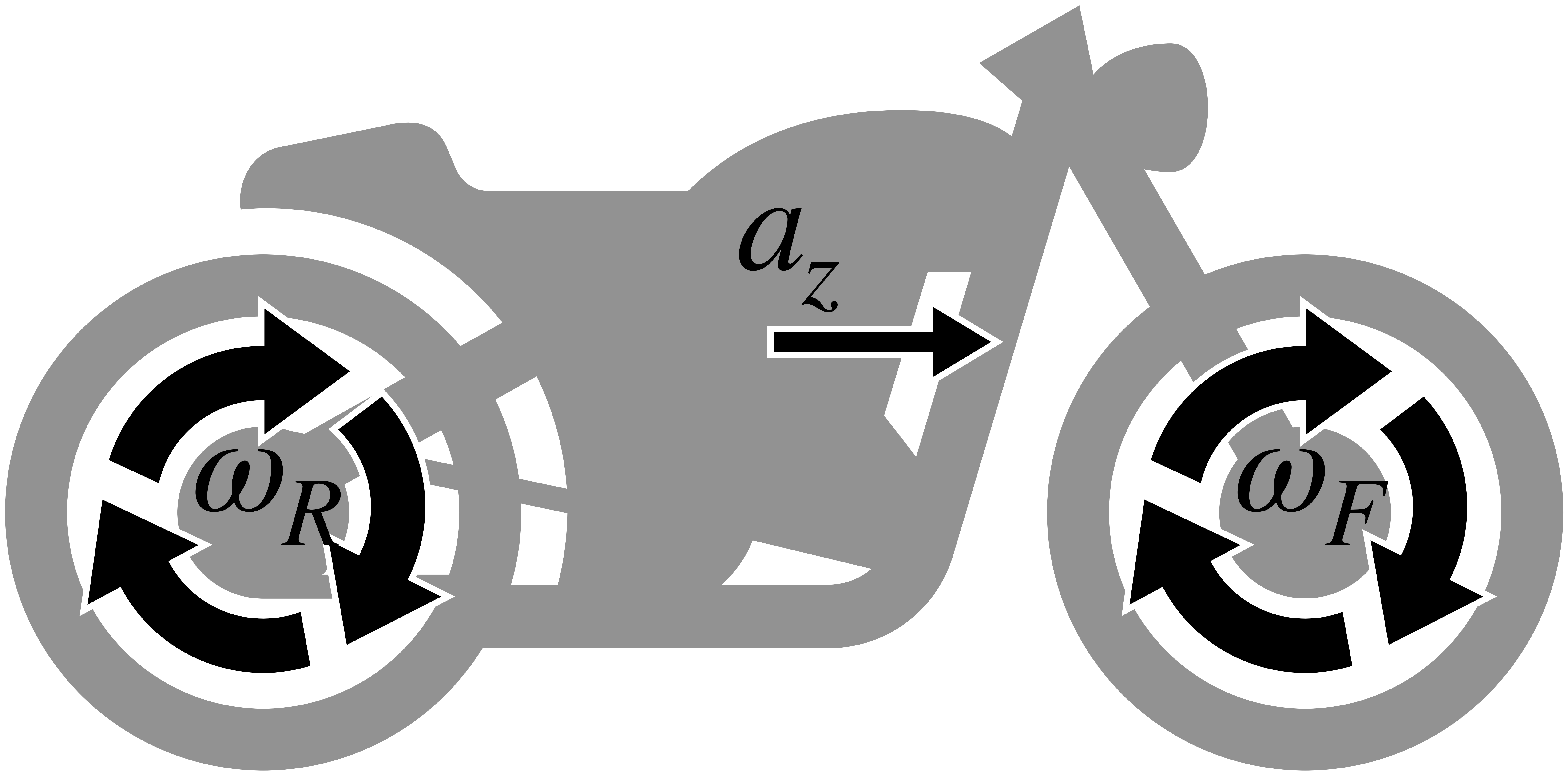
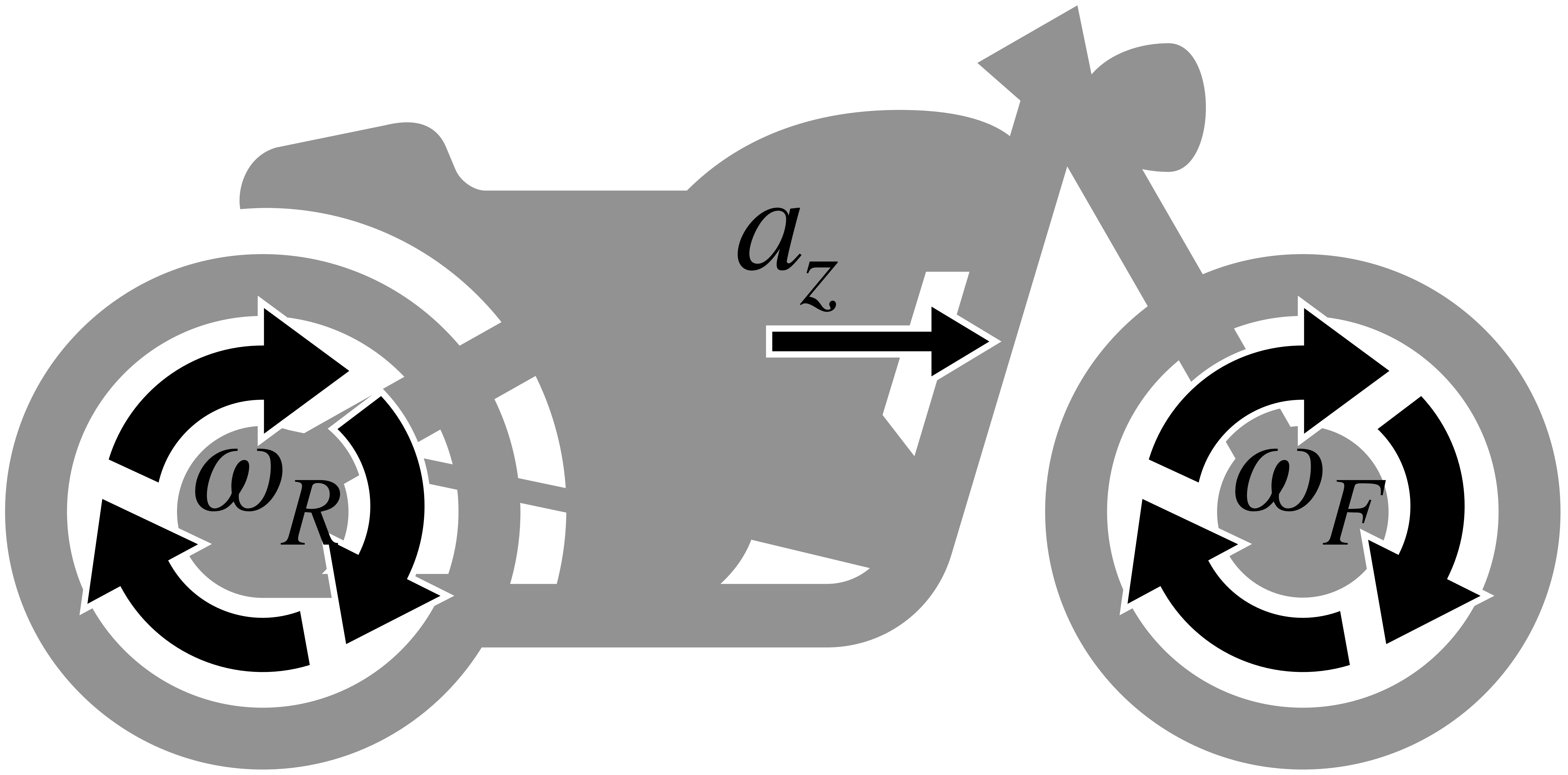
Pipit: reactive systems in F*

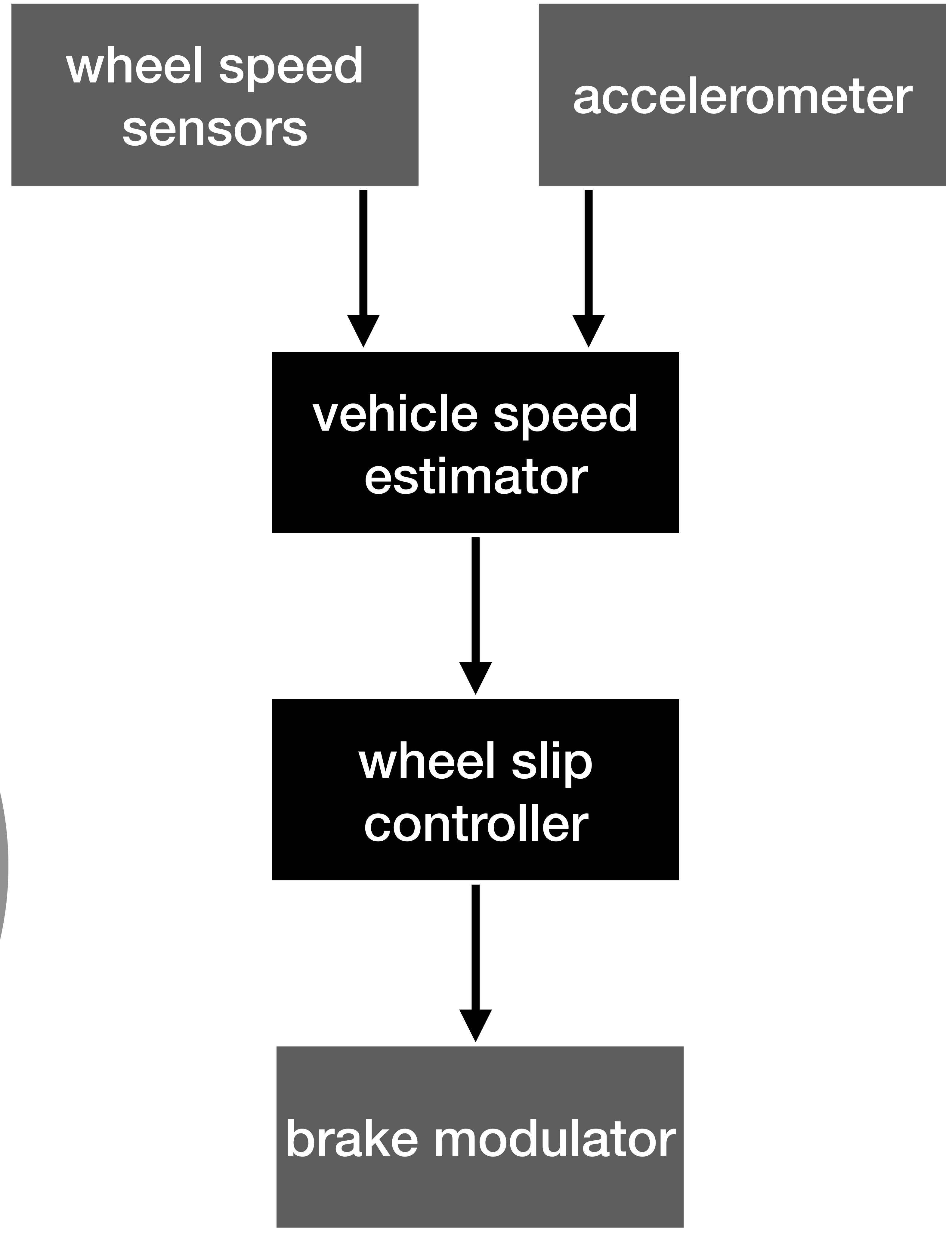
Amos Robinson, August 2023

Anti-lock brakes for a motorcycle



Anti-lock brakes for a motorcycle





Vehicle speed estimator

let veh_speed_estimator $\omega_F\ \omega_R\ a_z\ \lfloor \hat{v} \rfloor\ \lceil \hat{v} \rceil$ =

…called every 10ms…

let $\lfloor \hat{v}' \rfloor$ = …updated lower bound… **in**

let $\lceil \hat{v}' \rceil$ = …updated upper bound… **in**

$(\lfloor \hat{v}' \rfloor, \lceil \hat{v}' \rceil)$

Vehicle speed estimator

let veh_speed_estimator $\omega_F\ \omega_R\ a_z\ \lfloor\hat{v}\rfloor\ \lceil\hat{v}\rceil$ =

let $v_F = \omega_F \cdot$ radius **in**

let $v_R = \omega_R \cdot$ radius **in**

let $\lfloor\hat{v}'\rfloor$ = **if** $v_F \approx_\epsilon v_R$ **then** $\min v_F\ v_R$ **else** $\lfloor\hat{v}\rfloor + a_z - \epsilon$ **in**

let $\lceil\hat{v}'\rceil$ = **if** $v_F \approx_\epsilon v_R$ **then** $\max v_F\ v_R$ **else** $\lceil\hat{v}\rceil + a_z + \epsilon$ **in**

$(\lfloor\hat{v}'\rfloor, \lceil\hat{v}'\rceil)$

What properties do we want our estimator to have?

- if the wheels agree, the estimate is pretty good

$$v_F \approx_\epsilon v_R \implies \lfloor \hat{v}' \rfloor \approx_\epsilon \lceil \hat{v}' \rceil$$

What properties do we want our estimator to have?

- if the wheels agree, the estimate is pretty good

  $v_F \approx_\epsilon v_R \implies \lfloor \hat{v}' \rfloor \approx_\epsilon \lceil \hat{v}' \rceil$

  easy proof:

  $\lfloor \hat{v}' \rfloor$ = **if** $v_F \approx_\epsilon v_R$ **then** $\min v_F \ v_R$ **else** …

  $\lceil \hat{v}' \rceil$ = **if** $v_F \approx_\epsilon v_R$ **then** $\max v_F \ v_R$ **else** …

What properties do we want our estimator to have?

- if the wheels agreed within time t, the estimate is not *too* bad

$$\blacklozenge_t (v_F \approx_\epsilon v_R) \implies \lfloor \hat{v}' \rfloor \approx_{t\epsilon} \lceil \hat{v}' \rceil$$

What properties do we want our estimator to have?

- if the wheels agreed within time t, the estimate is not *too* bad

$$\blacklozenge_t (v_F \approx_\epsilon v_R) \implies \lfloor \hat{v}' \rfloor \approx_{t\epsilon} \lceil \hat{v}' \rceil$$

how do we even state this? not trivial!

val veh_speed_estimator (ω_F ω_R: wheel) (a_z: accel) ($\lfloor \hat{v} \rfloor$ $\lceil \hat{v} \rceil$: vel)
: (vel & vel)

As a reactive system

let node veh_speed_estimator ω_F ω_R a_z=

let $v_F = \omega_F \cdot$ radius **in**

let $v_R = \omega_R \cdot$ radius **in**

let rec $\lfloor \hat{v} \rfloor$ = **if** $v_F \approx_\epsilon v_R$

then $\min v_F \ v_R$

else $(\min v_F \ v_R \rightarrow$ **pre** $\lfloor \hat{v} \rfloor) + a_z - \epsilon$ **in**

let rec $\lceil \hat{v} \rceil$ = **if** $v_F \approx_\epsilon v_R$

then $\max v_F \ v_R$

else $(\max v_F \ v_R \rightarrow$ **pre** $\lceil \hat{v} \rceil) + a_z + \epsilon$ **in**

$(\lfloor \hat{v} \rfloor, \lceil \hat{v} \rceil)$

As a reactive system

let node veh_speed_estimator $\omega_F\ \omega_R\ a_z$=

let $v_F = \omega_F \cdot$ radius **in**

let $v_R = \omega_R \cdot$ radius **in**

let rec $\lfloor \hat{v} \rfloor$ = **if** $v_F \approx_\epsilon v_R$

then $\min v_F\ v_R$

else $(\min v_F\ v_R \rightarrow$ **pre** $\lfloor \hat{v} \rfloor) + a_z - \epsilon$ **in**

let rec $\lceil \hat{v} \rceil$ = **if** $v_F \approx_\epsilon v_R$

then $\max v_F\ v_R$

else $(\max v_F\ v_R \rightarrow$ **pre** $\lceil \hat{v} \rceil) + a_z + \epsilon$ **in**

check ($\blacklozenge_t (v_F \approx_\epsilon v_R) \implies \lfloor \hat{v} \rfloor \approx_{t\epsilon} \lceil \hat{v} \rceil$);

($\lfloor \hat{v} \rfloor$, $\lceil \hat{v} \rceil$)

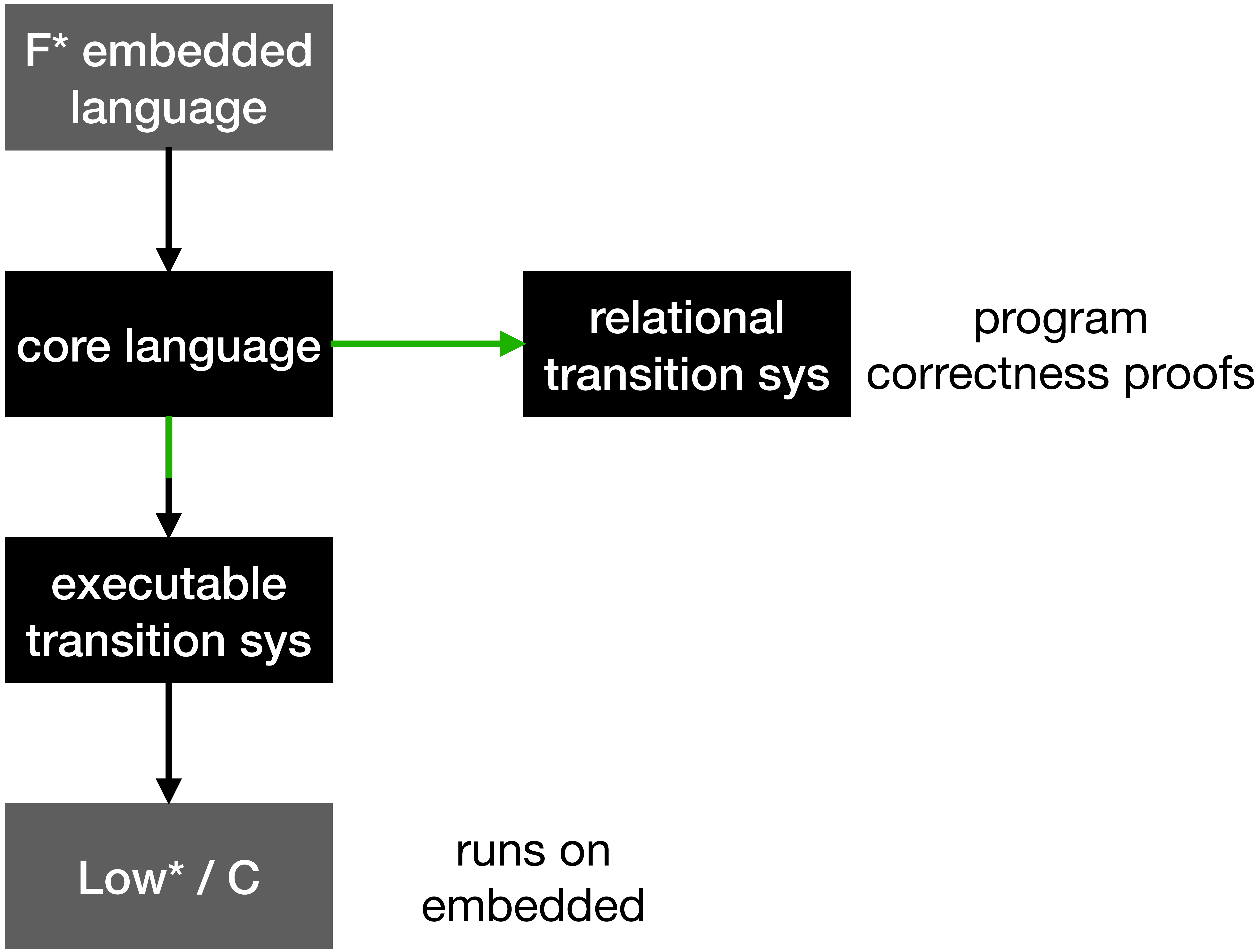
Past-time modal logic

let node $\blacklozenge_t(p) =$

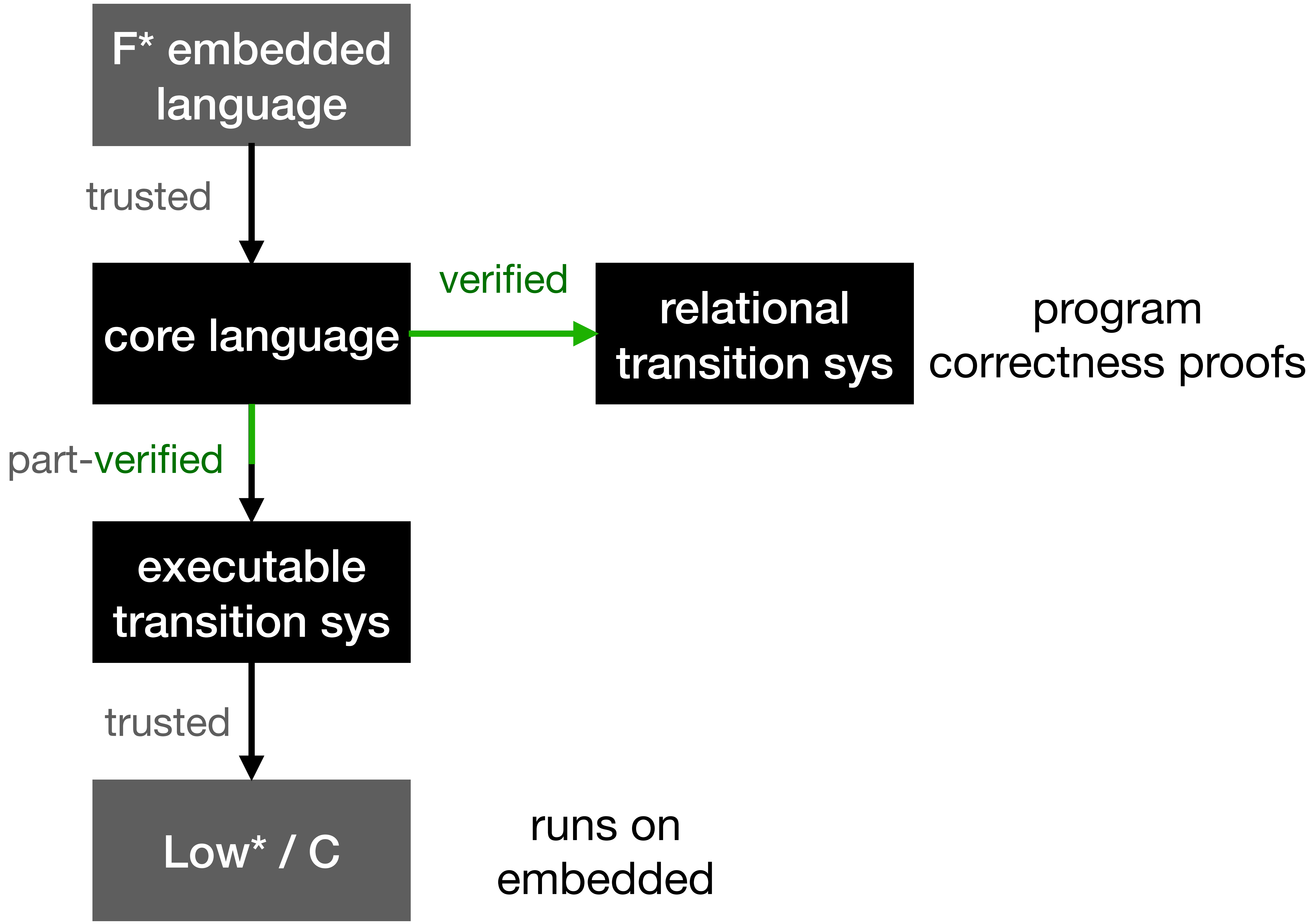
if $t = 0$ **then** false **else** $(p \vee \blacklozenge_{t-1}$ (false $\rightarrow$ **pre** p))

Language

Pipit structure



Pipit structure



Core language

$$e :=$$

$$| \; v \; | \; x \; | \; p(e\ldots)$$

$$| \; \text{pre } e$$

$$| \; e \to e'$$

$$| \; \mu x. \; e[x]$$

$$| \; \text{let } x = e \text{ in } e'[x]$$

$$| \; \text{check } e$$

Core language

e :=

$\mid v \mid x \mid p(e\ldots)$

$\mid$ pre e

$\mid e \to e'$

$\mid \mu x.\ e[x]$

$\mid$ let $x = e$ in $e'[x]$

$\mid$ check e

$$\frac{}{\Sigma \vdash v \Downarrow v}\ \text{(Value)}$$

$$\frac{}{\Sigma_{\perp}; \sigma \vdash x \Downarrow \sigma(x)}\ \text{(Var)}$$

Core language

e :=

$\mid v \mid x \mid p(e\ldots)$

$\mid$ pre e

$\mid e \to e'$

$\mid \mu x.\ e[x]$

$\mid$ let $x = e$ in $e'[x]$

$\mid$ check e

$$\frac{\Sigma \vdash e \Downarrow v}{\Sigma; \sigma \vdash \text{pre } e \Downarrow v} \text{ (Pre)}$$

Core language

$e :=$

$| \, v \, | \, x \, | \, p(e \ldots)$

$| \, \text{pre } e$

$| \, e \rightarrow e'$

$| \, \mu x. \; e[x]$

$| \, \text{let } x = e \text{ in } e'[x]$

$| \, \text{check } e$

$$\frac{\sigma \vdash e \Downarrow v}{\sigma \vdash e \rightarrow e' \Downarrow v} \; (\rightarrow_1)$$

$$\frac{\Sigma; \sigma \vdash e' \Downarrow v'}{\Sigma; \sigma \vdash e \rightarrow e' \Downarrow v'} \; (\rightarrow_S)$$

Core language

$$
\begin{aligned}
e \; := \; & \\
& \mid v \mid x \mid p(e\ldots) \\
& \mid \text{pre } e \\
& \mid e \rightarrow e' \\
& \mid \mu x.\; e[x] \\
& \mid \text{let } x = e \text{ in } e'[x] \\
& \mid \text{check } e
\end{aligned}
$$

$$
\frac{\Sigma \vdash e[x := \mu x.\, e] \Downarrow v}{\Sigma \vdash \mu x.\, e \Downarrow v}\;(\mu)
$$

Core language

$e :=$

$| v | x | p(e\ldots)$

$|$ pre e

$| e \to e'$

$| \mu x.\ e[x]$

$|$ let $x = e$ in $e'[x]$

$|$ check e

$$\frac{\Sigma \vdash e'[x := e] \Downarrow v}{\Sigma \vdash \text{let } x = e \text{ in } e' \Downarrow v} \text{ (Let)}$$

Core language

e :=

$\mid v \mid x \mid p(e\ldots)$

$\mid$ pre e

$\mid e \to e'$

$\mid \mu x.\ e[x]$

$\mid$ let $x = e$ in $e'[x]$

$\mid$ check e

$$\frac{}{\Sigma \vdash \text{check } e \Downarrow ()}\ \text{(Check)}$$

Metatheory: evaluation

$$\frac{\Sigma \text{ non-empty} \qquad e \text{ causal}}{\exists v \,.\, \Sigma \vdash e \Downarrow v}$$

Metatheory: causality

$\mu x.\ x + 1$ not causal

$\mu x.\ (0 \to \text{pre}\ (x + 1))$ causal

Implementation

Executable transition systems

type exp $\Gamma\ \tau$ =
  | Var: index $\Gamma\ \tau \rightarrow$ exp $\Gamma\ \tau$
  | Value: $\tau \rightarrow$ exp $\Gamma\ \tau$
  | Pre: exp $\Gamma\ \tau \rightarrow$ exp $\Gamma\ \tau$
  | …

type system $\Gamma\ s\ \tau$ = {
  init: s;
  step: row $\Gamma \rightarrow s \rightarrow (s \times \tau)$
}

Executable transition systems

```
type exp Γ τ =
  | Var:  index Γ τ → exp Γ τ
  | Value:         τ → exp Γ τ
  | Pre:    exp Γ τ → exp Γ τ
  | …

type system Γ s τ = {
  init:   s;
  step: row Γ → s → (s × τ)
}

let rec state_of_exp (e: exp Γ τ): Type =
  match e with
  | Var _     → unit
  | Value _ → unit
  | Pre e'   → τ × state_of_exp e'
  | …

let rec system_of_exp (e: exp Γ τ): system Γ (state_of_exp e) τ =
  match e with
  | …
```

Code generation: Low*

```
type system Γ s τ = {
  init:   s;
  step: row Γ → s → (s × τ)
}

let mk_reset (t: system Γ s τ) (stref: pointer s): ST unit
    (requires (fun h        → live h  stref))
    (ensures (fun h _ h'  → live h' stref)) =
  stref *= t.init

let mk_step (t: system Γ s τ) (input: row Γ) (stref: pointer s): ST τ
    (requires (fun h        → live h  stref))
    (ensures (fun h _ h'  → live h' stref)) =
  let st          = !*stref in
  let (st', res) = t.step inp st in
  stref *= st';
  res
```

Code generation: extraction

```
let state: Type =
  state_of_exp veh_speed_estimator

noextract
let sys: system [vel; vel; accel] state (vel × vel) =
 system_of_exp veh_speed_estimator

[@@(postprocess_with tac_normalize)]
let reset: pointer state → ST unit =
  mk_reset sys

[@@(postprocess_with tac_normalize)]
let step: row [vel; vel; accel] → pointer state → ST (vel × vel) =
  mk_step sys
```

Code generation: extraction

```
typedef struct state_s {
  ...
} state;

typedef struct result_s {
  vel fst;
  vel snd;
} result;

void reset(state *stref);

result step(input inp, state *stref);
```

Trust

Trusted computing base

Pipit (compiler+checker)

- OCaml
- F*
- Z3
- F*->C translation

- remaining proofs…

Vélus (compiler)

- OCaml
- Coq

- +CompCert

Kind2 (checker)

- OCaml
- Kind2 impl
- Z3

- +compiler