

Deriving, transforming, optimizing programs

MPRI 2.4

François Pottier



2017

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.)

(Pure) functional 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
- 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 .

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 equivalence

Fix some notion of “**success**”, e.g. t succeeds iff t computes 42.

- Note that this notion depends on the evaluation **strategy**.

With respect to this notion of success, or “observation”,

t_1 and t_2 are **observationally equivalent** ($t_1 \simeq t_2$) iff,

for every (well-typed) context C ,

$C[t_1]$ succeeds if and only if $C[t_2]$ succeeds.

When is a rewriting step valid?

Is full β a valid law?

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

When is a rewriting step valid?

Is **full β** 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 non-termination, **yes**.

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

full β is invalid under call-by-value

Repeat after me:

full β is invalid under call-by-value

full β is invalid under call-by-value

full β is invalid under call-by-value

After 20+ years, I keep making this mistake from time to time!

$(\lambda x. t_2) t_1$ cannot be “simplified” to $t_2[t_1/x]$

let $x = t_1$ in t_2 cannot be “simplified” to $t_2[t_1/x]$

What about call-by-value? β_v

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

One must restrict it to the case where t_1 is **pure**.

$$(\lambda x. t_2) t_1 \longrightarrow t_2[t_1/x] \quad \text{provided } t_1 \text{ is pure}$$

Roughly, a closed term t is pure if there exists a value v such that t reduces to v , independently of the store.

Whether a non-closed term t is closed depends on purity hypotheses about its free variables. E.g., is “ $f\ x$ ” pure? Yes, **IF** f has no side effects.

As a simple special case, one can use β_v , which is **valid**:

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

This follows from the theory of parallel reduction.

See [LambdaCalculusStandardization/pcbv_adequacy](#).

When is a rewriting step profitable?

When it is valid, is full β 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 β** 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 λ 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 $t_1 \simeq t_2$ 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.

let-reduction

So far, I have discussed full β versus β_v .

If the language has a primitive construct,
then an analogous discussion applies to “full let” versus let_v .

$$\begin{array}{ll} \text{let } x = t_1 \text{ in } t_2 & \longrightarrow t_2[t_1/x] \\ \text{let } x = v_1 \text{ in } t_2 & \longrightarrow t_2[v_1/x] \end{array}$$

η -reduction and η -expansion

Is this optimization valid?

$$\lambda x.t \ x \simeq t \quad \text{provided } x \notin \text{fv}(t)$$

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

Under **call-by-name**, in the presence of non-termination, I think it is...

Under **call-by-value**, in the presence of side effects, it definitely **isn't**.

When it is valid, is it profitable? **Possibly**. E.g., after a naïve CPS transformation, η -reduction turns $\lambda x.k \ x$ into k , which amounts to **tail call optimization**.

Yet η -reduction **can** be costly and η -expansion can be profitable. Tricky!

- 1 Equational reasoning
- 2 Inlining and simplification
- 3 Call-pattern specialization
- 4 Deforestation
 - A direct approach
 - Shortcut deforestation
 - Stream fusion
- 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 β_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 λ -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)

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

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
```

Example, continued

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

Optionally, one more application of IR1 & IR2 could yield:

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

This would not improve the machine code that we get in the end, though.

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)**
$$\text{case inj}_i \ v \text{ of } x_1.t_1 \parallel x_2.t_2 \longrightarrow \text{let } x_i = v \text{ in } t_i$$

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 β_v is **not** applicable here.

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 β_v is **not** applicable here.

Under call-by-need, this let construct can be reduced:

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

We then seem to need a “case-of-case” simplification rule.

What happens under call-by-value, though?

E of case

Under call-by-value, one could argue that the right-hand side is pure and apply full β .

One can do better and directly apply a new rule:

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

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

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

Case-of-case is a special case of it!

Exercise (recommended): Write the rule “case-of-case”.

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, η -reduction for sums, yields:

```
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 is meant to represent a code [label](#).

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

The idea is, it should [not](#) require a closure allocation.

```
let 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

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
- 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:

```
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)
```

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

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

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
- 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
- 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.

Example

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 that $e1 :: e2$ is evaluated 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 β** !

It is valid under call-by-need. (Assuming no side effects but divergence.)

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

- `f` must not read or write mutable data, and must terminate.

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**!

- 1 Equational reasoning
- 2 Inlining and simplification
- 3 Call-pattern specialization
- 4 **Deforestation**
 - A direct approach
 - Shortcut deforestation**
 - Stream fusion
- 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 are 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.

Gill et al.'s paper does not explicitly use the above polymorphic type.
I follow them.

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.

We could 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) []
```

We could define:

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

Isomorphism

The idea is that we have an **isomorphism** between lists and (certain well-behaved) folds.

The following law holds:

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

The reverse law holds if `f` is **pure and terminating**:

- `import (export f)` is equivalent to `f`.

Naturally, the law that's needed when composing two components is...

Isomorphism

The idea is that we have an **isomorphism** between lists and (certain well-behaved) folds.

The following law holds:

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

The reverse law holds if `f` is **pure and terminating**:

- `import (export f)` is equivalent to `f`.

Naturally, the law that's needed when composing two components is...
...**the second one**.

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

Challenge: 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”:

$$\text{foldr } c \ n \ (\text{build } g) = g \ c \ n$$

An example consumer-and-producer

In the list library, `map` is written as follows:

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

The list `xs` is imported using `foldr`, yielding a fold.

A new fold is then constructed on top of it.

This new fold is converted back to a list using `build`.

An example consumer-and-producer

Similarly, `filter` is written 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**
- 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 in week 3.

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.

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

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 (certain) streams.

The following law holds:

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

The reverse law holds if `str` is **pure and terminating**:

- `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)  
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  
let bar p f xs =  
  let rec unfold s =  
    match  
      match s with  
      | [] -> Done
```

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 λ -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.

The way of the future?

Do not let the compiler's **heuristics** decide
which reductions and simplifications should take place at compile time.

Instead, give explicit **staging** annotations to distinguish
pipeline-construction-time computation and **pipeline-runtime** computation!

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.

- 1 Equational reasoning
- 2 Inlining and simplification
- 3 Call-pattern specialization
- 4 Deforestation
 - A direct approach
 - Shortcut deforestation
 - Stream fusion
- 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.

Supercompilation can do this, too!

Secher and Sørensen, **On Perfect Supercompilation**, 1999.

A few things to remember

- **Equational reasoning** can be a powerful means of **transforming** or **deriving** programs.
- λ -calculus-based (intermediate) languages allow expressing a wide range of program **transformations** and **optimizations**.
- **Side effects** (non-termination, mutable state...) complicate matters.