

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

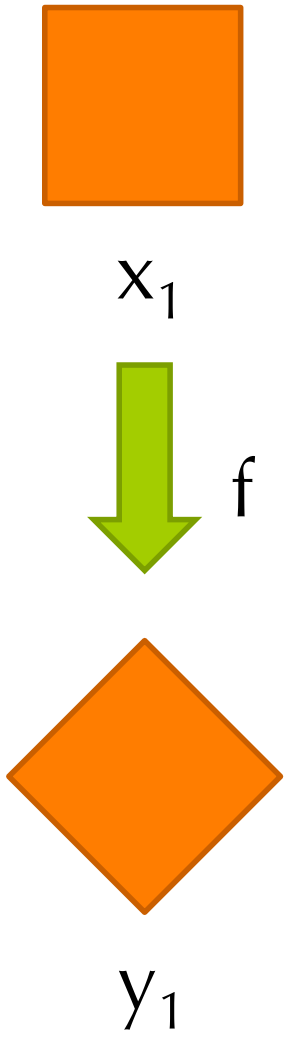


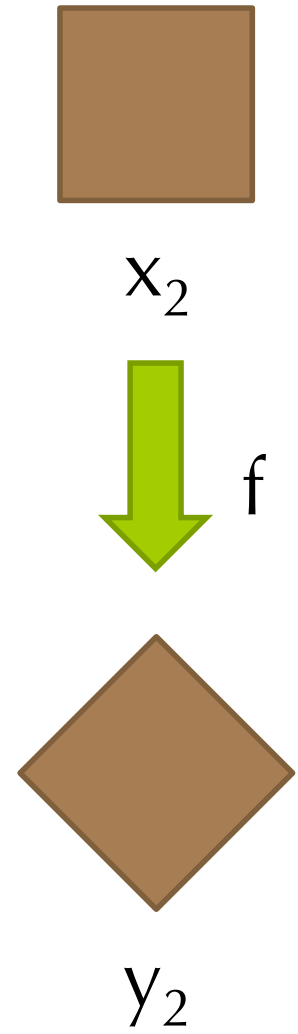
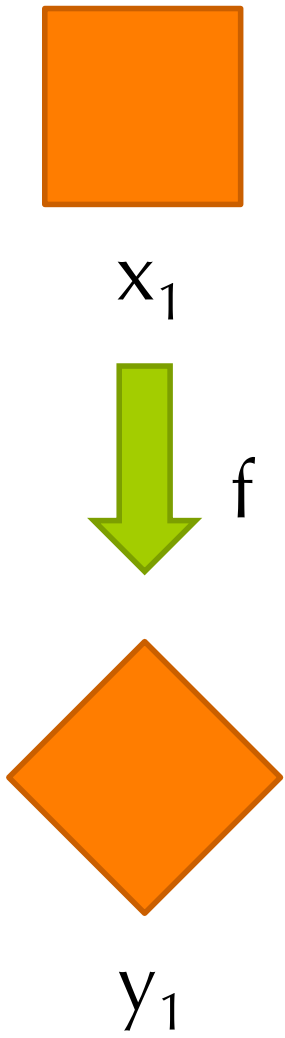
Incrementalizing λ -Calculi by Static Differentiation

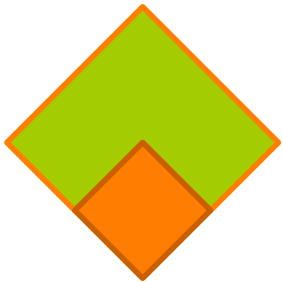
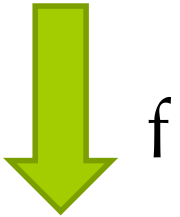
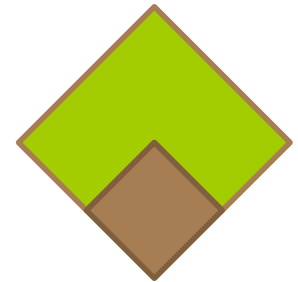
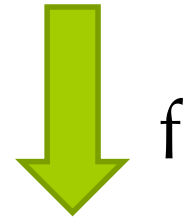


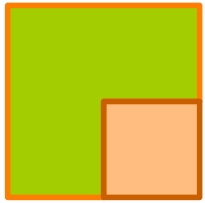
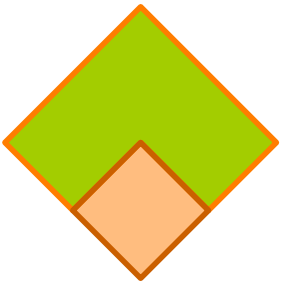
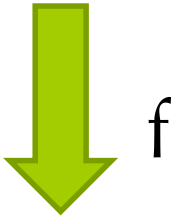
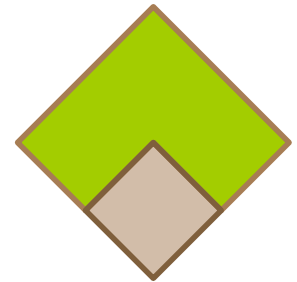
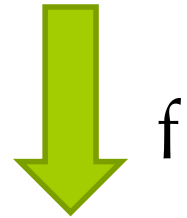
Problem: Incremental computation

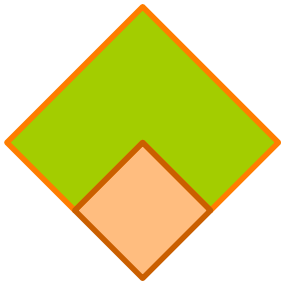
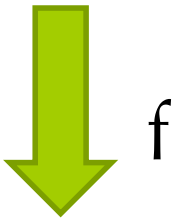
- ✓ Support for a language with first-class functions!
- ✓ Mechanized proof in Agda
- ✓ Implementation in Scala
- ✓ Performance case-study



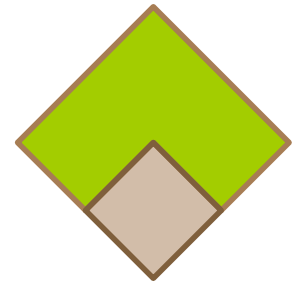


 X_1  Y_1  X_2  Y_2

 x_1  y_1  x_2  y_2

 x_1  y_1

**f invoked
again! ☹️**

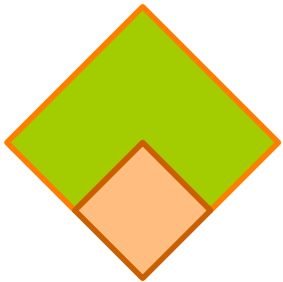
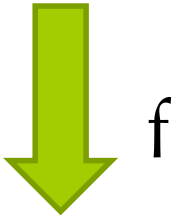
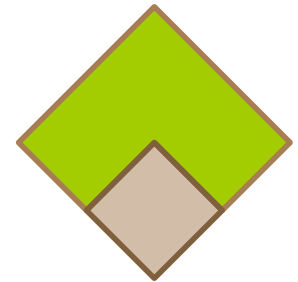
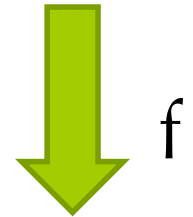
 x_2  y_2

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  Y_1  X_2  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

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

base input

$y_1 = 1 + 1 + 2 + 3 + 4 = 11$

base output

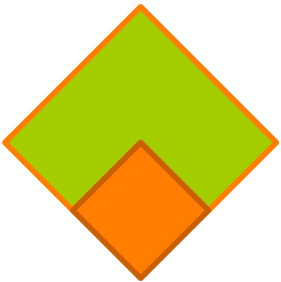
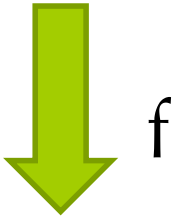
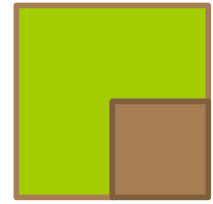
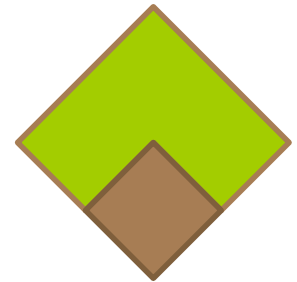
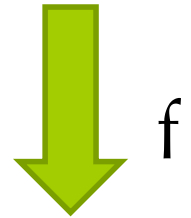
$x_2 = \{\{1, 2, 3, 4, 5\}\}$

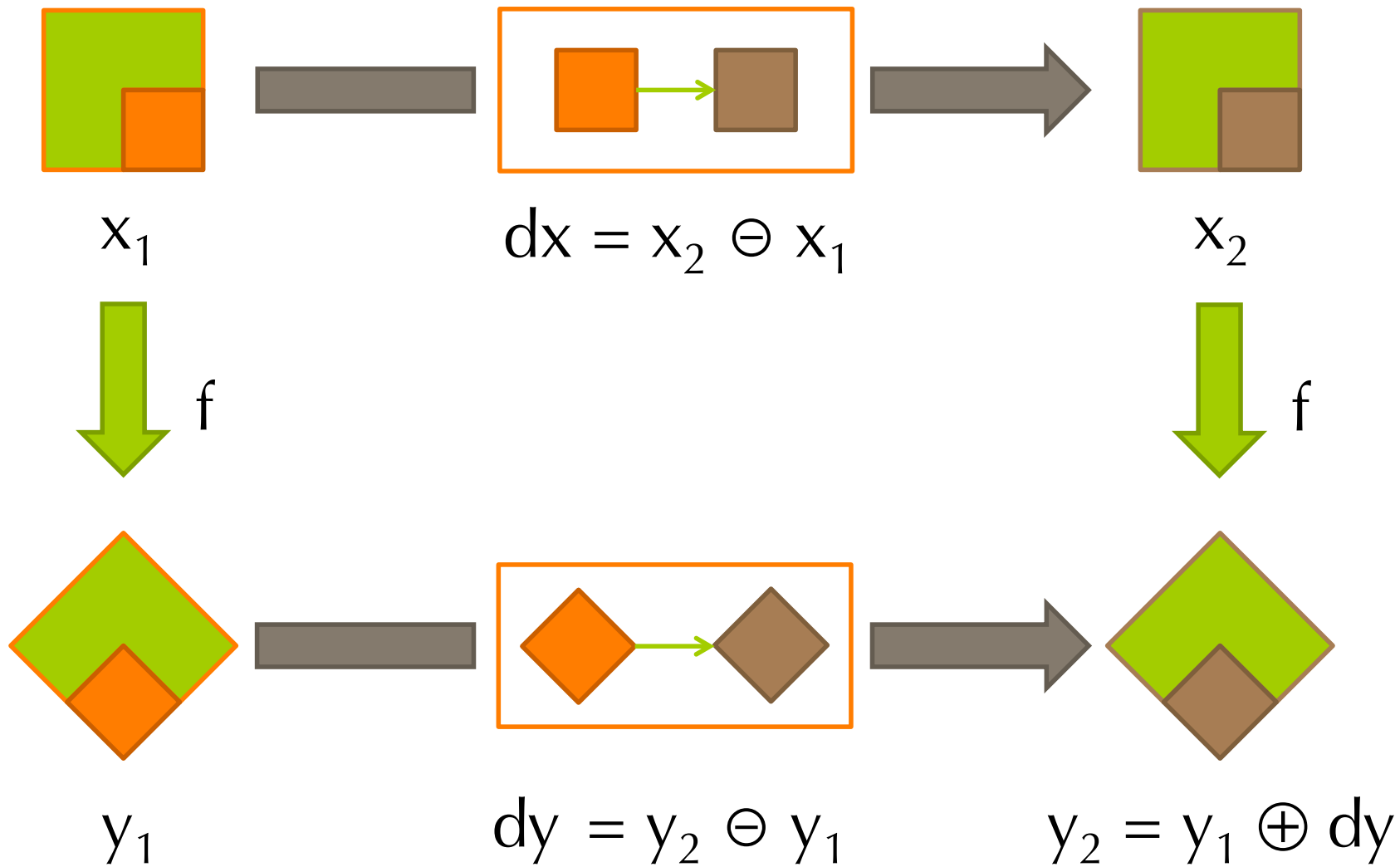
upd. input

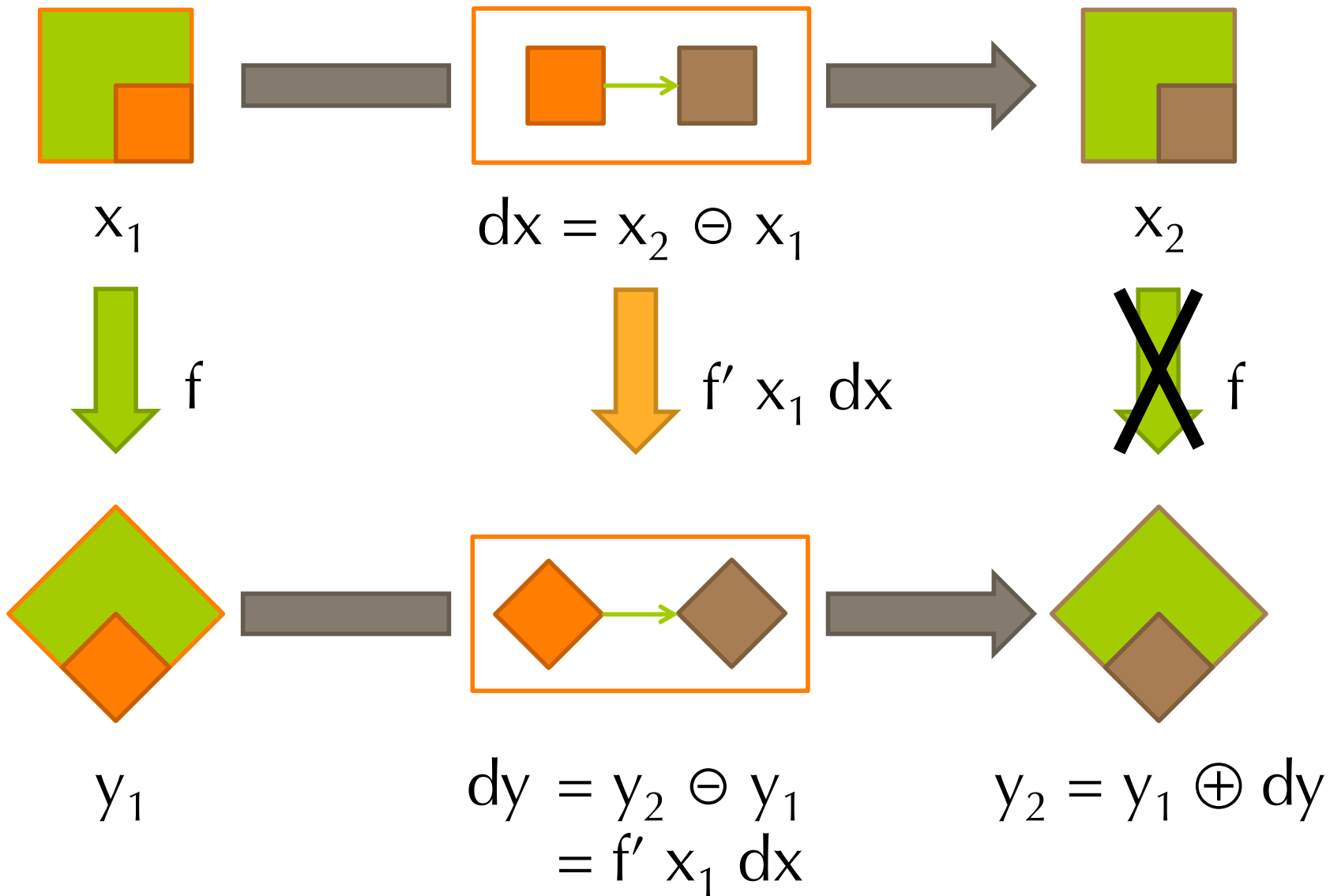
$y_2 = 1 + 2 + 3 + 4 + 5$

upd. output

$= 15 = s_1 - 1 + 5$

 X_1  Y_1  X_2  Y_2





Example

$$f \text{ coll} = \text{fold } (+) \ 0 \ \text{coll}$$

$$y = f \ x$$

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

$$y_1 = 11$$

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

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

$$\begin{aligned} y_2 &= x_1 \oplus f' \ x_1 \ dx = 11 \oplus (-1 + 5) \\ &= 15 \end{aligned}$$

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) \rightarrow 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 $\forall x : V$ *change types*

Operations

- (C3) $x_1 \oplus dx : V$ $\forall dx : \Delta x_1$ *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*:

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

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

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

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

Example derivatives

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

$$\text{id } x = x$$

$$\text{id}' x dx = dx$$

$$f x = x + 5$$

$$f' x dx = dx$$

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:
 $\{\{1,2,3,4,5\}\} \ominus \{\{1,1,2,3,4\}\}$ 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.

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

$\{\{5, \underline{1}\}\} \triangleq$ “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 \triangle changes) going from a to b

Derived ops give a category

Derived ops

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

$$dx_1 \odot dx_2 = (x_1 \oplus dx_1) \oplus dx_2 \ominus x_1 \quad \textit{change composition}$$

Derived algebraic equations

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

$$dx \odot \mathbf{0} \triangleq \mathbf{0} \odot dx \triangleq dx \quad \textit{composition unit}$$

$$(dx_1 \odot dx_2) \odot dx_3 \triangleq dx_1 \odot (dx_2 \odot dx_3) \quad \textit{composition associativity}$$

(Static) Differentiation

- Given a (simply-typed) λ -term f :

Derive

Program f



Derivative f'

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

$$\llbracket f (a \oplus da) \rrbracket = \llbracket f a \oplus \mathbf{Derive}(f) a da \rrbracket$$

“Derivatives” are non-linear!

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

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.

- $\llbracket f(a \oplus da) \rrbracket = \llbracket f a \oplus f' a da \rrbracket$
- $\llbracket f(a \oplus da \oplus db) \rrbracket = \llbracket f a \oplus f' a (da \odot db) \rrbracket$
- $\llbracket f a \oplus f' a (da \odot db) \rrbracket = \llbracket f(a \oplus da \oplus db) \rrbracket = \llbracket f a \oplus f' a da \oplus f' (a \oplus da) db \rrbracket$

Derivative examples #1

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

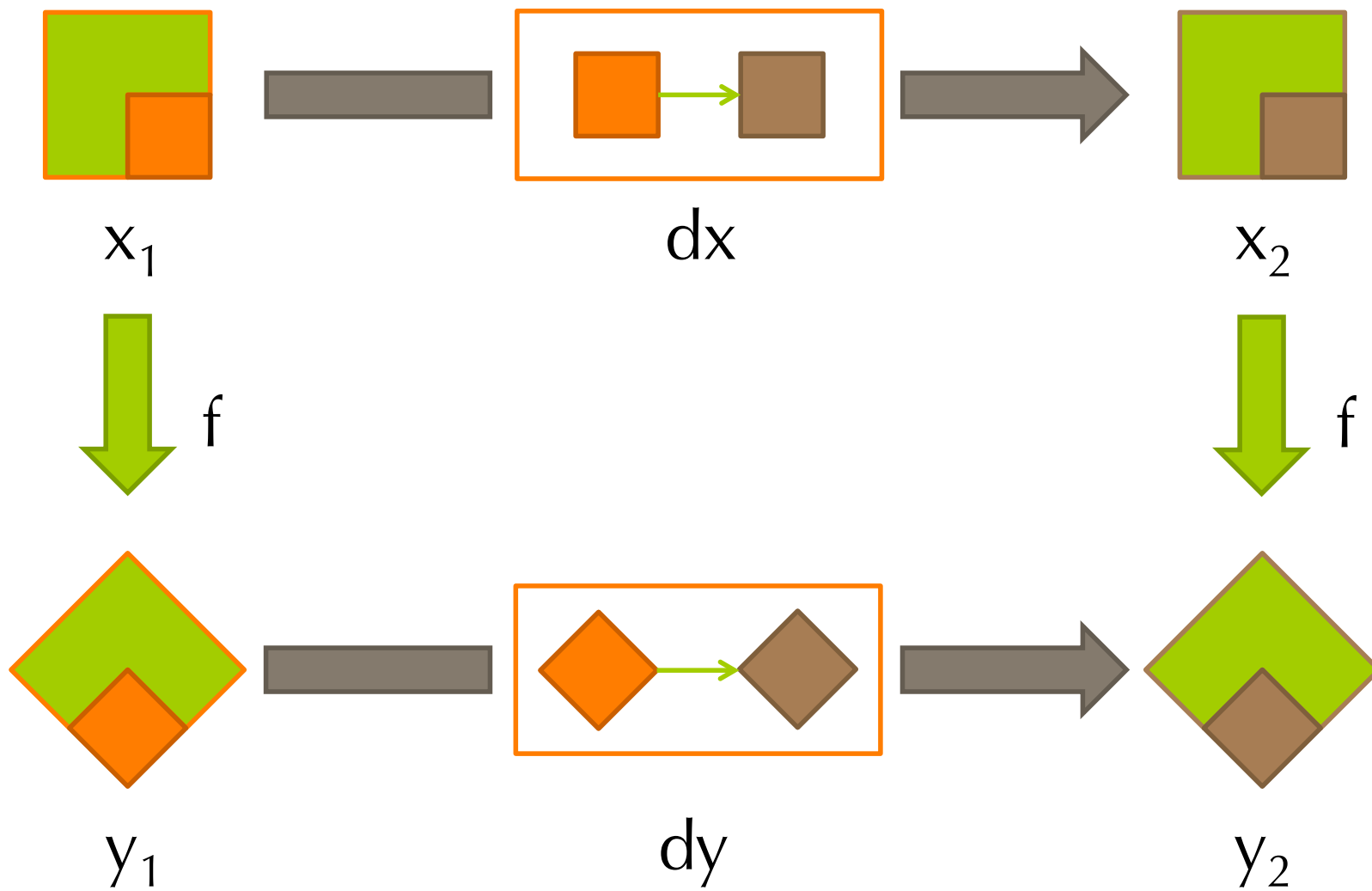
$$\text{id}_T' = \mathbf{Derive}(\text{id}_T) = \lambda (x : T) (dx : \Delta T). dx$$

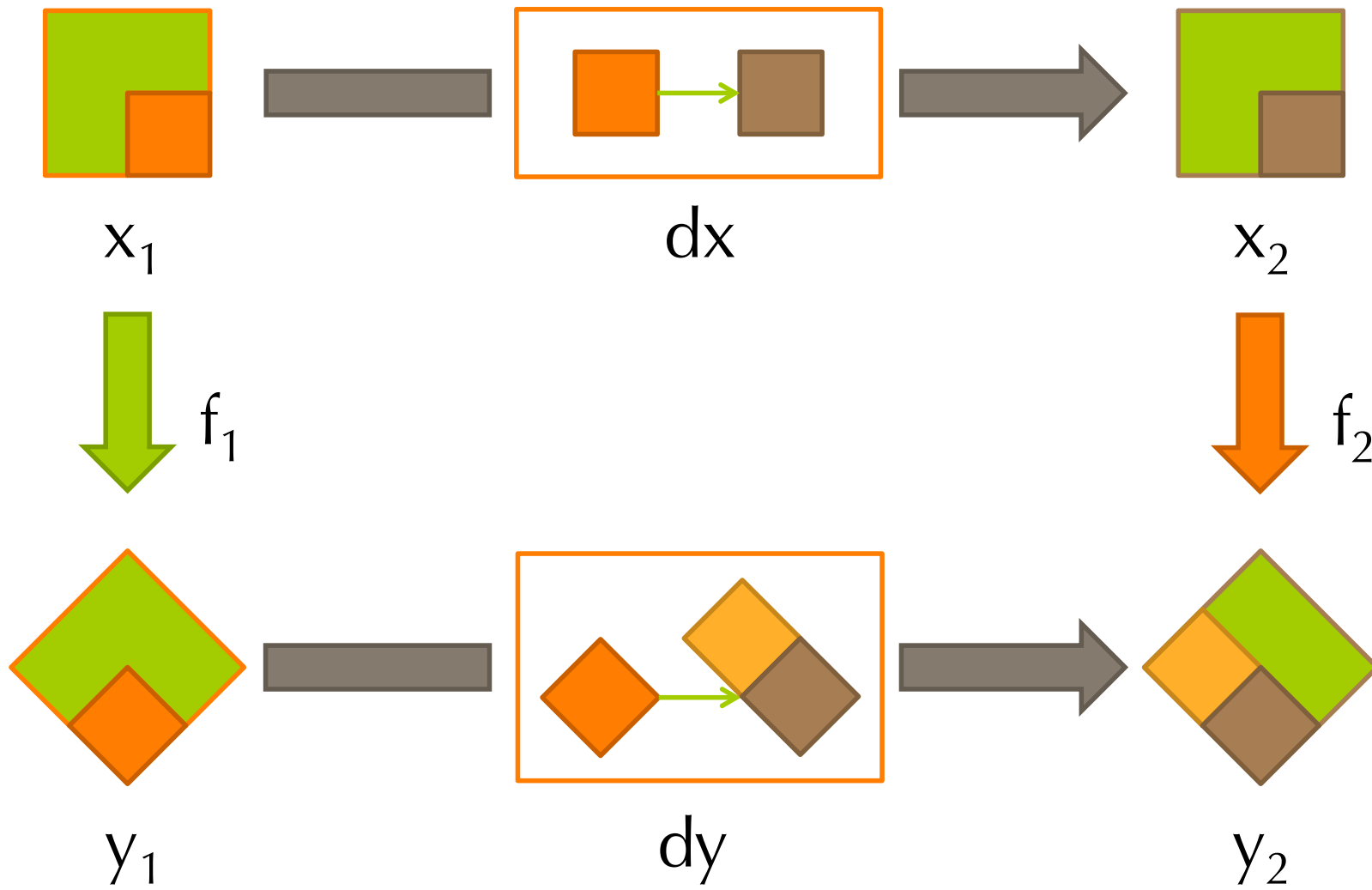
- ΔT , not Δx
 - no dependent types
- ΔT is expanded by **Derive**
- changes (dx) are first-class

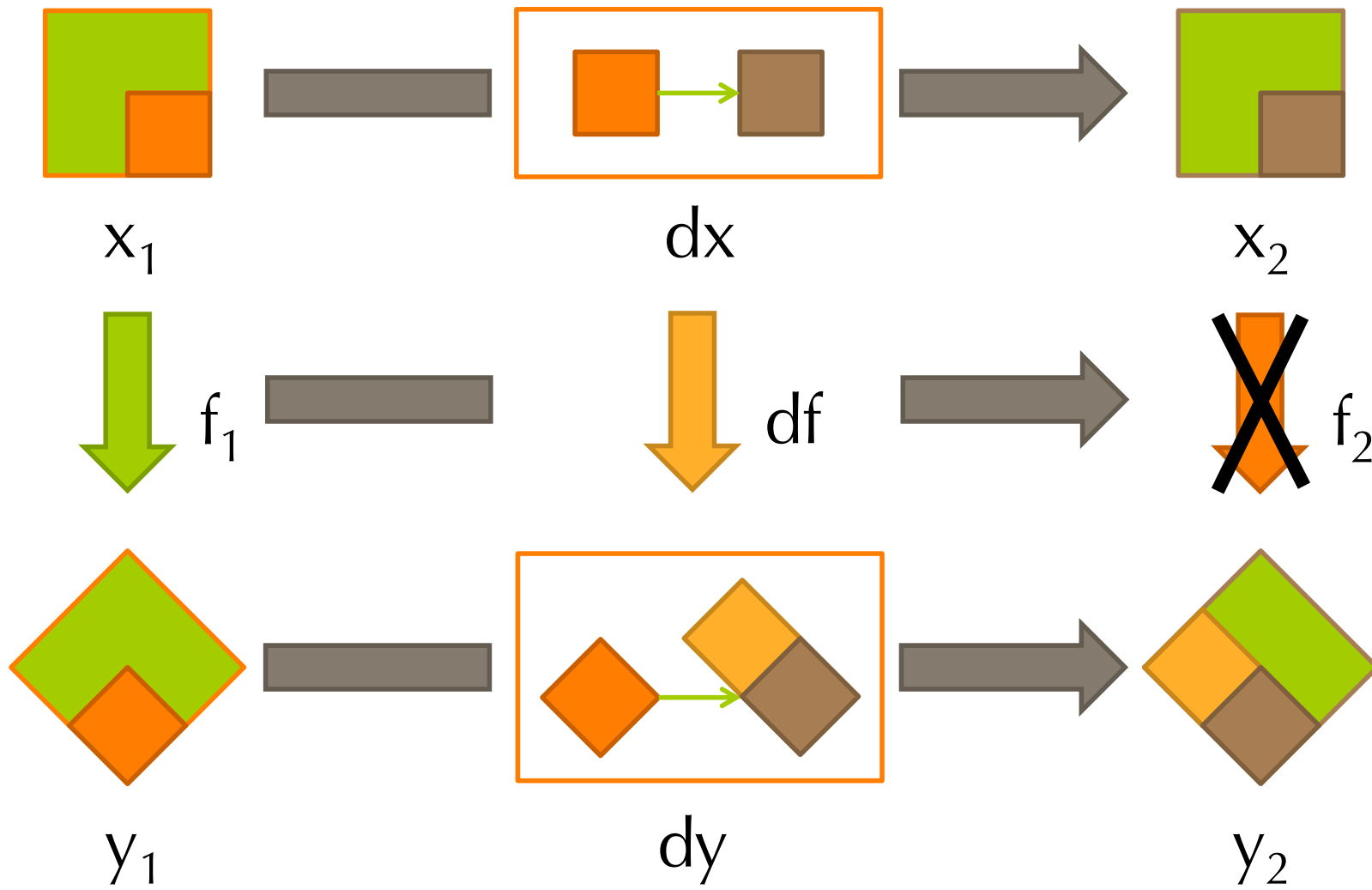
First-class functions

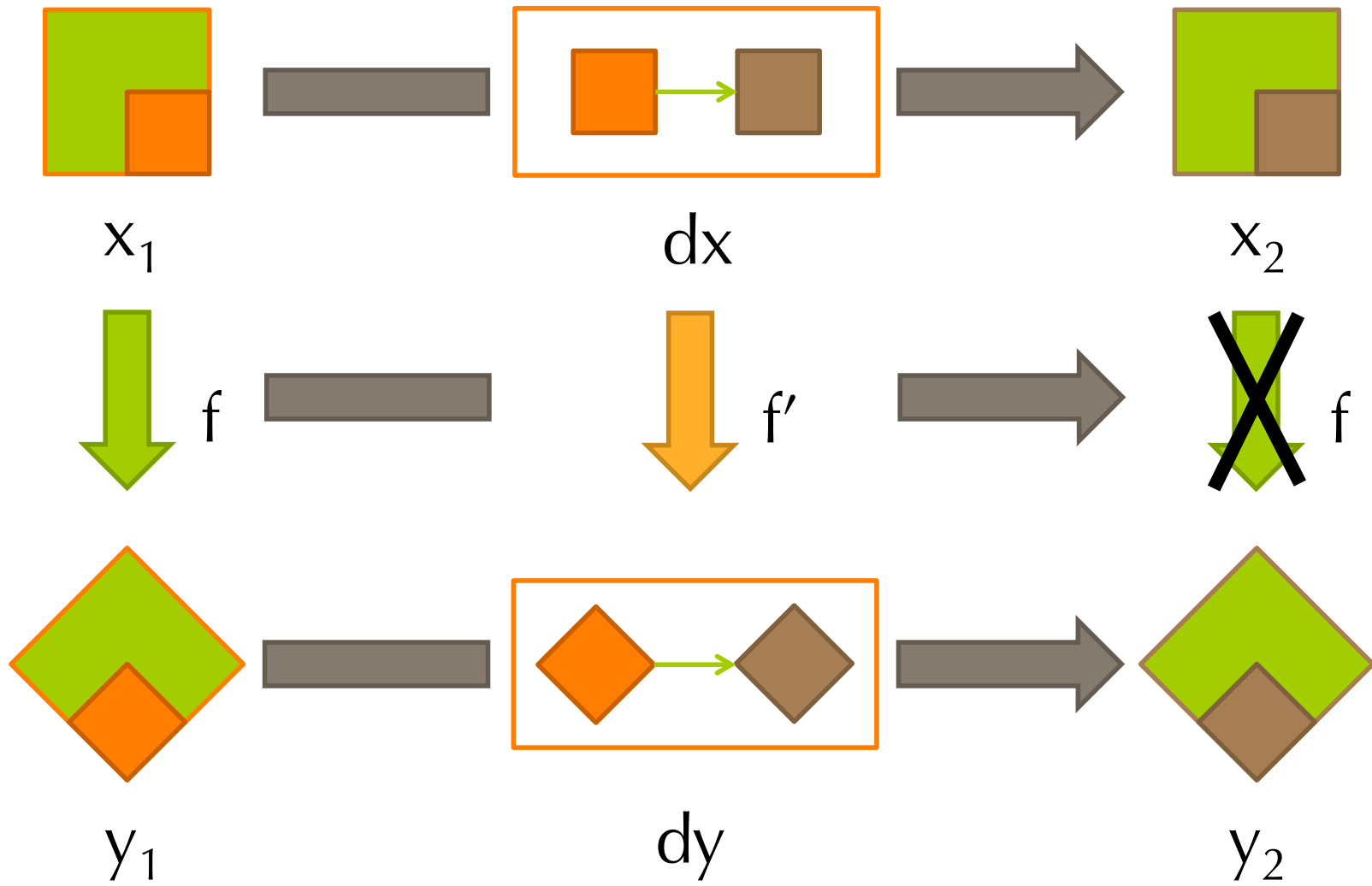
First-class functions

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









Derivatives → function changes

From:

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

to:

$$df_{x_1} dx = f_2_{x_2} \ominus f_1_{x_1} = y_2 \ominus y_1 = dy$$

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

Change structure for functions

$$\Delta_{\sigma \rightarrow \tau} = \lambda (f: \llbracket \sigma \rightarrow \tau \rrbracket) \rightarrow$$

$$\sum_{df : \forall (x : \llbracket \sigma \rrbracket) (dx : \Delta x) \rightarrow \Delta (f x)} \textit{valid} (f, df)$$

Derivative examples #2

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

$$\text{id}_T' = \mathbf{Derive}(\text{id}_T) = \lambda (x : T) (dx : \Delta T). dx$$

$$\text{app}_{TU} = \lambda (f : T \rightarrow U) (x : T). f x$$

$$\begin{aligned} \text{app}_{TU}' &= \mathbf{Derive}(\text{app}_{TU}) = \\ &\lambda (f : T \rightarrow U) (df : \Delta(T \rightarrow U)) \\ &\quad (x : T) (dx : \Delta T). df x dx \end{aligned}$$

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

Language

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

$$t ::=$$

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

Base types and constants specified by a language plugin.

Deriving terms

We require that `Derive` satisfies admissible rule:

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

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

$$\Delta\iota = \dots$$

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

Deriving terms

Propagate changes:

$$\text{Derive}(s \ t) = \text{Derive}(s) \ t \ \text{Derive}(t)$$
$$\text{Derive}(\lambda x. \ t) = \lambda x \ dx. \ \text{Derive}(t)$$

Return changes:

$$\text{Derive}(x) = dx$$

Change of primitives:

$$\text{Derive}(c) = dc$$

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 \Rightarrow few primitives (e.g. folds) required, other ops (e.g. map) in libraries
- Primitives encapsulate efficiently incrementalizable skeletons

Example

$f \text{ coll} = \text{fold } (+) \ 0 \ \text{coll}$

$y = f \ x$

$\text{coll}_0 = \{\{1, 1, 2, 3, 4\}\}$

$\text{coll}_1 = \{\{1, 2, 3, 4, 5\}\}$

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

What about the removal of 1?

Example

$\text{sum } s = \text{fold } (+) \ 0 \ s$

$y = \text{sum coll}$

$\text{dsum } s \ ds = \dots = \text{fold } (+) \ 0 \ ds$

$dy = \text{dsum coll dcoll}$

$\text{coll}_0 = \{\{1,2,3,4\}\}$

$\text{coll}_1 = \{\{2,3,4,5\}\}$

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

Running example & primitives

$f \text{ coll} = \text{fold } (+) \ 0 \ \text{coll}$

$y = f \ x$

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

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

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

What about 1, i.e. the removal of 1?

Running example & primitives

f coll = fold **G** coll **G** abelian group!

y = f x

x₁ = {{**1**, 1, 2, 3, 4}}

x₂ = {{1, 2, 3, 4, **5**}}

dx = {{1, 2, 3, 4, **5**}} ⊖ {{**1**, 1, 2, 3, 4}} = {{5, 1}}

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

df x₁ dx = fold' **G** d**G** x₁ dx = ... = fold **G** dx = 4

dy = df x₁ dx

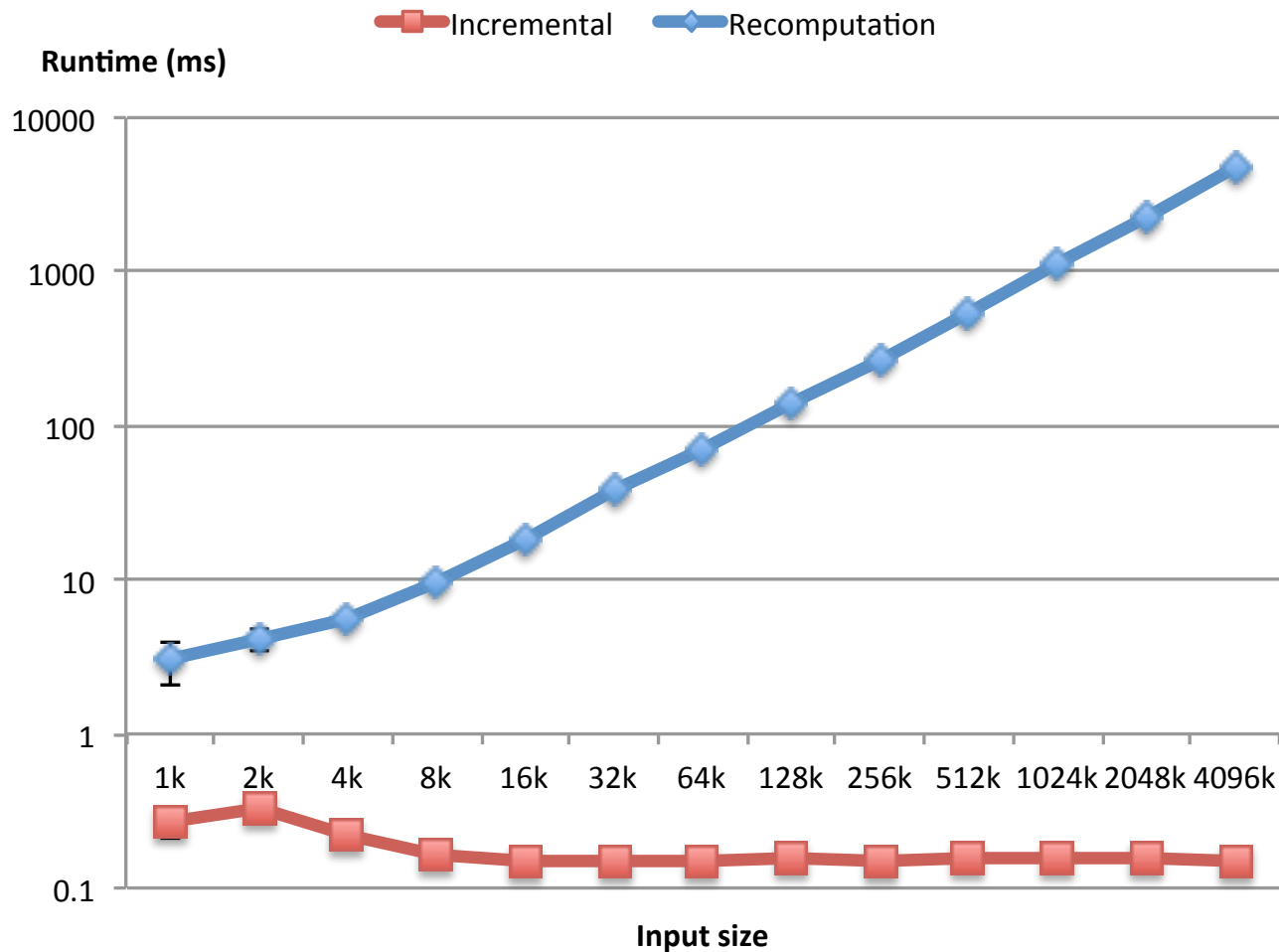
Caching intermediate results

The derivative reuses results:

$$\text{Derive}(s \ t) = \text{Derive}(s) \ t \ \text{Derive}(t)$$

- 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...

- 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

A Theory of Changes for Higher-Order Languages

Incremental λ -Calculus by Static Differentiation

Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, Klaus Ostermann
Philippe-Université Marburg



Abstract

In the context of an expressive computation modelled by a small change to the input, the old result should be updated incrementally instead of recomputing the whole computation. We formalise this intuition through their denotations. A denotational model changes the program's input directly to changes in the program's output. We present a transformation taking programs in this denotation, which is fully static and automatic, requires function functions, and produces denotations amenable to standard optimisations.

We prove the program transformation correct in Agda for a small set of simple typed λ -calculi, generalised to both types and patterns. A generic analysis specifies what is required to mechanise the chosen properties.

We investigate performance by a case study. We implement in Scala the program transformation, a plugin and improve performance of a denotational program by means of optimisation.

Keywords: Incremental computation, first-class functions, pattern matching, Agda, formalisation.

Categories and Subject Descriptors: D.3.1 [Programming Techniques]: Application Formalisms; D.3.2 [Programming Techniques]: Language Constructs and Features; D.3.4 [Programming Techniques]: Processors—Optimizers.

1. Introduction

Incremental computation has a long-standing history in computer science [21]. Often, a program needs to update its output efficiently in response to changes [22]. Instead of re-running such a program from scratch to its original input, incremental computation reuses what the input change is much smaller than the input itself.

For instance, consider the `sum_of_squares` program, which calculates the sum of all numbers in collection `xs`.

```
sum_of_squares :: Int -> Int
sum_of_squares xs = sum [x2 | x <- xs]
sum_of_squares [1..10] = 385
```

With `[1..10]` the program is evaluated to `385`, but, if we use an external collection (like a set) where elements are allowed to be

added or removed, we can change the input to `[1..11]` and get a change from `[2..5, 10]` or `[1..2, 3, 5, 10]`. Instead of recomputing `sum_of_squares` from scratch, we could compute it incrementally. For example, a representation of the change to the system may be `[[sums [1], sum [2], sum [3], sum [5], sum [10]]]`, we can compute the new result through a function `sum_of_squares'` that takes the old input `xs :: [1..10]`, `ys :: [2..5, 10]` and the change `ds` due to previous input changes. Then, it would compute the change `sum_of_squares' xs ds` as `385 + 121` to yield the updated result 506. We call `sum_of_squares'` the structure of `sum_of_squares` in a function in the same language as `sum_of_squares` and produce changes, which are equal to the value of the language. This requires some changes to the original input `xs` and the new computation of `sum_of_squares'` is no longer linearly with the size complexity of `sum_of_squares` as it can be only depend on the size of `ds` and `ys`, which is smaller both in our example and in general.

To support incremental computation, this paper introduces an automatic program transformation that differentiates programs that compute their denotations. These programs are:

$$f \text{ (in } ds) \Rightarrow f \text{ (in } xs) \oplus (\text{change } f \text{ in } ds) \quad (1)$$

where \oplus is denotational equality, ds is a change on `xs` and ds is denoted a update with change `ds`, that is, the updated input of `f`. Hence, we can separate programs by updating the old input `xs`, which computes the output from scratch, with the right hand side, which computes the output incrementally using derivatives.

It is based on a simple typed λ -calculus parameterised by effects. A first derivative is the type and pattern generation, and a change computation is the type and pattern generation for each position. In other words, the plugin specifies the generation and first derivative derivatives, and it can be updated these simple derivatives in such a way that derivatives for arbitrary simple typed λ -calculus expressions exist and derivatives can be computed. Both the implementation and our contributions can be generalised to the plugin, hence it is easy to support (and prove) arbitrary new effects.

This paper makes the following contributions:

- We present a mechanised theory of changes and derivatives, which is more general than other work in the field because changes are first-class entities. They are defined in the denotation and they are defined also for functions (Sec. 2).
- We present the first approach to incremental computation for pattern matching. Previous work generalises the denotational model to the higher-order case, but it does not support pattern matching. Our approach requires no new type support. The transformation produces an incremental program in the same language of operations. Techniques for the original program are applicable to the incremental program as well. We prove that the incrementalizing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted by ACM, provided that the copies are not made for general distribution, for advertising or promotional purposes, for creating new collective works, or for resale. Copyright for this article is owned by ACM. Copyright for the text of this article is owned by the author(s). This article is intended to be read online in the ACM Digital Library. Request permission to publish in this journal from the ACM Press. For more information on other permissions, contact the ACM Press. ACM Press, 475 Rte. 9W, Reading, MA 01961-1000, USA. Copyright 2016 ACM 978-1-4503-4383-1/16/0001...\$15.00. ACM is not responsible for the content of the articles or for any consequences arising from the use of the information contained in these articles.

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 \triangle - 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  $\oplus$ : v -> dv -> v
      val  $\ominus$ : 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

- $\Sigma_{x:V}(\Delta x/\triangle) \cong V \times V$
- $(A \rightarrow B) \times (A \rightarrow B) \cong (A \rightarrow B \times B) \cong A \rightarrow \Sigma_{x:B}(\Delta x/\triangle)$
- $A \rightarrow \Sigma_{x:B}(\Delta x/\triangle) \cong \{ f : \Sigma_{x:A}(\Delta x/\triangle) \rightarrow \Sigma_{x:B}(\Delta x/\triangle) \mid f \text{ is a valid derivative} \}$

Understanding our semantics

- $\lambda V. \Sigma_{x:V} (\Delta x / \triangle) \cong \lambda V. V \times V$ 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
 - $\Sigma_{V:V} (\Delta V / \cong) \cong V \times V$

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)