Towards
Certified Incremental Functional Programming

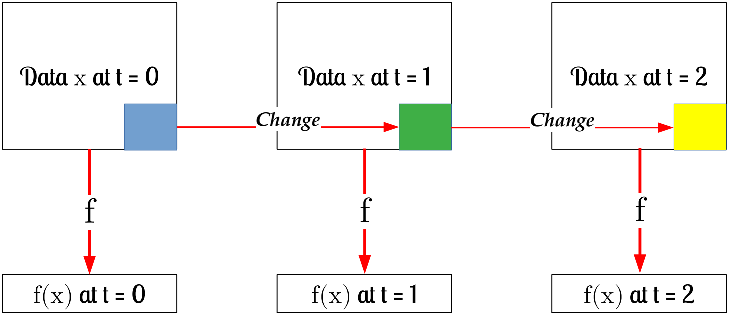
Yann Régis-Gianas
(IRIF, Univ. Paris, Inria $\pi.r^2$) – yrg@irif.fr

with Paolo Giarrusso (Univ. Delft), Philip Schutser (Univ. Marburg), Lourdes Gonzalez Huesca (Univ. Mexico), Lelio Brun (ENS), Olivier Martinot (Univ. Paris)

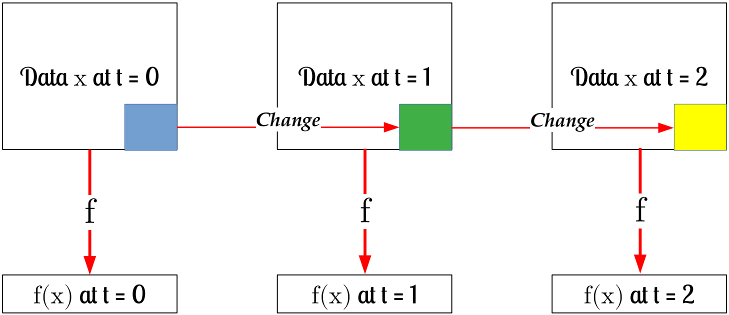
2019-11-14

Plan

Data constantly change

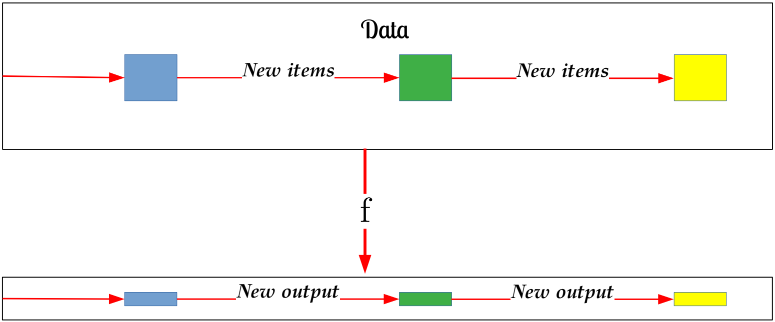


Data constantly change



- Now, take size(x) $= 2^{50}$ and size(modified part of x) $= 2^{10}$.
- Recomputation is not an option!

Stream-based processing



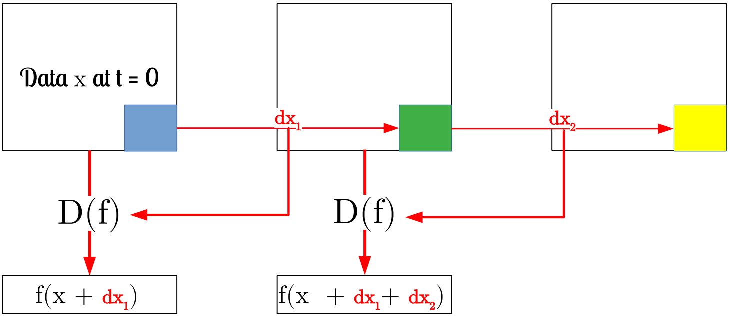
- f only reacts to new items by producing a new version of its output.
- We are back to a reasonable computational setting.

What about large structured data?

- Stream-based processing is relevant for computations:
  - that are dealing with **linearizable** data ;
  - whose output only depends on a bounded number of previous items.
- Examples: tweets, financial data, machine learning datasets, ...

How should we program systems that
perform **non local** computations
over **interdependent** and ever-changing **structured** values?
(e.g. commits in a large source code repository, complex simulations, ...)

Incremental programming with first-class changes



Incremental programming with first-class changes

If $\begin{cases} f : A \to B \\ \Delta A \text{ are changes over } A \text{ and } \Delta B \text{ are changes over } B \\ \oplus_A : A \to \Delta A \to A \text{ and } \oplus_B : A \to \Delta B \to B \end{cases}$

then use $D(f)$ such that:

$$f\ (x \oplus_A dx) = f\ x \oplus_B D(f)\ x\ dx$$

Incremental programming with first-class changes

If $\begin{cases} f : A \to B \\ \Delta A \text{ are changes over } A \text{ and } \Delta B \text{ are changes over } B \\ \oplus_A : A \to \Delta A \to A \text{ and } \oplus_B : A \to \Delta B \to B \end{cases}$

then use $D(f)$ such that:

$$f\ (x \oplus_A dx) = f\ x \oplus_B D(f)\ x\ dx$$

where the complexity of $D(f)$

- (should ideally) only depends on the size of dx, and
- (always) be better than the complexity of f.

A first-order example

Let us assume that **`rows`** is a bunch of grades coming from a CSV file [1]:

```
let rows = [ "Pédrot" , "Pierre-Marie", "17";
             "Knuth"   , "Donald"      , "12";
             "Torvald", "Linus"        , "10"  ]
```

[1] Please, imagine that there is 10^{40} rows in that file.

A first-order example

Let us assume that **rows** is a bunch of grades coming from a CSV file [1]:

```
let rows = [ "Pédrot"  , "Pierre-Marie", "17";
             "Knuth"   , "Donald"      , "12";
             "Torvald", "Linus"       , "10"  ]
```

The Ministry wants us to compute the mean of these grades. That's easy:

```
let myprog rows =
  let grades = List.map (fun ℓ -> List.nth ℓ 2 |> int_of_string) rows in
  let mean = List.fold_left ( + ) 0 grades / List.length grades in
  mean
```

We run the program, get some value **m**, and we are done!

[1] Please, imagine that there is 10^{40} rows in that file.

A first-order example

Let us assume that `rows` is a bunch of grades coming from a CSV file [1]:

```
let rows = [ "Pédrot" , "Pierre-Marie", "17";
             "Knuth"  , "Donald"      , "12";
             "Torvald", "Linus"       , "10"  ]
```

The Ministry wants us to compute the mean of these grades. That's easy:

```
let myprog rows =
  let grades = List.map (fun ℓ -> List.nth ℓ 2 |> int_of_string) rows in
  let mean = List.fold_left ( + ) 0 grades / List.length grades in
  mean
```

We run the program, get some value `m`, and we are done!
Unfortunately, students often make us notice that we did not grade them correctly.
So you look back at the work and produce a **change**, e.g.:

[1] Please, imagine that there is 10^{40} rows in that file.

A first-order example

Let us assume that `rows` is a bunch of grades coming from a CSV file [1]:

```
let rows = [ "Pédrot"  , "Pierre-Marie", "17";
             "Knuth"   , "Donald"      , "12";
             "Torvald", "Linus"        , "10"  ]
```

The Ministry wants us to compute the mean of these grades. That's easy:

```
let myprog rows =
  let grades = List.map (fun ℓ -> List.nth ℓ 2 |> int_of_string) rows in
  let mean = List.fold_left ( + ) 0 grades / List.length grades in
  mean
```

We run the program, get some value `m`, and we are done!
Unfortunately, students often make us notice that we did not grade them correctly.
So you look back at the work and produce a **change**, e.g.:

In row 0, the value of column 2 moves from 17 to 11.

How should we update `m` without recomputing everything?

[1] Please, imagine that there is 10^{40} rows in that file.

Can we find a derivative for `myprog`?

```
let extract = fun ℓ -> List.nth ℓ 2 |> int_of_string

let sigma = List.fold_left ( + ) 0

let myprog rows =
  let grades  = List.map extract rows in
  let count   = List.length grades in
  let sum     = sigma grades in
  let mean    = sum / count in
  mean

type 'da dlist = ChangeItem of int * 'da

type dstring = ReplaceString of string

and dint = ReplaceInt of int | Shift of int

let apply_dint x = function
  | ReplaceInt y -> y
  | Shift dx -> x + dx

let apply_dstring _ (ReplaceString s) = s
```

Can we find a derivative for `myprog`?

```
let dint_of_string (ReplaceString new_value) =
  ReplaceInt (int_of_string new_value)

let dextract (ChangeItem (k, new_value)) =
  if k = 2 then dint_of_string new_value else Shift 0

let dmap df (ChangeItem (k, dx)) = ChangeItem (k, df dx)

let dlength _ = Shift 0

let dsigma (ChangeItem (_, Shift dx)) = Shift dx

let dmyprog rows drows =
  let dgrades = dmap dextract drows in
  let dcount  = dlength dgrades in
  let dsum    = dsigma dgrades in
  let dmean   = ...
  in
  dmean
```

Can we find a derivative for myprog?

```
let dint_of_string (ReplaceString new_value) =
  ReplaceInt (int_of_string new_value)

let dextract (ChangeItem (k, new_value)) =
  if k = 2 then dint_of_string new_value else Shift 0

let dmap df (ChangeItem (k, dx)) = ChangeItem (k, df dx)

let dlength _ = Shift 0

let dsigma (ChangeItem (_, Shift dx)) = Shift dx

let dmyprog rows drows =
  let dgrades = dmap dextract drows in
  let dcount  = dlength dgrades in
  let dsum    = dsigma dgrades in
  let dmean   = Shift (
       (apply_dint sum dsum) / (apply_dint count dcount)
       - sum / count
  )
  in
  dmean
```

Can we find a derivative for `myprog`?

```
let dint_of_string (ReplaceString new_value) =
  ReplaceInt (int_of_string new_value)

let dextract (ChangeItem (k, new_value)) =
  if k = 2 then dint_of_string new_value else Shift 0

let dmap df (ChangeItem (k, dx)) = ChangeItem (k, df dx)

let dlength _ = Shift 0

let dsigma (ChangeItem (_, Shift dx)) = Shift dx

let dmyprog rows drows =
  let grades  = List.map extract rows in
  let dgrades = dmap dextract drows in
  let count    = List.fold_left ( + ) 0 grades in
  let dcount  = dlength dgrades in
  let sum     = List.fold_left ( + ) 0 grades in
  let dsum    = dsigma dgrades in
  let dmean   = Shift (
       (apply_dint sum dsum) / (apply_dint count dcount)
       - sum / count
  )
  in
  dmean
```

Can we find a derivative for `myprog`?

(We will try again later.)

Next level : An higher-order example

```
let product l1 l2 =
  List.map (fun x ->
    List.map (fun y -> (x, y)) l2
  ) l1

let dproduct l1 dl1 l2 dl2 = ?
```

Next level : An higher-order example

```
let product l1 l2 =
  let inner x = (fun y -> (x, y)) in
  let outer x =
    let inner' = inner x in
    List.map inner' l2
  in
  List.map outer l1

let dproduct l1 dl1 l2 dl2 =
  ...
     List.dmap inner' dinner' l2 dl2
  ...
  List.dmap outer douter l1 dl1
```

Next level : An higher-order example

```
let product l1 l2 =
  let inner x = (fun y -> (x, y)) in
  let outer x =
    let inner' = inner x in
    List.map inner' l2
  in
  List.map outer l1

let dproduct l1 dl1 l2 dl2 =
  ...
     List.dmap inner' dinner' l2 dl2
  ...
  List.dmap outer douter l1 dl1
```

`douter` is a **function change**. What is that?

Two subproblems

1. How should we define ΔA, ΔB, $\oplus_A$, and $\oplus_B$?
2. How to get this miraculous $D(f)$?

Plan

Introduction

Some structure for first-class changes

Incrementalize this!

How should we equip incremental programmers?

Where we are and what we are up to

Change structures

1. How should we define ΔA, ΔB, $\oplus_A$, and $\oplus_B$?

Change structures

1. How should we define ΔA, ΔB, $\oplus_A$, and $\oplus_B$?

Incremental language designers do not actually agree on this question...

Giarrusso's change structures

A **complete change structure** is a tuple $(A, \Delta, \oplus, \ominus)$ such that:

- A is a type.
- $\Delta : A \to \texttt{Type}$
  where for all a of type A, the inhabitants of Δa are valid changes for a.
- $\oplus : \forall (x : A), \Delta x \to A$
  where $a \oplus da$ is the application of the change da to a.
- $\ominus : A \to \forall (x : A), \Delta x$
  where $a \oplus (b \ominus a) = b$.

Alvarez and Ong's change actions

A **change action** is a tuple $(A, \Delta A, \oplus, \odot, \mathbf{0})$ such that:

- ΔA is a type for changes.
- $M_\Delta = (\Delta A, \odot, \mathbf{0})$ is a monoid.
- $\oplus : A \times \Delta A \to A$ is an action of the monoid M_Δ on A.

Gonzalez' displaceable types

A type A is **displaceable** by $(\Delta A, \oplus, \ominus, \mathbf{0}, \odot)$ if

- ΔA is a type for changes.
- $M_\Delta = (\Delta A, \odot, \mathbf{0})$ is a monoid.
- $\oplus : A \times \Delta A \nrightarrow A$ is an "action" of the monoid M_Δ on A.
- $\ominus : A \to A \to \Delta A$ where $a \oplus (b \ominus a) = b$.

Rich change structures

A **rich change structure** is a tuple $(A, \Delta A, \mathcal{V}, \oplus, \odot, \mathbf{0}, \ominus, !)$ such that:

- A is a type and ΔA is a type for changes.
- $\mathcal{V} : A \to \Delta A \to \texttt{Prop}$ is a validity predicate for change.
- $\Delta : A \to \texttt{Type}$ as a subset type $\Delta x \triangleq \{dx : A \mid \mathcal{V}\, x\, dx\}$
- $\oplus : \forall (x : A), \Delta x \to A$
  where $a \oplus da$ is the application of the change da to a.
- $\odot : \forall (x : A)(dx : \Delta x) \to \Delta (x \oplus dx) \to \Delta x$
  is an associative change composition operator, behaving as an action on A.
- $\mathbf{0} : \forall (x : A), \Delta x$
  is such that $\forall x, x \oplus \mathbf{0}\, x = x$ and behaves as an identity for $\odot$.
- $\ominus : A \to \forall (x : A), \Delta x$
  where $a \oplus (b \ominus a) = b$.
- $! : \forall (y : A), A \to \Delta y$

Change-related definitions

Equivalence of changes

Let $x : A$ and $dx_1\ dx_2 : \Delta x$.
The two changes dx_1 and dx_2 are **equivalent**, written $dx_1 \equiv dx_2$, if:

$$x \oplus dx_1 = x \oplus dx_2$$

Change structure examples : natural numbers

- Take $\Delta\mathbb{N} = \mathbb{Z}$ and $\odot = +_{\mathbb{Z}}$
- The validity predicate $\mathcal{V}\,n\,k$ is defined as $(k < 0) \rightarrow (-k < n)$.
- Then, $n \oplus k = n +_{\mathbb{Z}} k$ and $\ominus = -_{\mathbb{Z}}$.
- The nil change is 0 for all n.

Change structure examples : products

If $(A, \Delta A, \mathcal{V}_A, \oplus_A, \odot_A, \mathbf{0}_A, \ominus_A)$ and $(B, \Delta B, \mathcal{V}_B, \oplus_B, \odot_B, \mathbf{0}_B, \ominus_B)$ are two change structures, then, by lifting the two set of operations to products, $(A \times B, \Delta A \times \Delta B, \mathcal{V}_{A\times B}, \oplus_{A\times B}, \odot_{A\times B}, \mathbf{0}_{A\times B}, \ominus_{A\times B})$ is also a change structure.

Change structure examples : sums

- Take $\Delta(A+B) = \Delta A + \Delta B + A + B$
- $\mathcal{V}_{A+B}\, s\, ds$ if

$$
\begin{aligned}
(\exists\, a\, da, s = \mathbf{in}_1\ a \wedge ds = \mathbf{in}_1\ da) \quad &\vee \quad (\exists\, b\, db, s = \mathbf{in}_2\ b \wedge ds = \mathbf{in}_2\ db) \vee \\
(\exists\, a', ds = \mathbf{in}_3\ a') \quad &\vee \quad (\exists\, b', ds = \mathbf{in}_4\ b')
\end{aligned}
$$

- $\mathbf{0}(\mathbf{in}_1 a) = \mathbf{0}\ a$ and $\mathbf{0}(\mathbf{in}_2 b) = \mathbf{0}\ b$.
- Exercise: Define $\oplus$, $\ominus$ and $\odot$!

Change structure examples : functions (Gonzalez' style)

- Take $\Delta(A \to B) = A \to \Delta B$.
- Lift the change structure over B in a pointwise way.
- For instance, change application is:

$$f \oplus df = \lambda x. f\ x \oplus df\ x$$

- For nil change:

$$\mathbf{0} f = \lambda x. \mathbf{0}(f\ x)$$

Change structure examples : functions (Giarrusso's style)

- Take $\Delta(A \to B) = A \to \Delta A \to \Delta B$.
- For the change application, Giarrusso uses:

$$f \oplus df = \lambda x. f\ x \oplus df\ x\ (\mathbf{0}\ x)$$

- Because of the need for:

$$(f \oplus df)\ (x \oplus dx) = f\ x \oplus df\ x\ dx$$

- In that setting, $\mathbf{0} f$ must therefore enjoy:

$$(f \oplus (\mathbf{0}\ f))\ (x \oplus dx) = f\ x \oplus (\mathbf{0}\ f)\ x\ dx = f\ (x \oplus dx)$$

- That is, $\mathbf{0}\ f$ must be a derivative of f.

Validity for function changes

$$\mathcal{V}\, f\, df = \begin{cases} \forall a\, da, \mathcal{V}_A\, a\, da \to \mathcal{V}_B\, (f\, a)\, (df\, a\, da) \wedge \\ \forall a\, da, f\, a \oplus df\, a\, da = f\, (a \oplus da) \oplus df\, (a \oplus da)\, (\mathbf{0}\, (a \oplus da)) \end{cases}$$

Plan

Introduction

Some structure for first-class changes

Incrementalize this!

How should we equip incremental programmers?

Where we are and what we are up to

A toy compiler for arithmetic expressions

```
(** Abstract syntax trees for arithmetic expressions. *)
type exp = EInt of int | EBin of op * exp * exp and op = Add | Mul

(** Instructions of a stack machine. *)
type instr = IPush of int | IAdd | IMul

(** We want a compiler from arithmetic expressions to instructions. *)
type source = exp and target = instr list

(** [compile] is defined by induction over arithmetic expressions. *)
let rec compile : source -> target = function
  | EInt d -> [IPush d]
  | EBin (op, lhs, rhs) -> compile lhs @ compile rhs @ [to_instr op]

and to_instr = function Add -> IAdd | Mul -> IMul
```

Source code changes

```
(** A rich set of changes for the abstract syntax trees. *)
type dexp =
    ReplaceEInt   of int                (* Replace a literal. *)
  | ReplaceOp     of op                 (* Replace an operation. *)
  | ChangeLeft    of dexp               (* Apply a change on lhs. *)
  | ChangeRight   of dexp               (* Apply a change on rhs. *)
  | LeftInsertOp  of op * exp           (* Insert an operation with rhs *)
  | RightInsertOp of op * exp           (* Insert an operation with lhs *)
  | ProjLeft                            (* Keep only lhs. *)
  | ProjRight                           (* Keep only rhs. *)
  | BinOpToEInt   of int                (* Change an operation into a literal. *)
  | EIntToBinOp   of op * exp * exp (* Change a literal into an operation. *)
  | DExpNil                             (* Change nothing. *)
```

Source change application

```
(** Here is how some of these changes can be applied to ASTs. *)
let apply_dexp e de =
  match e, de with
  | EInt x, ReplaceEInt y -> EInt y
  | EInt x, EIntToBinOp (op, lhs, rhs) -> EBin (op, lhs, rhs)
  | EBin (b, lhs, rhs), BinOpToEInt x -> EInt x
  | EBin (b, lhs, rhs), ProjLeft -> lhs
  | EBin (b, lhs, rhs), ProjRight -> rhs
  | EBin (b, lhs, rhs), ReplaceOp b' -> EBin (b, lhs, rhs)
  | e, LeftInsertOp (op, lhs) -> EBin (op, lhs, e)
  | e, RightInsertOp (op, rhs) -> EBin (op, e, rhs)
  | _, _ -> failwith "Invalid change"
```

Source change application

```
(** Here is how some of these changes can be applied to ASTs. *)
let apply_dexp e de =
  match e, de with
  | EInt x, ReplaceEInt y -> EInt y
  | EInt x, EIntToBinOp (op, lhs, rhs) -> EBin (op, lhs, rhs)
  | EBin (b, lhs, rhs), BinOpToEInt x -> EInt x
  | EBin (b, lhs, rhs), ProjLeft -> lhs
  | EBin (b, lhs, rhs), ProjRight -> rhs
  | EBin (b, lhs, rhs), ReplaceOp b' -> EBin (b, lhs, rhs)
  | e, LeftInsertOp (op, lhs) -> EBin (op, lhs, e)
  | e, RightInsertOp (op, rhs) -> EBin (op, e, rhs)
  | _, _ -> failwith "Invalid change"
```

- Did I miss some cases?
- With some extra pain, you can define `compose_dexp`.

...and now?

```
(** [compile] is defined by induction over arithmetic expressions. *)
let rec compile : source -> target = function
  | EInt d -> [IPush d]
  | EBin (op, lhs, rhs) -> compile lhs @ compile rhs @ [to_instr op]

and to_instr = function Add -> IAdd | Mul -> IMul

(** [dcompile source dsource] computes how [compile source] should be
    changed if [source] is changed by [dsource]. *)
let dcompile : source -> dsource -> dtarget = ?
```

A programming challenge

- Derivatives are often **partial functions**.

> Can you remove an element from an empty list?
> The program safety depends on the **validity of changes**.

A programming challenge

- Derivatives are often **partial functions**.
- Derivatives are defined by **many cases**.

  If a datatype has n cases and if there is m distinct kind of changes, prepare yourself to consider $n * m$ cases (and many make no sense)!

A programming challenge

- Derivatives are often **partial functions**.
- Derivatives are defined by **many cases**.
- Efficient derivatives are often **program dependent**.

There is no magic wand.
Efficient derivatives exploit mathematical properties of functions.

A programming challenge

- Derivatives are often **partial functions**.
- Derivatives are defined by **many cases**.
- Efficient derivatives are often **program dependent**.
- Incremental programming is **algorithmically challenging**.

> An incrementalization must share information with its base computation.
> Use **retroactive data structures** to efficiently store and update it.

A programming challenge

- Derivatives are often **partial functions**.
- Derivatives are defined by **many cases**.
- Efficient derivatives are often **program dependent**.
- Incremental programming is **algorithmically challenging**.
- Incremental programming **hardly scales** to large programs.

  Manual incrementalization of small functions is hard but feasible.
  Large programs have no obvious derivatives.

A programming challenge

- Derivatives are often **partial functions**.
- Derivatives are defined by **many cases**.
- Efficient derivatives are often **program dependent**.
- Incremental programming is **algorithmically challenging**.
- Incremental programming **hardly scales** to large programs.
- The complexity of incremental programs is **hard to reason about**.

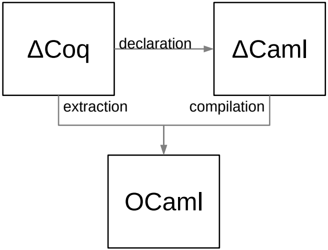
> A tiny change of the inputs can have a large impact on the outputs.
> The complexity is better expressed w.r.t the size of the output update.
> Require reasoning about $f\ x$, $f\,(x \oplus dx)$ and $D(f)\ x\ dx$.

A programming challenge

- Derivatives are often **partial functions**.
- Derivatives are defined by **many cases**.
- Efficient derivatives are often **program dependent**.
- Incremental programming is **algorithmically challenging**.
- Incremental programming **hardly scales** to large programs.
- The complexity of incremental programs is **hard to reason about**.

Plan

Our take on this programming challenge



- For a function `f` for which a "smart" incrementalization is not obvious:

$\Rightarrow$ ΔCaml provides **`derive f`**, an automatic incrementalization of `f`.

- For a function `f` for which the programmer has some intuition:

$\Rightarrow$ ΔCoq assists the programmer through the incrementalization process.

The quest for automatic differentiation

2. How to get this miraculous $D(f)$?

The quest for automatic differentiation

2. How to get this miraculous $D(f)$?

- Easy! Take:

$$D(f)\,x\,dx = \lambda x\,dx. f\,(x \oplus dx) \ominus f\,x$$

The quest for automatic differentiation

2. How to get this miraculous $D(f)$?

- Easy! Take:

$$D(f)\,x\,dx = \lambda x\,dx.f\,(x \oplus dx) \ominus f\,x$$

- This is a too naive! $D(f)$ must be more efficient than recomputation!

The quest for automatic differentiation

2. How to get this miraculous $D(f)$?

- Easy! Take:

$$D(f)\,x\,dx = \lambda x\,dx. f\,(x \oplus dx) \ominus f\,x$$

- This is a too naive! $D(f)$ must be more efficient than recomputation!
- Two more realistic approaches:
  - Gonzalez' partial derivatives ;
  - Giarrusso's static differentiation.

Partial derivatives à la Gonzalez

Let's extend the standard call-by-value λ-calculus with $\mathcal{D}(\bullet)$ ruled by:

$$\mathcal{D}(\lambda x.t) \to \lambda x\, dx. \frac{\partial t}{\partial x} \quad \text{where}$$

$$\begin{cases} \dfrac{\partial y}{\partial x} = \begin{cases} dx & \text{if } y = x \\ \mathbf{0}\, y & \text{otherwise} \end{cases} \\ \dfrac{\partial(\lambda y.t)}{\partial x} = \lambda y. \dfrac{\partial t}{\partial x} \qquad \text{if } x \neq y \\ \dfrac{\partial \mathcal{D}(t)}{\partial x} = \mathcal{D}(\dfrac{\partial t}{\partial x}) \\ \dfrac{\partial (r\ s)}{\partial x} = \left(\mathcal{D}(r)\ s\, \dfrac{\partial s}{\partial x}\right) \odot \left(\dfrac{\partial r}{\partial x}\, (x \oplus \dfrac{\partial s}{\partial x})\right) \end{cases}$$

Partial derivatives à la Gonzalez

Theorem (Chain rule)

The chain rule holds for the deterministic differential λ-calculus.

$$\mathcal{D}(\lambda x.(f \circ g)\, x) \to \lambda x\, dx.\mathcal{D}(f)\, (g\, x)\, (\mathcal{D}(g)\, x\, dx)$$

Theorem (Soundness of dynamic differentiation)

Let f be function. The following equation holds:

$$f\, (x \oplus dx) = f\, x \oplus \mathcal{D}(f)\, x\, dx$$

where the equality stands for the definitional equivalence.

- Add a rule for your favorite primitives and their derivatives, and voilà!
- $\mathcal{D}(\bullet)$ lifts primitive derivatives to higher-order programs.
- A framework to reason about derivatives, inspired by Differential λ-calculus.

Partial derivatives à la Gonzalez

Theorem (Chain rule)

The chain rule holds for the deterministic differential λ-calculus.

$$\mathcal{D}(\lambda x.(f \circ g)\, x) \to \lambda x\, dx.\mathcal{D}(f)\,(g\, x)\,(\mathcal{D}(g)\, x\, dx)$$

Theorem (Soundness of dynamic differentiation)

Let f be function. The following equation holds:

$$f\,(x \oplus dx) = f\, x \oplus \mathcal{D}(f)\, x\, dx$$

where the equality stands for the definitional equivalence.

- ▶ Add a rule for your favorite primitives and their derivatives, and voilà!
- ▶ $\mathcal{D}(\bullet)$ lifts primitive derivatives to higher-order programs.
- ▶ A framework to reason about derivatives, inspired by Differential λ-calculus.
- ✗ Unfortunately, partial derivatives require huge implementation efforts...

Static differentiation (Giarrusso et al, PLDI'14)

Giarrusso et al study the following stunningly simple **program transformation**:

$$\begin{array}{rcl} \mathcal{D}(x) & = & dx \\ \mathcal{D}(t\,u) & = & \mathcal{D}(t)\,u\,\mathcal{D}(u) \\ \mathcal{D}(\lambda x.t) & = & \lambda x\,dx.\mathcal{D}(t) \end{array}$$

Static differentiation (Giarrusso et al, PLDI'14)

Giarrusso et al study the following stunningly simple **program transformation**:

$$\begin{array}{rcl} \mathcal{D}(x) & = & dx \\ \mathcal{D}(t\,u) & = & \mathcal{D}(t)\,u\,\mathcal{D}(u) \\ \mathcal{D}(\lambda x.t) & = & \lambda x\,dx.\mathcal{D}(t) \end{array}$$

- It performs **static differentiation** w.r.t. all free variables at once.
- As a program transformation, it can be easily embedded in a compiler.

Static differentiation (Giarrusso et al, PLDI'14)

Giarrusso et al study the following stunningly simple **program transformation**:

$$\begin{array}{rcl} \mathcal{D}(x) & = & dx \\ \mathcal{D}(t\,u) & = & \mathcal{D}(t)\,u\,\mathcal{D}(u) \\ \mathcal{D}(\lambda x.t) & = & \lambda x\,dx.\mathcal{D}(t) \end{array}$$

- It performs **static differentiation** w.r.t. all free variables at once.
- As a program transformation, it can be easily embedded in a compiler.

Theorem (Soundness of static differentiation)

If $f : A \to B$, $a : A$ and $da : \Delta A$ is a valid change for a, then the following holds:

$$f\ (a \oplus da) \simeq f\ a \oplus \mathcal{D}(f)\ a\ da$$

were $\simeq$ denotes the (definitional) equality of denotations.

Inefficiency of Giarrusso's static differentiation

```
let average : int list -> int = fun xs ->
  let s = sum xs in
  let n = len xs in
  let d = div s n in
  d
```

Applied to `average`, static differentiation produces the following derivative:

```
let daverage : int list -> (int, Δint) Δlist -> Δint
= fun xs dxs ->
  let s = sum xs and ds = dsum xs dxs in
  let n = len xs and dn = dlen xs dxs in
  let d = div s n and dd = ddiv s ds n dn in
  dd

let ddiv s ds n dn = (s ⊕ ds) / (n ⊕ dn)
```

Inefficiency of Giarrusso's static differentiation

```
let average : int list -> int = fun xs ->
  let s = sum xs in
  let n = len xs in
  let d = div s n in
  d
```

Applied to `average`, static differentiation produces the following derivative:

```
let daverage : int list -> (int, Δint) Δlist -> Δint
= fun xs dxs ->
  let s = sum xs and ds = dsum xs dxs in
  let n = len xs and dn = dlen xs dxs in
  let d = div s n and dd = ddiv s ds n dn in
  dd

let ddiv s ds n dn = (s ⊕ ds) / (n ⊕ dn)
```

`ddiv` needs `s` (i.e. `sum xs`) even though `average xs` already computed it!

Static differentiation in Cache Transfer Style (ESOP'19)

In CTS, a function returns a cache of its intermediate results:

```
let cts_average : int list -> int * cache_average = fun xs ->
  let s, cache_sum = cts_sum xs in
  let n, cache_len = cts_len xs in
  let d, cache_div = cts_div s n in
  (d, (s, cache_sum, n, cache_len, d, cache_div))
```

In CTS, a derivative exploits and updates this cache:

```
let cts_daverage
: cache_average -> int list -> (int, Δint) Δlist -> Δint * cache_average
= fun cache xs dxs ->
  let (s, cache_sum, n, cache_length, d, cache_div) = cache in
  let ds, cache_sum = dsum cache_sum xs dxs in
  let dn, cache_len = dlen cache_len xs dxs in
  let dd, cache_div = ddiv cache_div s ds n dn in
  (dd, (s ⊕ ds, cache_sum, n ⊕ dn, cache_len, d ⊕ dd, cache_div))
```

Status of CTS differentiation

In the paper

- A new soundness proof of differentiation (in an untyped setting).
- A soundness proof of the CTS differentiation.
- Preliminary benchmarks show that resulting incrementalizations are of an order of magnitude faster than recomputing.

Now

- The implementation of ΔCaml is work-in-progress.
- ΔCaml is core ML + change structures + derivatives.
- The transformation requires terms to be in λ-lifted A-normal form.

Towards the certification of hand-written CTS derivatives

How should we design the ΔCoq library?

We are trying to answer this through a case study : an incremental **`List`** module.

Which change structure for Lists?

If $(A, \Delta A, \mathcal{V}_A, \oplus_A, \odot_A, \mathbf{0}_A, \ominus_A)$ is a change structure, then let us take

$$\Delta \mathsf{list}\ A ::= \mathsf{Insert}_k\ a \mid \mathsf{Remove}_k\ a \mid \mathsf{Update}_k\ a\ da \mid \mathsf{Compose}\ dl\ dl \mid \mathsf{NilChange}$$

where we take $k \in \mathbb{N}$, $a \in A$, $da \in \Delta A$, and $dl \in \Delta \mathsf{list}\ A$.

`List.map`

How would you incrementalize `List.map`?

List.map

How would you incrementalize List.map?

```
let rec dmap_nil f df dl =
  match dl with
  | Insert k a -> Insert k (f a)
  | Remove k a -> Remove k (f a)
  | Update k a da -> Update k (f a) (df a da)
  | Compose dl1 dl2 -> Compose (dmap_nil f df dl1) (dmap_nil f df dl2)
  | NilChange -> NilChange

let dmap f df l dl =
  if is_nil df then dmap_nil f df dl else ! (map (f ⊕ df) (l ⊕ dl))
```

- The caches are omitted because they are not necessary for List.map.

`List.fold_left`

How would you incrementalize `List.fold_left`?

`List.fold_left`

How would you incrementalize `List.fold_left`?

- If you know nothing about `f`:
  - Take a cache that remembers all the intermediate values of the accumulator.
  - Restart the iteration from the position of the change.
  - Worst-case: $O(|l|)$.
- If you know that `f` is commutative and inversible:
  - There is no need for a cache.
  - Undo/Update the contribution of the element at the change position.
  - Worst-case: $(O(1))$
- If you know that `f` is associative:
  - Take a cache which is a (differential variant of a) fingertree.
  - Split the fingertree at the change position, apply the change and join the fingertree back.
  - Worst-case: $O(log_2(l))$.

Plan

Introduction

Some structure for first-class changes

Incrementalize this!

How should we equip incremental programmers?

Where we are and what we are up to

How does it compare with self-adjusting computations?

Why don't you use Acar's self-adjusting computations?

How does it compare with self-adjusting computations?

Why don't you use Acar's self-adjusting computations?

What are these self-adjusting computations?

- They are instrumented to build a graph representing their execution traces.
- Output changes are obtained by **propagating changes** in the graph.
- Tremendous performances thanks to aggressive memoization.

How does it compare with self-adjusting computations?

Why don't you use Acar's self-adjusting computations?

What are these self-adjusting computations?

- They are instrumented to build a graph representing their execution traces.
- Output changes are obtained by **propagating changes** in the graph.
- Tremendous performances thanks to aggressive memoization.

But ...

- It is a dynamic and imperative process in a graph: hard to reason about.

$\Rightarrow$ A derivative is simply a new program compatible with usual verification tools.

How does it compare with self-adjusting computations?

Why don't you use Acar's self-adjusting computations?

What are these self-adjusting computations?

- They are instrumented to build a graph representing their execution traces.
- Output changes are obtained by **propagating changes** in the graph.
- Tremendous performances thanks to aggressive memoization.

But ...

- It is a dynamic and imperative process in a graph: hard to reason about.

$\Rightarrow$ A derivative is simply a new program compatible with usual verification tools.

- Acar's notion of changes is based on replacement.

$\Rightarrow$ We believe that more structured changes open better opportunities.

Towards cache communication

```
let rec sort = function
  ...
| x :: xs ->
  let cmp, cmp_cache = less_than x in
  let (xless, xmore), partition_cache = partition cmp xs in
  ...
```

```
let rec dsort (sorted_list, cmp_cache, partition_cache, ...) =
  ...
  (* Case for ``Insert k a'' *)
  let dcmp, dcmp_cache = dless_than cmp_cache dx in
  let (dxless, dxmore), partition_cache =
    dpartition partition_cache dcmp (Insert k a)
  in
  ...
```

- `dpartition` has a $O(n)$ worst-case complexity.

Towards cache communication

```
let rec sort = function
  ...
| x :: xs ->
  let cmp, cmp_cache = less_than x in
  let (xless, xmore), partition_cache = partition cmp xs in
  ...
```

```
let rec dsort (sorted_list, cmp_cache, partition_cache, ...) =
  ...
  (* Case for ``Insert k a'' *)
  let dcmp, dcmp_cache = dless_than cmp_cache dx in
  let (dxless, dxmore), partition_cache =
    dpartition partition_cache dcmp (Insert k a)
  in
  ...
```

- `dpartition` has a $O(n)$ worst-case complexity.
- But by exploiting `sorted_list` this could be reduced to $log(n)$!

Towards cache communication

```
let rec sort = function
  ...
| x :: xs ->
  let cmp, cmp_cache = less_than x in
  let (xless, xmore), partition_cache = partition cmp xs in
  ...
```

```
let rec dsort (sorted_list, cmp_cache, partition_cache, ...) =
  ...
  (* Case for ``Insert k a'' *)
  let dcmp, dcmp_cache = dless_than cmp_cache dx in
  let (dxless, dxmore), partition_cache =
    dpartition partition_cache dcmp (Insert k a)
  in
  ...
```

- `dpartition` has a $O(n)$ worst-case complexity.
- But by exploiting `sorted_list` this could be reduced to $log(n)$!
- The cache of `sort` has information about values processed by `partition`.
- Can we share information between caches?

Conclusion

Where we are

- Cache-Transfer-Style differentiation is a program transformation to incrementalize higher-order programs.
- We have a Coq proof and several experiments in OCaml.

Conclusion

Where we are

- Cache-Transfer-Style differentiation is a program transformation to incrementalize higher-order programs.
- We have a Coq proof and several experiments in OCaml.

What we are up to

- Implementing ΔCaml and ΔCoq to conduct large experiments.
- Studying a theory of caches.

Conclusion

Where we are

- Cache-Transfer-Style differentiation is a program transformation to incrementalize higher-order programs.
- We have a Coq proof and several experiments in OCaml.

What we are up to

- Implementing ΔCaml and ΔCoq to conduct large experiments.
- Studying a theory of caches.

Thank you for attention! Any questions?