

# Deriving, transforming, optimizing programs

MPRI 2.4

François Pottier



2022

## Let us be dreamers

An old dream:

- write high-level, abstract, **modular** code;
- let the compiler produce low-level, **efficient** code.

“Zero-cost abstraction”. (A C++/Rust slogan.)

“Abstraction without regret”. (A Scala/LMS slogan.)

**Pure** prog. languages should lend themselves well to this idea.

- **No mutable state.** Aliasing not a danger.  
Syntactically obvious where each variable receives its value.
- **Equational reasoning.**  
Programs denote values. Replace equals with equals.
- **Simple, rich language.**  
Many transformations easily expressed as rewriting rules.

Perhaps not quite true (do need **side effects** in some form), but let's see.

## 1 Equational reasoning

## 2 Inlining and simplification

## 3 Call-pattern specialization

## 4 Deforestation

A direct approach

Shortcut deforestation

Stream fusion

Stream fusion, with staging

## 5 Conclusion

## Equational reasoning

If two terms  $t_1$  and  $t_2$  are **observationally equivalent**,  
and if we have reason to believe that  $t_2$  is more efficient than  $t_1$ ,

- or that this rewriting step will enable further optimizations,

then we can **optimize** a program by replacing  $t_1$  with  $t_2$ .  
In fact, **observational refinement**  $t_2 \leq t_1$  suffices.

# Equality

In a **pure** & **total** language, such as Coq, a term is **equal** to its value.

Two terms that have the same value are **equal**.

Equal terms are **interchangeable** – Leibniz's Principle.

Life in an ideal (mathematical) world. See **DemoEqReasoning**.

## Observational refinement

In an impure world, a notion of **observational refinement** is needed.

Fix a notion of **observable behavior** of a closed term, e.g., reducing to a value, reducing to a stuck term, or diverging.

$t_2$  **refines**  $t_1$  (which we write  $t_2 \leq t_1$ ) iff,

for every (well-typed) context  $C$  such that  $C[t_1]$  and  $C[t_2]$  are closed,  
every behavior of  $C[t_2]$  is also a behavior of  $C[t_1]$ .

$t_1$  and  $t_2$  are **observationally equivalent** if  $t_1 \leq t_2$  and  $t_2 \leq t_1$  both hold.

## When is a rewriting step valid?

To prove that it is safe to transform a program fragment  $t_1$  into  $t_2$ , without exploiting any information about the context where  $t_1$  appears, one must prove  $t_2 \leq t_1$ .

Such a goal is often **difficult** to prove, and can in fact be **too strong**.

Some laws are valid only under an assumption about the context or about the initial state.

- e.g., a law might hold if a certain function has no side effect,
- or if the context cannot access a certain piece of private state.

Yang, **Relational Separation Logic**, 2007.

Frumin, Krebbers, Birkedal, **ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity**, 2021.

## When is a rewriting step valid?

Is full  $\beta$  a valid law?

$$(\lambda x. t_2) \ t_1 \simeq t_2[t_1/x]$$



## When is a rewriting step valid?

Is full  $\beta$  a valid law?

$$(\lambda x. t_2) t_1 \simeq t_2[t_1/x]$$

In a **pure** & **total** language, such as Coq,

## When is a rewriting step valid?

Is full  $\beta$  a valid law?

$$(\lambda x. t_2) t_1 \simeq t_2[t_1/x]$$

In a **pure** & **total** language, such as Coq, **yes**. Part of **definitional equality**.

Under **call-by-name**,

## When is a rewriting step valid?

Is full  $\beta$  a valid law?

$$(\lambda x. t_2) t_1 \simeq t_2[t_1/x]$$

In a **pure** & **total** language, such as Coq, **yes**. Part of **definitional equality**.

Under **call-by-name**, even in the presence of divergence, **yes**.

Under **call-by-value**,

## When is a rewriting step valid?

Is full  $\beta$  a valid law?

$$(\lambda x. t_2) t_1 \simeq t_2[t_1/x]$$

In a **pure & total** language, such as Coq, **yes**. Part of **definitional equality**.

Under **call-by-name**, even in the presence of divergence, **yes**.

Under **call-by-value**, in the presence of divergence or other effects, **no**.

## What about call-by-value? $\beta_v$

Under call-by-value, in the presence of side effects, full  $\beta$  is invalid.

As a simpler special case, one can use  $\beta_v$ , which is **valid**:

$$(\lambda x. t_2) v_1 \simeq t_2[v_1/x]$$

This follows from the theory of call-by-value parallel reduction.

See [LambdaCalculusStandardization/pcbv\\_adequacy](#).

## When is a rewriting step profitable?

When it is valid, is **full  $\beta$**  a profitable optimization?

$$(\lambda x. t_2) t_1 \longrightarrow t_2[t_1/x]$$

## When is a rewriting step profitable?

When it is valid, is **full  $\beta$**  a profitable optimization?

$$(\lambda x. t_2) t_1 \longrightarrow t_2[t_1/x]$$

Under **call-by-name**, it is safe for **time** and **space**,  
but can increase **code size**.

Under **call-by-need**, if  $x$  has multiple occurrences in  $t_2$ , or if  $x$  occurs under a  $\lambda$  within  $t_2$ , then the right-hand side risks **repeating** the computation of  $t_1$ , wasting **time** and **space**. This danger exists even if  $t_1$  is a value!

In short, this optimization step seems profitable when  $x$  is used “**at most once**” in  $t_2$ , for a suitable definition of this notion.

Turner, Wadler, Mossin, **Once upon a type**, 1995.

Peyton Jones, Santos, **A transformation-based optimiser for Haskell**, 1997.

## Summary so far

A proposed rewriting rule  $t_1 \longrightarrow t_2$  is **valid** if  $t_2 \leq t_1$  holds.

- This is influenced by the evaluation strategy, the presence or absence of side effects, and type hypotheses.

A proposed rewriting rule  $t_1 \longrightarrow t_2$  may or may not be **profitable**.

- This is influenced by many factors, including further optimizations and transformations.



- 1 Equational reasoning
- 2 Inlining and simplification
- 3 Call-pattern specialization
- 4 Deforestation
  - A direct approach
  - Shortcut deforestation
  - Stream fusion
  - Stream fusion, with staging
- 5 Conclusion

## What is inlining?

**Inlining** is the action of replacing a call to a known function with the suitably instantiated body of this function.

So, is **inlining** just another name for  $\beta_v$ ?

$$(\lambda x. t_2) v_1 \longrightarrow t_2[v_1/x]$$

## What is inlining?

No. Inlining can be more accurately described by several rewriting rules:

Looking up a definition: (IR1)

$$\text{let } x = v \text{ in } C[x] \longrightarrow \text{let } x = v \text{ in } C[v] \quad \text{if } x \notin bv(C)$$

Eliminating dead code: (IR2)

$$\text{let } x = v \text{ in } t \longrightarrow t \quad \text{if } x \notin fv(t)$$

Binding formals to actuals: (IR3)

$$(\lambda x. t_2) t_1 \longrightarrow \text{let } x = t_1 \text{ in } t_2$$

These rules are **valid** under every strategy and in the face of side effects.

Rule IR1 works for every value  $v$ , not just  $\lambda$ -abstractions.

Rules IR1 and IR2 work for “let rec”, too!

Rule IR1 **duplicates**  $v$  and can cause non-termination at compile-time (!) or an explosion in code size.

## Simplification rules

A few additional **simplification** rules are useful:

Eliminating an alias: (SR1)

$$\text{let } y = x \text{ in } t \longrightarrow t[x/y]$$

Hoisting a binding: (SR2)

$$E[\text{let } x = t_1 \text{ in } t_2] \longrightarrow \text{let } x = t_1 \text{ in } E[t_2]$$

These rules are **valid** under every strategy and in the face of side effects.

## Example

Consider this tiny example:

```
let succ x = x + 1
let even x = x mod 2 = 0
let test x = even (succ x)
```

This could be call-by-value (OCaml) or call-by-need (Haskell).

## Example, continued

```
let succ x = x + 1
let even x = x mod 2 = 0
let test x = even (succ x)
```

Inlining `succ` and `even` (IR1, applied twice) yields:

```
let succ x = x + 1
let even x = x mod 2 = 0
let test x = (fun x -> x mod 2 = 0) ((fun x -> x + 1) x)
```

## Example, continued

```
let succ x = x + 1
let even x = x mod 2 = 0
let test x = (fun x -> x mod 2 = 0) ((fun x -> x + 1) x)
```

Eliminating dead code (IR2, applied twice) yields:

```
let test x = (fun x -> x mod 2 = 0) ((fun x -> x + 1) x)
```

## Example, continued

```
let test x = (fun x -> x mod 2 = 0) ((fun x -> x + 1) x)
```

Binding (IR3) yields:

```
let test x = (fun x -> x mod 2 = 0) (let x = x in x + 1)
```



## Example, continued

```
let test x = (fun x -> x mod 2 = 0) (let x = x in x + 1)
```

Renaming (SR1) yields:

```
let test x =  
  (fun x -> x mod 2 = 0) (x + 1)
```

## Example, continued

```
let test x =  
  (fun x -> x mod 2 = 0) (x + 1)
```

Binding (IR3) yields:

```
let test x =  
  let x = x + 1 in  
  x mod 2 = 0
```

## Case of known constructor

IR3 is the simplification rule that actually **saves one step** of computation.

It is applicable when a **function** value is **eliminated**, that is, called.

What if a value of an **algebraic data type** is eliminated?

## Case of known constructor

IR3 is the simplification rule that actually **saves one step** of computation.

It is applicable when a **function** value is **eliminated**, that is, called.

What if a value of an **algebraic data type** is eliminated?

A new rule is needed:

Case of known constructor: (IR4)  
$$\text{case inj}_j v \text{ of } x_1.t_1 \parallel x_2.t_2 \longrightarrow \text{let } x_j = v \text{ in } t_j$$

## Example

Suppose Booleans are user-defined:

```
type bool = False | True
```

Now, consider this tiny example:

```
let not x = match x with False -> True | True -> False  
let test x = not (not x)
```

## Example, continued

```
let not x = match x with False -> True | True -> False
let test x = not (not x)
```

Inlining (IR1, applied twice) and dead code elimination (IR2) yield:

```
let test x =
  (fun x -> match x with False -> True | True -> False)
  ((fun x -> match x with False -> True | True -> False) x)
```

Binding (IR3) and renaming (SR1) yield:

```
let test x =
  (fun x -> match x with False -> True | True -> False)
  (match x with False -> True | True -> False)
```

## Example, continued

```
let test x =  
  (fun x -> match x with False -> True | True -> False)  
  (match x with False -> True | True -> False)
```

Binding (IR3) yields:

```
let test x =  
  let x = match x with False -> True | True -> False in  
  match x with False -> True | True -> False
```

## Example, continued

```
let test x =  
  let x = match x with False -> True | True -> False in  
  match x with False -> True | True -> False
```

Now, what? The rule  $\text{let}_v$  is **not** applicable here.



One can propose a new rule:

$$\begin{array}{c} \text{E of case: (SR3)} \\ E[\text{case } t \text{ of } x_1.t_1 \parallel x_2.t_2] \longrightarrow \text{case } t \text{ of } x_1.E[t_1] \parallel x_2.E[t_2] \end{array}$$

This rule is **valid** under every strategy.

It is known as a **commuting conversion**.

The rule that we need, “case-of-case”, is a special case of it!

**Exercise:** Write the rule “case-of-case”. (Answer in 4 slides.)

## Example, continued

```
let test x =  
  let x = match x with False -> True | True -> False in  
  match x with False -> True | True -> False
```

By E-of-case (SR3), we obtain:

```
let test x =  
  match x with  
  | False -> (  
    let x = True in  
    match x with False -> True | True -> False  
  )  
  | True -> (  
    let x = False in  
    match x with False -> True | True -> False  
  )
```

## Example, continued

```
let test x =  
  match x with  
  | False -> (  
    let x = True in  
    match x with False -> True | True -> False  
  )  
  | True -> (  
    let x = False in  
    match x with False -> True | True -> False  
  )
```

Inlining (IR1, IR2) and case-of-known-constructor (IR4) yield:

```
let test x =  
  match x with  
  | False -> False  
  | True -> True
```

## Example, continued

```
let test x =  
  match x with  
  | False -> False  
  | True  -> True
```

Yet another simplification rule, a form of  $\eta$ -reduction for sums, would yield:

```
let test x = x
```



## Case of case, improved

This rule **duplicates** the evaluation context:

$$\begin{array}{c} \text{E of case: (SR3)} \\ E[\text{case } t \text{ of } x_1.t_1 \parallel x_2.t_2] \quad \longrightarrow \quad \text{case } t \text{ of } x_1.E[t_1] \parallel x_2.E[t_2] \end{array}$$

This is potentially devastating!

E.g., suppose  $E$  is “case [ ] of  $y_1.u_1 \parallel y_2.u_2$ ”:

$$\begin{array}{c} \text{Case of case: (SR3c)} \\ \text{case (case } t \text{ of } x_1.t_1 \parallel x_2.t_2) \text{ of } y_1.u_1 \parallel y_2.u_2 \quad \longrightarrow \\ \text{case } t \text{ of } x_1.(\text{case } t_1 \text{ of } y_1.u_1 \parallel y_2.u_2) \\ \parallel x_2.(\text{case } t_2 \text{ of } y_1.u_1 \parallel y_2.u_2) \end{array}$$

The branches  $u_1$  and  $u_2$  are duplicated! What to do?

## Case of case, improved

A solution is to introduce **join points** to limit duplication.

Case of case, with join points: (SR3cj)

$$\text{case (case } t \text{ of } x_1.t_1 \parallel x_2.t_2) \text{ of } y_1.u_1 \parallel y_2.u_2 \longrightarrow$$

$$\text{let } k_1 = \lambda y_1.u_1 \text{ and } k_2 = \lambda y_2.u_2 \text{ in}$$

$$\text{case } t \text{ of } x_1.(\text{case } t_1 \text{ of } y_1.k_1 y_1 \parallel y_2.k_2 y_2)$$

$$\parallel x_2.(\text{case } t_2 \text{ of } y_1.k_1 y_1 \parallel y_2.k_2 y_2)$$

The names  $k_1$  and  $k_2$  can be thought of as **labels** to which one jumps.

We have intentionally **allowed** the outer case to be duplicated. The two copies scrutinize  $t_1$  and  $t_2$ , so further simplifications should be possible.

## Example

Suppose the function `bor` implements Boolean disjunction. Consider this:

```
match bor b1 b2 with
| False -> <foo>
| True  -> <bar>
```

Inlining yields:

```
match
  match b1 with False -> b2 | True -> True
with
| False -> <foo>
| True  -> <bar>
```

## Example, continued

```
match
  match b1 with False -> b2 | True -> True
with
| False -> <foo>
| True  -> <bar>
```

Applying rule SR3cj yields:

```
let foo () = <foo>
and bar () = <bar> in
match b1 with
| False -> (match b2 with False -> foo() | True -> bar())
| True  -> (match True with False -> foo() | True -> bar())
```



## Example, continued

```
let foo () = <foo>
and bar () = <bar> in
match b1 with
| False -> (match b2 with False -> foo() | True -> bar())
| True -> (match True with False -> foo() | True -> bar())
```

By case-of-known-constructor (IR4), we obtain:

```
let foo () = <foo>
and bar () = <bar> in
match b1 with
| False -> (match b2 with False -> foo() | True -> bar())
| True -> bar()
```

## Example, continued

```
let foo () = <foo>
and bar () = <bar> in
match b1 with
| False -> (match b2 with False -> foo() | True -> bar())
| True   -> bar()
```

Because there is only one jump to `foo`, it can be inlined:

```
let bar () = <bar> in
match b1 with
| False -> (match b2 with False -> <foo> | True -> bar())
| True   -> bar()
```

## Example, continued

bar is a “join point”, a local function that represents a code [label](#).

It is always called via a [tail call](#).

It should [not](#) require a closure allocation. `[@local]` indicates this.

```
let[@local] bar () = <bar> in
match b1 with
| False -> (match b2 with False -> <foo> | True -> bar())
| True   -> bar()
```

It must [not](#) be naïvely inlined: that would cause duplication again!

During further transformations, one should ensure that it remains a “join point” and is not inadvertently turned into a full-fledged first-class function.

Maurer, Ariola, Downen, Peyton Jones,  
[Compiling without continuations](#), 2017.

## Redundant case elimination

Equational  
reasoning

Inlining

Call-pattern  
specializat°

Deforestation

A direct  
approachShortcut  
deforestation

Stream fusion

Stream fusion,  
with staging

Conclusion

Can we optimize this code?

```
match xs with
| []      -> []
| y :: ys ->
    match xs with
    | []      -> <foo>
    | z :: zs -> <bar>
```

The rules shown so far can simplify this **only** if there is a binding of the form `let xs = <value>` higher up. This is case-of-known-constructor.

## Redundant case elimination

Can we optimize this code?

```
match xs with
| []      -> []
| y :: ys ->
    match xs with
    | []      -> <foo>
    | z :: zs -> <bar>
```

The rules shown so far can simplify this **only** if there is a binding of the form `let xs = <value>` higher up. This is case-of-known-constructor.

We could insert `let xs = y :: ys` at line 4, but that would be potentially pessimizing.

Better **keep track** of which **equations are known** at each program point, and improve **case-of-known-constructor** to exploit these equations.

See **Peyton Jones and Marlow**, §6.3.

## Inlining recursive functions

The rule IR1, as stated, does not allow inlining a function into itself. This could be relaxed.

Inlining a recursive function into itself amounts to **loop unrolling**.

Inlining a recursive function at its call site amounts to **loop peeling**.

## Summary

An old idea. Particularly **important** in very high-level languages.

It eliminates the function call overhead, and **enables other optimizations**.

The **danger** of inlining is an increase in **code size** and potential non-termination at compile time. This must be controlled via **heuristics** or via user annotations (**partial evaluation**; **staging**).

Aggressive inliners can be guided by **program analyses**.

Peyton Jones, Santos,  
**A transformation-based optimiser for Haskell**, 1997.

Peyton Jones, Marlow,  
**Secrets of the Glasgow Haskell Compiler inliner**, 2002.

Jagannathan and Wright, **Flow-directed inlining**, 1996.

## 1 Equational reasoning

## 2 Inlining and simplification

## 3 Call-pattern specialization

## 4 Deforestation

A direct approach

Shortcut deforestation

Stream fusion

Stream fusion, with staging

## 5 Conclusion



## Example

Here is a reasonably elegant way of obtaining the last element of a list:

```
let rec last xs =  
  match xs with  
  | [] -> assert false  
  | [x] -> x  
  | _ :: x :: xs -> last (x :: xs)
```

Unfortunately, it is inefficient...

## Example

Here is a reasonably elegant way of obtaining the last element of a list:

```
let rec last xs =  
  match xs with  
  | [] -> assert false  
  | [x] -> x  
  | _ :: x :: xs -> last (x :: xs)
```

Unfortunately, it is inefficient...

- The cell `x1 :: xs` is re-allocated; CSE can recognize and avoid this.
- **Two** list cells are inspected to find that the third branch must be taken.

Every cell is tested **twice**! We **forget** information through the recursive call.

How would you remedy this (by hand)?

## Example, hand-optimized

By hand, one might write this optimized code:

```
let rec last xs =  
  match xs with  
  | [] -> assert false  
  | x :: xs -> last_cons x xs  
  
and last_cons x xs =  
  match xs with  
  | [] -> x  
  | x :: xs -> last_cons x xs
```

`last_cons` is a loop with **two** registers `x` and `xs`.

Keeping track of `x` does the trick. Each list cell is examined **once**.

## Call-pattern specialization

Could a compiler do this **automatically**?

Inlining `last` into itself would amount to **loop unrolling** (i.e., doing two iterations at a time) but would **not** eliminate the problem entirely.

The problem lies in the call `last (x :: xs)`, where information is lost.

We must **specialize** `last` for this call pattern.

## Example, optimized

The first step is to create a specialized function, `last_cons`.

```
let rec last xs =  
  match xs with  
  | [] -> assert false  
  | [x] -> x  
  | _ :: x :: xs -> last (x :: xs)  
  
and last_cons x xs =  
  last (x :: xs)
```

The equation `last (x :: xs) = last_cons x xs` holds (obviously).

We **record** (remember) this equation for later use.

## Example, optimized

The second step is to inline `last` into `last_cons`.

```
let rec last xs =  
  match xs with  
  | [] -> assert false  
  | [x] -> x  
  | _ :: x :: xs -> last (x :: xs)
```

```
and last_cons x xs =  
  let xs = x :: xs in  
  match xs with  
  | [] -> assert false  
  | [x] -> x  
  | _ :: x :: xs -> last (x :: xs)
```

## Example, optimized

By inlining `xs` and exploiting case-of-known-constructor, we get:

```
let rec last xs =  
  match xs with  
  | [] -> assert false  
  | [x] -> x  
  | _ :: x :: xs -> last (x :: xs)
```

```
and last_cons x xs =  
  match xs with  
  | [] -> x  
  | x :: xs -> last (x :: xs)
```

What should be the last step?

## Example, optimized

The last step is to replace `last (x :: xs)` with `last_cons x xs`.

There are two occurrences, one of which lies within `last_cons` itself.

We get the code that we would have written, with one iteration unrolled:

```
let rec last xs =  
  match xs with  
  | [] -> assert false  
  | [x] -> x  
  | _ :: x :: xs -> last_cons x xs  
  
and last_cons x xs =  
  match xs with  
  | [] -> x  
  | x :: xs -> last_cons x xs
```

This [exploits an equation](#) that was recorded earlier.



Danger!

The correctness of **exploiting an equation within itself** is nonobvious.

Recall this situation:

```
let rec last xs =  
  match xs with  
  | [] -> assert false  
  | [x] -> x  
  | _ :: x :: xs -> last (x :: xs)  
  
and last_cons x xs =  
  last (x :: xs)
```

The equation `last (x :: xs) = last_cons x xs` holds (obviously).

There are **two** places where it can be used **right now**... What if we did so?

Danger!

We get a **non-terminating** version of the loop:

```
let rec last xs =  
  match xs with  
  | [] -> assert false  
  | [x] -> x  
  | _ :: x :: xs -> last_cons x xs  
  
and last_cons x xs =  
  last_cons x xs
```

This “obviously correct” transformation is actually **incorrect**.

We have in fact **rolled** the loop so it jumps to itself after 0 iterations!

Exploiting  $x = v$  within itself leads to  $x = x$ , which is nonsensical.

## Summary

Call-pattern specialization is also known as **constructor specialization**.

It is **simple**, but runs a risk of generating **uninteresting** specializations and a risk of **nontermination** at compile-time. **Heuristics** are needed.

Peyton Jones, **Call-pattern specialisation for Haskell programs**, 2007.

- 1 Equational reasoning
- 2 Inlining and simplification
- 3 Call-pattern specialization
- 4 Deforestation
  - A direct approach
  - Shortcut deforestation
  - Stream fusion
  - Stream fusion, with staging
- 5 Conclusion

# Deforestation

Programs expressed in a high-level style often build **intermediate data structures** (lists, trees, ...) which are immediately used and discarded.

They typically allow **communication** between a **producer** and a **consumer**.

**Deforestation** (Wadler, 1990) aims to get rid of them.

## 1 Equational reasoning

## 2 Inlining and simplification

## 3 Call-pattern specialization

## 4 Deforestation

A direct approach

Shortcut deforestation

Stream fusion

Stream fusion, with staging

## 5 Conclusion

## Example

The composition of `filter` and `map` allocates [an intermediate list](#).

As a direct attempt at deforestation, let us try and optimize it.

```
let bar p f xs =  
  List.filter p (List.map f xs)
```

Let us [specialize](#) for the call pattern `List.filter p (List.map f xs)...`

I am using an [expression](#) as a call pattern – this goes beyond GHC.

After creating a specialized copy  
and inlining `List.filter` and `List.map` into it, we get:

```
let filter_map p f xs =  
  match  
    match xs with  
    | [] -> []  
    | x :: xs -> f x :: List.map f xs  
  with  
  | [] -> []  
  | x :: xs ->  
    if p x then x :: List.filter p xs  
    else List.filter p xs  
  
let bar p f xs =  
  filter_map p f xs
```



## Example

Performing case-case conversion yields:

```
let filter_map p f xs =  
  match xs with  
  | [] -> []  
  | x :: xs ->  
    let x :: xs = f x :: List.map f xs in  
    if p x then x :: List.filter p xs  
    else List.filter p xs
```

## Example

Deciding to evaluate  $e1 :: e2$  from left to right, we get:

```
let filter_map p f xs =  
  match xs with  
  | [] -> []  
  | x :: xs ->  
    let x = f x in  
    let xs = List.map f xs in  
    if p x then x :: List.filter p xs  
    else List.filter p xs
```

Evaluation order is left undecided by OCaml.

## Example

We **wisely** choose to inline `xs`, as it is used only once (in each branch):

```
let filter_map p f xs =  
  match xs with  
  | [] -> []  
  | x :: xs ->  
    let x = f x in  
    if p x then x :: List.filter p (List.map f xs)  
    else List.filter p (List.map f xs)
```

This is **full  $\beta$** !

It is valid under call-by-need.

It is **invalid** under call-by-value (with side effects), unless `f` is pure.

- `f` must not access mutable data, raise an exception, or diverge.

The OCaml compiler won't do this!

## Example

We now recognize the call pattern `List.filter p (List.map f xs)`.

```
let rec filter_map p f xs =  
  match xs with  
  | [] -> []  
  | x :: xs ->  
    let x = f x in  
    if p x then x :: filter_map p f xs  
    else filter_map p f xs
```

We get the code that an OCaml programmer would write by hand.

No intermediate list! Successful **deforestation**.

## Summary

The equation `List.filter p (List.map f xs) = filter_map p f xs`

- holds under call-by-need;
- holds under call-by-value (with side effects) if `f` is **pure**.

Pure languages offer greater potential for **aggressive optimization**!

Bolingbroke, **Call-by-need supercompilation**, 2013.

Yet, **no mainstream compiler** performs such aggressive specialization.

Simpler, less ambitious techniques are presented next...

## 1 Equational reasoning

## 2 Inlining and simplification

## 3 Call-pattern specialization

## 4 Deforestation

A direct approach

Shortcut deforestation

Stream fusion

Stream fusion, with staging

## 5 Conclusion

## Idea 1: focus on lists

Focus on **lists**, a universal type for exchanging **sequences** of elements.

Some functions are list **producers**; some are list **consumers**.

Some, such as `filter` and `map`, are both. (Not a problem.)

Some, such as `zip` and `unzip`, have two inputs or two outputs.

Composing these functions yields producer-consumer **pipelines**.

## Idea 2: use a custom internal data format

Producers and consumers use lists as an **exchange** format.

They can work internally using a **different** data representation.

They can then be wrapped in **conversions** to and from lists.

When a producer and consumer are composed,

- two conversions, to and from lists, should **cancel out**,
- so there remains to optimize a composition at the **internal** data type.

This is an instance of the **worker/wrapper transformation**.

Gill and Hutton, **The worker/wrapper transformation**, 2009.



## Idea 3: avoid recursion

The internal data format should have a **nonrecursive** type, so that:

- Most producers and consumers are **not** recursive!
- At least one of the conversions, to and from lists, must be recursive.

Two approaches, based on two internal data formats, have been proposed:

- **shortcut deforestation**, based on **folds**;
- **stream fusion**, based on **streams**.

Gill, Launchbury, Peyton Jones,  
**A short cut to deforestation**, 1993.

Coutts, Leshchinskiy, Stewart, **Stream fusion:  
from lists to streams to nothing at all**, 2007.

## The internal data format

In shortcut deforestation, a sequence is internally represented as a **fold**.

A fold is a **function** that allows traversing the sequence.

```
type 'a fold =  
  { fold: 'b. ('a -> 'b -> 'b) -> 'b -> 'b }
```

It is a producer which **pushes** elements towards a consumer.

This is the standard **Church encoding** of lists.

## Converting a list to a fold

This is OCaml's `List.fold_right`, with the last two parameters swapped:

```
let rec foldr c n xs =  
  match xs with  
  | [] -> n  
  | x :: xs -> c x (foldr c n xs)
```

If `xs` is a list then `fun c n -> foldr c n xs` is the corresponding fold.

One can define:

```
let import (xs : 'a list) : 'a fold =  
  { fold = fun c n -> foldr c n xs }
```

## Converting a fold to a list

To convert a fold to a list, we apply it to “cons” and “nil”:

```
let build g =  
  g (fun x xs -> x :: xs) []
```

One can define:

```
let export ({ fold } : 'a fold) : 'a list =  
  build fold
```

# Isomorphism

There is an **isomorphism** between lists and (well-behaved) folds.

The following law holds:

- `export (import xs)` is observationally equivalent to `xs`.

Converting a list to a fold, then back to a list, is the identity.

This equality requires `f` to be **pure and terminating**, so is not a valid law:

- `import (export f) = f`.

Converting a fold to a list forces its side effects (if any) to take place **now**.

The law that's needed to compose two components is...

# Isomorphism

There is an **isomorphism** between lists and (well-behaved) folds.

The following law holds:

- `export (import xs)` is observationally equivalent to `xs`.

Converting a list to a fold, then back to a list, is the identity.

This equality requires `f` to be **pure and terminating**, so is not a valid law:

- `import (export f) = f`.

Converting a fold to a list forces its side effects (if any) to take place **now**.

The law that's needed to compose two components is... **the second one**.

Let's just **pretend** that it holds unconditionally.

**Exercise:** formalize `build/foldr` in Coq and establish the isomorphism.

# Isomorphism

In Gill et al.'s paper, the second law is known as “the foldr/build rule”:

```
foldr c n (build g) = g c n
```

## An example consumer-and-producer: map

In the list library, map can be written as follows:

```
let map f xs =  
  let { fold } = import xs in  
  let fold c n = fold (fun x xs -> c (f x) xs) n in  
  export { fold }
```

Convert from a list to a fold, map, convert back.



## An example consumer-and-producer: map

In Gill *et al.*'s paper, map is written as follows:

```
let map f xs =  
  build (fun c n ->  
    foldr (fun x xs -> c (f x) xs) n xs  
  )
```

This is obtained by inlining away `import` and `export`.

## An example consumer-and-producer: filter

Similarly, `filter` can be written as follows:

```
let filter p xs =  
  let { fold } = import xs in  
  let fold c n =  
    fold (fun x xs -> if p x then c x xs else xs) n in  
  export { fold }
```

## An example consumer-and-producer: filter

Equational  
reasoning

Inlining

Call-pattern  
specializat°

Deforestation

A direct  
approach

Shortcut  
deforestation

Stream fusion  
Stream fusion,  
with staging

Conclusion

Gill *et al.* write it as follows:

```
let filter p xs =  
  build (fun c n ->  
    foldr (fun x xs -> if p x then c x xs else xs) n xs  
  )
```

## Back to (filter; map)

What happens when we compose `filter` and `map`?

```
let bar p f xs =  
  filter p (map f xs)
```

## Back to (filter; map)

Inlining filter and map yields:

```
let bar p f xs =  
  build (fun c n ->  
    foldr  
      (fun x xs -> if p x then c x xs else xs)  
      n  
      (build (fun c n ->  
        foldr (fun x xs -> c (f x) xs) n xs  
        ))  
    )
```

We recognize foldr \_ \_ (build \_).

## Back to (filter; map)

Exploiting the equation  $\text{foldr } c \ n \ (\text{build } g) = g \ c \ n$  yields:

```
let bar p f xs =  
  build (fun c n ->  
    let c x xs = if p x then c x xs else xs in  
    foldr (fun x xs -> c (f x) xs) n xs  
  )
```

This is where we save an intermediate list.

## Back to (filter; map)

Inlining `c` yields:

```
let bar p f xs =  
  build (fun c n ->  
    foldr (fun x xs ->  
      let x = f x in  
      if p x then c x xs else xs  
    ) n xs  
  )
```

This is where `filter` and `map` come into contact and combine.

## Back to (filter; map)

We are essentially finished, but can work a little more.

Inlining build yields:

```
let bar p f xs =  
  foldr (fun x xs ->  
    let x = f x in  
    if p x then x :: xs else xs  
  ) [] xs
```



## Back to (filter; map)

Call-pattern specialization for `foldr` yields:

```
let rec filter_map p f xs =  
  match xs with  
  | [] -> []  
  | x :: xs ->  
    let xs = filter_map p f xs in  
    let x = f x in  
    if p x then x :: xs else xs  
  
let bar p f xs =  
  filter_map p f xs
```

Assuming the language is pure,  
or assuming `p` and `f` are pure, we can inline `xs`...

## Back to (filter; map)

Inlining `xs` yields:

```
let rec filter_map p f xs =  
  match xs with  
  | [] -> []  
  | x :: xs ->  
    let x = f x in  
    if p x then x :: filter_map p f xs  
    else filter_map p f xs
```

We again get the code that an OCaml programmer would write by hand.

- 1 Equational reasoning
- 2 Inlining and simplification
- 3 Call-pattern specialization
- 4 **Deforestation**
  - A direct approach
  - Shortcut deforestation
  - Stream fusion**
  - Stream fusion, with staging
- 5 Conclusion

## The internal data format

In stream fusion, a sequence is internally represented as a **stream**.

A stream is a **function** that allows querying the sequence.

```
type 'a stream =  
  | S:  
    (* If you have a pair of a producer function... *)  
    ('s -> ('a, 's) step)  
    (* ...and an initial state, *)  
    * 's ->  
    (* then you have a stream. *)  
  'a stream
```

It is a producer from which a consumer can **pull** elements.

A typical object-oriented idiom, analogous to Java iterators, but not inherently mutable.

This is an **existential type**, very much like the type of closures.

## The internal data format

Querying a stream produces a result of the following form:

```
type ('a, 's) step =  
  | Done (* finished *)  
  | Yield of 'a * 's (* an element and a new state *)  
  | Skip of 's (* just a new state: please ask again *)
```

The types `stream` and `step` are nonrecursive.

This, and the existence of `Skip`, allows most stream producers to be **nonrecursive** functions.

A consumer must ask, ask, ask until a non-`Skip` result is produced.

Coutts, Leshchinskiy, Stewart, **Stream fusion:  
from lists to streams to nothing at all**, 2007.

## Converting a list to a stream

This conversion function is nonrecursive:

```
let stream (xs : 'a list) : 'a stream =  
  let next xs =  
    match xs with  
    | [] -> Done  
    | x :: xs -> Yield (x, xs)  
  in  
  S (next, xs)
```

**Exercise:** Here, what is the type 's of states?

## Converting a list to a stream

This conversion function is nonrecursive:

```
let stream (xs : 'a list) : 'a stream =  
  let next xs =  
    match xs with  
    | [] -> Done  
    | x :: xs -> Yield (x, xs)  
  in  
  S (next, xs)
```

**Exercise:** Here, what is the type 's of states?

It is 'a list.

## Converting a list to a stream

The local function `next` is in fact closed, so one can also write:

```
let stream_next xs =  
  match xs with  
  | [] -> Done  
  | x :: xs -> Yield (x, xs)  
  
let stream (xs : 'a list) : 'a stream =  
  S (stream_next, xs)
```



## Converting a stream to a list

This is a recursive **consumer** function:

```
let unstream (S (next, s) : 'a stream) : 'a list =
  let rec unfold s =
    match next s with
    | Done          -> []
    | Yield (x, s) -> x :: unfold s
    | Skip s       -> unfold s
  in
  unfold s
```

# Isomorphism

There is an **isomorphism** between lists and (well-behaved) streams.

The following law holds:

- `unstream (stream xs)` is observationally equivalent to `xs`.

This equality requires `str` to be **pure and terminating**, so is not a valid law:

- `stream (unstream str)` is equivalent to `str`.

Again, we need the **second** law, known as “stream/unstream”.

Let’s **pretend** that it holds unconditionally.

## Examples of stream producers

How would you implement a singleton stream?

## Examples of stream producers

How would you implement a singleton stream?

```
let return (x : 'a) : 'a stream =  
  let next s =  
    if s then Yield (x, false) else Done  
  in  
  S (next, true)
```

The type of `s` is `bool`: either we have already yielded an element, or we have not.

Each stream producer **freely chooses** its type of internal states.

**Exercise:** Write interval of type `int -> int -> int stream`.

**Exercise:** Write append of type `'a stream -> 'a stream -> 'a stream`.

## An example consumer-and-producer

Here is `map` on streams, known as `S.map` in the following:

```
let map (f : 'a -> 'b) (S(next, s) : 'a stream) : 'b stream =  
  let next s =  
    match next s with  
    | Done          -> Done  
    | Yield (x, s) -> Yield (f x, s)  
    | Skip s        -> Skip s  
  in  
  S (next, s)
```

Again, `not` a recursive function!

## An example consumer-and-producer

Composing with conversions to and from streams yields `map` on lists:

```
let map (f : 'a -> 'b) (xs : 'a list) : 'b list =  
  unstream (S.map f (stream xs))
```

## An example consumer-and-producer

Here is filter on streams, known as `S.filter` in the following:

```
let filter (p : 'a -> bool) (S (next, s) : 'a stream) =  
  let next s =  
    match next s with  
    | Done          -> Done  
    | Yield (x, s) -> if p x then Yield (x, s) else Skip s  
    | Skip s        -> Skip s  
  in  
  S (next, s)
```

Again, **not** a recursive function!

## An example consumer-and-producer

Composing with conversions to and from streams yields `filter` on lists:

```
let filter (p : 'a -> bool) (xs : 'a list) : 'a list =  
  unstream (S.filter p (stream xs))
```



## Back to (filter; map)

What happens when we compose `filter` and `map`?

```
let bar p f xs =  
  L.filter p (L.map f xs)
```

## Back to (filter; map)

Inline filter and map:

```
let bar p f xs =  
  unstream (S.filter p (stream (  
    unstream (S.map f (stream xs))  
  )))
```

## Back to (filter; map)

Use the stream/unstream rule:

```
let bar p f xs =  
  unstream (S.filter p (S.map f (stream xs)))
```

`S.filter` and `S.map` come in contact.

Let's inline the hell out of this code!

## Back to (filter; map)

Inline stream:

```
let bar p f xs =  
  unstream (S.filter p (S.map f (S (stream_next, xs))))
```

## Back to (filter; map)

Inline `S.map`:

```
let bar p f xs =  
  let next s =  
    match stream_next s with  
    | Done          -> Done  
    | Yield (x, s) -> Yield (f x, s)  
    | Skip s        -> Skip s  
  in  
  unstream (S.filter p (S (next, xs)))
```

## Back to (filter; map)

Inline stream\_next:

```
let bar p f xs =  
  let next s =  
    match  
      match s with  
      | [] -> Done  
      | x :: s -> Yield (x, s)  
    with  
      | Done -> Done  
      | Yield (x, s) -> Yield (f x, s)  
      | Skip s -> Skip s  
  in  
  unstream (S.filter p (S (next, xs)))
```

## Back to (filter; map)

Perform case-of-case conversion, followed with case-of-constructor:

```
let bar p f xs =  
  let next s =  
    match s with  
    | [] -> Done  
    | x :: s -> Yield (f x, s)  
  in  
  unstream (S.filter p (S (next, xs)))
```

## Back to (filter; map)

Inline `S.filter`:

```
let bar p f xs =
  let next s =
    match s with
    | [] -> Done
    | x :: s -> Yield (f x, s)
  in
  let next s =
    match next s with
    | Done -> Done
    | Yield (x, s) -> if p x then Yield (x, s) else Skip s
    | Skip s -> Skip s
  in
  unstream (S (next, xs))
```



## Back to (filter; map)

Inline the first `next` function into the second one:

```
let bar p f xs =  
  let next s =  
    match  
      match s with  
      | [] -> Done  
      | x :: s -> Yield (f x, s)  
    with  
      | Done -> Done  
      | Yield (x, s) -> if p x then Yield (x, s) else Skip s  
      | Skip s -> Skip s  
  in  
  unstream (S (next, xs))
```

## Back to (filter; map)

Apply case-of-case and case-of-constructor again:

```
let bar p f xs =  
  let next s =  
    match s with  
    | [] -> Done  
    | x :: s ->  
      let y = f x in if p y then Yield (y, s) else Skip s  
  in  
  unstream (S (next, xs))
```

## Back to (filter; map)

Inline unstream:

```
let bar p f xs =  
  let next s =  
    match s with  
    | [] -> Done  
    | x :: s ->  
      let y = f x in if p y then Yield (y, s) else Skip s  
  in  
  let rec unfold s =  
    match next s with  
    | Done -> []  
    | Yield (x, s) -> x :: unfold s  
    | Skip s -> unfold s  
  in  
  unfold xs
```

## Back to (filter; map)

Inline next into unstream:

```
let bar p f xs =
  let rec unfold s =
    match
      match s with
      | [] -> Done
      | x :: s ->
        let y = f x in if p y then Yield (y, s) else Skip s
    with
    | Done -> []
    | Yield (x, s) -> x :: unfold s
    | Skip s -> unfold s
  in
  unfold xs
```

## Back to (filter; map)

Apply case-of-case again, then a couple rules, then case-of-constructor:

```
let bar p f xs =  
  let rec unfold s =  
    match s with  
    | [] -> []  
    | x :: s ->  
      let y = f x in  
      if p y then y :: unfold s else unfold s  
  in  
  unfold xs
```

**Exercise:** Clarify which rewriting rules are used here.

## Back to (filter; map)

(Optional.) Hoist unfold out. (This is  $\lambda$ -lifting.)

```
let rec unfold p f s =  
  match s with  
  | [] -> []  
  | x :: s ->  
    let y = f x in  
    if p y then y :: unfold p f s  
    else unfold p f s  
  
let bar p f xs =  
  unfold p f xs
```

We get the code that an OCaml programmer would write by hand.

No intermediate data structure! Successful **deforestation** again.

## What's the point?

Why is **stream fusion** preferable to **shortcut deforestation**?

Shortcut deforestation cannot express `foldl` in a nice way.

**Exercise:** Implement `foldl` on streams, then on lists.

**Exercise:** Find out how `foldl (+) 0 (append xs ys)` is optimized.  
You should reach a sequence of two loops – no memory allocation.

- 1 Equational reasoning
- 2 Inlining and simplification
- 3 Call-pattern specialization
- 4 Deforestation**
  - A direct approach
  - Shortcut deforestation
  - Stream fusion
  - Stream fusion, with staging**
- 5 Conclusion



In the previous approaches, the compiler's **heuristics** decide which reductions and simplifications take place at compile time.

*Relying on a general-purpose compiler for library optimization is slippery. [...] A compiler offers no guarantee that optimization will be successfully applied. [...] An innocuous change to a program [can] make it much slower.*

Kiselyov, Biboudis, Palladinis, Smaragdakis,  
**Stream fusion, to completeness**, 2017.

Instead, in **MetaOCaml** and **Scala/LMS**, programmers give explicit **staging** annotations to distinguish **pipeline-construction-time** and **pipeline-runtime**.

## MetaOCaml in a nutshell

MetaOCaml extends OCaml with

- a type `'a code`, or  $\langle \alpha \rangle$ , of code fragments;
- a `quote` construct for constructing a code fragment;
  - `.<3>.` has type `int code`;
- an `antiquote` construct that `splices` (plugs) one fragment into another.
  - `let square c = .< .~(c) * .~(c) >.`  
defines a function of type `int code -> int code`;
  - `.< fun x -> .~(square .<x>.) >.`  
has type `(int -> int) code`  
and evaluates to `.<fun x -> x * x>.`
- a `run` operation of type `'a code -> 'a`.

## MetaOCaml in a nutshell

MetaOCaml is **purely generative**:

a code fragment cannot be deconstructed (inspected).

So, one cannot implement rewrite rules as functions of type  $\langle \alpha \rangle \rightarrow \langle \alpha \rangle$ .

One can still write programs which **by design** generate good code.

- Perform possibly complex computations in stage 1,
- to produce simple code that runs in stage 2.

## Staging stream fusion

Kiselyov et al. stage **stream fusion** using MetaOCaml.

Kiselyov, Biboudis, Palladinos, and Smaragdakis.  
**Stream fusion, to completeness**, 2017.

See **StreamFusion** for an explanation of the first two steps.

**Rompf**, **Parreaux** and others write on metaprogramming in Scala.

## Thoughts on staging

Staging is an old idea: it goes back at least to Lisp in the 1970s.

Bawden, *Quasiquote in Lisp*, 1999.

Quasiquote in Lisp is not hygienic. MetaML corrects this.

Sheard, *Using MetaML: a staged programming language*, 1998.

MetaOCaml requires *staging annotations* and performs type inference.

Scala/LMS requires *type annotations* and infers staging information.

Rompf, *Lightweight Modular Staging and Embedded Compilers:  
Abstraction Without Regret for High-Level High-Performance  
Programming*, 2012.

- 1 Equational reasoning
- 2 Inlining and simplification
- 3 Call-pattern specialization
- 4 Deforestation
  - A direct approach
  - Shortcut deforestation
  - Stream fusion
  - Stream fusion, with staging
- 5 Conclusion

## Program derivation

Equational reasoning can be used not just by compilers, but also by [programmers](#), by hand.

Starting from a simple, inefficient program, [derive](#) efficient code via a series of rewriting steps.

See my [blog post](#) on a derivation of Knuth-Morris-Pratt, based on a paper by Ager, Danvy, and Rohde.

## A few things to remember

- **Equational reasoning** can be a powerful means of **transforming** or **deriving** programs.
- $\lambda$ -calculus-based (intermediate) languages allow expressing a wide range of program **transformations** and **optimizations**.
- **Side effects** (non-termination, mutable state...) complicate matters.
- **Guiding** the transformations is difficult, requiring **heuristics** or **staging annotations**.