Formalising Executable Specifications of Low-Level Systems

Paolo Torrini, David Nowak, Narjes Jomaa, M. Sami Cherif
CRIStAL – Universite' de Lille

VSTTE 2018

July 18th, 2018

Work supported by the
European Celtic-Plus Project ODSI C2014/2-12.

Motivation

Verifying a separation kernel

Separation kernel: OS kernel with verified security policy

- ensures non-interference between different user applications runnning on the same machine
- based on memory access control (memory separation)

General approach: interactive theorem proving (e.g. Coq)

- embedded specific language, translatable automatically to the implementation language (e.g. C), to represent the executable model on top of hardware abstraction
- Hoare logic with properties expressed in the metalanguage

General model structure

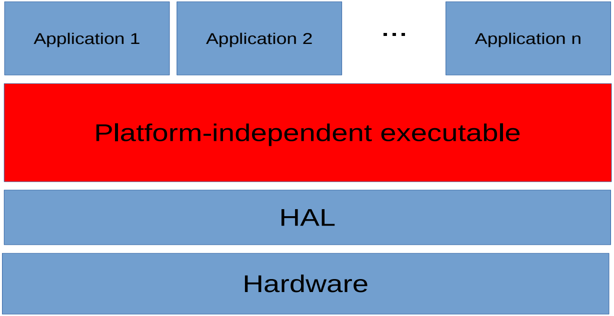
Service layer: platform-independent, executable model

- includes (at least) memory management
- ultimately implemented in C

HAL: mainly assembly functions

Hardware model: includes (at least)

- physical memory
- memory management unit (MMU)



Hoare logic compositional proof strategy

– use Hoare logic rules to reduce a service layer triple

Th : `{{ PreCond }}` *SL_prog* `{{ PostCond }}`

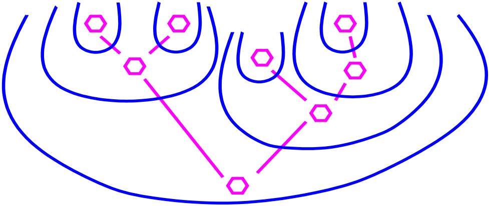
to lemmas (or triples) on HAL interface primitives

L1 : `{{ PreCond_1 }}` *HAL_prim_1* `{{ PostCond_1 }}`
...
Ln : `{{ PreCond_n }}` *HAL_prim_n* `{{ PostCond_n }}`

– maximise reuse!

Pip: a separation proto-kernel

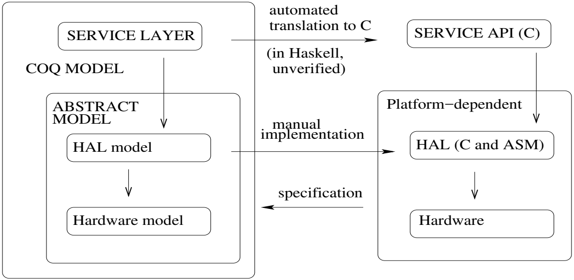
EU ODSI Project: minimal trusted computing base for isolation on demand (AVoCS paper tomorrow!)



Kernel mode:

- virtual memory management based on a partition tree
  - kernel isolation
  - horizontal isolation (siblings do not share)
  - vertical sharing (parents manage children)
- context switching

Pip: model and implementation



- Hardware, HAL: abstract functional spec
- Service layer: close to implementation
    - written in monadic code (MC): shallow embedding of a C-like (stateful, first-order) typed imperative language (IL) with a simple operational semantics

Problem: we want a verified translation

. . . but we do not want (and do not need) to compare denotationally Gallina and C (two large languages). In fact

- MC is much smaller than Gallina (maybe and state monads, no use of abstract dataypes, first-order)
- correspondingly small fragment of C: no structures, no arrays, no pointer passing, memory access through HAL, mainly functional, terminating programs

Goal: a verified translation to CompCert C

Intermediate step: *reification of IL . . .*

Language

Our imperative language (IL): syntax

Expressions and expression lists:

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

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

Primitive recursive functions, with fuel:

$$
\begin{aligned}
&\text{Fun } (t : \text{Typ})\ (ts : \text{Typs}) \;:= \\
&\quad \textbf{fun } (ts \to \text{Exp } t)\ ((ts \to \text{Exp } t) \to ts \to \text{Exp } t)\ \text{nat}
\end{aligned}
$$

Actions (generic effects, state type W):

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

Our imperative language (IL): reduction rules

Given a specification of CBV (either by congruence rules or evaluation contexts):

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

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

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

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

$$\begin{array}{l}\langle\ s \triangleright \mathbf{call}\ (\mathsf{fun}\ e_0\ e_1\ (S\ n))\ (\mathsf{map}\ \mathbf{val}\ vs)\ \rangle \longrightarrow \\ \qquad\qquad\qquad \langle\ s \triangleright e_1\ (\mathbf{fun}\ e_0\ e_1\ n)\ vs\ \rangle\end{array}$$

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

Shallow and deep embedding

Two ways of representing IL in the metalanguage:

- shallow embedding: semantic representation of the language
  - terms represented denotationally, use of monads for side-effects
  - evaluation delegated to Gallina
- deep embedding: language represented as an object (reified) using abstract datatypes
  - the operational semantics is defined explicitly
  - an interpreter is needed

Shallow vs Deep: pros and cons

	Shallow	Deep
maths	easier	harder (overhead)
language translation	very hard	easy
language extension	easy	very hard
execution for free	yes	no
separate execution	no	yes

Executable and Abstract layers

	Executable	Abstract
system implementation	automated translation	manually implemented
separate execution	desirable	not needed
language extension	HAL primitives only	generally needed

Hybrid embedding strategy

Ideal approach:

- executable model as deep embedding:
  fixed set of control primitives
- HAL primitives as external functions
- Hoare logic compositional proof strategy

Advantages:

- service layer: separate execution, translation support
- platform abstraction: easier maths, extensibility

Deeply-embedded Language Extension (DLE):
deep embedding as an extension of the metalanguage

- deep types as lifted metalanguage types
- allow metalanguage functions as external functions
- deep embedding as interface of specialised interpreters

DEC: IL as DLE in Coq

```
Parameter Id: Type
Parameter W: Type

Class ValTyp: Type → Prop
Definition VTyp := Σ ValTyp

Record XFun (T1 T2: Type) : Type :=
{ x_mod : W → T1 → W * T2 }

Inductive Fun : Type := FC (localFunEnv: funEnv)
                           (args: list (Id * VTyp))
                           (default: Exp) (body: Exp)
                           (funName: Id)
                           (fuel: nat)
```

DEC expressions

```
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 (q: QFun) (ps: Prms)
 | Modify {T1 T2: Type} {VT1: ValTyp T1} {VT2: ValTyp T2}
         (xf: XFun T1 T2) (q: QValue)
with QFun : Type := FVar (x: Id) | QF (f: Fun)
with Prms : Type := PS (es: list Exp)
```

Small-step semantics

Configurations (parametrised by a syntactic category):

```
AConf (C: Type): Type := Conf (st: W) (t: C)
```

Enviroment lookups (for q-values and q-functions):

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

Expression evaluation (small-step):

```
EStep: funEnv→valEnv→AConf Exp→AConf Exp→Type
PrmsStep: funEnv→valEnv→AConf Prms→AConf Prms→Type
```

Algorithmic static semantics

```
FTyp : Type := FT (args_t: valTC) (ret_t: VTyp)

ExpTyping : funTC→valTC→funEnv→Exp→VTyp→Type
PrmsTyping : funTC→valTC→funEnv→Prms→PTyp→Type
QFunTyping : funTC→funEnv→QFun→FTyp→Type
FunTyping : Fun→FTyp→Type := ...
| 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
      FEnvTyping 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)
```

Type soundness (SOS interpreter)

```
  Lemma ExpEval (ftenv: funTC) (tenv: valTC)
                (fenv: funEnv) (env: valEnv)
                (e: Exp) (t: VTyp):
ExpTyping ftenv tenv fenv e t →
FEnvTyping fenv ftenv →
EnvTyping env tenv → ∀ s: W, Σ(λv:Value,
  ValueTyping v t      *      Σ(λs':W,
  EClosure fenv env (Conf Exp s e) (Conf Exp s' (Val v))))
```

Each well-typed program in a well-typed environment can evaluate to a return value of matching type and a final state.
Proof: by induction on the typing relation (customised principle).

The proof (1-2k lines) is actually the SOS interpreter, and it can also be extracted to Haskell (more than 30k lines!)

Determinism also proven: strong normalisation follows.

Verification

Hoare triples for DEC

shallow properties (keep proofs on HAL primitives simpler)

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

nicer syntax (:->) `{{ P }} fenv >> env >> e {{ Q }}`

Hoare logic rules: examples

Biderectional rule for let-style binding:

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

Backward rule for external function call:

```
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}}
```

HL rules can be applied in a (tendentially) syntax-directed way: possibility to partially automate using Ltac

Example: recursive function definition

```
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 nil [("x", vtyp index)] (Val (cst unit tt))
              (initVAddrTableAux "initVAddrTable" "x" p)
              "initVAddrTable" tableSize))
      (PS [Val (cst index i)]).
```

Example: Pip invariant

```
Lemma initVAddrTableInv (p: page) (curr_idx idx: index)
(fenv: funEnv) (env: valEnv) :
{{λ s, idx < curr_idx →
    readVirtual table idx (memory s) = Some defaultVAddr }}
  fenv >> env >> initVAddrTable p curr_idx
{{λ _ s, readVirtual p idx (memory s) = Some defaultVAddr}}
```

Proof:

– induction on table size

– decompose by structural Hoare logic rules (automation!)

– instantiate with predicates

– apply lemmas on HAL primitives (reuse!)

Verification with DEC

Good for reuse: DLE character of DEC $+$ Hoare logic with shallow properties. In fact:

- significant reuse of proofs on the HAL primitives (abstract platform model)

Proofs on DEC code tend to be longer, though comparable in size with those on MC

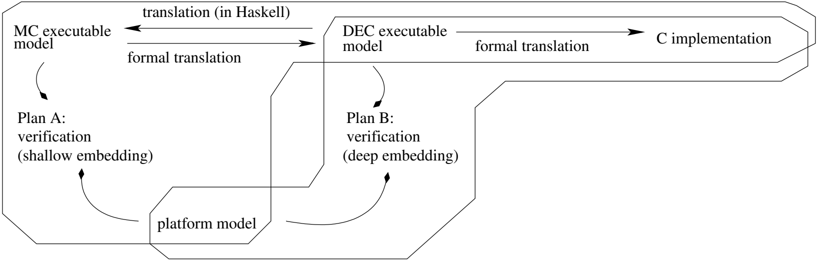
- overhead for the deep embedding mainly associated with type checking and with boxing/unboxing of values

On the other hand: comparatively easy to build tactics that take advantage of syntactic structure

Concluding remarks

Two verification plans: A and B

Plan B: verification in the deep embedding
Plan A: adequately translate between shallow and deep



Conclusions and further work

We have presented the development of DEC and a case study on the verification of Pip invariants reusing existing platform abstraction (plan B)

DLE approach: designing embedded specific languages to make domain specific reasoning easier, maximise reuse, lower maintenance costs

Concerning translations (work in progress):
– translation from DEC to CompCert C (for both plans)
– defining an interpretation of DEC into Gallina (for plan A)

DEC repository:

`https://github.com/2xs/dec`

EU Celtic-Plus ODSI Project:

`https://www.celticplus.eu/project-odsi/`

Paper on Pip tomorrow at AVoCS: Proof-oriented design of a separation kernel with minimal trusted base.
N. Jomaa, P. Torrini, D. Nowak, G. Grimaud, S. Hym.

Thanks!