# Translating to C an OS proto-kernel written in Coq

Paul Torrini

Université Lille 1, CRIStAL

DeepSpec Summer School, 20.07.2017

# Pip: an OS proto-kernel

proto-kernel: minimal trusted computing base for an OS

kernel mode limited to virtual memory management
(MMU configuration and context switching)

model written in Coq

verification of virtual memory isolation (work in progress)

translation to C
– currently a Haskell function to C source code
  unverified
  compiled by GCC
– to be replaced with a certified one to CompCert

# Pip: the model

model written in a shallow embedding of a C-like language using a monadic style

similarity of monadic code to C source,
1-1 match of monadic definitions and C commands (modulo types)

termination forced by timeout related to memory size

quasi-executable model, modulo specification of low-level functions (implemented in C and assembly)

well-typed monadic code implies nothing can go wrong – modulo low-level functions

isolation proof carried out on the shallow model using Hoare logic

# Adequate translation

- equivalent non-wrong behaviour:

  $\rho \Vdash P_1 \sim P_2$ iff in environment $\rho$ programs $P_1$ and $P_2$ don't go wrong and either converge to the same final state or both diverge

- adequate translation $\tau : \mathcal{L}_A \to \mathcal{L}_B$ preserves non-wrong behaviour equivalence:

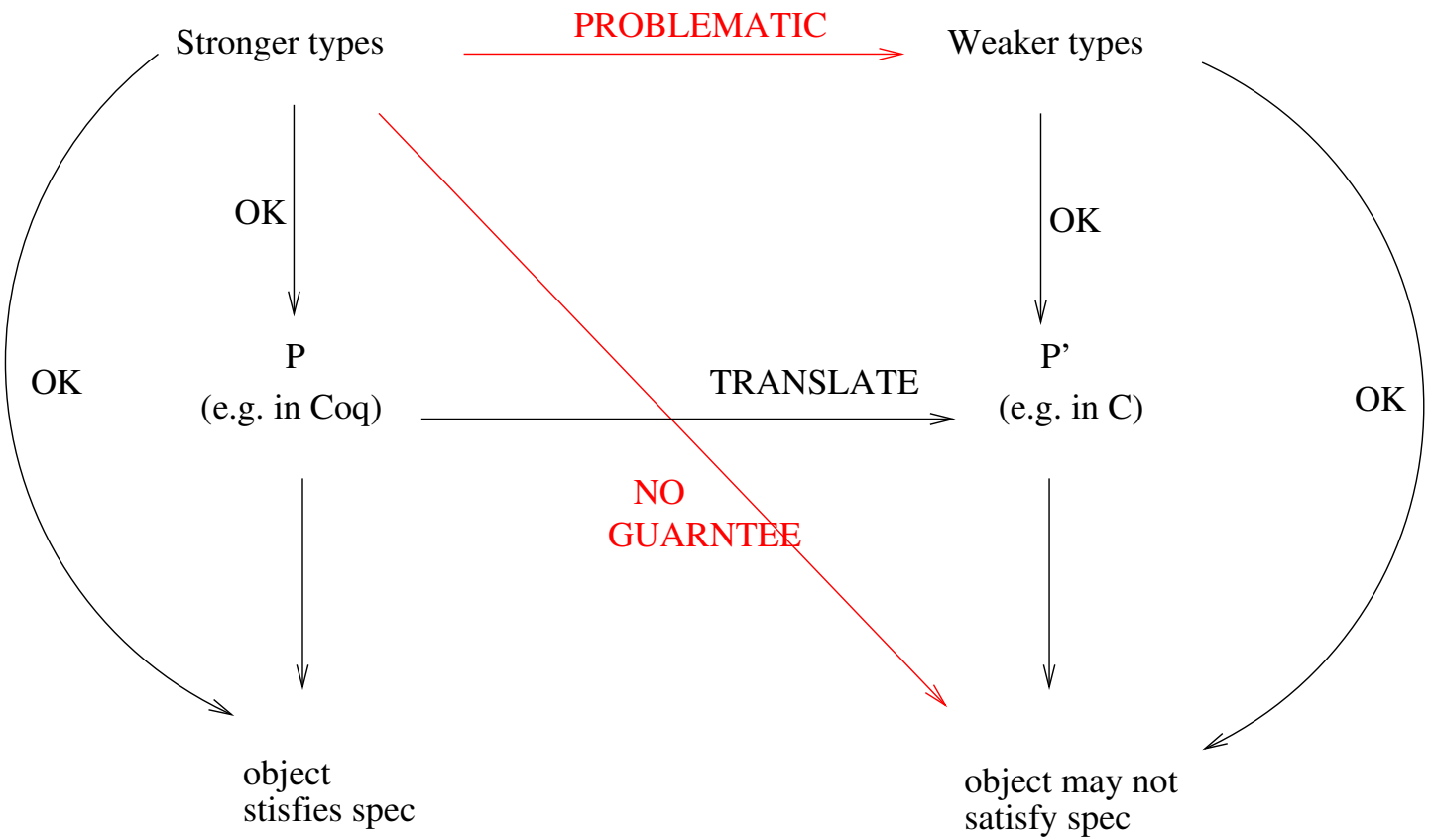
  $$\text{for each } \rho, P_1, P_2, \quad \rho \Vdash P_1 \sim P_2 \quad \to \quad \tau^v \rho \Vdash \tau P_1 \sim \tau P_2$$

- n-w equivalence: $P_1 \equiv P_2$ iff for each $\rho$, $\rho \Vdash P_1 \sim P_2$

  at best, shallow-embedding languages and translation, $\equiv$ follows from = (propositional equality)

# Translation verification problem

- translating from a safer language to a less safe one (safer = having stronger types) can make verification problematic



# Safe translation approach

- define an adequate one-to-one translation from less safe to safer, and invert it
- in our case: invert a semantic representation of the C fragment in Coq
- semantic representation includes: types, values, programs, definition of well-typedness (static semantics) and behaviour (dynamic semantics)
- monadic code could give us a denotational semantics – however, a monadic program can be equivalent to many Gallina terms, lots of maths, not very manageable

# Verified translation plan

- goal: translate adequately the monadic code (MC) to the corresponding fragment of C
- use a deep embedding (DEC) in Coq as a small intermediate language, specifying syntax and operational semantics (static and dynamic)
- interepret DEC into MC, prove equivalence by propositional equality
- translate adequately from DEC to CompCert C to obtain certified translation+compilation

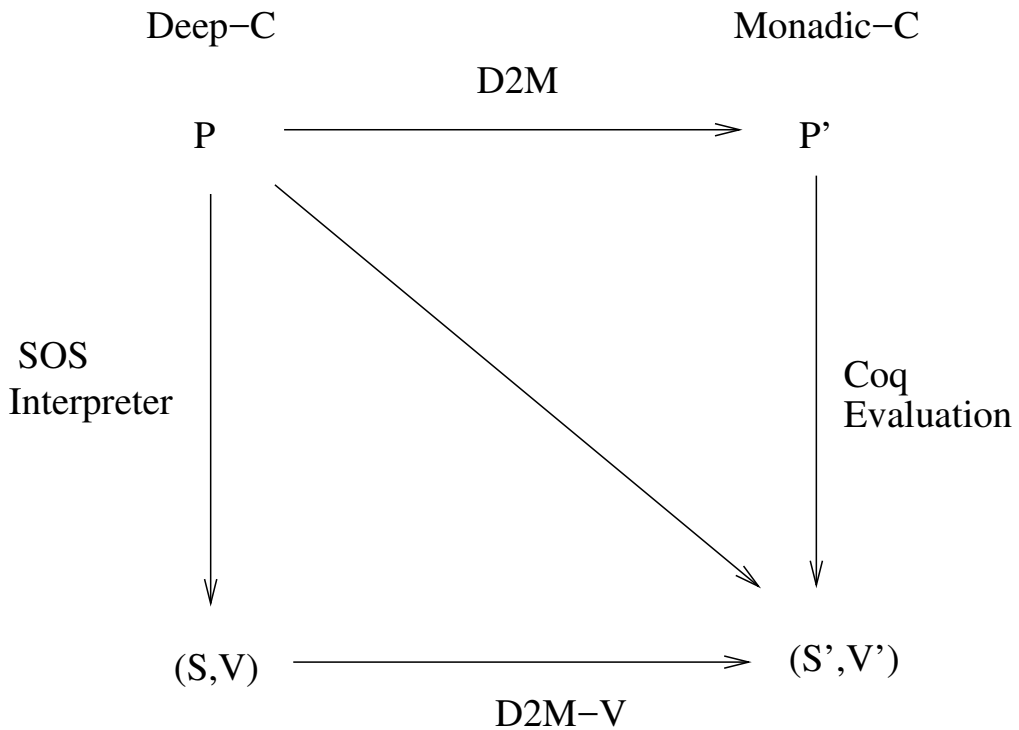
# DEC: a deeply embedded imperative language

based on structural operational semantics (SOS)

- inductive definition of the syntax (types, values and programs)
  - functions: primitive recursion, first-order, total application
  - monadic-style side effects
- inductive definition of the typing relation – ensures termination
- inductive definition of the SOS transition relation (small step), deterministic, terminanting
- an executable SOS interpreter, obtained as proof-term of type soundness
- an interpreter to MC (work in progress)

# From DEC to MC (in progress)

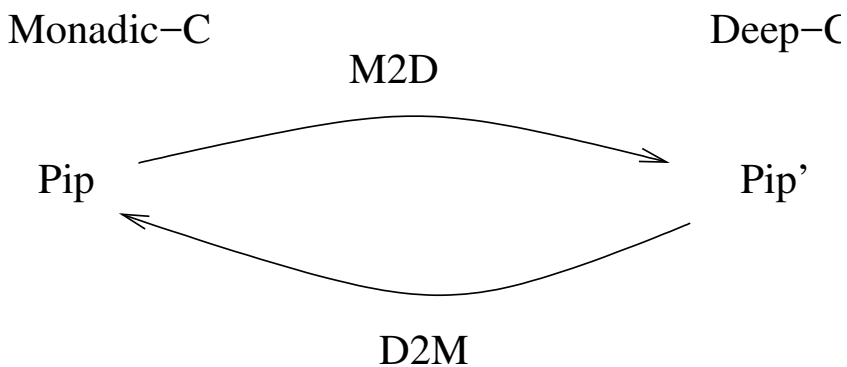
D2M to be defined in Coq and proved sound wrt the SOS interpreter




for each DEC program $p$, $D2M^V\ (SOS\_Int\ p)\ =\ D2M\ p$

# Back from MC to DC (in progress)

M2D defined in Haskell (replaces unverified translation to C)




$(D2M\ (M2D\ \mathrm{Pip})) = \mathrm{Pip}$

# DEC syntax 1

```
(* internal value type, parametrised by a semantic type *)
Inductive ValueI (T: Type) : Type := Cst (v: T).
(* value type, hiding the semantic type *)
Definition Value : Type := sigT ValueI.
(** Quasi-values *)
Inductive QValue : Type := Var (x: Id) | QV (v: Value).

(* mutual definition of program expressions
   with functions, quasi-functions and parameters *)
Inductive Fun : Type := FC (fenv: Envr Id Fun)
              (tenv: valTC) (e0 e1: Exp) (x: Id) (n: nat)
with QFun : Type := FVar (x: Id) | QF (v: Fun)
with Prms : Type := PS (es: list Exp)
```

# DEC syntax 2

```
(* program expressions *)
with Exp : Type :=
 | Lift (q: QValue)
 | BindN (e1: Exp) (e2: Exp)
 | BindS (x: Id) (e1: Exp) (e2: Exp)
 | BindMS (fenv: Envr Id Fun) (venv: valEnv) (e: Exp)
 | Apply (q: QFun) (ps: Prms)
 | IfThenElse (e1: Exp) (e2: Exp) (e3: Exp)
 | Modify (T1 T2: Type) (VT1: ValTyp T1) (VT2: ValTyp T2)
                  (XF: XFun T1 T2) (q: QValue).
```

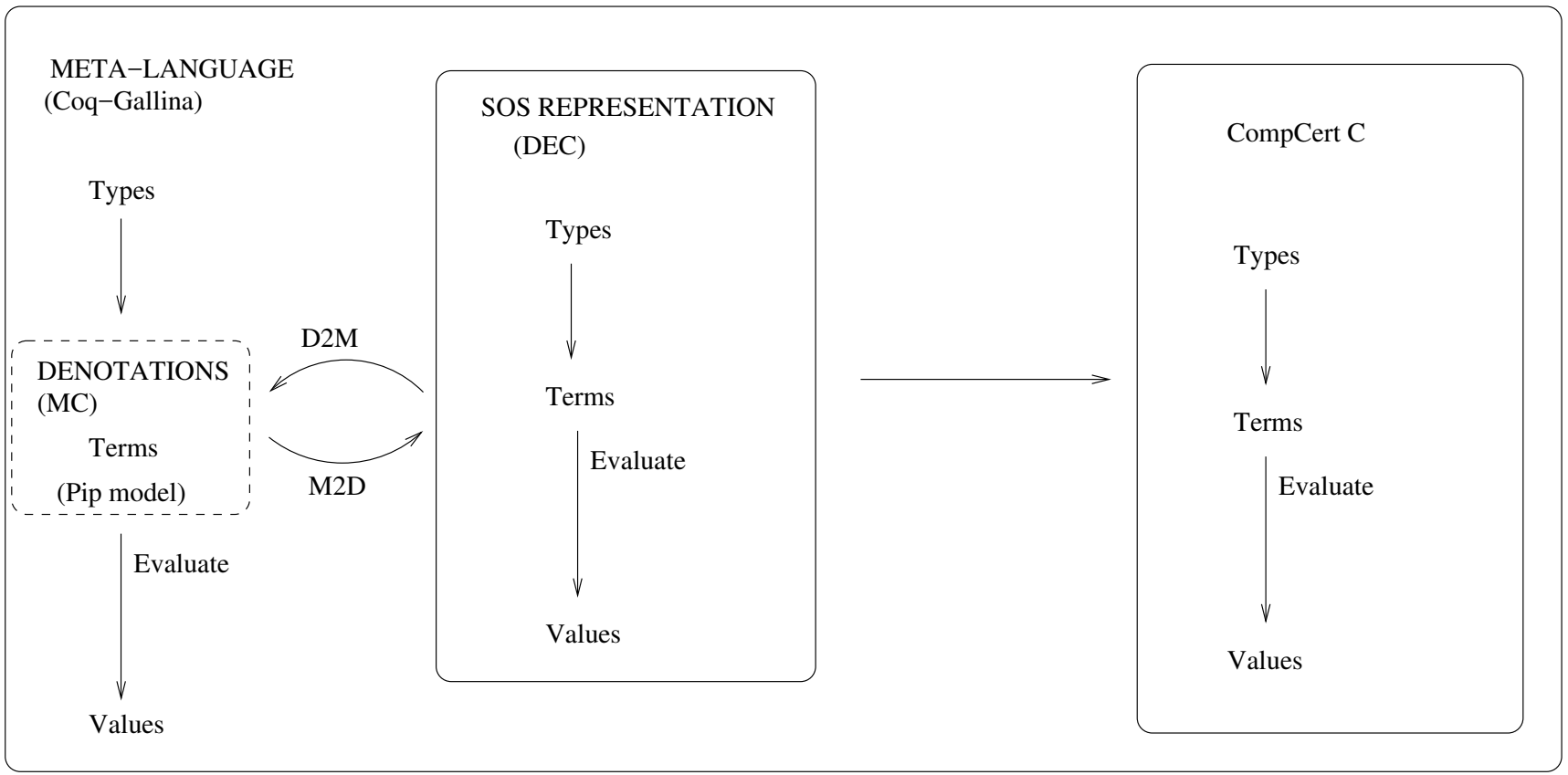
# DEC type soundness

```
(* TS *)
Lemma ExpEval (ftenv: funTC) (tenv: valTC) (fenv: funEnv)
   (e: Exp) (t: VTyp)
   (k: ExpTyping ftenv tenv fenv e t) :
   ExpSoundness ftenv tenv fenv e t k.
```

ensures that each well-typed program in a well-typed environment runs to a final state (a return value of matching type and a well-typed store), which can be extracted from the applied proof-term

TS proved by structural induction on the typing relation (relying on a form of type-based termination)

# Bigger picture



# ODSI Project

ODSI – EU Celtic project, involving Universite' Lille 1, Orange and other industrial partners

2XS Team, Universite' Lille 1 – members include

- Gilles Grimaud
- Julien Iguchi-Cartigny
- David Novak
- Samuel Hym
- Vlad Rusu
- Paul Torrini
- Narjes Jomaa
- Quentin Bergougnoux

*Thanks for your attention!*