

# Calculating Correct and Efficient Programs

David Castro-Perez

David.CastroPerez@rhul.ac.uk

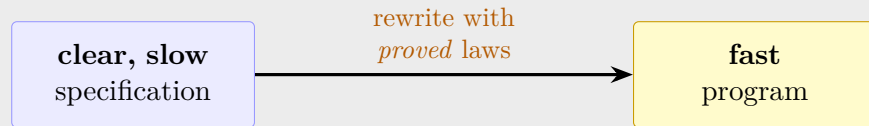


Based on joint work with Marco Paviotti and Michael Vollmer (University of Kent):  
*Program Optimisations via Hylomorphisms for Extraction of Executable Code*, ITP 2025.

slide 1 / 83

- Joint work with Marco Paviotti and Michael Vollmer (ITP 2025)
- Theme: fast *and* provably correct, without a new proof per program

## In One Slide



### The idea (1980s):

- Programs as algebra
- Optimise by rewriting
- Correct by construction

### Our work:

- Every step machine-checked
- No extra axioms
- Runnable code out

## In One Slide

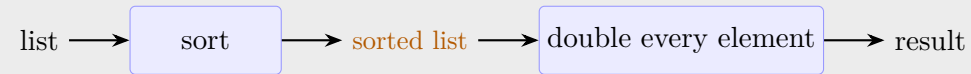
slide 2 / 83

- Point at the arrow: spec in, fast program out
- Each step is an equation, so the result equals the spec
- Old idea (Bird, Meertens); new: every step checked by Rocq, code you can run

# The Problem

Fast *and* Correct

## A Specification and a Program



## A Specification and a Program

slide 4 / 83    step 1 of 3

- **Spec: sort, then double. Two passes, a throwaway list**
- ▷ Program: doubles each pivot as it goes. One pass
- ▷ Equal? Ask the audience. Goal: derive it, not check it

## A Specification and a Program



```
let rec sort_times_two = function
| [] -> []
| h :: t ->
  let (l, r) = partition (fun x -> x < h) t in
  sort_times_two l @ (h * 2) :: sort_times_two r
```

## A Specification and a Program

slide 5 / 83    step 2 of 3

- Spec: sort, then double. Two passes, a throwaway list
- ▷ **Program: doubles each pivot as it goes. One pass**
- ▷ Equal? Ask the audience. Goal: derive it, not check it

## A Specification and a Program



```
let rec sort_times_two = function
| [] -> []
| h :: t ->
  let (l, r) = partition (fun x -> x < h) t in
  sort_times_two l @ (h * 2) :: sort_times_two r
```

Are they equal? Can we *derive* the second from the first?

## A Specification and a Program

slide 6 / 83    step 3 of 3

- Spec: sort, then double. Two passes, a throwaway list
- ▷ Program: doubles each pivot as it goes. One pass
- ▷ **Equal? Ask the audience. Goal: derive it, not check it**

## Three Ways to Fast and Correct Code

| Approach                        | Good        | But                           |
|---------------------------------|-------------|-------------------------------|
| Write it, then <b>verify</b> it | any program | a new proof for every program |

## Three Ways to Fast and Correct Code

slide 7 / 83    step 1 of 3

- **Verify afterwards: works for anything, but a new proof each time**
- ▷ Verified compilers (CompCert, CakeML): faithful, but keep the algorithm
- ▷ Calculation: proof is a by-product; needs general laws

## Three Ways to Fast and Correct Code

| Approach                        | Good                   | But                                           |
|---------------------------------|------------------------|-----------------------------------------------|
| Write it, then <b>verify</b> it | any program            | a new proof for every program                 |
| <b>Verified compiler</b>        | translation is correct | keeps your algorithm: $O(n^3)$ stays $O(n^3)$ |

## Three Ways to Fast and Correct Code

slide 8 / 83    step 2 of 3

- Verify afterwards: works for anything, but a new proof each time
- ▷ **Verified compilers (CompCert, CakeML): faithful, but keep the algorithm**
- ▷ Calculation: proof is a by-product; needs general laws

## Three Ways to Fast and Correct Code

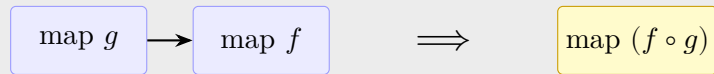
| Approach                        | Good                            | But                                           |
|---------------------------------|---------------------------------|-----------------------------------------------|
| Write it, then <b>verify</b> it | any program                     | a new proof for every program                 |
| <b>Verified compiler</b>        | translation is correct          | keeps your algorithm: $O(n^3)$ stays $O(n^3)$ |
| <b>Calculate</b> it from a spec | proof comes with the derivation | needs laws that <i>apply</i>                  |

## Three Ways to Fast and Correct Code

slide 9 / 83    step 3 of 3

- Verify afterwards: works for anything, but a new proof each time
- ▷ Verified compilers (CompCert, CakeML): faithful, but keep the algorithm
- ▷ **Calculation: proof is a by-product; needs general laws**

## You Already Trust Rewrite Rules



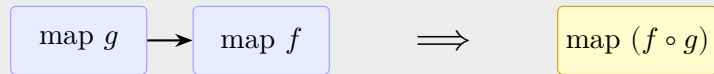
- ML compilers *fuse* tensor operations into one kernel.

## You Already Trust Rewrite Rules

slide 10 / 83    step 1 of 4

- **Map fusion: two passes become one**
- **ML compilers fuse tensor ops (XLA, TVM)**
- ▷ query optimisers
- ▷ compilers: inlining, loop fusion; GHC rewrite rules
- ▷ but: fixed rules, trusted; a wrong rule miscompiles silently

## You Already Trust Rewrite Rules



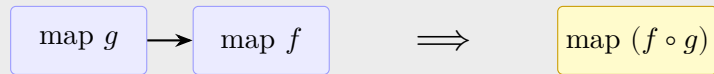
- ML compilers *fuse* tensor operations into one kernel.
- Database engines rewrite query plans.

## You Already Trust Rewrite Rules

slide 11 / 83    step 2 of 4

- Map fusion: two passes become one
- ML compilers fuse tensor ops (XLA, TVM)
- ▶ **query optimisers**
- ▶ compilers: inlining, loop fusion; GHC rewrite rules
- ▶ but: fixed rules, trusted; a wrong rule miscompiles silently

## You Already Trust Rewrite Rules



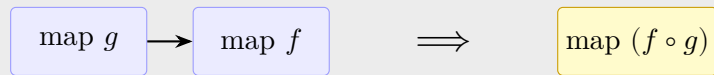
- ML compilers *fuse* tensor operations into one kernel.
- Database engines rewrite query plans.
- Compilers inline, fuse loops, remove temporary data.

## You Already Trust Rewrite Rules

slide 12 / 83    step 3 of 4

- Map fusion: two passes become one
- ML compilers fuse tensor ops (XLA, TVM)
- ▷ query optimisers
- ▷ **compilers: inlining, loop fusion; GHC rewrite rules**
- ▷ but: fixed rules, trusted; a wrong rule miscompiles silently

## You Already Trust Rewrite Rules



- ML compilers *fuse* tensor operations into one kernel.
- Database engines rewrite query plans.
- Compilers inline, fuse loops, remove temporary data.
- Usually: a **fixed** set of rules, **trusted**, not proved.

## You Already Trust Rewrite Rules

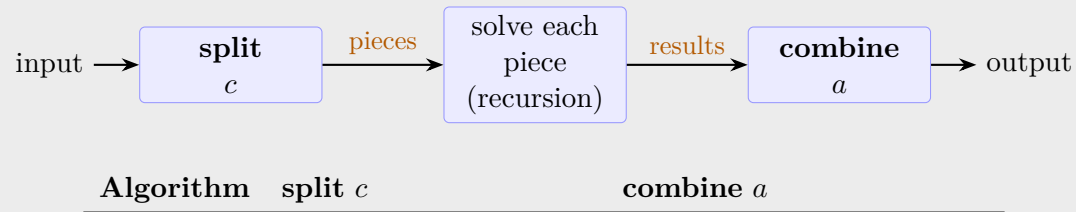
slide 13 / 83    step 4 of 4

- Map fusion: two passes become one
- ML compilers fuse tensor ops (XLA, TVM)
- ▷ query optimisers
- ▷ compilers: inlining, loop fusion; GHC rewrite rules
- ▷ **but: fixed rules, trusted; a wrong rule miscompiles silently**

# The Laws

Divide and Conquer as Algebra

## One Shape for Many Algorithms

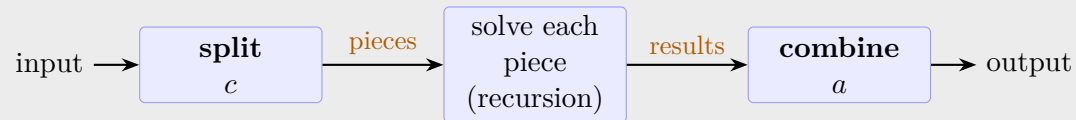


## One Shape for Many Algorithms

slide 15 / 83    step 1 of 5

- **Pipeline: split, recurse, combine**
- ▷ quicksort: partition, concatenate
- ▷ mergesort: the work moves to combine
- ▷ power: split off the last bit
- ▷ recursion written once: the hylomorphism

## One Shape for Many Algorithms



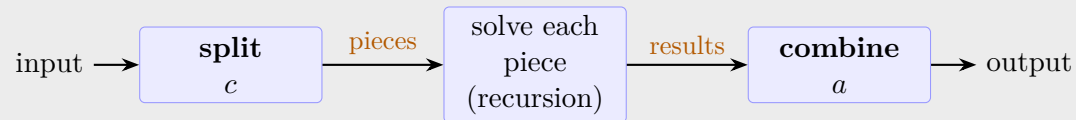
| Algorithm | split $c$                | combine $a$ |
|-----------|--------------------------|-------------|
| quicksort | partition around a pivot | concatenate |

## One Shape for Many Algorithms

slide 16 / 83    step 2 of 5

- Pipeline: split, recurse, combine
- ▷ **quicksort: partition, concatenate**
- ▷ mergesort: the work moves to combine
- ▷ power: split off the last bit
- ▷ recursion written once: the hylomorphism

## One Shape for Many Algorithms



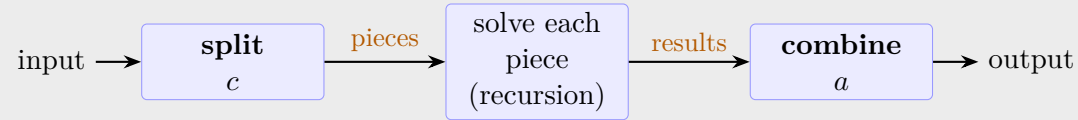
| Algorithm | split $c$                | combine $a$ |
|-----------|--------------------------|-------------|
| quicksort | partition around a pivot | concatenate |
| mergesort | cut in half              | merge       |

## One Shape for Many Algorithms

slide 17 / 83    step 3 of 5

- Pipeline: split, recurse, combine
- ▷ quicksort: partition, concatenate
- ▷ **mergesort: the work moves to combine**
- ▷ power: split off the last bit
- ▷ recursion written once: the hylomorphism

## | One Shape for Many Algorithms



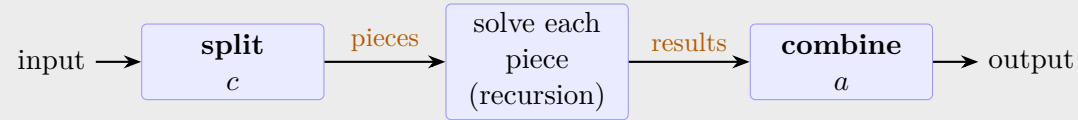
| Algorithm | split $c$                   | combine $a$                   |
|-----------|-----------------------------|-------------------------------|
| quicksort | partition around a pivot    | concatenate                   |
| mergesort | cut in half                 | merge                         |
| $x^n$     | $n \mapsto$ last bit, $n/2$ | square, maybe multiply by $x$ |

## One Shape for Many Algorithms

slide 18 / 83    step 4 of 5

- Pipeline: split, recurse, combine
- ▷ quicksort: partition, concatenate
- ▷ mergesort: the work moves to combine
- ▷ **power: split off the last bit**
- ▷ recursion written once: the hylomorphism

## | One Shape for Many Algorithms



| Algorithm | split $c$                   | combine $a$                   |
|-----------|-----------------------------|-------------------------------|
| quicksort | partition around a pivot    | concatenate                   |
| mergesort | cut in half                 | merge                         |
| $x^n$     | $n \mapsto$ last bit, $n/2$ | square, maybe multiply by $x$ |

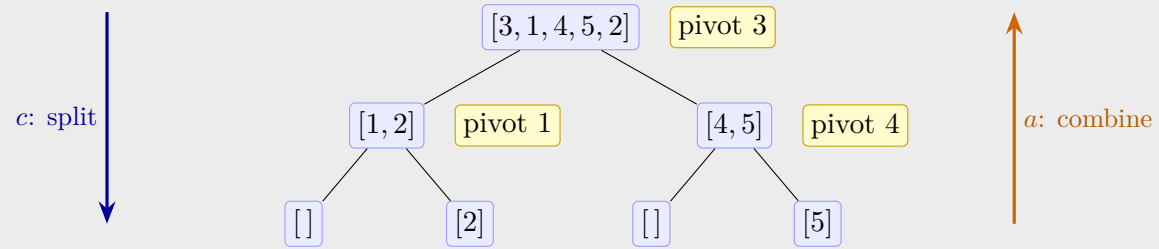
**Hylomorphism**  $\llbracket a, c \rrbracket$ : the recursion is *generic*; only  $a$  and  $c$  change.

## One Shape for Many Algorithms

slide 19 / 83    step 5 of 5

- Pipeline: split, recurse, combine
- ▷ quicksort: partition, concatenate
- ▷ mergesort: the work moves to combine
- ▷ power: split off the last bit
- ▷ **recursion written once: the hylomorphism**

## The Call Tree of Quicksort

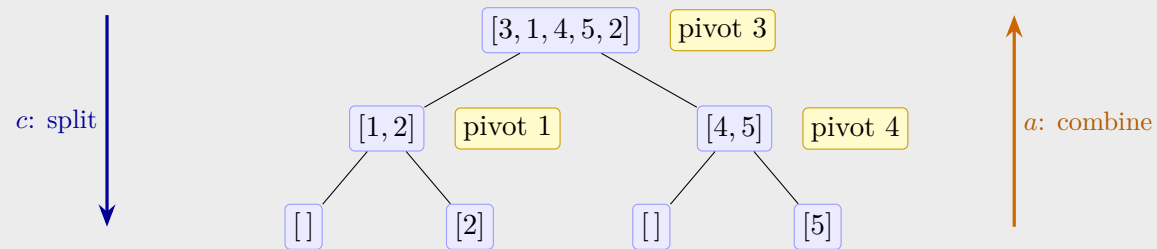


## The Call Tree of Quicksort

slide 20 / 83    step 1 of 2

- **Split goes down, combine comes up**
- ▷ the tree is only the call stack, never in memory

## The Call Tree of Quicksort



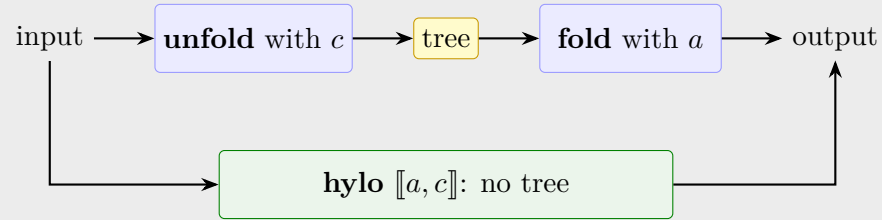
The tree is *virtual*: it is never built in memory.

## The Call Tree of Quicksort

slide 21 / 83 step 2 of 2

- Split goes down, combine comes up
- **▷ the tree is only the call stack, never in memory**

## Producers and Consumers



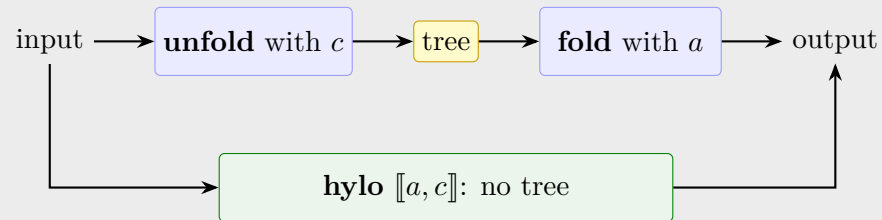
- Specification: *build*, then *consume*. Easy to read.

## Producers and Consumers

slide 22 / 83    step 1 of 2

- **Top:** build then consume; easy to read, wastes memory
- **Bottom:** hylo, no tree (deforestation, Wadler 1990)
- ▷ specs look like the top, programs like the bottom

## Producers and Consumers



- Specification: *build*, then *consume*. Easy to read.
- Program: do both at once. Fast.

## Producers and Consumers

slide 23 / 83    step 2 of 2

- Top: build then consume; easy to read, wastes memory
- Bottom: hylo, no tree (deforestation, Wadler 1990)
- ▷ **specs look like the top, programs like the bottom**

## Two Laws

Fusion

$$\boxed{\llbracket a, c \rrbracket} \rightarrow h = \boxed{\llbracket b, c \rrbracket}$$

if  $h$  distributes over  $a$

Deforestation

$$\boxed{\llbracket \text{build}, c \rrbracket} \rightarrow \boxed{\llbracket a, \text{take\_apart} \rrbracket} = \boxed{\llbracket a, c \rrbracket}$$

- Proved **once**, for every split, combine, data type.

## Two Laws

slide 24 / 83    step 1 of 2

- **Fusion:** push post-processing into combine, if it distributes
- **Deforestation:** build then take apart cancels out
- **proved once, for every data type**
- **▷** an optimisation is a chain of these

## Two Laws

**Fusion**  $\llbracket a, c \rrbracket \rightarrow h = \llbracket b, c \rrbracket$   
if  $h$  distributes over  $a$

**Deforestation**  $\llbracket \text{build}, c \rrbracket \rightarrow \llbracket a, \text{take\_apart} \rrbracket = \llbracket a, c \rrbracket$

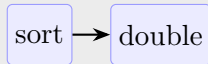
- Proved **once**, for every split, combine, data type.
- Each use is a *rewrite step*.

## Two Laws

slide 25 / 83    step 2 of 2

- Fusion: push post-processing into combine, if it distributes
- Deforestation: build then take apart cancels out
- proved once, for every data type
- **▷ an optimisation is a chain of these**

## Deriving the Example

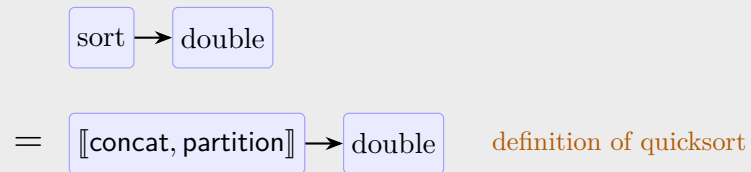


## Deriving the Example

slide 26 / 83    step 1 of 4

- **Quicksort = hylo with partition and concat**
- ▷ doubling distributes over concat: fusion
- ▷ result: the hand-written program
- ▷ each step a theorem, applied in Rocq

## Deriving the Example

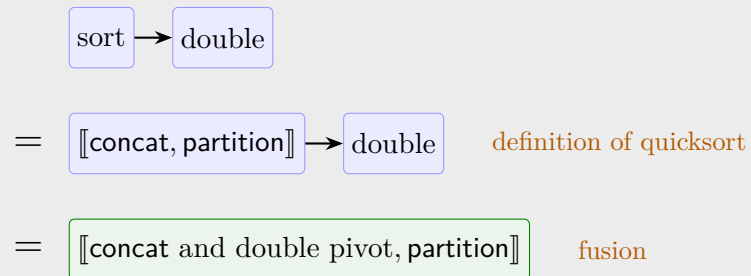


## Deriving the Example

slide 27 / 83    step 2 of 4

- Quicksort = hylo with partition and concat
- ▷ **doubling distributes over concat: fusion**
- ▷ result: the hand-written program
- ▷ each step a theorem, applied in Rocq

## Deriving the Example

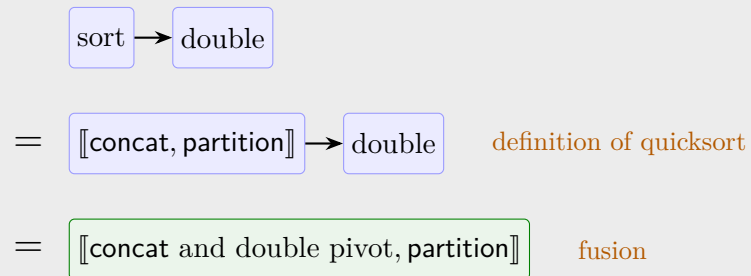


## Deriving the Example

slide 28 / 83    step 3 of 4

- Quicksort = hylo with partition and concat
- ▷ doubling distributes over concat: fusion
- ▷ **result: the hand-written program**
- ▷ each step a theorem, applied in Rocq

## Deriving the Example



Each = is a **theorem** applied by a proof assistant.

## Deriving the Example

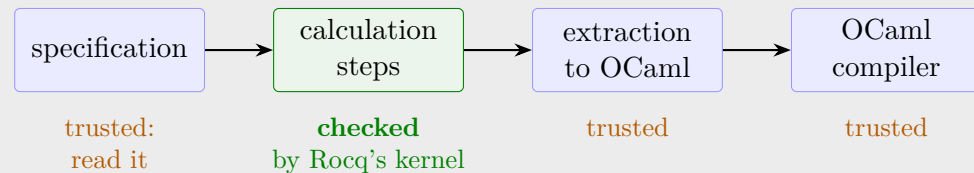
slide 29 / 83    step 4 of 4

- Quicksort = hylo with partition and concat
- ▷ doubling distributes over concat: fusion
- ▷ result: the hand-written program
- ▷ **each step a theorem, applied in Rocq**

# Machine-Checked

## Hylomorphisms in Rocq

## What You Have to Trust



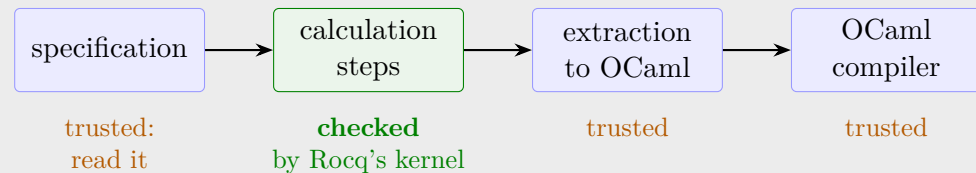
- Who picked the steps does not matter: a person, a script, a model.

## What You Have to Trust

slide 31 / 83    step 1 of 2

- **Green box:** checked by Rocq's small kernel
- **Trusted:** spec, kernel, extraction, OCaml compiler
- **steps can come from anyone:** person, script, model
- ▷ wrong step = rejected

## What You Have to Trust



- Who picked the steps does not matter: a person, a script, a model.
- A wrong step is **rejected**, not miscompiled.

## What You Have to Trust

slide 32 / 83    step 2 of 2

- Green box: checked by Rocq's small kernel
- Trusted: spec, kernel, extraction, OCaml compiler
- steps can come from anyone: person, script, model
- ▷ **wrong step = rejected**

## Three Obstacles

### Termination

every function  
must provably stop

### No extra axioms

comparing func-  
tions needs one

### Clean code

generic proofs ex-  
tract to unsafe casts

## Three Obstacles

slide 33 / 83    step 1 of 2

- **Termination:** Rocq only accepts recursion that provably stops
- **Axioms:** function equality needs one; we avoid it
- **Clean code:** textbook encoding extracts badly (next slide)
- ▷ Haskell has the laws, no termination

## Three Obstacles

### Termination

every function  
must provably stop

### No extra axioms

comparing func-  
tions needs one

### Clean code

generic proofs ex-  
tract to unsafe casts

Haskell has the laws but not termination; Rocq has termination but rejects the textbook definitions.

## Three Obstacles

slide 34 / 83    step 2 of 2

- Termination: Rocq only accepts recursion that provably stops
- Axioms: function equality needs one; we avoid it
- Clean code: textbook encoding extracts badly (next slide)
- ▶ **Haskell has the laws, no termination**

## What the Textbook Encoding Extracts To

```
type sh = __ (* = Obj.t *)
type pos = __
type 'x ext = { shape : sh; cont : (pos -> 'x) }

let merge x =
  let { shape = shape0; cont = k } = x in
  (match Obj.magic shape0 with
   | Some h -> app (Obj.magic k True) (Cons (h, (Obj.magic k False)))
   | None -> Nil)
```

(simplified)

## What the Textbook Encoding Extracts To

slide 35 / 83    step 1 of 2

- Real extraction output, trimmed
- ▷ point at Obj.magic: unchecked casts everywhere

## What the Textbook Encoding Extracts To

```
type sh = __ (* = Obj.t *)
type pos = __
type 'x ext = { shape : sh; cont : (pos -> 'x) }

let merge x =
  let { shape = shape0; cont = k } = x in
  (match Obj.magic shape0 with
   | Some h -> app (Obj.magic k True) (Cons (h, (Obj.magic k False)))
   | None -> Nil)
```

(simplified)

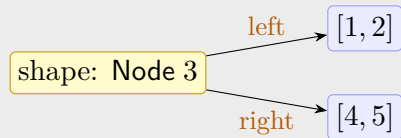
Obj.magic: an **unchecked cast**. Unsafe as soon as this code meets other code.

## What the Textbook Encoding Extracts To

slide 36 / 83    step 2 of 2

- Real extraction output, trimmed
- **▷ point at Obj.magic: unchecked casts everywhere**

## Our Fix: Shapes and Positions



shape: Leaf    no positions

| valid? | left | right |
|--------|------|-------|
| Node h | yes  | yes   |
| Leaf   | no   | no    |

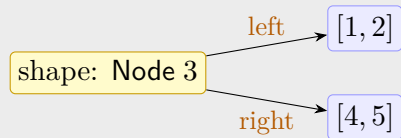
- One type of positions + a **yes/no check**: plain OCaml types.

## Our Fix: Shapes and Positions

slide 37 / 83    step 1 of 2

- **Data = shape + positions (containers)**
- **Point at the table: one position type, yes/no validity**
- **plain OCaml types**
- ▷ proofs erased at extraction: no casts

## Our Fix: Shapes and Positions



shape: Leaf    no positions

| valid? | left | right |
|--------|------|-------|
| Node h | yes  | yes   |
| Leaf   | no   | no    |

- One type of positions + a **yes/no check**: plain OCaml types.
- Proofs about the check are **erased** from the code.

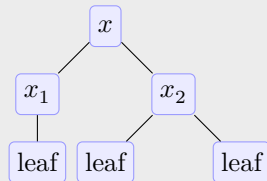
## Our Fix: Shapes and Positions

slide 38 / 83    step 2 of 2

- Data = shape + positions (containers)
- Point at the table: one position type, yes/no validity
- plain OCaml types
- ▷ **proofs erased at extraction: no casts**

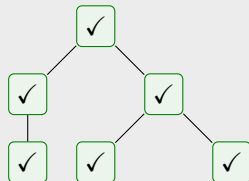
## Termination: Recursion on a Proof

call tree



$\Leftrightarrow$

termination proof



- Proved **once per split**, e.g. “pieces are shorter”.

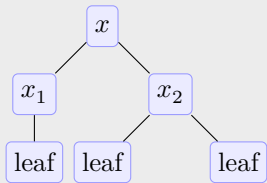
## Termination: Recursion on a Proof

slide 39 / 83    step 1 of 2

- **Proof tree has the same shape as the call tree; recurse on it**
- **one proof per split, e.g. pieces are shorter**
- $\triangleright$  proof erased: code is just the recursion

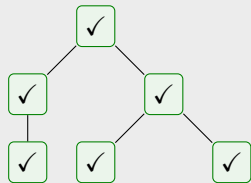
## Termination: Recursion on a Proof

call tree



$\leftrightarrow$

termination proof



- Proved **once per split**, e.g. “pieces are shorter”.
- The proof is scaffolding: **erased** at extraction.

## Termination: Recursion on a Proof

slide 40 / 83    step 2 of 2

- Proof tree has the same shape as the call tree; recurse on it
- one proof per split, e.g. pieces are shorter
- $\triangleright$  **proof erased: code is just the recursion**

## No Extra Axioms

```
Print Assumptions hylo_fusion_l.      Closed under the global context
Print Assumptions mss_calc.           Closed under the global context
Print Assumptions qsort_times_two_calc. Axioms: PrimInt63.int, ltb, mul
```

(simplified)

- Equality = *same results on equal inputs* (setoids).

## No Extra Axioms

slide 41 / 83    step 1 of 2

- Point at Closed under the global context
- Only Rocq's machine integers otherwise
- **equality = same results on equal inputs**
- ▷ laws usable as rewrite rules

## No Extra Axioms

```
Print Assumptions hylo_fusion_l.      Closed under the global context
Print Assumptions mss_calc.           Closed under the global context
Print Assumptions qsort_times_two_calc. Axioms: PrimInt63.int, ltb, mul
```

(simplified)

- Equality = *same results on equal inputs* (setoids).
- Laws become **rewrite** steps.

## No Extra Axioms

slide 42 / 83    step 2 of 2

- Point at Closed under the global context
- Only Rocq's machine integers otherwise
- equality = same results on equal inputs
- **▷ laws usable as rewrite rules**

# In Practice

From Calculation to Code

## A Calculation in Rocq

```
Derive qsort_times_two
  in (Lmap times_two \o cata merge \o rana tsplit =e qsort_times_two)
  as qsort_times_two_calc.
Proof.
  rewrite map_merge_fusion, hylo_ana, deforest by apply l_out_in.
  finished.
Qed.
```

## A Calculation in Rocq

slide 44 / 83    step 1 of 3

- **Derive:** the program starts as an unknown
- ▷ the spec: quicksort, then double
- ▷ one line of laws; Rocq fills in the program

## A Calculation in Rocq

```
Derive qsort_times_two
  in (Lmap times_two \o cata merge \o rana tsplit =e qsort_times_two)
  as qsort_times_two_calc.
Proof.
  rewrite map_merge_fusion, hylo_ana, deforest by apply l_out_in.
  finished.
Qed.
```



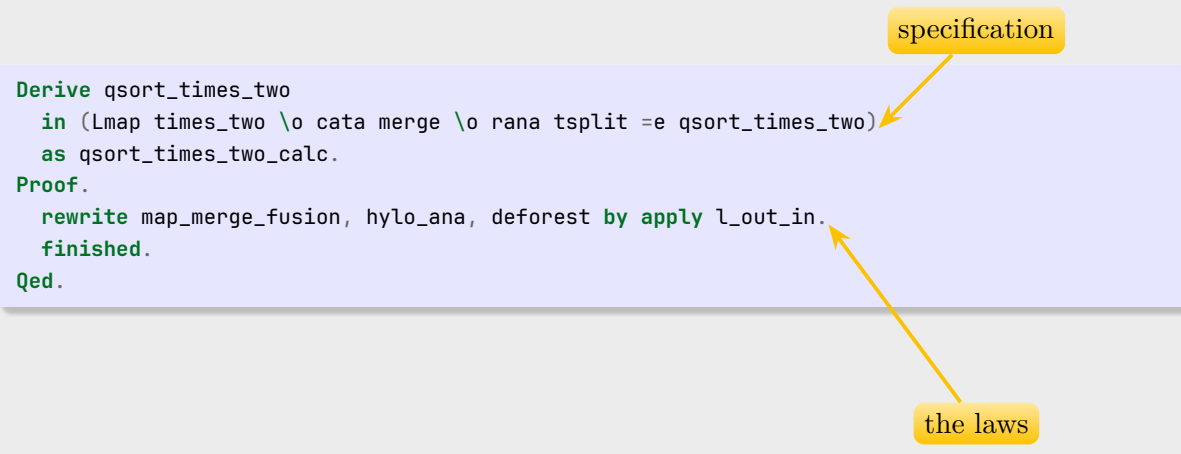
## A Calculation in Rocq

slide 45 / 83    step 2 of 3

- Derive: the program starts as an unknown
- ▷ **the spec: quicksort, then double**
- ▷ one line of laws; Rocq fills in the program

## A Calculation in Rocq

```
Derive qsort_times_two
  in (Lmap times_two \o cata merge \o rana tsplit =e qsort_times_two)
  as qsort_times_two_calc.
Proof.
  rewrite map_merge_fusion, hylo_ana, deforest by apply l_out_in.
  finished.
Qed.
```



## A Calculation in Rocq

slide 46 / 83    step 3 of 3

- Derive: the program starts as an unknown
- ▷ the spec: quicksort, then double
- ▷ **one line of laws; Rocq fills in the program**

## What Comes Out

```
let rec qsort_times_two_code = function
| [] -> []
| h :: t ->
  let (l, r) = partition (fun x0 -> ltb x0 h) t in
  let x0 = fun e ->
    qsort_times_two_code (match e with Lbranch -> l | Rbranch -> r) in
  app (x0 Lbranch) ((mul (Uint63.of_int (2)) h) :: (x0 Rbranch))
```

(simplified)

## What Comes Out

slide 47 / 83    step 1 of 2

- **Extracted: one pass, doubles each pivot**
- ▶ compare: same as the hand-written one, now with a proof

## What Comes Out

```
let rec qsort_times_two_code = function
| [] -> []
| h :: t ->
  let (l, r) = partition (fun x0 -> ltb x0 h) t in
  let x0 = fun e ->
    qsort_times_two_code (match e with Lbranch -> l | Rbranch -> r) in
  app (x0 Lbranch) ((mul (Uint63.of_int (2)) h) :: (x0 Rbranch))
```

(simplified)

Hand-written (start of the talk):

```
let rec sort_times_two = function
| [] -> []
| h :: t ->
  let (l, r) = partition (fun x -> x < h) t in
  sort_times_two l @ (h * 2) :: sort_times_two r
```

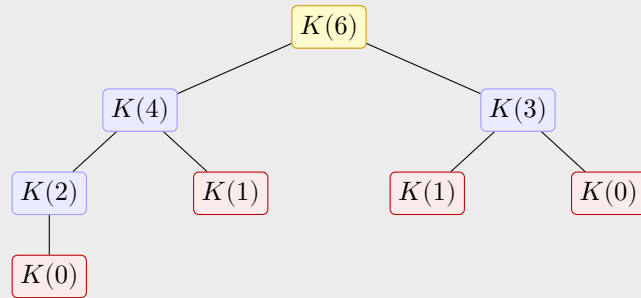
## What Comes Out

slide 48 / 83    step 2 of 2

- Extracted: one pass, doubles each pivot
-  **compare: same as the hand-written one, now with a proof**

## Dynamic Programming: Knapsack

$$K(0) = 0 \quad K(c) = \max \left( \{0\} \cup \{ K(c - w_i) + v_i \mid 0 < w_i \leq c \} \right)$$



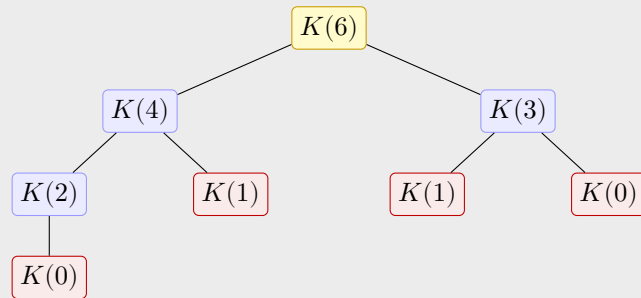
## Dynamic Programming: Knapsack

slide 49 / 83    step 1 of 2

- **Classic DP recurrence**
- **Point at red nodes: the same capacities computed again and again**
- **▷ naive recursion is a hylo, but exponential**

## Dynamic Programming: Knapsack

$$K(0) = 0 \quad K(c) = \max \left( \{0\} \cup \{ K(c - w_i) + v_i \mid 0 < w_i \leq c \} \right)$$



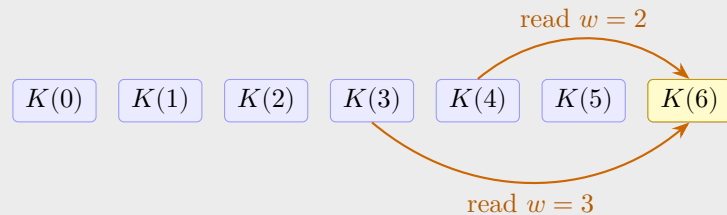
A hylomorphism, but it **recomputes**: exponential.

## Dynamic Programming: Knapsack

slide 50 / 83    step 2 of 2

- Classic DP recurrence
- Point at red nodes: the same capacities computed again and again
- ▷ **naive recursion is a hylo, but exponential**

## Remember, Don't Recompute



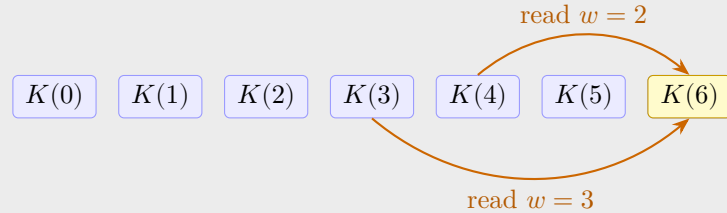
- **Dynamorphism:** a hylomorphism that keeps a *table* of earlier results.

## Remember, Don't Recompute

slide 51 / 83    step 1 of 3

- Keep a table of results for every capacity
- Point at arrows:  $K(6)$  reads  $K(4)$  and  $K(3)$  from the table
- dynamorphism (Hinze, Wu, Gibbons)
  - ▷ a law turns naive recursion into the table version
  - ▷ exponential to one entry per capacity

## Remember, Don't Recompute



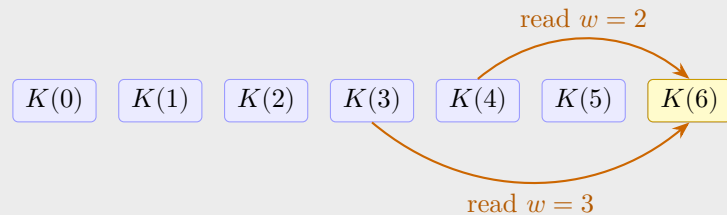
- **Dynamorphism:** a hylomorphism that keeps a *table* of earlier results.
- A law: naive recursion = table version, if each step only *reads* the table.

## Remember, Don't Recompute

slide 52 / 83    step 2 of 3

- Keep a table of results for every capacity
- Point at arrows:  $K(6)$  reads  $K(4)$  and  $K(3)$  from the table
- dynamorphism (Hinze, Wu, Gibbons)
- ▷ **a law turns naive recursion into the table version**
- ▷ exponential to one entry per capacity

## Remember, Don't Recompute



- **Dynamorphism:** a hylomorphism that keeps a *table* of earlier results.
- A law: naive recursion = table version, if each step only *reads* the table.
- Exponential  $\implies$  one entry per capacity.

## Remember, Don't Recompute

slide 53 / 83    step 3 of 3

- Keep a table of results for every capacity
- Point at arrows:  $K(6)$  reads  $K(4)$  and  $K(3)$  from the table
- dynamorphism (Hinze, Wu, Gibbons)
- $\triangleright$  a law turns naive recursion into the table version
- $\triangleright$  **exponential to one entry per capacity**

## | From $O(n)$ to $O(1)$ Lookups, by Rewriting

table:  
walk,  $O(n)$

|        |          |          |        |
|--------|----------|----------|--------|
| $K(c)$ | $K(c-1)$ | $\cdots$ | $K(0)$ |
|--------|----------|----------|--------|

array:  
index,  $O(1)$

|        |          |          |        |
|--------|----------|----------|--------|
| $K(0)$ | $\cdots$ | $K(c-1)$ | $K(c)$ |
|--------|----------|----------|--------|

## From $O(n)$ to $O(1)$ Lookups, by Rewriting

slide 54 / 83    step 1 of 2

- **Generic table, like a list: each lookup walks,  $O(n)$**
- **As an array: each lookup is one index,  $O(1)$**
- $\triangleright$  arrays are just another memo table; dyna does not care which
- Derive: naive exponential hylo to array dyna, three rewrites
- No hand-written index arithmetic

## | From $O(n)$ to $O(1)$ Lookups, by Rewriting

```
(* Arrays are one more memo table: an initial algebra of Memo *)
Instance ATab_initial : Initial (Memo NatF A) (ATab A).

(* dyna does not depend on the memo table *)
Theorem dyna_iso : dyna (T := T) a c =e dyna (T := U) (a \o fmap (iso U T)) c.

Derive knapsack_arr in (knapsack_naive wvs =e knapsack_arr) as calc.
Proof.
  rewrite <- (knapsack_correct (T := ATab int)).  (* naive hylo = dyna *)
  unfold knapsack; rewrite dyna_memo. finished.  (* share the table *)
Qed.
```

(simplified)

## From $O(n)$ to $O(1)$ Lookups, by Rewriting

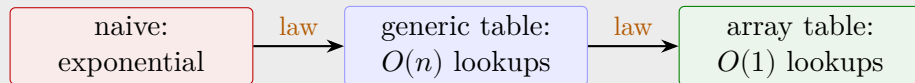
slide 55 / 83    step 2 of 2

- Generic table, like a list: each lookup walks,  $O(n)$
- As an array: each lookup is one index,  $O(1)$
- **▶ arrays are just another memo table; dyna does not care which**
- **Derive: naive exponential hylo to array dyna, three rewrites**
- **No hand-written index arithmetic**

## Knapsack: the Extracted Program

```
let rec memo t c = function
| [] -> []
| (w, v) :: wvs ->
  if 0 < w && w <= c then (get t (c - w) + v) :: memo t c wvs
  else memo t c wvs
let rec table wvs c =
  if c = 0 then make 1 0
  else let t = table wvs (c - 1) in
    push t (max_int 0 (memo t c wvs))
let knapsack wvs c = newest (table wvs c)
```

(simplified)



## Knapsack: the Extracted Program

slide 56 / 83

- Simplified from the extracted code
- table: one entry per capacity, built bottom-up
- Point at get: constant-time lookup
- Ladder: exponential, table, array; each step a law
- Linear in the capacity, for fixed items

## More Calculations

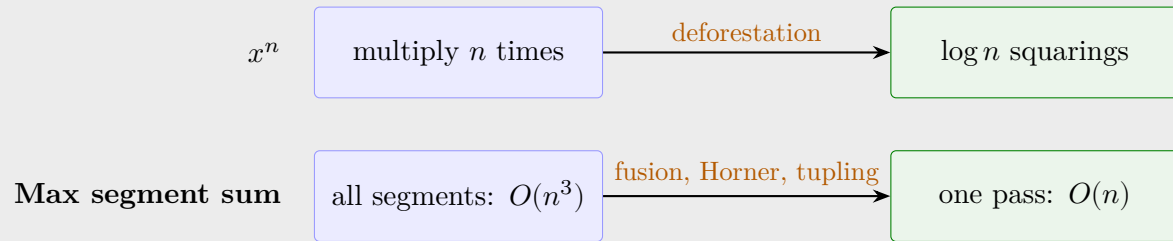


## More Calculations

slide 57 / 83    step 1 of 2

- **Fast power: deforest the list of binary digits**
- ▷ max segment sum: Bird's classic, cubic to linear

## More Calculations



## More Calculations

slide 58 / 83    step 2 of 2

- Fast power: deforest the list of binary digits
- ▷ **max segment sum: Bird's classic, cubic to linear**

## Maximum Segment Sum: Spec and Result

$\text{mss} = \text{maximum} \circ \text{map sum} \circ \text{segments}$

3 -5 2 4 -1    best segment:  $2 + 4 = 6$

## Maximum Segment Sum: Spec and Result

slide 59 / 83    step 1 of 2

- **Spec is the definition: all segments, sum, max**
- ▷ one pass, two accumulators; not obviously correct, but proved

## Maximum Segment Sum: Spec and Result

$\text{mss} = \text{maximum} \circ \text{map sum} \circ \text{segments}$

3 -5 2 4 -1    best segment:  $2 + 4 = 6$

```
let mss_code xs =  
  let rec f = function  
    | [] -> (Some 0, Some 0)  
    | h :: t ->  
      let (m, s) = f t in  
      let s' = match s with  
        | Some b -> Some (max 0 (h + b))  
        | None -> Some 0 in  
      (omax s' m, s')  
  in fst (f xs)
```

(simplified)

## Maximum Segment Sum: Spec and Result

slide 60 / 83    step 2 of 2

- Spec is the definition: all segments, sum, max
- ▶ **one pass, two accumulators; not obviously correct, but proved**

## Summary So Far

- A **library**: hylomorphisms and their laws in Rocq.
- Optimisations are **proofs by rewriting**.
- Termination proved **once per split**.
- **No extra axioms**; OCaml **without unsafe casts**.

## Summary So Far

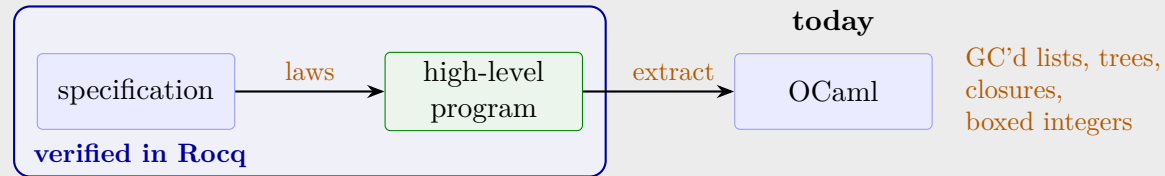
slide 61 / 83

- The library today: ITP 2025 plus later work
- Next: work in progress

# Work in Progress

Down to Low-Level Code

## Two Ways Out of Rocq

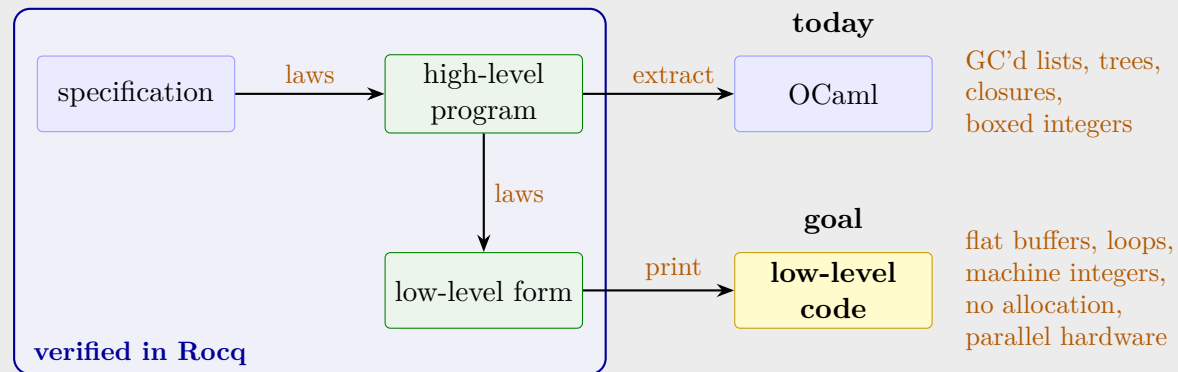


## Two Ways Out of Rocq

slide 63 / 83    step 1 of 3

- **Box = verified in Rocq**
- **Today: exit through OCaml (GC, closures, boxed integers)**
- ▷ Goal: more laws to a low-level form, printed directly
- ▷ layout and scheduling become calculation steps, not a compiler pass

## Two Ways Out of Rocq

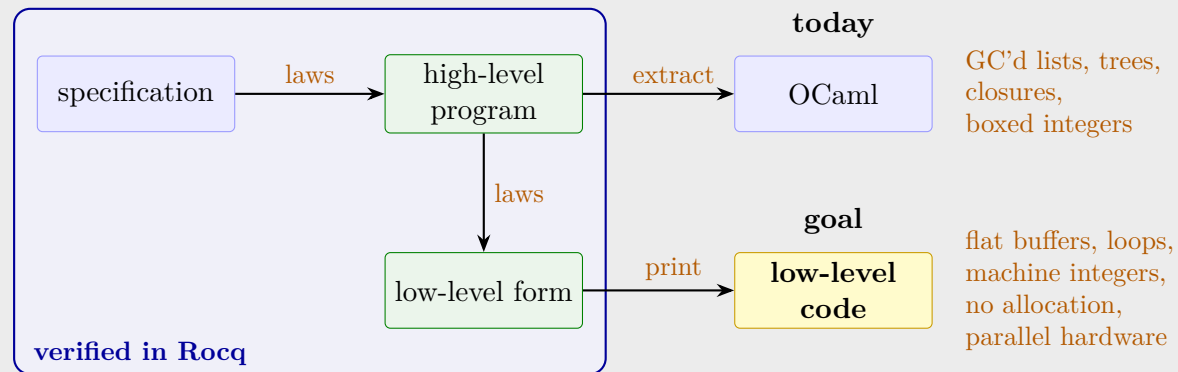


## Two Ways Out of Rocq

slide 64 / 83    step 2 of 3

- Box = verified in Rocq
- Today: exit through OCaml (GC, closures, boxed integers)
- ▷ **Goal: more laws to a low-level form, printed directly**
- ▷ layout and scheduling become calculation steps, not a compiler pass

## Two Ways Out of Rocq



Not “OCaml, then lower”: **more calculation** reaches the low-level form; extraction is a direct print.

## Two Ways Out of Rocq

slide 65 / 83    step 3 of 3

- Box = verified in Rocq
- Today: exit through OCaml (GC, closures, boxed integers)
- ▷ Goal: more laws to a low-level form, printed directly
- ▷ **layout and scheduling become calculation steps, not a compiler pass**

## | What We Are Working On

Low-level, **correct-by-construction** code,  
extracted from **high-level calculation steps**,  
possibly guided by a **cost model**.

- You choose *laws*, not pointers and indices.

## What We Are Working On

slide 66 / 83    step 1 of 3

- **Read the goal box**
- **you choose laws, not indices**
- ▷ every step proved
- ▷ cost model picks among equal programs

## What We Are Working On

**Low-level, correct-by-construction** code,  
extracted from **high-level calculation steps**,  
possibly guided by a **cost model**.

- You choose *laws*, not pointers and indices.
- Every step proved; the output is machine-level code.

## What We Are Working On

slide 67 / 83    step 2 of 3

- Read the goal box
- you choose laws, not indices
- ▷ **every step proved**
- ▷ cost model picks among equal programs

## What We Are Working On

**Low-level, correct-by-construction** code,  
extracted from **high-level calculation steps**,  
possibly guided by a **cost model**.

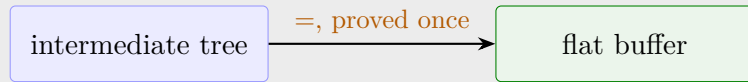
- You choose *laws*, not pointers and indices.
- Every step proved; the output is machine-level code.
- A cost model says *which* equal program to pick.

## What We Are Working On

slide 68 / 83    step 3 of 3

- Read the goal box
- you choose laws, not indices
- ▷ every step proved
- ▷ **cost model picks among equal programs**

## Ongoing: Laws That Change Representation

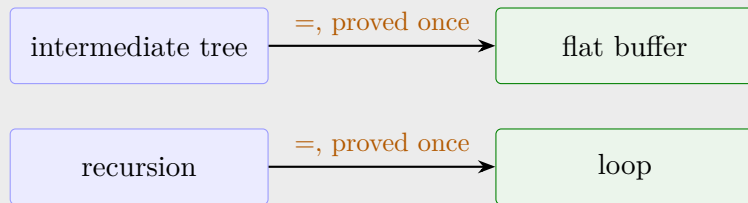


## Ongoing: Laws That Change Representation

slide 69 / 83    step 1 of 4

- **Today's laws only know trees**
- ▷ recursion to loops
- ▷ independent pieces run in parallel
- ▷ all proved once, used as rewrites

## Ongoing: Laws That Change Representation

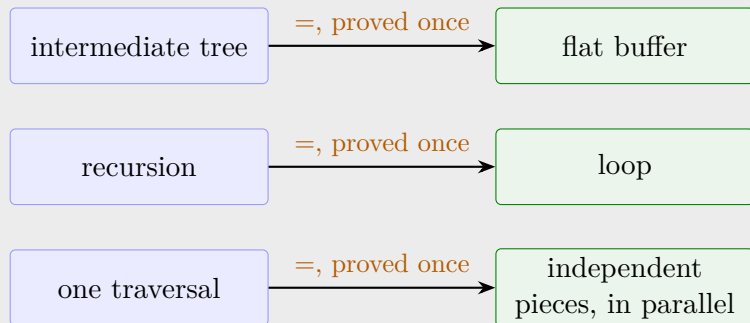


## Ongoing: Laws That Change Representation

slide 70 / 83    step 2 of 4

- Today's laws only know trees
- ▷ **recursion to loops**
- ▷ independent pieces run in parallel
- ▷ all proved once, used as rewrites

## Ongoing: Laws That Change Representation

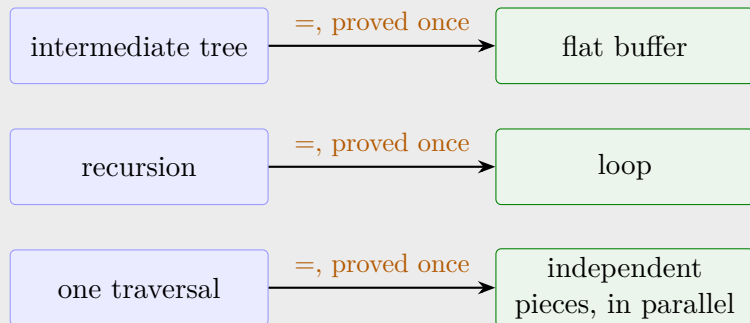


## Ongoing: Laws That Change Representation

slide 71 / 83    step 3 of 4

- Today's laws only know trees
- ▷ recursion to loops
- ▷ **independent pieces run in parallel**
- ▷ all proved once, used as rewrites

## Ongoing: Laws That Change Representation



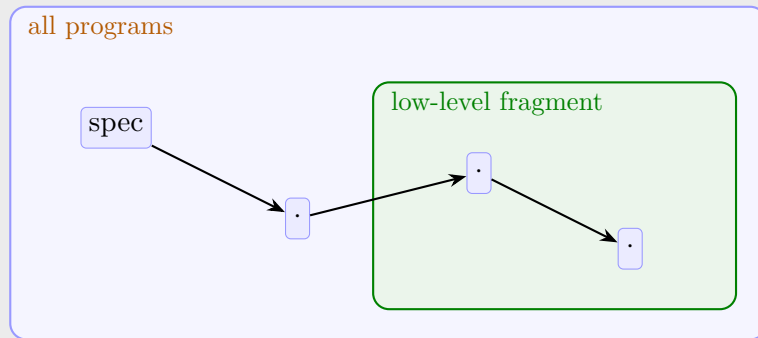
Today's laws change the *algorithm*; these change *where data lives* and *how work is scheduled*.

## Ongoing: Laws That Change Representation

slide 72 / 83    step 4 of 4

- Today's laws only know trees
- ▷ recursion to loops
- ▷ independent pieces run in parallel
- ▷ **all proved once, used as rewrites**

## Ongoing: A Target the Laws Never Leave



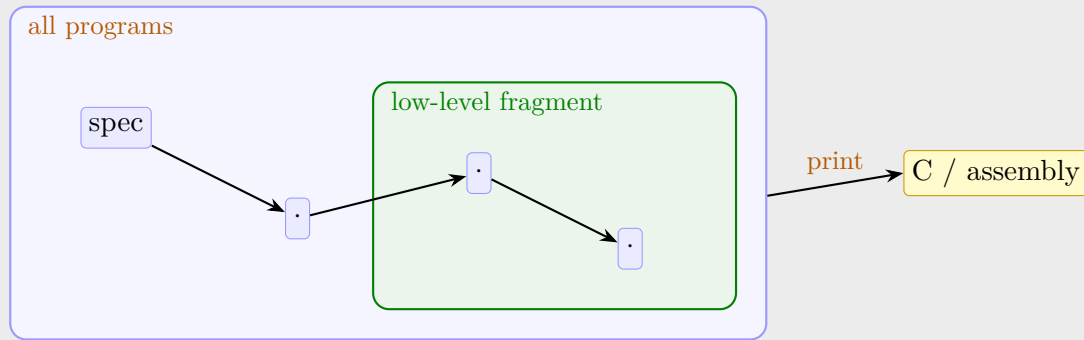
- Every law keeps programs **inside** the fragment.

## Ongoing: A Target the Laws Never Leave

slide 73 / 83    step 1 of 3

- **Low-level fragment: no closures, explicit memory, machine integers**
- **Laws keep programs inside it**
- ▷ printing is almost syntactic: small trusted base
- ▷ overflow and bounds as side conditions

## Ongoing: A Target the Laws Never Leave



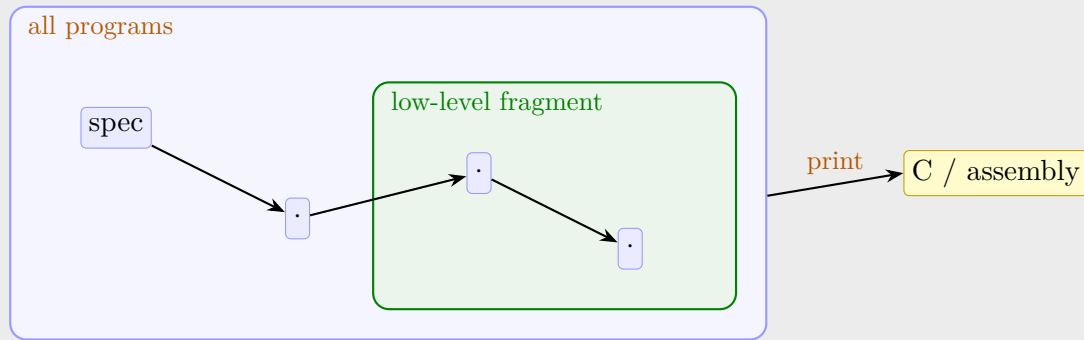
- Every law keeps programs **inside** the fragment.
- Only a **small printer** is trusted.

## Ongoing: A Target the Laws Never Leave

slide 74 / 83    step 2 of 3

- Low-level fragment: no closures, explicit memory, machine integers
- Laws keep programs inside it
- ▷ **printing is almost syntactic: small trusted base**
- ▷ overflow and bounds as side conditions

## Ongoing: A Target the Laws Never Leave



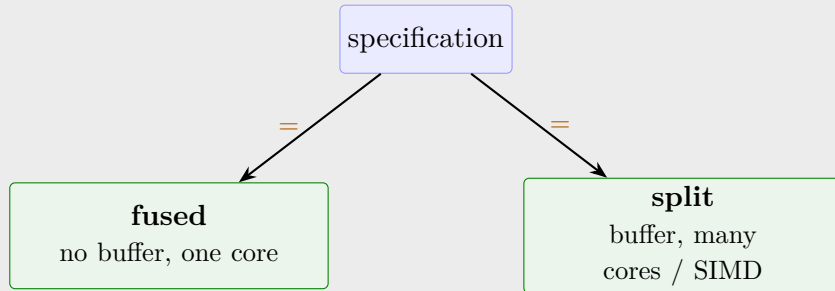
- Every law keeps programs **inside** the fragment.
- Only a **small printer** is trusted.
- Overflow and bounds: **side conditions** of the laws.

## Ongoing: A Target the Laws Never Leave

slide 75 / 83    step 3 of 3

- Low-level fragment: no closures, explicit memory, machine integers
- Laws keep programs inside it
- ▷ printing is almost syntactic: small trusted base
- ▷ **overflow and bounds as side conditions**

## Ongoing: A Cost Model

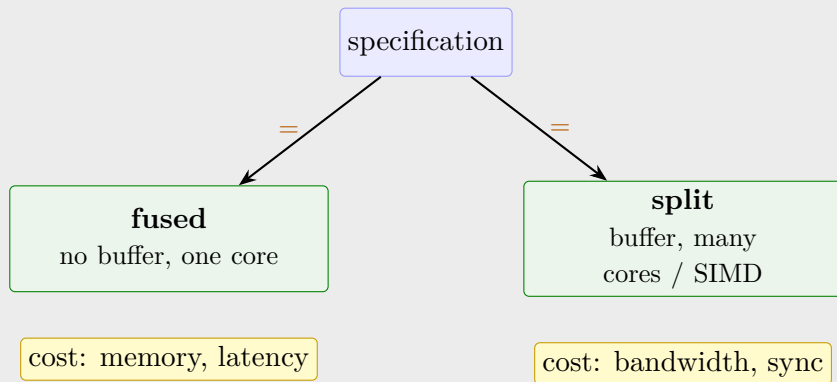


## Ongoing: A Cost Model

slide 76 / 83    step 1 of 3

- **Two equal programs from one spec**
- ▷ fused saves memory; split enables parallelism
- ▷ laws say equal, cost model says better

## Ongoing: A Cost Model

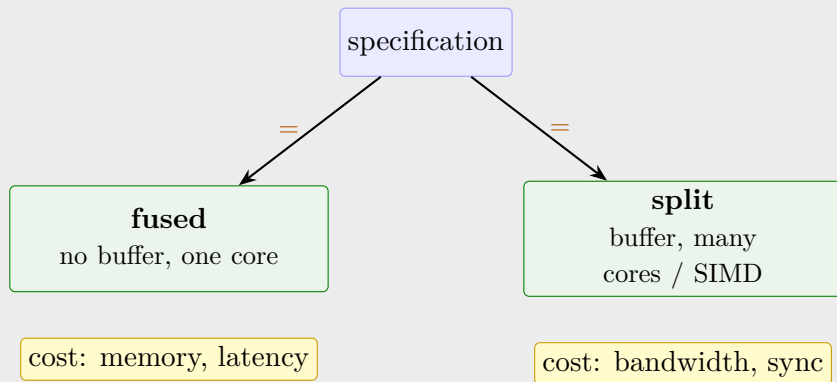


## Ongoing: A Cost Model

slide 77 / 83    step 2 of 3

- Two equal programs from one spec
- ▷ **fused saves memory; split enables parallelism**
- ▷ laws say equal, cost model says better

## Ongoing: A Cost Model



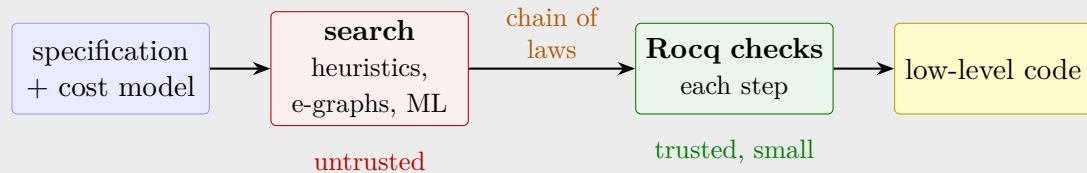
The laws say what is *equal*; a cost model says what is *better*.

## Ongoing: A Cost Model

slide 78 / 83    step 3 of 3

- Two equal programs from one spec
- ▷ fused saves memory; split enables parallelism
- ▷ **laws say equal, cost model says better**

## Untrusted Search, Trusted Checking



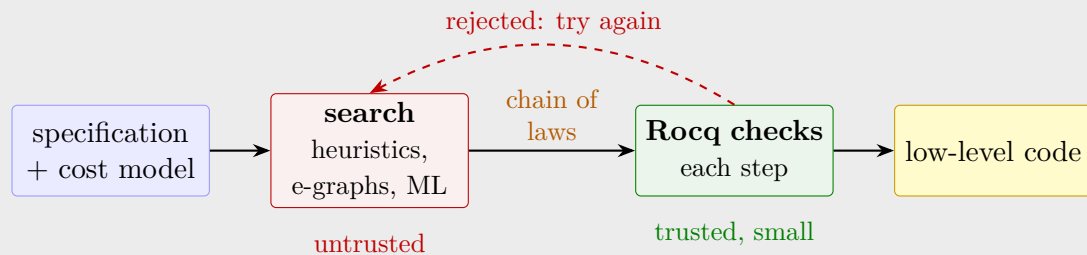
- The derivation is a **certificate**.

## Untrusted Search, Trusted Checking

slide 79 / 83    step 1 of 2

- **Search:** heuristics, e-graphs, or an ML model
- **Rocq checks the chain of laws**
  - ▷ wrong proposals rejected; output correct regardless
  - ML: like a model proposing proofs; security: small trusted base

## Untrusted Search, Trusted Checking



- The derivation is a **certificate**.
- Search can be as clever, or as wrong, as it likes.

## Untrusted Search, Trusted Checking

slide 80 / 83    step 2 of 2

- Search: heuristics, e-graphs, or an ML model
- Rocq checks the chain of laws
- ▷ **wrong proposals rejected; output correct regardless**
- **ML: like a model proposing proofs; security: small trusted base**

## Where We Are



### In Rocq, axiom-free:

- Splitting index ranges  $k$  ways
- Machine integers, no overflow (proved)
- Buffers, compared by their reads
- Tree reductions over ranges
- Writes to a buffer, driven by any split

### Not yet:

- Laws that add or remove a buffer
- Each index written exactly once
- The low-level target and printer
- The cost model and search

## Where We Are

slide 81 / 83

- Early stage; be honest
- Done: exact-cover splits, machine integers, buffers, tree reductions, writes
- Not yet: buffer laws, exactly-once, target, cost model

## Summary

- Program optimisation as **algebra**: rewrite with laws.
- Divide and conquer as **hylomorphisms**: laws proved once.
- In Rocq: **machine-checked**, no extra axioms, clean OCaml.
- Ongoing: **low-level, correct-by-construction code** from high-level steps, guided by a **cost model**.

## Summary

slide 82 / 83

- Recap the four bullets

# Thank you!

Questions?

`https://github.com/  
dcastrop/rocq-hylomorphisms`

Based on joint work with Marco  
Paviotti and Michael Vollmer.

slide 83 / 83

- Thank Marco and Michael; questions