Formal Development of the Pip Protokernel

Narjes Jomaa, David Nowak, Paolo Torrini
Joint work with the Pip team

ENTROPY 2018

January 25, 2018

This work is partially supported by the
European Celtic-Plus Project ODSI C2014/2-12.

The Pip protokernel: a brief system overview (*David Nowak*)

Pip design principles and security properties (*Narjes Jomaa*)

From the executable specification to C (*Paolo Torrini*)

The Pip protokernel: a brief system overview (*David Nowak*)

Pip design principles and security properties (*Narjes Jomaa*)

From the executable specification to C (*Paolo Torrini*)

On-Demand Secure Isolation

- This research is part of the European project ODSI.
  - Led by Orange
  - 1 academic partner: The university of Lille
  - 8 industrial partners from France, Romania, and Spain
- In Lille: 3 PhD students and 1 postdoctoral researcher.
- The Pip protokernel is one of the foundations of this project.
- Security protocols are designed on top of Pip.
- Case studies by industrial partners: IoT, M2M, SCADA
- Common Criteria certification

Memory isolation between applications

Why? For safety and security

How? By software (OS kernel), and hardware (MMU, CPU kernel mode)

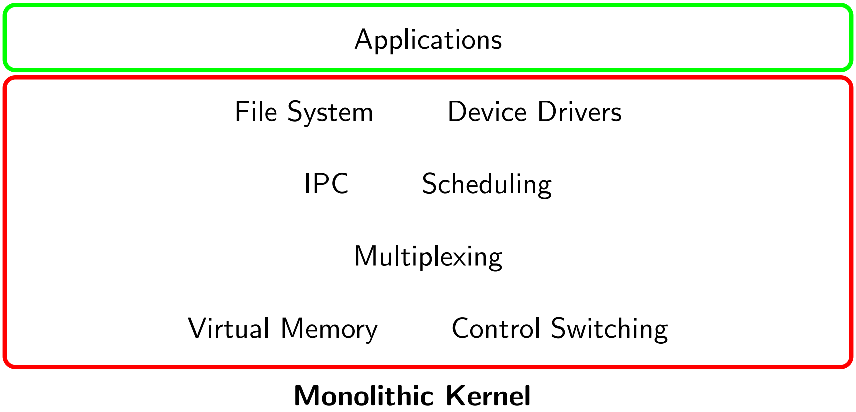
Correct? Ensured by a formal proof in Coq

Feasible? Yes, by reducing the trusted computing base to its bare bone

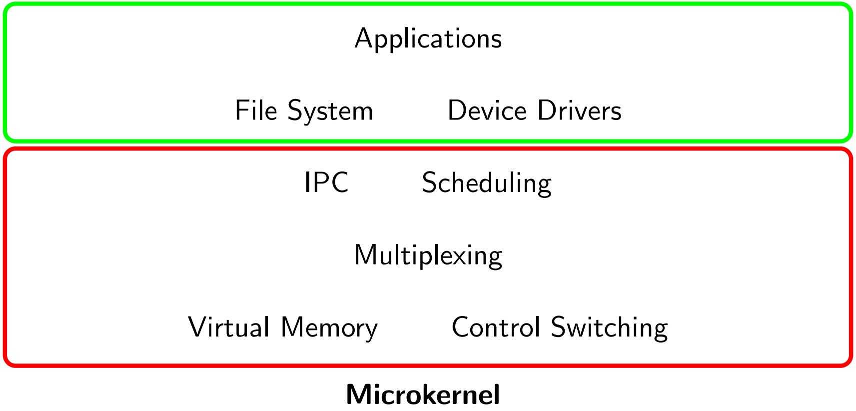
reducing the TCB $\Rightarrow$ increasing feasibility of a formal proof & reducing the attack surface

simplifying the specification language $\Rightarrow$ increasing feasibility of verified translation to C

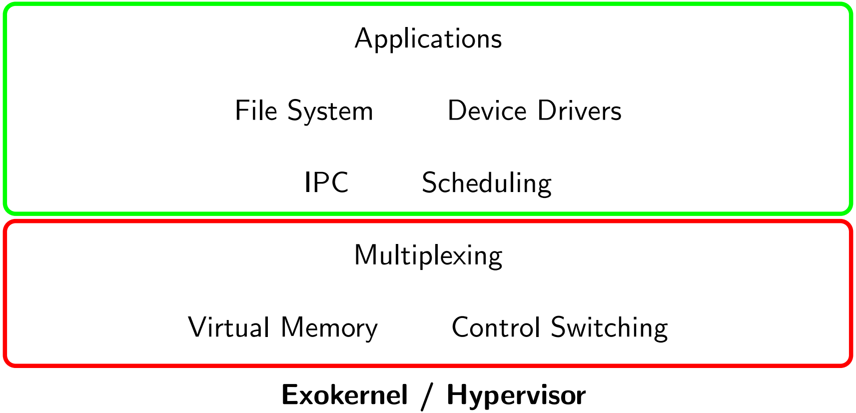
From monolithic kernel to the Pip protokernel



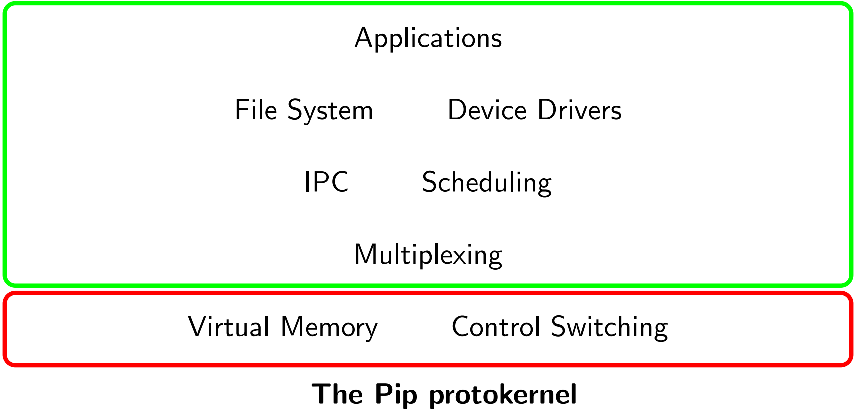
From monolithic kernel to the Pip protokernel



From monolithic kernel to the Pip protokernel



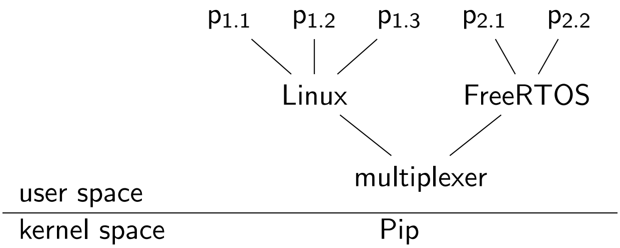
From monolithic kernel to the Pip protokernel



Partition tree

Pip organizes the memory into hierarchical partitions.

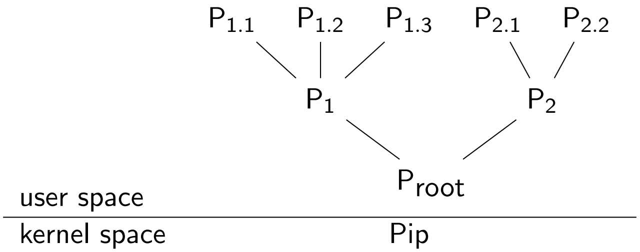
Example



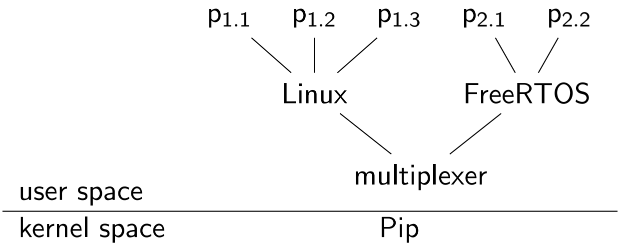
Partition tree: the point of view of Pip

The contents of each partition is not relevant for Pip.

- **Horizontal isolation**
  Partitions in different subtrees are isolated from each other,
  e.g. $P_{1.1}$ cannot access memory of $P_{1.2}$ or P_2.
- **Vertical sharing**
  A partition has access to the memory of its descendants.
- **Kernel isolation**
  Pip is isolated from all partitions.



Partition tree: dealing with interrupts



- **Software interrupts**
  - Pip deals with software interrupts to itself,
    e.g. FreeRTOS asks Pip to create a new partition.
  - Pip forwards other software interrupts to the caller's parent,
    e.g. $p_{1.2}$ make a system call to Linux.
- Pip forwards **hardware interrupts** to the root partition,
  e.g. a network packet has arrived.

Pip system calls

10 elementary system calls

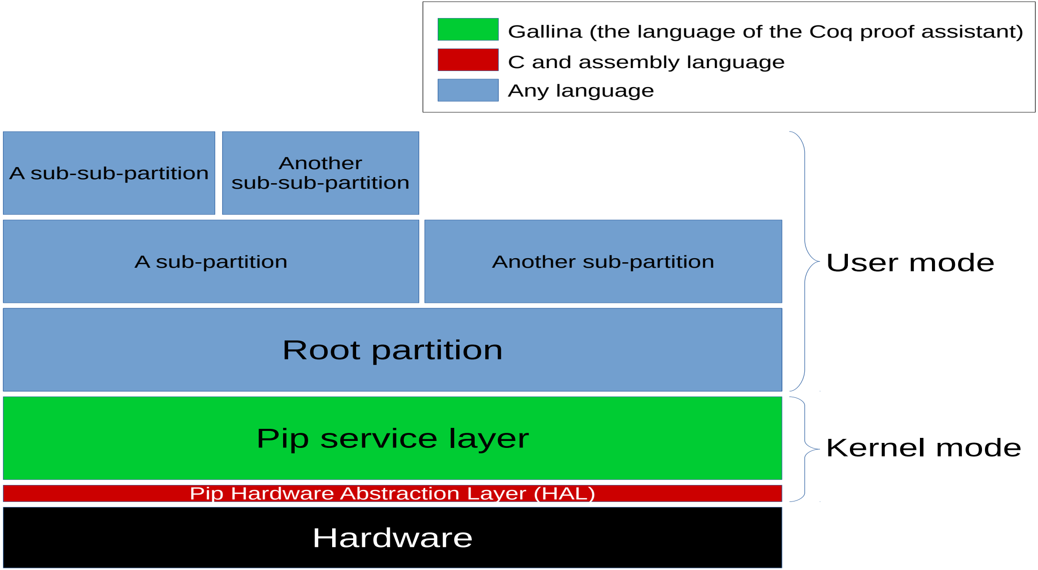
- **Memory management**

`createPartition`	creates a child partition
`removePartition`	deletes a child partition
`addVaddr`	lends a memory page to a child
`removeVaddr`	removes a memory page from a child
`pageCount`	the number of needed configuration pages
`prepare`	gives needed configuration pages
`collect`	takes back unused configuration pages
`mappedInChild`	returns the child using a given page

- **control switching**

`dispatch`	notifies a partition about an interrupt
`resume`	restores the context of a partition

Software layers



Applications

- The HAL of Pip has been ported to:
  - QEMU (x86)
  - x86
  - The Galileo board (Intel Pentium-compliant embedded board)
- Kernels ported on Pip
  - FreeRTOS: Tasks can be isolated in sibling partitions.
  - Linux 4.10.4: More involved because Linux configures MMU.
- Porting a kernel to Pip essentially consists of:
  - removing privileged instructions and operations, and
  - replacing them with system calls to Pip (paravirtualization).
- Drhystone benchmark: low overhead of 2,6% in terms of CPU cycles

Formal verification

- Formal verification of an executable specification of Pip

  Addressed by Narjes Jomaa in the next part of this presentation

- Verified translation of the executable specification into C

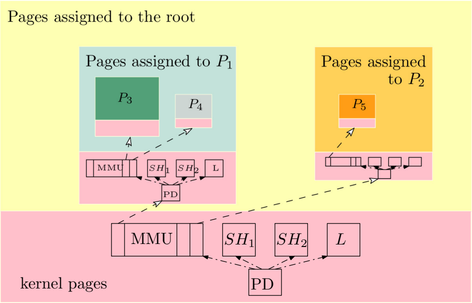
  Addressed by Paolo Torrini in the final part of this presentation

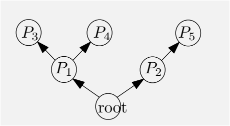
The Pip protokernel: a brief system overview (*David Nowak*)

Pip design principles and security properties (*Narjes Jomaa*)

From the executable specification to C (*Paolo Torrini*)

Partition tree management





The configuration of a partition

- Partition descriptor (*PD*)
- MMU tables
- Shadow 1 (*SH1*) and Shadow 2 (*SH2*)
- Linked list (*L*)

MMU briefly

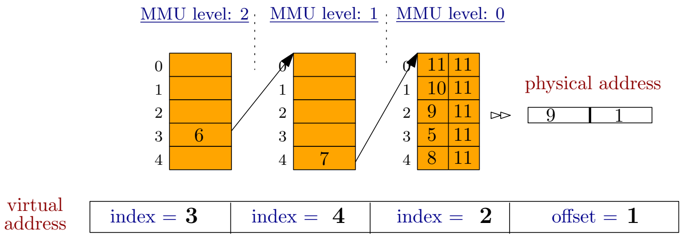
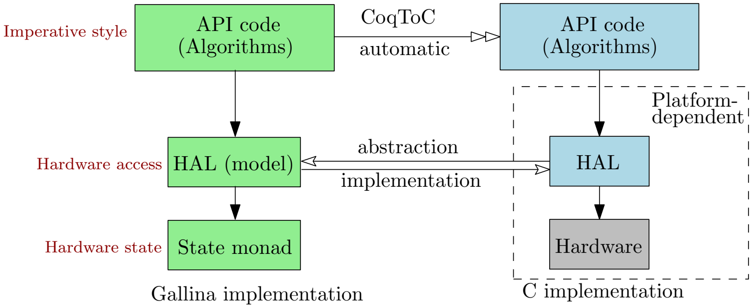


Figure: MMU with 3 levels of indirection

Data structure of partitions

- MMU structure: Define assigned pages and access control
- Mirror the MMU structure
  - Shadow 1: Find out which pages are assigned to children and which pages are used as a partition descriptor identifier *(security)*
  - Shadow 2: Ease getting back the ownership of assigned pages *(efficiency)*
- List (L): Ease getting back the ownership of pages lent to the kernel *(efficiency)*

Pip design principles

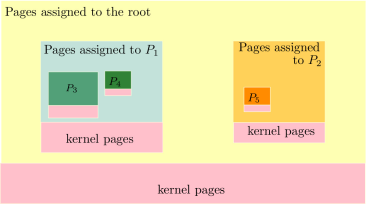


- Hardware state: the part that is relevant to model the partition tree
  - the partition that is currently active
  - the physical memory where Pip stores its own data
- Exclude the use of all objects that would require a GC: lists, trees → Encoding these structure in physical memory using the HAL

Security properties

The horizontal isolation property

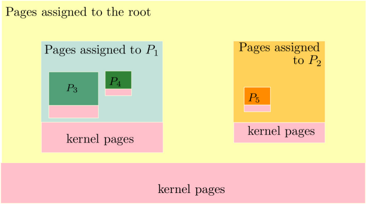
```
Definition HI s : Prop :=

∀ parent child1 child2 : page,

  parent ∈ (partitionTree s) →

  child1 ∈ (children parent s) →

  child2 ∈ (children parent s) →

  child1 ≠ child2 →

  (allocatedPages child1 s) ∩ (allocatedPages child2 s) = ∅.
```



- Sibling partitions cannot access each others memory.

Hierarchical TCB (vertical sharing)

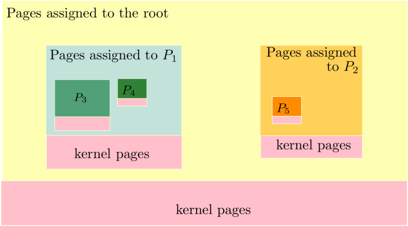
```
Definition VS s : Prop :=
∀ parent child : page,
 parent ∈ (partitionTree s) →
 child ∈ (children parent s) →
 (allocatedPages child s) ⊆ (assignedPages parent s).
```



- All the pages allocated for a partition are included in the pages assigned to its ancestors

The kernel isolation property

```
Definition KI s : Prop :=
∀ partition1 partition2 : page,
  partition1 ∈ (partitionTree s) →
  partition2 ∈ (partitionTree s) →
  (ownedPages partition1 s) ∩ (kernelPages partition2 s) = ∅.
```



- No partition can access to the pages owned by the kernel.

Information flow property

- As a corollary to VS and HI:
  Non-influence property for isolated partition was proved
- Abstract information flow model
- Assumption about hardware side effects

Verification approach

Verification approach

Hoare logic on top of the LLI (Low Level Interface) monad

{{Precondition}} Program {{Postcondition}}

- Program: a monadic function (of type `LLI A`)
- Precondition: a unary predicate on the starting state
- Postcondition: binary predicate on the returned value and on the ending state

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

{{ P }} m {{ Q }}

States that if the precondition holds then

- the postcondition holds; and
- there is no undefined behavior

The need of consistency properties

- We cannot prove the following invariant

  {{HI & VS & KI}} API_service {{HI & VS & KI}}

- Properties about the Pip's data structure are missing
  - The precondition should be strengthened with consistency properties
  - The consistency properties must also be preserved

  {{HI & VS & KI & C}} API_service {{HI & VS & KI & C}}

- consistency $\approx$ well-formedness of Pip's data structures

Example: createPartition invariant

{{HI & VS & KI & C}} createPartition v1 v2 v3 v4 v5 {{HI & VS & KI & C}}

Proceed forward using transitivity (1/2)

```
{{HI & VS & KI & C}}

  perform currentPart := getCurPartition in
  perform ptv1FromPD := getTableAddr currentPart v1 nbL in
          ...
  if negb accessv1 then ret false else
  writeAccessible ptv1FromPD idxv1 false ;;
  ...
{{HI & VS & KI & C}}
```

Proceed forward using transitivity (2/2)

First sub-goal:

{{HI & VS & KI & C}}

getCurPartition

{{HI & VS & KI & C & P currentPart }}

Second sub-goal:

{{HI & VS & KI & C & P currentPart}}

```
perform ptv1FromPD := getTableAddr currentPart v1 nbL in
        ...
if negb accessv1 then ret false else
writeAccessible ptv1FromPD idxv1 false ;;
        ...
```

{{HI & VS & KI & C}}

Verification overview

Invariants (`Qed`)	**line of proof**
createPartition (300 loc)	$\approx$ 60000
createPartition + addVaddr (50 loc)	$\approx$ 78000
createPartition + addVaddr + mappedInChild(20 loc)	$\approx$ 78250

Table: Overview of the proof

The Pip protokernel: a brief system overview (*David Nowak*)

Pip design principles and security properties (*Narjes Jomaa*)

From the executable specification to C (*Paolo Torrini*)

Translating to C

Coq executable model and extracted OCaml code:

- needs big runtime environment
- not efficient enough

We need a translation to low level languages:

- HAL: manual implementation in assembly and C
- Service Layer: C code automatically generated from Gallina
- currently compiled by GCC

Translating to C

Coq executable model and extracted OCaml code:

- needs big runtime environment
- not efficient enough

We need a translation to low level languages:

- HAL: manual implementation in assembly and C
- Service Layer: C code automatically generated from Gallina
- currently compiled by GCC

However: we want a verified translation to CompCert C

- certified compilation
- tail-recursive optimisation

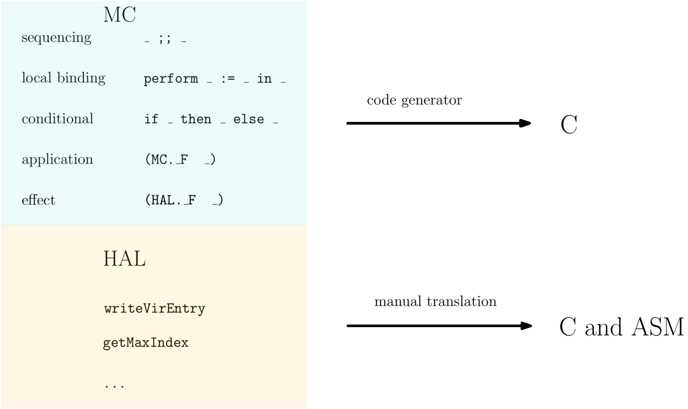
Pip monadic code (MC)

- Low-level HAL primitives
- Higher-level monadic code (MC)

```
Fixpoint initVTable timeout shadow1 idx :=
  match timeout with
  | 0 ⇒ ret tt
  | S timeout1 ⇒
    perform max := getMaxIndex in
    perform res := Index.ltb idx max in
    if (res)
    then
      perform daddr := getDefaultVAddr in
      writeVirEntry shadow1 idx daddr ;;
      perform nidx :=  Index.succ idx in
      initVTable timeout1 shadow1 nidx
    else ...
  end.
```

Translation to C

We use a Haskell-implemented translator (*digger*) to translate from the Gallina AST of MC to C.



Shallow embedding

MC is a shallow embedding, i.e. a semantic representation of a language in Coq, based on a set of Gallina definitions.

```
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.
```

```
perform x := m in e   for   bind m (fun x => e)

              m ;; e   for   bind m (fun _ => e)
```

Value types: `bool` and subtypes of `nat`

Sample translation

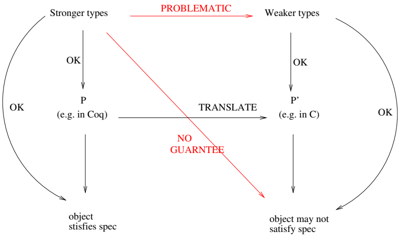
Example: a function defined in Coq, using the monadic code:

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

and its generated translation to C:

```
uintptr_t getFstShadow (const uintptr_t partition) {
  const uint32_t idx = getSh1idx ();
  const uint32_t idxSucc = succ (idx);
  return readPhysical (partition, idxSucc); }
```

Problem: generating verified code



General solution: define a semantic translation from weak to strong (w.r.t. types), and reverse it

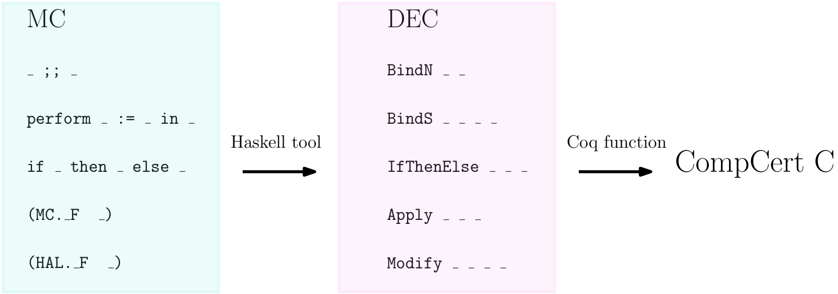
However: we do not want to define a semantics of C in Coq, we want to use an existing one which also provides compilation – CompCert C.

Verified translation: our approach

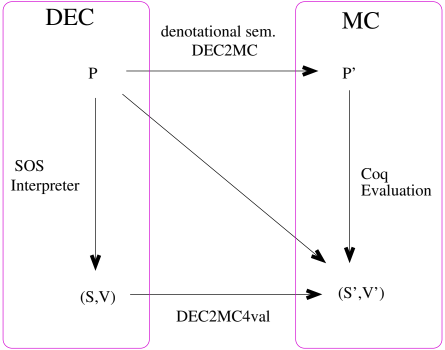
1. we build a Coq representation of MC as a deep embedding (DEC) and specify formally its semantics
   – operationally, implementing an SOS interpreter
   – denotationally, as interpretation of DEC into Gallina
2. use the denotational semantics to verify the translation of Pip into DEC
3. use the operational semantics to verify the translation to CompCert C

Translation through DEC

DEC is defined in terms of abstract datatypes: possible to manipulate it as an object in Coq – e.g. to define a formal translation from it



From DEC to MC (in Gallina) and back (in Haskell)

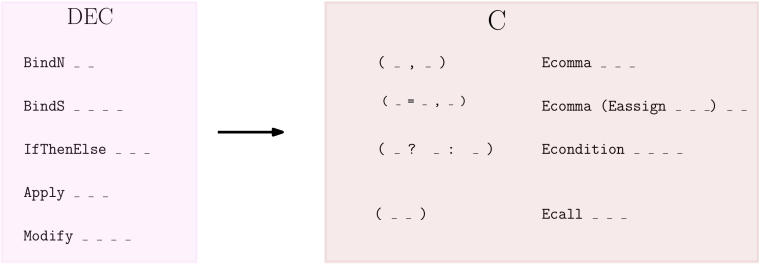


For the two semantics to agree:

for P a DEC program, DEC2MC4val (SOS_Int P) $=$ DEC2MC P

Pip $=$ DEC2MC (*Haskell_MC2DEC* Pip)

From DEC to C



Semantic soundness: need for a proof that behaviour is preserved.

Essentially – like adding a compilation step.

DEC expressions

```
Inductive Exp : Type :=
| Val (v: Value) | Var (x: Id)
| BindN (e1: Exp) (e2: Exp)
| BindS (x: Id) (t: option VTyp) (e1: Exp) (e2: Exp)
| IfThenElse (e1: Exp) (e2: Exp) (e3: Exp)
| Apply (f: Id) (prms: Prms) (fuel: Exp)
| Modify (t1 t2: VTyp) (xf: XFun t1 t2) (prm: Exp)
| BindMS (env: valEnv) (e: Exp)
| Call (f: Id) (prms: Prms)
with Prms : Type := PS (es: list Exp).
```

Recursive functions terminate (as in MC)

Modules, mutual recursion and side-effects

```
Parameter Id: Type.
Parameter State: Type.

Inductive Fun : Type :=
FC (formal_prms: list (Id * VTyp) (ret_type: VTyp)
   (default: Value) (body: Exp).

Record XFun (dt1 dt2: VTyp) : Type :=
{ x_modify : State → (mcTyp dt1) → State * (mcTyp dt2) }.
```

Operational semantics (small-step)

ϕ function environment

δ datavalue environement

Static:

$$\vdash \phi :: \Phi$$
$$\Phi; \Delta \vdash \mathit{exp} :: \mathit{vtyp}$$
$$\vdash \text{well\_typed } \phi$$

$$\vdash \delta :: \Delta$$
$$\Phi; \Delta \vdash \mathit{prms} :: \mathit{ptyp}$$

Dynamic:

$$\phi;\ \delta \Vdash (\mathit{state},\ \mathit{fuel},\ \mathit{exp}) \longrightarrow (\mathit{state}',\ \mathit{fuel}',\ \mathit{exp}')$$
$$\phi;\ \delta \Vdash (\mathit{state},\ \mathit{fuel},\ \mathit{prms}) \longrightarrow (\mathit{state}',\ \mathit{fuel}',\ \mathit{prms}')$$

Type soundness (SOS interpreter)

Type soundness for expressions (similarly for parameters):

$$
\begin{array}{ll}
\forall\ \Phi\ \Delta\ \mathit{exp}\ \mathit{vtyp}, & \Phi; \Delta \vdash \mathit{exp} :: \mathit{vtyp}\ \rightarrow \\
\forall\ \phi\ \delta\ \mathit{state}\ \mathit{fuel}, & \vdash \text{well\_typed}\ \phi\ \rightarrow \\
 & \vdash \phi :: \Phi\ \rightarrow \\
 & \vdash \delta :: \Delta\ \rightarrow \\
\multicolumn{2}{l}{\Sigma!\ \mathit{state}'\ \mathit{fuel}'\ v,} \\
\multicolumn{2}{l}{\quad \phi;\ \delta \Vdash (\mathit{state},\ \mathit{fuel},\ \mathit{exp}) \longrightarrow (\mathit{state}',\ \mathit{fuel}',\ \text{Val}\ v)}
\end{array}
$$

Proved in Coq, by double induction on fuel and the mutually defined typing relations.

Operational semantics (Coq code)

```
Inductive ExpTyping :
  list (Id*FTyp) → list (Id*Value) → Exp → VTyp → Type
with PrmsTyping :
  list (Id*FTyp) → list (Id*Value) → Prms → PTyp → Type

Inductive FEnv_WT (fenv: list (Id*Fun)) : Type

Inductive AConfig (T: Type) : Type :=
                     Conf (state: W) (fuel: nat) (qq: T)

Inductive EStep (fenv: list (Id*Fun)) :
   list (Id*FCall) → list (Id*Value) →
                         AConfig Exp → AConfig Exp → Type
with PrmsStep (fenv: list (Id*Fun)) :
   list (Id*FCall) → list (Id*Value) →
                         AConfig Prms → AConfig Prms → Type
```

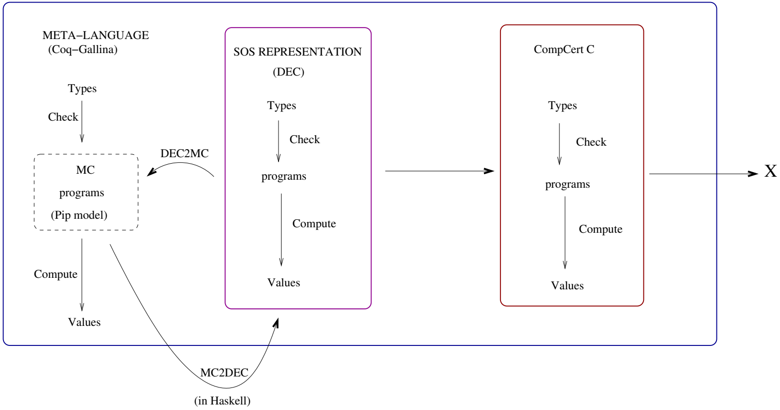
Denotational semantics

$$\Theta_e \;:\; \Theta_t\ \mathsf{funEnv} \to \Theta_t\ \mathsf{valEnv}\ \to \forall\ e : \mathsf{Exp},\ \mathsf{ILL\ State}\ (\Theta_t\ (\tau e))$$

$$
\begin{array}{ll}
\Theta_e\ \_\ \_\ (\mathsf{Val}\ v) & = \ \mathsf{ret}\ (\mathsf{ext}\ v) \\
\Theta_e\ \_\ VS\ (\mathsf{Var}\ x) & = \ \mathsf{ret}\ (\mathsf{find}\ x\ VS) \\
\ldots & \\
\Theta_e\ FS\ \ VS\ (\mathsf{BindS}\ x\ \_\ e_1\ e_2) & = \mathsf{let}\ t = \Theta_t\ (\tau e_1)\ \mathsf{in} \\
\qquad \mathsf{bind}\ \ (\Theta_e\ FS\ \ VS\ \ e_1) & (\Theta_e\ FS\ ((x, t) :: VS)\ e_2) \\
\ldots & \\
\Theta_e\ FS\ \ VS\ (\mathsf{Call}\ f\ \ prms) & = \\
\qquad \mathsf{bind}\ \ (\Theta_{es}\ FS\ \ VS\ prms) & (\mathsf{find}\ f\ \ FS) \\
\Theta_e\ FS\ \ VS\ (\mathsf{Modify}\ xf\ \ prm) & = \\
\qquad \mathsf{bind}\ \ (\Theta_e\ FS\ \ VS\ prm) & (\mathsf{x\_modify}\ xf)
\end{array}
$$

Provable in Coq: the two semantics (operational and denotational) agree

Summarising



Documentation

System:
Q. Bergougnoux, N. Jomaa, M. Yaker, J. Cartigny, G. Grimaud, S. Hym, D. Nowak, Proved Memory Isolation in Real-Time Embedded Systems through Virtualization, submitted

Formal modelling and verification of security properties:
N. Jomaa, P. Torrini, D. Nowak, G. Grimaud, Proof-oriented Design of a Separation Kernel with Minimal TCB, submitted

Translation:
P. Torrini, D. Nowak, DEC: Coq repository, https://github.com/2xs/dec.git
S. Hym, V. Oudjail, Digger: Haskell repository, https://github.com/2xs/digger

To find out more

`http://pip.univ-lille1.fr`

The Pip Development Team thanks you for your attention