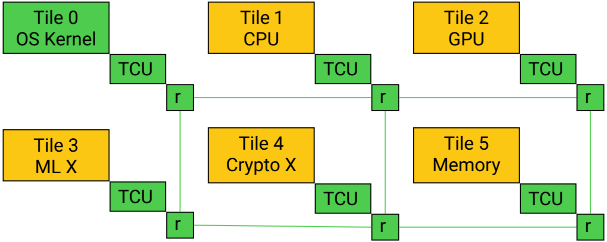
On the Potential of Coq as the Platform of Choice for Hardware Design

Sebastian Ertel, Max Kurze, Michael Raitza

Verified System Design Automation, Barkhausen Institute, Dresden, Germany

The TCB reduction argument for operating system design

- **Monolithic OS:** all hardware and a large OS kernel (with drivers, file system, network stack etc.)
- **Micro-kernel OS:** all hardware and a tiny OS kernel (w/o drivers, file system, network stack etc.)
- **M^3:** tiny hardware (called TCU) and a tiny kernel.[1]



- **Our goal:** a tiny hardware, i.e., NoC and TCU, *formally-verified* in Coq and a tiny kernel

---

[1] Nils Asmussen et al. “M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores”. In: ASPLOS ’16

Programming in Kôika

Registers: Definition

```
Inductive reg_t :=
  | r0
  | r1
  .
```

Types

```
Definition R idx := match idx with
  | r0 => bits_t 4
  | r1 => bits_t 10
end.
```

Initialization

```
Definition r idx : R idx := match idx with
  | r0 => Bits.of_nat sz 18
  | r1 => Bits.of_nat sz 18
end.
```

Functions/Actions

```
Definition _divide :
  uaction reg_t empty_ext_fn_t :=
{{ let v := read0(r0) in
   let odd := v[0b~0~0~0~0] in
   if !odd then
     write0(r1, v >> 0b~1)
   else
     write0(r1,v) }}.
```

Rules

```
Inductive rule_name_t :=
  | divide
  | multiply
  .
```

Rules $\mapsto$ Actions

```
Definition rules r := match r with
  | divide   => _divide
  | multiply => _multiply
  end.
```

Schedule

```
Definition collatz : scheduler := divide |> multiply |> done.
```

---

[1] Thomas Bourgeat et al. "The essence of Bluespec: a core language for rule-based hardware design". In: PLDI 2020

[2] This example code is due to the Kôika authors.

From Tests to Lemmas

A test with fixed inputs and register initializations:

```
Example cannot_write_feature_without_kernel_bit:

  let inputs := init_inputs in
  let r := init_registers in (* kernel bit is [0] *)

  run_function r ext_ifaces inputs handle_requests
    (fun ctxt out =>
       get_kernel_bit(ctxt.[FEATURES]) = Ob~0).
  check.
Defined.
```

A lemma reasoning generalizing over inputs and register initializations:

```
Lemma cannot_write_feature_without_kernel_bit:
  ∀ (reg:reg_t)
    (inputs:input)
    (r : R reg), get_kernel_bit(r FEATURES) = Ob~0 ->

  run_function r ext_ifaces inputs handle_requests
    (fun ctxt out =>
       get_kernel_bit(ctxt.[FEATURES]) = Ob~0).
  check.
Defined.
```

Some First Performance Numbers

- Area and timing comparison: Original Verilog design vs Kôika design (up to implementation status)
- Running on an FPGA at 100 MHz

	Timing (ps)		Area			
	WNS	WHS	LUT	FF	DSP	Nets
Verilog	4112	12	3361	1566	1	7541
Kôika	5067	5	3224	1950	0	6603

WNS – worst negative slack, WHS – worst hold slack

LUT – look up table, FF – flip flop, DSP – dig. sig. proc.

- **Promising**: Our design is on-par with the original Verilog implementation.[1]

---

[1] Oguzhan Türk. "A formally verified Hardware Design of a Communication Unit in a Micro-Kernel Operating System". MA thesis. University of Technology, Dresden, Germany, 2022. URL: `https://github.com/Barkhausen-Institut/tcu-koika/blob/main/documentation/Report/Thesis/Turk_Oguzhan_Master_Thesis.pdf`

Lessons learned

Kôika is a great abstraction for designing HW

and junior hardware engineers can get into it (easily with a little tutorial) but

- long-time HW engineers love their Verilog
- transactional execution is problematic

Notations work well for designing an EDSL but

- error messages still look rough and
- it is easier to fix broken definitions, lemmas than Notations.

Interpretation is great for writing tests

especially for HW engineers because HW tooling is not as easy but

- is tough when Kôika type checking/evaluation gets stuck.

Performance was mostly depending on `vm_compute`

- Having `vm_compute` in the types did not work well.
- Often `vm_compute` resulted in long evaluation times or failed.

Fun with `vm_compute`

- Actions and functions in Kôika are untyped and need to be type-checked.
- Type checking a Kôika term heavy relies on `vm_compute`.
- Example:

```
Definition min (size : nat) {reg_t : Type} : UInternalFunction reg_t empty_ext_fn_t :=
{{ fun min (a: bits_t size) (b: bits_t size) : bits_t size =>
     if a < b then a else b }}.
```

- Fails to type check and reports a massive (3000 lines <) term.
- **Diagnosis:** `vm_compute` does not solve but unfold decidable equality, e.g., `eq_dec size size`.

Fun with `vm_compute`

- Approaches:
  1. Evolution of the tactic for type-checking to be able to solve decidable equalities.
     - Some performance numbers:

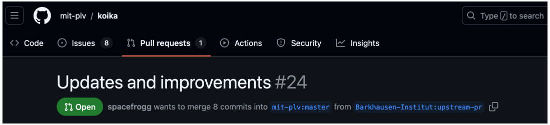
size	**tc w/** `vm_compute`	**tc with evolved tactic**
10	0.008 s	0.75 s
100	0.015 s	1.67 s
1000	0.061 s	10.2 s

     - Our evolved tactic (based on `cbn`) is 100 – 150 times slower.
  2. Direct construction of well-typed terms.
     - Drop-in replacement plus minor type changes:

```
Definition min {reg_t} {R : reg_t -> type} (size : nat) : function R empty_Sigma :=
{{ fun min (a: bits_t size) (b: bits_t size) : bits_t size =>
     if a < b then a else b }}.
```

     - **And** type checking performance is on-par with `vm_compute`.
     - PR coming soon!

Kôika Updates / Improvements



Updates and improvements #24

Open spacefrogg wants to merge 8 commits into `mit-plv:master` from `Barkhausen-Institut:upstream-pr`

- Coq 8.18 / OCaml 4.14
  - Add necessary scope hints
  - Update proofs and OCaml code
- Tool chain → dune 3
  - Generate tests/examples rules w/ OCaml (retires `etc/configure`)
  - Future: decouple `cuttlec` and experiment `Makefile`
- Stable development env. + CI
  - Nix flake + Makes CI runner
  - Provides software provenance
  - Future: integrate with `Coq nix toolbox`

Outlook and Research Directions

- Introduce tactics into Kôika for more complex proofs.
- Try MetaCoq for implementating the dynamics of a NoC.
- Introduce compiler transformations to
  - reduce the transactional overheads and
  - optimize Kôika programs.
- Towards composition of Kôika designs and
- Hardware modules.