QED
    BY ⟨3⟩1, ⟨3⟩A, ⟨3⟩B, ⟨3⟩C
⟨2⟩B.CASE $Done$
  ⟨3⟩0. SUFFICES ASSUME UNCHANGED $\langle in, vs \rangle$,
                        $end(I0)$
                 PROVE  $y' = A(x)$
    BY ⟨2⟩B DEF $Done, vs, A, M!BigOpP, M!BigOp, M!bigOp, end$
    ⟨3⟩1. $y = A(x)$
      BY ⟨2⟩0, ⟨3⟩0  DEF $Correctness$
    ⟨3⟩2. $y' = y$
      BY ⟨3⟩0  DEF $vs$
    ⟨3⟩ QED
      BY ⟨3⟩1, ⟨3⟩2
  ⟨2⟩C.CASE UNCHANGED $\langle in, vs \rangle$
    ⟨3⟩0. SUFFICES ASSUME UNCHANGED $\langle in, vs \rangle$,
                                  $end(I0)$
                           PROVE  $y' = A(x)$
      BY ⟨2⟩C  DEF $vs$, $Correctness$, $A$, $M!BigOpP$, $M!BigOp$, $M!bigOp$, $end$
    ⟨3⟩1. $y = A(x)$
      BY ⟨2⟩0, ⟨3⟩0  DEF $Correctness$
    ⟨3⟩2. $y' = y$
      BY ⟨3⟩0  DEF $vs$
    ⟨3⟩ QED
      BY ⟨3⟩1, ⟨3⟩2
  ⟨2⟩ QED
    BY ⟨2⟩0, ⟨2⟩A, ⟨2⟩B, ⟨2⟩C  DEF $Next$
⟨1⟩ QED
  BY ⟨1⟩1, ⟨1⟩2, $Thm\_Inv$, $PTL$ DEF $Spec$
```

C.2 Basic PCR: Refinement of a basic PCR in one step

```
──────────── MODULE $PCR\_A\_Thms$ ────────────

THEOREM $Thm\_Refinement \triangleq Spec \Rightarrow A1step!Spec$
⟨1⟩ DEFINE $x \triangleq X[I0]$
           $y \triangleq r[I0]$
           $m \triangleq lBnd(x)$
           $n \triangleq uBnd(x)$
           $Q(j) \triangleq prop(j)$
           $f(i) \triangleq Fr(x, Fc(x, Gp(x)))[i]$
⟨1⟩1. $Init \Rightarrow A1step!Init$
  ⟨2⟩ SUFFICES ASSUME $Init$
               PROVE  $A1step!Init$
    OBVIOUS
  ⟨2⟩1. $\land x \in T$
        $\land pre(x)$
        $\land y = id$
    BY $H\_Type$ DEF $Init$
  ⟨2⟩ QED
    BY ⟨2⟩1 DEF $A1step!Init$, $inS$, $outS$
⟨1⟩2. $\land Inv$
      $\land Correctness'$
      $\land [Next]_{\langle in,\, vs\rangle}$
      $\Rightarrow [A1step!Next]_{A1step}!vs$
  ⟨2⟩0. SUFFICES ASSUME $IndexInv$, $TypeInv$, $PInv$,
                        $CInv$, $RInv1$, $RInv2$,
                        $Correctness'$,
                        $[Next]_{\langle in,\, vs\rangle}$
                 PROVE  $[A1step!Next]_{\langle inS,\, outS\rangle}$
    BY DEF $A1step!vs$, $Inv$
  ⟨2⟩A.CASE $Step$
    ⟨3⟩0. SUFFICES ASSUME $\exists\, i \in It(x) : \lor P(I0, i)$
                                              $\lor C(I0, i)$
                                              $\lor R(I0, i)$
                   PROVE  $[A1step!Next]_{\langle inS,\, outS\rangle}$
      BY ⟨2⟩0, ⟨2⟩A DEF $Step$, $IndexInv$
    ⟨3⟩1. PICK $i \in It(x) : \lor P(I0, i)$
                              $\lor C(I0, i)$
                              $\lor R(I0, i)$
      BY ⟨3⟩0
    ⟨3⟩2. $x \in T$
      BY ⟨2⟩0 DEF $IndexInv$, $TypeInv$, $WDIndex$, $wrt$
    ⟨3⟩3. $\land I0 \in Seq(Nat)$
          $\land I0 \in WDIndex$
          $\land i \in Nat$
          $\land i \in \{k \in m \,..\, n : Q(k)\}$
          $\land It(x) \subseteq Nat$
      BY ⟨2⟩0, ⟨3⟩2, $H\_Type$ DEF $IndexInv$, $WDIndex$, $It$
    ⟨3⟩4. $m \in Nat \land n \in Nat$
      BY ⟨2⟩0, ⟨3⟩2, $H\_Type$ DEF $TypeInv$, $m$, $n$
    ⟨3⟩5. $\forall\, j \in It(x)$ :
            $\land red(I0, j) \Rightarrow wrt(c[I0][j])$
            $\land wrt(c[I0][j]) \Rightarrow wrt(p[I0][j])$
    BY ⟨2⟩0  DEF CInv, RInv1, wrts, deps

⟨3⟩A.CASE P(I0, i)
  ⟨4⟩0. SUFFICES ASSUME P(I0, i)
                 PROVE  UNCHANGED ⟨inS, outS⟩
     BY ⟨2⟩0, ⟨3⟩A  DEF P
  ⟨4⟩1. ∧ ¬wrt(p[I0][i])
        ∧ ⟨X, rs⟩' = ⟨X, rs⟩             BY ⟨4⟩0  DEF P
  ⟨4⟩2. ¬wrt(p[I0][i]) ⇒ ¬red(I0, i)    BY ⟨3⟩5
  ⟨4⟩3. ¬red(I0, i)                      BY ⟨4⟩1, ⟨4⟩2
  ⟨4⟩4. ¬end(I0)                         BY ⟨4⟩3  DEF end
  ⟨4⟩5. ¬end(I0)'                        BY ⟨4⟩4, ⟨4⟩1  DEF end, It
  ⟨4⟩6. ⟨inS, outS⟩' = ⟨inS, outS⟩      BY ⟨4⟩1, ⟨4⟩4, ⟨4⟩5  DEF inS, outS
  ⟨4⟩ QED
    BY ⟨4⟩6

⟨3⟩B.CASE C(I0, i)
  ⟨4⟩0. SUFFICES ASSUME C(I0, i)
                 PROVE  UNCHANGED ⟨inS, outS⟩
    BY ⟨2⟩0, ⟨3⟩B  DEF C
  ⟨4⟩1. ∧ ¬wrt(c[I0][i])
        ∧ ⟨X, rs⟩' = ⟨X, rs⟩             BY ⟨4⟩0  DEF C
  ⟨4⟩2. ¬wrt(c[I0][i]) ⇒ ¬red(I0, i)    BY ⟨3⟩5
  ⟨4⟩3. ¬red(I0, i)                      BY ⟨4⟩1, ⟨4⟩2
  ⟨4⟩4. ¬end(I0)                         BY ⟨4⟩3  DEF end
  ⟨4⟩5. ¬end(I0)'                        BY ⟨4⟩4, ⟨4⟩1  DEF end, It
  ⟨4⟩6. ⟨inS, outS⟩' = ⟨inS, outS⟩      BY ⟨4⟩1, ⟨4⟩4, ⟨4⟩5  DEF inS, outS
  ⟨4⟩ QED
    BY ⟨4⟩6

⟨3⟩C.CASE R(I0, i) ∧ end(I0)'
  ⟨4⟩0. SUFFICES ASSUME R(I0, i),
                        end(I0)'
                 PROVE  outS' = A1step!A(x)
    BY ⟨2⟩0, ⟨3⟩C  DEF R, A1step!Next, inS
  ⟨4⟩ DEFINE g(j) ≜ fr(x, c[I0], j)
  ⟨4⟩1. ∧ ¬red(I0, i)
        ∧ wrts(c[I0], deps(x, Dep_cr, i))
        ∧ y'  = Op(y, g(i))
        ∧ rs' = [rs EXCEPT ![I0][i] = TRUE]
        ∧ X' = X
    BY ⟨2⟩0, ⟨4⟩0, ⟨3⟩3  DEF TypeInv, R, St

  ⟨4⟩2. ∧ ∀ j ∈ It(x) \ {i} : red(I0, j)
        ∧ ¬red(I0, i)
    ⟨5⟩1. It(x)' = It(x)
      BY ⟨4⟩1  DEF It, m, n
    ⟨5⟩2. ¬red(I0, i)
      BY ⟨4⟩1
    ⟨5⟩3. rs' = [rs EXCEPT ![I0][i] = TRUE]
      BY ⟨4⟩1
    ⟨5⟩4. ∀ j ∈ It(x) : red(I0, j)'
      BY ⟨4⟩0, ⟨5⟩1  DEF end
    $\langle 5\rangle$ QED
      BY $\langle 5\rangle 2$, $\langle 5\rangle 3$, $\langle 5\rangle 4$

  $\langle 4\rangle 3.\ \land \forall j \in It(x) : wrt(c[I0][j])$
        $\land \forall j \in It(x) : wrt(p[I0][j])$
    $\langle 5\rangle 1.\ \forall j \in It(x) : wrt(c[I0][j])$
      $\langle 6\rangle 1.\ wrt(c[I0][i])$
        BY $\langle 4\rangle 1$ DEF $wrts$, $deps$
      $\langle 6\rangle 2.\ \forall j \in It(x) \setminus \{i\} : wrt(c[I0][j])$
        BY $\langle 2\rangle 0$, $\langle 3\rangle 3$, $\langle 3\rangle 5$, $\langle 4\rangle 2$
      $\langle 6\rangle$ QED
        BY $\langle 2\rangle 0$, $\langle 3\rangle 3$, $\langle 6\rangle 1$, $\langle 6\rangle 2$
    $\langle 5\rangle 2.\ \forall j \in It(x) : wrt(p[I0][j])$
      BY $\langle 3\rangle 5$, $\langle 5\rangle 1$
    $\langle 5\rangle$ QED
      BY $\langle 5\rangle 1$, $\langle 5\rangle 2$

  $\langle 4\rangle 4.\ \land \forall j \in It(x) : Gp(x)[j] \quad = p[I0][j]$
        $\land Gp(x) \in St(Tp) \land p[I0] \in St(Tp)$
    $\langle 5\rangle 1.\ Gp(x) \in St(Tp)$
      $\langle 6\rangle 1.\ \forall j \in Nat : gp(x, j) \in Tp \cup \{Undef\}$
        BY $\langle 3\rangle 2$, $\langle 3\rangle 3$, $H\_BFunType$
      $\langle 6\rangle$ QED
        BY $\langle 3\rangle 3$, $\langle 6\rangle 1$ DEF $Gp$, $St$
    $\langle 5\rangle 2.\ p[I0] \in St(Tp)$
      BY $\langle 2\rangle 0$, $\langle 3\rangle 3$ DEF $TypeInv$
    $\langle 5\rangle 3.\ \forall j \in It(x) : Gp(x)[j] = p[I0][j]$
      $\langle 6\rangle 0.$ SUFFICES ASSUME NEW $j \in It(x)$
                       PROVE $gp(x, j) = fp(x, p[I0], j)$
        BY $\langle 2\rangle 0$, $\langle 3\rangle 3$, $\langle 4\rangle 3$ DEF $PInv$, $Gp$
      $\langle 6\rangle$ QED
        BY $\langle 3\rangle 2$, $\langle 4\rangle 3$, $H\_ProdEqInv$
    $\langle 5\rangle$ QED
      BY $\langle 5\rangle 1$, $\langle 5\rangle 2$, $\langle 5\rangle 3$ DEF $St$

  $\langle 4\rangle 5.\ \land \forall j \in It(x) : Fc(x, Gp(x))[j] \quad = c[I0][j]$
        $\land Fc(x, Gp(x)) \in St(Tc) \land c[I0] \in St(Tc)$
    $\langle 5\rangle 1.\ Fc(x, Gp(x)) \in St(Tc)$
      $\langle 6\rangle 1.\ \forall j \in Nat : fc(x, Gp(x), j) \in Tc \cup \{Undef\}$
        BY $\langle 3\rangle 2$, $\langle 3\rangle 3$, $\langle 4\rangle 4$, $H\_BFunType$
      $\langle 6\rangle$ QED
        BY $\langle 3\rangle 3$, $\langle 6\rangle 1$ DEF $Fc$, $St$
    $\langle 5\rangle 2.\ c[I0] \in St(Tc)$
      BY $\langle 2\rangle 0$, $\langle 3\rangle 3$ DEF $TypeInv$
    $\langle 5\rangle 3.\ \forall j \in It(x) : Fc(x, Gp(x))[j] = c[I0][j]$
      $\langle 6\rangle 0.$ SUFFICES ASSUME NEW $j \in It(x)$
                       PROVE $fc(x, Gp(x), j) = fc(x, p[I0], j)$
        BY $\langle 2\rangle 0$, $\langle 3\rangle 3$, $\langle 4\rangle 3$ DEF $CInv$, $Fc$
      $\langle 6\rangle 1.\ eqs(Gp(x), p[I0], deps(x, Dep\_pc, j))$
        $\langle 7\rangle 1.\ deps(x, Dep\_pc, j) \subseteq It(x)$
          BY $\langle 3\rangle 2$, $\langle 3\rangle 3$, $H\_Type$ DEF $deps$, $It$
        $\langle 7\rangle 2.\ wrts(p[I0], deps(x, Dep\_pc, j))$
          BY $\langle 4\rangle 3$, $\langle 7\rangle 1$ DEF $wrts$
        $\langle 7\rangle 3.\ \forall k \in deps(x, Dep\_pc, j) :$
                $wrt(p[I0][k]) \land Gp(x)[k] = p[I0][k]$
          BY $\langle 4\rangle 4, \langle 7\rangle 1, \langle 7\rangle 2$  DEF $wrts$
        $\langle 7\rangle$ QED
          BY $\langle 7\rangle 3$  DEF $eqs$
      $\langle 6\rangle 2.\ Gp(x) \in St(Tp) \land p[I0] \in St(Tp)$
        BY $\langle 4\rangle 4$
      $\langle 6\rangle 3.\ fc(x, Gp(x), j) = fc(x, p[I0], j)$
        BY $\langle 3\rangle 2, \langle 3\rangle 3, \langle 6\rangle 1, \langle 6\rangle 2, H\_fcRelevance$
      $\langle 6\rangle$ QED
        BY $\langle 6\rangle 3$
    $\langle 5\rangle$ QED
      BY $\langle 5\rangle 1, \langle 5\rangle 2, \langle 5\rangle 3$  DEF $St$

  $\langle 4\rangle 6.\ \land \forall j \in It(x) : f(j) = g(j)$
        $\land \forall j \in It(x) : f(j) \in D \land g(j) \in D$
    $\langle 5\rangle 1.\ \forall j \in It(x) : f(j) = g(j)$
      $\langle 6\rangle 0.$ SUFFICES ASSUME NEW $j \in It(x)$
                     PROVE  $fr(x, Fc(x, Gp(x)), j) = fr(x, c[I0], j)$
        BY $\langle 3\rangle 3$  DEF $Fr$
      $\langle 6\rangle 1.\ eqs(Fc(x, Gp(x)), c[I0], deps(x, Dep\_cr, j))$
        $\langle 7\rangle 1.\ deps(x, Dep\_cr, j) \subseteq It(x)$
          BY $\langle 3\rangle 2, \langle 3\rangle 3, H\_Type$ DEF $deps, It$
        $\langle 7\rangle 2.\ wrts(c[I0], deps(x, Dep\_cr, j))$
          BY $\langle 4\rangle 3, \langle 7\rangle 1$  DEF $wrts$
        $\langle 7\rangle 3.\ \forall k \in deps(x, Dep\_cr, j) :$
                $wrt(c[I0][k]) \land Fc(x, Gp(x))[k] = c[I0][k]$
          BY $\langle 4\rangle 5, \langle 7\rangle 1, \langle 7\rangle 2$  DEF $wrts$
        $\langle 7\rangle$ QED
          BY $\langle 7\rangle 3$  DEF $eqs$
      $\langle 6\rangle 2.\ Fc(x, Gp(x)) \in St(Tc) \land c[I0] \in St(Tc)$
        BY $\langle 4\rangle 5$
      $\langle 6\rangle 3.\ fr(x, Fc(x, Gp(x)), j) = fr(x, c[I0], j)$
        BY $\langle 3\rangle 2, \langle 3\rangle 3, \langle 6\rangle 1, \langle 6\rangle 2, H\_frRelevance$
      $\langle 6\rangle$ QED
        BY $\langle 6\rangle 3$
    $\langle 5\rangle 2.\ \forall j \in It(x) : f(j) \in D$
      $\langle 6\rangle 1.\ \forall j \in It(x) : deps(x, Dep\_cr, j) \subseteq It(x)$
        BY $\langle 3\rangle 2, \langle 3\rangle 3, H\_Type$ DEF $deps, It$
      $\langle 6\rangle 2.\ \forall j \in It(x) : fr(x, Fc(x, Gp(x)), j) \in D$
        BY $\langle 3\rangle 2, \langle 4\rangle 3, \langle 4\rangle 5, \langle 6\rangle 1, H\_BFunWD$ DEF $wrts$
      $\langle 6\rangle$ QED
        BY $\langle 3\rangle 3, \langle 6\rangle 2$  DEF $Fr, St$
    $\langle 5\rangle 3.\ \forall j \in It(x) : g(j) \in D$
      $\langle 6\rangle 1.\ \forall j \in It(x) : deps(x, Dep\_cr, j) \subseteq It(x)$
        BY $\langle 3\rangle 2, \langle 3\rangle 3, H\_Type$ DEF $deps, It$
      $\langle 6\rangle 2.\ \forall j \in It(x) : fr(x, c[I0], j) \in D$
        BY $\langle 3\rangle 2, \langle 4\rangle 3, \langle 4\rangle 5, \langle 6\rangle 1, H\_BFunWD$ DEF $wrts$
      $\langle 6\rangle$ QED
        BY $\langle 6\rangle 2$
    $\langle 5\rangle$ QED
      BY $\langle 5\rangle 1, \langle 5\rangle 2, \langle 5\rangle 3$
```

```
$\langle 4 \rangle$ DEFINE $Q1(j) \triangleq Q(j) \land j \neq i$
                 $Q2(j) \triangleq Q(j) \land red(I0, j)$
$\langle 4 \rangle$ HIDE  DEF $Q, Q1, Q2, f, g, m, n$

$\langle 4 \rangle 7.$ $M!BigOpP(m, n, Q, f) = Op(M!BigOpP(m, n, Q1, f), f(i))$
  $\langle 5 \rangle 1.$ $m \in Nat \land n \in Nat \land m \leq n$
    BY $\langle 3 \rangle 3, \langle 3 \rangle 4$
  $\langle 5 \rangle 2.$ $i \in m \,..\, n \land Q(i)$
    BY $\langle 3 \rangle 3$  DEF $Q$
  $\langle 5 \rangle 3.$ $\forall j \in m \,..\, n : Q(j) \in$ BOOLEAN
    BY $\langle 3 \rangle 3, \langle 3 \rangle 2, H\_Type, m \,..\, n \subseteq Nat$ DEF $It, Q, m, n$
  $\langle 5 \rangle 4.$ $\forall j \in \{k \in m \,..\, n : Q(k)\} : f(j) \in D$
    BY $\langle 4 \rangle 6$  DEF $It, m, n, Q$
  $\langle 5 \rangle 5.$ $MT!BigOpP(m, n, Q, f) = Op(MT!BigOpP(m, n, Q1, f), f(i))$
    BY $\langle 5 \rangle 1, \langle 5 \rangle 2, \langle 5 \rangle 3, \langle 5 \rangle 4, H\_AMon, MT!SplitRandomP, Isa$ DEF $Q1$
  $\langle 5 \rangle$ QED
    BY $\langle 5 \rangle 5, H\_MeqMT$

$\langle 4 \rangle 8.$ $M!BigOpP(m, n, Q1, f) = M!BigOpP(m, n, Q2, f)$
  $\langle 5 \rangle 1.$ $m \in Nat \land n \in Nat$
    BY $\langle 3 \rangle 4$
  $\langle 5 \rangle 2.$ $\forall j \in \{k \in m \,..\, n : Q1(k) \land Q2(k)\} : f(j) \in D$
    BY $\langle 4 \rangle 6$  DEF $Q, Q1, Q2, It, m, n$
  $\langle 5 \rangle 3.$ $\land \forall j \in m \,..\, n : Q1(j) \in$ BOOLEAN
       $\land \forall j \in m \,..\, n : Q2(j) \in$ BOOLEAN
    BY  DEF $Q1, Q2$
  $\langle 5 \rangle 4.$ $\forall j \in m \,..\, n : Q1(j) \equiv Q2(j)$
    BY $\langle 3 \rangle 3, \langle 3 \rangle 4, \langle 4 \rangle 2$  DEF $It, Q, Q1, Q2, m, n$
  $\langle 5 \rangle 5.$ $MT!BigOpP(m, n, Q1, f) = MT!BigOpP(m, n, Q2, f)$
    BY $\langle 5 \rangle 1, \langle 5 \rangle 2, \langle 5 \rangle 3, \langle 5 \rangle 4, H\_AMon, MT!PredicateEq, Isa$
  $\langle 5 \rangle$ QED
    BY $\langle 5 \rangle 5, H\_MeqMT$

$\langle 4 \rangle 9.$ $\land M!BigOpP(m, n, Q2, f) = M!BigOpP(m, n, Q2, g)$
      $\land f(i) = g(i)$
  $\langle 5 \rangle 1.$ $m \in Nat \land n \in Nat$
    BY $\langle 3 \rangle 4$
  $\langle 5 \rangle 2.$ $\forall j \in m \,..\, n : Q2(j) \in$ BOOLEAN
    BY  DEF $Q2$
  $\langle 5 \rangle 3.$ $\land \forall j \in \{k \in m \,..\, n : Q2(k)\} : f(j) \in D$
       $\land \forall j \in \{k \in m \,..\, n : Q2(k)\} : g(j) \in D$
    BY $\langle 4 \rangle 6$  DEF $Q, Q2, It, m, n$
  $\langle 5 \rangle 4.$ $\forall j \in \{k \in m \,..\, n : Q2(k)\} : f(j) = g(j)$
    BY $\langle 2 \rangle 0, \langle 3 \rangle 3, \langle 4 \rangle 6$  DEF $Q, Q2, It, m, n$
  $\langle 5 \rangle 5.$ $MT!BigOpP(m, n, Q2, f) = MT!BigOpP(m, n, Q2, g)$
    BY $\langle 5 \rangle 1, \langle 5 \rangle 2, \langle 5 \rangle 3, \langle 5 \rangle 4, H\_AMon, MT!FunctionEqP, IsaM($"blast"$)$
  $\langle 5 \rangle 6.$ $f(i) = g(i)$
    BY $\langle 4 \rangle 6$
  $\langle 5 \rangle$ QED
    BY $\langle 5 \rangle 5, \langle 5 \rangle 6, H\_MeqMT$

$\langle 4 \rangle$E1. $A(x) = M!BigOpP(m, n, Q, f)$                BY  DEF $A, Q, f, m, n$
$\langle 4 \rangle$E2.     $@ = Op(M!BigOpP(m, n, Q1, f), f(i))$  BY $\langle 4 \rangle 7$
```

```
    $\langle 4\rangle$E3.  $@ = Op(M!BigOpP(m, n, Q2, f), f(i))$  BY $\langle 4\rangle 8$
    $\langle 4\rangle$E4.  $@ = Op(M!BigOpP(m, n, Q2, g), g(i))$  BY $\langle 4\rangle 9$
    $\langle 4\rangle$E5.  $@ = Op(y, g(i))$                BY $\langle 2\rangle 0$, $\langle 3\rangle 3$  DEF $RInv2, Q, Q2, g, m, n$
    $\langle 4\rangle$E6.  $@ = y'$                          BY $\langle 4\rangle 1$

    $\langle 4\rangle 10$. $outS' = y'$           BY $\langle 4\rangle 0$  DEF $outS, y$
    $\langle 4\rangle 11$.     $@ = A(x)$         BY $\langle 4\rangle$E1, $\langle 4\rangle$E2, $\langle 4\rangle$E3, $\langle 4\rangle$E4, $\langle 4\rangle$E5, $\langle 4\rangle$E6
    $\langle 4\rangle 12$.     $@ = A1step!A(x)$  BY $H\_A1stepEqA$
    $\langle 4\rangle$ QED
      BY $\langle 4\rangle 10$, $\langle 4\rangle 11$, $\langle 4\rangle 12$
```

A shorter proof, reusing the *Correctness* property:

```
$\langle 4\rangle 2$. $outS' = y'$        BY  $\langle 4\rangle 0$  DEF  $outS, y$
$\langle 4\rangle 3$.    $@ = A(x)'$      BY  $\langle 2\rangle 0$, $\langle 4\rangle 0$  DEF  $Correctness$
$\langle 4\rangle 4$.    $@ = A(x)$       BY  $\langle 4\rangle 1$ DEF A, $M!BigOpP$, $M!BigOp$, $M!bigOp$
$\langle 4\rangle 5$.    $@ = A1step!A(x)$BY  $H\_A1stepEqA$
$\langle 4\rangle$ QED
 BY  $\langle 4\rangle 2$, $\langle 4\rangle 3$, $\langle 4\rangle 4$, $\langle 4\rangle 5$
```

```
  $\langle 3\rangle$D.CASE $R(I0, i) \land \neg end(I0)'$
    $\langle 4\rangle 0$. SUFFICES ASSUME $R(I0, i)$,
                                  $\neg end(I0)'$
                          PROVE  UNCHANGED $\langle inS, outS\rangle$
      BY $\langle 2\rangle 0$, $\langle 3\rangle$D  DEF $R$, $A1step!Next$, $inS$
    $\langle 4\rangle 1$. $\land \neg red(I0, i)$
         $\land X' = X$
      BY $\langle 4\rangle 0$  DEF $R$
    $\langle 4\rangle 2$. $\neg end(I0)$
      BY $\langle 4\rangle 1$  DEF $end$
    $\langle 4\rangle 3$. $outS\ = id$
      BY $\langle 4\rangle 2$  DEF $outS$
    $\langle 4\rangle 4$. $outS' = id$
      BY $\langle 4\rangle 0$  DEF $outS$
    $\langle 4\rangle$ QED
      BY $\langle 4\rangle 1$, $\langle 4\rangle 3$, $\langle 4\rangle 4$  DEF $inS$

  $\langle 3\rangle$ QED
    BY $\langle 3\rangle 1$, $\langle 3\rangle$A, $\langle 3\rangle$B, $\langle 3\rangle$C, $\langle 3\rangle$D
$\langle 2\rangle$B.CASE $Done$
  $\langle 3\rangle 0$. SUFFICES ASSUME UNCHANGED $\langle in, vs\rangle$
                   PROVE  UNCHANGED $\langle inS, outS\rangle$
    BY $\langle 2\rangle$B  DEF $Done, vs$
  $\langle 3\rangle 1$. $\langle inS, outS\rangle' = \langle inS, outS\rangle$
    BY $\langle 3\rangle 0$  DEF $inS, outS, vs, end$
  $\langle 3\rangle$ QED
    BY $\langle 3\rangle 1$
$\langle 2\rangle$C.CASE UNCHANGED $\langle in, vs\rangle$
  $\langle 3\rangle 0$. SUFFICES ASSUME UNCHANGED $\langle in, vs\rangle$
                   PROVE  UNCHANGED $\langle inS, outS\rangle$
    BY $\langle 2\rangle$C  DEF $vs$
  $\langle 3\rangle 1$. $\langle inS, outS\rangle' = \langle inS, outS\rangle$
    BY $\langle 3\rangle 0$  DEF $inS, outS, vs, end$
    ⟨3⟩ QED
      BY ⟨3⟩1
  ⟨2⟩ QED
    BY ⟨2⟩0, ⟨2⟩A, ⟨2⟩B, ⟨2⟩C  DEF Next
⟨1⟩ QED
  BY ⟨1⟩1, ⟨1⟩2, Thm_Inv, Thm_Correctness, PTL DEF Spec, A1step!Spec
```

C.3 Basic PCR with left reducer: Correctness

```
┌──────────────────── MODULE PCR_ArLeft_Thms ────────────────────┐
THEOREM Thm_Correctness ≜ Spec ⇒ □Correctness!1
⟨1⟩ DEFINE x    ≜ X[I0]
           y    ≜ r[I0]
           m    ≜ lBnd(x)
           n    ≜ uBnd(x)
           f(i) ≜ Fr(x, Fc(x, Gp(x)))[i]
⟨1⟩1. Init ⇒ Correctness!1
  ⟨2⟩0. SUFFICES ASSUME Init,
                        end(I0)
                 PROVE  y = A(x)
    BY DEF Correctness
  ⟨2⟩1. ∧ I0    ∈ Seq(Nat)
        ∧ x     ∈ T
        ∧ It(x) ⊆ Nat
    BY ⟨2⟩0, H_Type DEF Init, It
  ⟨2⟩2. m ∈ Nat ∧ n ∈ Nat
    BY ⟨2⟩1, H_Type
  ⟨2⟩A.CASE It(x) = {}
    ⟨3⟩1. m > n
      BY ⟨2⟩A, ⟨2⟩2 DEF It, m, n
    ⟨3⟩2. A(x) = id
      BY ⟨2⟩2, ⟨3⟩1, H_Mon, H_MeqMT DEF A, M!BigOp
    ⟨3⟩3. y = id
      BY ⟨2⟩0, ⟨2⟩1 DEF Init
    ⟨3⟩ QED
      BY ⟨3⟩2, ⟨3⟩3
  ⟨2⟩B.CASE It(x) ≠ {}
    ⟨3⟩1. ∀ i ∈ It(x) : red(I0, i)        BY ⟨2⟩0 DEF end
    ⟨3⟩2. ∀ i ∈ Nat : ¬red(I0, i)         BY ⟨2⟩0, ⟨2⟩1 DEF Init
    ⟨3⟩3. FALSE                            BY ⟨2⟩1, ⟨2⟩B, ⟨3⟩1, ⟨3⟩2
    ⟨3⟩ QED                                 BY ⟨3⟩3
  ⟨2⟩ QED
    BY ⟨2⟩A, ⟨2⟩B
⟨1⟩2. ∧ Inv
      ∧ Correctness!1
      ∧ [Next]_⟨in, vs⟩
      ⇒ Correctness!1′
  ⟨2⟩0. SUFFICES ASSUME IndexInv, TypeInv, PInv,
                        CInv, RInv1, RInv2,
                        Correctness!1,
                        [Next]_⟨in, vs⟩
                 PROVE  Correctness!1′
    BY DEF Inv
  ⟨2⟩A.CASE Step
    ⟨3⟩0. SUFFICES ASSUME ∃ i ∈ It(x) : ∨ P(I0, i)
                                        ∨ C(I0, i)
                                        ∨ R(I0, i)
                   PROVE Correctness!1′
      BY ⟨2⟩0, ⟨2⟩A DEF Step, IndexInv
    ⟨3⟩1. PICK i ∈ It(x) : ∨ P(I0, i)
```

```
                                  $\lor C(I0, i)$
                                  $\lor R(I0, i)$
      BY $\langle 3\rangle 0$
    $\langle 3\rangle 2.\ x \in T$
      BY $\langle 2\rangle 0$  DEF $IndexInv$, $TypeInv$, $WDIndex$, $wrt$
    $\langle 3\rangle 3.\ \land I0 \in Seq(Nat)$
         $\land I0 \in WDIndex$
         $\land i \in Nat$
         $\land i \in m \,..\, n$
         $\land It(x) \subseteq Nat$
      BY $\langle 2\rangle 0$, $\langle 3\rangle 2$, $H\_Type$ DEF $IndexInv$, $WDIndex$, $It$
    $\langle 3\rangle 4.\ m \in Nat \land n \in Nat$
      BY $\langle 2\rangle 0$, $\langle 3\rangle 2$, $H\_Type$ DEF $TypeInv$, $m$, $n$
    $\langle 3\rangle 5.\ \forall j \in It(x) :$
           $\land red(I0, j) \qquad \Rightarrow wrt(c[I0][j])$
           $\land wrt(c[I0][j]) \ \Rightarrow wrt(p[I0][j])$
      BY $\langle 2\rangle 0$  DEF $CInv$, $RInv1$, $wrts$, $deps$

    $\langle 3\rangle$A.CASE $P(I0, i)$
      $\langle 4\rangle 0.$ SUFFICES ASSUME $P(I0, i)$,
                                     $end(I0)'$
                             PROVE   FALSE
        BY $\langle 2\rangle 0$, $\langle 3\rangle$A  DEF $P$, $Correctness$
      $\langle 4\rangle 1.\ \land \neg wrt(p[I0][i])$
           $\land \langle X, rs\rangle' = \langle X, rs\rangle$              BY $\langle 4\rangle 0$  DEF $P$
      $\langle 4\rangle 2.\ \neg wrt(p[I0][i]) \Rightarrow \neg red(I0, i)$  BY $\langle 3\rangle 5$
      $\langle 4\rangle 3.\ \neg red(I0, i)$                     BY $\langle 4\rangle 1$, $\langle 4\rangle 2$
      $\langle 4\rangle 4.\ \neg red(I0, i)'$                    BY $\langle 4\rangle 1$, $\langle 4\rangle 3$
      $\langle 4\rangle 5.\ \exists j \in It(x)' : \neg red(I0, j)'$   BY $\langle 4\rangle 1$, $\langle 4\rangle 4$  DEF $It$
      $\langle 4\rangle 6.$ FALSE                           BY $\langle 4\rangle 5$, $end(I0)'$ DEF $end$
      $\langle 4\rangle$ QED
        BY $\langle 4\rangle 6$

    $\langle 3\rangle$B.CASE $C(I0, i)$
      $\langle 4\rangle 0.$ SUFFICES ASSUME $C(I0, i)$,
                                     $end(I0)'$
                             PROVE   FALSE
        BY $\langle 2\rangle 0$, $\langle 3\rangle$B  DEF $C$, $Correctness$
      $\langle 4\rangle 1.\ \land \neg wrt(c[I0][i])$
           $\land \langle X, rs\rangle' = \langle X, rs\rangle$            BY $\langle 4\rangle 0$  DEF $C$
      $\langle 4\rangle 2.\ \neg wrt(c[I0][i]) \Rightarrow \neg red(I0, i)$  BY $\langle 3\rangle 5$
      $\langle 4\rangle 3.\ \neg red(I0, i)$                   BY $\langle 4\rangle 1$, $\langle 4\rangle 2$
      $\langle 4\rangle 4.\ \neg red(I0, i)'$                  BY $\langle 4\rangle 1$, $\langle 4\rangle 3$
      $\langle 4\rangle 5.\ \exists j \in It(x)' : \neg red(I0, j)'$  BY $\langle 4\rangle 1$, $\langle 4\rangle 4$  DEF $It$
      $\langle 4\rangle 6.$ FALSE                         BY $\langle 4\rangle 5$, $end(I0)'$ DEF $end$
      $\langle 4\rangle$ QED
        BY $\langle 4\rangle 6$

    $\langle 3\rangle$C.CASE $R(I0, i)$
      $\langle 4\rangle 0.$ SUFFICES ASSUME $R(I0, i)$,
                                     $end(I0)'$
                             PROVE   $y' = A(x)$
        BY $\langle 2\rangle 0$, $\langle 3\rangle$C  DEF $R$, $Correctness$, $A$, $M!BigOpP$, $M!BigOp$, $M!bigOp$
```

```
$\langle 4 \rangle$ DEFINE $g(j) \triangleq fr(x, c[I0], j)$
$\langle 4 \rangle 1.$ $\land \neg red(I0, i)$
     $\land wrts(c[I0], deps(x, Dep\_cr, i))$
     $\land i - 1 \geq m \Rightarrow red(I0, i - 1)$
     $\land y' = Op(y, g(i))$
     $\land rs' = [rs \text{ EXCEPT } ![I0][i] = \text{TRUE}]$
     $\land X' = X$
  BY $\langle 2 \rangle 0$, $\langle 4 \rangle 0$, $\langle 3 \rangle 3$ DEF $TypeInv$, $R$, $St$

$\langle 4 \rangle 2.$ $\land \forall j \in It(x) \setminus \{i\} : red(I0, j)$
     $\land \forall j \in It(x) : j < i \Rightarrow red(I0, j)$
     $\land \neg red(I0, i)$
  $\langle 5 \rangle 1.$ $It(x)' = It(x)$
    BY $\langle 4 \rangle 1$ DEF $It$, $m$, $n$
  $\langle 5 \rangle 2.$ $\neg red(I0, i)$
    BY $\langle 4 \rangle 1$
  $\langle 5 \rangle 3.$ $rs' = [rs \text{ EXCEPT } ![I0][i] = \text{TRUE}]$
    BY $\langle 4 \rangle 1$
  $\langle 5 \rangle 4.$ $\forall j \in It(x) : red(I0, j)'$
    BY $\langle 4 \rangle 0$, $\langle 5 \rangle 1$ DEF $end$
  $\langle 5 \rangle 5.$ $\forall j \in It(x) \setminus \{i\} : red(I0, j)$
    BY $\langle 5 \rangle 2$, $\langle 5 \rangle 3$, $\langle 5 \rangle 4$
  $\langle 5 \rangle 6.$ $\forall j \in It(x) : j < i \Rightarrow red(I0, j)$
    BY $\langle 5 \rangle 5$ DEF $It$
  $\langle 5 \rangle$ QED
    BY $\langle 5 \rangle 2$, $\langle 5 \rangle 5$, $\langle 5 \rangle 6$

$\langle 4 \rangle 3.$ $\land \forall j \in It(x) : wrt(c[I0][j])$
     $\land \forall j \in It(x) : wrt(p[I0][j])$
  $\langle 5 \rangle 1.$ $\forall j \in It(x) : wrt(c[I0][j])$
    $\langle 6 \rangle 1.$ $wrt(c[I0][i])$
      BY $\langle 4 \rangle 1$ DEF $wrts$, $deps$
    $\langle 6 \rangle 2.$ $\forall j \in It(x) \setminus \{i\} : wrt(c[I0][j])$
      BY $\langle 2 \rangle 0$, $\langle 3 \rangle 3$, $\langle 3 \rangle 5$, $\langle 4 \rangle 2$
    $\langle 6 \rangle$ QED
      BY $\langle 2 \rangle 0$, $\langle 3 \rangle 3$, $\langle 6 \rangle 1$, $\langle 6 \rangle 2$
  $\langle 5 \rangle 2.$ $\forall j \in It(x) : wrt(p[I0][j])$
    BY $\langle 3 \rangle 5$, $\langle 5 \rangle 1$
  $\langle 5 \rangle$ QED
    BY $\langle 5 \rangle 1$, $\langle 5 \rangle 2$

$\langle 4 \rangle 4.$ $\land \forall j \in It(x) : Gp(x)[j] = p[I0][j]$
     $\land Gp(x) \in St(Tp) \land p[I0] \in St(Tp)$
  $\langle 5 \rangle 1.$ $Gp(x) \in St(Tp)$
    $\langle 6 \rangle 1.$ $\forall j \in Nat : gp(x, j) \in Tp \cup \{Undef\}$
      BY $\langle 3 \rangle 2$, $\langle 3 \rangle 3$, $H\_BFunType$
    $\langle 6 \rangle$ QED
      BY $\langle 3 \rangle 3$, $\langle 6 \rangle 1$ DEF $Gp$, $St$
  $\langle 5 \rangle 2.$ $p[I0] \in St(Tp)$
    BY $\langle 2 \rangle 0$, $\langle 3 \rangle 3$ DEF $TypeInv$
  $\langle 5 \rangle 3.$ $\forall j \in It(x) : Gp(x)[j] = p[I0][j]$
    $\langle 6 \rangle 0.$ SUFFICES ASSUME NEW $j \in It(x)$
                   PROVE $gp(x, j) = fp(x, p[I0], j)$
```

```
        BY $\langle 2\rangle 0$, $\langle 3\rangle 3$, $\langle 4\rangle 3$ DEF $PInv$, $Gp$
      $\langle 6\rangle$ QED
        BY $\langle 3\rangle 2$, $\langle 4\rangle 3$, $H\_ProdEqInv$
    $\langle 5\rangle$ QED
      BY $\langle 5\rangle 1$, $\langle 5\rangle 2$, $\langle 5\rangle 3$ DEF $St$

  $\langle 4\rangle 5.$ $\land \forall j \in It(x) : Fc(x, Gp(x))[j] \quad = c[I0][j]$
       $\land Fc(x, Gp(x)) \in St(Tc) \land c[I0] \in St(Tc)$
    $\langle 5\rangle 1.$ $Fc(x, Gp(x)) \in St(Tc)$
      $\langle 6\rangle 1.$ $\forall j \in Nat : fc(x, Gp(x), j) \in Tc \cup \{Undef\}$
        BY $\langle 3\rangle 2$, $\langle 3\rangle 3$, $\langle 4\rangle 4$, $H\_BFunType$
      $\langle 6\rangle$ QED
        BY $\langle 3\rangle 3$, $\langle 6\rangle 1$ DEF $Fc$, $St$
    $\langle 5\rangle 2.$ $c[I0] \in St(Tc)$
      BY $\langle 2\rangle 0$, $\langle 3\rangle 3$ DEF $TypeInv$
    $\langle 5\rangle 3.$ $\forall j \in It(x) : Fc(x, Gp(x))[j] = c[I0][j]$
      $\langle 6\rangle 0.$ SUFFICES ASSUME NEW $j \in It(x)$
                         PROVE $fc(x, Gp(x), j) = fc(x, p[I0], j)$
        BY $\langle 2\rangle 0$, $\langle 3\rangle 3$, $\langle 4\rangle 3$ DEF $CInv$, $Fc$
      $\langle 6\rangle 1.$ $eqs(Gp(x), p[I0], deps(x, Dep\_pc, j))$
        $\langle 7\rangle 1.$ $deps(x, Dep\_pc, j) \subseteq It(x)$
          BY $\langle 3\rangle 2$, $\langle 3\rangle 3$, $H\_Type$ DEF $deps$, $It$
        $\langle 7\rangle 2.$ $wrts(p[I0], deps(x, Dep\_pc, j))$
          BY $\langle 4\rangle 3$, $\langle 7\rangle 1$ DEF $wrts$
        $\langle 7\rangle 3.$ $\forall k \in deps(x, Dep\_pc, j) :$
                $wrt(p[I0][k]) \land Gp(x)[k] = p[I0][k]$
          BY $\langle 4\rangle 4$, $\langle 7\rangle 1$, $\langle 7\rangle 2$ DEF $wrts$
        $\langle 7\rangle$ QED
          BY $\langle 7\rangle 3$ DEF $eqs$
      $\langle 6\rangle 2.$ $Gp(x) \in St(Tp) \land p[I0] \in St(Tp)$
        BY $\langle 4\rangle 4$
      $\langle 6\rangle 3.$ $fc(x, Gp(x), j) = fc(x, p[I0], j)$
        BY $\langle 3\rangle 2$, $\langle 3\rangle 3$, $\langle 6\rangle 1$, $\langle 6\rangle 2$, $H\_fcRelevance$
      $\langle 6\rangle$ QED
        BY $\langle 6\rangle 3$
    $\langle 5\rangle$ QED
      BY $\langle 5\rangle 1$, $\langle 5\rangle 2$, $\langle 5\rangle 3$ DEF $St$

  $\langle 4\rangle 6.$ $\land \forall j \in It(x) : f(j) = g(j)$
       $\land \forall j \in It(x) : f(j) \in D \land g(j) \in D$
    $\langle 5\rangle 1.$ $\forall j \in It(x) : f(j) = g(j)$
      $\langle 6\rangle 0.$ SUFFICES ASSUME NEW $j \in It(x)$
                         PROVE $fr(x, Fc(x, Gp(x)), j) = fr(x, c[I0], j)$
        BY $\langle 3\rangle 3$ DEF $Fr$
      $\langle 6\rangle 1.$ $eqs(Fc(x, Gp(x)), c[I0], deps(x, Dep\_cr, j))$
        $\langle 7\rangle 1.$ $deps(x, Dep\_cr, j) \subseteq It(x)$
          BY $\langle 3\rangle 2$, $\langle 3\rangle 3$, $H\_Type$ DEF $deps$, $It$
        $\langle 7\rangle 2.$ $wrts(c[I0], deps(x, Dep\_cr, j))$
          BY $\langle 4\rangle 3$, $\langle 7\rangle 1$ DEF $wrts$
        $\langle 7\rangle 3.$ $\forall k \in deps(x, Dep\_cr, j) :$
                $wrt(c[I0][k]) \land Fc(x, Gp(x))[k] = c[I0][k]$
          BY $\langle 4\rangle 5$, $\langle 7\rangle 1$, $\langle 7\rangle 2$ DEF $wrts$
          $\langle 7\rangle$ QED
            BY $\langle 7\rangle 3$  DEF $eqs$
        $\langle 6\rangle 2.\ Fc(x,\ Gp(x)) \in St(Tc) \wedge c[I0] \in St(Tc)$
          BY $\langle 4\rangle 5$
        $\langle 6\rangle 3.\ fr(x,\ Fc(x,\ Gp(x)),\ j) = fr(x,\ c[I0],\ j)$
          BY $\langle 3\rangle 2$, $\langle 3\rangle 3$, $\langle 6\rangle 1$, $\langle 6\rangle 2$, $H\_frRelevance$
        $\langle 6\rangle$ QED
          BY $\langle 6\rangle 3$
      $\langle 5\rangle 2.\ \forall j \in It(x) : f(j) \in D$
        $\langle 6\rangle 1.\ \forall j \in It(x) : deps(x,\ Dep\_cr,\ j) \subseteq It(x)$
          BY $\langle 3\rangle 2$, $\langle 3\rangle 3$, $H\_Type$ DEF $deps$, $It$
        $\langle 6\rangle 2.\ \forall j \in It(x) : fr(x,\ Fc(x,\ Gp(x)),\ j) \in D$
          BY $\langle 3\rangle 2$, $\langle 4\rangle 3$, $\langle 4\rangle 5$, $\langle 6\rangle 1$, $H\_BFunWD$ DEF $wrts$
        $\langle 6\rangle$ QED
          BY $\langle 3\rangle 3$, $\langle 6\rangle 2$  DEF $Fr$, $St$
      $\langle 5\rangle 3.\ \forall j \in It(x) : g(j) \in D$
        $\langle 6\rangle 1.\ \forall j \in It(x) : deps(x,\ Dep\_cr,\ j) \subseteq It(x)$
          BY $\langle 3\rangle 2$, $\langle 3\rangle 3$, $H\_Type$ DEF $deps$, $It$
        $\langle 6\rangle 2.\ \forall j \in It(x) : fr(x,\ c[I0],\ j) \in D$
          BY $\langle 3\rangle 2$, $\langle 4\rangle 3$, $\langle 4\rangle 5$, $\langle 6\rangle 1$, $H\_BFunWD$ DEF $wrts$
        $\langle 6\rangle$ QED
          BY $\langle 6\rangle 2$
      $\langle 5\rangle$ QED
        BY $\langle 5\rangle 1$, $\langle 5\rangle 2$, $\langle 5\rangle 3$

    $\langle 4\rangle 7.\ i = n$
      $\langle 5\rangle 0.$ SUFFICES ASSUME $i \neq n$
                     PROVE  FALSE
        OBVIOUS
      $\langle 5\rangle 1.\ \forall j \in It(x) \setminus \{i\} : red(I0,\ j)$
        BY $\langle 4\rangle 2$
      $\langle 5\rangle 2.\ red(I0,\ n)$
        BY $\langle 3\rangle 2$, $\langle 3\rangle 3$, $\langle 5\rangle 0$, $\langle 5\rangle 1$, $H\_Type$ DEF $It$
      $\langle 5\rangle 3.\ \forall j \in It(x) : j < n \Rightarrow red(I0,\ j)$
        BY $\langle 2\rangle 0$, $\langle 3\rangle 2$, $\langle 3\rangle 3$, $\langle 5\rangle 2$, $H\_Type$ DEF $RInv1$, $It$
      $\langle 5\rangle 4.\ i < n$
        BY $\langle 3\rangle 2$, $\langle 5\rangle 0$, $H\_Type$ DEF $It$
      $\langle 5\rangle 5.\ red(I0,\ i)$
        BY $\langle 5\rangle 0$, $\langle 5\rangle 1$, $\langle 5\rangle 3$, $\langle 5\rangle 4$  DEF $It$
      $\langle 5\rangle 6.\ \neg red(I0,\ i)$
        BY $\langle 4\rangle 2$
      $\langle 5\rangle$ QED
        BY $\langle 5\rangle 5$, $\langle 5\rangle 6$

    $\langle 4\rangle$ DEFINE $Q(j)$      $\triangleq$ $red(I0,\ j)$
                 $IfQg(j)$ $\triangleq$ IF $Q(j)$ THEN $g(j)$ ELSE  $id$
    $\langle 4\rangle$ HIDE  DEF $Q$, $f$, $g$, $m$, $n$

    $\langle 4\rangle 8.\ M!BigOp(m,\ n,\ f) = Op(M!BigOp(m,\ n-1,\ f),\ f(n))$
      $\langle 5\rangle 1.\ m \in Nat \wedge n \in Nat$
        BY $\langle 3\rangle 4$
      $\langle 5\rangle 2.\ m \leq n$
        BY $\langle 3\rangle 2$, $H\_Type$ DEF $It$, $m$, $n$
```

```
  $\langle 5\rangle 3.\ \forall j \in m \,..\, n : f(j) \in D$
    BY $\langle 4\rangle 6$ DEF $It$, $m$, $n$
  $\langle 5\rangle 4.\ MT!BigOp(m, n, f) = Op(MT!BigOp(m, n-1, f), f(n))$
    BY $\langle 5\rangle 1$, $\langle 5\rangle 2$, $\langle 5\rangle 3$, $H\_Mon$, $MT!SplitLast$, $Isa$
  $\langle 5\rangle$ QED
    BY $\langle 5\rangle 4$, $H\_MeqMT$

$\langle 4\rangle 9.\ M!BigOp(m, n-1, f) = M!BigOpP(m, n-1, Q, f)$
  $\langle 5\rangle$A.CASE $m = n$
    $\langle 6\rangle 1.\ m \in Int \land n-1 \in Int$
      BY $\langle 3\rangle 3$, $\langle 3\rangle 4$
    $\langle 6\rangle 2.\ m > n-1$
      BY $\langle 5\rangle$A, $\langle 6\rangle 1$
    $\langle 6\rangle 3.\ MT!BigOp(m, n-1, f) = id$
      BY $\langle 6\rangle 1$, $\langle 6\rangle 2$, $H\_Mon$, $H\_MeqMT$ DEF $MT!BigOp$
    $\langle 6\rangle 4.\ MT!BigOpP(m, n-1, Q, f) = id$
      BY $\langle 6\rangle 1$, $\langle 6\rangle 2$, $H\_Mon$, $H\_MeqMT$, $MT!EmptyIntvAssumpP$
    $\langle 6\rangle$ QED
      BY $\langle 6\rangle 3$, $\langle 6\rangle 4$, $H\_MeqMT$
  $\langle 5\rangle$B.CASE $m < n$
    $\langle 6\rangle 1.\ m \in Nat \land n-1 \in Nat$
      BY $\langle 3\rangle 4$, $\langle 5\rangle$B
    $\langle 6\rangle 2.\ \forall j \in m \,..\, n-1 : Q(j) \in$ BOOLEAN
      BY $\langle 2\rangle 0$, $\langle 3\rangle 2$, $\langle 3\rangle 3$, $\langle 6\rangle 1$ DEF $TypeInv$, $Q$
    $\langle 6\rangle 3.\ \forall j \in \{k \in m \,..\, n-1 : Q(k)\} : f(j) \in D$
      BY $\langle 3\rangle 4$, $\langle 4\rangle 6$ DEF $It$, $m$, $n$
    $\langle 6\rangle 4.\ \forall j \in m \,..\, n-1 : Q(j)$
      $\langle 7\rangle 1.\ \forall j \in It(x) : j < n \Rightarrow red(I0, j)$
        BY $\langle 4\rangle 2$, $\langle 4\rangle 7$
      $\langle 7\rangle$ QED
        BY $\langle 7\rangle 1$, $\langle 4\rangle 7$ DEF $Q$, $It$, $m$, $n$
    $\langle 6\rangle 5.\ MT!BigOp(m, n-1, f) = MT!BigOpP(m, n-1, Q, f)$
      BY $\langle 6\rangle 1$, $\langle 6\rangle 2$, $\langle 6\rangle 3$, $\langle 6\rangle 4$, $H\_Mon$, $MT!TruePredicate$, $Isa$
    $\langle 6\rangle$ QED
      BY $\langle 6\rangle 5$, $H\_MeqMT$
  $\langle 5\rangle$ QED
    BY $\langle 3\rangle 3$, $\langle 3\rangle 4$, $\langle 5\rangle$A, $\langle 5\rangle$B

$\langle 4\rangle 10.\ \land M!BigOpP(m, n-1, Q, f) = M!BigOpP(m, n-1, Q, g)$
        $\land f(n) = g(n)$
  $\langle 5\rangle 1.\ M!BigOpP(m, n-1, Q, f) = M!BigOpP(m, n-1, Q, g)$
    $\langle 6\rangle$A.CASE $m = n$
      $\langle 7\rangle 1.\ m \in Int \land n-1 \in Int$
        BY $\langle 3\rangle 3$, $\langle 3\rangle 4$
      $\langle 7\rangle 2.\ m > n-1$
        BY $\langle 6\rangle$A, $\langle 7\rangle 1$
      $\langle 7\rangle 3.\ MT!BigOpP(m, n-1, Q, f) = id$
        BY $\langle 7\rangle 1$, $\langle 7\rangle 2$, $H\_Mon$, $H\_MeqMT$, $MT!EmptyIntvAssumpP$
      $\langle 7\rangle 4.\ MT!BigOpP(m, n-1, Q, g) = id$
        BY $\langle 7\rangle 1$, $\langle 7\rangle 2$, $H\_Mon$, $H\_MeqMT$, $MT!EmptyIntvAssumpP$
      $\langle 7\rangle$ QED
        BY $\langle 7\rangle 3$, $\langle 7\rangle 4$, $H\_MeqMT$
        $\langle 6\rangle$B.CASE $m < n$
          $\langle 7\rangle 1.\ m \in Nat \land n - 1 \in Nat$
            BY $\langle 3\rangle 4$, $\langle 6\rangle$B
          $\langle 7\rangle 2.\ \forall j \in m \,..\, n-1 : Q(j) \in \text{BOOLEAN}$
            BY $\langle 2\rangle 0$, $\langle 3\rangle 2$, $\langle 3\rangle 3$, $\langle 7\rangle 1$  DEF $TypeInv$, $Q$
          $\langle 7\rangle 3.\ \land \forall j \in \{j \in m \,..\, n-1 : Q(j)\} : f(j) \in D$
                $\land \forall j \in \{j \in m \,..\, n-1 : Q(j)\} : g(j) \in D$
            BY $\langle 3\rangle 2$, $\langle 4\rangle 6$, $\langle 4\rangle 7$  DEF $Q$, $It$, $m$, $n$
          $\langle 7\rangle 4.\ \forall j \in \{j \in m \,..\, n-1 : Q(j)\} : f(j) = g(j)$
            BY $\langle 3\rangle 3$, $\langle 4\rangle 6$, $\langle 4\rangle 7$  DEF $Q$, $It$, $m$, $n$
          $\langle 7\rangle 5.\ MT!BigOpP(m, n-1, Q, f) = MT!BigOpP(m, n-1, Q, g)$
            BY $\langle 7\rangle 1$, $\langle 7\rangle 2$, $\langle 7\rangle 3$, $\langle 7\rangle 4$, $H\_Mon$, $MT!FunctionEqP$, $IsaM$("blast")
          $\langle 7\rangle$ QED
            BY $\langle 7\rangle 5$, $H\_MeqMT$
        $\langle 6\rangle$ QED
          BY $\langle 3\rangle 3$, $\langle 3\rangle 4$, $\langle 6\rangle$A, $\langle 6\rangle$B
      $\langle 5\rangle 2.\ f(n) = g(n)$
        BY $\langle 4\rangle 6$, $\langle 4\rangle 7$
      $\langle 5\rangle$ QED
        BY $\langle 5\rangle 1$, $\langle 5\rangle 2$

    $\langle 4\rangle 11.\ y = M!BigOpP(m, n-1, Q, g)$
      $\langle 5\rangle 1.\ \neg red(I0, n)$
        BY $\langle 4\rangle 2$, $\langle 4\rangle 7$
      $\langle 5\rangle$ QED
        BY $\langle 2\rangle 0$, $\langle 3\rangle 3$, $\langle 4\rangle 7$, $\langle 5\rangle 1$  DEF $RInv2$, $Q$, $g$, $m$, $n$

    $\langle 4\rangle$E1. $A(x) = M!BigOp(m, n, f)$                        BY  DEF $A$, $f$, $m$, $n$
    $\langle 4\rangle$E2.    $@ = Op(M!BigOp(m, n-1, f), f(n))$             BY $\langle 4\rangle 8$
    $\langle 4\rangle$E3.    $@ = Op(M!BigOpP(m, n-1, Q, f), f(n))$         BY $\langle 4\rangle 9$
    $\langle 4\rangle$E4.    $@ = Op(M!BigOpP(m, n-1, Q, g), g(n))$         BY $\langle 4\rangle 10$
    $\langle 4\rangle$E5.    $@ = Op(y, g(n))$                              BY $\langle 4\rangle 11$
    $\langle 4\rangle$E6.    $@ = y'$                                       BY $\langle 4\rangle 1$, $\langle 4\rangle 7$

    $\langle 4\rangle$ QED
      BY $\langle 4\rangle$E1, $\langle 4\rangle$E2, $\langle 4\rangle$E3, $\langle 4\rangle$E4, $\langle 4\rangle$E5, $\langle 4\rangle$E6

  $\langle 3\rangle$ QED
    BY $\langle 3\rangle 1$, $\langle 3\rangle$A, $\langle 3\rangle$B, $\langle 3\rangle$C
$\langle 2\rangle$B.CASE $Done$
  $\langle 3\rangle 0$. SUFFICES ASSUME UNCHANGED $\langle in, vs\rangle$,
                                  $end(I0)$
                    PROVE  $y' = A(x)$
    BY $\langle 2\rangle$B  DEF $Done$, $vs$, $A$, $M!BigOpP$, $M!BigOp$, $M!bigOp$, $end$
  $\langle 3\rangle 1.\ y = A(x)$
    BY $\langle 2\rangle 0$, $\langle 3\rangle 0$  DEF $Correctness$
  $\langle 3\rangle 2.\ y' = y$
    BY $\langle 3\rangle 0$  DEF $vs$
  $\langle 3\rangle$ QED
    BY $\langle 3\rangle 1$, $\langle 3\rangle 2$
$\langle 2\rangle$C.CASE UNCHANGED $\langle in, vs\rangle$
  $\langle 3\rangle 0$. SUFFICES ASSUME UNCHANGED $\langle in, vs\rangle$,
                                  $end(I0)$
```

```
                          PROVE  $y' = A(x)$
      BY ⟨2⟩C  DEF $vs$, $Correctness$, $A$, $M!BigOpP$, $M!BigOp$, $M!bigOp$, $end$
    ⟨3⟩1. $y = A(x)$
      BY ⟨2⟩0, ⟨3⟩0  DEF $Correctness$
    ⟨3⟩2. $y' = y$
      BY ⟨3⟩0  DEF $vs$
    ⟨3⟩ QED
      BY ⟨3⟩1, ⟨3⟩2
  ⟨2⟩ QED
    BY ⟨2⟩0, ⟨2⟩A, ⟨2⟩B, ⟨2⟩C  DEF $Next$
⟨1⟩ QED
  BY ⟨1⟩1, ⟨1⟩2, $Thm\_Inv$, $PTL$ DEF $Spec$
```

C.4 Basic PCR with left reducer: Refinement of a basic PCR

```
┌──────────────── MODULE PCR_ArLeft_Thms ────────────────┐

THEOREM Thm_Refinement ≜ Spec ⇒ PCR_A!Spec
⟨1⟩ DEFINE x ≜ X[I0]
           m ≜ lBnd(x)
           n ≜ uBnd(x)
⟨1⟩ USE DEF PCR_A!Index, PCR_A!Assig, PCR_A!red
⟨1⟩1. Init ⇒ PCR_A!Init
  BY H_Type DEF Init, PCR_A!Init, Undef, PCR_A!Undef, inS
⟨1⟩2. ∧ Inv
      ∧ [Next]_⟨in, vs⟩
      ⇒ [PCR_A!Next]_⟨inS, PCR_A!vs⟩
  ⟨2⟩0. SUFFICES ASSUME IndexInv, TypeInv,
                        [Next]_⟨in, vs⟩
                 PROVE  [PCR_A!Next]_PCR_A!vs
    BY DEF Inv, inS, PCR_A!vs
  ⟨2⟩A.CASE Step
    ⟨3⟩0. SUFFICES ASSUME ∃ i ∈ It(x) : ∨ P(I0, i)
                                        ∨ C(I0, i)
                                        ∨ R(I0, i)
                   PROVE [PCR_A!Next]_PCR_A!vs
      BY ⟨2⟩0, ⟨2⟩A DEF Step, IndexInv
    ⟨3⟩1. PICK i ∈ It(x) : ∨ P(I0, i)
                           ∨ C(I0, i)
                           ∨ R(I0, i)
      BY ⟨3⟩0
    ⟨3⟩2. x ∈ T
      BY ⟨2⟩0 DEF IndexInv, TypeInv, WDIndex, wrt
    ⟨3⟩3. ∧ I0 ∈ Seq(Nat)
          ∧ I0 ∈ WDIndex
          ∧ i ∈ Nat
          ∧ i ∈ m . . n
          ∧ It(x) ⊆ Nat
      BY ⟨2⟩0, ⟨3⟩2, H_Type DEF IndexInv, WDIndex, It
    ⟨3⟩4. m ∈ Nat ∧ n ∈ Nat
      BY ⟨2⟩0, ⟨3⟩2, H_Type DEF TypeInv, m, n
    ⟨3⟩5. ∧ I0 ∈ PCR_A!WDIndex
          ∧ i  ∈ PCR_A!It(x)
      BY ⟨3⟩3 DEF WDIndex, It, wrt, Undef, PCR_A!WDIndex,
           PCR_A!It, PCR_A!wrt, PCR_A!Undef, propS

    ⟨3⟩A.CASE P(I0, i)
      ⟨4⟩0. SUFFICES ASSUME P(I0, i)
                     PROVE  PCR_A!P(I0, i)
        BY ⟨2⟩0, ⟨3⟩5, ⟨3⟩A DEF P, inS, PCR_A!Next, PCR_A!Step
      ⟨4⟩ QED
        BY ⟨4⟩0 DEF P, wrt, wrts, deps, Undef, PCR_A!P,
             PCR_A!wrt, PCR_A!wrts, PCR_A!deps, PCR_A!Undef

    ⟨3⟩B.CASE C(I0, i)
      ⟨4⟩0. SUFFICES ASSUME C(I0, i)
                     PROVE  PCR_A!C(I0, i)
        BY ⟨2⟩0, ⟨3⟩5, ⟨3⟩B DEF C, inS, PCR_A!Next, PCR_A!Step
      ⟨4⟩ QED
        BY ⟨4⟩0  DEF C, wrt, wrts, deps, Undef, PCR_A!C,
            PCR_A!wrt, PCR_A!wrts, PCR_A!deps, PCR_A!Undef

    ⟨3⟩C.CASE R(I0, i)
      ⟨4⟩0. SUFFICES ASSUME R(I0, i)
                     PROVE  PCR_A!R(I0, i)
        BY ⟨2⟩0, ⟨3⟩5, ⟨3⟩C  DEF R, inS, PCR_A!Next, PCR_A!Step
      ⟨4⟩ QED
        BY ⟨4⟩0  DEF R, wrt, wrts, deps, Undef, PCR_A!R,
            PCR_A!wrt, PCR_A!wrts, PCR_A!deps, PCR_A!Undef

    ⟨3⟩ QED
      BY ⟨3⟩1, ⟨3⟩A, ⟨3⟩B, ⟨3⟩C
  ⟨2⟩B.CASE Done
    ⟨3⟩0. SUFFICES ASSUME UNCHANGED ⟨in, vs⟩
                   PROVE  UNCHANGED PCR_A!vs
      BY ⟨2⟩B  DEF Done, vs
    ⟨3⟩1. PCR_A!vs' = PCR_A!vs
      BY ⟨3⟩0  DEF PCR_A!vs, vs, end
    ⟨3⟩ QED
      BY ⟨3⟩1
  ⟨2⟩C.CASE UNCHANGED ⟨in, vs⟩
    ⟨3⟩0. SUFFICES ASSUME UNCHANGED ⟨in, vs⟩
                   PROVE  UNCHANGED PCR_A!vs
      BY ⟨2⟩C  DEF vs
    ⟨3⟩1. PCR_A!vs' = PCR_A!vs
      BY ⟨3⟩0  DEF PCR_A!vs, vs, end
    ⟨3⟩ QED
      BY ⟨3⟩1
  ⟨2⟩ QED
    BY ⟨2⟩0, ⟨2⟩A, ⟨2⟩B, ⟨2⟩C  DEF Next
⟨1⟩ QED
  BY ⟨1⟩1, ⟨1⟩2, Thm_Inv, PTL DEF Spec, PCR_A!Spec
```

C.5 Composition through consumer: Refinement of a basic PCR

```
┌──────────────────── MODULE PCR_A_c_B_Thms ────────────────────┐

THEOREM Thm_Refinement ≜ Spec ⇒ PCR_A!Spec
⟨1⟩ DEFINE x1    ≜ X1[I0]
           m1    ≜ lBnd1(x1)
           n1    ≜ uBnd1(x1)
           Q1(i) ≜ prop1(i)
⟨1⟩ USE  DEF PCR_A!Index, PCR_A!Assig, PCR_A!red
⟨1⟩1. Init ⇒ PCR_A!Init
  ⟨2⟩ SUFFICES ASSUME Init
               PROVE  PCR_A!Init
    OBVIOUS
  ⟨2⟩1. ∧ x1 ∈ T ∧ pre(x1)
        ∧ X1  = [I ∈ Seq(Nat) ↦ IF I = I0 THEN x1 ELSE  Undef]
        ∧ p1  = [I ∈ Seq(Nat) ↦ [i ∈ Nat ↦ Undef]]
        ∧ c1  = [I ∈ Seq(Nat) ↦ [i ∈ Nat ↦ Undef]]
        ∧ rs1 = [I ∈ Seq(Nat) ↦ [i ∈ Nat ↦ FALSE]]
        ∧ r1  = [I ∈ Seq(Nat) ↦ id1]
    BY H_TypeA DEF Init, InitA
  ⟨2⟩ QED
    BY ⟨2⟩1, H_UndefRestrict DEF PCR_A!Init, inS
⟨1⟩2. ∧ InvA
      ∧ CorrectnessB
      ∧ [Next]_⟨in, vs1, vs2⟩
      ⇒ [PCR_A!Next]_⟨inS, PCR_A!vs⟩
  ⟨2⟩0. SUFFICES ASSUME IndexInvA, TypeInvA, CInv1A,
                        CorrectnessB,
                        [Next]_⟨in, vs1, vs2⟩
                 PROVE  [PCR_A!Next]_PCR_A!vs
    BY  DEF InvA, inS, PCR_A!vs
  ⟨2⟩A.CASE StepA
    ⟨3⟩0. SUFFICES ASSUME ∃ i ∈ ItA(x1) : ∨ P1(I0, i)
                                         ∨ C1ini(I0, i)
                                         ∨ C1end(I0, i)
                                         ∨ R1(I0, i)
                   PROVE [PCR_A!Next]_PCR_A!vs
      BY ⟨2⟩0, ⟨2⟩A  DEF StepA, IndexInvA
    ⟨3⟩1. PICK i ∈ ItA(x1) : ∨ P1(I0, i)
                             ∨ C1ini(I0, i)
                             ∨ C1end(I0, i)
                             ∨ R1(I0, i)
      BY ⟨3⟩0
    ⟨3⟩2. x1 ∈ T
      BY ⟨2⟩0  DEF IndexInvA, TypeInvA, WDIndexA, wrt
    ⟨3⟩3. ∧ I0 ∈ Seq(Nat)
          ∧ I0 ∈ WDIndexA
          ∧ i ∈ Nat
          ∧ i ∈ {k ∈ m1 .. n1 : Q1(k)}
          ∧ ItA(x1) ⊆ Nat
      BY ⟨2⟩0, ⟨3⟩2, H_TypeA DEF IndexInvA, WDIndexA, ItA
    ⟨3⟩4. m1 ∈ Nat ∧ n1 ∈ Nat
      BY ⟨2⟩0, ⟨3⟩2, H_TypeA DEF TypeInvA, m1, n1
```

```
⟨3⟩5. ∧ I0 ∈ PCR_A!WDIndex
      ∧ i  ∈ PCR_A!It(x1)
  BY ⟨3⟩3, H_UndefRestrict DEF WDIndexA, ItA, wrt,
     PCR_A!WDIndex, PCR_A!It, PCR_A!wrt

⟨3⟩A.CASE P1(I0, i)
  ⟨4⟩0. SUFFICES ASSUME P1(I0, i)
                 PROVE  PCR_A!P(I0, i)
    BY ⟨2⟩0, ⟨3⟩5, ⟨3⟩A  DEF P1, inS, PCR_A!Next, PCR_A!Step
  ⟨4⟩ QED
    BY ⟨4⟩0, H_UndefRestrict DEF P1, wrt, wrts, depsA,
       PCR_A!P, PCR_A!wrt, PCR_A!wrts, PCR_A!deps

⟨3⟩B.CASE C1ini(I0, i)
  ⟨4⟩0. SUFFICES ASSUME C1ini(I0, i)
                 PROVE  UNCHANGED PCR_A!vs
    BY ⟨2⟩0, ⟨3⟩5, ⟨3⟩B  DEF C1ini, PCR_A!Next, PCR_A!Step
  ⟨4⟩1. UNCHANGED ⟨X1, p1, c1, r1, rs1⟩
    BY ⟨4⟩0  DEF C1ini
  ⟨4⟩ QED
    BY ⟨4⟩1  DEF PCR_A!vs

⟨3⟩C.CASE C1end(I0, i)
  ⟨4⟩0. SUFFICES ASSUME C1end(I0, i)
                 PROVE  PCR_A!C(I0, i)
    BY ⟨2⟩0, ⟨3⟩5, ⟨3⟩C  DEF C1end, inS, PCR_A!Next, PCR_A!Step
  ⟨4⟩1. ∧ ¬wrt(c1[I0][i])
        ∧ wrt(X2[I0 ∘ ⟨i⟩])
        ∧ endB(I0 ∘ ⟨i⟩)
        ∧ c1′ = [c1 EXCEPT ![I0][i] = r2[I0 ∘ ⟨i⟩]]
        ∧ UNCHANGED ⟨X1, p1, r1, rs1, X2⟩
    BY ⟨4⟩0  DEF C1end
  ⟨4⟩2. wrts(p1[I0], depsA(x1, Dep_pc1, i))
    BY ⟨2⟩0, ⟨4⟩1  DEF CInv1A
  ⟨4⟩3. PICK vp ∈ StA(Tp1) :
                ∧ eqs(vp, p1[I0], depsA(x1, Dep_pc1, i))
                ∧ X2[I0 ∘ ⟨i⟩] = ⟨x1, vp, i⟩
    BY ⟨2⟩0, ⟨4⟩1  DEF CInv1A
  ⟨4⟩4. p1[I0] ∈ StA(Tp1)
    BY ⟨2⟩0, ⟨3⟩3  DEF TypeInvA, StA
  ⟨4⟩5. I0 ∘ ⟨i⟩ ∈ WDIndexB
    BY ⟨3⟩3, ⟨4⟩1  DEF WDIndexB

  ⟨4⟩E1. c1[I0][i]′  = r2[I0 ∘ ⟨i⟩]      BY ⟨2⟩0, ⟨3⟩3, ⟨4⟩1 DEF TypeInvA, StA
  ⟨4⟩E2.            @ = B(X2[I0 ∘ ⟨i⟩]) BY ⟨2⟩0, ⟨4⟩1, ⟨4⟩5 DEF CorrectnessB
  ⟨4⟩E3.            @ = B(⟨x1, vp, i⟩)      BY ⟨4⟩3
  ⟨4⟩E4.            @ = fcS(x1, vp, i)       BY  DEF fcS
  ⟨4⟩E5.            @ = fcS(x1, p1[I0], i)   BY ⟨3⟩2, ⟨4⟩3, ⟨4⟩4, H_fcSRelevance

  ⟨4⟩6. c1[I0][i]′ = fcS(x1, p1[I0], i)
    BY ⟨4⟩1, ⟨4⟩E1, ⟨4⟩E2, ⟨4⟩E3, ⟨4⟩E4, ⟨4⟩E5
  ⟨4⟩7. c1′ = [c1 EXCEPT ![I0][i] = fcS(x1, p1[I0], i)]
    BY ⟨4⟩1, ⟨4⟩6
```

```
      $\langle 4\rangle 8.\ \land \neg wrt(c1[I0][i])$
            $\land\ wrts(p1[I0],\ depsA(x1,\ Dep\_pc1,\ i))$
            $\land\ c1' = [c1$ EXCEPT $![I0][i] = fcS(x1,\ p1[I0],\ i)]$
            $\land$ UNCHANGED $\langle X1,\ p1,\ r1,\ rs1\rangle$
        BY $\langle 4\rangle 1,\ \langle 4\rangle 2,\ \langle 4\rangle 7$

      $\langle 4\rangle$ QED
        BY $\langle 4\rangle 8,\ H\_UndefRestrict$ DEF $PCR\_A!C,\ wrt,\ wrts,\ depsA,$
            $PCR\_A!wrt,\ PCR\_A!wrts,\ PCR\_A!deps$

    $\langle 3\rangle$D.CASE $R1(I0,\ i)$
      $\langle 4\rangle 0.$ SUFFICES ASSUME $R1(I0,\ i)$
                         PROVE  $PCR\_A!R(I0,\ i)$
        BY $\langle 2\rangle 0,\ \langle 3\rangle 5,\ \langle 3\rangle$D  DEF $R1,\ inS,\ PCR\_A!Next,\ PCR\_A!Step$
      $\langle 4\rangle$ QED
        BY $\langle 4\rangle 0,\ H\_UndefRestrict$ DEF $R1,\ wrt,\ wrts,\ depsA,$
            $PCR\_A!R,\ PCR\_A!wrt,\ PCR\_A!wrts,\ PCR\_A!deps$

    $\langle 3\rangle$ QED
      BY $\langle 3\rangle 1,\ \langle 3\rangle$A, $\langle 3\rangle$B, $\langle 3\rangle$C, $\langle 3\rangle$D

  $\langle 2\rangle$B.CASE $StepB$
    $\langle 3\rangle 0.$ SUFFICES ASSUME $\exists\, I \in WDIndexB :$
                                    $\exists\, i \in ItB(X2[I]) :\ \lor P2(I,\ i)$
                                                        $\lor C2(I,\ i)$
                                                        $\lor R2(I,\ i)$
                   PROVE UNCHANGED $PCR\_A!vs$
      BY $\langle 2\rangle 0,\ \langle 2\rangle$B  DEF $StepB$
    $\langle 3\rangle 1.\ \langle in,\ vs1\rangle' = \langle in,\ vs1\rangle$
      BY $\langle 2\rangle$B  DEF $StepB$
    $\langle 3\rangle 2.\ PCR\_A!vs' = PCR\_A!vs$
      BY $\langle 3\rangle 1$  DEF $vs1,\ PCR\_A!vs,\ wrt$
    $\langle 3\rangle$ QED
      BY $\langle 3\rangle 2$

  $\langle 2\rangle$C.CASE $Done$
    $\langle 3\rangle 0.$ SUFFICES ASSUME UNCHANGED $\langle in,\ vs1,\ vs2\rangle$
                   PROVE  UNCHANGED $PCR\_A!vs$
      BY $\langle 2\rangle$C  DEF $Done,\ vs1,\ vs2$
    $\langle 3\rangle 1.\ PCR\_A!vs' = PCR\_A!vs$
      BY $\langle 3\rangle 0$  DEF $PCR\_A!vs,\ vs1,\ vs2,\ wrt$
    $\langle 3\rangle$ QED
      BY $\langle 3\rangle 1$
  $\langle 2\rangle$D.CASE UNCHANGED $\langle in,\ vs1,\ vs2\rangle$
    $\langle 3\rangle 0.$ SUFFICES ASSUME UNCHANGED $\langle in,\ vs1,\ vs2\rangle$
                   PROVE  UNCHANGED $PCR\_A!vs$
      BY $\langle 2\rangle$D  DEF $vs1,\ vs2$
    $\langle 3\rangle 1.\ PCR\_A!vs' = PCR\_A!vs$
      BY $\langle 3\rangle 0$  DEF $PCR\_A!vs,\ vs1,\ vs2,\ wrt$
    $\langle 3\rangle$ QED
      BY $\langle 3\rangle 1$
  $\langle 2\rangle$ QED
    BY $\langle 2\rangle 0,\ \langle 2\rangle$A, $\langle 2\rangle$B, $\langle 2\rangle$C, $\langle 2\rangle$D  DEF $Next$
```

```
⟨1⟩ QED
  BY ⟨1⟩1, ⟨1⟩2, Thm_Inv1, Thm_CorrectnessB, PTL DEF Spec, PCR_A!Spec
```

Appendix D

Specification of concrete PCRs

D.1 PCR FibPrimes1

MODULE *PCR_FibPrimes1*

PCR FibPrimes1.

```
----------------------------------------------------------------
fun fibs(p,i)  = if i <= 2 then 1 else p[-1] + p[-2]
fun isPrime(p) = n > 1 and not (some (\m. divides(m,n)) [2..n-1])
fun count(r,c) = r + if c then 1 else 0

dep p(i-1) -> p(i)
dep p(i-2) -> p(i)

lbnd FibPrimes1 = \N. 1
ubnd FibPrimes1 = \N. x

PCR FibPrimes1(N)

  par
    p = produce fibs p
    c = consume isPrime p
    r = reduce count 0 c
----------------------------------------------------------------
```

EXTENDS *Naturals*, *Sequences*, *ArithUtils*, *TLC*

Concrete elements of *FibPrimes1*

$T \triangleq Nat$
$Tp \triangleq Nat$
$Tc \triangleq \text{BOOLEAN}$
$D \triangleq Nat$

$Dep\_pp \triangleq \langle \{1, 2\}, \{\} \rangle$
$Dep\_pc \triangleq \langle \{\}, \{\} \rangle$
$Dep\_cr \triangleq \langle \{\}, \{\} \rangle$

$lBnd(N) \triangleq 1$
$uBnd(N) \triangleq N$
$prop(i) \triangleq \text{TRUE}$

$fibs(p, i) \triangleq \text{IF } i \leq 2 \text{ THEN } 1 \text{ ELSE } p[i-1] + p[i-2]$
$isPrime(p) \triangleq p > 1 \land \neg \exists\, m \in 2 \,..\, (p - 1) : Divides(m, p)$
$toNat(c) \triangleq \text{IF } c \text{ THEN } 1 \text{ ELSE } 0$

$id \triangleq 0$
$Op(x, y) \triangleq x + y$

FibPrimes1 is a concrete instance of the abstract model *PCR_A*

VARIABLES *in*, *X*, *p*, *c*, *r*, *rs*

$I0 \triangleq \langle\rangle$
$pre(x) \triangleq \text{TRUE}$

$fp(x, vp, i) \triangleq fibs(vp, i)$
$fc(x, vp, i) \triangleq isPrime(vp[i])$
$fr(x, vc, i) \triangleq toNat(vc[i])$
$gp(x, i) \triangleq fibonacci[i]$

INSTANCE *PCR_A_Thms*

Most axioms can be promoted to ordinary mathematical lemmas. These are proved in module *PCR_FibPrimes1_Lems*.

$Lem\_Type \triangleq H\_Type$
$Lem\_BFunWD \triangleq H\_BFunWD$
$Lem\_fpRelevance \triangleq H\_fpRelevance$
$Lem\_fcRelevance \triangleq H\_fcRelevance$
$Lem\_frRelevance \triangleq H\_frRelevance$
$Lem\_Algebra \triangleq H\_Algebra$

The invariant axiom for the producer can be proved alongside the other basic invariants. This is done in module *PCR_FibPrimes1_Thms*.

$ProdEqInv \triangleq$
$\quad \forall\, i \in It(X[I0]) :$
$\quad\quad wrt(p[I0][i]) \Rightarrow fp(X[I0],\, p[I0],\, i) = gp(X[I0],\, i)$

The ordinary lemmas re-stated as invariant properties for model checking.

They are relativized to $I0$ for a more tractable verification, but even so it is only feasible for very small bounds.

$TypeCheck \triangleq$
$\quad \land\, lBnd(X[I0]) \in Nat$
$\quad \land\, uBnd(X[I0]) \in Nat$
$\quad \land\, prop(X[I0]) \in \text{BOOLEAN}$
$\quad \land\, pre(X[I0]) \in \text{BOOLEAN}$
$\quad \land\, Dep\_pp \in (\text{SUBSET } (Nat \setminus \{0\})) \times (\text{SUBSET } \{\})$
$\quad \land\, Dep\_pc \in (\text{SUBSET } (Nat \setminus \{0\})) \times (\text{SUBSET } (Nat \setminus \{0\}))$
$\quad \land\, Dep\_cr \in (\text{SUBSET } (Nat \setminus \{0\})) \times (\text{SUBSET } (Nat \setminus \{0\}))$

$BFunWDCheck \triangleq$
$\quad \forall\, i \in It(X[I0]) :$
$\quad\quad \land\, gp(X[I0],\, i) \in Tp$
$\quad\quad \land\, wrts(p[I0],\, deps(X[I0],\, Dep\_pp,\, i) \setminus \{i\}) \Rightarrow fp(X[I0],\, p[I0],\, i) \in Tp$
$\quad\quad \land\, wrts(p[I0],\, deps(X[I0],\, Dep\_pc,\, i)) \Rightarrow fc(X[I0],\, p[I0],\, i) \in Tc$
$\quad\quad \land\, wrts(c[I0],\, deps(X[I0],\, Dep\_cr,\, i)) \Rightarrow fr(X[I0],\, c[I0],\, i) \in D$

$fpRelevanceCheck \triangleq$
$\quad \forall\, i \in It(X[I0]),\, vp1 \in St(Tp),\, vp2 \in St(Tp) :$
$\quad\quad eqs(vp1,\, vp2,\, deps(X[I0],\, Dep\_pp,\, i) \setminus \{i\}) \Rightarrow fp(X[I0],\, vp1,\, i) = fp(X[I0],\, vp2,\, i)$

$fcRelevanceCheck \triangleq$
$\quad \forall\, i \in It(X[I0]),\, vp1 \in St(Tp),\, vp2 \in St(Tp) :$
$\quad\quad eqs(vp1,\, vp2,\, deps(X[I0],\, Dep\_pc,\, i)) \Rightarrow fc(X[I0],\, vp1,\, i) = fc(X[I0],\, vp2,\, i)$

$frRelevanceCheck \triangleq$
$\quad \forall\, i \in It(X[I0]),\, vc1 \in St(Tc),\, vc2 \in St(Tc) :$
$\quad\quad eqs(vc1,\, vc2,\, deps(X[I0],\, Dep\_cr,\, i)) \Rightarrow fr(X[I0],\, vc1,\, i) = fr(X[I0],\, vc2,\, i)$

$AlgebraCheck \triangleq$
$\quad \land\, \forall\, x,\, y,\, z \in D : Op(Op(x,\, y),\, z) = Op(x,\, Op(y,\, z))$
$\quad \land\, \forall\, x \in D : Op(x,\, id) = x \land Op(id,\, x) = x$
$\quad \land\, \forall\, x,\, y \in D : Op(x,\, y) = Op(y,\, x)$

$LemmaCheck \triangleq \land\, TypeCheck$
$\quad\quad\quad\quad \land\, BFunWDCheck$
$\quad\quad\quad\quad \land\, fpRelevanceCheck$

```
                    ∧ fcRelevanceCheck
                    ∧ frRelevanceCheck
                    ∧ AlgebraCheck
```

Alternative correctness

```
CountFibPrimes(N) ≜ LET fibSeq ≜ [i ∈ 1 . . N ↦ fibonacci[i]]
                    IN  Len(SelectSeq(fibSeq, LAMBDA f : IsPrime(f)))

CorrectnessAlt ≜ end(I0) ⇒ r[I0] = CountFibPrimes(X[I0])
```

D.2 PCR IsPrime2

MODULE *PCR_IsPrime2*

PCR IsPrime2.

```
------------------------------------------------------------------
fun divs(i)     = i
fun notDiv(N,p) = N > 1 and (p > 1 implies not divides(p,N))

lbnd IsPrime2 = \N. 0
ubnd IsPrime2 = \N. sqrt(N)
prop IsPrime2 = \i. i <= 2 or odd(i)

PCR IsPrime2(N)
  par
    p = produce divs p
    c = consume notDiv N p
    r = reduce and true c
------------------------------------------------------------------
```

EXTENDS *Naturals*, *Sequences*, *ArithUtils*, *TLC*

Concrete elements of *IsPrime2*

$T \triangleq Nat$
$Tp \triangleq Nat$
$Tc \triangleq \text{BOOLEAN}$
$D \triangleq \text{BOOLEAN}$

$Dep\_pp \triangleq \langle \{\}, \{\} \rangle$
$Dep\_pc \triangleq \langle \{\}, \{\} \rangle$
$Dep\_cr \triangleq \langle \{\}, \{\} \rangle$

$lBnd(N) \triangleq 0$
$uBnd(N) \triangleq Sqrt(N)$
$prop(i) \triangleq i \leq 2 \lor Odd(i)$

$divs(i) \triangleq i$
$notDiv(N, p) \triangleq N > 1 \land (p > 1 \Rightarrow \neg Divides(p, N))$

$id \triangleq \text{TRUE}$
$Op(x, y) \triangleq x \land y$

IsPrime2 is a concrete instance of the abstract model *PCR_A*

VARIABLES *in*, *X*, *p*, *c*, *r*, *rs*

$I0 \triangleq \langle\rangle$
$pre(x) \triangleq \text{TRUE}$

$fp(x, vp, i) \triangleq divs(i)$
$fc(x, vp, i) \triangleq notDiv(x, vp[i])$
$fr(x, vc, i) \triangleq vc[i]$
$gp(x, i) \triangleq fp(x, p, i)$

INSTANCE *PCR_A*

Alternative correctness

$CorrectnessAlt \triangleq end(I0) \Rightarrow r[I0] = IsPrime(X[I0])$

D.3 PCR FibPrimes2

MODULE *PCR_FibPrimes2*

PCR FibPrimes2.

```
  ----------------------------------------------------------------
  fun fibs(p,i)  = if i <= 2 then 1 else p[-1] + p[-2]
  fun count(r,c) = r + if c then 1 else 0

  dep p(i-1) -> p(i)
  dep p(i-2) -> p(i)

  lbnd FibPrimes1 = \N. 1
  ubnd FibPrimes1 = \N. x

  PCR FibPrimes2(N)
    par
      p1 = produce fibs p1
      c1 = consume IsPrime p1
      r1 = reduce count 0 c1

  fun divs(i)     = i
  fun notDiv(F,p) = F > 1 and (p > 1 implies not divides(p,F))

  lbnd IsPrime2 = \F. 0
  ubnd IsPrime2 = \F. sqrt(F)
  prop IsPrime2 = \i. i <= 2 or odd(i)

  PCR IsPrime2(F)
    par
      p2 = produce divs p2
      c2 = consume notDiv F p2
      r2 = reduce and true c2
----------------------------------------------------------------
```

EXTENDS *Naturals*, *Sequences*, *ArithUtils*, *TLC*

Concrete elements of *FibPrimes2*

$T \triangleq Nat$
$Tp1 \triangleq Nat$
$Tc1 \triangleq$ BOOLEAN
$D1 \triangleq Nat$

$Dep\_pp1 \triangleq \langle\{1, 2\}, \{\}\rangle$
$Dep\_pc1 \triangleq \langle\{\}, \{\}\rangle$
$Dep\_cr1 \triangleq \langle\{\}, \{\}\rangle$

$lBnd1(N) \triangleq 1$
$uBnd1(N) \triangleq N$
$prop1(i) \triangleq$ TRUE

$fibs(p, i) \triangleq$ IF $i \leq 2$ THEN 1 ELSE $p[i-1] + p[i-2]$
$toNat(c) \triangleq$ IF c THEN 1 ELSE 0

$id1 \triangleq 0$
$Op1(x, y) \triangleq x + y$

Concrete elements of *IsPrime2*

$Tp2 \triangleq Nat$
$Tc2 \triangleq$ BOOLEAN
$D2 \triangleq$ BOOLEAN

$Dep\_pp2 \triangleq \langle\{\}, \{\}\rangle$

$Dep\_pc2 \triangleq \langle \{\}, \{\} \rangle$
$Dep\_cr2 \triangleq \langle \{\}, \{\} \rangle$

$lBnd2(F) \triangleq 0$
$uBnd2(F) \triangleq Sqrt(F)$
$prop2(i) \triangleq i \leq 2 \lor Odd(i)$

$divs(i) \triangleq i$
$notDiv(F, p) \triangleq F > 1 \land (p > 1 \Rightarrow \neg Divides(p, F))$

$id2 \triangleq \text{TRUE}$
$Op2(x, y) \triangleq x \land y$

*FibPrimes*2 is a concrete instance of the abstract model *PCR_A_c_B*

$\text{VARIABLES } in, X1, p1, c1, r1, rs1,$
$\qquad X2, p2, c2, r2, rs2$

$I0 \triangleq \langle\rangle$
$pre(x1) \triangleq \text{TRUE}$

$fp1(x1, vp, i) \triangleq fibs(vp, i)$
$fr1(x1, vc, i) \triangleq toNat(vc[i])$
$gp1(x1, i) \triangleq fibonacci[i]$

$\_uBnd2(x2) \triangleq \text{LET } F \triangleq x2[2][x2[3]] \text{ IN } \quad uBnd2(F)$
$fp2(x2, vp, i) \triangleq divs(i)$
$fc2(x2, vp, i) \triangleq \text{LET } F \triangleq x2[2][x2[3]] \text{ IN } \quad notDiv(F, vp[i])$
$fr2(x2, vc, i) \triangleq vc[i]$
$gp2(x2, i) \triangleq fp2(x2, p2, i)$

$\text{INSTANCE } PCR\_A\_c\_B \text{ WITH } uBnd2 \leftarrow \_uBnd2$

Alternative correctness

$CountFibPrimes(N) \triangleq \text{LET } fibSeq \triangleq [i \in 1 \mathrel{..} N \mapsto fibonacci[i]]$
$\qquad \text{IN } \quad Len(SelectSeq(fibSeq, \text{LAMBDA } f : IsPrime(f)))$

$CorrectnessAlt \triangleq endA(I0) \Rightarrow r1[I0] = CountFibPrimes(X1[I0])$

D.4 PCR MergeSort1

MODULE *PCR_MergeSort1*

PCR MergeSort1.

```
-----------------------------------------------------------------
fun div(L)    = let m = floor(len(L) / 2)
                in [L[1..m], L[m+1..len(L)]]
fun isBase(p) = len(p) <= 1
fun base(p)   = p

PCR MergeSort1(L)
  par
    p = produce iterDiv L
    c = consume subproblem L p
    r = reduce merge [] c
-----------------------------------------------------------------
```

EXTENDS *Naturals*, *Sequences*, *SeqUtils*, *ArithUtils*, *TLC*

Concrete elements of *MergeSort1*

CONSTANT *Elem*

$T \triangleq Seq(Elem)$
$D \triangleq Seq(Elem)$

$Dep\_pc \triangleq \langle \{\}, \{\} \rangle$
$Dep\_cr \triangleq \langle \{\}, \{\} \rangle$

$div(L) \triangleq \text{LET } mid \triangleq Len(L) \div 2$
$\quad \text{IN } \langle SubSeq(L, 1, mid), SubSeq(L, mid + 1, Len(L)) \rangle$
$isBase(p) \triangleq Len(p) \leq 1$
$base(p) \triangleq p$

$id \triangleq \langle\rangle$
$Op(x, y) \triangleq x \uplus y$

MergeSort1 is a concrete instance of the abstract model *PCR_DC*

VARIABLES *in*, *X*, *p*, *c*, *r*, *rs*

$I0 \triangleq \langle\rangle$
$pre(x) \triangleq \text{TRUE}$

$\_base(x, vp, i) \triangleq base(vp[i])$
$\_isBase(x, vp, i) \triangleq isBase(vp[i])$
$fr(x, vc, i) \triangleq vc[i]$

INSTANCE *PCR_DC* WITH $base \leftarrow \_base$, $isBase \leftarrow \_isBase$

Alternative correctness

$CorrectnessAlt \triangleq end(I0) \Rightarrow r[I0] = SortSeq(X[I0], <)$

D.5 PCR Merge

```
──────────────────── MODULE PCR_Merge ────────────────────
  PCR Merge.

    ----------------------------------------------------------------
    fun binarySearch(L,e) = ...
    fun div(L1,L2) = [(L11,L21), (L21,L22)]    // ensure len(L1) >= len(L2)
                     where m   = floor(len(L1) / 2)
                           L11 = L1[1..m]
                           L12 = L1[m+1..len(L1)]
                           k   = binarySearch(L2, L12[1])
                           L21 = L2[1..k]
                           L22 = L2[k+1..len(L2)]
    fun isBase(p) = len(p[1]) <= 1 and len(p[2]) <= 1
    fun base(p)   = merge(p[1],p[2])

    PCR Merge(L1, L2)
      par
        p = produce iterDiv L1 L2
        c = consume subproblem L1 L2 p
        r = reduce ++ [] c
  ----------------------------------------------------------------
EXTENDS Naturals, Sequences, SeqUtils, ArithUtils, TLC
───────────────────────────────────────────────────────────

  Concrete elements of Merge
CONSTANT Elem

T ≜ Seq(Elem)
D ≜ Seq(Elem)

Dep_pc ≜ ⟨{}, {}⟩
Dep_cr ≜ ⟨{}, {}⟩

binarySearch(seq, e) ≜
  LET f[s ∈ Seq(Elem)] ≜
    IF s = ⟨⟩ THEN 0
     ELSE LET m ≜ (Len(s) + 1) ÷ 2
           IN  CASE e = s[m] → m
                  □  e < s[m] → f[SubSeq(s, 1, m − 1)]
                  □  e > s[m] → LET pv ≜ f[SubSeq(s, m + 1, Len(s))]
                                IN  IF pv > 0 THEN pv + m + 1 ELSE  m − pv
  IN  f[seq]

div(_L1, _L2) ≜                                       Make sure len(L1) ≥ len(L2)
  LET L1 ≜ IF Len(_L1) ≥ Len(_L2) THEN _L1 ELSE _L2
      L2 ≜ IF Len(_L1) ≥ Len(_L2) THEN _L2 ELSE _L1
  IN  LET m    ≜ Len(L1) ÷ 2                           a. Split L1 in halves L11 and L12
          L11  ≜ SubSeq(L1, 1, m)
          L12  ≜ SubSeq(L1, m + 1, Len(L1))
          k    ≜ binarySearch(L2, L12[1])              b. Search position of L12[1] in L2
          L21  ≜ CASE k = 0         → ⟨⟩               c. Split L2 in that position
                      □k > Len(L2) → L2
                      □OTHER       → SubSeq(L2, 1, k)
           L22 ≜ CASE k = 0         → L2
                      □k > Len(L2) → ⟨⟩
                      □OTHER       → SubSeq(L2, k + 1, Len(L2))
        IN   ⟨⟨L11, L21⟩, ⟨L12, L22⟩⟩          Produce two sub-merges

isBase(p) ≜ Len(p[1]) ≤ 1 ∧ Len(p[2]) ≤ 1
base(p)   ≜ p[1] ⊎ p[2]

id        ≜ ⟨⟩
Op(x, y)  ≜ x ∘ y

├──────────────────────────────────────────────────────────────┤

  Merge is a concrete instance of the abstract model PCR_DCrLeft
VARIABLES in, X, p, c, r, rs

I0      ≜ ⟨⟩
pre(x)  ≜ IsOrdered(x[1]) ∧ IsOrdered(x[2])

_div(x)              ≜ div(x[1], x[2])
_base(x, vp, i)      ≜ base(vp[i])
_isBase(x, vp, i)    ≜ isBase(vp[i])
fr(x, vc, i)         ≜ vc[i]

INSTANCE PCR_DCrLeft WITH div ← _div, base ← _base, isBase ← _isBase

├──────────────────────────────────────────────────────────────┤

  Alternative correctness
CorrectnessAlt ≜ end(I0) ⇒ r[I0] = X[I0][1] ⊎ X[I0][2]

└──────────────────────────────────────────────────────────────┘
```

D.6 PCR MergeSort2

MODULE *PCR_MergeSort2*

PCR MergeSort2.

```
----------------------------------------------------------------
fun div1(L)    = let m = floor(len(L) / 2)
                 in [L[1..m], L[m+1..len(L)]]
fun isBase1(p) = len(p) <= 1
fun base1(p)   = p

PCR MergeSort2(L)
  par
    p1 = produce iterDiv1 L
    c1 = consume subproblem1 L p1
    r1 = reduce Merge [] c1

fun binarySearch(L,e) = ...
fun div2(L1,L2) = [(L11,L21), (L21,L22)]    // ensure len(L1) >= len(L2)
                  where m   = floor(len(L1) / 2)
                        L11 = L1[1..m]
                        L12 = L1[m+1..len(L1)]
                        k   = binarySearch(L2, L12[1])
                        L21 = L2[1..k]
                        L22 = L2[k+1..len(L2)]
fun isBase2(p) = len(p[1]) <= 1 and len(p[2]) <= 1
fun base2(p)   = merge(p[1],p[2])

PCR Merge(L1, L2)
  par
    p2 = produce iterDiv2 L1 L2
    c2 = consume subproblem2 L1 L2 p2
    r2 = reduce ++ [] c2
----------------------------------------------------------------
```

EXTENDS *Naturals*, *Sequences*, *SeqUtils*, *ArithUtils*, *TLC*

Concrete elements of *MergeSort2*

CONSTANT *Elem*

$T \triangleq Seq(Elem)$
$D \triangleq Seq(Elem)$

$Dep\_pc1 \triangleq \langle \{\}, \{\} \rangle$
$Dep\_cr1 \triangleq \langle \{\}, \{\} \rangle$

$div1(L) \triangleq$ LET $mid \triangleq Len(L) \div 2$
  IN $\langle SubSeq(L, 1, mid), SubSeq(L, mid+1, Len(L)) \rangle$
$isBase1(p) \triangleq Len(p) \leq 1$
$base1(p) \triangleq p$

$id \triangleq \langle\rangle$

Concrete elements of Merge

$Dep\_pc2 \triangleq \langle \{\}, \{\} \rangle$
$Dep\_cr2 \triangleq \langle \{\}, \{\} \rangle$

```
binarySearch(seq, e) ≜
  LET f[s ∈ Seq(Elem)] ≜
    IF s = ⟨⟩ THEN 0
     ELSE LET m ≜ (Len(s) + 1) ÷ 2
          IN CASE e = s[m] → m
```

```
                         □  e < s[m] → f[SubSeq(s, 1, m − 1)]
                         □  e > s[m] → LET pv ≜ f[SubSeq(s, m + 1, Len(s))]
                                        IN  IF pv > 0 THEN pv + m + 1 ELSE  m − pv
  IN   f[seq]

div2(_L1, _L2) ≜                                          Make sure len(L1) ≥ len(L2)
  LET L1 ≜ IF Len(_L1) ≥ Len(_L2) THEN _L1 ELSE  _L2
      L2 ≜ IF Len(_L1) ≥ Len(_L2) THEN _L2 ELSE  _L1
  IN  LET m     ≜ Len(L1) ÷ 2                              a. Split L1 in halves L11 and L12
          L11   ≜ SubSeq(L1, 1, m)
          L12   ≜ SubSeq(L1, m + 1, Len(L1))
          k     ≜ binarySearch(L2, L12[1])                 b. Search position of L12[1] in L2
          L21   ≜ CASE k = 0          → ⟨⟩                 c. Split L2 in that position
                       □k > Len(L2) → L2
                       □OTHER       → SubSeq(L2, 1, k)
           L22  ≜ CASE k = 0          → L2
                       □k > Len(L2) → ⟨⟩
                       □OTHER       → SubSeq(L2, k + 1, Len(L2))
      IN   ⟨⟨L11, L21⟩, ⟨L12, L22⟩⟩                    Produce two sub-merges

isBase2(p) ≜ Len(p[1]) ≤ 1 ∧ Len(p[2]) ≤ 1
base2(p)   ≜ p[1] ⊎ p[2]

Op2(x, y) ≜ x ∘ y

|------------------------------------------------------------------------------|

  MergeSort2 is a concrete instance of the abstract model PCR_DC_r_DCrLeft

VARIABLES in, X1, p1, c1, r1, rs1,
              X2, p2, c2, r2, rs2

I0      ≜ ⟨⟩
pre(x) ≜ TRUE

_div1(x)              ≜ div1(x)
_base1(x, vp, i)      ≜ base1(vp[i])
_isBase1(x, vp, i)    ≜ isBase1(vp[i])
fr1(x, vc, i)         ≜ vc[i]

_div2(x)              ≜ div2(x[1], x[2])
_base2(x, vp, i)      ≜ base2(vp[i])
_isBase2(x, vp, i)    ≜ isBase2(vp[i])
fr2(x, vc, i)         ≜ vc[i]

INSTANCE PCR_DC_r_DCrLeft
   WITH div1 ← _div1, base1 ← _base1, isBase1 ← _isBase1,
        div2 ← _div2, base2 ← _base2, isBase2 ← _isBase2

|------------------------------------------------------------------------------|

  Alternative correctness

CorrectnessAlt ≜ endA(I0) ⇒ r1[I0] = SortSeq(X1[I0], < )

└──────────────────────────────────────────────────────────────────────────────┘
```

D.7 PCR NQueensDC

```
──────────────────── MODULE PCR_NQueensDC ────────────────────
  PCR NQueensDC.

    ----------------------------------------------------------------
    fun validPos(C,i,j)   = ...
    fun addQInRow(C,i)    = ...
    fun canAddQInRow(C,i) = ...
    fun canAddQueens(C)   = ...
    fun complete(C)       = all (\j. j != 0) C

    fun div(C)    = [addQInRow(C,i) | 1 <= i <= len(C), canAddQInRow(C,i)]
    fun isBase(p) = complete(p) or not canAddQueens(p)
    fun base(p)   = if complete(p) then { p } else {}

    PCR NQueensDC(C)
      par
        p = produce iterDiv C
        c = consume subproblem C p
        r = reduce union {} c
  ----------------------------------------------------------------
EXTENDS Naturals, Sequences, SeqUtils, ArithUtils, TLC
├──────────────────────────────────────────────────────────────┤
  Concrete elements of NQueensDC
Config ≜ Seq(Nat)
T ≜ SUBSET Config
D ≜ SUBSET Config

Dep_pc ≜ ⟨{}, {}⟩
Dep_cr ≜ ⟨{}, {}⟩

validPos(C, i, j) ≜
  ∧ C[i] = 0
  ∧ ∀ k ∈ DOMAIN C : C[k] ≠ j
  ∧ ∀ k ∈ DOMAIN C : C[k] ≠ 0 ⇒ abs(C[k] − j) ≠ abs(k − i)

addQInRow(C, i)    ≜ LET j ≜ CHOOSE j ∈ DOMAIN C : validPos(C, i, j)
                       IN [C EXCEPT ![i] = j]
canAddQInRow(C, i) ≜ ∃ j ∈ DOMAIN C : validPos(C, i, j)
canAddQueens(C)    ≜ ∀ i ∈ DOMAIN C : C[i] = 0 ⇒ canAddQInRow(C, i)
complete(C)        ≜ ∀ i ∈ DOMAIN C : C[i] ≠ 0

div(C)    ≜ Map(LAMBDA i : addQInRow(C, i),
                 SelectSeq(1 . . Len(C), LAMBDA i : canAddQInRow(C, i)))
isBase(p) ≜ complete(p) ∨ ¬canAddQueens(p)
base(p)   ≜ IF complete(p) THEN {p} ELSE {}

id       ≜ {}
Op(x, y) ≜ x ∪ y
├──────────────────────────────────────────────────────────────┤
  NQueensDC is a concrete instance of the abstract model PCR_DC
VARIABLES in, X, p, c, r, rs

I0     ≜ ⟨⟩
pre(C) ≜ ∀ i ∈ DOMAIN C : C[i] = 0
```

```
_base(x, vp, i)   ≜ base(vp[i])
_isBase(x, vp, i) ≜ isBase(vp[i])
fr(x, vc, i)      ≜ vc[i]

INSTANCE PCR_DC WITH base ← _base, isBase ← _isBase

├──────────────────────────────────────────────────────────────┤

  Alternative correctness
Solutions(x) ≜ CASE Len(x) = 0      → {}
                  □ Len(x) = 1      → {⟨1⟩}
                  □ Len(x) ∈ 2 . . 3 → {}
                  □ Len(x) = 4      → {⟨3, 1, 4, 2⟩,
                                        ⟨2, 4, 1, 3⟩}
                  □ Len(x) = 5      → {⟨1, 3, 5, 2, 4⟩,
                                        ⟨1, 4, 2, 5, 3⟩,
                                        ⟨2, 4, 1, 3, 5⟩,
                                        ⟨2, 5, 3, 1, 4⟩,
                                        ⟨3, 1, 4, 2, 5⟩,
                                        ⟨3, 5, 2, 4, 1⟩,
                                        ⟨4, 1, 3, 5, 2⟩,
                                        ⟨4, 2, 5, 3, 1⟩,
                                        ⟨5, 2, 4, 1, 3⟩,
                                        ⟨5, 3, 1, 4, 2⟩}
                  □ Len(x) = 6      → {⟨2, 4, 6, 1, 3, 5⟩,
                                        ⟨3, 6, 2, 5, 1, 4⟩,
                                        ⟨4, 1, 5, 2, 6, 3⟩,
                                        ⟨5, 3, 1, 6, 4, 2⟩}
                  □ Len(x) > 6      → "ask google..."

CorrectnessAlt ≜ end(I0) ⇒ r[I0] = Solutions(X[I0])

└──────────────────────────────────────────────────────────────┘
```

D.8 PCR NQueensIT

```
──────────────── MODULE PCR_NQueensIT ────────────────
```

PCR NQueensIT.

```
----------------------------------------------------------------
fun cnd(s,j) = j > 1 and s == s[-1]

PCR NQueensIT(C)
  c1 = iterate cnd NQueensITstep {C}

fun validPos(C,i,j)   = ...
fun addQInRow(C,i)    = ...
fun canAddQInRow(C,i) = ...
fun complete(C)       = all (\j. j != 0) C

fun elem(CS,i) = enum(CS)[i]
fun div(p)     = {addQInRow(p,i) | 1 <= i <= len(p), canAddQInRow(p,i)}
fun extend(p)  = if complete(p) then {p} else div(p)

lbnd NQueensITstep = \CS. 1
ubnd NQueensITstep = \CS. #(CS)

PCR NQueensITstep(CS)
  par
    p  = produce elem CS
    c2 = consume extend p
    r  = reduce union {} c2
```
```
----------------------------------------------------------------
```

EXTENDS *Naturals*, *Sequences*, *SequencesExt*, *FiniteSets*, *ArithUtils*, *TLC*

```
──────────────────────────────────────────────────────
```

Concrete elements of *NQueensIT*

```
Config ≜ Seq(Nat)
T      ≜ Config
Tp1    ≜ Config
D1     ≜ SUBSET Config

Dep_pp1 ≜ ⟨{}, {}⟩
Dep_pc1 ≜ ⟨{}, {}⟩
Dep_cr1 ≜ ⟨{}, {}⟩

lBnd1(C)  ≜ 0
uBnd1(C)  ≜ 0
prop1(i)  ≜ TRUE

v0(C)      ≜ {C}
cnd(s, j)  ≜ j > 1 ∧ s[j] = s[j − 1]

id1        ≜ CHOOSE x ∈ D1 : TRUE
Op1(x, y)  ≜ y
```

Concrete elements of *NQueensITstep*

```
Tp2 ≜ Config
Tc2 ≜ SUBSET Config
D2  ≜ SUBSET Config

Dep_pp2 ≜ ⟨{}, {}⟩
Dep_pc2 ≜ ⟨{}, {}⟩
Dep_cr2 ≜ ⟨{}, {}⟩

lBnd2(CS)  ≜ 1
```

```
uBnd2(CS) ≜ Cardinality(CS)
prop2(i)  ≜ TRUE

validPos(C, i, j) ≜
   ∧ C[i] = 0
   ∧ ∀ k  ∈ DOMAIN C : C[k] ≠ j
   ∧ ∀ k  ∈ DOMAIN C : C[k] ≠ 0 ⇒ abs(C[k] − j) ≠ abs(k − i)

addQInRow(C, i)    ≜ LET j ≜ CHOOSE j ∈ DOMAIN C : validPos(C, i, j)
                      IN  [C EXCEPT ![i] = j]
canAddQInRow(C, i) ≜ ∃ j ∈ DOMAIN C : validPos(C, i, j)
complete(C)        ≜ ∀ i ∈ DOMAIN C : C[i] ≠ 0

elem(CS, i) ≜ SetToSeq(CS)[i]
div(p)      ≜ {addQInRow(p, i) : i ∈ {i ∈ 1 .. Len(p) : canAddQInRow(p, i)}}
extend(p)   ≜ IF complete(p) THEN {p} ELSE  div(p)

id2       ≜ {}
Op2(x, y) ≜ x ∪ y

├──────────────────────────────────────────────────────────────────────┤

  NQueensIT is a concrete instance of the abstract model PCR_A_it_B

VARIABLES in, X1, p1, c1, s, r1, rs1,
              X2, p2, c2, r2, rs2

I0      ≜ ⟨⟩
pre(C)  ≜ ∀ i ∈ DOMAIN C : C[i] = 0

fp1(x1, vp, i) ≜ x1
fr1(x1, vc, i) ≜ vc[i]
gp1(x1, i)     ≜ fp1(x1, p1, i)

_uBnd2(x2)     ≜ LET CS ≜ x2[1]IN   uBnd2(CS)
fp2(x2, vp, i) ≜ LET CS ≜ x2[1]IN   elem(CS, i)
fc2(x2, vp, i) ≜ extend(vp[i])
fr2(x2, vc, i) ≜ vc[i]
gp2(x2, i)     ≜ fp2(x2, p2, i)

INSTANCE PCR_A_it_B WITH uBnd2 ← _uBnd2

├──────────────────────────────────────────────────────────────────────┤

  Alternative correctness

Solutions(x) ≜ CASE Len(x) = 0      → {}
                  □ Len(x) = 1      → {⟨1⟩}
                  □ Len(x) ∈ 2 .. 3 → {}
                  □ Len(x) = 4      → {⟨3, 1, 4, 2⟩,
                                        ⟨2, 4, 1, 3⟩}
                  □ Len(x) = 5      → {⟨1, 3, 5, 2, 4⟩,
                                        ⟨1, 4, 2, 5, 3⟩,
                                        ⟨2, 4, 1, 3, 5⟩,
                                        ⟨2, 5, 3, 1, 4⟩,
                                        ⟨3, 1, 4, 2, 5⟩,
                                        ⟨3, 5, 2, 4, 1⟩,
                                          $\langle 4, 1, 3, 5, 2\rangle,$
                                          $\langle 4, 2, 5, 3, 1\rangle,$
                                          $\langle 5, 2, 4, 1, 3\rangle,$
                                          $\langle 5, 3, 1, 4, 2\rangle\}$
                    $\Box$  $Len(x) = 6$   $\rightarrow \{\langle 2, 4, 6, 1, 3, 5\rangle,$
                                          $\langle 3, 6, 2, 5, 1, 4\rangle,$
                                          $\langle 4, 1, 5, 2, 6, 3\rangle,$
                                          $\langle 5, 3, 1, 6, 4, 2\rangle\}$
                    $\Box$  $Len(x) > 6$   $\rightarrow$ "ask google..."

$CorrectnessAlt \triangleq endA(I0) \Rightarrow r1[I0] = Solutions(X1[I0])$
```

Appendix E

Model checking results

Model checking of concrete PCRs was powered by a clustered machine whose access and control was facilitated by the Jupyter kernel for TLA$^+$ [81]. The TLC version used was v2.14 of 10 July 2019 (rev: 0cae24f). Relevant characteristics of the mentioned machine are the following:

Model name:	Intel(R) Xeon(R) W-2195 CPU @ 2.30GHz
Core(s) per socket:	18
Thread(s) per core:	2
On-line CPU(s) list:	0-35
CPU max MHz:	4300.0000
CPU min MHz:	1000.0000

Table E.2 presents our results with a sub-table for each concrete PCR which is an instance of some abstract PCR model. TLC was always instructed to use 18 worker threads, the same number of physical cores on the machine, which makes most sense for a CPU bound activity like model checking. In general, there would be almost no improvement in using more worker threads than physical cores. Results are reported with respect to different *inputs* (or *inputs* sizes), informed by the first column. The other columns are divided on two kinds:

1. Information about the state graph. This includes a) the amount of *states*, which TLC reports as the "distinct states found" and b) the *depth*, which TLC reports as the "depth of the complete state graph search". The "distinct states" correspond with the cardinality of the set of reachable vertices of the state graph, and is typically less than the total number of states examined by TLC as part of model checking

[40]. An interesting feature of TLC is the option to automatically, after verification, export the state graph to a file and produce a visualization. For example, figure E.1 correspond to the state graph of PCR FibPrimes1 when $N = 2$. Notice that it has 13 vertices and height/depth 7, which are the same values reported in table E.2. However, this makes sense only for relatively small inputs, because the state graph could be too big to be rendered by any tool.

2. Time taken to verify interesting properties. In general, this encompasses *Correctness*, *Termination* and *Refinement* with respect to some high-level PCR model. When refinement is applicable, it is further separated into the *only safety* case and the complete case considering fairness, as both normally vary wildly on the computational effort required. Additionally, TLC is always by default checking for deadlock. When reported time have the form "$> t$" it means we aborted verification beyond that point.

For PCRs FibPrimes1, IsPrime2 and FibPrimes2 the result is reported with respect to different *inputs* (i.e. the value N), whereas for the rest the result is reported with respect to different *input sizes* (e.g. the length of the input list L). More precisely, for input list L:

- Input size $\#L = k$ represents all the possible combinations taking *exactly* k elements from the set $\{0, 1, 2, 3\}$.
- Input size $\#L \leq k$ represents all the possible combinations taking *at most* k elements from the set $\{0, 1, 2, 3\}$.

There are some special cases whorty of mention. Inputs for PCR IsPrime2 are conveniently grouped by ranges of the form $k_1 \leq N \leq k_2$ because the amount of states and depth on those ranges is the same and the time taken to verify each property does not vary too much for small inputs. Nevertheless, when there is a difference between extremes, we report the time for both extremes in the form $Time(k_1)$ - $Time(k_2)$. Any *input size* for PCR NQueensDC and NQueensIT represent exactly one input, as the initial input C for the NQueens problem is expected to be the empty configuration (also a list) modeling an empty board of some size.

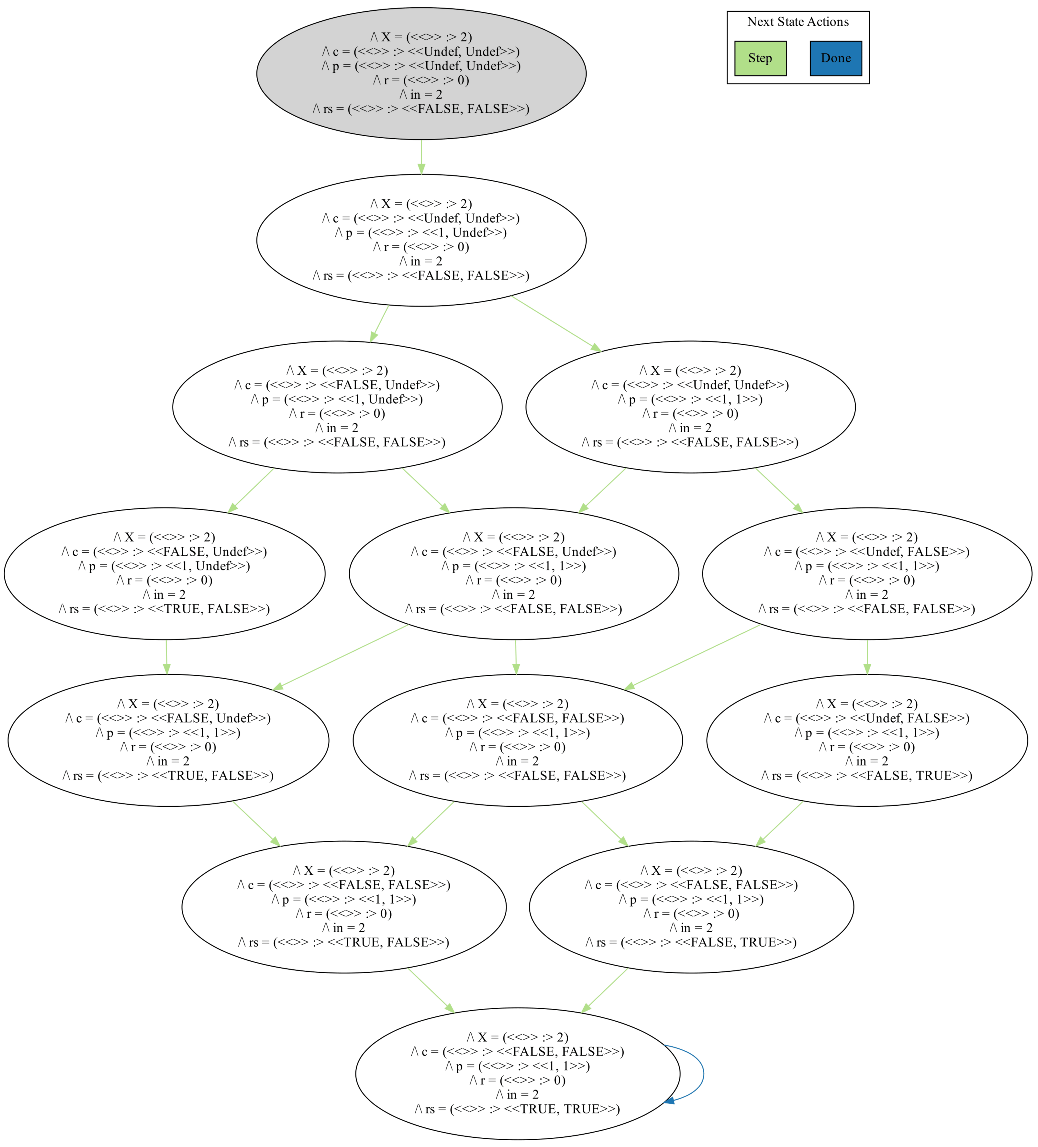

Next State Actions
Step
Done
∧ X = (<<>> :> 2)
∧ c = (<<>> :> <<Undef, Undef>>)
∧ p = (<<>> :> <<Undef, Undef>>)
∧ r = (<<>> :> 0)
∧ in = 2
∧ rs = (<<>> :> <<FALSE, FALSE>>)
∧ X = (<<>> :> 2)
∧ c = (<<>> :> <<Undef, Undef>>)
∧ p = (<<>> :> <<1, Undef>>)
∧ r = (<<>> :> 0)
∧ in = 2
∧ rs = (<<>> :> <<FALSE, FALSE>>)
∧ X = (<<>> :> 2)
∧ c = (<<>> :> <<FALSE, Undef>>)
∧ p = (<<>> :> <<1, Undef>>)
∧ r = (<<>> :> 0)
∧ in = 2
∧ rs = (<<>> :> <<FALSE, FALSE>>)
∧ X = (<<>> :> 2)
∧ c = (<<>> :> <<Undef, Undef>>)
∧ p = (<<>> :> <<1, 1>>)
∧ r = (<<>> :> 0)
∧ in = 2
∧ rs = (<<>> :> <<FALSE, FALSE>>)
∧ X = (<<>> :> 2)
∧ c = (<<>> :> <<FALSE, Undef>>)
∧ p = (<<>> :> <<1, Undef>>)
∧ r = (<<>> :> 0)
∧ in = 2
∧ rs = (<<>> :> <<TRUE, FALSE>>)
∧ X = (<<>> :> 2)
∧ c = (<<>> :> <<FALSE, Undef>>)
∧ p = (<<>> :> <<1, 1>>)
∧ r = (<<>> :> 0)
∧ in = 2
∧ rs = (<<>> :> <<FALSE, FALSE>>)
∧ X = (<<>> :> 2)
∧ c = (<<>> :> <<Undef, FALSE>>)
∧ p = (<<>> :> <<1, 1>>)
∧ r = (<<>> :> 0)
∧ in = 2
∧ rs = (<<>> :> <<FALSE, FALSE>>)
∧ X = (<<>> :> 2)
∧ c = (<<>> :> <<FALSE, Undef>>)
∧ p = (<<>> :> <<1, 1>>)
∧ r = (<<>> :> 0)
∧ in = 2
∧ rs = (<<>> :> <<TRUE, FALSE>>)
∧ X = (<<>> :> 2)
∧ c = (<<>> :> <<FALSE, FALSE>>)
∧ p = (<<>> :> <<1, 1>>)
∧ r = (<<>> :> 0)
∧ in = 2
∧ rs = (<<>> :> <<FALSE, FALSE>>)
∧ X = (<<>> :> 2)
∧ c = (<<>> :> <<Undef, FALSE>>)
∧ p = (<<>> :> <<1, 1>>)
∧ r = (<<>> :> 0)
∧ in = 2
∧ rs = (<<>> :> <<FALSE, TRUE>>)
∧ X = (<<>> :> 2)
∧ c = (<<>> :> <<FALSE, FALSE>>)
∧ p = (<<>> :> <<1, 1>>)
∧ r = (<<>> :> 0)
∧ in = 2
∧ rs = (<<>> :> <<TRUE, FALSE>>)
∧ X = (<<>> :> 2)
∧ c = (<<>> :> <<FALSE, FALSE>>)
∧ p = (<<>> :> <<1, 1>>)
∧ r = (<<>> :> 0)
∧ in = 2
∧ rs = (<<>> :> <<FALSE, TRUE>>)
∧ X = (<<>> :> 2)
∧ c = (<<>> :> <<FALSE, FALSE>>)
∧ p = (<<>> :> <<1, 1>>)
∧ r = (<<>> :> 0)
∧ in = 2
∧ rs = (<<>> :> <<TRUE, TRUE>>)

Figure E.1: State graph for PCR FibPrimes1 when $N = 2$ (in the graph this means $in = 2$). Image produced by TLC.

Table E.1 presents metrics we identified for the PCR models under some assumptions on the iteration space size. For PCR_A, it its assumed $|I_x| = n$. For $PCR\_A\_c\_B$, it its assumed $|I_x| = n$ and $|J_{x_1^i}| = m$ for all $i \in I_x$. Those metrics are:

- **# Interleavings:** the number of possible sequentializations between operations satisfying appropriate dependence constrains. As parallelism is modeled by interleaving, more parallelism means more interleaving which in turns means more states.

A serial producer reduces the number of interleavings as imposes more dependencies (i.e. it is less parallel). For example, PCR FibPrimes1 has 10 interleavings when $N = 2$, and looking at figure E.1, the specific interleaving of actions (where $I_0 = \langle\rangle$)

$$P(I_0, 1),\ P(I_0, 2),\ C(I_0, 2),\ R(I_0, 2),\ C(I_0, 1),\ R(I_0, 1)$$

correspond to the rightmost path from the initial state to the final state. The formulas are developed by elementary counting methods, and verified by algorithmic counting of linear/topological orderings.[1]

- **# States:** the number of reachable states. The formulas are inferred from the data obtained exercising TLC on different inputs. We don't have (yet) a formula for the *PCR_A_c_B* model, an exponential regression fit could be used but there is little data available to be reliable as a predictor.

- **Degree of parallelism:** this is the (relative) speedup concerning the total serial work and the parallel span. For simplicity, here it is assumed any operation have a constant cost. For the basic model *PCR_A*, it is evident that parallelism of a linear PCR scales linearly on n (the size of the iteration space, which depends on the input).[2] When the producer is serial, as $n \to \infty$ parallelism is bounded above by a constant. For the nested model *PCR_A_c_B* with only linear dependencies, it scales linearly with either n or m. However, if the outer PCR has a serial producer, good scalability depends now of m. In the worst case, if $m = 1$ then parallelism is bounded above by a constant.

The metrics of PCR models can be helpful to understand the complexity related to model verification. More specifically, the number of states can be used as a rough *a priori* estimation of verification time for large inputs. As an example, let us consider the data collected for PCR FibPrimes1 in table E.2. The largest input is $N = 11$, this requires

[1] In general, there is no nice closed formula to count linear orderings. For some special partial orders, the *hook length* formula (associated to standard Young tableaux) can be used but is not general enough for our setting [82]. According to [83], exact counting for this problem is #P-complete.

[2] Linear scalability is considered the holy grail of parallelism, as it is mathematically optimal and in practice occurrences of super-linear scalability are very rare. However, it should be noted that in our abstraction, by the interleaving assumption, reductions are considered as independent actions which is not true in reality. A more realistic quantification of parallelism should account for a serial or parallel reduction like we discussed in Chapter 2, and in either case parallelism will not scale linearly.

PCR Model	# Interleavings	# States	Degree of paralellism
PCR_A	$\frac{(3n)!}{6^n}$	4^n	$\frac{3n}{3} = n$
PCR_A with serial producer	$\frac{(3n)!}{n!\,6^n}$	$\frac{1}{2}(3^{n+1} - 1)$	$\frac{3n}{n+2}$
PCR_A_c_B	$\frac{(n(3m+4))!}{(3m+4)^{\underline{4}\,n}\,6^{mn}}$	-	$\frac{n(3m+4)}{7}$
PCR_A_c_B where A have serial producer	$\frac{(n(3m+4))!}{n!\,(3m+4)^n\,(3m+3)^{\underline{3}\,n}\,6^{mn}}$	-	$\frac{n(3m+4)}{n+6}$

Table E.1: Metrics for some PCR models. $x^{\underline{k}}$ denotes the falling factorial power: $x(x-1)(x-2)\dots(x-(k-1))$.

to construct a graph with 265,720 states for which checking correctness is still cheap as it takes only 12 seconds. But, what would be the situation for a much larger input like $N = 20$?. We can measure that TLC works at an average rate of 56,983 s/m[3] on $N = 20$ without checking any property (not even deadlock, i.e. just constructing the graph). Assuming work rate is maintained, to verify any property on $N = 20$ for which there are 5,230,176,601 states according to our formula, it would take *at least* 63 days 17h 44m. It is also interesting to compare the numbers with other machine. In our desktop PC (an Intel(R) Core i5-2467M CPU @ 1.60GHz with two physical cores) TLC works at 17,177 s/m, thus by the same reasoning it would take *at least* 211 days 10h 47m. In short, verification for $N = 20$ in the cluster machine would take two months, whereas in the desktop PC would take half year.

[3]At each minute, TLC reports it's runtime statistics, in particular the number of distinct states found per minute (ds/m). This number depends on specification, input and machine. According to our observations, this number tends to stabilize. However, several factors can affect the rate. For example, after some time, thermal throttling due to heat will lower CPU frequency rates affecting TLC performance.

PCR FibPrimes1 (PCR_A)						
Input	**States**	**Depth**	**Correctness**	**Termination**	**Refines** PCR_A1step **(only safety)**	**Refines** PCR_A1step
$N = 1$	4	4	< 1s	< 1s	< 1s	< 1s
$N = 2$	13	7	< 1s	< 1s	< 1s	< 1s
$N = 3$	40	10	< 1s	< 1s	< 1s	< 1s
$N = 4$	121	13	< 1s	< 1s	< 1s	< 1s
$N = 5$	364	16	< 1s	1s	< 1s	1s
$N = 6$	$1,093$	19	< 1s	1s	1s	1s
$N = 7$	$3,280$	22	1s	1s	2s	4s
$N = 8$	$9,841$	25	1s	1s	2s	10s
$N = 9$	$29,524$	28	2s	3s	4s	34s
$N = 10$	$88,573$	31	4s	6s	13s	5m50s
$N = 11$	$265,720$	34	12s	30s	5m24s	> 3h

PCR IsPrime2 (PCR_A)						
Input	**States**	**Depth**	**Correctness**	**Termination**	**Refines** PCR_A1step **(only safety)**	**Refines** PCR_A1step
$N = 0$	4	4	< 1s	< 1s	< 1s	< 1s
$1 \leq N \leq 3$	16	7	< 1s	< 1s	< 1s	< 1s
$4 \leq N \leq 8$	64	10	< 1s	< 1s	< 1s	< 1s
$9 \leq N \leq 24$	256	13	< 1s	1s	1s	1s
$25 \leq N \leq 48$	$1,024$	16	1s	1s	1s	2s
$49 \leq N \leq 80$	$4,096$	19	1s	2s	2s	7s - 8s
$81 \leq N \leq 120$	$16,384$	22	2s	6s - 8s	7s - 10s	31s - 46s
$121 \leq N \leq 168$	$65,536$	25	4s	31s - 42s	50s - 1m2s	4m31s - 5m24s
$169 \leq N \leq 224$	$262,144$	28	16s - 19s	3m28s - 4m25s	5m57s - 6m42s	33m39s - 35m20s

PCR FibPrimes2 ($PCR\_A\_c\_B$)						
Input	**States**	**Depth**	**CorrectnessA**	**TerminationA**	**Refines** PCR_A **(only safety)**	**Refines** PCR_A
$N = 1$	20	11	< 1s	< 1s	< 1s	< 1s
$N = 2$	723	21	1s	1s	1s	1s
$N = 3$	$33,388$	31	2s	8s	3s	8s
$N = 4$	$2,861,933$	41	51s	13m33s	3m37s	15m37s
$N = 5$	$> 637,161,621$	> 41	> 3h	-	-	-

Table E.2: Model checking results for concrete PCRs.

PCR MergeSort1 (*PCR_DC*)				
Input Size $Elem = \{0, 1, 2, 3\}$	**States**	**Depth**	**Correctness**	**Termination**
$\#L = 0$	16	7	< 1s	< 1s
$\#L = 1$	64	7	< 1s	< 1s
$\#L = 2$	256	7	< 1s	< 1s
$\#L = 3$	$5,120$	14	1s	3s
$\#L = 4$	$102,400$	21	7s	9s
$\#L = 5$	$1,720,320$	28	32s	2m59s
$\#L = 6$	$28,901,376$	35	9m27s	1h5m

PCR Merge (*PCR_DCrLeft*)						
Input Size $Elem = \{0, 1, 2, 3\}$	**States**	**Depth**	**Correctness**	**Termination**	**Refines** *PCR_DC* **(only safety)**	**Refines** *PCR_DC*
$\#L1 \leq 0,\ \#L2 \leq 0$	13	7	< 1s	< 1s	< 1s	< 1s
$\#L1 \leq 1,\ \#L2 \leq 1$	325	7	< 1s	1s	< 1s	2s
$\#L1 \leq 2,\ \#L2 \leq 2$	$5,850$	14	2s	5s	2s	6s
$\#L1 \leq 3,\ \#L2 \leq 3$	$336,830$	28	18s	1m14s	25s	2m
$\#L1 \leq 4,\ \#L2 \leq 4$	$11,013,223$	42	8m1s	1h26m	17m1s	2h38m

PCR MergeSort2 (*PCR_DC_r_DCrLeft*)						
Input Size $Elem = \{0, 1, 2, 3\}$	**States**	**Depth**	**CorrectnessA**	**TerminationA**	**Refines** *PCR_DC* **(only safety)**	**Refines** *PCR_DC*
$\#L = 0$	120	21	1s	1s	1s	1s
$\#L = 1$	484	21	1s	2s	1s	3s
$\#L = 2$	$1,936$	21	2s	6s	2s	6s
$\#L = 3$	$144,960$	42	21s	7m35s	25s	6m31s
$\#L = 4$	$5,879,536$	70	10m40s	> 3h	15m1s	> 3h

PCR NQueensDC (*PCR_DC*)				
Input Size	**States**	**Depth**	**Correctness**	**Termination**
$\#C = 1$	4	4	< 1s	< 1s
$\#C = 2$	16	7	< 1s	< 1s
$\#C = 3$	$8,000$	31	2s	6s
$\#C = 4$	$> 54,531,473$	> 31	> 3h	-

PCR NQueensIT (*PCR_A_it_B*)						
Input Size	**States**	**Depth**	**CorrectnessA**	**TerminationA**	**Refines** *PCR_A_it* **(only safety)**	**Refines** *PCR_A_it*
$\#C = 1$	15	15	< 1s	< 1s	< 1s	1s
$\#C = 2$	29	20	< 1s	< 1s	< 1s	1s
$\#C = 3$	$4,174$	43	6s	13s	38s	22m8s
$\#C = 4$	-	-	-	-	-	-

Table E.2: Model checking results for concrete PCRs (continued).

We end with some observations of our results:

- Assuming identical iteration space size, the number of interleavings and states for IsPrime2 grows faster than for FibPrimes1 as it has more parallelism. However, TLC can handle inputs for IsPrime2 much larger than for FibPrimes1 because the size of IsPrime2's iteration space is proportional to the square root of input N.

- As expected, checking nested PCR models is harder than checking their basic equivalents. This can be appreciated comparing FibPrimes1 vs FibPrimes2 and MergeSort1 vs MergeSort2.

- In general, checking termination (a liveness property) or a complete refinement (which involves liveness conditions, i.e. fairness) is notably harder than checking safety properties. This is in part because, when checking liveness properties, TLC uses Tarjan's algorithm to find strongly connected components on the state graph. It is an efficient but serial algorithm which does not benefit from the presence of more processing units.

- Checking the safety refinement of PCR_A1step in both FibPrimes1 and IsPrime2 seems to be harder than checking termination. However, for the other refinements the opposite is true. This may seem a bit strange considering that PCR_A1step is a very simple specification.

- Unfortunately, for the NQueens problem, we couldn't completely check Correctness for any of the proposed PCRs on input $\#C = 4$. In the case of NQueensDC, after some hours we aborted. In the case of NQueensIT, TLC reports the error:

  Attempted to construct a set with too many elements (> 1000000)

  Default maximum set size is one million, so we tuned TLC to lift the value up to one hundred million, but wasn't enough. The problem is that operator *SetToSeq*, used by NQueensIT, requires TLC to explicitly enumerate a large set of functions.[4] For this reason, this operator is overridden with a much efficient java implementation, but we don't know yet how to take advantage of the java overrides on the Jupyter TLA^{+} installation.

We haven't done any optimization effort on the specification itself, as the priority was to keep it readable and similar as possible to the divide-and-conquer version. As a last resort, we had to conform ourselves with random simulation on a desktop PC, which can be used also for inputs larger than $\#C = 4$. Hours of simulation didn't find any error while checking Correctness.
For comparison, we add that a specific TLA$^+$ solution in imperative style (i.e. not a declarative formula) for the NQueens problem, included as an example in the TLA Toolbox installation, can be model checked in reasonable time up to $\#C = 5$. Besides, for an imperative style solution, research shows that a symbolic model checker like ProB can't do better [84].

[4] It is defined as: $\mathit{SetToSeq}(S) \triangleq \text{CHOOSE } f \in [1..\mathit{Cardinality}(S) \rightarrow S] : \mathit{IsInjective}(f)$.