A Theory of Changes for Higher-Order Languages

Incrementalizing λ-Calculi by Static Differentiation

Abstract

If the result of an expensive computation is invalidated by a small change to the input, the old result should be updated incrementally instead of reexecuting the whole computation. We incrementalize programs through their *derivative*. A derivative maps changes in the program's input directly to changes in the program's output, without reexecuting the original program. We present a program transformation taking programs to their derivatives, which is fully static and automatic, supports first-class functions, and produces derivatives amenable to standard optimization.

We prove the program transformation correct in Agda for a family of simply-typed λ-calculi, parameterized by base types and primitives. A precise interface specifies what is required to incrementalize the chosen primitives.

We investigate performance by a case study: We implement in Scala the program transformation, a plugin and improve performance of a nontrivial program by orders of magnitude.

Keywords Incremental computation, first-class functions, performance, Agda, formalization

1. Introduction

Incremental computation has a long-standing history in computer science [20]. Often, a program needs to update its output efficiently to reflect input changes [22]. Instead of rerunning such a programs from scratch on its updated input, incremental computation research looks for alternatives that are cheaper in a common scenario: namely, when the input change is much smaller than the input itself.

For instance, consider the following program which adds all members of a collection s of numbers.

$$sum\ s = fold\ (+)\ 0\ s$$
$$y = sum\ \{1, 2, 3, 4\}$$

Now assume that the input to sum changes from $\{1, 2, 3, 4\}$ to $\{2, 3, 4, 5\}$. Instead of recomputing y from scratch, we could also compute it incrementally. If we have a representation for the change to the input (say, $ds = \{\textbf{remove}\ 1, \textbf{add}\ 5\}$), we can compute the new result through a function sum' that takes the old input $s = \{1, 2, 3, 4\}$ and the change ds and produces a change dy to the output y. In this case, it would compute the change $dy = sum'\ s\ ds = \textbf{plus}\ 4$, which can then be used to update the original output $y = 10$ to yield the updated result 14. We call sum' the *derivative* of sum. It is a function in the same language of sum, accepting and producing changes, which are simple first-class values of this language. If we increase the size of the original input s, the complexity of $sum\ s$ increases linearly, while the complexity of $sum'\ s\ ds$ only depends on the size of ds, which is smaller both in our example and typically.

To address this problem, in this paper we introduce the IλC (incrementalizing λ-calculi) framework. We define an automatic program transformation *Derive* that *differentiates* programs, that is, computes their derivatives; *Derive* guarantees that

$$f\ (a \oplus da) \cong (f\ a) \oplus (\mathit{Derive}(f)\ a\ da)\,. \tag{1}$$

where $\cong$ is denotational equality, da is a change on a and $a \oplus da$ denotes a updated with change da, that is, the updated input of f. Hence, we can optimize programs by replacing the left-hand side, which recomputes the output from scratch, with the right-hand side, which computes the output incrementally using derivatives.

IλC is based on a simply-typed λ-calculus parameterized by *plugins*. A plugin defines (a) base types and primitive operations, and (b) a change representation for each base type, and an incremental version for each primitive. In other words, the plugin specifies the primitives and their respective derivatives, and IλC can glue together these simple derivatives in such a way that derivatives for arbitrary simply-typed λ-calculus expressions using these primitives can be computed. Both our implementation and our correctness proof is parametric in the plugins, hence it is easy to support (and prove correct) new plugins.

This paper makes the following contributions:

- We present a novel mathematical theory of changes and derivatives, which is more general than other work in the field because changes are first-class entities, they are distinct from base values and they are defined also for functions (Sec. 2).
- We present the first approach to incremental computation for pure λ-calculi by a source-to-source transformation, *Derive*, that requires no run-time support. The transformation produces an incremental program in the same language; all optimization techniques for the original program are applicable to the incremental program as well. We prove that our incrementalizing transformation *Derive* is correct (Eq. (1)) by a machine-checked formalization in Agda [6]. The proof gives insight into the definition of *Derive*: we first construct the derivative $[\![-]\!]^{\Delta}$ of the denotational semantics of a simply-typed λ-calculus term, that is, its *change semantics*. Then, we show that *Derive* is produced by erasing $[\![-]\!]^{\Delta}$ to a simply-typed program (Sec. 3).
- While we focus mainly on the theory of changes and derivatives, we also provide an initial experimental evaluation. We implement the derivation transformation in Scala. The implementation is organized as a plug-in architecture that can be extended with new base types and primitives. We define a plugin with support for different collection types and use the plugin to incrementalize a variant of the MapReduce programming model [15].

Benchmarks show that incrementalization can reduce asymptotic complexity and can turn $O(n)$ performance into $O(1)$, improving running time by over 4 orders of magnitude (Sec. 4).

Our Agda formalization, Scala implementation and benchmark results are available at the URL `https://www.dropbox.com/sh/3vg8pikd6wbgck5/SaZPRvqB2p`. All lemmas and theorems presented in this paper have been proven in Agda. In the paper, we present an overview of the formalization in more human-readable form, glossing over some technical details.

2. A theory of changes

This section introduces a formal concept of changes; this concept was already used informally in Eq. (1) and is central to our approach. We first define change structures formally, then construct change structures for functions between change structures, and conclude with a theorem that relates function changes to derivatives.

2.1 Change structures

Consider a set of values, for instance the set of natural numbers $\mathbb{N}$. A change dv for $v \in \mathbb{N}$ should describe the difference between v and another natural $v_{\text{new}} \in \mathbb{N}$. We do not define changes directly, but we specify operations which must be defined on them. They are:

- We can *update* a base value v with a change dv to obtain an updated or *new* value v_{new}. We write $v_{\text{new}} = v \oplus dv$.
- We can compute a change between two arbitrary values v_{old} and v_{new} of the set we are considering. We write $dv = v_{\text{new}} \ominus v_{\text{old}}$.

For naturals, it is usual to describe changes using standard subtraction and addition. That is, for naturals we can define $v \oplus dv = v + dv$ and $v_{\text{new}} \ominus v_{\text{old}} = v_{\text{new}} - v_{\text{old}}$. To ensure that $\oplus$ and $\ominus$ are always defined, we need to define the set of changes carefully. $\mathbb{N}$ is too small, because subtraction does not always produce a natural; the set of integers $\mathbb{Z}$ is instead too big, since adding a natural and an integer does not always produce a natural. In fact, we cannot use the same set of all changes for all naturals. Hence we must adjust the requirements: for each base value v we introduce a set Δv of changes for v, and require $v_{\text{new}} \ominus v_{\text{old}}$ to produce values in Δv_{old}, and $v \oplus dv$ to be defined for dv in Δv. For natural v, we set $\Delta v = \{dv \mid v + dv \geq 0\}$; $\ominus$ and $\oplus$ are then always defined.

The following definition sums up the discussion so far:

Definition 2.1 (Change structures)**.** A quadruple $\widehat{V} = (V, \Delta, \oplus, \ominus)$ is a *change structure* (for V) if the following holds.

(a) V is a set.
(b) Given $v \in V$, Δv is a set, called the *change set*.
(c) Given $v \in V$ and $dv \in \Delta v$, $v \oplus dv \in V$.
(d) Given $u, v \in V$, $u \ominus v \in \Delta v$.
(e) Given $u, v \in V$, $v \oplus (u \ominus v)$ equals u. □

We overload operators Δ, $\ominus$ and $\oplus$ to refer to the corresponding operations of different change structures; we will subscript these symbols when needed to prevent ambiguity. For any $\widehat{S}$, we write S for its first component, as above.

One might expect a further assumption that $(v \oplus dv) \ominus v = dv$. While it does hold for the change structure of $\mathbb{N}$, it is not needed in general. This means that multiple changes can represent the difference between the same two base values. Throughout our theory, we only discuss equality of base values, not of changes.

Examples. One way to define change structures is from abelian groups. In algebra, an abelian group is a quadruple $(G, \boxplus, \boxminus, e)$, where $\boxplus$ is a commutative and associative binary operation, e is its identity element, and $\boxminus$ produces inverses of elements g of G, such that $(\boxminus g) \boxplus g = g \boxplus (\boxminus g) = e$. For instance, integers, unlike naturals, form the abelian group $(\mathbb{Z}, +, -, 0)$ (where $-$ represents the unary minus). Each abelian group $(G, \boxplus, \boxminus, e)$ induces a change structure, namely $(G, \lambda g.\ G, \boxplus, \lambda g\ h.\ g \boxplus (\boxminus h))$, where the change set for any $g \in G$ is the whole G. Change structures are more general, though, as the example with natural numbers illustrates.

The abelian group on integers induces also a change structure on integers, namely $\widehat{\mathbb{Z}} = (\mathbb{Z}, (\lambda v.\ \mathbb{Z}), +, -)$, where $\ominus$ and $\oplus$ have the same definitions as for naturals.

Another useful example is the definition of an abelian group (and the induced change structure) on bags with signed multiplicities [14]. These are unordered collections where each element can appear an integer number of times. Element can appear a negative number $-n$ of times in a bag change to represent n removals of that element. If $\emptyset$ represents the empty bag, $union$ performs bag union, and $negate$ negates the multiplicities of elements, we can define the abelian group $(\mathbf{Bag}\ \iota, union, negate, \emptyset)$, which induces the change structure $\widehat{\mathbf{Bag}\ \iota} = (\mathbf{Bag}\ \iota, (\lambda v.\ \mathbf{Bag}\ \iota), union, \lambda x\ y.\ union\ x\ (negate\ y))$.

Nil changes and derivatives. A particularly important change is the *nil change* of a value:

Definition 2.2 (Nil change)**.** Given a change structure $\widehat{V}$ and a value $v \in V$, the change $v \ominus v$ is the nil change for v.

$$\mathbf{0}_v = v \ominus v \qquad \square$$

The nil change for a value does indeed not change it.

Lemma 2.3 (Behavior of $\mathbf{0}$)**.** Given a change structure $\widehat{V}$ and a value $v \in V$, $v \oplus \mathbf{0}_v = v$. □

Equipped with the preceding definition, we can now restate the definition of derivatives from Eq. (1).

Definition 2.4 (Derivatives)**.** Given change structures $\widehat{A}$ and $\widehat{B}$ and a function $f \in A \to B$ on the change sets of these change structures, we call a binary function f' the *derivative* of f if for all values $a \in A$ and corresponding changes $da \in \Delta_A a$,

$$f\ (a \oplus_A da) = (f\ a) \oplus_B (f'\ a\ da). \qquad \square$$

To avoid parentheses, we give function application precedence over $\oplus$ and $\ominus$ in the remainder of this paper. For instance, the equation above can be written as $f\ (a \oplus_A da) = f\ a \oplus_B f'\ a\ da$.

2.2 Function changes

We will now demonstrate that we can construct change structures for functions between change structures.

A higher-order function f can take other functions as arguments or return them as results. Hence, the derivative of f will respectively take function changes as arguments or return function changes as results. For instance, $f = \lambda x.\ \lambda y.\ x + y$ is a higher-order function, so its derivative gives us the change to the function $g = \lambda y.\ x + y$ in terms of x and its change dx.

The first important design decision is how to represent changes to functions. If a function has type $(\sigma \to \tau)$, we represent a change to that function by a function of type $\sigma \to \Delta\sigma \to \Delta\tau$. By syntactically abusing Δ as a type operator, we can write this as:

$$\Delta(\sigma \to \tau) = \sigma \to \Delta\sigma \to \Delta\tau. \tag{2}$$

A function change df hence takes as input the original value a and its change da. Once we define change structures for functions, we will show that a function change produces as output the difference between the updated output $(f \oplus df)\ (a \oplus da)$ and the original output $f\ a$. This difference is caused by two changes: the change to a given by da and the change of f itself given by df.

$$\begin{array}{rll} \iota & ::= & \ldots & \text{(base types)} \\ \sigma, \tau & ::= & \iota \mid \tau \to \tau & \text{(types)} \\ \Gamma & ::= & \varepsilon \mid \Gamma, x : \tau & \text{(typing contexts)} \\ c & ::= & \ldots & \text{(constants)} \\ s, t & ::= & c \mid \lambda x.\, t \mid t\ t \mid x & \text{(terms)} \end{array}$$

(a) Syntax.

$$\frac{\cdots}{\vdash c : \tau}\ \textsc{Const} \qquad \frac{}{\Gamma_1, x : \tau, \Gamma_2 \vdash x : \tau}\ \textsc{Lookup} \qquad \boxed{\Gamma \vdash t : \tau}$$

$$\frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x.\, t : \sigma \to \tau}\ \textsc{Lam} \qquad \frac{\Gamma \vdash s : \sigma \to \tau \qquad \Gamma \vdash t : \sigma}{\Gamma \vdash s\ t : \tau}\ \textsc{App}$$

(b) Typing.

Figure 1. Our base calculus.

We now define the set of function changes for function $f \in A \to B$. To fulfill the definition of change structure (Definition 2.1), function changes must produce valid changes for their codomain; moreover, it must be possible to "flip" an element change da from a function change to its associated function:

Definition 2.5. Given change structures $\widehat{A}$ and $\widehat{B}$, the set $\Delta_{A\to B} f$ contains all binary functions df so that

(a) $df\ a\ da \in \Delta_B\ (f\ a)$ and
(b) $f\ a \oplus_B df\ a\ da = f\ \ (a \oplus_A da) \oplus_B df\ \ (a \oplus_A da)\ \ \mathbf{0}_{(a \oplus_A da)}$

for all values $a \in A$ and corresponding changes $da \in \Delta_A a$. □

The change structure operations on functions can now be defined as a distributive law.

Definition 2.6 (Operations on function changes)**.** Given change structures $\widehat{A}$ and $\widehat{B}$, the operations $\oplus_{A\to B}$ and $\ominus_{A\to B}$ are defined as follows.

$$\begin{aligned} (f\ \ \oplus_{A\to B} df)\ v \quad &= f\ v \qquad\qquad \oplus_B df\ v\ \mathbf{0}_v \\ (f_2 \ominus_{A\to B} f_1)\ v\ dv &= f_2\ \ (v \oplus_A dv) \ominus_B f_1\ v \end{aligned}$$ □

All these definitions have been carefully set up to ensure that we have in fact lifted change structures to function spaces.

Theorem 2.7. Given change structures $\widehat{A}$ and $\widehat{B}$, the quadruple $(A \to B, \Delta_{A\to B}, \oplus_{A\to B}, \ominus_{A\to B})$ is a change structure, which we denote by $\widehat{A} \to \widehat{B}$. □

As promised, we can show that a function change df reacts to input changes da like the incremental version of f, that is, $df\ a\ da$ computes the change from $f\ a$ to $(f \oplus df)\ \ (a \oplus da)$:

Lemma 2.8 (Incrementalization)**.** Given change structures $\widehat{A}$ and $\widehat{B}$, a function $f \in A \to B$ and a value $a \in A$ with corresponding changes $df \in \Delta_{A\to B} f$ and $da \in \Delta_A a$, we have that

$$(f \oplus_{A\to B} df)\ \ (a \oplus_A da) = f\ a \oplus_B df\ a\ da.$$ □

The lemma is just a restatement of Property 2.5b, which uses $\oplus$ on functions as defined in Definition 2.6.

For instance, incrementalizing

$$\mathbf{app} = \lambda f.\, \lambda x.\, f\ x$$

with respect to the input changes df, dx amounts to calling df on the original second argument x_{old} and on the change dx.

2.3 Nil changes are derivatives

Lemma 2.8 tells us about the form an incremental program may take. If df doesn't change f at all, that is, if $f \oplus df = f$, then Lemma 2.8 becomes

$$f\ \ (a \oplus da) = f\ a \oplus df\ a\ da.$$

It says that df computes the change upon the output of f given a change da upon the input a of f. In other words, the nil change to a function is exactly its derivative (see Definition 2.4):

$\Delta : * \to *$	the type of changes
$\oplus : \tau \to \Delta\tau \to \tau$	update a value with a change
$\ominus : \tau \to \tau \to \Delta\tau$	the change between two values

Figure 2. Erased change structures on simple types.

Theorem 2.9 (Nil changes are derivatives)**.** Given change structures $\widehat{A}$ and $\widehat{B}$ and a function $f \in A \to B$, the change $\mathbf{0}_f$ is the derivative f' of f. □

In this section, we developed the theory of changes to define formally what a derivative is (Definition 2.4) and to recognize that in order to find the derivative of a function, we only have to find its nil change (Theorem 2.9). Next, we want to provide a fully automatic method for finding the nil change of a given function.

3. Incrementalizing λ-calculi

In this section, we show how to incrementalize an arbitrary program in simply-typed λ-calculus. To this end, we define the source-to-source transformation *Derive*. Using the denotational semantics $[\![-]\!]$ we define later (in Sec. 3.4), we can specify *Derive*'s intended behavior: to ensure Eq. (1), $[\![\, \mathit{Derive}(f)\,]\!]$ must be the derivative of $[\![f]\!]$ for any closed term $f : A \to B$. We will overload the word "derivative" and say simply that $\mathit{Derive}(f)$ is the derivative of f.

It is easy to define derivatives of arbitrary functions as:

$$f'\ x\ dx = f\ \ (x \oplus dx) \ominus f\ x.$$

We could implement *Derive* following the same strategy. However, the resulting incremental programs would be no faster than recomputation. We cannot do better for arbitrary mathematical functions, since they are infinite objects which we cannot fully inspect. Therefore, we resort to a source-to-source transformation on simply-typed λ-calculus as defined in Fig. 1. The sets of base types and primitive constants, as well as the typing rules for primitive constants, are on purpose left unspecified and only defined by plugins — they are *extensions points*. Defining different plugins allows to experiment with sets of base types, associated primitives and incrementalization strategies. We show an example plugin in our case study (Sec. 4.4). In this section, we focus on the incrementalization of the features that are shared among all instances of the plugin interface, that is, function types and the associated syntactic forms, λ-abstraction, application and variable references. Throughout the section, we collect requirements on the plugins that instantiate the framework. Definitions provided by the plugin are replaced, in figures, by ellipses ("..."). Satisfying these requirements is sufficient to ensure correct incrementalization.

3.1 Change types and erased change structures

We developed the theory of change structures in the previous section to guide our implementation of *Derive*. By Theorem 2.9, *Derive* has only to find the nil change to the program itself, because nil changes *are* derivatives. However, the theory of change structures is not

$$\Delta\,(\sigma \to \tau) = \sigma \to \Delta\sigma \to \Delta\tau$$
$$\ominus_{\sigma\to\tau} = \lambda g\ f\ x\ dx.\ (g\ \ (x \oplus dx)) \ominus (f\ x)$$
$$\oplus_{\sigma\to\tau} = \lambda f\ df\ x.\ (f\ x) \oplus (df\ x\ \ (x \ominus x))$$

Figure 3. The erased change structures for function types.

directly applicable to the simply-typed λ-calculus, because a precise implementation of change structures requires dependent types. For instance, we cannot describe the set of changes $\Delta_\tau v$ precisely as a non-dependent type, because it depends on the value we plan to update with these changes.

To work around this limitation of our object language, we use a form of *erasure* of dependent types to simple types. In Fig. 2 and Fig. 4(a), we define change types $\Delta\tau$ as an approximate description of change sets $\Delta_\tau v$ (Fig. 4(b)). In particular, all changes in $\Delta_\tau v$ correspond to values of terms with type $\Delta\tau$, but not necessarily the other way around. For instance, in the change structure for natural numbers described in Sec. 2.1, we would have $\Delta\mathbf{Nat} = \mathbf{Int}$, even though not every integer is a change for every natural number. For primitive types ι, $\Delta\iota$ and its associated $\oplus$ and $\ominus$ operator must be provided by the plugin developer. For function types, erased change structures are given by Fig. 3. Erasing dependent types in all components of a change structure, we obtain *erased change structures*, which represent change structures as simply-typed λ-terms where $\oplus$ and $\ominus$ are families of λ-terms.

Erased change structures are not change structures themselves. However, we will show how change structures and erased changes structures have "almost the same" behavior (Sec. 3.6). We will hence be able to apply our theory of changes.

3.2 Differentiation

When f is a closed term of function type, $\mathcal{D}\mathit{erive}(f)$ should be its derivative. More in general, as discussed, we want that when t is a closed term, $\mathcal{D}\mathit{erive}(t)$ is its nil change. Since $\mathcal{D}\mathit{erive}$ recurses on open terms, we need a more general specification. We require that if $\Gamma \vdash t : \tau$, then $\mathcal{D}\mathit{erive}(t)$ represents the change in t (of type $\Delta\tau$) in terms of changes to the values of its free variables. As a special case, when t is a closed term, there is no free variable to change; hence, the change to t will be as desired the nil change of t.

The following typing rule shows the static semantics of $\mathcal{D}\mathit{erive}$:

$$\frac{\Gamma \vdash t : \tau}{\Gamma, \Delta\Gamma \vdash \mathcal{D}\mathit{erive}(t) : \Delta\tau}\ \textsc{Derive}$$

We see that $\mathcal{D}\mathit{erive}(t)$ has access both to the free variables in t (from Γ) and to their changes (from $\Delta\Gamma$, defined in Fig. 4(d)). For example, if a well-typed term t contains x free, then Γ contains an assumption $x : \tau$ for some τ and $\Delta\Gamma$ contains the corresponding assumption $dx : \Delta\tau$. Hence, $\mathcal{D}\mathit{erive}(t)$ can access the change of x by using dx. For simplicity, we assume that the original program contains no variable names that start with d.The definition of $\mathcal{D}\mathit{erive}$ will ensure that the dx variables are bound if the original term is closed.

Let us analyzes each case of the definition of $\mathcal{D}\mathit{erive}(u)$ (Fig. 4(g)):

- If $u = x$, $\mathcal{D}\mathit{erive}(x)$ must be the change of x, that is dx.
- If $u = \lambda x.\ t$, $\mathcal{D}\mathit{erive}(t)$ is the change of u given the change in its free variables. The change of u is then the change of t as a function of the *base input* x and its change dx, with respect to changes in other open variables. Hence, we simply need to bind dx by defining $\mathcal{D}\mathit{erive}(\lambda x.\ t) = \lambda x.\ \lambda dx.\ \mathcal{D}\mathit{erive}(t)$.
- If $u = s\ t$, $\mathcal{D}\mathit{erive}(s)$ is the change of s as a function of its base input and change. Hence, we simply apply $\mathcal{D}\mathit{erive}(s)$ to the actual base input t and change $\mathcal{D}\mathit{erive}(t)$, giving $\mathcal{D}\mathit{erive}(s\ t) = \mathcal{D}\mathit{erive}(s)\ t\ \mathcal{D}\mathit{erive}(t)$.
- If $t = c$: since c is a closed term, its change is a nil change, hence (by Theorem 2.9) c's derivative. We can obtain a correct derivative for constants by setting:
$$\mathcal{D}\mathit{erive}(c) = c \ominus c = \mathbf{0}_c = c'$$
This definition is inefficient for functional constants; hence plugins must provide derivatives of the primitives they define.

This might seem deceptively simple. But λ-calculus only implements binding of values, leaving "real work" to primitives; likewise, differentiation for λ-calculus only implement binding of changes, leaving "real work" to derivatives of primitives. However, our support for λ-calculus allows to *glue* the primitives together.

We have now informally derived the definition of $\mathcal{D}\mathit{erive}$ (Fig. 4(g)) from its specification (Eq. (1)) and its typing. But formally speaking, we have defined $\mathcal{D}\mathit{erive}$, hence we must prove that $\mathcal{D}\mathit{erive}$ satisfies Eq. (1). This proof is discussed in the remainder of the section.

3.3 Architecture of the proof

$\mathcal{D}\mathit{erive}(t)$ is defined using change types. As discussed in Sec. 3.1, change types impose on their members less restrictions than corresponding change structures – they contain "junk" (such as the change -5 for the natural number 3). We cannot constrain the behavior of $\mathcal{D}\mathit{erive}(t)$ on such junk; a direct correctness proof fails. To avoid this problem, our proof defines a version of $\mathcal{D}\mathit{erive}$ which uses change structures instead.

To this end, we first present a standard denotational semantics $[\![-]\!]$ for simply-typed λ-calculus. Using our theory of changes, we associate change structures to our domains. We define a non-standard denotational semantics $[\![-]\!]^\Delta$, which is analogous to $\mathcal{D}\mathit{erive}$ but operates on elements of change structures, so that it needn't deal with junk. As a consequence, we can prove that $[\![t]\!]^\Delta$ is the derivative of $[\![t]\!]$: this is our key result.

Finally, we define a correspondence between change sets and domains associated with change types, and show that whenever $[\![t]\!]^\Delta$ has a certain behavior on an input, $[\![\mathcal{D}\mathit{erive}(t)]\!]$ has the corresponding behavior on the corresponding input. Our correctness property follows as a corollary.

3.4 Denotational semantics

In order to prove that incrementalization preserves the meaning of terms, we define a denotational semantics of the object language. We first associate a domain with every type, given the domains of base types provided by the plugin. Since our calculus is strongly normalizing and all functions are total, we can avoid using domain theory to model partiality: our domains are simply sets. Likewise, we can use functions as the domain of function types.

Definition 3.1 (Domains)**.** The domain $[\![\tau]\!]$ of a type τ is defined as in Fig. 4(c). □

Given this domain construction, we can now define an evaluation function for terms. The plugin has to provide the evaluation function for constants. In general, the evaluation function $[\![t]\!]$ computes the value of a well-typed term t given the values of all free variables in t. The values of the free variables are provided in an environment.

Definition 3.2 (Environments)**.** An environment ρ assigns values to the names of free variables.

$$\rho ::= \varepsilon \mid \rho, x = v$$

We write $[\![\Gamma]\!]$ for the set of environments that assign values to the names bound in Γ (see Fig. 4(f)). □

$\boxed{\Delta\tau}$ $\boxed{dv, df \in \Delta_\tau v}$ $\boxed{v, f \in [\![\tau]\!]}$

$$\Delta\iota = \dots \qquad \Delta_\iota v = \dots \subseteq [\![\Delta\iota]\!] \qquad [\![\iota]\!] = \dots$$

$$\Delta(\sigma \to \tau) = \sigma \to \Delta\sigma \to \Delta\tau \qquad \Delta_{(\sigma\to\tau)} f = \{df \in (x : [\![\sigma]\!]) \to \Delta_\sigma x \to \Delta_\tau (f\ x) \mid (f \oplus_{A\to B} df)\ (a \oplus_A da) = f\ a \oplus_B df\ a\ da\} \qquad [\![\sigma \to \tau]\!] = [\![\sigma]\!] \to [\![\tau]\!]$$

(a) Change types. (b) Change values. (c) Standard values.

$\boxed{\Delta\Gamma}$ $\boxed{d\rho \in \Delta_\Gamma \rho}$ $\boxed{\rho \in [\![\Gamma]\!]}$

$$\Delta\varepsilon = \varepsilon \qquad \Delta_\varepsilon \varnothing = \{\varnothing\} \qquad [\![\varepsilon]\!] = \{\varnothing\}$$

$$\Delta(\Gamma, x : \tau) = \Delta\Gamma, dx : \Delta\tau \qquad \Delta_{(\Gamma,x:\tau)}(\rho, x = v) = \{(d\rho, dx = dv) \mid d\rho \in \Delta_\Gamma\rho \wedge dv \in \Delta_\tau v\} \qquad [\![\Gamma, x : \tau]\!] = \{(\rho, x = v) \mid \rho \in [\![\Gamma]\!] \wedge v \in [\![\tau]\!]\}$$

(d) Change contexts. (e) Change environments. (f) Standard environments.

$\boxed{\Delta t}$ $\boxed{[\![t]\!]^\Delta \rho\ d\rho}$ $\boxed{[\![t]\!]\ \rho}$

$$\mathit{Derive}(c) = \dots \qquad [\![c]\!]^\Delta \rho\ d\rho = \dots \qquad [\![c]\!]\ \rho = \dots$$

$$\mathit{Derive}(\lambda x.\ t) = \lambda x\ dx.\ \mathit{Derive}(t) \qquad [\![\lambda x.\ t]\!]^\Delta \rho\ d\rho = \lambda v\ dv.\ [\![t]\!]^\Delta (\rho, x = v)\ (d\rho, dx = dv) \qquad [\![\lambda x.\ t]\!]\ \rho = \lambda v.\ [\![t]\!]\ (\rho, x = v)$$

$$\mathit{Derive}(s\ t) = \mathit{Derive}(s)\ t\ \mathit{Derive}(t) \qquad [\![s\ t]\!]^\Delta \rho\ d\rho = ([\![s]\!]^\Delta \rho\ d\rho)\ ([\![t]\!]\ \rho)\ ([\![t]\!]^\Delta \rho\ d\rho) \qquad [\![s\ t]\!]\ \rho = ([\![s]\!]\ \rho)\ ([\![t]\!]\ \rho)$$

$$\mathit{Derive}(x) = dx \qquad [\![x]\!]^\Delta \rho\ d\rho = \mathit{lookup}\ dx\ \mathit{in}\ d\rho \qquad [\![x]\!]\ \rho = \mathit{lookup}\ x\ \mathit{in}\ \rho$$

(g) Differentiation. (h) Differential evaluation. (i) Standard evaluation.

Figure 4. Standard and differential behavior of the simply-typed λ-calculus. The left column defines differentiation as a source-to-source transformation. The right column defines the standard semantics of the simply-typed lambda calculus. The middle column connects these artifacts via a differential semantics that maps λ-terms to the derivative of their standard semantics.

Definition 3.3 (Evaluation)**.** Given $\Gamma \vdash t : \tau$, the meaning of t is defined by the function $[\![t]\!]$ of type $[\![\Gamma]\!] \to [\![\tau]\!]$ in Fig. 4(i). □

This is the standard semantics of the simply-typed λ-calculus. We can now specify what it means to incrementalize the simply-typed λ calculus with respect to this semantics.

3.5 Change semantics

The informal specification of differentiation is to map changes in a program's input to changes in the program's output. In order to formalize this specification in terms of change structures and the denotational semantics of the object language, we now define a non-standard denotational semantics of the object language that computes changes. The evaluation function $[\![t]\!]^\Delta$ computes how the value of a well-typed term t changes given both the values and the changes of all free variables in t. In the special case that none of the free variables change, $[\![t]\!]^\Delta$ computes the nil change. By Theorem 2.9, this is the derivative of $[\![t]\!]$ which maps changes to the input of $[\![t]\!]$ to changes of the output of $[\![t]\!]$, as required.

First, we define a change structure on $[\![\tau]\!]$ for all τ. The carrier Δ_τ of these change structures will serve as non-standard domain for the change semantics. The plugin provides a change structure $\widehat{C}_\iota$ on base type ι such that $\forall v.\Delta_\iota v \subseteq [\![\Delta\iota]\!]$.

Definition 3.4 (Changes)**.** Given a type τ, we define a change structure $\widehat{C}_\tau$ for $[\![\tau]\!]$ by induction on the structure of τ. If τ is a base type ι, then the result $\widehat{C}_\iota$ is supplied by the plugin. Otherwise we use the construction from Theorem 2.7 and define

$$\widehat{C}_{\sigma\to\tau} = \widehat{C}_\sigma \to \widehat{C}_\tau. \qquad \square$$

To talk about the derivative of $[\![t]\!]$, we need a change structure on its domain, that is on the set of environments. Since environments are (heterogeneous) lists of values, we can lift operations on change structures to change structures on environments, acting pointwise in the obvious way.

Definition 3.5 (Change environments)**.** Given a context Γ, we define a change structure $\widehat{C}_\Gamma$ on the corresponding environments $[\![\Gamma]\!]$ and change environments $\Delta_\Gamma\rho$ in Fig. 4(e).

The operations $\oplus_\rho$ and $\ominus_\rho$ are defined as follows.

$$\varepsilon \oplus \varepsilon = \varepsilon$$
$$(\rho, x = v) \oplus (d\rho, dx = dv) = (\rho \oplus d\rho), x = (v \oplus dv)$$

$$\varepsilon \ominus \varepsilon = \varepsilon$$
$$(\rho_2, x = v_2) \ominus (\rho_1, x = v_1) = (\rho_2 \ominus \rho_1), x = (v_2 \ominus v_1)$$

The properties in Definition 2.1 follow directly from the same properties for the underlying change structures $\widehat{C}_\tau$. □

At this point, we can define the change semantics of terms and prove that $[\![t]\!]^\Delta$ it is the derivative of $[\![t]\!]$. For each constant c, the plugin provides $[\![c]\!]^\Delta$, the derivative of $[\![c]\!]$.

Definition 3.6 (Change semantics)**.** The function $[\![t]\!]^\Delta$ is defined in Fig. 4(h). □

Lemma 3.7. Given $\Gamma \vdash t : \tau$, $[\![t]\!]^{\Delta}$ is the derivative of $[\![t]\!]$. □

3.6 Correctness of differentiation

We can now prove that the behavior of $[\![\mathcal{D}erive(t)]\!]$ is consistent with the behavior of $[\![t]\!]^{\Delta}$. This leads us to the proof of the correctness theorem mentioned in the introduction.

The logical relation [18, Chapter 8] of *erasure* captures the idea that an element of a change structure stays almost the same after we erase all traces of dependent types from it.

Definition 3.8 (Erasure)**.** Let $dv \in \Delta_\tau v$ and $dv' \in [\![\Delta\tau]\!]$. We say dv erases to dv', or $dv \sim^v_\tau dv'$, if one of the following holds:

(a) τ is a base type and $dv = dv'$.
(b) $\tau = \sigma_0 \to \sigma_1$ and for all w, dw, dw' such that $dw \sim^w_{\sigma_0} dw'$, we have $(dv\ w\ dw) \sim^{(v\ w)}_{\sigma_1} (dv'\ w\ dw')$. □

Sometimes we shall also say that $dv \in \Delta_\tau v$ erases to a closed term $dt : \Delta t$, in which case we mean dv erases to $([\![dt]\!]\ \varnothing)$.[1]

The following lemma makes precise what we meant by "almost the same".

Lemma 3.9. Suppose $dv \sim^v_\tau dv'$. If $\oplus'$ is the erased version of the update operator $\oplus$ of the change structure of τ (Sec. 3.1), then

$$v \oplus dv = v \oplus' dv'.$$ □

It turns out that $[\![t]\!]^{\Delta}$ and $\mathcal{D}erive(t)$ are "almost the same". For closed terms, we make this precise by:

Lemma 3.10. If $(t : \tau)$ is closed, then $([\![t]\!]^{\Delta} \varnothing\ \varnothing)$ erases to $\mathcal{D}erive(t)$. □

We omit for lack of space a more general version of Lemma 3.10, which holds also for open terms, but requires defining erasure on environments. The main correctness theorem is a corollary of Lemmas 3.7, 3.9 and 3.10.

Theorem 3.11 (Correctness of differentiation)**.** Let $f : \sigma \to \tau$ be a closed term of function type. For every closed base term $s : \sigma$ and for every closed change term $ds : \Delta\sigma$ such that some change $dv \in \Delta_\sigma [\![s]\!]$ erases to ds, we have

$$f\ (s \oplus ds) \cong (f\ s) \oplus (\mathcal{D}erive(f)\ s\ ds),$$

where $\cong$ is denotational equality ($a \cong b$ iff $[\![a]\!] = [\![b]\!]$). □

Theorem 3.11 is a more precise restatement of Eq. (1). Requiring the existence of dv ensures that ds evaluates to a change, and not to junk in $[\![\Delta\sigma]\!]$.

3.7 Plugins

Both our correctness proof and the differentiation framework (which is the basis for our implementation) are parametric in the plugin. Instantiating the differentiation framework requires a *differentiation plugin*; instantiating the correctness proof for it requires a *proof plugin*.

To allow executing and differentiating λ-terms, a differentiation plugin must provide:

- base types, and for each base type ι, the erased change structure of ι as specified in Fig. 2,
- primitives, and for each primitive c, the term $\mathcal{D}erive(c)$.

To instantiate the correctness proof to a plugin, one must provide additional definitions and lemmas. For each base type ι, a proof plugin must provide:

- a semantic domain $[\![\iota]\!]$,
- a change structure $\widehat{C}_\iota$ such that $\forall v. \Delta_\iota v \subseteq [\![\Delta\iota]\!]$,
- a proof that $\widehat{C}_\iota$ erases to the corresponding erased change structure in the differentiation plugin.

For each primitive $c : \tau$, the proof plugin must provide:

- its value $[\![c]\!]$ in the domain $[\![\tau]\!]$,
- its derivative $([\![c]\!]^{\Delta} \varnothing\ \varnothing)$[1] in the change set of τ,
- a proof that $([\![c]\!]^{\Delta} \varnothing\ \varnothing)$ erases to the term $\mathcal{D}erive(c)$.

To show that the interface for proof plugins can be implemented, we wrote a small proof plugin with integers and bags of integers. To show that differentiation plugins are practicable, we have implemented the transformation and a differentiation plugin which allows the incrementalization of non-trivial programs. This is presented in the next section.

4. Differentiation in practice

In practice, successful incrementalization requires both correctness and performance of the derivatives. Correctness of derivatives is guaranteed by the theoretical development the previous sections, together with the interface for differentiation and proof plugins, whereas performance of derivatives has to come from careful design and implementation of differentiation plugins.

4.1 The role of differentiation plugins

Users of our approach need to (1) choose which base types and primitives they need, (2) implement suitable differentiation plugins for these base types and primitives, (3) rewrite (relevant parts of) their programs in terms of these primitives and (4) arrange for their program to be called on changes instead of updated inputs.

As discussed in Sec. 3.2, differentiation supports abstraction, application and variables, but since computation on base types is performed by primitives for those types, efficient derivatives for primitives are essential for good performance.

To make such derivatives efficient, change types must also have efficient implementations, and allow describing precisely what changed. The efficient derivative of sum in Sec. 1 is possible only if bag changes can describe deletions and insertions, and integer changes can describe additive differences.

For many conceivable base types, we do not have to design the differentiation plugins from scratch. Instead, we can reuse the large body of existing research on incrementalization in first-order and domain-specific settings. For instance, we reuse the approach from Gluche et al. [11] to support incremental bags and maps. By wrapping a domain-specific incrementalization result in a differentiation plugin, we adapt it to be usable in the context of a higher-order and general-purpose programming language, and in interaction with other differentiation plugins for the other base types of that language.

For base types with no known incrementalization strategy, the precise interfaces for differentiation and proof plugins can guide the implementation effort. These interfaces could also from the basis for a library of differentiation plugins that work well together.

Rewriting whole programs in our language would be an excessive requirements. Instead, we embed our object language as an EDSL in some more expressive meta-language (Scala in our case study), so that embedded programs are reified. The embedded language can be made to resemble the metalanguage [21]. To incrementalize a part of a computation, we write it in our embedded object language, invoke $\mathcal{D}erive$ on the embedded program, optionally optimize the resulting programs and finally invoke them. The metalanguage also acts as a macro system for the object language, as usual. This allows

[1] To evaluate a closed term t, we need no environment entries, so the empty environment $\varnothing$ suffices: $([\![t]\!]\ \varnothing)$ is the value of t in the empty environment, and $([\![t]\!]^{\Delta} \varnothing\ \varnothing)$ is the value of t using the change semantics, the empty environment and the empty change environment.

$histogram :: \text{Map Int (Bag } word) \to \text{Map } word \text{ Int}$
$histogram = mapReduce \text{ groupOnBags } additiveGroupOnIntegers\ histogramMap\ histogramReduce$
where $additiveGroupOnIntegers = \text{Group } (+)\ (\lambda n \to -n)\ 0$
$histogramMap\ \_ = \text{foldBag groupOnBags } (\lambda n \to \text{singletonBag } (n, 1))$
$histogramReduce\ \_ = \text{foldBag } additiveGroupOnIntegers \text{ id}$

-- Precondition:
-- For every $key_1 :: k_1$ and $key_2 :: k_2$, the terms $mapper\ key_1$ and $reducer\ key_2$ are homomorphisms.
$mapReduce :: \text{Group } v_1 \to \text{Group } v_3 \to (k_1 \to v_1 \to \text{Bag } (k_2, v_2)) \to (k_2 \to \text{Bag } v_2 \to v_3) \to \text{Map } k_1\ v_1 \to \text{Map } k_2\ v_3$
$mapReduce\ group_1\ group_3\ mapper\ reducer = reducePerKey \circ groupByKey \circ mapPerKey$
where $mapPerKey = \text{foldMap } group_1 \text{ groupOnBags}$ $mapper$
$groupByKey = \text{foldBag } (\text{groupOnMaps groupOnBags})$ $(\lambda(key, val) \to \text{singletonMap } key \text{ (singletonBag } val))$
$reducePerKey = \text{foldMap groupOnBags } (\text{groupOnMaps } group_3)\ (\lambda key\ bag \to \text{singletonMap } key \text{ (} reducer\ key\ bag))$

Figure 5. The λ-term $histogram$ with Haskell-like syntactic sugar. $additiveGroupOnIntegers$ is the group on integers described in Sec. 2.1.

us to simulate polymorphic collections such as (**Bag** ι) even though the object language is simply-typed; technically, our plugin exposes a family of base types to the object language.

4.2 Predicting nil changes

Handling changes to all inputs can induce excessive overhead in incremental programs[2]. It is also often unnecessary; for instance, the function argument of *fold* in Sec. 1 does not change since it is a closed subterm of the program, so *fold* will receive a nil change for it. A (conservative) static analysis can detect changes that are guaranteed to be nil at runtime. We can then specialize derivatives that receive this change, so that they need not inspect the change at runtime.

For our case study, we have implemented a simple static analysis which detects and propagates information about closed terms. The analysis is not interesting and we omit details for lack of space.

4.3 Self-maintainability

In databases, a self-maintainable view [12] is a function that can update its result from input changes alone, without looking at the actual input. By analogy, we call a derivative *self-maintainable* if it uses no base parameters, only their changes. Self-maintainable derivatives describe efficient incremental computations: since they do not use their base input, their running time does not have to depend on the input size.

For instance, $union$ on bags is self-maintainable with the change structure $\widehat{\textbf{Bag}\ \iota}$ described in Sec. 2.1; its derivative $\mathcal{D}erive(union) = (\lambda x\ dx\ y\ dy.\ union\ dx\ dy)$ does not use the base inputs x and y. On the other hand, $foldBag$ is not necessarily self-maintainable. However, $(foldBag\ f)$ is self-maintainable if we can predict that changes to f are going to be nil. We take advantage of this by implementing a specialized derivative.

To avoid the recomputation of base arguments for self-maintainable derivatives (which never need them), we currently employ lazy evaluation. Since we could use standard techniques for dead-code elimination [7] instead, laziness is not central to our approach, however. Derivatives which are not self-maintainable need their base arguments, which can be expensive to compute. Since they are also computed while running the base program, one could reuse the previously computed value through memoization or extensions of static caching (as discussed in Sec. 5). We leave implementing these optimizations for future work. As a consequence, our current implementation delivers good results only if most derivatives are self-maintainable.

4.4 Case study

To demonstrate that IλC can speed up realistic programs, we perform a case study on a nontrivial one. We take the MapReduce-based skeleton of the word-count example, as described by Lämmel [15]. We define a suitable differentiation plugin, adapt the program to use it and show that incremental computation is faster than recomputation on it. We designed and implemented the differentiation plugin following the requirements on the corresponding proof plugin, even though we did not yet formally (for example, in Agda) define the proof plugin. For lack of space, we focus on base types which are crucial for our example and its performance, that is, collections. The plugin also implements tuples, tagged unions, Booleans and integers with the usual introduction and elimination forms, with few optimizations for their derivatives.

$wordcount$ takes a map from document IDs to documents and produces a map from words appearing in the input to the count of their appearances, that is, a histogram:

$$wordcount : \textbf{Map ID Document} \to \textbf{Map Word Int}$$

For simplicity, instead of modeling strings, we model documents as bags of words and document IDs as integers. Hence, what we implement is:

$$histogram : \textbf{Map Int } (\textbf{Bag } a) \to \textbf{Map } a \textbf{ Int}$$

We model words by integers ($a = Int$), but treat them parametrically. Other than that, we adapt directly Lämmel's code to our language. Figure 5 shows the λ-term $histogram$.

Figure 6 shows a simplified Scala implementation of the primitives used in Fig. 5. As bag primitives, we provide constructors and a fold operation, following Gluche et al. [11]. The constructors for bags are $\emptyset$ (constructing the empty bag), $singleton$ (constructing a bag with one element), $union$ (constructing the union of two bags) and $negate$ ($negate\ b$ constructs a bag with the same elements as b but negated multiplicities); all but $singleton$ represent abelian group operations. Unlike for usual ADT constructors, the same bag can be constructed in different ways, which are equivalent by the equations defining abelian groups; for instance, since $union$ is commutative, $union\ x\ y = union\ y\ x$. Folding on a bag will represent the bag through constructors in an arbitrary way, and then replace constructors with arguments; to ensure a well-defined result, the arguments of fold should respect the same equations, that is, they should form an abelian group; for instance, the binary operator should be commutative. Hence, the fold operator $foldBag$ can be defined to take a function (corresponding to $singleton$) and an abelian group (for the other constructors). $foldBag$ is then defined by equations:

$$\begin{aligned}
&foldBag : \textbf{Group } \tau \to (\sigma \to \tau) \to && \textbf{Bag } \sigma \to \tau \\
&foldBag\ g@(_, \boxplus, \boxminus, e)\ f\ \emptyset && = e \\
&foldBag\ g@(_, \boxplus, \boxminus, e)\ f\ (union\ b_1\ b_2) && = foldBag\ g\ f\ b_1 \\
& && \quad \boxplus\ foldBag\ g\ f\ b_1 \\
&foldBag\ g@(_, \boxplus, \boxminus, e)\ f\ (negate\ b) && = \boxminus\ (foldBag\ g\ f\ b) \\
&foldBag\ g@(_, \boxplus, \boxminus, e)\ f\ (singleton\ v) && = f\ v
\end{aligned}$$

If g is a group, these equations specify $foldBag\ g$ precisely [11]. Moreover, since $foldBag\ g\ f$ satisfies the first three equations, it

```
// Abelian groups
abstract class Group[A] {
  def merge(value₁: A, value₂: A): A
  def inverse(value: A): A
  def zero: A
}

// Bags
type Bag[A] = collection.immutable.HashMap[A, Int]

def groupOnBags[A] = new Group[Bag[A]] {
  def merge(bag₁: Bag[A], bag₂: Bag[A]) = ...
  def inverse(bag: Bag[A]) = bag map {
    case (value, count) ⇒ (value, -count)
  }
  def zero = collection.immutable.HashMap()
}

def foldBag[A, B](group: Group[B], f: A ⇒ B, bag: Bag[A]): B =
  bag.flatMap {
    case (x, c) if c ≥ 0 ⇒ Seq.fill(c)(f(x))
    case (x, c) if c < 0 ⇒ Seq.fill(-c)(group.inverse(f(x)))
  }.fold(group.zero)(group.merge)

// Maps
type Map[K, A] = collection.immutable.HashMap[K, A]

def groupOnMaps[K, A](group: Group[A]) = new Group[Map[K, A]] {
  def merge(dict₁: Map[K, A], dict₂: Map[K, A]): Map[K, A] =
    (dict₁ merged dict₂) {
      case ((k, v₁), (_, v₂)) ⇒ (k, group.merge(v₁, v₂))
    } filter {
      case (k, v) ⇒ v ≠ group.zero
    }

  def inverse(dict: Map[K, A]): Map[K, A] = dict map {
    case (k, v) ⇒ (k, group.inverse(v))
  }

  def zero = collection.immutable.HashMap()
}

// The general map fold
def foldMapGen[K, A, B](zero: B, merge: (B, B) ⇒ B)
  (f: (K, A) ⇒ B, dict: Map[K, A]): B =
    dict.map(Function.tupled(f)).fold(zero)(merge)

// By using foldMap instead of foldMapGen, the user promises that
// f k is a homomorphism from groupA to groupB for each k : K.
def foldMap[K, A, B](groupA: Group[A], groupB: Group[B])
  (f: (K, A) ⇒ B, dict: Map[K, A]): B =
    foldMapGen(groupB.zero, groupB.merge)(f, dict)
```

Figure 6. A Scala implementation of primitives for bags and maps. In the code, we call $\boxplus$, $\boxminus$ and e respectively *merge*, *inverse*, and *zero*. We also omit the relatively standard primitives.

satisfies the definition of an *abelian group homomorphism* between the abelian group on bags and the group g (because those equations coincide with the definition). Figure 6 shows an implementation of $foldBag$ as specified above. Moreover, all functions which deconstruct a bag can be expressed in terms of $foldBag$ with suitable arguments. For instance, we can sum the elements of a bag of integers with $foldBag\ gZ\ (\lambda x.\ x)$, where gZ is the abelian group on integers defined in Sec. 2.1. Users of $foldBag$ can define different abelian groups to specify different operations (for instance, to multiply floating-point numbers).

If g and f do not change, $foldBag\ g\ f$ has a self-maintainable derivative. By the equations above,

$$\begin{aligned}
&foldBag\ g\ f\ (b \oplus db)\\
&= foldBag\ g\ f (union\ b\ db)\\
&= foldBag\ g\ f\ b \boxplus foldBag\ g\ f\ db\\
&= foldBag\ g\ f\ b \oplus GroupChange\ g\ \ (foldBag\ g\ f\ db)
\end{aligned}$$

We will describe the $GroupChange$ change constructor in a moment. Before that, we note that as a consequence, the derivative of $foldBag\ g\ f$ is

$$\lambda b\ db.\ GroupChange\ g\ \ (foldBag\ g\ f\ db)\,,$$

and we can see it does not use b: as desired, it is *self-maintainable*. Additional restrictions are require to make $foldMap$'s derivative self-maintainable. Those restrictions require the precondition on $mapReduce$ in Fig. 5. $foldMapGen$ has the same implementation but without those restrictions; as a consequence, its derivative is not self-maintainable, but it is more generally applicable. Lack of space prevents us from giving more details.

To define $GroupChange$, we need a suitable erased change structure on τ, such that $\oplus$ will be equivalent to $\boxplus$. Since there might be multiple groups on τ, we *allow the changes to specify a group*, and have $\oplus$ delegate to $\boxplus$:

$$\begin{aligned}
&\Delta\tau = Replace\ \tau \mid GroupChange\ (\textbf{AbelianGroup}\ \tau)\ \tau\\
&v \oplus (Replace\ u) = u\\
&v \oplus (GroupChange\ (\bullet, inverse, zero)\ dv) = v \bullet dv\\
&v \ominus u = Replace\ v
\end{aligned}$$

That is, a change between two values is either simply the new value (which replaces the old one, triggering recomputation), or their difference (computed with abelian group operations, like in the changes structures for groups from Sec. 2.1. The operator $\ominus$ does not know which group to use, so it does not take advantage of the group structure. However, $foldBag$ is now able to generate a group change.

4.5 Benchmarks

Benchmarking our case study shows that IλC can offer order-of-magnitude speedups for a realistic higher-order program.

Benchmarking setup We run object language programs by generating corresponding Scala code. To ensure rigorous benchmarking [10], we use the Scalameter benchmarking library. To show that the performance difference from the baseline is statistically significant, we show 99%-confidence intervals in graphs.

We verify Eq. (1) experimentally by checking that the two sides of the equation always evaluate to the same value.

Input generation Inputs are randomly generated to resemble English words over all webpages on the internet: The vocabulary size and the average length of a webpage stay relatively the same, while the number of webpages grows day by day. To generate a size-n input of type (**Map Int** (**Bag Int**)), we generate n random numbers between 1 and 1000 and distribute them randomly in $n/1000$ bags. Changes are randomly generated to resemble edits. A change has 50% probability to delete a random existing number, and has 50% probability to insert a random number at a random location.

Experimental units Thanks to Eq. (1), both recomputation $f\ (a \oplus da)$ and incremental computation $(f\ a) \oplus (\mathcal{D}erive(f)\ a\ da)$ produce the same result. To show that derivatives are faster, we compare these two computations. To compare with recomputation, we measure the *aggregated* running time for running the derivative on the change and for updating the original output with the result of the derivative.

4.6 Experimental results

We present our results in Fig. 7. As expected, the runtime of incremental computation is *essentially constant* in the size of the input, while the runtime of recomputation is linear in the input size. Hence, on our biggest inputs incremental computation is over 10^4 times faster.

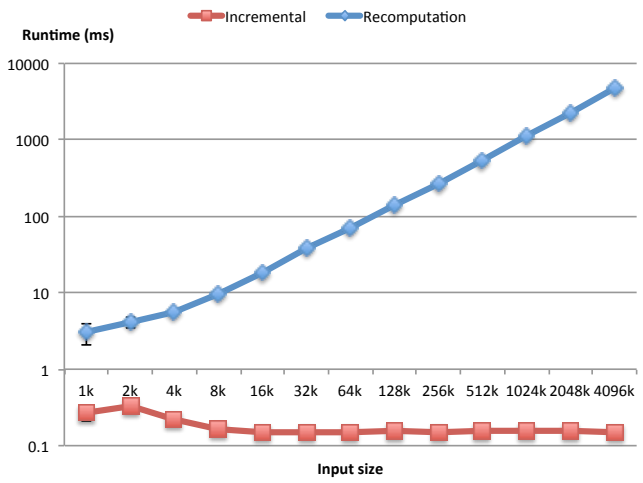

Figure 7. Performance results in log-log scale, with input size on the x-axis and runtime in ms on the y-axis. Confidence intervals are shown by the whiskers; most whiskers are simply too small to be visible.

Derivative time is in fact slightly irregular for the first few inputs, but this irregularity decreases slowly with increasing warmup cycles. In the end, for derivatives we use 10^4 warmup cycles. With fewer warmup cycles, running time for derivatives decreases significantly during execution, going from 2.6ms for $n = 1000$ to 0.2ms for $n = 512000$. Hence, we believe extended warmup is appropriate, and the changes do not affect our general conclusions. Considering confidence intervals, in our experiments the running time for derivatives varies between 0.139ms and 0.378ms.

In our current implementation, the code of the generated derivatives can become quite big. For the histogram example (which is around 1KB of code), a pretty-print of its derivative is around 40KB of code. The function application case in Fig. 4(g) can lead to a quadratic growth in the worst case. We believe that the code size of the derivative can be reduced again by common subexpression elimination, but we did not yet pursue that option.

Summary Our results show that the incrementalized program runs in essentially constant time and hence orders of magnitude faster than the alternative of recomputation from scratch.

5. Related work

Existing work on incremental computation can be divided into two groups: Static incrementalization and dynamic incrementalization. Static approaches analyze a program statically and generate an incremental version of it. Dynamic approaches create dynamic dependency graphs while the program runs and propagate changes along these graphs.

The trade-off between the two is that static approaches have the potential to be faster because no dependency tracking at runtime is needed, whereas dynamic approaches can support more powerful programming languages. The quick summary of how IλC fits into this landscape is that it pushes the envelope with regard to the expressive power of languages whose programs can be incrementalized statically.

In the remainder of this section, we analyze the relation to the most closely related prior works. Ramalingam and Reps [20] and Acar et al. [3] discuss further related work.

5.1 Dynamic approaches

One of the most advanced dynamic approach to incrementalization is self-adjusting computation, which has been applied to Standard ML and large subsets of C [2, 13]. In this approach, programs execute on the original input in an enhanced runtime environment that tracks the dependencies between values in a *dynamic dependence graph* [3]; intermediate results are memoized. Later, changes to the input propagate through dependency graphs from changed inputs to results, updating both intermediate and final results; this processing is often more efficient than recomputation.

However, creating dynamic dependence graphs imposes a large constant-factor overhead during runtime, ranging from 2 to 30 in reported experiments [4, 5], and affecting the initial run of the program on its base input. Acar et al. [5] show how to support high-level data types in the context of self-adjusting computation; however, the approach still requires expensive runtime bookkeeping during the initial run. Like other static approaches, our work needs no modified runtime environment and has no overhead during base computation, though it may be less efficient when processing changes. This pays off if the initial input is big compared to its changes.

Chen et al. [8] have developed a static transformation for purely functional programs, but this transformation just provides a superior inferface to use the runtime support with less boilerplate, and does not reduce this performance overhead. Hence, it is still a dynamic approach and should not be confused with the transformation we show in this work.

Another property of self-adjusting computation is that incrementalization is only efficient if the program has a suitable computation structure. For instance, a program folding the elements of a bag with a left or right fold will not have efficient incremental behavior; instead, it's necessary that the fold be shaped like a balanced tree. In general, incremental computations become efficient only if they are *stable* [1]. Hence one may need to massage the program to make it efficient. Our methodology is different: Since we do not aim to incrementalize arbitrary programs written in standard programming languages, we can select primitives that have efficient derivatives and thereby require the programmer to use them.

Functional reactive programming [9] can also be seen as a dynamic approach to incremental computation; recent work by Maier and Odersky [17] has focused on speeding up reactions to input changes by making them incremental on collections. Dynamic techniques are also used by Willis et al. [23] to incrementalize JQL queries.

5.2 Static approaches

Static approaches analyze a program at compile-time and produce an incremental version that efficiently updates the output of the original program according to changing inputs.

Static approaches have the potential to be more efficient than dynamic approaches, because no bookkeeping at runtime is required. Also, the computed incremental versions can often be optimized using standard compiler techniques such as constant folding or inlining. However, none of them support first-class functions; some approaches have further restrictions.

Our aim is to apply static incrementalization to more expressive languages; in particular, IλC supports first-class functions and an open set of base types with associated primitive operations.

5.2.1 Finite differencing

Our work and terminology is partially inspired by *finite differencing* [19]. Paige and Koenig [19] present derivatives for a first-order language with a fixed set of primitives. This work has inspired variants of finite differencing for queries on relational data, such as *algebraic differencing* [12], and *delta processing* [14].

However, most work in the database community is specialized to relational databases, hence does not support nested data (either nested collections, or algebraic data types). Incremental support is further designed monolithically for a whole language, rather than piecewise. The languages that are considered do not support first-class functions.

More general (non-relational) data types are considered in the work by Gluche et al. [11]; our support for bags and the use of groups is inspired by their work, but their architecture is still rather restrictive: they lack support for function changes and restrict incrementalization to self-maintainable views.

5.2.2 Static memoization

Liu [16]'s work allows to incrementalize a first-order base program $f(x_{\text{old}})$ to compute $f(x_{\text{new}})$, knowing how x_{new} is related to x_{old}. To this end, they transform $f(x_{\text{new}})$ into an incremental program which reuses the intermediate results produced while computing $f(x_{\text{old}})$, the base program. To this end, (i) first the base program is transformed to save all its intermediate results, then (ii) the incremental program is transformed to reuse those intermediate results, and finally (iii) intermediate results which are not needed are pruned from the base program. However, to reuse intermediate results, the incremental program must often be rearranged, using some form of equational reasoning, into some equivalent program where partial results appear literally. For instance, if the base program f uses a left fold to sum the elements of a list of integers x_{old}, accessing them from the head onwards, and x_{new} prepends a new element h to the list, at no point does $f(x_{\text{new}})$ recompute the same results. But since addition is commutative on integers, we can rewrite $f(x_{\text{new}})$ as $f(x_{\text{old}}) + h$. The author's CACHET system will try to perform such rewritings automatically, but it is not guaranteed to succeed. Similarly, CACHET will try to synthesize any additional results which can be computed cheaply by the base program to help make the incremental program more efficient.

Since it is hard to fully automate such reasoning, we move equational reasoning to the plugin design phase. A plugin provides general-purpose higher-order primitives for which the plugin authors have devised efficient derivatives (by using equational reasoning in the design phase). Then, the differentiation algorithm computes incremental versions of user programs without requiring further user intervention. It would be useful to combine IλC with some form of static caching to make the computation of derivatives which are not self-maintainable more efficient. We plan to do so in future work.

6. Conclusions and future work

We have presented IλC, an approach to lifting incremental computations on first-order programs to incremental computations on higher-order programs. We have presented a machine-checked correctness proof of a formalization of IλC and an initial experimental evaluation in the form of an implementation, a sample plugin for maps and bags, and a non-trivial example that was incrementalized successfully and efficiently.

Our work opens several avenues of future work. Our current implementation is not very efficient on derivatives that are not self-maintainable. However, as discussed (Sec. 3), we plan to investigate approaches to memoizing intermediate results to address this limitation. Our next step will be to develop language plugins which have efficient non-self-maintainable primitives.

Another area of future work is adding support for algebraic data types (including recursive types), polymorphism, subtyping, general recursion and other collection types. While support for algebraic data types could subsume support for specific collections, many collections have additional algebraic properties that enable faster incrementalization (like bags). Even lists (which have less algebraic properties) can benefit from special support [17].

Finally, we intend to perform a full and thorough experimental evaluation to demonstrate that IλC can incrementalize large-scale practical programs.

References

[1] U. A. Acar. *Self-Adjusting Computation*. PhD thesis, Princeton University, 2005.

[2] U. A. Acar. Self-adjusting computation: (an overview). In *PEPM*, pages 1–6. ACM, 2009.

[3] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. *TOPLAS*, 28(6):990–1034, Nov. 2006.

[4] U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. An experimental analysis of self-adjusting computation. *TOPLAS*, 32 (1):3:1–3:53, Nov. 2009.

[5] U. A. Acar, G. Blelloch, R. Ley-Wild, K. Tangwongsan, and D. Turkoglu. Traceable data types for self-adjusting computation. In *PLDI*, pages 483–496. ACM, 2010.

[6] Agda Development Team. The Agda Wiki. `http://wiki.portal.chalmers.se/agda/`, 2013. Accessed on 2013-10-30.

[7] A. W. Appel and T. Jim. Shrinking lambda expressions in linear time. *JFP*, 7:515–540, 1997.

[8] Y. Chen, J. Dunfield, M. A. Hammer, and U. A. Acar. Implicit self-adjusting computation for purely functional programs. In *ICFP*, pages 129–141. ACM, 2011.

[9] C. Elliott and P. Hudak. Functional reactive animation. In *ICFP*, pages 263–273. ACM, 1997.

[10] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *OOPSLA*, pages 57–76. ACM, 2007.

[11] D. Gluche, T. Grust, C. Mainberger, and M. Scholl. Incremental updates for materialized OQL views. In *Deductive and Object-Oriented Databases*, volume 1341 of *LNCS*, pages 52–66. Springer, 1997.

[12] A. Gupta and I. S. Mumick. Maintenance of materialized views: problems, techniques, and applications. In A. Gupta and I. S. Mumick, editors, *Materialized views*, pages 145–157. MIT Press, 1999.

[13] M. A. Hammer, G. Neis, Y. Chen, and U. A. Acar. Self-adjusting stack machines. In *OOPSLA*, pages 753–772. ACM, 2011.

[14] C. Koch. Incremental query evaluation in a ring of databases. In *Proc. Symp. Principles of Database Systems (PODS)*, pages 87–98. ACM, 2010.

[15] R. Lämmel. Google's MapReduce programming model — revisited. *Sci. Comput. Program.*, 68(3):208–237, Oct. 2007.

[16] Y. A. Liu. Efficiency by incrementalization: An introduction. *HOSC*, 13(4):289–313, 2000.

[17] I. Maier and M. Odersky. Higher-order reactive programming with incremental lists. In *ECOOP*, pages 707–731. Springer-Verlag, 2013.

[18] J. C. Mitchell. *Foundations of programming languages*. MIT Press, 1996.

[19] R. Paige and S. Koenig. Finite differencing of computable expressions. *TOPLAS*, 4(3):402–454, July 1982.

[20] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *POPL*, pages 502–510. ACM, 1993.

[21] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*, pages 127–136. ACM, 2010.

[22] G. Salvaneschi and M. Mezini. Reactive behavior in object-oriented applications: an analysis and a research roadmap. In *AOSD*, pages 37–48. ACM, 2013.

[23] D. Willis, D. J. Pearce, and J. Noble. Caching and incrementalisation in the Java Query Language. In *OOPSLA*, pages 1–18. ACM, 2008.