Translating to C

Coq executable model and extracted OCaml code:

- needs for garbage collection
- not efficient enough

We need a translation to low level languages:

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

However: we want a verified translation to CompCert C (and so a certified compilation)

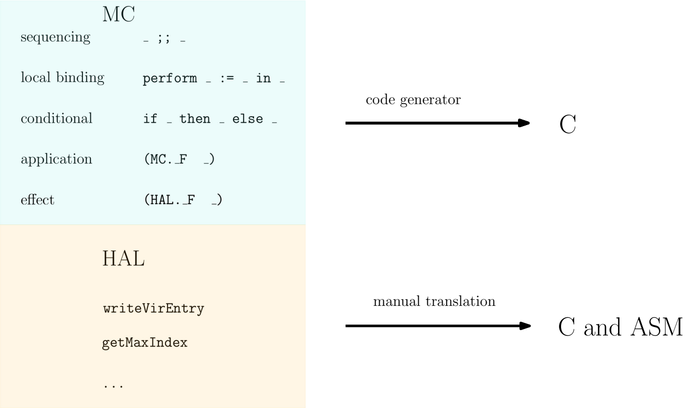
Pip monadic code (MC)

- 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
      perform daddr := getDefaultVAddr in
      writeVirEntry shadow1 idx daddr
  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.
```

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

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

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

Sample translation

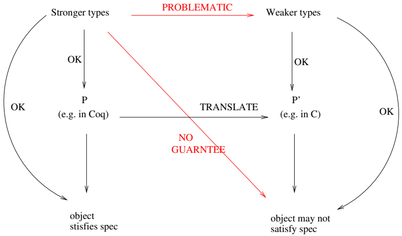
Example: a function defined in Coq, using the monadic code:

```
Definition getFstShadow (partition : page) : 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, in brief

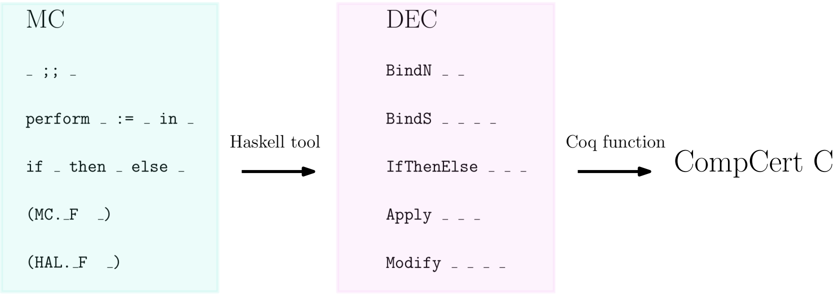
1. we build a Coq representation of MC as a deep embedding (DEC) and specify formally its semantics
   – operationally, implementing an SOS interpreter
   – denotationally, as interpretation of DEC into Gallina

2. use the denotational semantics to verify the translation of Pip into DEC

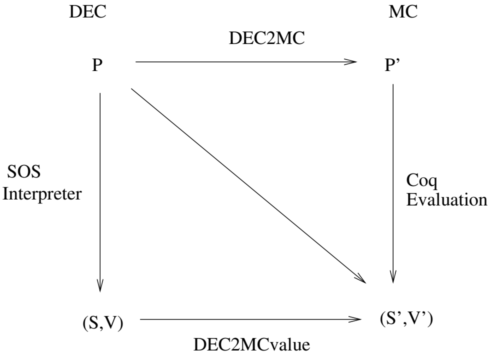
3. use the operational semantics to verify the translation to CompCert C

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 and back



for P a DEC program, $\text{DEC2MCvalue (SOS\_Int } P) \;=\; \text{DEC2MC } P$

$\text{Pip} \;=\; \text{DEC2MC (}\textit{Haskell\_MC2DEC}\text{ Pip)}$

From DEC to C

DEC

```
BindN _ _

BindS _ _ _ _

IfThenElse _ _ _

Apply _ _ _

Modify _ _ _ _
```

$\longrightarrow$

C

`( _ , _ )`	`Ecomma _ _ _`
`( _ = _ , _ )`	`Ecomma (Eassign _ _ _) _ _`
`( _ ? _ : _ )`	`Econdition _ _ _ _`
`( _ _ )`	`Ecall _ _ _`

Need for 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).
```

Modules, mutual recursion and side-effects

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

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

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

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 := ...
```

Operational semantics

ϕ function environment

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

δ datavalue environement

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

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

Type soundness

Type soundness (SOS interpreter):

$$\begin{array}{l}
\forall\ \Phi\ \Delta\ \textit{exp}\ \textit{vtyp},\ \Phi;\Delta \vdash \textit{exp} :: \textit{vtyp}\ \rightarrow \\
\forall\ \phi\ \delta\ \textit{state}\ \textit{fuel},\ \vdash \text{well_typed}\ \phi\ \rightarrow \vdash \phi :: \Phi\ \rightarrow \vdash \delta :: \Delta\ \rightarrow \\
\Sigma!\ \textit{state}'\ \textit{fuel}'\ v, \\
\quad \phi;\ \delta \Vdash (\textit{state},\ \textit{fuel},\ \textit{exp}) \longrightarrow (\textit{state}',\ \textit{fuel}',\ \text{Val}\ v)
\end{array}$$

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

Denotational semantics

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

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

Semantic soundness

Denotational semantics as delational translation from DEC to MC

```
Inductive ExpTrans : funEnvTrans -> valEnvTrans ->
                     forall T: Type, Exp -> MM W T -> Type

Inductive FunEnvTrans : funEnv -> nat -> funEnvTrans -> Type
```

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

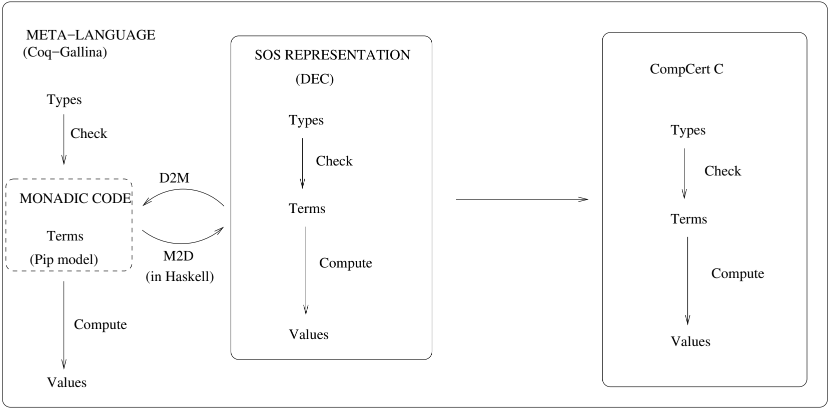
Translation to CompCert C

$$
\begin{array}{lll}
\Xi\ (\ \phi\ ;\ \delta\ \vdash\ \mathit{exp}\ :\ \mathit{vtyp}\ ) & = & \\
\text{let}\ (\mathit{cexp},\ \mathit{ctyp},\ \mathit{cdecs}) & = & \Xi_e\ \mathit{exp}\ (\Xi_t\ \Phi)\ (\Xi_t\ \Delta) \\
\text{in mk\_program}\ (\ \mathit{cexp},\quad \mathit{ctyp}, & & \text{map}\ \Xi_f\ \phi,\quad \text{map}\ \Xi_v\ \rho, \\
 & & \text{map mk\_globvar}\ \mathit{cdecs}\ )
\end{array}
$$

- assignment (=) and comma operator (,) to translate local variable binding and sequencing (BindS)
- conditional operator (?:) to translate conditional espressions (IfThenElse)
- function call (f e) to translate function application (Apply) and effectful application (Modify)

Provable in Coq: the DEC model of Pip behaves like its C translation

Summarising



To know more

P. Torrini DEC2: git repository,
https://github.com/2xs/dec/tree/master/src/DEC2

P. Torrini, D. Nowak, N. Jomaa, M. S. Cherif Formalising Executable Specifications of Low-level Systems, VSTTE'18

P. Torrini et al. DEC1: git repository,
https://github.com/2xs/dec/tree/master/src/DEC1

N. Jooma, P. Torrini, D. Nowak, G. Grimaud, S. Hym Proof-oriented Design of a Separation Kernel with Minimal Trusted Computing Base, AVoCS'18