Formalising Executable Specifications of Low-Level Systems

Paolo Torrini, David Nowak, Narjes Jomaa, and Mohamed Sami Cherif

CRIStAL, CNRS & University of Lille, France
{p.torrini,d.nowak,n.jomaa}@univ-lille.fr,mohamedsami.cherif@yahoo.com

Abstract. Formal models of low-level applications rely often on the distinction between executable layer and underlying hardware abstraction. This is also the case for the model of Pip, a separation kernel formalised and verified in Coq using a shallow embedding. DEC is a deeply embedded imperative typed language with primitive recursion and specified in terms of small-step semantics, which we developed in Coq as a reified counterpart of the shallow embedding used for Pip. In this paper, we introduce DEC and its semantics, we present its interpreter based on the type soundness proof and extracted to Haskell, we introduce a Hoare logic to reason about DEC code, and we use this logic to verify properties of Pip as a case study, comparing the new proofs with those based on the shallow embedding. Notably DEC can import shallow specifications as external functions, thus allowing for reuse of the abstract hardware model.[1]

1 Introduction

Formal modelling and verification of OS kernels involve different aspects of theorem proving: realistic modelling of low-level systems, scalable verification of program behaviour with respect to abstract specifications, executable models, generation of efficient, certified low-level code. Models have often complex structures in terms of components and levels of abstraction [2,3,4]. A natural distinction arises between the mathematical modelling of low-level requirements, typically associated with an abstract model of the platform, and the executable model of the platform-independent application which we also call the service layer. Primarily, the abstract model needs to be extensible with respect to concrete models of specific architectures, whereas the executable model needs to be translated to an efficient implementation language. Working with a theorem prover such as Coq [5] or Isabelle [6], this is a difference that matters for the choice of the representation in the base language.

A deep embedding of an object language captures its abstract syntax in terms of abstract datatypes, therefore providing a reified representation that supports manipulation, notably translations, as well as operational specifications of behaviour, thus allowing for a naturally executable characterisation

[1] DEC can be found at `https://github.com/2xs/dec.git` [1].

of control flow. However, reasoning about abstract datatypes involves a significant overhead in relation to the pervasive use of constructors and destructors. Moreover, conventional datatypes are not extensible. A shallow embedding consists of defining semantically the constructs of the object language in the base language, hence providing their characterisation in terms of denotational semantics, thus not only keeping the maths as simple as possible, but also allowing for extensibility in a non-problematic way. On the other hand, a shallow embedding does not provide any direct way to manipulate language constructs, and it makes it hard to separate object execution from evaluation in the base language.

Going back to our problem, we would generally like to associate the executable model with a deep embedding and the abstract platform model with a shallow one. Our approach consists in deeply embedding an object language that allows importing specifications written in the metalanguage as external function calls. We call the object language thus formalised a *deeply embedded language extension* (DLE). A DLE can be thought of as a domain specific extension of the base language, taking it closer to the target domain in the sense of syntax (i.e. with language constructs close to the instructions to be modelled) and of behavioural specification (i.e. with an operational semantics close to its model of execution).

2 Motivation

In this paper, we focus on the development of a specific DLE in connection with the formal development of the Pip protokernel [4,7,8,9]. Separation kernels [10] are systems designed to provably ensure noninterference properties with respect to distinct applications running on the same machine [3,2,11]. Usually a separation kernel is based on a formal model that is verified with respect to its security policy and translated to a low-level language for efficiency.

Pip is a separation kernel in which the kernel functionalities are reduced to a minimum needed to allow for efficient memory management and context switching (hence its characterisation as protokernel) [4]. It provides a service API allowing for partitions to be created, allocated memory and removed at runtime according to a hierarchical model. Partitions form a tree and each partition manages its own subpartitions. The security policy of Pip is based on three memory access properties: the parent partition can access the memory of its children (*vertical sharing*), sibling partitions cannot share memory with each other (*horizontal isolation*), and no partition can access kernel memory (*kernel isolation*). The Pip system is implemented in C and assembly, relying on a model written in Coq, the structure of which is shown in Fig. 1. The executable model, corresponding to the service API, is built on top of an abstract platform model that covers hardware abstraction layer (HAL) and hardware, the former in terms of the high-level specification of low-level, platform-specific C and assembly functions, the latter in terms of an abstract model of the physical memory and the MMU.

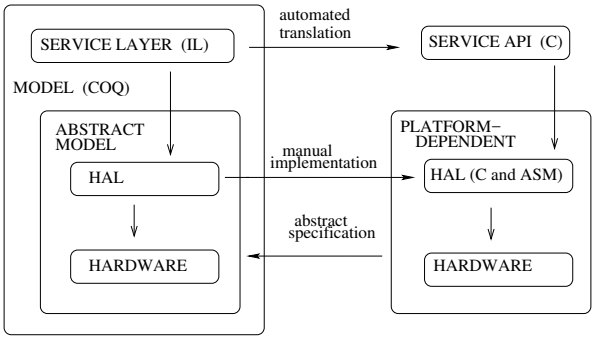

Fig. 1: The design of Pip: the system and its model

The service layer of Pip relies on a fragment of C that can be represented as a comparatively simple imperative language, one that is not difficult to capture in functional terms. This fragment, which we call IL here, corresponds to a typed first-order sequential language with call-by-value, primitive recursion and mutable references. Crucially, we do not need to return pointers, to use call by reference, structures or arrays. Non-termination can always be ruled out relying on hardware parameters.

The executable model has been formalised in Coq on top of hardware abstraction using a shallow embedding of IL based on its monadic semantics, and the verification of the security properties, presented in [4], has been carried out directly on the MC code using an associated Hoare logic, without going through a higher-level model of the service layer. The service layer is about 1300 lines of what we specifically call *monadic code* (MC), semantically corresponding to IL, while the HAL is about 300 lines of monadic specification. The verification involves rather long proofs [9] (several tens of thousands of lines), of which over ten thousands for the abstract platform model.

Modelling Pip at the shallow level has made it possible to focus on system development from the start, independent of any work on language development. However, translation to a low-level language is mandatory for efficiency. The closeness of IL to a fragment of C made it possible to carry out automatically the translation of MC to C source code. Such translation, defined on Gallina abstract syntax, has been implemented in Haskell [12] and it returns efficient, yet unverified code. Unfortunately, verifying code obtained in this way seems rather hard, as it would involve comparing semantically two large languages such as C and Gallina.

We would like to obtain a certified translation at a comparatively lower cost, by focusing on a smaller source language, building a reified representation of our object language, and by targeting an existing formalisation of C such as CompCert C [13], defining a verified translator in Coq. For this reason, we have developed in Coq a DLE that we call DEC [14,1], as an object-level counterpart of IL. As a deep embedding, DEC can be the source of a translation function

defined in Coq by pattern matching on the abstract syntax. Unlike MC, DEC has an interpreter based on its small-step operational semantics. This makes it possible to analyse the control flow in Coq, and it could help significantly in comparing formally the behaviour of a program with that of its translation to another language. Although in practice it is difficult to run the interpreter in Coq, it is possible to rely on the extraction mechanism to obtain an efficient program based specifically on the operational semantics of DEC rather than on generic Gallina evaluation.

Given DEC, there are indeed two distinct possibilities for the verification of Pip, as shown in Fig. 2. Both plans involve translating the MC model to DEC. Plan A consists additionally in proving the semantic equivalence between the two models, relying for the rest on the verification of the security properties in the shallow embedding as presented in [4]. Plan B, on the other hand, consists in verifying properties directly on the DEC model. In this case, given the large size of the existing verification based on MC, one of the main priorities is to maximise its reuse in verifying DEC code. This can be achieved particularly for the abstract platform model, relying on the DLE character of DEC. In this

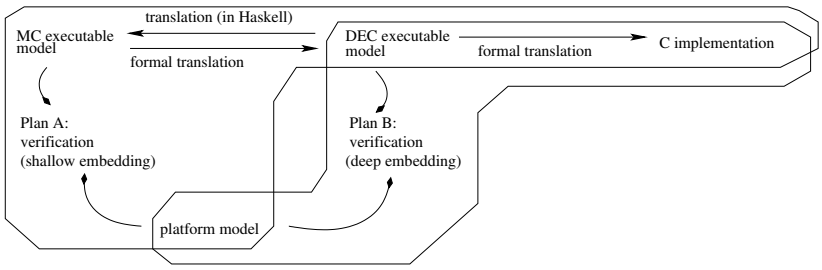

Fig. 2: Two verification plans: A and B

paper we focus on the distinctive part of plan B, i.e. on DEC and on verification in the deep embedding, omitting the translation to C (which is ongoing work). Our contribution (beyond the semiformal characterisation of IL) consists in two aspects that are essential to our verification approach: the development of DEC as DLE (ca. 10,000 lines code), and the development of an appropriate Hoare logic for DEC (ca. 2,000 lines code), allowing for syntax-driven, compositional proofs and reuse of the abstract platform model. We applied our approach to a case study (ca. 2,000 lines code on top of a significant amount of reuse), proving auxiliary invariants of Pip functions in their translation to DEC.

In section 3 we give a preliminary overview of IL, characterising it mathematically in terms of its operational semantics. In section 4 we present the development of DEC, with its interpreter (section 5) and the associated Hoare logic (section 6). Section 7 presents the verification of Pip invariants based on DEC code, and compares it with their verification based on MC. In section 8 we discuss related work, in section 9 conclusions and further work.

3 Preliminaries

We start with a semiformal, mathematical specification of IL, the language we use to illustrate succinctly the operational semantics DEC is based on, and to connect it with MC through a mathematical specification of its denotational semantics. IL is parametric in the primitive types Typ (ranged over by t), in the state type W, and in the actions Act, ranged over by a, associated with side effects and used to represent external functions. Values of primitive types are informally regarded as a set Val ranged over by v. Function types, ranged over by *ft*, are each defined by a tuple of primitive types for the parameters, and a primitive type for the return value. The syntax of IL is constituted of intrinsically well-typed expressions Exp and functions Fun (ranged over by e and f respectively). We rely on higher-order abstract syntax (HOAS), treating identifiers as formal variables and programs as closed terms, thus avoiding the need for environments. For brevity, we represent tuples as heterogeneous lists, which we treat as typed by type lists (denoted Typs), extending to them a standard list notation (including map). We use the Haskell convention for naming lists.

$$\begin{aligned}\mathsf{Exp}\ (t : \mathsf{Typ}) \ :=\ & \mathsf{val}\ t \mid \mathsf{cond}\ (\mathsf{Exp}\ \mathsf{Bool})\ (\mathsf{Exp}\ t)\ (\mathsf{Exp}\ t)\\ & \mid \mathsf{binds}\ (t' :\ \mathsf{Typ})\ (\mathsf{Exp}\ t')\ (t' \to \mathsf{Exp}\ t)\\ & \mid \mathsf{call}\ (ts :\ \mathsf{Typs})\ (\mathsf{Fun}\ t\ ts)\ (\mathsf{Exps}\ ts) \mid \mathsf{xcall}\ (ts :\ \mathsf{Typs})\ (\mathsf{Act}\ t\ ts)\ (\mathsf{Exps}\ ts)\end{aligned}$$

$$\mathsf{Exps}\ (ts : \mathsf{Typs}) \ :=\ \mathsf{map}\ \mathsf{Exp}\ ts$$

$$\mathsf{Fun}\ (t : \mathsf{Typ})\ (ts : \mathsf{Typs}) \ :=\ \mathsf{fun}\ (ts \to \mathsf{Exp}\ t)\ ((ts \to \mathsf{Exp}\ t) \to ts \to \mathsf{Exp}\ t)\ \mathsf{Nat}$$

$$\mathsf{Act}\ (t : \mathsf{Typ})\ (ts : \mathsf{Typs}) \ :=\ ts \to \mathsf{W} \to (\mathsf{W} * t)$$

We specify the small-step transition relation using configurations defined as pairs (s, X) where $s : \mathsf{W}$ is a state and X may be either an expression or a list of them. We make the presentation more concise by giving only the reduction rules and relying on evaluation contexts to specify call-by-value. Evaluation contexts allow us to compute the redex at each step. As usual [15], we write $[\_]$ to denote the hole in which to plug the redex in, and $C[e]$ to denote the splitting of an expression into context and redex.

$$\begin{aligned}\mathsf{Ctx} \ :=\ & [\_] \mid \mathsf{binds}\ \_\ \mathsf{Ctx}\ (\mathsf{Exp}\ \_) \mid \mathsf{binds}\ \_\ \mathsf{Val}\ \mathsf{Ctx} \mid \mathsf{cond}\ \mathsf{Ctx}\ (\mathsf{Exp}\ \_)\ (\mathsf{Exp}\ \_)\\ & \mid \mathsf{call}\ \_\ (\mathsf{Fun}\ \_\ \_)\ \mathsf{Ctxs} \mid \mathsf{xcall}\ \_\ (\mathsf{Act}\ \_\ \_)\ \mathsf{Ctxs}\\ \mathsf{Ctxs} \ :=\ & \mathsf{Ctx} :: (\mathsf{Exps}\ \_) \mid \mathsf{Val} :: \mathsf{Ctxs}\end{aligned}$$

$$\frac{\langle\ s \triangleright e\ \rangle \longrightarrow \langle\ s' \triangleright e'\ \rangle}{\langle\ s \triangleright C[e]\ \rangle \longrightarrow \langle\ s' \triangleright C[e']\ \rangle}$$

$$\langle\ s \triangleright \mathsf{binds}\ \_\ (\mathsf{val}\ v)\ e\ \rangle \longrightarrow \langle\ s \triangleright e\ v\ \rangle$$

$$\langle\ s \triangleright \mathsf{cond}\ (\mathsf{val}\ \mathsf{true})\ e_1\ e_2\ \rangle \longrightarrow \langle\ s \triangleright e_1\ \rangle$$

$$\langle\ s \triangleright \mathsf{cond}\ (\mathsf{val}\ \mathsf{false})\ e_1\ e_2\ \rangle \longrightarrow \langle\ s \triangleright e_2\ \rangle$$

$$\langle\ s \triangleright \mathsf{call}\ (\mathsf{fun}\ e_0\ e_1\ 0)\ (\mathsf{map}\ \mathsf{val}\ vs)\ \rangle \longrightarrow \langle\ s \triangleright e_0\ vs\ \rangle$$

$$\langle\ s \triangleright \mathsf{call}\ (\mathsf{fun}\ e_0\ e_1\ (S\ n))\ (\mathsf{map}\ \mathsf{val}\ vs)\ \rangle \longrightarrow \langle\ s \triangleright e_1\ (\mathsf{fun}\ e_0\ e_1\ n)\ vs\ \rangle$$

$$\langle\ s \triangleright \mathsf{xcall}\ ts\ a\ (\mathsf{map}\ \mathsf{val}\ vs)\ \rangle \longrightarrow \langle\ s' \triangleright \mathsf{val}\ v\ \rangle \qquad \text{where} \quad a\ vs\ s\ =\ (s', v)$$

Notice that val simply lifts values to expressions, and the final value can be obtained by unlifting from an expression of form val v. The denotational semantics of IL can be defined along the lines of the monadic translation in [16], using a state monad with state W (see Appendix A). The result corresponds to the shallow embedding used in the formalisation of Pip [4].

4 The Deep Embedding

DEC [1] is a strongly normalising, functional imperative language with primitive recursion, implemented in Coq as an DLE based on IL, parametric in the type of the mutable state. The constructs of DEC are internally specified as functional ones. Nonetheless, Coq functions can be imported as external functions, and these can be stateful, although the totality requirement of the metalanguage ensures that they are terminating. In this sense, DEC is a functional language that can be extended with generic effects, as well as a deeply embedded functional interface which can be used to extend a stateful model. Unlike the HOAS-style presentation of IL, DEC relies on environments, on explicit typing relations, and on a semantic representation which uses propagation rules rather than evaluation contexts, following an approach closer to [17] and to the original presentation of structural operational semantics (SOS) [18], a choice made to allow for explicit manipulation of identifiers without the need to implement α-renaming. As a distinctive computational feature, DEC has typing relations with inductive principles which are strong enough to carry the weight of the type soundness proof, while minimising type annotation.

Relying on Coq modules, the definition of DEC is parametric in the type of the mutable state `W` and in the type of the identifiers `Id`, the latter required to have decidable equality. We model environments as homogeneous lists, and to this purpose, unlike in HOAS, we need to introduce a deep embedding of object types and values. Our object types (i.e. *deep* types) are lifted Gallina types (i.e. *shallow* ones). Their type could be treated as trivial hiding, i.e. Σ`(λX:Type,X)`, but we prefer to rely on a type class `ValTyp:Type→Prop` to ensure lifting is explicitly allowed, hence defining our type `VTyp` of value types as Σ`ValTyp`, with associated lifting function `vtyp: Type → VTyp`. Deep values are defined by lifting shallow values, hiding their type, and their type `Value` is defined as Σ`ValueI`, where

```
Inductive ValueI (T: Type) : Type := Cst (v: T).
```

and lifting is `cst: ∀ T:Type, T → Value`. Value environments and value typing contexts are then given types `list(Id*Value)` and `list(Id*VTyp)`, respectively abbreviated as `valEnv` and `valTC`. The value typing relation `ValueTyping:Value→VTyp→Type` reduces to extracting and equating the shallow types of the two arguments, whereas the identifier typing relation `IdTyping:valTC→Value→VTyp→Type` relies on the application of the lookup function `findE`.

From the deep typing point of view, DEC is intended as a first-order language, therefore it would not be strictly necessary to allow for the body of program expressions to contain occurrences of function definitions, as opposed to function variables. However, allowing function definitions to be syntactical subexpressions leads to a stronger built-in induction principle. Partly for this reason, DEC is essentially designed as first-order fragment of a higher-order language. The deep type of function types

```
Inductive FTyp : Type := FT (prms_type: valTC) (ret_type: VTyp).
```

ensures the first-order restriction, requiring that parameters are deep datavalues. Function environments (`funEnv`) and function typing contexts (`funTC`) are defined as lists, in analogy to `valEnv` and `valTC`. The namespace distinction between value identifiers and function identifiers is enforced at the level of head normal forms, here called q-values and q-functions.

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

The inductive type of expressions is mutually defined with functions, q-functions, and lifted expression lists that represent parameters.

```
Inductive Fun : Type := FC (fenv: funEnv) (tenv: valTC)
                           (e0 e1: Exp) (x: Id) (n: nat)
with QFun : Type := FVar (x: Id) | QF (f: Fun)
with Exp : Type := Val (v: Value) | Return (q: QValue)
         | IfThenElse (e1 e2 e3: Exp)
         | BindN (e1 e2: Exp) | BindS (x: Id) (e1 e2: Exp)
         | BindMS (fenv: funEnv) (venv: valEnv) (e: Exp)
         | Apply (qf: QFun) (args: Prms)
         | Modify {T1 T2: Type} {VT1: ValTyp T1} {VT2: ValTyp T2}
                  (XF: XFun T1 T2) (arg: QValue)
with Prms : Type := PS (es: list Exp).
```

The function constructor `FC` represents an *iterate*-style construct, where `tenv` gives the list of the formal parameters with their types, `n` is a natural number that represents fuel, `e0` is the function body for `n=0`, `e1` is the function body for `n>0`, `x` is the function identifier used in recursive calls, and `fenv` is the local function environment. Constructors `Var` and `FVar` lift identifiers to the corresponding head-normal forms. Similarly `QV` and `QF` lift normal forms. `PS` lifts expression lists to parameters. Concerning expressions, `Val` and `Return` are lifting constructors, `IfThenElse` represents conditional branching, `BindN` sequencing and `BindS` local binding of identifiers to expressions (i.e. let-style binding). `BindMS` allows for multiple binding of identifiers to normal forms, i.e. for local environments, and it is needed for internal processing in our environment-based representation. `Apply` represents application of recursive functions. `Modify` represents application of external one-argument functions, where the function type is `T1→T2`. `Modify` works as a constructor of generic effects, handled by the stateful functions associated with the corresponding record of type

```
Record XFun (T1 T2: Type) : Type := { x_mod : W→T1→W*T2 ;
    x_exec : W→T1→W := λw x, fst (x_mod w x) ;
    x_eval : W→T1→T2 := λw x, snd (x_mod w x) }.
```

For example, generic read and write actions can be defined as follows

```
Definition xf_read {T: Type} (f: W → T) : XFun unit T  := {|
   x_mod := fun x _ ⇒ (x, f x)  |}.
Definition xf_write {T: Type} (f: T → W) : XFun T unit := {|
   x_mod := fun _ x ⇒ (f x, tt) |}.
```

Read and write instructions can then be defined, given `UnitVT:ValTyp unit` (here @ is used to make implicit arguments explicit).

```
Definition Read {T: Type} (VT: ValTyp T) (f: W → T) : Exp :=
  @Modify unit T UnitVT VT (xf_read f) (QV (cst unit tt)).
Definition Write {T: Type} (VT: ValTyp T) (f: T → W) (x: T) : Exp :=
  @Modify T unit VT UnitVT (xf_write f) (QV (cst T x)).
```

Notice that function definitions are meant to represent closed terms, as they may occur as subterms in expressions. For this reason, a function definition is defined as a closure with respect to its function identifiers, by including `fenv` as local function environment. This measure prevents variable capture and suffices to ensure we can type check recursive functions without annotating them with their return type.

The typing relations on expressions, functions, q-functions and parameters are defined by mutual induction, where `MatchEnvs` maps a binary relation over two lists.[2]

```
Inductive ExpTyping : funTC→valTC→funEnv→Exp→VTyp→Type := ...
  | Apply_Typing : ∀ (ftenv: funTC) (tenv fps: valTC) (fenv: funEnv)
                   (q: QFun) (ps: Prms) (pt: PTyp) (t: VTyp),
      pt = PT (map snd fps) → MatchEnvs FunTyping fenv ftenv →
      QFunTyping ftenv fenv q (FT fps t) →
      PrmsTyping ftenv tenv fenv ps pt →
      ExpTyping ftenv tenv fenv (Apply q ps) t
  | Modify_Typing : ∀ (ftenv: funTC) (tenv: valTC) (fenv: funEnv)
                    (T1 T2: Type) (VT1: ValTyp T1) (VT2: ValTyp T2)
                    (XF: XFun T1 T2) (q: QValue),
          QValueTyping tenv q (vtyp T1) →
          ExpTyping ftenv tenv fenv (@Modify T1 T2 VT1 VT2 XF q) (vtyp T2)
with QFunTyping : funTC→funEnv→QFun→FTyp→Type := ...
```

In the typing of function application, the type of the actual parameters is compared with that of the formal ones obtained from the q-function typing, which means either consulting the function typing context (`ftenv`) in case of an identifier, or else checking the function type. In the typing of external function calls the relevant types and the function definition are passed as a record.

Our function typing relation has a comparatively non-standard, algorithmic character.

```
with FunTyping : Fun→FTyp→Type :=
  | Fun0_Typing: ∀ (ftenv: funTC) (tenv: valTC) (fenv: funEnv)
                 (e0 e1: Exp) (x: Id) (t: VTyp),
      MatchEnvs FunTyping fenv ftenv →
      ExpTyping ftenv tenv fenv e0 t →
```

[2] Details in `2xs/dec/src/langspec/LangSpec.v` [14]

```
    FunTyping (FC fenv tenv e0 e1 x 0) (FT tenv t)
| FunS_Typing: ∀ (ftenv: funTC) (tenv: valTC) (fenv: funEnv)
               (e0 e1: Exp) (x: Id) (n: nat) (t: VTyp),
    let ftenv' := (x, FT tenv t) :: ftenv in
    let fenv' := (x, FC fenv tenv e0 e1 x n) :: fenv in
    MatchEnvs FunTyping fenv ftenv → ExpTyping ftenv' tenv fenv' e1 t →
    FunTyping (FC fenv tenv e0 e1 x n) (FT tenv t) →
    FunTyping (FC fenv tenv e0 e1 x (S n)) (FT tenv t)
```

Given a function `f := FC fenv tenv e0 e1 x n` to type, while the types of the parameters are supplied by `tenv`, the return type needs to be inferred, either from `e0` when `n = 0`, or else from `e1`. This involves also inferring the types of the local functions in `fenv`, not supplied by `f`. Hence the typing relation requires a function environment as argument, rather than just a function typing context, and given the function environment update in case of `n > 0`, type inference requires induction on the fuel.

We have developed our typing definitions in parallel with the proof of a type soundness theorem which in fact we carry out by mutual induction on the typing relations. However, the induction principle supplied automatically by Coq turned out to be weak, particularly given our use of lists to represent parameters and our typing of parameters

```
with PrmsTyping : funTC→valTC→funEnv→Prms→PTyp→Type :=
  | PS_Typing: ∀ (ftenv: funTC) (tenv: valTC) (fenv: funEnv)
                (es: list Exp) (ts: list VTyp),
          Forall2T (ExpTyping ftenv tenv fenv) es ts →
          PrmsTyping ftenv tenv fenv (PS es) (PT ts).
```

where `Forall2T` maps a relation on lists.

```
Inductive Forall2T {A B : Type} (R: A→B→Type): list A→list B→Type :=
  | Forall2_nilT : Forall2T R nil nil
  | Forall2_consT : ∀ x y l l',
          R x y → Forall2T R l l' → Forall2T R (x::l) (y::l').
```

We solved this problem by supplying customised and stronger mutual induction principles (called `ExpTyping_str_rect` for expressions and similarly for the other categories), obtained by instantiating a more general one, proved by means of the mutually recursive version of the `fix` tactic [5]. Reasoning by induction on the typing relations, we can prove that each well-typed object is uniquely typed. This is also the case for functions.

```
Lemma UniqueFunType (f: Fun) (ft1 ft2: FTyp)
      (k1: FunTyping f ft1) (k2: FunTyping f ft2) :  ft1 = ft2.
```

Although the typing of functions depends on their fuel, we can prove

```
FunTyping (FC fenv tenv e0 e1 x (S n)) ft →
FunTyping (FC fenv tenv e0 e1 x n) ft
```

and conversely

```
sigT (fun ft0 ⇒ FunTyping (FC fenv tenv0 e0 e1 x (S n)) ft0) →
FunTyping (FC fenv tenv0 e0 e1 x n) ft →
FunTyping (FC fenv tenv0 e0 e1 x (S n)) ft.
```

The dynamic semantics of DEC, defined in terms of small-step rules, is comparatively standard and close to the IL presentation, though far less concise, as propagation rules are needed for each constructs. It relies on a notion of configuration parametrised by syntactic categories (i.e. expressions, parameters, q-values and q-functions).

```
Inductive AConf (T: Type) : Type := Conf (state: W) (qq: T).
```

The step rules for q-values and q-functions are just environment lookups.

```
Inductive QVStep : valEnv → AConf QValue → AConf QValue → Type
Inductive QFStep : funEnv → AConf QFun → AConf QFun → Type
```

The step rules for expressions and parameters (evaluated from left to right) are defined by mutual induction, using the principle supplied by Coq.

```
Inductive EStep: funEnv→valEnv→AConf Exp→AConf Exp→Type := ...
with  PrmsStep: funEnv→valEnv→AConf Prms→AConf Prms→Type := ...
```

The reduction rules for `Apply` and particularly the decreasing character of the recursive one (shown below), supplemented by the call-by-value propagation rules, ensures the termination of recursive functions in a way that corresponds to the *iterate*-style construct of IL. Here `isValueList2T` is used to check whether a list of expressions equals a list of lifted values, and `mkVE:valTC→list Value→valEnv` constructs a value environment from a typing context and a list of values of the same length.

```
| Apply_RS1 : ∀(fenv fenv': funEnv) (env: valEnv) (n: W) (e0 e1: Exp)
               (es:list Exp) (vs:list Value) (x: Id) (i: nat) (pt:valTC),
 isValueList2T es vs → length pt = length vs → EStep fenv env
  (Conf Exp n (Apply (QF (FC fenv' pt e0 e1 x (S i))) (PS es)))
  (Conf Exp n (BindMS ((x,(FC fenv' pt e0 e1 x i))::fenv') (mkVE pt vs) e1))
```

Notice the use of `BindMS` to introduce a local environment, with the following step rules.

```
| BindMS_RS : ∀(fenv fenv': funEnv) (env env': valEnv) (n: W) (v: Value),
 EStep fenv env (Conf Exp n (BindMS fenv' env' (Val v))) (Conf Exp n (Val v))
| BindMS_PS : ∀(fenv fenvL:funEnv) (env envL:valEnv) (n n': W) (e e': Exp),
 EStep (fenvL++fenv) (envL++env) (Conf Exp n e) (Conf Exp n' e') →
 EStep fenv env (Conf Exp n (BindMS fenvL envL e))
               (Conf Exp n' (BindMS fenvL envL e'))
```

The reduction rule of `Modify` enacts the monadic behaviour of the stateful action associated with `xf`, returning the value computed by `x_eval` and changing the state according to `x_exec`.

5 The SOS Interpreter

The small-step semantics can be used to compute well-typed programs in well-typed environments. First of all, we extend the definitions of transition steps to reflexive-transitive closures (represented by inductive types, e.g. `EClosure : funEnv → valEnv → AConf Exp → AConf Exp → Type`). Then, after using double induction on the step relation and its reflexive-transitive ex-

tension to prove a weakening lemma (for expressions as shown, and similarly for parameters)

```
Lemma weaken (fenv fenv':funEnv) (env env':valEnv) (n1 n2:W) (e1 e2:Exp):
EClosure fenv env (Conf Exp n1 e1) (Conf Exp n2 e2) →
EClosure (fenv ++ fenv') (env ++ env') (Conf Exp n1 e1) (Conf Exp n2 e2).
```

our strong mutual induction principle on typing suffices to prove a type soundness theorem, with the following formulation for expressions (and similarly for the other mutually defined categories).[3]

```
Lemma ExpEval (ftenv:funTC) (tenv:valTC) (fenv:funEnv) (e:Exp) (t:VTyp):
ExpTyping ftenv tenv fenv e t →
MatchEnvs FunTyping fenv ftenv → ∀ env: valEnv,
MatchEnvs ValueTyping env tenv → ∀ n: W,
Σ (λ v: Value, ValueTyping v t * Σ (λ n': W,
                EClosure fenv env (Conf Exp n e) (Conf Exp n' (Val v)))).
```

The use of Σ types ensures that the witnesses can be extracted from the proof. The proof can then be applied as a function, ensuring that a value of the expected type can always be obtained together with a final state for well-typed expressions in well-typed environments by a finite number of steps. Notice that usually induction on the typing relation only suffices to prove subject reduction, i.e.

```
Lemma ExpSubjectRed (ftenv:funTC) (tenv:valTC) (fenv:funEnv) (e:Exp) (t:VTyp):
   ExpTyping ftenv tenv fenv e t → MatchEnvs FunTyping fenv ftenv →
   ∀ (env: valEnv), MatchEnvs ValueTyping env tenv →
   ∀ (e': Exp) (n n': W), EStep fenv env (Conf Exp n e) (Conf Exp n' e') →
                              ExpTyping ftenv tenv fenv e' t.
```

whereas type soundness, in the case of a terminating language, involves a weak normalisation result typically provable by induction on the step relations. Our typing relations incorporate the inductive aspect on fuel, and therefore suffice to prove normalisation. We prove determinism of evaluation, again by induction on typing.

```
Lemma ExpDeterm (ftenv:funTC) (tenv:valTC) (fenv:funEnv) (e:Exp) (t:VTyp):
  ExpTyping ftenv tenv fenv e t → FEnvTyping fenv ftenv →
  ∀ (env: valEnv), EnvTyping env tenv → ∀ (n n1 n2: W) (e1 e2: Exp),
  EStep fenv env (Conf Exp n e) ((Conf Exp n1 e1)) →
  EStep fenv env (Conf Exp n e) ((Conf Exp n2 e2)) → (n1 = n2) ∧ (e1 = e2).
```

Determinism together with weak normalisation give us strong normalisation, and indeed this makes it possible to ensure that our type soundness proof can be used as an SOS interpreter to evaluate DEC programs. We can run the interpreter on simple expressions, but Coq's evaluation mechanism (notoriously fragile [19]) currently does not carry us far enough, particularly in connection with our extensive use of dependent types.

Nonetheless, we can rely on the Coq extraction mechanism to obtain a certified and efficient implementation of the SOS interpreter. We used extraction to Haskell to generate code which we compiled and run with GHC. The pres-

[3] Specification and proofs in `2xs/dec/src/DEC1` [1]

ence of dependent types in our Coq code required some adjustments. In fact, when Coq types have no direct translation into Haskell, the extraction mechanism will use the Haskell type `Any` (which can be understood as the union of all possible types). This means that in order to print the result of running the interpreter, we need to supply explicitly the translated type using the Haskell function `unsafeCoerce`. As expected, the Haskell interpreter is recursively defined on a term that in Coq has the dependent type of the typing relation. In fact, the computational content of our carefully designed algorithm rests entirely on that relation, rather than on its arguments. Although such arguments have no computational role, they are still present in the extracted code, as they have computational types. But the lazy evaluation strategy of Haskell ensures that they are not evaluated, and thus they can be safely given the value `undefined`.

In the future we would like to tackle the aspect of evaluation in Coq too, in order to show the semantic adequacy of DEC with respect to MC. We have defined a translation of DEC to Gallina, relying on the strong induction principle on typing as we did for type soundness. Ideally we would like to show that for each DEC program, the proof term of this translation is equal to the term obtained from the SOS interpreter.

6 Hoare Logic

We defined a Hoare logic to verify well-typed DEC programs with respect to state properties expressed in Gallina. Our definitions of Hoare triples allow for the postcondition to depend on the value returned by the computation, following [20,21], and for the computation to depend on function and value environments. We provide the syntax `{{ P }} fenv >> env >> e {{ Q }}` to write triples for expressions, where the unary predicate `P` gives the precondition and the binary predicate `Q` the postcondition of running the SOS interpreter on a well-typed expression `e` in well-typed environments `fenv` for functions and `env` for values, corresponding to the following definition

```
Definition THoareTriple_Eval (P : W → Prop) (Q : Value → W → Prop)
           (fenv: funEnv) (env: valEnv) (e: Exp) : Prop :=
  ∀ (ftenv: funTC) (tenv: valTC) (t: VTyp)
    (k1: MatchEnvs FunTyping fenv ftenv) (k2: MatchEnvs ValueTyping env tenv)
    (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'.
```

where the transitive closure hypothesis states that the expression `e`, evaluated in state `s`, leads to value `v` in an updated state `s'`. The syntax `{{ P }} fenv >> env >> ps {{ Q }}` and an analogous definition are used for Hoare triples for parameters. Notice that in contrast with the triples for MC [4] where well-typedness is shallow and implicit, here the typing information is deep and thus needs to be explicit. In principle, this explicitness could bring additional discriminating power, making it easier to distinguish between types that are meant to be different, with different actions associated to them, though modelled by the same shallow type. However, this comes to the cost of an overhead in the proofs. On the other hand, an untyped version of the triples could

not rely on termination, and therefore would be rather weak in comparison with the shallow counterpart.

We supply a Hoare logic library based on our triples, notably including Hoare logic structural rules for each DEC construct, in order to allow for a verification style that is essentially syntax-driven. Most of these rules support bidirectional use, i.e. both by weakest precondition and strongest postcondition, and correspond to big-step rules. For example, the following is the main rule for `BindS`

```
{{P0}} fenv >> env >> e1 {{P1}} →
(∀ v: Value, {{P1 v}} fenv >> (x,v)::env >> e2 {{P2}}) →
{{P0}} fenv >> env >> BindS x e1 e2 {{P2}}
```

This rule allows a triple for the expression `BindS x e1 e2` to be broken down into sequential triples for `e1` and `e2` (the latter in an updated value environment). The main rule for the `Apply` constructor can be conveniently split into two distinct ones, in order to deal with the recursive update of the function environment, which does not take place with zero fuel

```
let f := FC fenv' tenv' e0 e1 x 0  in
  {{P0}} fenv >> env >> PS es {{P1}} →
  (∀ vs: list Value, {{P1 vs}} fenv' >> (mkVE tenv' vs) >> e0 {{P2}}) →
  {{{P0}} fenv >> env >> Apply (QF f) (PS es) {{P2}}
```

whereas it does otherwise (i.e. `fenv'` is updated with the assignment of function `f0` to the identifier `x`)

```
let f0 :=  FC fenv' tenv' e0 e1 x n  in
let f1 :=  FC fenv' tenv' e0 e1 x (S n)  in
  {{P0}} fenv >> env >> PS es {{P1}} →
  (∀ vs, {{P1 vs}} (x, f0)::fenv' >> (mkVE tenv' vs) >> e1 {{P2}}) →
  {{P0}} fenv >> env >> Apply (QF f1) (PS es) {{P2}}
```

As another example, given an external function record `xf: XFun t1 t2`, the rule for `Modify` has more naturally the form of a weakest precondition (we show the case when the argument q is already a lifted value):

```
let q :=  QV (cst t1 v)  in let g :=  λs, xf.x_eval s v  in
let h :=  λs, xf.x_exec s v  in {{λ s. Q (g s) (h s)}}
          fenv >> env >> Modify xf q {{Q}}
```

The validity of these rules is proved by inverting the corresponding operational semantic rules, making use of the determinism of DEC.

7 Case Study: Verifying Properties of Pip

The model of Pip in the shallow embedding [4] is based on Gallina code which can be regarded as a monadic representation of IL. The `LLI` monad used in that representation is defined as an abstract datatype and it wraps together hardware state and undefined behaviours, analogously to applying a state transformer to an error monad [16,22].

```
Definition LLI (A: Type) : Type := state → result (A * state).
Inductive result (A: Type) : Type :=
```

```
  val: A → result A | undef: nat → state → result A.
```

The primitive types are Booleans and subsets of naturals. The HAL functions correspond to the actions in IL. The monadic operations `ret` and `bind`, which can be easily proved to satisfy the monadic laws

```
Definition ret : A → LLI A := fun a s ⇒ val (a, s).
Definition bind : LLI A → (A → LLI B) → LLI B := fun m f s ⇒
 match m s with | val (a, s') ⇒ f a s' | undef a s' ⇒ undef a s' end.
```

provide the semantics for sequencing, let binding and function application. Primitive recursion and conditional expressions are encoded in terms of the corresponding Gallina notions (see Appendix A for a semiformal definition of the corresponding denotational semantics).

The executable specification of Pip rests on a platform model which includes the representation of physical memory as association lists of physical addresses and values, and the specification of HAL primitives corresponding to architecture-dependent functions [4,9]. Stateful functions such as `get` and `put` are only used in the definition of the HAL primitives, thus ensuring that Pip services can access the state only through specific actions. A physical address is modelled as a page identifier (corresponding to a fixed-size chunk of memory) and an offset value called index.

```
Record page := {p :> nat; Hp: p < memorySize}.
Record index := {i :> nat;  Hi: i < pageSize}.
Definition paddr := page * index.
```

The `value` datatype sums up the types of values that can be found in the configuration pages.

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

Here `Pentry` stands for physical entry, `Ventry` for virtual entry, and `vaddr` for virtual address. Physical entries (PTEs) associate a page with its accessibility information.

```
Record Pentry : Type:= {pa: page; present: bool; accessible: bool}.
```

The management of memory is based on a tree-like partition structure. The partition tree is a hierarchical graph in which each node contains a handle called partition descriptor (PD) together with the configuration of the partition, defined as a set of entities, the main one being the MMU configuration. This has the structure of a tree of fixed `levelNum` depth where physical addresses (including those pointing to possible children in the partition tree) are essentially leaves, whereas valid virtual addresses represent maximal branches. In fact, virtual addresses are modelled as lists of indices of length `levelNum+1`. Each of them is translated by the MMU either to the null address or to a physical one, by interpreting each index in the list as offset in the page table at the corresponding level in the MMU. Partitioning management also uses two auxiliary entities, which can be described as *shadows* of the MMU. The first shadow is used to find out which pages are assigned children, and it uses the type `Ventry`. The second shadow is used to associate each PD to the virtual address it has in

the parent partition. The comparatively low-level representation of these structures in the Coq model is based on lists and relies on consistency invariants to ensure e.g. that a list represents a tree. The physical state in Pip is defined by the PD identifier of the currently active partition and the relevant part of the memory state (i.e. essentially, the configuration pages).

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

In the monadic model of Pip, this defines the state for the `LLI` monad [4,9].

In the deep embedding formalisation [1,23], we rely on a concrete module where `Id` is instantiated with strings, and `W` with `state`. The HAL primitives correspond to the actions which are executed as external function calls by means of `Modify`. Since the current definition of DEC does not include rules for error handling, we delegate undefined behaviour to each action, using `option` types. This involves some adjustments. For example, the original HAL primitive in [9] to read a physical address in a given page

```
  Definition readPhysical (p: page) (i: index) : LLI page := bind get (λs,
  match (lookup p i (memory s) page_beq index_beq) with
  | Some (PP a) ⇒ ret a | Some _ ⇒ undefined 5 | None  ⇒ undefined 4 end).
```

gets translated to the following

```
Definition readPhysical' (p: page) (i: index) (s: state) : option page :=
 match (lookup p i (memory s) page_beq index_beq) with
  | Some (PP a) ⇒ Some a | _ ⇒ None end.
```

which can be lifted to DEC as external function.

```
Definition xf_read (p: page) : XFun (option index) (option page) :=
{| x_mod := fun (s: W) (x: option index) ⇒ (s, match x with
  | Some i ⇒ readPhysical' p i s | None ⇒ None end) |}.
Definition ReadPhysical (p:page) (x:Id) : Exp := Modify (xf_read p) (Var x).
```

The fact that the composition of state and error is essentially inverted by the translation is not problematic in our model: all the proofs on the hardware primitives turned out to be easy to adjust.

We translated to DEC three auxiliary functions which are defined in the MC model [9], called `getFstShadow`, `writeVirtual` and `initVAddrTable`. For each function, we proved the main invariant associated with it for its DEC translation, along the lines of the proofs in the shallow embedding, reusing the HAL model in the sense we have described above. The top-level proofs in the shallow embedding are comparatively small (about 50, 100 and 250 lines, respectively), but they are quite representative as they involve using several HAL primitives, sequencing and let-binding, reading from the state (`getFstShadow`), updating the state (`writeVirtual`), as well as a conditional and a recursive function (`initVAddrTable`). Following in the footsteps of the shallow proofs, the deep embedding results in comparable top-level proofs (about 100, 100 and 350 lines respectively), with an overhead due mainly to DEC type-checking, and to the lifting-unlifting of values on which properties may depend. In the case of `getFstShadow`, we have experimented with alternative definitions of HAL primitives, showing that the top-level proof does not change and therefore,

in principle, that DEC could support refinement of abstract specifications. In the case of `initVAddrTable`, we also proved the invariant following a more thoroughly syntax-driven approach, in the spirit of our Hoare logic. This resulted in a significantly shorter top-level proof (about 170 lines, mainly instantiations of metavariables), though it involved proving additional HAL-level lemmas (about 300 lines) not supplied by the original model, yet general enough to be potentially reusable.

The function `getFstShadow` is used to return the physical page of the first shadow for a given partition, and it is implemented monadically using `readPhysical` as well as `getSh1idx` and `Index.succ` as HAL primitives.

```
Definition getFstShadow (p : page) : LLI page :=
  bind getSh1idx (λ x, bind (Index.succ x) (λ y, readPhysical p y)).
```

This function can be translated to DEC as follows

```
Definition GetFstShadow (p : page) : Exp :=
  BindS "x" GetSh1idx (BindS "y" (IndexSucc "x") (ReadPhysical p "y")).
```

where all the subexpressions are based on lifted primitives. The invariant that has been proved for the DEC code [1,23] is the following

```
Lemma GetFstShadowBind (p: page) (P: W → Prop) (fenv: funEnv) (env: valEnv):
{{λ s, P s ∧ PartitionDescriptorEntry s ∧ p ∈ (GetPartitions mltplxr s) }}
    fenv >> env >> (GetFstShadow p)
{{λ sh1 s, P s ∧ NextEntryIsPP p sh1idx sh1 s }}.
```

closely matching the shallow version in [9]. Here again `GetPartitions` and `NextEntryIsPP` are lifted HAL primitives. `GetPartitions` returns the list of all sub-partitions of a given partition, `NextEntryIsPP` returns a Boolean depending on whether the successor of the given index in the given configuration page points to the given physical page, whereas `PartitionDescriptorEntry` defines a specific property of the partition descriptor. The typing information that is explicit in the shallow embedding needs to be extracted from the deep type, and this makes for most of the overhead in the deep proof.

The function `writeVirtual` writes a virtual address to the physical memory and it is used to update configuration pages in the second shadow. It is a HAL primitive, and therefore can be adjusted and lifted to DEC as an external function. The associated invariant ensures that the given value is actually written to the given location, while the properties which do not depend on the updated part of the state are preserved. This example illustrates reuse quite well. Although the invariant requires a comparatively long proof in the shallow embedding, this proof can be replicated almost exactly in the deep embedding, using the same (of many) HAL lemmas.

The recursive function `initVAddrTable` is used to initialise the virtual addresses in the second shadow. Its translation has a comparatively complex DEC structure, involving a conditional and the use of `tableSize` as fuel.

```
Definition InitVAddrTableAux (f i: Id) (p:page) : Exp :=
BindN (WriteVirtual p i defaultVAddr)
      (IfThenElse (LtLtb i maxIndex) (BindS "y" (BindS "idx" (IndexSucc i)
          (ExtractIndex "idx")) (Apply (FVar f) (PS [VLift (Var "y")])))
```

```
                (Val (cst unit tt))).

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

The associated invariant ensures that after execution each entry of the given configuration table contains the default value `defaultVAddr`, regardless of the current index (`cidx`).

```
Lemma InitVaddrTableInv (p: page) (cidx: index)
  (fenv: funEnv) (env: valEnv) : {{λ  s, (λ idx : index , idx < cidx →
     (ReadVirtual table idx (memory s) = Some defaultVAddr)) }}
  fenv >> env >> InitVAddrTable p cidx
{{λ  _ s idx, ReadVirtual p idx (memory s) = Some defaultVAddr}}.
```

The DEC structure of `InitVAddrTableAux` makes it convenient to adopt a syntax-driven approach based on the application of Hoare logic rules (as opposed to unfolding the definition of Hoare triple), in order to facilitate automation and maximise reuse. Indeed, it has been easy to write a tactic in Ltac (the scripting language of Coq) to semi-automate the application of such rules. In comparison, pattern-matching on terms in MC might be trickier. On the other hand, the impact of this basic form of automation is restricted by the need to instantiate metavariables with comparatively complex terms for properties. Moreover, the proof uses induction on `tableSize` (in analogy to the shallow one).

8 Related Work

Differences and complementarity between shallow and deep embedding have been widely discussed in functional programming, in relationship with the development of embedded domain specific languages (EDSLs) [24,25,26,27]. Combinations of shallow and deep embedding have been proposed e.g. in [25] to deal at once with the expression problem (related to extending a deeply embedded language) and the interpretation problem (related to extending the semantic interpretation of a shallow embedding). Their approach consists in extending a deeply embedded core language with a shallowly embedded front-end, thus the opposite of what we do with a DLE. In fact, they share our intent of separating the interpretation of the EDSL from that of the metalanguage. However, they want the high-level qualities of a shallow embedding (e.g. usability and extensibility of the syntax) for the top level part of the EDSL, whereas we need those qualities in the abstract model underneath (where in facts proofs tend to be, mathematically speaking, higher-level ones).

In applications of theorem proving, the difference between shallow and deep embedding has often been associated with a tradeoff between ease in dealing with mathematically higher-level proofs and language manipulation [28]. For example, Cogent [29] is a domain-specific language that has been used to verify file systems in the context of the seL4 project [3]. Targeting Isabelle, Cogent compiles both to a shallow embedding, used in higher-level verification,

and to a deep one, used in verifying C source code (an approach that bears some analogy with our plan A mentioned in section 2).

In the Coq community, refinement from abstract models based on shallow embedding to deeply embedded lower-level ones has been discussed in the context of higher-level formal development [30] as well as in hardware design [31,32]. CertikOS [33,2] provides a method to formally develop low-level applications in Coq, targeting an extension of CompCert Clight and assembly code [13], allowing for composition and refinement of modular specifications which can be imported as external functions into the deeply embedded CompCert frontend. In comparison, our notion of DLE has a radically lighter-weight, domain specific character, relying on a separation of concerns between verification of the executable model (discussed in this paper) and translation to the implementation language. Moreover, unlike our basic Hoare logic, CertikOS provides advanced support for modular reasoning through contextual refinement [33].

9 Conclusions and Further Work

We have presented the core development of DEC as a DLE, with an interpreter based on its small-step operational semantics. The translation of DEC to C is ongoing work, and so is the proof that its denotational translation to the monadic code agrees with its operational semantics. As a preliminary experiment in using DEC as modelling language, we formalised functions of Pip in DEC and we proved model invariants associated with them. The DLE approach has proved fruitful in two main respects: it has enabled us to match neatly the modelling distinction between platform abstraction and service layer with a linguistic one between external and internal functions, hence defining a formal interface between the two; it has supported modular reuse of abstract platform components and associated proofs along that interface, within a framework that allows for direct manipulation of the executable code.

The notion of DLE is essentially oriented toward the design of intermediate, executable models. In the case of DEC, the DLE has been designed to ensure well-typedness and termination. This choice has been made to match the original model in the shallow embedding, rather than the ultimate C target. More generally, the idea we presented is to build domain specific modelling languages that support program development by refining stateful specifications into imperative code, while preserving in Coq the separation of concerns between layered modelling in a language with a comparatively simple model of execution, and translation to a richer implementation language.

Acknowledgments. We wish to thank all the other members of the Pip Development Team, especially Gilles Grimaud and Samuel Hym, Vlad Rusu and the anonymous reviewers for feedback and discussion. This work has been funded by the European Celtic-Plus Project ODSI C2014/2-12.

A Appendix: Denotational Semantics

We can define a denotational semantics of IL relying on a monadic translation similar to the one in [16] based on a state monad M with fixed state type W. The semantics is defined by a translation of IL to the monadic metalanguage (4–7), for types (Θ_t), expressions (Θ_e), expression lists (Θ_{es}) and functions (Θ_f), using the auxiliary definitions here also included (1–3).

$$\begin{array}{l}\mathsf{condM}: \ M\ \mathsf{Bool} \to M\ t_1 \to M\ t_1 \to M\ t_1\ := \\ \qquad \lambda x_0\ x_1\ x_2.\ \mathsf{bind}\ x_0 \\ \qquad (\lambda v_0.\ \mathsf{bind}\ x_1\ (\lambda v_1.\ \mathsf{bind}\ x_2\ (\lambda v_2.\ \mathsf{if\_then\_else}\ v_0\ v_1\ v_2)))\end{array} \tag{1}$$

$$\begin{array}{l}\mathsf{mapM}: \ (\forall t.\ \mathsf{Exp}\ t \to M\ t) \to \mathsf{Exps}\ ts\ \to M\ \ ts\ := \\ \qquad \lambda f\ es.\ \mathsf{match}\ es\ \mathsf{with}\ []\ \Rightarrow\ [] \\ \qquad\qquad\qquad \mid e :: es' \Rightarrow \mathsf{bind}\ (f\ e)\ (\lambda x.\ (\mathsf{bind}\ (\mathsf{mapM}\ f\ es') \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\lambda xs.\ \mathsf{ret}\ (x :: xs))))\end{array} \tag{2}$$

$$\begin{array}{l}\mathsf{iterateM}\ (ts : \mathsf{Typs})\ (t : \mathsf{Typ})\ (e_0 :\ ts \to M\ t) \\ \qquad (e_1 :\ (ts \to M\ t) \to (ts \to M\ t)) \\ \qquad (n : \mathsf{Nat})\ (xs :\ ts) : M\ t\ := \mathsf{match}\ n\ \mathsf{with} \\ \qquad\quad 0\ \Rightarrow\ e_0\ xs\quad \mid S\ n'\ \Rightarrow\ e_1\ (\mathsf{iterateM}\ ts\ t\ e_0\ e_1\ n')\ xs\end{array} \tag{3}$$

$$\begin{array}{lll}\Theta_t\ (\mathsf{Exp}\ t) & := & M\ t \\ \Theta_t\ (\mathsf{Exps}\ ts) & := & M\ ts \\ \Theta_t\ (\mathsf{Fun}\ t\ ts) & := & ts \to M\ t \\ \Theta_t\ (\mathsf{Act}\ t\ ts) & := & ts \to M\ t\end{array} \tag{4}$$

$$\begin{array}{lll}\Theta_e\ :\ \forall t.\ \mathsf{Exp}\ t \to M\ t & & \\ \Theta_e\ (\mathsf{val}\ x) & = & \mathsf{ret}\ x \\ \Theta_e\ (\mathsf{binds}\ e_1\ (\lambda x : t.\ e_2)) & = & \mathsf{bind}\ (\Theta_e\ e_1)\ (\lambda x : t.\ \Theta_e\ e_2) \\ \Theta_e\ (\mathsf{cond}\ e_1\ e_2\ e_3) & = & \mathsf{condM}\ (\Theta_e\ e_1)\ (\Theta_e\ e_2)\ (\Theta_e\ e_3) \\ \Theta_e\ (\mathsf{call}\ fc\ es) & = & \mathsf{bind}\ (\Theta_{es}\ es)\ (\Theta_f\ fc) \\ \Theta_e\ (\mathsf{xcall}\ a\ es) & = & \mathsf{bind}\ (\Theta_{es}\ es)\ a\end{array} \tag{5}$$

$$\begin{array}{l}\Theta_{es}\ :\ \mathsf{Exps}\ ts \to M\ ts \\ \Theta_{es}\ es = \ \mathsf{mapM}\ \Theta_e\ es\end{array} \tag{6}$$

$$\begin{array}{l}\Theta_f :\ \mathsf{Fun}\ t\ ts \to ts \to M\ t \\ \Theta_f\ (\mathsf{fun}\ (\lambda\ x :\ ts.\ e_0)\ (\lambda\ (r :\ ts \to \mathsf{Exp}\ t)\ (x :\ ts).\ e_1)\ \ n)\ = \\ \qquad \mathsf{iterateM}\ ts\ t\ (\lambda\ x :\ ts.\ \Theta_e\ e_0) \\ \qquad\qquad\qquad (\lambda\ (r :\ ts \to M\ t)\ (x :\ ts).\ \Theta_e\ e_1))\ \ n\end{array} \tag{7}$$

References

1. Torrini, P., Nowak, D., Cherif, M.S., Jomaa, N.: The repository of DEC (2018) https://github.com/2xs/dec.git.
2. Gu, R., Shao, Z., Chen, H., C., W.S., Kim, J., Sjoberg, V., Costanzo, D.: CertiKOS: an extensible architecture for building certified concurrent OS kernels. In: OSDI. (2016) 653–669
3. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tich, H., Winwood, S.: seL4: formal verification of an OS kernel. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. (2009) 207–220
4. Jomaa, N., Torrini, P., Nowak, D., Grimaud, G., Hym, S.: Proof-oriented design of a separation kernel with minimal trusted computing base. In: AVOCS'18, Proceedings. (2018) 16 pages http://www.cristal.univ-lille.fr/~nowakd/pipdesign.pdf.
5. Bertot, Y., Casteran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer (2004)
6. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: A Proof Assistant for Higher-order Logic. Springer-Verlag (2002)
7. Bergougnoux, Q., Grimaud, G., Iguchi-Cartigny, J.: Porting the pip proto-kernel's model to multi-core environments. In: IEEE-DASC'18. (2018) 8 pages
8. Yaker, M., Gaber, C., Grimaud, G., Wary, J.P., Sanchez-Leighton, V., J., I.C., Han, X.: Ensuring IoT security with an architecture based on a separation kernel. In: Fi-Cloud'18. (2018) 8 pages
9. Bergougnoux, Q., Grimaud, G., Jomaa, N., Hauspie, M., Helluy-Lafont, E., Hym, S., Iguchi-Cartigny, J., Nowak, D., Torrini, P., Yaker, M.: The repository of Pip (2018) http://pip.univ-lille1.fr.
10. Zhao, Y., Sanan, D., Zhang, F., Liu, Y.: High-assurance separation kernels: a survey on formal methods. arXiv preprint arXiv:1701.01535 (2017)
11. Dam, M., Guanciale, R., Khakpour, N., Nemati, H., Schwarz, O.: Formal verification of information flow security for a simple ARM-based separation kernel. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. CCS '13, ACM (2013) 223–234
12. Hym, S., Oudjail, V.: The repository of Digger (2017) https://github.com/2xs/digger.
13. Blazy, S., Leroy, X.: Mechanized semantics for the Clight subset of the C language. Journal of Automated Reasoning (2009) 263–288
14. Torrini, P., Nowak, D.: DEC 1.0 specification (2018) https://github.com/2xs/dec.git.
15. Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Theor. Comput. Sci. **103**(2) (1992) 235–271
16. Moggi, E.: Notions of computation and monads. Information and computation (1991) 55–92
17. Churchill, M., Mosses, P.D., Sculthorpe, N., Torrini, P.: Reusable components of semantic specifications. In: TAOSD 12. LNCS 8989. Springer (2015) 132–179
18. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. **60-61** (2004) 17–139
19. Leroy, X.: Using Coq's evaluation mechanisms in anger (2015) http://gallium.inria.fr/blog/coq-eval/.
20. Cock, D., Klein, G., Sewell, T.: Secure microkernels, state monads and scalable refinement. In: Theorem Proving in Higher Order Logics, 21st International Conference,

TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings. Volume 5170 of Lecture Notes in Computer Science., Springer (2008) 167–182
21. Swierstra, W.: A Hoare logic for the state monad. In: Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings. Volume 5674 of Lecture Notes in Computer Science., Springer (2009) 440–451
22. Wadler, P.: Comprehending monads. Mathematical Structures in Computer Science (1992) 461–493
23. Cherif, M.S.: Project report – modelling and verifying the Pip protokernel in a deep embedding of C (2017) `https://github.com/2xs/dec.git`.
24. Gibbons, J., Wu, N.: Folding domain-specific languages: Deep and shallow embeddings (functional pearl). In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP. Volume 49. (2014)
25. Svenningsson, J., Axelsson, E.: Combining deep and shallow embedding of domain-specific languages. Computer Languages, Systems & Structures **44** (2015) 143–165
26. Jovanovic, V., Shaikhha, A., Stucki, S., Nikolaev, V., Koch, C., Odersky, M.: Yin-yang: Concealing the deep embedding of dsls. In: Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences. GPCE 2014, ACM (2014) 73–82
27. Carette, J., Kiselyov, O., Shan, C.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. J. Funct. Program. **19** (2009) 509–543
28. Wildmoser, M., Nipkow, T.: Certifying machine code safety: Shallow versus deep embedding. In: Theorem Proving in Higher Order Logics, 17th International Conference, TPHOL '04. (2004) 305–320
29. O'Connor, L., Chen, Z., Rizkallah, C., Amani, S., Lim, J., Murray, T.C., Nagashima, Y., Sewell, T., Klein, G.: Refinement through restraint: bringing down the cost of verification. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP'16, ACM (2016) 89–102
30. Delaware, B., Pit-Claudel, C., Gross, J., Chlipala, A.: Fiat: Deductive synthesis of abstract data types in a proof assistant. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '15. (2015) 689–700
31. Chlipala, A.: The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In: ICFP'13, Springer (2013)
32. Vijayaraghavan, M., Chlipala, A., Arvind, Dave, N.: Modular deductive verification of multiprocessor hardware designs. In: Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II. (2015) 109–127
33. Gu, R., Koenig, J., Ramananadro, T., Shao, Z., Wu, X.N., Weng, S.C., Zhang, H., Guo, Y.: Deep specifications and certified abstraction layers. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '15, ACM (2015) 595–608