Incrementalizing λ-Calculi by Static Differentiation

A Theory of Changes for Higher-Order Languages and Ongoing Work

Paolo Giarrusso PPS, 22-01-2015
(with Yufei Cai, Tillmann Rendel, Klaus Ostermann)

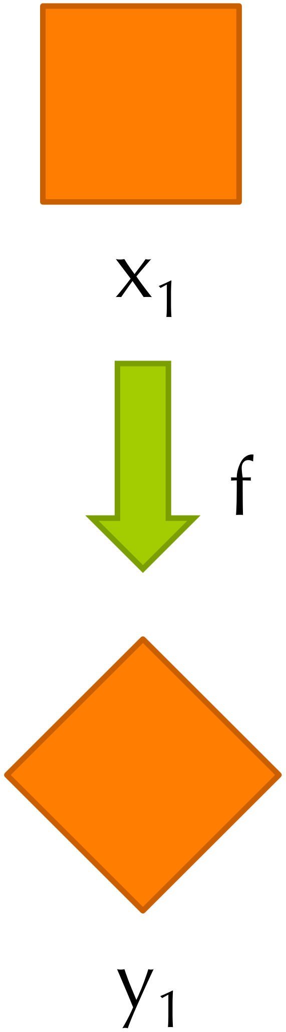
Tübingen University

Incrementalizing λ-Calculi by Static Differentiation

Program → **Incremental program** → **Optimized incremental program**

Problem: Incremental computation

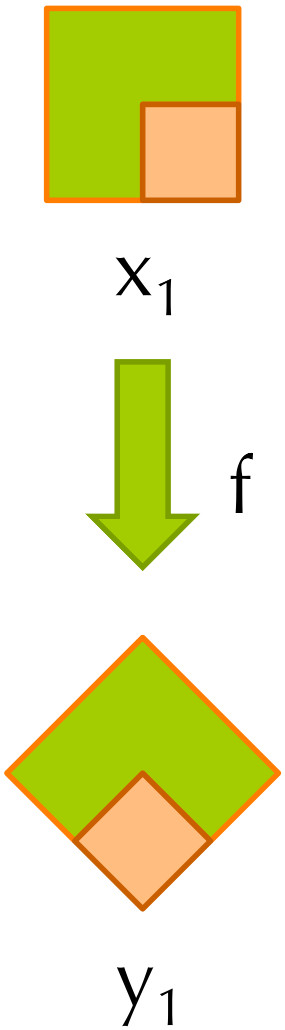
- ✔ Support for a language with first-class functions!
- ✔ Mechanized proof in Agda
- ✔ Implementation in Scala
- ✔ Performance case-study


x_1
f
y_1

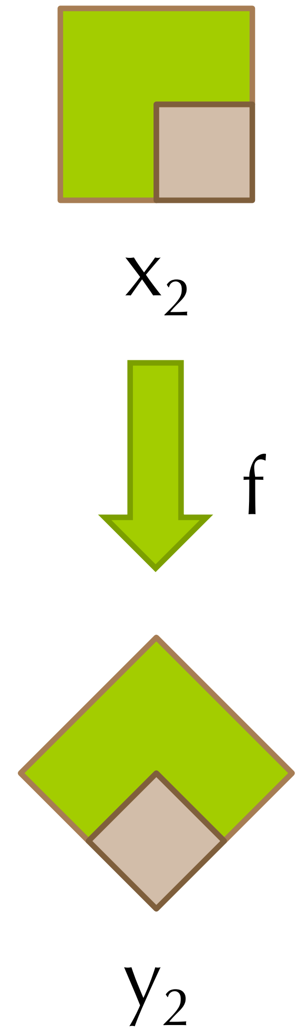

x_1
x_2
f
f
y_1
y_2


x_1
x_2
f
f
y_1
y_2


x_1
x_2
f
f
y_1
y_2



f invoked again! ☹



General examples

- Task: Compute statistics on a database of all citizens of France
  - Each time something changes, update statistics
  - Changes are small
  - Can update results without recomputation?
- Variant: statistics on Twitter timelines
  - And keep these statistics up-to-date in real-time.

Examples

- Task: typecheck & compile a program, or a proof script (say, in Coq)
  - Change: Update a basic definition of the program
  - Changes are still “small”
  - Can update results without recomputation?


x_1
x_2
f
f
y_1
y_2

Running example

- Sum numbers from a collection
- **Base** input collection x_1: {{**1**,1,2,3,4}}
- **Updated** input collection x_2: {{1,2,3,4,**5**}}
- The collection is a bag (that is, a multiset)
  - Like in sequences, elements can be repeated
  - Like in sets, order is irrelevant

Example

$f\ coll = fold\ (+)\ 0\ coll$

$y = f\ x$

$x_1 = \{\{\mathbf{1},1,2,3,4\}\}$ base input

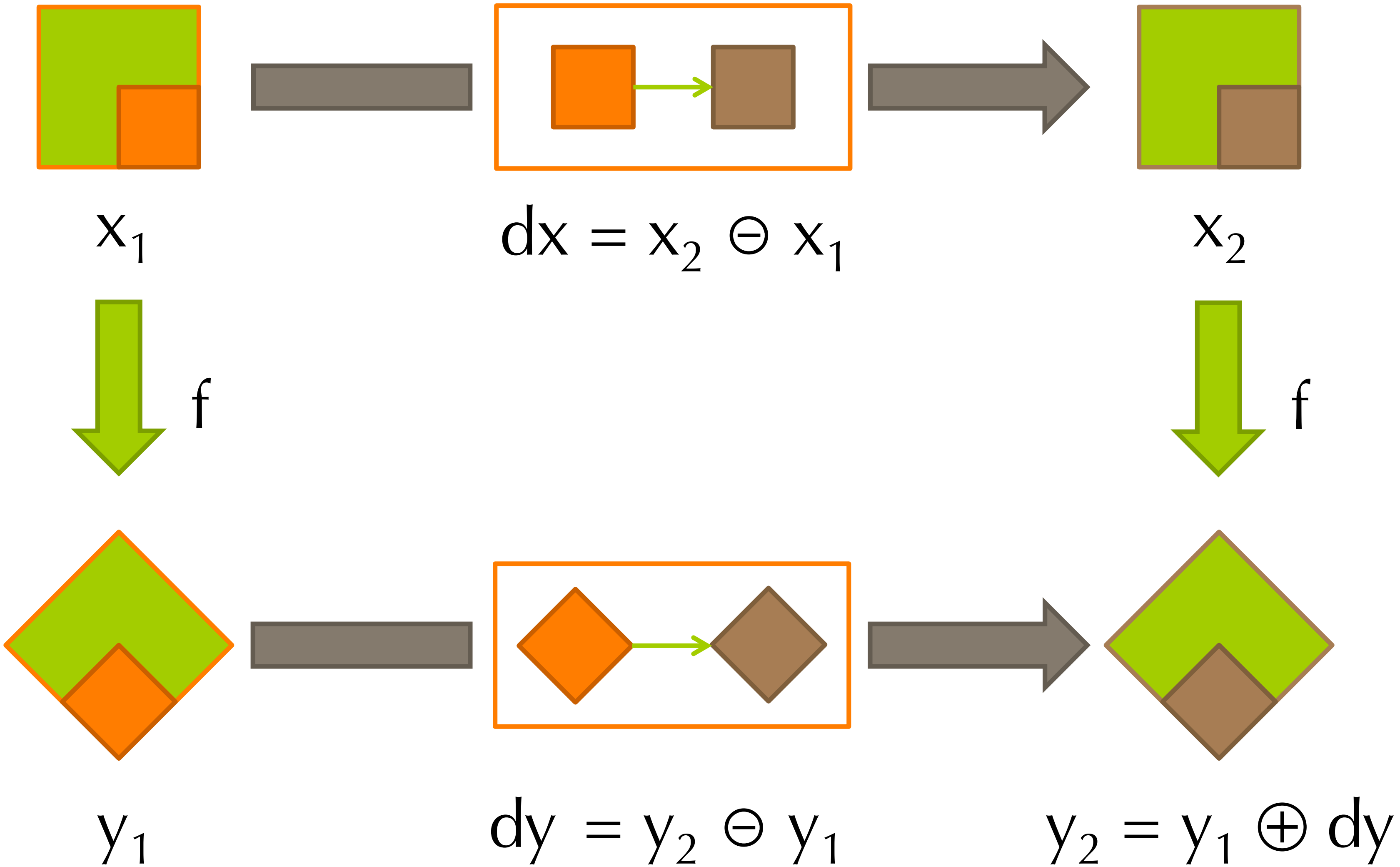
$y_1 = \mathbf{1} + 1 + 2 + 3 + 4 = \mathbf{11}$ base output

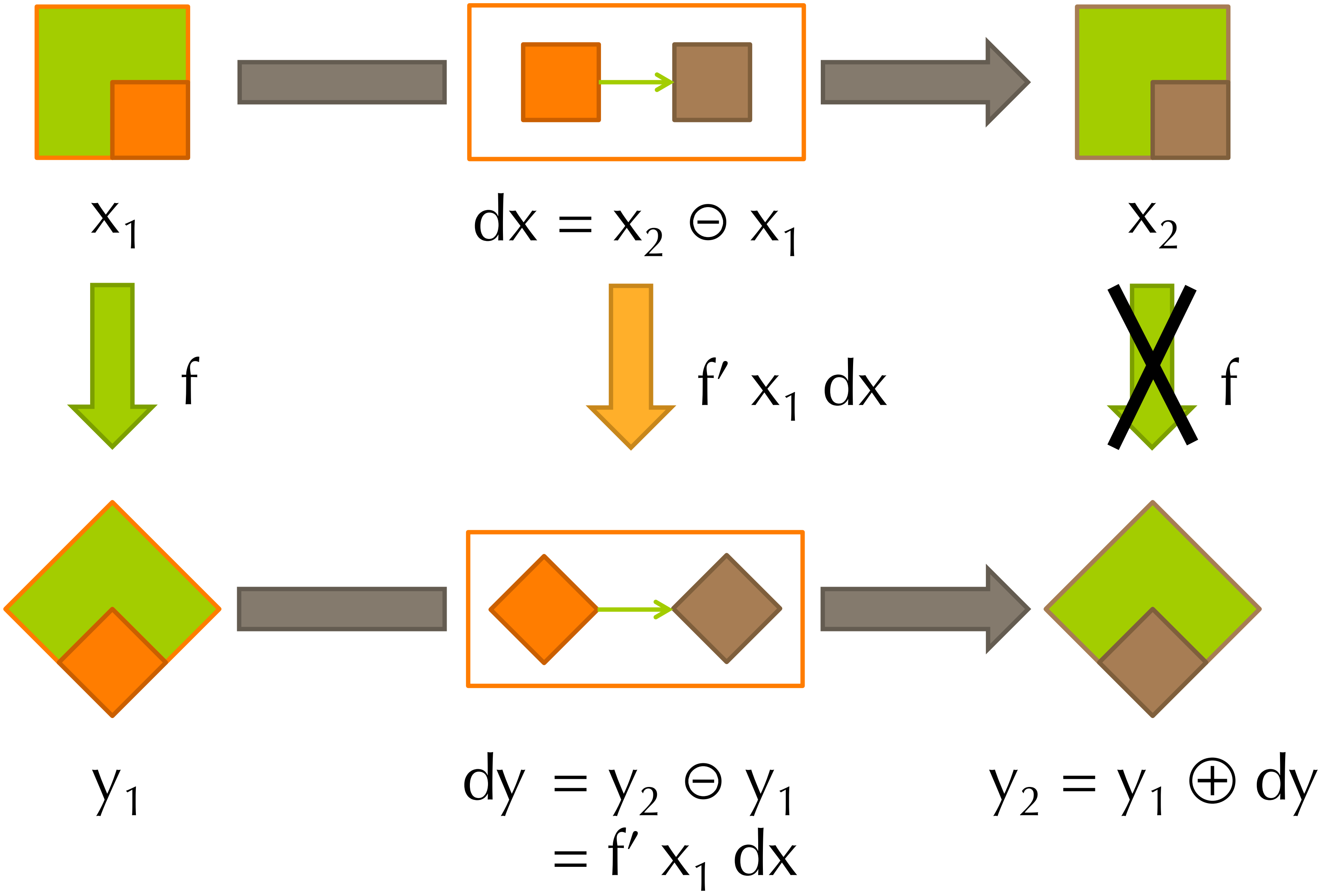
$x_2 = \{\{1,2,3,4,\mathbf{5}\}\}$ upd. input

$y_2 = 1 + 2 + 3 + 4 + \mathbf{5}$ upd. output

$= \mathbf{15} = s_1 - 1 + 5$


x1
f
y1
x2
f
y2


x_1
$dx = x_2 \ominus x_1$
x_2
f
f
y_1
$dy = y_2 \ominus y_1$
$y_2 = y_1 \oplus dy$


x_1
$dx = x_2 \ominus x_1$
x_2
f
$f' \; x_1 \; dx$
f
y_1
$dy = y_2 \ominus y_1$
$= f' \; x_1 \; dx$
$y_2 = y_1 \oplus dy$

Example

$$
\begin{aligned}
f\ coll &= fold\ (+)\ 0\ coll \\
y &= f\ x
\end{aligned}
$$

$$
\begin{aligned}
x_1 &= \{\{1,1,2,3,4\}\} \\
y_1 &= 11 \\
x_2 &= \{\{1,2,3,4,5\}\} \\
dx &= \{\{1,2,3,4,\mathbf{5}\}\} \ominus \{\{\mathbf{1},1,2,3,4\}\} = \{\{5, \underline{1}\}\} \\
y_2 &= x_1 \oplus f'\ x_1\ dx = 11 \oplus (-1 + 5) \\
&= 15
\end{aligned}
$$

Derivatives

f' is the *derivative* of f if

- input: base input x_1; a change dx valid for x
- output: change dy valid for *base output* $(f\ x)$
- correctness:

$$(f\ x_1) \oplus (f'\ x_1\ dx) = f\ (x_1 \oplus dx)$$

- Notation: application binds tighter than anything

$$f\ x_1 \oplus f'\ x_1\ dx = f\ (x_1 \oplus dx)$$

Using derivatives: idea

First, base computation:

$$y_1 = f\ x_1$$

Later, incremental computation "algorithm":

$$y_2 = y_1 \oplus dy = y_1 \oplus f'\ \ x_1\ \ dx$$

instead of

$$y_2 = f\ (x_1 \oplus dx)$$

Setting

- An algebraic theory of change structures for functions
  - To specify and reason about the problem
  - Using dependent types!
- A code transformation *Derive* produces derivatives of programs
  - simply-typed λ-calculus **programs** (STLC), parameterized by a plugin for constants and base types

Proof strategy

We decompose our transformation into 2 phases

- non-*standard denotational semantics*
  - simply-typed λ-calculus **programs** (STLC) → type theory **functions** (Agda)
- erasure to extract STLC programs
  - we should have used *modified realizability*?
- Proof each phase correct

Signature of change structures

Types

(C1) V type — *base type*

(C2) Δx type $\quad \forall\, x : V$ — *change types*

Operations

(C3) $x_1 \oplus dx : V \quad \forall\, dx : \Delta x_1$ — *update*

(C4) $x_2 \ominus x_1 : \Delta x_1$ — *difference*

Algebraic equations

(C5) $x_1 \oplus (x_2 \ominus x_1) = x_2$ — *cancellation*

Change structure for naturals

Let's define a change structure such that:

$$x \oplus dx = x + dx$$

$$x_2 \ominus x_1 = x_2 - x_1$$

like in the examples in the beginning of the talk.

Change structure for naturals

So we define:

(C1) *base type*: $\mathbb{N}$

(C2) *change types*:

$\boldsymbol{\Delta x = \{ dx \in \mathbb{Z} \mid x + dx \geq 0 \}}$

(C3) $x_1 \oplus dx = x_1 + dx : \mathbb{N}$

(C4) $x_2 \ominus x_1 = x_2 - x_1 : \Delta x_1$

(C5) $x_1 \oplus (x_2 \ominus x_1) = x_1 + (x_2 - x_1) = x_2$

Example derivatives

Remember: $y_2 = y_1 \oplus dy = y_1 \oplus f' \;\; x_1 \;\; dx$

id x = x

id′ x dx = dx

f x = x + 5

f′ x dx = dx

Change structures

- **Algebraic** theory of changes (ToC)
  - for **equational reasoning**
- Change types $\neq$ base type
  - (unlike calculus in math, Koch [2010], Gluche et al. [1997] in CS)
- ToC is about mathematical functions (in type theory), not programs
- ToC extended to programs through denotational semantics

An equivalence of changes?

- There can be multiple changes which "do the same thing"
- Example:

{{1,2,3,4,**5**}} ⊖ {{**1**,1,2,3,4}} can be represented by {{5, $\underline{1}$}} or by "change **1** through **+4**".

Change equivalence (d.o.e.)

Take $x \in V$, dx_1, $dx_2 \in \Delta x$

$dx_1 \triangleq dx_2$ iff

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

that is, have same effect when applied.

{{1,2,3,4,**5**}} ⊖ {{**1,**1,2,3,4}} can be represented by {{5, $\underline{1}$}} or by "change **1** through **+4**", so

{{5, $\underline{1}$}} ≜ "change **1** through **+4**"

Changes also form a category

- Objects: values of type V
- Arrows: an arrow from *a* to *b* is a (set of ≜ changes) going from *a* to *b*

Derived ops give a category

Derived ops

$\mathbf{0}_x = x \ominus x$ *nil change*

$dx_1 \odot dx_2 = (x_1 \oplus dx_1) \oplus dx_2 \ominus x_1$

change composition

Derived algebraic equations

$x \oplus \mathbf{0}_x = x$ *right unit for* $\oplus$

$dx \odot \mathbf{0} \triangleq \mathbf{0} \odot dx \triangleq dx$ *composition unit*

$(dx_1 \odot dx_2) \odot dx_3 \triangleq dx_1 \odot (dx_2 \odot dx_3)$

composition associativity

(Static) Differentiation

- Given a (simply-typed) λ-term f:

$$\boxed{\textbf{Program f}} \xrightarrow{\textbf{\textit{Derive}}} \boxed{\textbf{Derivative f}'}$$

- f′ is a λ-term, the *derivative* of f
- f′ can be optimized further!
- Correctness (proved in Agda):

$$[\![\ f\ (a \oplus da)\ ]\!] = [\![\ f\ a \ \oplus\ \textbf{\textit{Derive}}(f)\ a\ da]\!]$$

“Derivatives” are non-linear!

- Set $f' = \boldsymbol{Derive}(f)$
- $f'\ a\ (da \odot db) =$
  $f'\ a\ da \odot f'\ (a \oplus da)\ db \neq$
  $f'\ a\ da \odot f'\ a\ db$

Vs calculus

- That’s because $a \oplus da$ can’t be approximated with a, unlike in calculus:
    - changes do not “tend to zero” (“infinitesimal”), they are finite
- Incremental calculi (ours and other ones) are thus closer to the calculus of *finite differences* than the one of *derivatives*.

Vs *differential lambda calculus*

- Contrast with linearity in ***differential lambda calculus***:
  $\partial f/\partial x\cdot(dx + dy) = \partial f/\partial x\cdot dx + \partial f/\partial x\cdot dy$
- You can model $\partial f/\partial x\cdot dx$ with the substitution $x \mapsto x \oplus dx$… as $f[x \mapsto x \oplus dx] \ominus f$
- But it cannot be linear substitution!
- We must compute f on the new value of *x*, that is $x \oplus dx$, so we substitute everywhere.

- $[\![f\ (a \oplus da)]\!] = [\![f\ a \oplus f'\ a\ da]\!]$
- $[\![f\ (a \oplus da \oplus db)]\!] = [\![f\ a \oplus f'\ a\ (da \odot db)]\!]$

- $[\![f\ a \oplus f'\ a\ (da \odot db)]\!] = [\![f\ (a \oplus da \oplus db)]\!] =$
  $[\![f\ a \oplus f'\ a\ da \oplus f'\ (a \oplus da)\ db)]\!]$

Derivative examples #1

$$\mathrm{id}_T = \lambda\,(x : T).\ x$$

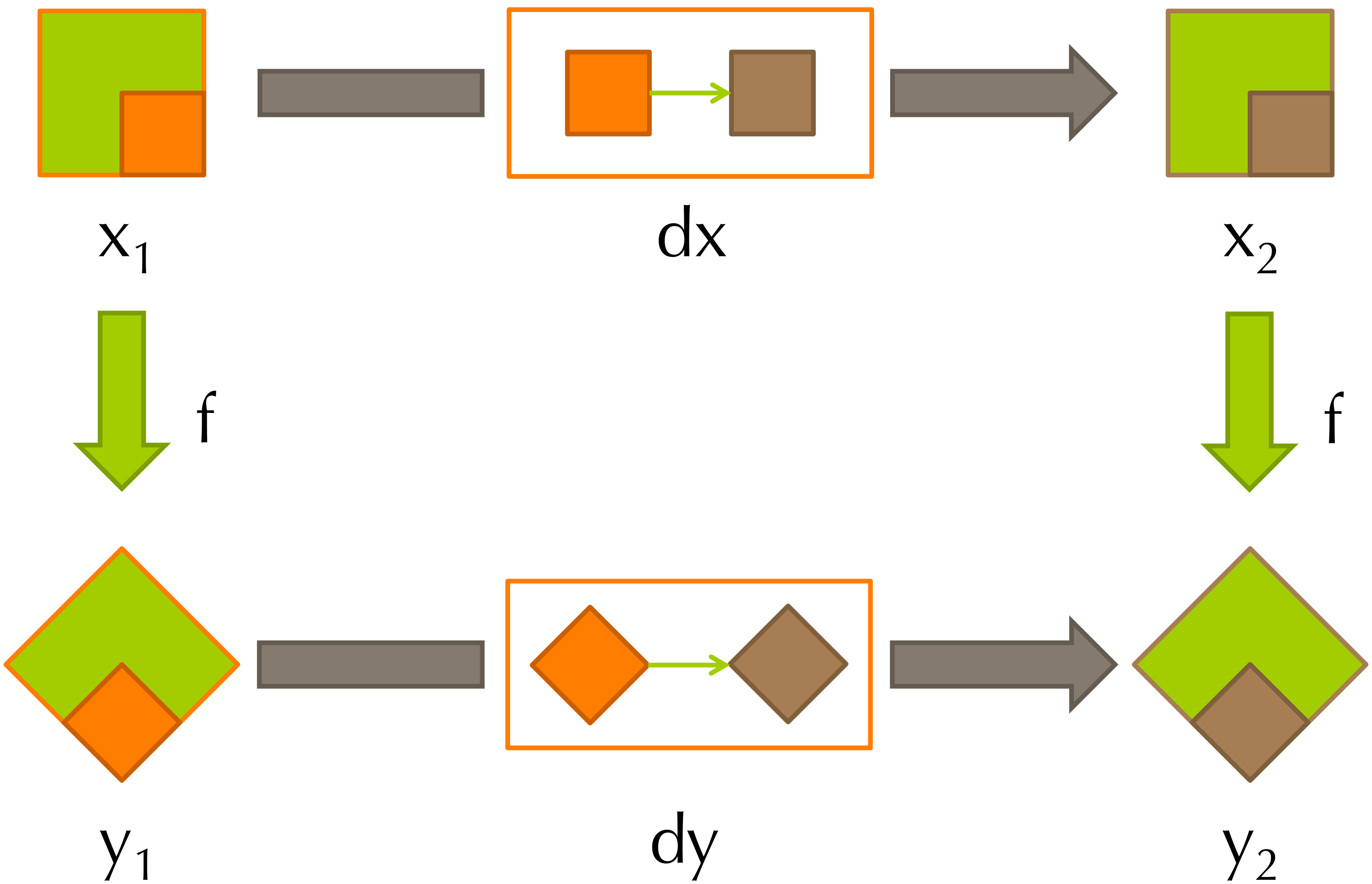
$$\mathrm{id}_T' = \boldsymbol{\mathit{Derive}}(\mathrm{id}_T) = \lambda\,(x : T)\,(dx : \Delta T).\ dx$$

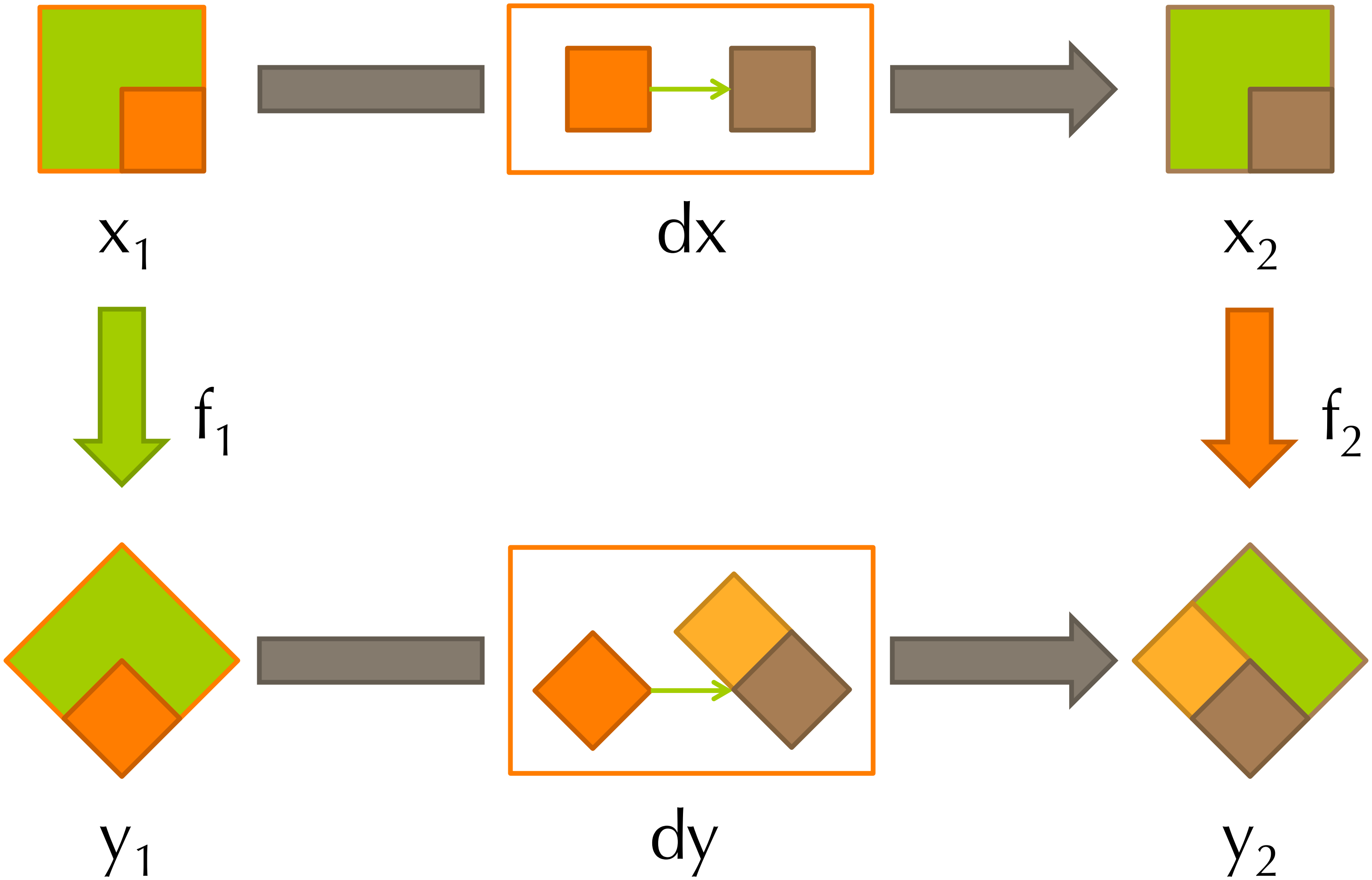
- ΔT, not Δx
  - no dependent types
- ΔT is expanded by ***Derive***
- changes (*dx*) are first-class

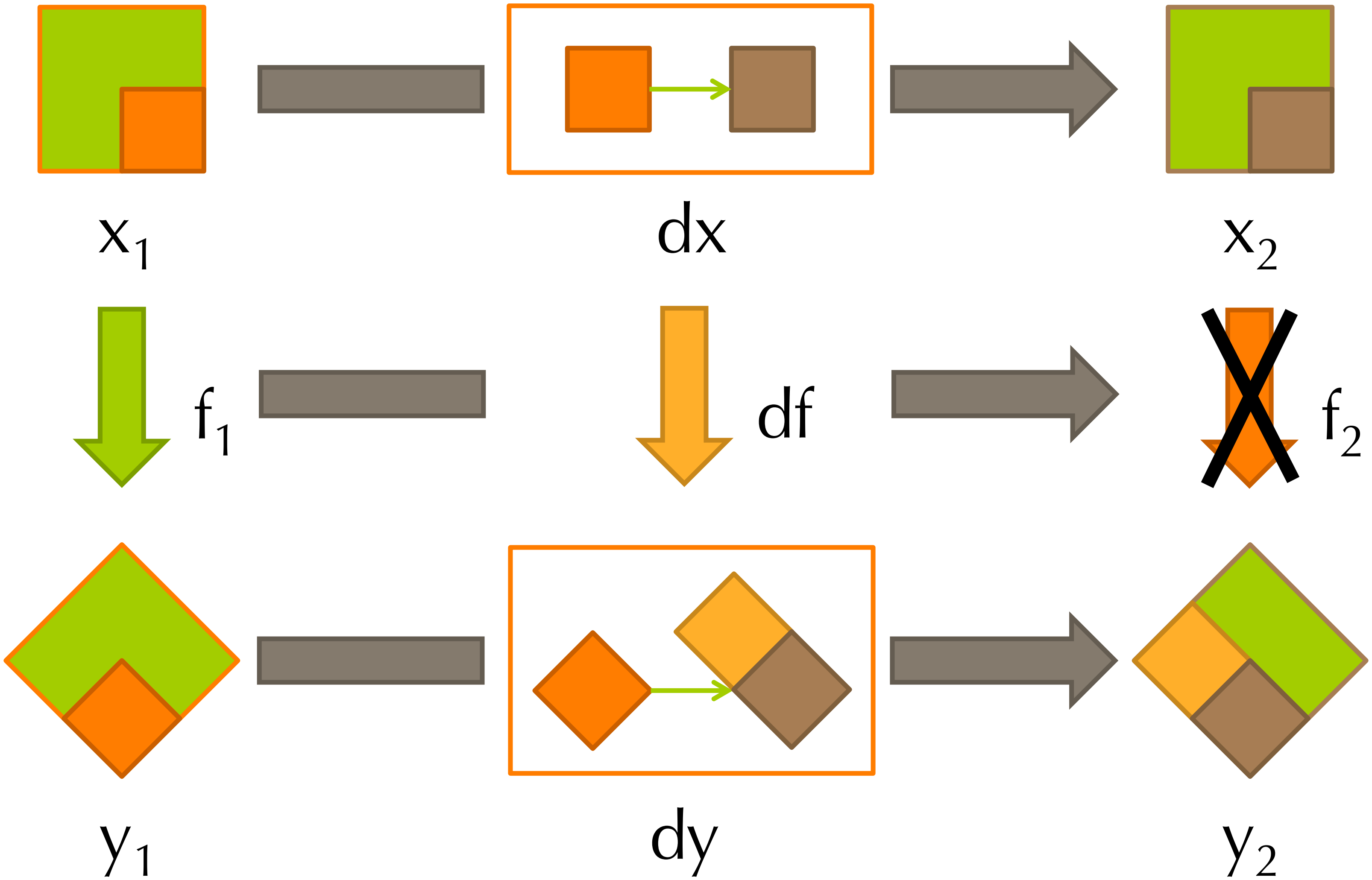
First-class functions

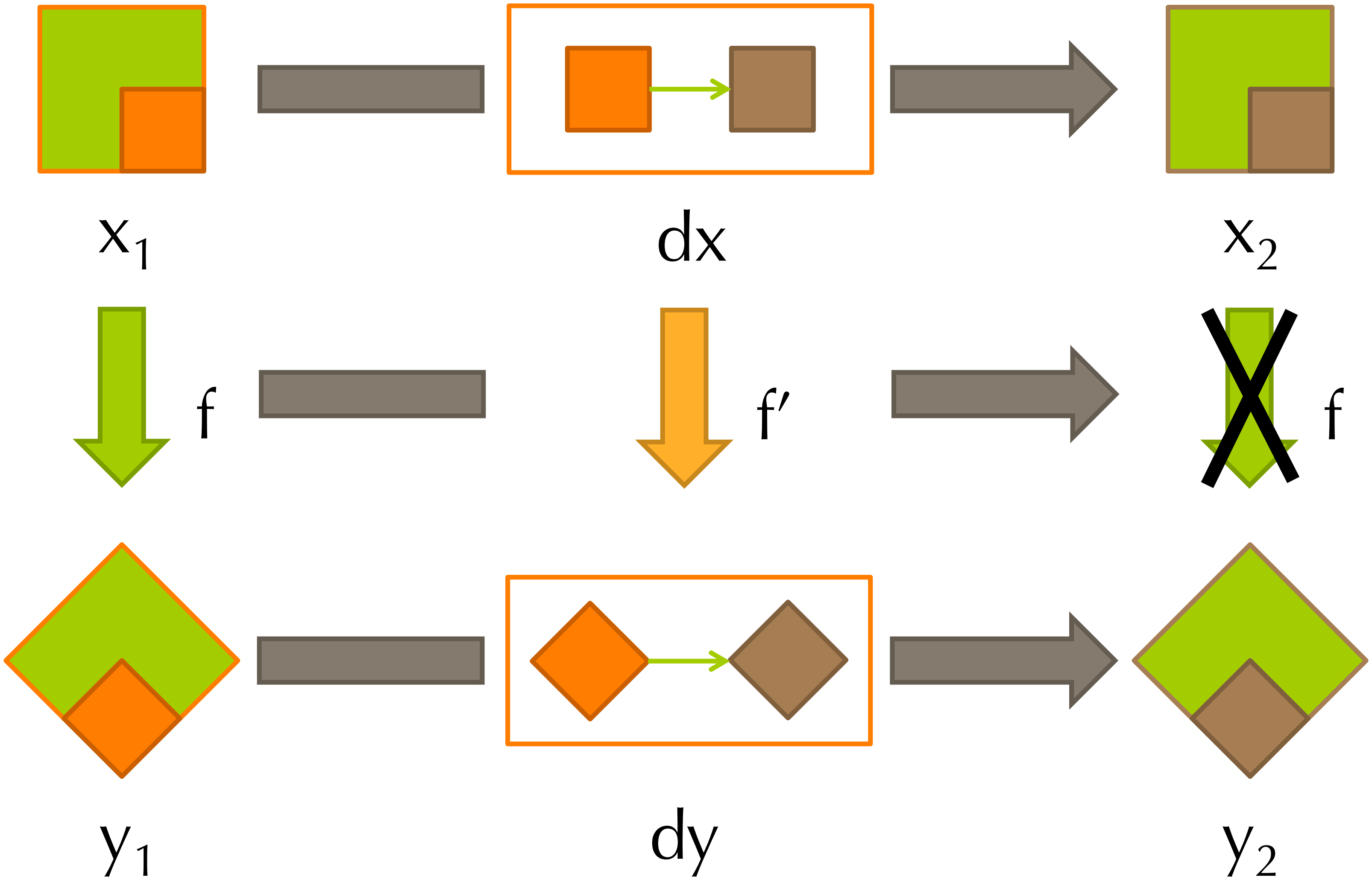
First-class functions

- Functions are data
- So they can change!
- Concretely, a closure changes if data in its environment changes


x1
dx
x2
f
f
y1
dy
y2


x_1
dx
x_2
f_1
f_2
y_1
dy
y_2


x1
dx
x2
f1
df
f2
y1
dy
y2


x1
dx
x2
f
f′
f
y1
dy
y2

Derivatives → function changes

From:

$$\mathbf{f'}\ x_1\ dx = \mathbf{f}\ x_2 \ominus \mathbf{f}\ x_1 = y_2 \ominus y_1 = dy$$

to:

$$\mathbf{df}\ x_1\ dx = \mathbf{f_2}\ x_2 \ominus \mathbf{f_1}\ x_1 = y_2 \ominus y_1 = dy$$

- Function values change, e.g. because data in closures change!
- Change structure for functions in paper

Change structure for functions

$$\Delta_{\sigma \to \tau} = \lambda\ (f : [\![\sigma \to \tau]\!]) \to \Sigma_{df\ :\ \forall (x\ :\ [\![\sigma]\!])\ (dx\ :\ \Delta\ x) \to \Delta\ (f\ x)}\ \mathit{valid}\ (f, df)$$

Derivative examples #2

$$\mathrm{id}_T = \lambda\,(x : T).\ x$$

$$\mathrm{id}_T{}' = \textbf{\textit{Derive}}(\mathrm{id}_T) = \lambda\,(x : T)\ (dx : \Delta T).\ dx$$

$$\mathrm{app}_{TU} = \lambda\,(f : T \to U)\ (x : T).\ f\ x$$

$$\mathrm{app}_{TU}{}' = \textbf{\textit{Derive}}(\mathrm{app}_{TU}) = \lambda\ (f : T \to U)\ (df : \Delta(T \to U))\ (x : T)\ (dx : \Delta T).\ df\ x\ dx$$

$$\Delta(T \to U) = T \to \Delta T \to \Delta U$$

Language

$$T ::= \iota \mid T_1 \to T_2$$

$$t ::=$$

$$s\ t \mid \lambda x^T.\ t \mid x^T \mid c$$

Base types and constants specified by a language plugin.

Deriving terms

We require that `Derive` satisfies admissible rule:

$$\frac{\Gamma \vdash t : T}{\Gamma, \Delta\Gamma \vdash \mathrm{Derive}\ (t) : \Delta T}$$

$\Delta\iota = \ldots$

$\Delta(T_1 \to T_2) = T_1 \to \Delta T_1 \to \Delta T_2$

Deriving terms

Propagate changes:

```
Derive(s t) = Derive(s) t Derive(t)
Derive(λx. t) = λx dx. Derive(t)
```

Return changes:

```
Derive(x) = dx
```

Change of primitives:

```
Derive(c) = dc
```

Deriving terms

- The derivative only "follows" the computation propagating changes
- Derivatives of primitives receive inputs and changes, and should compute output changes **efficiently**

Incrementalizing λ-calculi

- **Language plugins** define datatypes and their change structures
- They also define primitives and how to handle them
- Use existing/new research

Which primitives?

- 1st-class functions ⇒ few primitives (e.g. folds) required, other ops (e.g. map) in libraries
- Primitives encapsulate efficiently incrementalizable skeletons

Example

f coll = fold (+) 0 coll

y = f x

$coll_0 = \{\{1,1,2,3,4\}\}$

$coll_1 = \{\{1,2,3,4,5\}\}$

$dcoll = \{\{1,2,3,4,5\}\} \ominus \{\{1,1,2,3,4\}\} = \{\{5, \underline{1}\}\}$

What about the removal of $\underline{1}$?

Example

sum s = fold (+) 0 s
y = sum coll

dsum s ds = … = fold (+) 0 ds
dy = dsum coll dcoll

$coll_0 = \{\{1,2,3,4\}\}$
$coll_1 = \{\{2,3,4,5\}\}$
$dcoll = \{\{2,3,4,5\}\} \ominus \{\{1,2,3,4\}\} = \{\{\underline{1}, 5\}\}$

Running example & primitives

f coll = fold (+) 0 coll

y = f x

$x_1 = \{\{\mathbf{1},1,2,3,4\}\}$

$x_2 = \{\{1,2,3,4,\mathbf{5}\}\}$

$dx = \{\{1,2,3,4,\mathbf{5}\}\} \ominus \{\{\mathbf{1},1,2,3,4\}\} = \{\{5, \underline{1}\}\}$

What about $\underline{1}$, i.e. the removal of 1?

Running example & primitives

$f\ coll \quad = fold\ \mathbf{G}\ coll$ $\qquad$ **G** abelian group!

$y \quad = f\ x$

$x_1 \quad = \{\{\mathbf{1},1,2,3,4\}\}$

$x_2 \quad = \{\{1,2,3,4,\mathbf{5}\}\}$

$dx \quad = \{\{1,2,3,4,\mathbf{5}\}\} \ominus \{\{\mathbf{1},1,2,3,4\}\} = \{\{5, \underline{1}\}\}$

// if d**G** is the nil change of **G**

$df\ x_1\ dx = fold'\ \mathbf{G}\ d\mathbf{G}\ x_1\ dx = \ldots = fold\ \mathbf{G}\ dx = 4$

$dy = df\ x_1\ dx$

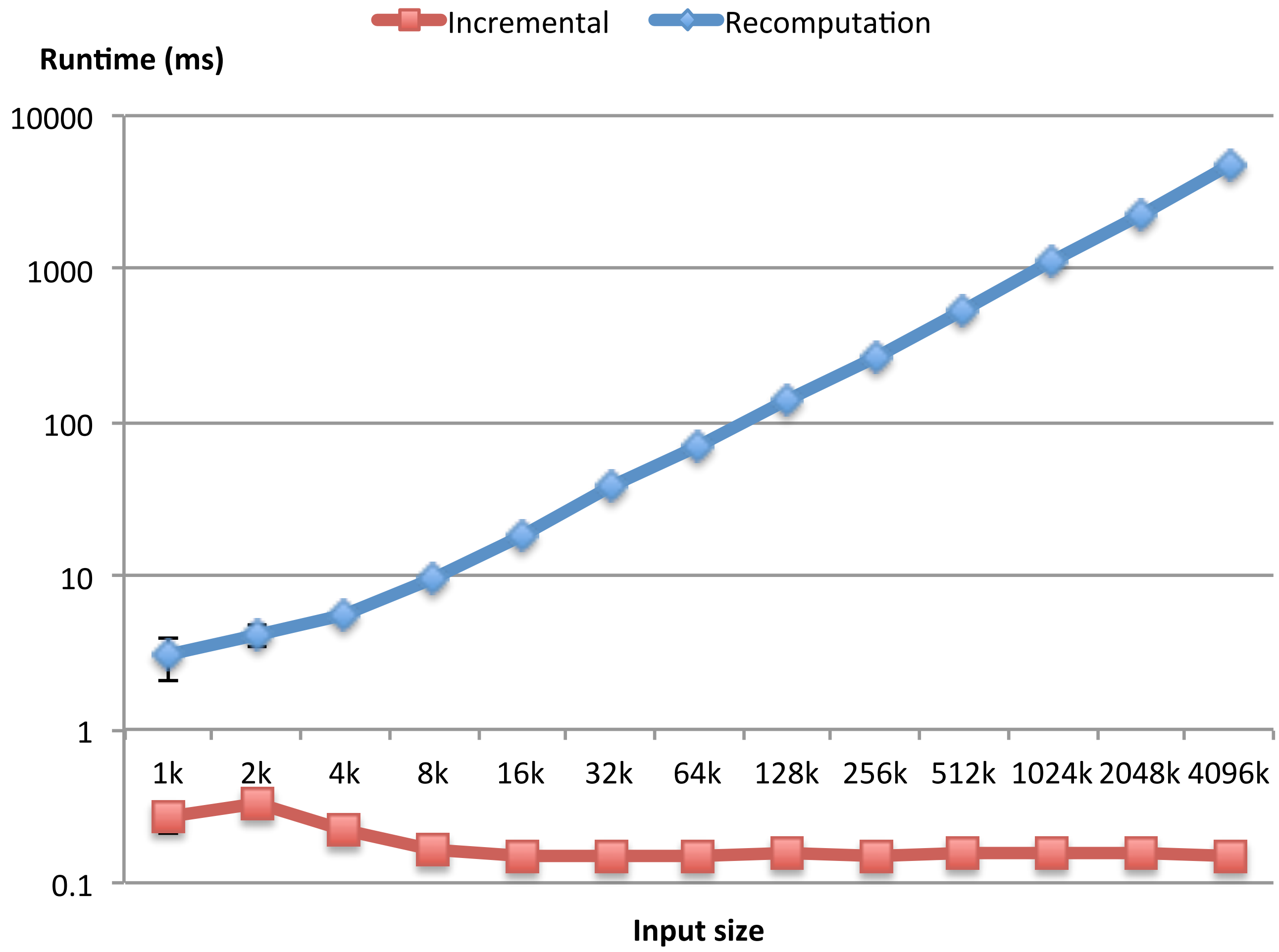
Caching intermediate results

The derivative reuses results:

```
Derive(s t) = Derive(s) t Derive(t)
```

- Term `t` was already computed! We could reuse the result, but we do not save it…
- Right now, if `t` is needed, you must recompute it.
- Up to now: focus on cases you don't need it
- Present work: reusing Liu&Teitelbaum [1995]

Performance case study (based on) MapReduce:



In the paper…


A Theory of Changes for Higher-Order Languages
Incrementalizing λ-Calculi by Static Differentiation
Yufei Cai Paolo G. Giarrusso Tillmann Rendel Klaus Ostermann
Philipps-Universität Marburg

- Change structure for first-class functions!
- A code transformation for λ-calculi
- A mechanized correctness proof (in Agda, with *denotational semantics* & *logical relations*)
- Some hints on applying ToC
- Implementation, language plugin with bags and maps, and performance case study in Scala

Conclusions

- Incremental computation can give great performance advantages
- Theory of Changes for describing incremental computation
  - maybe applicable to other approaches
- Lots of work to do
  - Lots of avenues for future work — talk to us!

Further optimizations

- Since we create an incremental program, we can optimize it!
- To avoid computing intermediate results we don't use, this time we transform abstractions to be by-name lambdas.
- We could use absence analysis in the future.
- Further transformations possible.

References

- [Liu&Teitelbaum 1995] Caching intermediate results for program improvement. PEPM 1995.
- [Koch 2010] Incremental query evaluation in a ring of databases. PODS 2010.
- [Gluche et al. 1997] Gluche, Grust, Mainberger, and Scholl. Incremental updates for materialized OQL views. In Deductive and Object-Oriented Databases, Springer.

Questions?

Static caching

Static caching

- Based on work of Liu&Teitelbaum [1995]
- Basic idea: remember and save intermediate results of all computations
- **Whenever a *computation* returns a *value***, save the value for future reuse
- Each function returns a tuple:
  - its original return value
  - all intermediate results

Static caching & CBPV

- What's the correct notion of *computation* and *value*?
  - First attempt: A-normal form
  - Is a partially applied curried function a value?
  - Should we save the result of primitives?
    - Result of pair constructors, introduction forms: not needed, because they create values
    - Result of elimination forms: needed
    - Answer: we should save the result of computations, not of values, and divide primitives accordingly

Change equivalence

Change equivalence

- A change can have different but $\triangleq$ representations, but they should not be distinguished.
- Change operations (the ones in the signature) preserve $\triangleq$.
- If a function only accesses changes via operations in the signature, it preserves $\triangleq$.
- We'll restrict attention to such functions.

Restrict attention to $\triangleq$-respecting functions

- We just restrict attention to function with "abstract enough" types
  - Change types must be abstract
- Those functions can only access changes with the change interface …
- … so those functions can't distinguish equivalent changes!

In Ocaml

```
module type Base = sig type v end;;
module type Change =
  functor (B: Base) ->
    sig
      type v = B.v
      type dv (*sealed in structures!*)
      val ⊕: v -> dv -> v
      val ⊖: v -> v -> dv
    end;;
```

In Ocaml

```
module type Change = sig
  type v (*concrete in structures*)
  type dv (*sealed in structures!*)
  val oplus: v -> dv -> v
  val ominus: v -> v -> dv
end;;
```

```
module type ChangeInt
= sig
    include Change with type v =
int
    val plusDeriv :
        v -> dv -> v -> dv -> dv
  end;;
```

In Ocaml

```
module ChangeIntStruct : ChangeInt
  = struct
    type v = int
    type dv = int (* sealed! *)
    let oplus v dv = v + dv
    let ominus v2 v1 = v2 - v1
    let plusDeriv x dx y dy = dx +
dy
  end;;
```

Conjecture on d.o.e.

“D.o.e. ($\triangleq$) implies observational equivalence.”

Open questions:

- must check that functions have “abstract enough” dependent types
- we need a proof of parametricity for the type theory we use
  - we can express the change signature with ML module system, and translate that to System Fomega through techniques by (XXX citation) *F-ing modules* paper

Understanding our changes

- $\Sigma_{x:\,V}\,(\Delta x/\triangleq) \cong V \times V$
- $(A \longrightarrow B) \times (A \longrightarrow B) \cong (A \longrightarrow B \times B) \cong A \longrightarrow \Sigma_{x:\,B}\,(\Delta x/\triangleq)$
- $A \longrightarrow \Sigma_{x:\,B}\,(\Delta x/\triangleq) \cong \{\, f : \Sigma_{x:\,A}\,(\Delta x/\triangleq) \longrightarrow \Sigma_{x:\,B}\,(\Delta x/\triangleq) \mid f$ is a valid derivative $\}$

Understanding our semantics

- $\lambda\, V.\ \Sigma_{x:\,V}(\Delta x/\triangleq) \cong \lambda\, V.\ V \times V$ monad
- Is our semantics related to "just" a standard categorical semantics in the Eilenberg-Moore category of this monad?

A categorically-inspired semantics

- Claim: it's useful to design the definition of change structures using category theory
- If we do that, we see that semantically
  - $\Sigma_{v:\,V}\,(\Delta v/\triangleq) \cong \mathrm{V} \times \mathrm{V}$

New slides

XXX

- Add extension of ToC to programs through denotational semantics?
- Or just add proof strategy?
- Relate erasure to realizability!

Change equality: multiple representations

A change can have multiple $\triangleq$ representations, but they should not be distinguished.

Semantic functions should respect $\triangleq$; that's guaranteed if they only use the change signature.

Change equivalence (conjecture)

- Thanks to parametricity for abstract types, clients of `Change` can't observe the difference between d.o.e. changes, so d.o.e. changes are ***observationally equivalent***!
- We conjecture that all programs we want are valid clients of `Change` & c. (we just didn't check yet).
- We need parametricity for the right language — we conjecture *F-ing modules* is enough.

Warning

- This presentation (and the paper) uses set theory for "simplicity"
- In fact, our Agda formalization uses *type theory*!
- Δv is a dependent type of changes!
- Δv_1 and Δv_2 are disjoint iff $v_1 \neq v_2$
- (XXX This is needed for the categorical semantics)

- Changes DT for a type T have:
- a source of type T
- a destination of type T
- We have functions from
- (These aren't necessarily computable)