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)

2019-09-30
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
Data constantly change

Data \( x \) at \( t = 0 \)

\[ f(x) \text{ at } t = 0 \]

Change

Data \( x \) at \( t = 1 \)

\[ f(x) \text{ at } t = 1 \]

Data \( x \) at \( t = 2 \)

\[ f(x) \text{ at } t = 2 \]

Now, take size \((x) = 2^{50}\) and size \((\text{modified part of } x) = 2^{10}\).

Recomputation is not an option!
Data constantly change

▶ Now, take $\text{size}(x) = 2^{50}$ and $\text{size}(\text{modified part of } x) = 2^{10}$.
▶ 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, ...)
Incremental programming with first-class changes

Data $x$ at $t = 0$

$D(f)$

$f(x + dx_1)$

$D(f)$

$f(x + dx_1 + dx_2)$
Incremental programming with first-class changes

\[
\begin{align*}
\text{If } & \quad f : A \rightarrow B \\
\Delta A & \text{ are changes over } A \text{ and } \Delta B \text{ are changes over } B \\
\oplus_A : A \rightarrow \Delta A \rightarrow A \text{ and } \oplus_B : A \rightarrow \Delta B \rightarrow B
\end{align*}
\]

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 \( \begin{align*}
\{ 
& f : A \to B \\
\Delta A \text{ are changes over } A \text{ and } \Delta B \text{ are changes over } B \\
\oplus_A : A \to \Delta A \to A \text{ and } \oplus_B : A \to \Delta B \to B
\end{align*} \)

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 \( f \).
Incremental programming with first-class changes

\[
\begin{align*}
    f : A & \to B \\
    \Delta A & \text{ are changes over } A \text{ and } \Delta B \text{ are changes over } B \\
    \oplus_A : A & \to \Delta A \to A \text{ and } \oplus_B : A & \to \Delta B \to B
\end{align*}
\]

then use \( D(f) \) such that:

\[
f (x \oplus_A dx) = f x \oplus_B D(f) x \, dx
\]

where the complexity of \( D(f) \)

\begin{itemize}
    \item (should ideally) only depends on the size of \( dx \), and
    \item (always) be better than the complexity of \( f \).
\end{itemize}

1. How should we define \( \Delta A, \Delta B, \oplus_A \), 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
1. How should we define $\Delta A$, $\Delta B$, $\oplus_A$, and $\oplus_B$?
Change structures

1. How should we define $\Delta A$, $\Delta B$, $\oplus_A$, 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.
- \(\Delta : A \rightarrow \text{Type}\)
  where for all \(a\) of type \(A\), the inhabitants of \(\Delta a\) are valid changes for \(a\).
- \(\oplus : \forall (x : A), \Delta x \rightarrow A\)
  where \(a \oplus da\) is the application of the change \(da\) to \(a\).
- \(\ominus : A \rightarrow \forall (x : A), \Delta x\)
  where \(a \ominus (b \ominus a) = b\).
Alvarez and Ong’s change actions

A change action is a tuple \((A, \Delta A, \oplus, \circ, \emptyset)\) such that:

▶ \(\Delta A\) is a type for changes.
▶ \(M_{\Delta} = (\Delta A, \circ, \emptyset)\) is a monoid.
▶ \(\oplus : A \times \Delta A \to A\) is an action of the monoid \(M_{\Delta}\) on \((A, \oplus)\).
A type $A$ is **displaceable** by $(\Delta A, \oplus, \ominus, \Theta, \odot)$ if

- $\Delta A$ is a type for changes.
- $M_{\Delta} = (\Delta A, \odot, \Theta)$ is a monoid.
- $\oplus : A \times \Delta A \to A$ is an action of the monoid $M_{\Delta}$ on $(A, \oplus)$.
- $\ominus : A \to A \to \Delta A$ where $a \oplus (b \ominus a) = b$. 
Rich change structures

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

- \(A\) is a type and \(\Delta A\) is a type for changes.
- \(\mathcal{V} : A \to \Delta A \to \text{Prop}\) is a validity predicate for change.
- \(\Delta : A \to \text{Type}\) is defined as a Prop irrelevant 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 \(a\).
- \(\odot : \forall(x : A)(dx : \Delta x) \to \Delta(x \odot dx) \to \Delta x\)
  is an associative change composition operator, behaving as an action on
  \((A, \oplus)\).
- \(\Theta : \forall(x : A), \Delta x\)
  is such that \(\forall x, x \oplus \Theta x = x\) and behaves as an identity for \(\odot\).
- \(\ominus : A \to \forall(x : A), \Delta x\)
  where \(a \ominus (b \ominus a) = b\).
- \(! : \forall(y : A), A \to \Delta y\)
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 $\Delta N = \mathbb{Z}$ 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 \((A, \Delta A, \nu_A, \oplus_A, \odot_A, \ominus_A, \oslash_A)\) and \((B, \Delta B, \nu_B, \oplus_B, \odot_B, \ominus_B, \oslash_B)\) are two change structures, then, by lifting the two set of operations to products, \((A \times B, \Delta A \times \Delta B, \nu_{A \times B}, \oplus_{A \times B}, \odot_{A \times B}, \ominus_{A \times B}, \oslash_{A \times B})\) is also a change structure.
Change structure examples: sums

- Take $\Delta(A + B) = \Delta A + \Delta B + A + B$
- $\forall A+B s \, ds$ if

  \[(\exists a \, da, s = \text{in}_1 a \land ds = \text{in}_1 da) \lor (\exists b \, db, s = \text{in}_2 b \land ds = \text{in}_2 db) \lor (\exists a' \, ds = \text{in}_3 a') \lor (\exists b' \, ds = \text{in}_4 b')\]

- $\mathcal{O}(\text{in}_1 a) = \mathcal{O} a$ and $\mathcal{O}(\text{in}_2 b) = \mathcal{O} b$.
- 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:

$$\emptyset f = \lambda x. \emptyset(f x)$$
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:

$$f \oplus df = \lambda x. f \ x \oplus df \ x \ (\emptyset x)$$

- Because of the need for:

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

- In that setting, $\emptyset f$ must therefore enjoy:

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

- That is, $\emptyset f$ must be a derivative of $f$. 
Validity for function changes

\[ \forall f \, df = \begin{cases} 
\forall a \, da, \forall A \, a \, da \rightarrow \forall B \, (f \, a) \, (df \, a \, da) \land \\
\forall a \, da, f \, a \oplus df \, a \, da = f (a \oplus da) \oplus df (a \oplus da) (\emptyset (a \oplus da)) 
\end{cases} \]
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. *)

```ocaml
type exp = EInt of int | EBin of op * exp * exp and op = Add | Mul
```

(** Instructions of a stack machine. *)

```ocaml
type instr = IPush of int | IAdd | IMul
```

(** We want a compiler from arithmetic expressions to instructions. *)

```ocaml
type source = exp and target = instr list
```

(** [compile] is defined by induction over arithmetic expressions. *)

```ocaml
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. *)

```ocaml
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. *)
```
(** Here is how some of these changes can be applied to ASTs. *)

```ocaml
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.
Source change application

```ocaml
(* 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?

```plaintext
(** [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 dsourse] computes how [compile source] should be changed if [source] is changed by [dsourse]. *)

let dcompile : source -> dsourse -> 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 \( n \times m \) 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 $D(f) \cdot dx$. 
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

Introduction

Some structure for first-class changes

Incrementalize this!

How should we equip incremental programmers?

Where we are and what we are up to
For a function \( f \) for which a “smart” incrementalization is not obvious:

\[ \Rightarrow \] \( \Delta \) Caml provides \texttt{derive} \( f \), an automatic incrementalization of \( f \).

For a function \( f \) for which the programmer has some intuition:

\[ \Rightarrow \] \( \Delta \) Coq assists the programmer through the incrementalization process.
2. How to get this miraculous $D(f)$?
The quest for automatic differentiation

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

Easy! Take:

$$D(f) \, dx = \lambda x \, dx \cdot f(x \oplus dx) \oplus f \, x$$
The quest for automatic differentiation

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

- Easy! Take:
  \[
  D(f) \, x \, dx = \lambda x \, dx \cdot f(x \oplus dx) \ominus f \, x
  \]

- This is a too naive! \( D(f) \) must be more efficient than recomputation!
2. How to get this miraculous $D(f)$?

- Easy! Take:
  $$D(f) \ dx = \lambda x \ dx. f(x \oplus dx) \ominus f \ x$$

- 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 $\lambda$-calculus with $D(\bullet)$ ruled by:

\[
D(\lambda x.t) \rightarrow \lambda x \ dx \ . \ \frac{\partial t}{\partial x}
\]

where

\[
\begin{align*}
\frac{\partial y}{\partial x} &= \begin{cases} 
    dx & \text{if } y = x \\
    0 & \text{otherwise}
\end{cases} \\
\frac{\partial (\lambda y.t)}{\partial x} &= \lambda y . \frac{\partial t}{\partial x} & \text{if } x \neq y \\
\frac{\partial D(t)}{\partial x} &= D \left( \frac{\partial t}{\partial x} \right) \\
\frac{\partial (r s)}{\partial x} &= \left( D(r) \ s \ \frac{\partial s}{\partial x} \right) \otimes \left( \frac{\partial r}{\partial x} \ (x \oplus \frac{\partial s}{\partial x}) \right)
\end{align*}
\]
Partial derivatives à la Gonzalez

Theorem (Chain rule)
The chain rule holds for the deterministic differential $\lambda$-calculus.

$$D(\lambda x. (f \circ g) x) \rightarrow \lambda x dx.D(f)(g x)(D(g) x dx)$$

Theorem (Soundness of dynamic differentiation)
Let $f$ be function. The following equation holds:

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

where the equality stands for the definitional equivalence.

- Add a rule for your favorite primitives and their derivatives, and voilà!
- $D(\bullet)$ lifts primitive derivatives to higher-order programs.
- A framework to reason about derivatives, inspired by Differential $\lambda$-calculus.
Partial derivatives à la Gonzalez

Theorem (Chain rule)
The chain rule holds for the deterministic differential $\lambda$-calculus.

$$D(\lambda x. (f \circ g) x) \rightarrow \lambda x dx. D(f) (g x) (D(g) x dx)$$

Theorem (Soundness of dynamic differentiation)
Let $f$ be function. The following equation holds:

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

where the equality stands for the definitional equivalence.

- Add a rule for your favorite primitives and their derivatives, and voilà!
- $D(\bullet)$ lifts primitive derivatives to higher-order programs.
- A framework to reason about derivatives, inspired by Differential $\lambda$-calculus.
- Unfortunately, partial derivatives require huge implementation efforts...
Giarrusso et al study the following stunningly simple program transformation:

\[
\begin{align*}
\mathcal{D}(x) &= dx \\
\mathcal{D}(tu) &= \mathcal{D}(t) u \mathcal{D}(u) \\
\mathcal{D}(\lambda x. t) &= \lambda x \ dx \mathcal{D}(t)
\end{align*}
\]
Giarrusso et al study the following stunningly simple program transformation:

\[
\begin{align*}
D(x) &= dx \\
D(tu) &= D(t)uD(u) \\
D(\lambda x.t) &= \lambda x dx.D(t)
\end{align*}
\]

- It performs static differentiation w.r.t. all free variables at once.
- As a program transformation, it can be easily embedded in a compiler.
Giarrusso et al study the following stunningly simple **program transformation**:

\[
\begin{align*}
D(x) & = dx \\
D(t \ u) & = D(t) \ u \ D(u) \\
D(\lambda x. t) & = \lambda x \ dx \ D(t)
\end{align*}
\]

- 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 \( f : A \rightarrow B, \ a : A \) and \( da : \Delta A \) is a valid change for \( a \), then the following holds:

\[
\begin{align*}
f (a \oplus da) \simeq f \ a \oplus D(f) \ a \ da
\end{align*}
\]

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, \int) \Delta list -> \Delta int
    = 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, △int) △list -> △int
  = 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!
In CTS, a function returns a cache of its intermediate results:

```plaintext
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:

```plaintext
let cts_daverage : cache_average -> int list -> (int, △int) △list -> △int * 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 △Caml is work-in-progress.
➢ △Caml 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 Coq library?

We are trying to answer this through a case study: an incremental List module.
Which change structure for Lists?

If \((A, \Delta A, \nu_A, \oplus_A, \odot_A, \Theta_A, \Theta_A)\) is a change structure, then let us take

\[
\Delta \text{list } A ::= \text{Insert}_k a \mid \text{Remove}_k a \mid \text{Update}_k a \, da \mid \text{Compose } dl \, dl \mid \text{NilChange}
\]

where we take \(k \in \mathbb{N}, a \in A, da \in \Delta A,\) and \(dl \in \Delta \text{list } A.\)
How would you incrementalize `List.map`?
How would you incrementalize `List.map`?

```
let rec dmapNil 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 (dmapNil f df dl1) (dmapNil f df dl2)
  | NilChange -> NilChange

let dmap f df l dl =
  if isNil df then dmapNil 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`?
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 inverse:
  - 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?

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 …

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

Acar’s notion of changes is based on replacement.

We believe that more structured changes open better opportunities.
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.
- A derivative is simply a new program compatible with usual verification tools.
How does it compare with 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.
- A derivative is simply a new program compatible with usual verification tools.
- Acar’s notion of changes is based on replacement.
- We believe that more structured changes open better opportunities.
Towards cache communication

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

```ocaml
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

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

```ocaml
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.

Where we go

▶ Implementing $\Delta$Caml and $\Delta$Coq to conduct large experiments.
▶ Studying a theory of caches.

Thank you for attention! Any questions?
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.

Where we go

- Implementing △Caml and △Coq 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.

Where we go

► Implementing \(\Delta\)Caml and \(\Delta\)Coq to conduct large experiments.
► Studying a theory of caches.

Thank you for attention! Any questions?