Contents

List of Scripts

List of Figures

Acronyms

API Application Programming Interface.

HAL Hardware Abstraction Layer.

IAL Interrupt Abstraction Layer.

IPC Inter-Process Communication.

MAL Memory Abstraction Layer.

MMU Memory Management Unit.

OS Operating System.

TCB Trusted Computing Base.

1.   CRIStAL laboratry

1.1   History

CRIStAL[1] (Research Center in Computer Science, Signal and Automatic Control of Lille), founded on the 1st of January 2015, is a laboratory of CNRS[2] (National Center for Scientific Research), Lille 1 university and Centrale Lille in partnership with Lille 3 University , Inria (French National Institute for computer science and applied mathematics) and Mines Telecom Institute. It is the result of the fusion of LAGIS[4] (Laboratory of Automatic Control, Computer Engineering and Signal) and LIFL[4] (Laboratory of Fundamental Computing of Lille) to federate their complementary competencies in information sciences. It is a member the interdisciplinary research institute IRCICA[5] (Research Institute on Software and Hardware Components for Advanced Information and Communication in Lille).

CRIStAL is located in Villeneuve d'Ascq city in Lille, the capital of the Hauts-de-France region and the prefecture of the Nord department and the fourth largest urban area in France after Paris, Lyon and Marseille.Chaired by *Prof.Olivier Colot*, it harbors about 430 personnel with exactly 228 permanents and more than 200 non-permanents. Permanent researchers are divided among more than 30 teams working on different themes and projects.

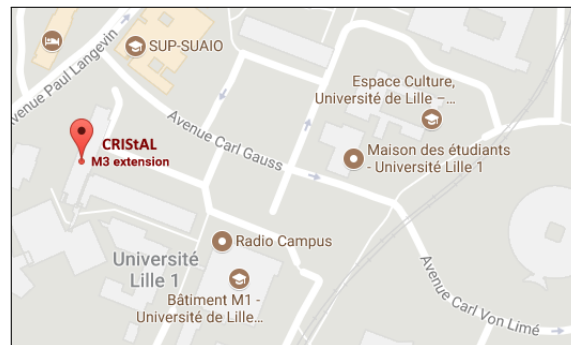Figure 1.1: CRIStAL laboratry Location

---

[1] "Centre de Recherche en Informatique, Signal et Automatique de Lille"
[2] "Centre national de la recherche scientifique"
[3] "Laboratoire d'Automatique, Génie Informatique et Signal"
[4] "Laboratoire d'Informatique Fondamentale de Lille"
[5] "Institut de recherche sur les composants logiciels et matériels pour l'information et la communication avancée de Lille"

1.2 Research activities

CRIStAL's research activities concern topics related to the major scientific and societal issues of the moment such as: BigData, software, computer imaging, human-machine interactions, robotics, control and supervision of large systems, intelligent embedded systems, bioinformatics... The laboratry is involved in the developpment of revolutionary platforms such as Pharo, a pure object oriented language and a powerful yet simple developpment environment used worldwide.

1.3 2XS team

The 2XS (eXtra Small, eXtra Safe) team is working on highly constrained embedded devices, precisely on designing software and hardware that are secure, safe and efficient. Research in this team is focused on defining new system architectures or new languages to allow fast development of reliable embedded software. The team addresses issues concerning memory footprint, energy consumption and security and takes profit from proficiencies in formal verification, hardware/software co-design and operating system architectures to tackle the aforementioned issues.

The team is lead by *Prof.Gilles Grimaud* and has 15 members[1] as well as several trainees and most of its current work mainly revolves around the PIP project and its possible applications.

Permanent	Temporary	Associated
• **Professor**	• **Postdoc**	• **Research scientist**
○ Gilles Grimaud (team leader)	○ Paolo Torrini	○ Vlad Rusu
• **Associate professors**	• **Phd students**	
○ Michaël Hauspie	○ Christophe Bacara	
○ Samuel Hym	○ Quentin Bergougnoux	
○ Julien Iguchi-Cartigny	○ Nadir Cherifi	
○ Thomas Vantroys	○ Narjes Jomaa	
• **Research scientist**	○ Valentin Lefils	
○ David Nowak	○ Mahieddine Yaker	

Figure 1.2: 2XS team organigram

---

[1]as of August 6, 2017

2. PIP project

2.1 PIP protokernel

2.1.1 A minimal OS kernel with proovable isolation

An operating system is organized as a hierarchy of layers, each one constructed upon the one below it. Each layer focuses on an essential role of the operating system such as memory management, multiprogramming, input/output . . . Generally speaking, while developing a kernel for an operating system based on the layered approach, the designers have a choice where to draw the kernel-user boundary. Traditionally, all the layers went in the kernel, but that is not necessary. In fact, putting as little as possible in kernel mode is safer because kernel bugs can bring down the system instantly. In contrast, user processes/layers are set up to have less power so that a bug there may not be fatal.

Various studies on bug density, relatively to the developed module size, age as well as other factors, have been conducted (e.g. *Basilli* and *Perricone* in 1984; and *Ostrand* and *Weyuker* in 2002). A ballpark figure for serious industrial systems is ten bugs per a thousand lines of code. Operating systems are sufficiently buggy that computer manufacturers put reset buttons on them, something the manufacturers of cars, TV sets and stereos do not do, despite the large amount of software in these devices. Furthermore, Operating systems generally present hardware resources to applications through high-level abstractions such as (virtual) file systems.

Therefore, we can distinguish several OS kernel families such as :

- **Microkernels :** The basic idea behind a microkernel design is to achieve high reliability by splitting the operating system up into small, well-defined modules, only one of which, the microkernel, runs in kernel mode;

- **Exokernels :** The idea behind an exokernel is to force as few abstractions as possible on application developers, enabling them to make as many decisions as possible about hardware abstractions.

Although the closest kernel design to PIP is the exokernel, PIP does not belong to any of the kernel families featured in the state of art, but it is the

first member of a new kernel family, **protokernels**, as compared to most microkernels and exokernels, the TCB in PIP is even more restricted :

- Scheduling and IPC are done in user mode unlike a microkernel;

- Multiplexing is also done in user mode unlike an exokernel.

whereas the kernel mode is only for **multi-level MMU control and configuration** (virtual memory) and **context switching**. This not only ensures less bugs density but also more feasibility of formal proof that will warrant the memory isolation property of the protokernel.

As a **minimal OS Kernel with provable isolation**, PIP focuses more on security and safety without sacrificing efficiency and ensures memory isolation between different tasks running on the same device.PIP's algorithmic part is written in Gallina, the language of the Coq proof assistant, in a monadic style that allows direct translation into free-standing C. We will refer to this implementation in *Gallina* as the shallow embedding in contrast to the deep embedding introduced in section 2.3 p.20.
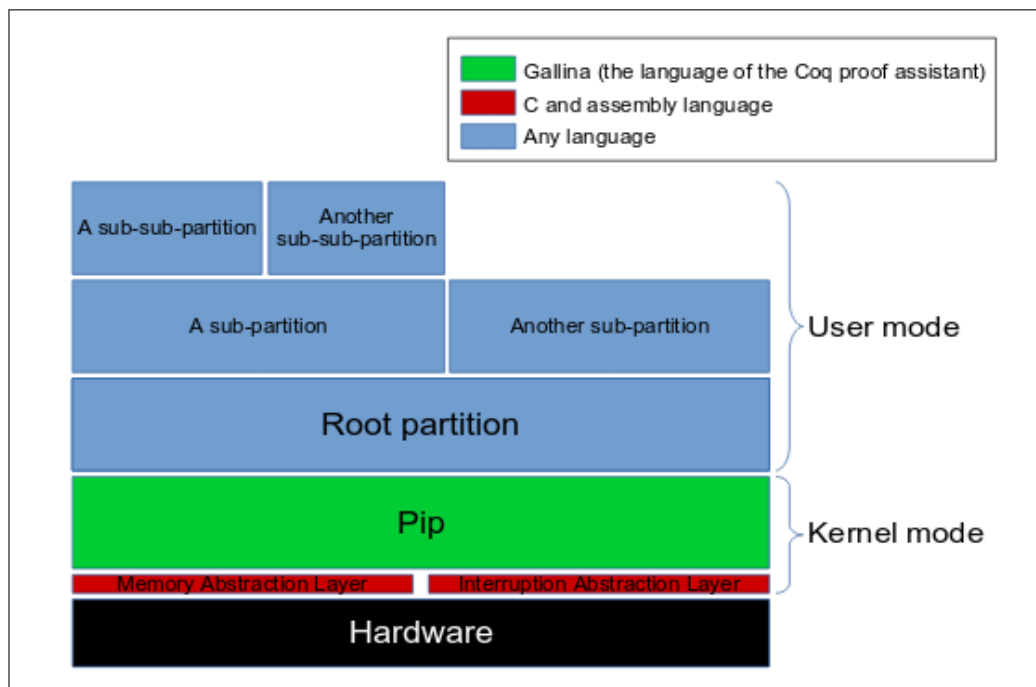
Figure 2.1: Software layers of an OS built on top of PIP

2.1.2   Horizontal isolation & vertical sharing

PIP can be used to partition the available memory which will be initially allocated to the root partition on top of PIP. Any partition can read and write in the memory of its children. However, partitions in different branches of the partition tree are disjoint. The former is referred to as **vertical sharing** and the latter as **horizontal isolation**. Needless to say, all memory lent to PIP for storing kernel data,such as when creating partitions, is inaccessible for all partitions to prevent messing up PIP data structures which means that the kernel data is totally isolated.
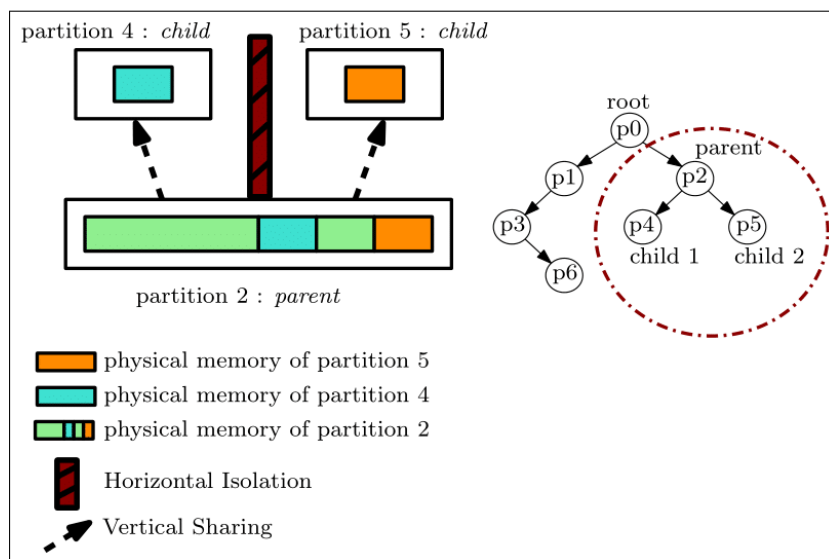
Figure 2.2: Horizontal isolation & vertical sharing in PIP

Let us consider a more realistic partition tree, as shown in figure 2.1.2, in which we consider *Linux* and *FreeRTOS* as sub-partitions of a root partition, multiplexer. Knowing that *FreeRTOS* is a real-time OS that does not isolate its tasks, we have easily secured it with task isolation by porting it on PIP.
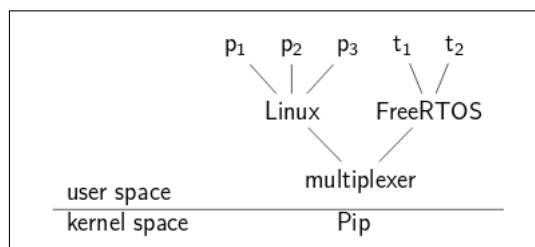
Figure 2.3: FreeRTOS task isolation using PIP

2.1.3 Proof-oriented design

PIP's design

PIP's isolation properties are meant to be formally proven independently from the platform it's running onto. Consequently, the algorithm and the architecture dependant part were separated. Indeed, as shown in figure 2.4, PIP is split into two distinct layers :

- **HAL :** gives direct access to the architecture and hardware;

- **API :** implements the algorithmic part to configure the virtual memory and the hardware.

The API code is written and proven using the Coq proof assistant, and uses the interface provided by the HAL to perform any hardware related operation. the proofs are based on Hoare logic theory introduced in section 2.2.2 p.15.
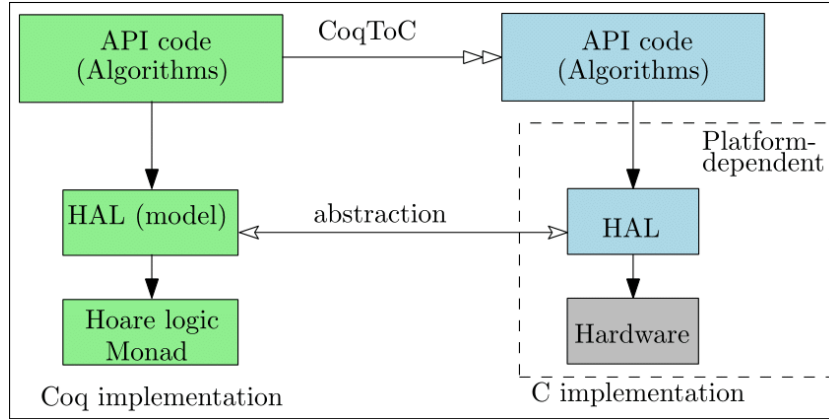
Figure 2.4: PIP design

PIP's HAL is split into three components, as shown in figure 2.5, each handling a specific part of the target platform's hardware :

- **Memory Abstraction Layer (MAL) :** provides an interface for the configuration of the MMU chip;

- **Interrupt Abstraction Layer (IAL) :** provides an interface to dispatch interrupts and configure hardware;

- **Bootstrap :** contains the low-level code required to boot the system.

6

PIP only provides system calls for management of the partitions and for context switching, thus reducing the TCB to its bare minimum as explained in section 2.1.1 p.3. This user exposed API can be called by any partition using a platform dependant call :

- **createPartition :** creates a new child (sub-partition) into the current partition;

- **deletePartition :** removes a child partition and puts all its used pages back in the current partition;

- **prepare :** adds required configuration tables into a child partition to map a new virtual address;

- **collect :** removes the empty configuration tables which are not used anymore and gives it back to the current partition;

- **countToMap :** returns the amount of configuration tables needed to perform a mapping for a given virtual address;

- **addVaddr :** maps a virtual address into the given child;

- **removeVAddr :** removes a given mapping from a given child.

This API is sufficient as far as memory requirements are concerned but it lacks a way to handle interrupts. Hardware interrupts are implicitly handled by PIP and automatically dispatched to the root partition, while software interrupts, such as system calls, are notified to the parent partition of the caller and can be managed by these two additional services of PIP :

- **dispatch :** notifies an interrupt to a given partition, interrupting its current control flow and backing it up for a further resume call;

- **resume :** restores a previously interrupted context.
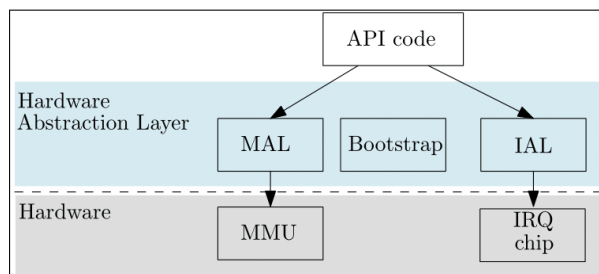
Figure 2.5: HAL and API relationship

Formal proofs using Coq

Coq is the result of about 30 years of research. It started in 1984 as an implementation of the Calculus of Constructions, an expressive formal language, at INRIA by *Thierry Coquand* and *Gérard Huet* and was extended,later in 1991, by Christine to the Calculus of Inductive Constructions.
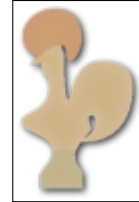
Coq is a **formal proof management system**. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs. Typical applications include the certification of properties of programming languages (e.g. the *CompCert* compiler certification project, or the *Bedrock* verified low-level programming library), the formalization of mathematics (e.g. the full formalization of the Feit-Thompson theorem or homotopy type theory) and teaching. It implements a program specification and mathematical higher-level language called *Gallina* that is based on the Calculus of Inductive Constructions combining both a higher-order logic and a richly-typed functional programming language.

As a **proof development system**, Coq provides interactive proof methods, decision and semi-decision algorithms as well as a tactic language letting the user define his own proof methods. Furthermore, as a **platform for the formalization of mathematics or the development of programs**, Coq provides support for high-level notations, implicit contents and various other useful kinds of macros.

For this project, the latest version of Coq, 8.6 released in December 2016, was used. It features among other things a faster universe checker, asynchronous error processing, proof search improvements, generalized introduction patterns, a new warning system, patterns in abstractions and a new subterm selection algorithm.

The Coq proof assistant provides us with powerful tactics to perform proofs. Here is a non exhaustive list of these tactics :

- **simpl :** simplifies the current goal;

- **assumption :** solves the current goal if it is computationally equal to a hypothesis;

- **reflexivity :** solves the current goal if it is a valid equality;

- **unfold t :** unfolds the definition of t in the current goal;

- **clear H :** removes hypothesis H from the context;

- **auto :** tries to solve the current goal automatically by using a collection of tactics;

- **rewrite H :** uses an equality hypothesis H to replace a term in the current goal;

- **induction t:** applies the induction principle for the type of t, generates subgoals as many as there are constructors, and adds the inductive hypotheses in the contexts;

- **inversion H :** resembles the induction tactic, but pays attention to the particular form of the type of H, and will only consider the cases that could have been used, so it discards some impossible cases quickly and efficiently;

- **omega :** carries out an automatic decision procedure for *Presburger* arithmetic;

- **apply H :** mainly tries to unify the current goal with the conclusion of the type of H. If it succeeds, then the tactic returns as many subgoals as non-dependent premises of the type of H;

- **f_equal :** applies to a goal of the form $f\ a_1 \ldots a_n = g\ b_1 \ldots b_n$ and leads to subgoals $f = g$ and $a_1 = b_1$ and so on up to $a_n = b_n$. Amongst these subgoals, the simple ones are automatically solved;

- **contradiction :** tries to find a contradiction amongst the hypotheses.

Coq also provides several tactics to deal with first-order logic, particularly targeting binary connectives and quantifiers such as **split**, **left** and **right**, **intros**, **exists** and **apply** respectively devised for conjunction, disjunction, implication, existential quantifiers and universal quantifiers. Furthermore, many tactics can be preceded with the letter *e* like **eauto**, **eassumption** and **eapply** to deal with existential variables. Some tactics can also be applied on hypotheses if we use the reserved clause *in* with the name of the hypothesis such as unfold, apply and simpl.

2.1.4   In-depth understanding of PIP's Data structures

The memory

PIP uses several data structures per partition, which will represent the global state of the partition's memory state. This is necessary as it has to keep track of pages allocated to partitions in order to allow or deny derivation and partition creation while preserving the required properties.

Figure 2.6 shows a partition tree example consisting of a partition, Parent, that has a single child, Child1. A partition is identified by a partition descriptor which is a page number essential to access the whole data structure of a partition. In our case, the partition descriptors of Parent and Child1 are respectively 1 and 12. The pages lent to PIP to manage a partition are organized in a tree with three branches : the MMU tables, the first shadow and the second shadow. The aim of this organization is to keep additional information about each page lent to a partition. Moreover these structures have multiple goals :

- **Access control :** the first shadow is used to avoid deriving the same page multiple times;

- **Performance :** the second shadow and the configuration list are used to quickly find the virtual address of a page without having to parse the whole virtual space when the parent partition reclaims it.

As such, adding an indirection table in the MMU configuration requires two additional pages for the shadows. Therefore, this model is estimated to require roughly three times the amount of memory a simple virtual environment would need, nevertheless it provides a secure and an efficient API.

To prove the isolation properties on this memory structure it is essential to assure its consistency relative to the partition tree, the well-typedness as well as some other consistency properties that will be detailed in section 2.2.2 p.15.

Figure 2.6: An example of a partition tree

The state

The PIP state is defined as follows :

Script 2.1: PIP state definition

```
Record state : Type :=
{ currentPartition: page ; memory: list (paddr * value) }.
```

The list in the state corresponds to the physical memory and maps physical addresses to values. A physical address is defined by a physical page number and a position into this page :

Script 2.2: paddr type definition

```
Definition paddr : Type := page * index.
```

11

where pages and indexes are positive integers bounded respectively by the overall number of pages and the table size :

Script 2.3: page & index type definitions

```
Record page  := { p :> nat ; Hp : p < nbPage }.
Record index := { i :> nat ;  Hi : i < tableSize }.
```

Different types of values could be stored in the physical memory by Pip. So, the type *value* is an inductive type defined as follows :

Script 2.4: value type definition

```
Inductive value : Type :=
          | PE: Pentry → value
          | VE: Ventry → value
          | PP: page   → value
          | VA: vaddr  → value
          | I:  index  → value.
```

The type *Pentry*, which represents a physical entry, consists of a physical page number along with several flags :

Script 2.5: Pentry type definition

```
Record Pentry : Type := {
          read:    bool ;
          write:   bool ;
          exec:    bool ;
          present: bool ;
          user:    bool ;
          pa:      page }.
```

Finally, the *Ventry* type consists of a virtual address with a unique boolean flag :

Script 2.6: Ventry type definition

```
Record Ventry : Type := { pd: bool ; va: vaddr }.
```

with virtual addresses modelled as a list of indexes of length the number of levels of the MMU plus one :

Script 2.7: vaddr type definition

```
Record vaddr : Type :=
{ va :> list index ; Hva : length va = nbLevel + 1  }.
```

2.2 Hoare logic

2.2.1 Introduction to Hoare logic theory

How can we argue that a program is correct ? Nowadays, Building reliable software is becoming more and more difficult considering the growing scale, specifications and complexity of modern systems. Therefore, tests alone can no longer ascertain the reliability of programs especially if we're talking about critical systems. Logicians, computer scientists and software engineers have responded to these challenges by developing different kinds of techniques some of which are based on formal reasoning about properties of software and tools for helping validate these properties. One of these reasoning techniques that was used to prove PIP's properties is *Floyd–Hoare logic*, often shortened to just **Hoare Logic**. It was proposed in 1969 by the British computer scientist and logician Tony Hoare and it continues to be the subject of intensive research right up to the present day. It is not only a natural way of **writing down specifications of programs** but also a technique for **proving that programs are correct** with respect to such specifications.

Let S be a program that we want to execute starting from a certain state s, P a predicate on the state describing the condition S relies on for correct execution and Q a predicate on the resulting state after the execution of S describing the condition S establishes after correctly running. Knowing that P is verified on s, if we prove that Q is verified after the execution of s, we can ascertain that S is partially correct. And by partial correctness of S we mean that S is correct if it terminates. Using standard Hoare logic, only partial correctness can be proven, while termination needs to be proven separately. This triple S, P and Q, written as **{P} S {Q}** , is referred to as a **Hoare triple**. The assertions P and Q are respectively referred to as the **precondition** and the **postcondition**.

For example let's consider simple Hoare triples about an assignment command :

- **$\{X = 2\}X := X + 1\{X = 3\}$** : is a valid Hoare triple, that can be easily formally proved in Coq, since the postcondition is verified after the execution of the assign command relatively to the precondition;

- **$\{X = 2\}X := X * 2\{X = 3\}$** : is not a valid Hoare triple since the postcondition would not be verified after the execution of the command.

Now, let's introduce some facts and rules about Hoare triples :

1. If an assertion P implies another precondition P' of a valid Hoare triple {P'} S {Q} then {P} S {Q} is also a valid Hoare triple. This is referred to as **weakening the precondition of a Hoare triple** and can be formally defined as follows :

$$\frac{P \rightarrow P' \qquad \{P'\} \ S \ \{Q\}}{\{P\} \ S \ \{Q\}} \ \text{Hoare\_weaken}$$

2. If we can weaken a Hoare triple, we expect it to have a **weakest precondition**. This notion was introduced by Dijkstra in 1976 and is very important since it enables us to prove total correctness and in particular program termination. Indeed, if a program doesn't terminate, its weakest precondition would be *True* and it would verify any postcondition.

3. If we consider a language containing a **SKIP instruction** which practically does nothing, we can affirm that this command preserves any property which means :

$$\frac{}{\{P\} \ \text{SKIP} \ \{P\}} \ \text{Hoare\_skip}$$

4. If we consider a language that allows **assignments**, in the form of *X := a*, then we can conclude that an arbitrary property Q holds after such assignment if we assume Q[X→a] which means Q with all occurrences of X replaced by a :

$$\frac{}{\{Q[X\rightarrow a]\} \ X := a \ \{Q\}} \ \text{Hoare\_assign}$$

5. Generally speaking, every Program is built using **sequencing** of commands that we will write as C1**;**C2. Our aim is to prove a Hoare triple on this sequence with P and Q respectively as the precondition and the postcondition. This requires proving that C1 takes any state where P holds to a state where an intermediate assertion I holds and C2 takes any state where I holds to one where Q holds which could be formally stated as :

$$\frac{\{P\} \ C1 \ \{I\} \qquad \{I\} \ C2 \ \{Q\}}{\{P\} \ C1;C2 \ \{Q\}} \ \text{Hoare\_seq}$$

6. Finally, let's not forget **conditional structures** that we can write in the form of *IF B THEN C1 ELSE C2*. To verify a Hoare triple about this instruction with P and Q respectively as the precondition and postcondition, we need to consider both cases where B is evaluated to *true* and *false* and prove that Q holds on the resulting state in each one. We can also disregard a case if there is some sort of contradiction relatively to the property P. This rule can be written as follows :

$$\frac{\{P \bigwedge B\} \; C1 \; \{Q\} \qquad \{P \bigwedge \neg \, B\} \; C2 \; \{Q\}}{\{P\} \; IF \; B \; THEN \; C1 \; ELSE \; C2 \; \{Q\}} \; \text{Hoare\_if}$$

The rules defined above are in their simplest form and we need to adapt them to the language or semantics we're working on. Also, This list isn't exhaustive. For example, we didn't define a rule for *While* loops because the Coq model of PIP doesn't actually use them and uses recursion instead. Likewise, We're sure that all PIP's functions terminate since recursive functions are bounded by a maximum number of iterations.

2.2.2 Hoare logic in the shallow embedding

The Hoare logic devised for the shallow embedding is slightly different from what we saw in the previous section. A program in the shallow embedding is defined in a monadic style that returns a pair of values, the first being the resulting state and the second the value returned by the program which generally corresponds to a function. Thus, as shown in script 2.8, **the postcondition must not only reason on the resulting state but also on the returned value** so that we can verify it and propagate it in the case of long proofs. The notation chosen for the Hoare triple stays the same as the one mentioned in the previous section except for the brackets that are doubled since single brackets are already used in coq.

Script 2.8: Hoare triple in the shallow embedding

```
Definition hoareTriple {A : Type} (P : state → Prop)
(m : LLI A) (Q : A → state → Prop) : Prop :=
  ∀ s, P s → match (m s) with
    | val (a, s') => Q a s'
    | undef _ _=> False
    end.


Notation "{{ P }} m {{ Q }}" := (hoareTriple P m Q)
  (at level 90, format "'[' '[' {{  P  }}  ']' '/  '
  '[' m ']' '['  {{  Q  }} ']' ']'") : state_scope.
```

The weakening lemma on Hoare triples was also defined and proven as shown in script 2.9. This Lemma is extensively used in PIP's proofs.

Script 2.9: Weakening Hoare triples in the shallow embedding

```
Lemma weaken (A : Type) (m : LLI A)
(P Q : state → Prop) (R : A → state → Prop) :
  {{ Q }} m {{ R }} →
  (∀ s, P s → Q s) →
  {{ P }} m {{ R }}.
Proof.
intros H1 H2 s H3.
case_eq (m s); [intros [a s'] H4 | intros a H4 ];
apply H2 in H3; apply H1 in H3;
try rewrite H4 in H3; trivial.
intros. rewrite H in H3. assumption.
Qed.
```

Some simple instructions were considered as primitive like conditional structures while some others were explicitly defined, like assignments in the form of ***perform x := m in e*** where the value of the program m gets assigned to x in the evaluation of the program e. Furthermore, a Hoare triple rule was devised for assignments.

Script 2.10: Hoare triples assignment rule in the shallow embedding

```
Lemma bind (A B : Type) (m : LLI A) (f : A → LLI B)
(P : state → Prop) ( Q : A → state → Prop)
(R : B → state → Prop) :
  (∀ a, {{ Q a }} f a {{ R }}) →
  {{ P }} m {{ Q }} →
  {{ P }} perform x := m in f x {{ R }}.
Proof.
intros H1 H2 s H3; unfold bind;
case_eq (m s); [intros [a s'] H4 | intros k s' H4];
apply H2 in H3; rewrite H4 in H3; trivial.
case_eq (f a s'); [intros [b s''] H5 | intros k s'' H5];
apply H1 in H3; rewrite H5 in H3; trivial.
Qed.
```

Finally, to reason with Hoare logic we need to formally specify the properties we want to prove, the most important being partition memory isolation, kernel data isolation, vertical sharing and consistency mentioned in 2.1.2 p.5. To that end, the following necessary functions on PIP's data structures were defined :

- **getChildren :** returns the list of all children of a given parent partition in the partition tree of a given state;

- **getAncestors :** returns the list of all ancestors of a given partition in the partition tree of a given state;

- **getMappedPages :** returns the list of mapped pages of a given partition;

- **getAccessibleMappedPages :** returns the list of all mapped pages, of a given partition, marked as accessible;

- **getUsedPages :** returns the list of all the pages that are used by a given partition including the pages lent to PIP;

- **getConfigPages :** returns the list of configuration pages lent to PIP to manage a given partition;

- **getPartitions :** returns the list of all existing partitions of a given state which is naturally obtained by a search on the partition tree.

The first property defines horizontal isolation between children i.e., for any state s of the system, all memory pages owned by two different children of a given parent partition in the partition tree must be distinct. This property is formally defined in the Coq proof assistant as follows :

Script 2.11: Horizontal isolation

```
Definition horizontalIsolation s :=
∀ parent child1 child2,
parent ∈ getPartitions s →
child1 ∈ getChildren parent s →
child2 ∈ getChildren parent s →
child1 ≠ child2 →
(getUsedPages child1 s) ∩ (getUsedPages child2 s) = ∅.
```

The second property defines vertical sharing between a parent partition and its children i.e. it ensures that all pages mapped by a child are mapped in its parent. This property is formally defined as follows :

Script 2.12: Vertical sharing

```
Definition verticalSharing s :=
∀ parent child,
parent ∈ getPartitions s →
child ∈ getChildren parent s →
getUsedPages child s ⊆ getMappedPages parent s.
```

The third property defines kernel data isolation meaning it ensures that for any state of the system and any given two possibly-equal partitions, the code running in the second partition cannot access the pages containing configuration tables lent to PIP to manage the first partition. This property is formally defined as follows :

Script 2.13: Kernel data isolation

```
Definition kernelDataIsolation s :=
∀ partition1 partition2,
partition1 ∈ getPartitions s →
partition2 ∈ getPartitions s →
(getAccessibleMappedPages partition1 s) ∩
(getConfigPages partition2 s) = ∅.
```

The last property of consistency is mandatory to prove the previously detailed properties. Consistency encompasses different sub-properties that were divided into four categories. As the definition of this property is quite cumbersome, we will only disclose these categories and illustrate each one with an example :

- **Partition tree structure :** the properties in this category ensure that the partition tree of a given state is consistent relatively to its awaited structure preset in section 2.1.4 p.10. For example, one of the properties defined in this category and called *noCycleInPartitionTree* ensures that there is no cycle in the partition tree of a given system i.e. each partition is different from all its ancestors in the partition tree. This property is formally defined as follows :

  Script 2.14: Example of partition-tree-consistency property

  ```
  Definition noCycleInPartitionTree s :=
  ∀ ancestor partition,
  partition ∈ getPartitions s →
  ancestor ∈ getAncestors partition s →
  ancestor ≠ partition.
  ```

- **Flag semantics :** This category focuses on the signification of flags used in the data structures. For instance, a property called *isPresentNotDefaultIff* states that the associated page number of a physical entry whose *present* flag is set to *false*, corresponds to the predefined default page value. This property is formally defined as shown in script 2.15 where *readPhyEntry* and *readPresent* are predefined functions that respectively read, for a given table, index and memory state, the present flag of a physical entry and its value :

Script 2.15: Example of a flags-semantics-consistency property

```
Definition isPresentNotDefaultIff s :=
∀ table idx ,
readPresent table idx (s.memory) = Some false ↔
readPhyEntry table idx (s.memory) = Some defaultPage.
```

- **Pages properties :** This category concerns several properties about the partitions' mapped pages, the pages lent to the kernel as well as the relations between them. For example, the *noDupMappedPagesList* requires that the mapped pages of partitions be distinct from each other, using the function *NoDup* which verifies whether a list contains duplicates or not. This property is formally defined as follows :

Script 2.16: Example of a pages-consistency property

```
Definition noDupMappedPagesList s :=
∀ partition , partition ∈ getPartitions s →
NoDup (getMappedPages partition s).
```

- **Well-typedness :** This last category concerns kernel data types and ensures that PIP doesn't contain any type confusion. For instance, when PIP writes a virtual address in the memory, it should be read later as a virtual address which is ensured by a property called *dataStructurePdSh1Sh2asRoot*. Otherwise, it is considered as an undefined behaviour in the model.

The only Hoare triple that was sucessfully proven to this date[1] is the one about the *createPartition* call that required about 60000 lines of code to prove and which is defined as follows :

Script 2.17: createPartition Hoare triple

```
Lemma createPartition (descChild pdChild
          shadow1 shadow2 list : vaddr) :
{{fun s ⇒ partitionsIsolation s ⋀ kernelDataIsolation s
          ⋀ verticalSharing s ⋀ consistency s }}
createPartition descChild pdChild shadow1 shadow2 list
{{fun _ s  ⇒ partitionsIsolation s ⋀ kernelDataIsolation
                s ⋀ verticalSharing s ⋀ consistency s }}.
```

---

[1]August 6, 2017

2.3 The deep embedding

2.3.1 Deep versus shallow embedding

A deep embedding is mainly an **abstract representation of a language by modelling its expressions as data**. Strictly speaking, The main difference between a shallow and deep embedding is that the former is **extensible with regards to adding language constructs** while the latter is **extensible with regards to adding interpretations of its abstract data types**. A typical approach in PIP's case would have been reasoning on a predefined deep embedding as structural abstraction enables us to perform cost-effective software verification by inversion on formulas' structure. However, defining such an abstract language, at the beginning, is quite tedious and rather intricate since we need to worry about the wingspan of such language and expression soundness among others. Therefore, the tasks of proof engineering and developing the deep embedding were separated. At first, a formal reasoning system based on Hoare logic was built on the shallow embedding as shown in section 2.2.2 p.15. Then, an abstract representation of the shallow model was written in *Gallina* which corresponds to the deep embedding. Needless to say, at some point later on, we need to worry about the preservation of proven properties between both embeddings either by proving that there is a bijection between them or by rewriting all the proofs in the deep embedding. By and large, we need to **ascertain the feasibility of such proofs in the deep embedding** as well as **compare examples of conducted proofs in both embeddings**.

2.3.2 Deep embedding constructs

Before we move on to the main constructs of the deep embedding, we need to introduce essential background types and structures :

- **Id type :** represents the identifiers' type in the deep embedding. Typically, identifiers of variables are strings but we can use any other type;

- **Value type :** represents the actual values the programs will be manipulating. They can be built using the constructor *cst* followed by a shallow type and a value that should be of that type. The value type and its constructor are defined as follows :

Script 2.18: Values in the deep embedding

```
(** the internal type of values ,
        parametrised by a semantic type *)
Inductive ValueI (T: Type) : Type := Cst (v: T).

(** the external type of values ,
        hiding the semantic type *)
Definition Value : Type := sigT ValueI.

(** smart value constructor *)
Definition cst (T: Type) (v: T) : Value :=
        @existT Type ValueI T (Cst T v).
```

- **QValue type :** represents quasi-values in the deep embedding which can be either actual values lifted to quasi-values using the constructor *QV* or identifiers of variables in the variable environment that we lift to quasi-functions using the constructor *Var*, as shown in the following script :

Script 2.19: Quasi-values in the deep embedding

```
Inductive QValue : Type := Var (x: Id)
                          | QV (v: Value).
```

- **Fun type :** represents functions in the deep embedding. The definition of this type is detailed in script 2.20 where :

  - **fenv** corresponds to the function environment which maps identifiers to functions;

  - **tenv** corresponds to the typing environment for parameters which maps identifiers to value types;

  - **e0** and **e1** are expressions in the deep embedding corresponding to the first and second inductive cases;

  - **x** corresponds to the name of the function;

  - **n** is the bound. If its value is 0, the first expression *e0* is evaluated else the second expression *e1* gets evaluated. This enables us to define terminating recursive functions.

Script 2.20: Functions in the deep embedding

```
Inductive Fun : Type :=
        FC (fenv: Envr Id Fun)
           (tenv: valTC)
           (e0 e1: Exp)
           (x: Id)
           (n: nat).
```

- **QFun type :** represents quasi-functions in the deep embedding which can be either actual functions that we lift to quasi-functions using the constructor *QF* or identifiers of functions in the function environment that we lift to quasi-functions using the constructor *FVar*, as shown in the following script :

Script 2.21: Quasi-functions in the deep embedding

```
Inductive QFun : Type := FVar (x: Id)
                        | QF (f: Fun).
```

- **XFun type :** represents effect handlers in the deep embedding used to define prospective operations on the state and/or an input. It is used by the *Modify* construct of the deep embedding defined in script 2.26 p.24, when we want to read the state or perform changes on it and also when we need to implement non-recursive functions. As shown in script 2.22, it requires an input type *T1* as well as an output type *T2*. Thus, to define state operations, we will mainly need effect handlers with *unit* as the input and output type. Moreover, when specifying an effect handler, we only need to worry about the b_mod attribute as b_exec and b_eval are already preset to build respectively the resulting state and value. Although this implementation deals with the issue of construct extensibility since it enables us to add shallow constructs and even use shallow functions, it can be risky as we can confuse shallow with deep. This will be further discussed in section **??** :

Script 2.22: effect handlers in the deep embedding

```
Record XFun (T1 T2: Type) : Type := {
    b_mod : W → T1 → prod W T2 ;
    b_exec : W → T1 → W :=
     fun state input ⇒ fst (b_mod state input) ;
    b_eval : W → T1 → T2 :=
     fun state input ⇒ snd (b_mod state input)
}.
```

As shown in script 2.26, The deep embedding provides us with 8 constructs to build expressions :

1. **Val** *v* **:** lifts the value *v* to an expression using the constructor *Val*;

2. **BindN** *e1 e2* **:** represents a **sequence** of instructions where the expression *e1* is evaluated first then *e2* next. It is equivalent to "*e1;;e2*" in the shallow embedding;

3. **BindS** *x e1 e2* **:** represents an **assignment**. It is equivalent to "*perform x := e1 in e2*" in the shallow embedding. More precisely, the expression *e2* is evaluated in the updated variable environment where *x* gets mapped to the resulting value of the evaluation of *e1*;

4. **BindMS** *fenv env e* **:** evaluates the expression *e* in the function environment *fenv* and variable environment *env*;

5. **IfThenElse** *e1 e2 e3* **:** represents a **conditional structure**. It is equivalent to "*if e1 then e2 else e3*" in the shallow embedding. This expression is considered well-typed only when *e1* gets evaluated to a deep boolean value;

6. **Return** *G qv* **:** lifts the quasi-value *qv* to an expression using the constructor *Return*. If *qv* is a variable, it gets evaluated in the variable environment. *G* is a flag which affirms whether the construct should get translated to an actual return in C or not. The type *Tag* of *G* is defined as follows :

<div align="center">

Script 2.23: Tag type in the deep embedding

```
Inductive Tag : Type := LL | RR.
```

</div>

7. **Apply** *qf ps* **:** represents **function application**. If *qf* is a variable it gets evaluated in the function environment. *ps* is the list of parameters which are expressions. They should be well-typed relatively to the typing environment of the function i.e. the resulting value of the evaluation of each parameter's expression should match its awaited type preset in the typing environment. The type *Prms* of *ps* is defined as follows :

<div align="center">

Script 2.24: Function parameters in the deep embedding

```
Inductive Prms : Type := PS (es: list Exp).
```

</div>

8. **Modify** *T1 T2 VT1 VT2 xf qv* **:** is used to define state operations and non-recursive functions. *xf* is the effect handler that specifies the operation, *T1* and *T2* are respectively its input and output types and *VT1* and *VT2* assure that *T1* and *T2* are admissible value types. If *qv* is a variable it gets evaluated in the variable environment. For instance, we can define a simple *SKIP* instruction with the *Modify* construct as follows :

Script 2.25: SKIP instruction in the deep embedding

```
Definition xf_skip : XFun unit unit := {|
        b_mod := fun state input => (state,tt) |}.

Definition SKIP : Exp := Modify unit unit
   UnitVT UnitVT xf_skip (QV(cst unit tt))
```

More examples of these constructs will be detailed in section **??** p.**??**.

Script 2.26: Deep embedding expressions

```
Inductive Exp : Type :=
 | Val (v: Value)
 | BindN (e1: Exp) (e2: Exp)
 | BindS (x: Id) (e1: Exp) (e2: Exp)
 | BindMS (fenv: Envr Id Fun) (env: valEnv) (e: Exp)
 | IfThenElse (e1: Exp) (e2: Exp) (e3: Exp)
 | Return (G: Tag) (qv: QValue)
 | Apply (qf: QFun) (ps: Prms)
 | Modify (T1 T2: Type) (VT1: ValTyp T1) (VT2: ValTyp T2)
              (xf: XFun T1 T2) (qv: QValue)
```

The deep expressions are meant to be evaluated in certain variable and function environments. To that end, single and multi-step evaluation were defined as follows :

- **EStep fenv env (Conf Exp n e) (Conf Exp n' e') :** represents a **single-step evaluation** of the expression *e*, in the function environment *fenv* and variable environment *env*, with *n* as the current state. The constructor *Conf* builds a configuration of the type we seek to evaluate which, in our case, is *Exp* and contains the current state as well as the starting expression. *n'* and *e'* are respectively the resulting state and expression. For example, we can affirm that forall function environment *fenv*, variable environment *env*, state *n* and value *v* :

  EStep fenv env (**Conf** Exp n (Val v)) (**Conf** Exp n (Val v))

- **PrmsStep fenv env (Conf Prms n es) (Conf Prms n' es') :** represents a single-step evaluation of the list of parameters *es* which conducts a single-step evaluation of the first expression in the list. If the first expression is a value, it moves on to the next parameter;

- **EClosure fenv env (Conf Exp n e) (Conf Exp n' e') :** represents a **multi-step evaluation** of the expression *e*. It uses the predefined single evaluation steps. It is important to note that there exists a state where any expression gets evaluated to a certain value. Furthermore, the evaluation process is deterministic i.e. an expression gets evaluated to a unique value;

- **PrmsClosure fenv env (Conf Prms n es) (Conf Exp n' es') :** represents a multi-step evaluation of the list of parameters *es* which conducts a multi-step evaluation of each expression in the list.

Finally, the state in the deep embedding is represented with an abstract type *W*, as shown in script 2.22 p.22, that can be specified as required. We also have at our disposal several proven **congruence** and **concatenation** lemmas for both expressions and parameters.

2.3.3 Hoare logic in the deep embedding

A Hoare triple in the deep embedding is slightly different from its counterpart in the shallow embedding. Indeed, this new version of the Hoare triple depends on the function environment as well as the variable environment we're evaluating the expression in. Both will be passed to the Hoare triple as parameters along with the expression, the precondition and the postcondition. The precondition is a predicate on the state while the postcondition is a predicate on both the resulting value and state. We must also ensure that the expression we are dealing with is well-typed. Therefore, as shown in script 2.27, a Hoare triple is considered valid if and only if the expression *e* is well-typed and there is an *EClosure* from a certain starting state verifying the precondition to a certain resulting state and value that verify the postcondition with *e* evaluated in the given function and variable environments *fenv* and *env*. **{{P}} fenv >> env >> e {{Q}}** is the chosen notation for the Hoare triple in the deep embedding where *P*, *Q*, *e*, *fenv*, and *env* are respectively the precondition, the postcondition, the expression, the function environment and the variable environment. Furthermore, as the evaluation of parameters is different from that of expressions, we need to devise a separate Hoare triple for parameters as shown in script 2.28 where *EClosure* is replaced by *PrmsClosure*.

Script 2.27: Hoare triple for expressions in the deep embedding

```
Definition THoareTriple_Eval
    (P : W -> Prop) (Q : Value → W → Prop)
    (fenv: funEnv) (env: valEnv) (e: Exp) : Prop :=
 ∀ (ftenv: funTC) (tenv: valTC)
   (k1: FEnvTyping fenv ftenv)
   (k2: EnvTyping env tenv)
   (t: VTyp)
   (k3: ExpTyping ftenv tenv fenv e t)
   (s s': W) (v: Value),
 EClosure fenv env (Conf Exp s e) (Conf Exp s' (Val v))
 → P s → Q v s'.

Notation "{{ P }} fenv >> env >> e {{ Q }}" :=
(THoareTriple_Eval P Q fenv env e) (at level 90).
```

Script 2.28: Hoare triple for parameters in the deep embedding

```
Definition THoarePrmsTriple_Eval
    (P : W → Prop) (Q : list Value → W → Prop)
    (fenv: funEnv) (env: valEnv) (ps: Prms) : Prop :=
 ∀ (ftenv: funTC) (tenv: valTC)
   (k1: FEnvTyping fenv ftenv)
   (k2: EnvTyping env tenv)
   (pt: PTyp)
   (k3: PrmsTyping ftenv tenv fenv ps pt)
   (s s': W) (vs: list Value),
 PrmsClosure fenv env (Conf Prms s ps) (Conf Prms s'
        (PS (map Val vs))) → P s → Q vs s'.
```

Several Hoare triple rules were devised and proven. We show the importance of these rules in section **??** p.**??**. For simplicity's sake, We will only give the formal definition of some of these rules and not their actual implementation in Coq :

- **weakening lemmas :** the weakening lemma on expressions is the same as the one defined for the shallow embedding in script 2.9 p.16. However, we need another weakening lemma for a Hoare triple on parameters. The weakest preconditions for expression and parameter triples were also defined as functions called respectively *wp* and *wpPrms*. The two weakening lemmas are formally defined as follows :

$$\frac{\{\{P'\}\} \text{ fenv} >> \text{env} >> \text{e} \{\{Q\}\} \qquad P \to P'}{\{\{P\}\} \text{ fenv} >> \text{env} >> \text{e} \{\{Q\}\}} \text{ weakenEval}$$

$$\frac{\text{THoarePrmsTriple\_Eval P' Q fenv env ps} \qquad \text{P} \rightarrow \text{P'}}{\text{THoarePrmsTriple\_Eval P Q fenv env ps}} \text{ weakenPrms}$$

- **BindS rule :** the assignment rule in the deep embedding is quite different from the shallow one, defined in script 2.10 p.16, as environments are now explicitly manipulated. Therefore, to prove a Hoare triple on an assignment of the form *BindS x e1 e2*, we need to prove a Hoare triple on *e2* where *x* is mapped to the resulting value of *e1* in the updated variable environment. To that end, we need an intermediate predicate that will ascertain the validity of the resulting value. This rule is formally defined as follows :

$$\frac{\{\{P0\}\} \text{ fenv} >> \text{env} >> \text{e1 } \{\{P1\}\}}{\forall \text{ v, } \{\{P1 \text{ v}\}\} \text{ fenv} >> (x,v)\text{:env} >> \text{e2 } \{\{P2\}\}}{\{\{P0\}\} \text{ fenv} >> \text{env} >> \text{BindS x e1 e2 } \{\{P2\}\}} \text{ BindS\_VHTT1}$$

- **BindN rule :** a variant of the *Hoare_seq* rule, introduced in section 2.2.1 p.13, which requires an intermediate predicate to prove a Hoare triple on a sequence of instructions. It is formally defined as follows :

$$\frac{\{\{P0\}\} \text{ fenv} >> \text{env} >> \text{e1 } \{\{\text{fun } \_ \Rightarrow P1\}\}}{\{\{P1\}\} \text{ fenv} >> \text{env} >> \text{e2 } \{\{P2\}\}}{\{\{P0\}\} \text{ fenv} >> \text{env} >> \text{BindN e1 e2 } \{\{P2\}\}} \text{ BindN\_VHTT1}$$

- **IfThenElse rule :** a variant of the *Hoare_if* rule, introduced in section 2.2.1 p.13. In the deep embedding, we need an intermediate predicate to evaluate the condition then we reason on both possible cases. This rule is formally defined as follows :

$$\frac{\{\{P0\}\} \text{ fenv} >> \text{env} >> \text{e1 } \{\{P1\}\}}{\{\{P1 \text{ (cst bool true)}\}\} \text{ fenv} >> \text{env} >> \text{e2 } \{\{P2\}\}}{\{\{P1 \text{ (cst bool false)}\}\} \text{ fenv} >> \text{env} >> \text{e3 } \{\{P2\}\}}{\{\{P0\}\} \text{ fenv} >> \text{env} >> \text{IfThenElse e1 e2 e3 } \{\{P2\}\}} \text{ IfTheElse\_VHTT1}$$

- **Apply rules :** the apply construct has several rules :

  - **Apply_VHTT1 :** the main rule of the Apply construct which evaluates not only the parameters but also the next recursive call of the function by performing a pattern matching on the bound. It is formally defined in Coq as follows :

Script 2.29: Main Hoare triple rule for the Apply construct

```
Lemma Apply_VHTT1 (P0: W → Prop)
   (P1: list Value → W → Prop)
   (P2: Value → W → Prop)
   (fenv: funEnv) (env: valEnv)
   (f: Fun) (es: list Exp) :
 THoarePrmsTriple_Eval P0 P1 fenv env (PS es) →
 match f with
  | FC fenv' tenv' e0 e1 x n ⇒
    length tenv' = length es /\
    match n with
    | 0 ⇒ (∀ vs: list Value,
       THoareTriple_Eval (P1 vs) P2 fenv'
           (mkVEnv tenv' vs) e0)
    | S n' ⇒ (∀ vs: list Value,
       THoareTriple_Eval (P1 vs) P2
           ((x,FC fenv' tenv' e0 e1 x n')::fenv')
           (mkVEnv tenv' vs) e1)
    end
  end →
 THoareTriple_Eval P0 P2 fenv env (Apply (QF f) (PS es)).
```

– **Apply_VHTT2 :** only evaluates the parameters in a function application. It's de defined as follows :

$$\frac{\text{THoarePrmsTriple\_Eval P0 P1 fenv env (PS es)} \quad \forall \text{ vs, } \{\{P1 \text{ vs}\}\} \text{ fenv} >> \text{env} >> \text{Apply (QF f) (PS (map Val vs)) } \{\{P2\}\}}{\{\{P0\}\} \text{ fenv} >> \text{env} >> \text{Apply (QF f) (PS es) } \{\{P2\}\}} \text{Apply\_VHTT2}$$

– **QFun_VHTT :** evaluates a function variable in an *Apply* construct by searching its value in the function environment. It is formally defined as follows :

$$\frac{\text{findET fenv x f} \quad \{\{P\}\} \text{ fenv} >> \text{env} >> \text{Apply (QF f) (PS es) } \{\{Q\}\}}{\{\{P\}\} \text{ fenv} >> \text{env} >> \text{Apply (FVar x) (PS es) } \{\{Q\}\}} \text{QFun\_VHTT}$$