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. Proving invariants in the deep embedding

1.1 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 2.1.4 p.11 are replicated in the file *PIP_state.v*. All the axioms, constructors, comparison functions as well as predefined values were also replicated in this file as shown in annex A.1 p.45. Then, we need to define a module where the type parameter W corresponds to the *state* record type defined in script 2.1 p.11. Furthermore, We have to 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: Replicating the 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.
```

1.2 1st invariant and proof

1.2.1 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 A.1 p.45, is predefined in PIP as the 4th 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 into 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 A.1 p.45, 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 into 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.

1.2.2 Modelling the *getFstShadow* function

We worked on several possible definitions for this function in the deep embedding, each corresponding to a certain approach we wanted to further look into. 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: Index of the first shadow in the deep embedding

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

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.
```

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 an effect handler, 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 two. In this version we don't want to call the shallow function *succIndexInternal*. Instead, we are trying to devise a practically deep 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 constructors *Cindex* and *Some* successively and *SuccR* is a function that calculates the successor of a natural number. 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))
          ).
```

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 3.14, *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 A.2 p.47. 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* permorms 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")
           ).
```

1.2.3 Invariant in the deep embedding

To model this invariant in the deep embedding we will use the Hoare triple we defined in section 2.3.3 p.25. However, we need to adapt some of the properties mentioned in section 3.2.1 p.30. In particular, the property *nextEntryIsPP*, defined in script 3.4 p.31, 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 ?? p.??. Also, when we call this property in *partitionDescriptorEntry*, defined in script ?? p.??, we need to lift the page to an *option page* typed deep value. This property is now defined as follows :

Script 1.18: Definition of getFstShadowBind

```
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: Definition of getFstShadowBind

```
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 script3.17 p.36.

1.2.4 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 2.3.3 p.2.3.3. 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 3.2.1, two lines change in the the proof for the different *getFst-Shadow* 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 8th line. The version which corresponds to our case is defined and proven as follows :

Script 1.22: getSh1idxWp 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 11th line of the proof changes according the the called successor function. The lemma defined in the 11th 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 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 A.3 p.pagerefgetFstFile. For The third lemma, which evaluates the *SuccRec* function defined in script **??** p.**??**, we did two different proofs : one using only inversions and the other using the predefined Hoare triple rules mentioned in section 2.3.3 p.25.

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 and proved the lemma *readPhysicalW* 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'}}.
Proof.
intros.
unfold THoareTriple_Eval.
intros.
intuition.
destruct H.
destruct H.
intuition.
inversion H0;subst.
clear k3 t k2 k1 ftenv tenv H1.
inversion X;subst.
inversion X0;subst.
repeat apply inj_pair2 in H7.
subst.
inversion X2;subst.
inversion X3;subst.
inversion H0;subst.
destruct IdEqDec in H3.
inversion H3;subst.
clear H3 e X3 H0 XF1.
inversion X0;subst.
repeat apply inj_pair2 in H7.
repeat apply inj_pair2 in H11.
subst.
inversion X1;subst.
inversion X4;subst.
repeat apply inj_pair2 in H7.
```

```
apply inj_pair2 in H9.
subst.
unfold xf_read at 2 in X4.
unfold b_eval,b_exec,b_mod in X4.
simpl in *.
rewrite H in X4.
unfold xf_read,b_eval,b_exec,b_mod in X5.
simpl in *.
rewrite H in X5.
inversion X5;subst.
auto.
inversion X6.
inversion X6.
contradiction.
Qed.
```

1.3 2nd invariant and proof

1.4 3rd invariant and proof

1.5 Observations