Contents

List of Scripts

List of Figures

Acronyms

API Application Programming Interface.

DSL Domain Specific Language.

HAL Hardware Abstraction Layer.

IAL Interrupt Abstraction Layer.

IPC Inter-Process Communication.

MAL Memory Abstraction Layer.

MMU Memory Management Unit.

OS Operating System.

SOS Structural Operational Semantics.

TCB Trusted Computing Base.

1. Proving invariants in the deep embedding

In this section, we show how we proved three different invariants of PIP. The first one is about a function that reads the memory. The second one is about a function that writes in the memory. The last invariant is about a recursive function. We explain throughout this section our approach in modelling these functions and the way we engineered our proofs while trying to make them modular and as simple as possible. The first section is dedicated to a briefing the preliminary work we did to become more familiar with the deep embedding and the Hoare logic we built on it progressively.

1.1 Preliminary experiments with the deep embedding

For this preliminary work, we used an untyped form of the Hoare logic triple which was later refined to the one we defined in section **??** p.**??**. We started working with a natural number state on which we defined two functions. The first function called *ReadN* simply reads the current value of the state while the second one called *WriteN* writes a given value in the state. To model these functions, we used generic effects since they act directly on the state. We proved several Hoare triple rules about these functions. For instance we proved that if we write a value x in the state then instantly read it, we should get the same value x. This is quite similar to an invariant we proved on the PIP state, called *writeVirtualInvNewProp*, which is detailed in section 1.4 p.16. Then we switched to an association list state which resembles more the PIP state. We also devised shallow reading and writing functions on this state as the ones defined on the PIP state in annex **??** p.**??**. We used these functions to define deep reading and writing functions on such state using generic effects. Then we proved similar lemmas to the one we did on the natural number state.

1.2 Modelling the PIP state in the deep embedding

To prove invariants of PIP in the deep embedding, it is essential to replicate the PIP state. To that end, all type definitions mentioned in section **??** p.**??** are copied in the file *PIP_state.v*. All the axioms, constructors, comparison functions as well as predefined values were also copied in this file as shown in annex **??** p.**??**. Then, we need to define a module where the type parameter *W* corresponds to the *state* record type defined in script **??** p.**??**. Furthermore, We have to define the initial value of this type parameter which will correspond to an empty memory. This module, called *IdModP*, will be passed as a parameter to the modules we're going to work on later. It is defined in the the file *IdModPip.v* as follows :

Script 1.1: PIP state in the deep embedding

```
Require Import Pip_state.

Module IdModP <: IdModType.
  Definition Id := string.

  Definition W := state.

  Definition Loc_PI := valTyp_irrelevance.

  Definition BInit := {|
         currentPartition :=  defaultPage;
         memory:= @nil (paddr * value);
                    |}.
End IdModP.
```

1.3 1^{st} invariant and proof

1.3.1 Invariant in the shallow embedding

This invariant concerns a function named *getFstSadow*. We want to prove that if the necessary properties for the correct execution of this function are verified then any precondition on the state persists after its execution since this function doesn't change it. We also need to ascertain the validity of the returned value. This invariant is defined as follows :

Script 1.2: getFstShadow invariant in the shallow embedding

```
Lemma getFstShadow (partition : page) (P : state → Prop) :
{{fun s ⇒ P s ∧ partitionDescriptorEntry s ∧
          partition ∈ (getPartitions multiplexer s) }}
Internal.getFstShadow partition
{{fun (sh1 : page) (s : state) ⇒ P s ∧
          nextEntryIsPP partition sh1idx sh1 s }}.
```

where :

- **getFstShadow** : is a function that **returns the physical page of the first shadow** for a given partition. The index of the virtual address of the first shadow, called *sh1idx* as shown in annex **??** p.**??**, is predefined in PIP as the 4^{th} index of a partition. Furthermore, we know that the virtual address and the physical address of any page are consecutive. Therefore, we only need to fetch the predefined index of the first shadow, calculate its successor then read the corresponding page in the given partition. It is defined as follows :

  Script 1.3: getFstShadow function in the shallow embedding

  ```
  Definition getFstShadow (partition : page):=
    perform idx := getSh1idx in
    perform idxsucc := MALInternal.Index.succ idx in
    readPhysical partition idxsucc.
  ```

- **P** : is the propagated property on the state;

- **getPartitions :** is a function that returns the list of all sub-partitions of a given partition. In our case, since we give it the multiplexer partition which is the root partition, it returns all the partitions in the memory. This function is used to verify that the partition we give to the *getFstShadow* function is valid by checking its presence in the partition tree;

- **nextEntryIsPP :** returns *True* if the entry at position successor of the given index in the given table is a physical page and is equal to another given page. It is defined as follows :

Script 1.4: nextEntryIsPP property

```
Definition nextEntryIsPP table idxroot tableroot s : Prop:=
match Index.succ idxroot with
 | Some idxsucc =>
  match lookup table idxsucc (memory s) beqPage beqIndex with
   | Some (PP table) => tableroot = table
   |_ => False
  end
 | _ => False
end.
```

- **partitionDescriptorEntry :** defines some properties of the partition descriptor. All the predefined indexes in the file *PIPstate.v*, shown in annex **??** p.**??**, should be less than the table size minus one and contain virtual addresses. This is verified by the *isVA* property which returns *True* if the entry at the position of the given index in the given table is a virtual address and is equal to a given value. The successors of these indexes contain physical pages which should not be equal to the default page. This is verified by the *nextEntryIsPP* property. The *partitionDescriptorEntry* property is defined as follows :

Script 1.5: partitionDescriptorEntry property

```
Definition partitionDescriptorEntry s :=
∀ (partition: page),
 partition ∈ (getPartitions multiplexer s) →
 ∀ (idxroot : index),
  (idxroot = PDidx ∨ idxroot = sh1idx ∨
   idxroot = sh2idx ∨ idxroot = sh3idx ∨
   idxroot = PPRidx ∨ idxroot = PRidx) →
  idxroot < tableSize - 1 ∧ isVA partition idxroot s ∧
  ∃ entry, nextEntryIsPP partition idxroot entry s ∧
          entry ≠ defaultPage.
```

In the next sections we will rewrite the *getFstShadow* function as well as adapt these properties in order to prove this invariant in the deep embedding.

1.3.2 Modelling the *getFstShadow* function

We worked on several possible definitions in the deep embedding for the *getFstShadow* function, defined in script 1.3 p.3 taking into account the limitations of the deep embedding especially in dealing with inductive data types as it doesn't provide us with a pattern matching construct.

getSh1idx function

First, let's define *getSh1idx* which returns the value of *sh1idx*. In the deep embedding, we defined it as a deep index value of the predefined first shadow index :

Script 1.6: getSh1idx definition in the deep embedding

```
Definition getSh1idx : Exp := Val (cst index sh1idx).
```

Index successor function

Next, we need to implement the index successor function in the deep embedding. An index value has to be less then the preset table size. This property should be verified before calculating the successor. This function is defined as follows in the shallow embedding :

Script 1.7: Index successor function in the shallow embedding

```
Program Definition succ (n : index) : LLI index :=
 let isucc := n+1 in
 if (lt_dec isucc tableSize)
 then ret (Build_index isucc _)
 else  undefined 28.
```

Our approach is to rewrite this function in the deep embedding while leaving shallow bits that we progressively replace with deep definitions in order to make the shift from shallow proofs to deep ones gradual and modular. First, We rewrote this function so that we could use it in a *Modify* construct replacing the output type by *option index*. We used the index constructor *CIndex* to build the new index :

Script 1.8: Rewritten shallow index successor function

```
Definition succIndexInternal (idx:index) : option index :=
 let (i,_) := idx in
 if lt_dec i tableSize
 then Some (CIndex (i+1))
 else None.
```

Thus, the first version of the index successor function, called *Succ*, is defined as a *Modify* construct that uses a generic effect, called xf_succ, of input type *index* and output type *option index*, which calls the rewritten shallow function *succIndexInternal*. *Succ* is parametrised by the name of the input index variable which will be evaluated in the variable environment :

Script 1.9: Definition of Succ

```
Instance VT_index : ValTyp index.
Instance VT_option_index : ValTyp (option index).

Definition xf_succ : XFun index (option index) := {|
  b_mod := fun s (idx:index) ⇒ (s, succIndexInternal idx)
|}.

Definition Succ (x:Id) : Exp :=
  Modify index (option index) VT_index VT_option_index
                      xf_succ (Var x).
```

The second version of the successor function, called *SuccD*, is different from the former one. In this version we don't want to call the shallow function *succIndexInternal*. Instead, we are trying to devise a comparatively deeper definition of successor where the conditional structure is replaced by the deep construct *IfThenElse* and the assignments are defined using the *BindS* construct. The new version is defined as follows:

Script 1.10: Definition of SuccD

```
Definition SuccD (x:Id) :Exp :=
BindS "i" (prj1 x)
          (IfThenElse (LtDec "i" tableSize)
                      (Apply SomeCindexQF
                             (PS[SuccR "i"])
                      )
                      (Val(cst (option index) None))
          ).
```

where *prj1* is a projection of the first value of an index record which corresponds to the actual value of the index, *LtDec* is the definition of the comparison function using its shallow version, *SomeCindexQF* is a quasi-function that lifts a natural number to an *option index* typed value using the shallow constructors *Cindex* and *Some* successively and *SuccR* is a function that calculates the successor of a natural number. It is important to note that *SomeCindex* is defined as a *Modify* construct using a generic effect instead of a pure deep function since the deep embedding doesn't provide a pattern matching construct to deal with such cases. Their formal definitions in Coq are as follows :

Script 1.11: Functions called in SuccD

```
(* projection function *)
Definition xf_prj1 : XFun index nat := {|
   b_mod := fun s (idx:index) => (s,let (i,_) := idx in i)
|}.
Definition prj1 (x:Id) : Exp :=
  Modify index nat VT_index VT_nat xf_prj1 (Var x).

(* comparision function *)
Definition xf_LtDec (n: nat) : XFun nat bool := {|
   b_mod := fun s i ⇒ (s,if lt_dec i n then true else false)
|}.
Definition LtDec (x:Id) (n:nat): Exp :=
  Modify nat bool VT_nat VT_bool (xf_LtDec n) (Var x).

(* lifting funtion *)
Definition xf_SomeCindex : XFun nat (option index) := {|
   b_mod := fun s i ⇒ (s,Some (CIndex i))
|}.
Definition SomeCindex (x:Id) : Exp :=
  Modify nat (option index) VT_nat VT_option_index
             xf_SomeCindex (Var x).
Definition SomeCindexQF := QF
(FC emptyE [("i",Nat)] (SomeCindex "i")
    (Val (cst (option index) None)) "SomeCindex" 0).

(* successor function for natural numbers *)
Definition xf_SuccD : XFun nat nat := {|
   b_mod := fun s i => (s,S i)
|}.
Definition SuccR (x:Id) : Exp :=
  Modify nat nat VT_nat VT_nat xf_SuccD (Var x).
```

The last version calls a recursive function *plusR* that calculates the sum of two natural numbers. More precisely, we replace the call of *SuccR* in the former definition with *plusR 1* which adds one to its given parameter. *plusR* and the new successor function, called *SuccRec*, are defined as follows :

Script 1.12: Definition of PlusR

```
Definition plusR' (f: Id) (x:Id) : Exp :=
      Apply (FVar f) (PS [VLift (Var x)]).

Definition plusR (n:nat) := QF
(FC emptyE [("i",Nat)] (VLift(Var "i"))
    (BindS "p" (plusR' "plusR" "i") (SuccR "p")) "plusR" n).
```

Script 1.13: Definition of SuccRec

```
Definition SuccRec (x:Id) :Exp :=
BindS "i" (prj1 x)
          (IfThenElse (LtDec "i" tableSize)
                      (Apply SomeCindexQF
                             (PS[Apply (plusR 1)
                                       (PS[VLift(Var "i")])
                                ])
                      )
                      (Val(cst (option index) None))
          ).
```

readPhysical function

Now, we need to define the function that will read the physical page in the given index. This function, called *readPhysical* in the shallow embedding, uses the predefined lookup function that returns the value mapped to the address-index pair we give it. As shown in script 1.14, *readPhysical* checks whether the read page is actually a physical page by performing a match on the returned entry. *beqPage* and *beqIndex* are comparison functions respectively for pages and indexes.

Script 1.14: readPhysical function in the shallow embedding

```
Definition readPhysical (paddr: page) (idx: index) : LLI page:=
 perform s := get in
 let entry := lookup paddr idx s.(memory) beqPage beqIndex in
 match entry with
  | Some (PP a) ⇒ ret a
  | Some _ ⇒ undefined 5
  | None ⇒ undefined 4
 end.
```

To implement this function in the deep embedding, we first copied all the predefined association list functions in a file we named *Liv.v*, as shown in annex **??** p.**??**. We then rewrote this function so that we could use it in a *Modify* construct replacing the output type by *option page* as follows :

Script 1.15: Rewritten shallow readPhysical function

```
Definition readPhysicalInternal p i memory :option page :=
 match (lookup p i memory beqPage beqIndex) with
  | Some (PP a) ⇒ Some a
  | _ ⇒ None
 end.
```

The function is called *ReadPhysical* in the deep embedding and is parametrised by the name of the *option index* typed variable. *ReadPhysical* performs a match on its input to verify that its a valid index then naturally calls *readPhysicalInternal*. It is defined as follows :

Script 1.16: Definition of ReadPhysical

```
Instance VT_option_page : ValTyp (option page).

Definition xf_read (p: page): XFun (option index) (option page) :=
{| b_mod := fun s oi => (s,match oi with
        |None => None
        |Some i => readPhysicalInternal p i (memory s) end) |}.

Definition ReadPhysical (p:page) (x:Id) : Exp :=
  Modify (option index) (option page)
         VT_option_index VT_option_page
         (xf_read p) (Var x).
```

Using these definitions we are going to define three versions of *getFstShadow* in the deep embedding, each calling a different version of the index successor function. These functions are named *getFstShadowBind*, *getFstShadowBindDeep* and *getFstShadowBindDeepRec* and respectively call Succ, SuccD and SuccRec. Since their definitions are same, we will only give the definition of *getFstShadowBind* :

Script 1.17: Definition of getFstShadowBind

```
Definition getFstShadowBind (p:page) : Exp :=
 BindS "x" getSh1idx
           (BindS "y" (Succ "x")
                      (ReadPhysical p "y")
           ).
```

1.3.3 Invariant in the deep embedding

To model this invariant in the deep embedding we will use the Hoare triple we defined in section ?? p.??. However, we need to adapt some of the properties mentioned in section 1.3.1 p.2. In particular, the property *nextEntryIsPP*, defined in script 1.4 p.4, needs to be parametrised by the resulting deep value. So the page we need to compare is of type *Value* instead of *page*. the comparison between the fetched and given value now becomes a comparison between two deep values and not shallow ones. Also, to calculate the successor of the given index we use the rewritten successor function *succIndexInternal*, defined in script 1.8 p.5. Also, when we call this property in *partitionDescriptorEntry*, defined in script 1.5 p.4, we need to lift the page to an *option page* typed deep value. This property is now defined as follows :

Script 1.18: Rewritten nextEntryIsPP property

```
Definition nextEntryIsPP (p:page) (idx:index) (p':Value) (s:W) :=
match succIndexInternal idx with
 | Some i =>
  match lookup p i (memory s) beqPage beqIndex with
   | Some (PP table) => p' = cst (option page) (Some table)
   |_ => False
  end
 | _ => False
end.
```

Finally, we can write the deep Hoare triple which is not only parametrised by the partition and the propagated property but also by the function environment as well as the variable environment we want to evaluate the *getFstShadow* function in. It is formally defined in Coq as follows :

Script 1.19: getFstShadow invariant definition

```
Lemma getFstShadowBindH (partition : page) (P : W -> Prop)
                        (fenv: funEnv) (env: valEnv) :
{{fun s ⇒ P s ∧ partitionDescriptorEntry s ∧
           partition ∈ (getPartitions multiplexer s)}}
fenv >> env >> (getFstShadowBind partition)
{{fun sh1 s ⇒ P s ∧ nextEntryIsPP partition sh1idx sh1 s}}.
```

Naturally, since we defined three *getFstShadow* functions, each calling a different version of the deep index successor function, we only need to specify the name of version of *getFstShadow* we want to call. In this case we are calling *getFstShadowBind* defined in script1.17 p.9.

1.3.4 Invariant proof

Script 1.20: proof of the getFstShadow invariant

```
Proof.
unfold getFstShadowBind. (* or other called function *)
eapply BindS_VHTT1.
eapply getSh1idxWp.
simpl; intros.
eapply BindS_VHTT1.
eapply weakenEval.
eapply succWp. (* or other Lemma for called function *)
simpl; intros; intuition.
instantiate (1:=(fun s => P s ∧ partitionDescriptorEntry s ∧
                  partition ∈ (getPartitions multiplexer s))).
simpl. intuition. instantiate (1:=sh1idx).
eapply H0 in H3.
specialize H3 with sh1idx.
eapply H3. auto. auto.
simpl; intros.
eapply weakenEval.
eapply readPhysicalW.
simpl;intros; intuition.
destruct H3. exists x.
unfold partitionDescriptorEntry in H1.
apply H1 with partition sh1idx in H4.
clear H1; intuition.
destruct H5. exists x0. intuition.
unfold nextEntryIsPP in H4.
unfold readPhysicalInternal; subst.
inversion H2.
repeat apply inj_pair2 in H3.
unfold nextEntryIsPP in H5.
rewrite H3 in H5.
destruct (lookup partition x (memory s) beqPage beqIndex).
unfold cst in H5.
destruct v0;try contradiction.
apply inj_pairT2 in H5.
inversion H5. auto.
unfold isVA in H4.
destruct (lookup partition sh1idx (memory s) beqPage beqIndex)
in H2; try contradiction. auto.
Qed.
```

As shown in the previous script, we start the proof by evaluating the first assignment using the assignment rule *BindS_VHTT1* defined in section **??** p.**??**. This implies evaluating *getSh1idx* and mapping its resulting value to x in the variable environment then evaluating the rest of the function in this updated environment. To evaluate *getSh1idx*, we use a lemma that we have proven called *getSh1idxWp*. This lemma also propagates any property on the state.

Script 1.21: getSh1idxWp lemma definition and proof

```
Lemma getSh1idxW (P: Value -> W -> Prop)
                 (fenv: funEnv) (env: valEnv) :
  {{wp P fenv env getSh1idx}} fenv >> env >> getSh1idx {{P}}.
Proof.
(* the weakest precondition is a precondition *)
apply wpIsPrecondition.
Qed.

Lemma getSh1idxWp P fenv env :
{{P}} fenv >> env >> getSh1idx
{{fun (idxSh1 : Value) (s : state) ⇒ P s
            ∧ idxSh1 = cst index sh1idx }}.
Proof.
eapply weakenEval. (* weakening precondition *)
eapply getSh1idxW.
intros.
unfold wp.
intros.
unfold getSh1idx in X.
inversion X;subst.
auto.
inversion X0.
Qed.
```

In script 1.3.1, two lines change in the the proof for the different *getFstShadow* functions. Indeed, in the first line, we need to adapt the name of the unfolded function according to the one we call in the Hoare triple definition. Later, we call the assignment rule again and we need to evaluate the successor function which is different in each *getFstShadow* definition. Therefore, we need to define a lemma for each version and call it when needed in the 8^{th} line. The version which corresponds to our case is defined and proven as follows :

Script 1.22: succWp lemma definition and proof

```
1  Lemma succWp (x:Id) (v:Value) P (fenv: funEnv) (env: valEnv) :
2   ∀ (idx:index),
3   {{fun s ⇒ P s ∧ idx < tableSize - 1 ∧ v=cst index idx}}
4    fenv >> (x,v)::env >> Succ x (* or other successor function *)
5   {{fun (idxsuc : Value) (s : state) ⇒ P s ∧
6            idxsuc = cst (option index) (succIndexInternal idx) ∧
7            ∃ i, idxsuc = cst (option index) (Some i)}}.
8  Proof.
9  intros.
10 eapply weakenEval.
11 eapply succW. (* or other lemma for called function *)
12 intros.
13 simpl.
14 split.
15 instantiate (1:=idx).
16 intuition.
17 intros.
18 intuition.
19 destruct idx.
20 exists (CIndex (i + 1)).
21 f_equal.
22 unfold succIndexInternal.
23 case_eq (lt_dec i tableSize).
24 intros.
25 auto.
26 intros.
27 contradiction.
28 Qed.
```

This lemma proves that we get a valid index after executing successor since the precondition assures its correct execution. Only the 11^{th} line of the proof changes according the the called successor function. The lemma defined in the 11^{th} line proves that the resulting value of the execution of the successor function is equal the the value we get when we apply the shallow *succIndexInternal* function to the same given index. The proof of this evaluation lemma is quite different between the various versions which is logical since evaluating a *Modify* that calls the shallow *succIndexInternal* function is different from evaluating all the deep constructs in the deep and recursive versions of the successor function.

The proof of the first lemma, called *succW*, which evaluates the *Succ* function defined in script 1.9 p.6, goes naturally by inversion on the closure. It is defined and proven as follows :

Script 1.23: succW Lemma definition and proof

```
Lemma succW  (x : Id) (P: Value → W → Prop) (v:Value)
             (fenv: funEnv) (env: valEnv) :
∀ (idx:index),
 {{fun s ⇒ idx < (tableSize -1) ∧ ∀ l : idx + 1 < tableSize,
    P (cst (option index) (succIndexInternal idx)) s ∧
    v = cst index idx }}
  fenv >> (x,v)::env >> Succ x {{ P }}.
Proof.
intros.
unfold THoareTriple_Eval; intros; intuition.
destruct H1 as [H1 H1'].
omega.
inversion X;subst.
inversion X0;subst.
repeat apply inj_pair2 in H7;subst.
inversion X2;subst.
inversion X3;subst.
inversion H;subst.
destruct IdModP.IdEqDec in H3.
inversion H3;subst.
clear H3 e X3 H XF1.
inversion X1;subst.
inversion X3;subst.
repeat apply inj_pair2 in H7.
repeat apply inj_pair2 in H9. subst.
unfold b_exec,b_eval,xf_succ,b_mod in *.
simpl in *.
inversion X4;subst.
apply H1.
inversion X5.
inversion X5.
contradiction.
Qed.
```

The proof of the second lemma, called *succDW*, which evaluates the *SuccD* function defined in script **??** p.**??**, is quite longer than the previous one. Indeed, we need to proceed by inversion on the evaluation of every deep construct. This lemma is defined and proven in annex **??** p.**??**.

For The third lemma, which evaluates the *SuccRec* function defined in script **??** p.**??**, we did two different proofs : one using only inversions, called *succRecWByInversion*, and the other using the predefined Hoare triple rules mentioned in section **??** p.**??**, called *succRecW*. The former takes about 480 lines to prove while the latter takes approximatively 350 lines wich amounts to 130 lines less. Furthermore, the proof of the lemma *succRecW*, which uses Hoare triple rules seems more organised since we deal with the poof of each instruction at a time and not the function as a whole.

Finally, all what is left in the main proof is to evaluate the last function *ReadPhysical* and prove the implication between properties. To that end, we defined the following lemma, called *readPhysicalW* and proven in annex **??** p.**??**, as follows :

Script 1.24: readPhysicalW Lemma definition and proof

```
Lemma readPhysicalW (y:Id) table (v:Value)
      (P' : Value → W → Prop) (fenv: funEnv) (env: valEnv) :
 {{fun s ⇒ ∃ idxsucc p1, v = cst (option index) (Some idxsucc)
   ∧ readPhysicalInternal table idxsucc (memory s) = Some p1
   ∧ P' (cst (option page) (Some p1)) s}}
 fenv >> (y,v)::env >> ReadPhysical table y {{P'}}.
```

1.3.5 The Apply approach

For this approach, we chose to replace assignments in the *getFstShadow* function with function applications. For simplicity's sake, We chose to use the first version of the index successor function called *Succ* defined in script 1.9 p.6. However, considering our modular approach for its definition, we could easily replace it with its other versions and use the lemmas we devised for them in the new proof. To the *Apply* construct, we need to lift both *SuccD* and *readPhysical*, defined in script 1.16 p.9, to quasi-functions as follows :

Script 1.25: Lifting Succ and readPhysical to quasi-functions

```
Definition SuccQF := QF (FC emptyE [("y",indexType)]
 (Succ "y") (Val (cst (option index) None)) "Succ" 0).

Definition ReadPhysicalQF (p:page) := QF (FC
 emptyE [("x",optionIndexType)] (ReadPhysical p "x")
 (Val (cst (option page) None)) "ReadPhysical" 0).
```

The new version of the *getFstSadow* function, called *getFstShadowApply* is defined as follows :

Script 1.26: getFstShadowApply definition

```
Definition getFstShadowApply (p:page) :Exp :=
Apply (ReadPhysicalQF p) (PS [
                              Apply SuccQF (PS [getSh1idx])
                          ]).
```

For the proof, We didn't need to define any additional lemma about the intermediate functions. This is due to the use of the Hoare triple rules. The new invariant, called *getFstShadowApplyH'*, is defined and proven in annex **??** p.**??**.

1.4 2$^{\text{nd}}$ invariant and proof

This new invariant is about a function called *writeVirtual* that writes a virtual adress in the memory. We want to prove that this function verifies all of PIP's properties which include memory isolation, vertical sharing, kernel data isolation and consistency properties, mentioned in section **??** p.**??**. In the shallow embedding, *writeVirtual* is defined as follows :

Script 1.27: writeVirtual function in the shallow embedding

```
Definition writeVirtual (paddr: page) (idx: index)
                        (va: vaddr) :LLI unit :=
 modify (fun s ⇒ {|
  currentPartition := s.(currentPartition);
  memory := add paddr idx (VA va) s.(memory) beqPage beqIndex
 |}).
```

First we rewrote this function to act directly on the state as follows :

Script 1.28: Rewritten shallow writeVirtual function

```
Definition writeVirtualInternal (p:page) (i:index) (v:vaddr) :=
 fun s ⇒ {|
  currentPartition := s.(currentPartition);
  memory := add p i (VA v) s.(memory) beqPage beqIndex |}.
```

It is clear that the *writeVirtual* function is purely a generic effect. Its output value is irrelevant so its output type is declared as *unit*. We only need to call the *writeVirtualInternal* function in a generic effect that will be used in a Modify construct to define the deep version, called *WriteVirtual*, as follows :

Script 1.29: WriteVirtual definition

```
Definition xf_writeVirtual (p: page) (i: index) (v: vaddr)
                                : XFun unit unit :=
 {| b_mod := fun s _ ⇒ (writeVirtualInternal p i v s,tt) |}.

Definition WriteVirtual (p: page) (i: index) (v: vaddr) : Exp :=
  Modify unit unit VT_unit VT_unit (xf_writeVirtual p i v)
                  (QV (cst unit tt)).
```

To check that our implementation is correct, we first focused on a simpler invariant which asserts a relevant property of the *writeVirtual* function. This property is included in the postcondition of the main invariant we want to prove. Indeed, when we write a virtual address v in the page p in the memory, at a certain position, and when we instantly read the page p at the exact same position we should get the same value v. This Lemma is defined and proven as follows :

Script 1.30: writeVirtualInvNewProp invariant definition

```
Lemma writeVirtualInvNewProp (p : page) (i:index) (v:vaddr)
                                (fenv: funEnv) (env: valEnv) :
 {{fun _ ⇒ True}}
  fenv >> env >> WriteVirtual p i v
 {{fun _ s ⇒ readVirtualInternal p i s.(memory) = Some v}}.
Proof.
unfold THoareTriple_Eval; intros.
clear H k3 t k2 k1 tenv ftenv.
inversion X;subst.
inversion X0;subst.
repeat apply inj_pair2 in H5.
apply inj_pair2 in H7.
subst.
unfold b_eval, b_exec, xf_writeVirtual, b_mod in *.
simpl in *.
inversion X1;subst.
unfold writeVirtualInternal;simpl.
unfold add.
unfold readVirtualInternal;simpl.
specialize beqPairsTrue with p i p i.
intros;intuition.
rewrite H. reflexivity.
inversion X2.
inversion X2.
Qed.
```

To define the main invariant, we copied all the definitions of PIP's propagated properties as well as PIP's internal and dependant-type lemmas respectively in the files *Pip_Prop.v* *Pip_InternalLemmas.v* and *Pip_DependentTypeLemmas.v*. Then we defined and prooved the following lemma about the *writeVirtual* function :

Script 1.31: writeVirtualWp lemma definition and proof

```
Lemma writeVirtualWp (p: page) (idx: index) (vad: vaddr)
      (P: Value → state → Prop) (fenv: funEnv) (env: valEnv) :
 {{fun  s ⇒ P (cst unit tt) {|
   currentPartition := currentPartition s;
   memory := add p idx (VA vad) (memory s) beqPage beqIndex |} }}
 fenv >> env >> WriteVirtual table idx addr  {{P}}.
Proof.
unfold THoareTriple_Eval.
intros.
inversion X;subst.
inversion X0;subst.
repeat apply inj_pair2 in H6.
repeat apply inj_pair2 in H8.
subst.
unfold xf_writeVirtual, b_eval, b_exec, b_mod in *.
simpl in *.
inversion X1;subst.
auto.
inversion X2.
inversion X2.
Qed.
```

Finally we use the lemma above to prove the main invariant. As expected, the deep proof is practically identical to the shallow proof since we're weakening the Hoare triple first to use the *writeVirtualWp* lemma then we're proving a direct implication between properties on the state. The main invariant, called *writeVirtualInv*, is defined and proven in annex **??** p.**??**.

1.5 3$^{\text{rd}}$ invariant and proof

The third invariant is about a recursive function of PIP called *initVAddrTable* that initializes virtual addresses of a given table to the default value *defaultVAddr* defined in annex refstateFile p.**??**. This function is defined as follows in the shallow embedding :

Script 1.32: initVAddrTable in the shallow embedding

```
Fixpoint initVAddrTableAux timeout shadow2 idx :=
  match timeout with
  | 0 => ret tt
  | S timeout1 =>
    perform maxindex := getMaxIndex in
    perform res := MALInternal.Index.ltb idx maxindex in
    if (res)
    then
      perform defaultVAddr := getDefaultVAddr in
      writeVirtual shadow2 idx defaultVAddr;;
      perform nextIdx :=  MALInternal.Index.succ idx in
      initVAddrTableAux timeout1 shadow2 nextIdx
    else
      perform defaultVAddr := getDefaultVAddr in
      writeVirtual shadow2 idx defaultVAddr
  end.

(* Specifies the timeout of  initVAddrTableAux *)
Definition initVAddrTable sh2 n :=
  initVAddrTableAux tableSize sh2 n.
```

where :

- **getMaxIndex :** returns the value of the maximum index which equal to the table size minus one, knowing that the table size is different than 0. We wrote the following simplified definition to use the deep embedding :

  Script 1.33: Maximum index

  ```
  Axiom tableSizeNotZero : tableSize <> 0.
  Definition maxIndex : index := CIndex(tableSize-1).
  ```

- **succ :** is the index successor function defined in script 1.7 p.5. We will replace it with its deep implementation *SuccD* defined in script 1.10 p.6;

- **ltb :** is a comparison function for indexes similar to the mathematical comparison operator $<$ for natural numbers. In the deep embedding, we will define this function by calling its shallow version in a generic effect as follows :

Script 1.34: LtLtb definition

```
Definition xf_Ltb (i:index) : XFun index bool :=
 {| b_mod := fun s idx => (s,Index.ltb idx i) |}.

Definition LtLtb (x:Id) (i:index) : Exp :=
  Modify index bool VT_index VT_bool (xf_Ltb i) (Var x).
```

- **writeVirtual :** is the function, defined in script 1.27 p.16, that writes a virtual address in the memory. We couldn't use its deep implementation, called *WriteVirtual*, defined in script 1.29 p17 because we're not working directly with the value of the given index but with a variable which we need to evaluate in the variable environment. The new version of this function called *writeVirtual'* is defined as follows :

Script 1.35: LtLtb definition

```
Definition xf_writeVirtual' (p:page) (v:vaddr) :XFun index unit:=
  {| b_mod := fun s i => (writeVirtualInternal p i v s,tt) |}.

Definition WriteVirtual' (p:page) (i:Id) (v:vaddr) :Exp :=
 Modify index unit VT_index VT_unit (xf_writeVirtual' p v) (Var i).
```

We also need to define *ExtractIndex* which extracts an index from an *option index* typed value by performing a pattern matching. This is only possible by using a generic effect as follows :

Script 1.36: ExtractIndex definition

```
Definition xf_ExtractIndex : XFun (option index) index :=
{| b_mod := fun s idx => (s,match idx with
                            |Some i => i
                            |_ => index_d end) |}.

Definition ExtractIndex (x:Id):=
  Modify (option index) index VT_option_index VT_index
          xf_ExtractIndex (Var x).
```

To simplify the proof and avoid repetition, we will place the instruction that calls *WriteVirtual'* before the conditional structure. *initVAddrTableAux* and *initVAddrTable* are defined as follows :

Script 1.37: initVAddrTable definition in the deep embedding

```
Definition initVAddrTableAux (f i: Id) (p:page) : Exp :=
BindN (WriteVirtual' p i defaultVAddr)
      (IfThenElse (LtLtb i maxIndex)
                  (BindS "y" (BindS "idx" (SuccD i)
                                          (ExtractIndex "idx"))
                             (Apply (FVar f)
                                    (PS [VLift(Var "y")])))
                  (Val (cst unit tt))).

Definition initVAddrTable (p:page) (i:index) :=
Apply (QF (FC emptyE [("x",Index)] (Val (cst unit tt))
          (initVAddrTableAux "initVAddrTable" "x" p)
          "initVAddrTable" tableSize) (PS[Val (cst index i)]).
```

The invariant we want to prove, called *initVAddrTableNewProperty*, is defined in script 1.38. The proof is done by induction on the bound. More precisely, we suppose that the Hoare triple is valid for the bound n and we need to prove it's valid for $S\ n$. We mainly used the Hoare triple rules defined in section **??** p.**??** to evaluate each deep construct to get to the next function application where n is the bound which we have as a hypothesis. The main difficulty was to get to the exact same expression as well as the same function and variable environments as we have in the hypothesis without unfolding the Hoare triple. Indeed, if we unfold the Hoare triple at some point earlier, we have to reason by inversion on large expressions and we may encounter some typing problems which makes the proof much more complicated. It is also important to note that we defined a new lemma about the successor function, called *succWp'*, similar to the *succWp* lemma defined in script **??** p**??**, in which we propagate the specification of the value added to the environment. It is important to note that we used *succDW*, the evaluation lemma for the index successor funtion defined and proven in annex **??** p.**??**, which reinforces the importance of our modular approach. *succWp'* and the proof of the *initVAddrTableNewProperty* lemma is detailed in annex **??** p.**??**.

Script 1.38: initVAddrTableNewProperty invariant in the deep embedding

```
Lemma initVAddrTableNewProperty table (curidx : index)
                  (fenv: funEnv) (env: valEnv) :
 {{ fun s ⇒ (∀ idx : index, idx < curidx →
     (readVirtual table idx (memory s) = Some defaultVAddr) )}}
  fenv >> env >> initVAddrTable table curidx
 {{fun _ s ⇒ ∀ idx, readVirtual table idx s.(memory) =
                    Some defaultVAddr }}.
```

1.6 Observations