Stateful Differential Operators for Incremental Computing

RUNQING XU, KIT, Germany
SEBASTIAN ERDWEG, KIT, Germany

Differential operators map input changes to output changes and form the building blocks of efficient incremental computations. For example, differential operators for relational algebra are used to perform live view maintenance in database systems. However, few differential operators are known and it is unclear how to develop and verify new efficient operators. In particular, we found that differential operators often need to use internal state to selectively cache relevant information, which is not supported by prior work. To this end, we designed a specification for *stateful differential operators* that allows custom state, yet places sufficient constraints to ensure correctness. We model our specification in Rocq and show that the specification not only guides the design of novel differential operators, but also can capture some of the most sophisticated existing differential operators: database join and Datalog aggregation. We show how to describe complex incremental computations in OCaml by composing stateful differential operators, which we have extracted from Rocq.

CCS Concepts: • **Theory of computation → Design and analysis of algorithms**; **Semantics and reasoning**.

Additional Key Words and Phrases: Incremental computing, formal specification

ACM Reference Format:
Runqing Xu and Sebastian Erdweg. 2026. Stateful Differential Operators for Incremental Computing. *Proc. ACM Program. Lang.* 10, POPL, Article 86 (January 2026), 29 pages. https://doi.org/10.1145/3776728

1 Introduction

Incremental computing is concerned with programs that can react to input changes at run time. In particular, incremental computing is about *automatic* approaches that can incrementalize a family of programs. The following table summarizes the most important existing approaches:

Incremental-computing approach	Program family	Input change
incremental attribute grammars [Demers et al. 1981]	grammars with attribute definitions	AST changes
incremental build systems [Erdweg et al. 2015b]	build scripts	file changes
self-adjusting computations [Acar et al. 2006; Hammer et al. 2014]	functional programs	external variables
incremental lambda calculus [Cai et al. 2014; Giarrusso et al. 2019]	functional programs	function arguments
live view maintenance [Gupta and Mumick 1995]	relational algebra	relation changes
incremental Datalog [Pacak et al. 2022; Szabó et al. 2018]	logic programs with aggregation	relation and tree changes

Authors' Contact Information: Runqing Xu, KIT, Karlsruhe, Germany, runqing.xu@kit.edu; Sebastian Erdweg, KIT, Karlsruhe, Germany, erdweg@kit.edu.

ACM 2475-1421/2026/1-ART86
https://doi.org/10.1145/3776728

Incremental attribute grammars, incremental build systems, and self-adjusting computations perform *selective recomputing*: When an external input is changed, they rerun all sub-computations that transitively depend on that input. Selective recomputing has a fundamental limitation because it only knows if a variable is changed, but not how. That is, there only is a notion of *changed* (yes/no), but no notion of *change* (how). This is particularly problematic for computations that operate on complex data, where reacting to fine-grained changes is often asymptotically better than rerunning a sub-computation on the changed data.

The incremental lambda calculus (ILC) performs *differential updating*: When an input is changed, they propagate the input delta to derive the output delta. However, whenever the original computation uses a primitive operator, differential updating requires a corresponding *differential operator* to process the input change. For example, consider a grammar checker that analyzes text using a wide range of string operators such `trim`, `indexOf`, `substring`, or `toUpperCase`. As we propagate text changes, each operator (like `trim`) needs a corresponding differential operator (by convention called `Δtrim`) to propagate changes through the primitives. Indeed, the incremental performance substantially depends on the effectiveness of the available primitive differential operators. Unfortunately, virtually all known differential operators stem from incremental relational algebra and incremental Datalog, which employ powerful differential operators for relational data: projection, selection, join, and monotonic aggregation. It is these primitives that provide asymptotic speedups compared to the original non-incremental computation, which makes live view maintenance and incremental Datalog so effective in practice [Szabó et al. 2021]. If differential updating is to have a similar impact in other domains than relational algebra, we need to develop primitive differential operators to support them.

A key characteristic of many efficient differential operators is that they are stateful, that is, they retain information between incoming changes. For example, consider a differential operator `Δtrim` that removes leading and trailing whitespace from a string. Say we start with `trim("abc\n\n") = "abc"` and then append `"de"` to the input of `trim`, what is the corresponding output change? We must append `"\n\nde"` to the output, which includes the deleted whitespace, because it is no longer trailing in the current input `"abc\n\nde"`. Thus, `Δtrim` must remember which whitespace it deleted, so that it can be restored when required. As one investigates differential operators, it quickly becomes apparent that statefulness is the rule rather than the exception. Only a few studies have explored stateful differential operators [Budiu et al. 2023; Giarrusso et al. 2019], and they impose restrictions on the state's representation, which limits expressiveness and efficiency. In particular, no prior research described how to develop differential operators with custom states or what it means for them to be correct.

In this paper, we present the first framework for developing efficient differential operators. At it's heart, our framework comprises a specification of stateful differential operators that guides their development and ensures their correctness. Indeed, since differential operators try to cut corners to reduce work whenever possible, there is an imminent risk of breaking correctness. A differential operator Δf is correct if the derived output change correctly maps the old output $f(x)$ to the new output $f(x \oplus \Delta x)$. For stateless differential operators, we must prove $f(x \oplus \Delta x) = f(x) \oplus \Delta f(\Delta x)$. However, stateful differential operators are more complicated: they read and update an internal state. To this end, our specification allows differential operators to define an invariant $I(x, s)$. A stateful differential operator then is correct if it yields a correct output change and the invariant is maintained between incoming changes:

$$\begin{array}{rrl} & I(x,s) & \rightarrow \quad f(x \oplus \Delta x) = f(x) \oplus \Delta f(s, \Delta x) \\ \wedge & I(x,s) & \rightarrow \quad I(x \oplus \Delta x, s'). \end{array}$$

We have formalized our framework for stateful differential operators in the proof assistant Rocq and present the full details in this paper. We demonstrate the expressiveness and the utility of our framework with operators for two case studies (stateful operators are marked '*'):

Differential database operators: union, select, product, join*, user-defined aggregation*
Differential string operators: length, isEmpty*, toUpperCase, toLowerCase, trimLeft*, trimRight*, concat*, indexOf*, lastIndexOf*

Many operators require custom state that we can model and validate with our framework. To the best of our knowledge, we provide the first mechanized proof for incremental user-defined aggregation implemented in IncA [Szabó et al. 2018]. Moreover, all of the differential string operators are novel and many string operators yield asymptotic speedups, because they avoid the linear scan required by a recomputation. In the long run, we hope to establish a full standard library of provably correct differential operators for incremental computing.

While this work focuses on the theoretical aspects of differential operators, it is important to stress their practical relevance. In particular, we can construct complex incremental computations by composing (stateful) differential operators in a general-purpose language. Specifically, we extracted various differential operators from Rocq to OCaml and use modules to encapsulate the state of stateful differential operators. We provide an API for constructing networks of connected differential operators and show how to use it to model incremental relational queries and incremental string processing. Our empirical experiments confirm the asymptotic speedups.

In summary, we make the following contributions in this paper:

(1) We identify that efficient differential operators need user-defined state, which is not supported by self-adjusting computations or the incremental lambda calculus (Section 2).
(2) We define an expressive structure of changes, allowing us to precisely model the behavior of first-order and higher-order changes (Section 3).
(3) We design a specification for stateless and stateful differential operators and present a framework in Rocq for their development (Section 4 and Section 5).
(4) We develop various novel differential operator for strings and demonstrate the generality of our framework by modeling and verifying two of the most sophisticated differential operators known, database join and user-defined aggregation (Section 6).
(5) We show how to construct complex incremental computations in OCaml using our framework of differential operators and we measure their incremental performance (Section 7).

The artifact for our Rocq formalization and experiments can be found at https://zenodo.org/records/17428578.

2 Motivating Examples and the State of the Art

The goal of this paper is to establish a framework for developing and reasoning about differential operators that can be used in incremental computing. In this section, we discuss two motivating examples from relational algebra: the selection operator σ and the join operator $\bowtie$. While the differential behavior of these operators is well-known, we use these examples to explain (i) the limitations of prior approaches to incremental computing and (ii) why stateful differential operators are necessary. But first, we introduce a bit of terminology and notation for relations and relation changes.

We are interested in differential operators whose inputs are subject to change. We write $x \oplus \Delta x$ to denote the patching of input x with a change Δx (formally defined in Section 3). For relational operators, inputs can change in two fundamental ways: we can add new tuples or we can remove existing tuples. Thus, $\Delta x = \mathit{ins}\ S$ or $\Delta x = \mathit{del}\ S$ for some set of tuples S. Moreover, we define

$R \oplus \mathit{ins}\, S = R \cup S$ and $R \oplus \mathit{del}\, S = R \setminus S$. The challenge then is to define efficient differential operators that map an input change Δx to an output change Δy.

2.1 Selection – a stateless differential operator

The selection operator $\sigma_f(R)$ selects those tuples from relation R that satisfy predicate f. When new tuples S are added to the input $R \oplus \mathit{ins}\, S$, we can update the result of selection incrementally because $\sigma_f(R \oplus \mathit{ins}\, S) = \sigma_f(R) \ \oplus \ \mathit{ins}\, \sigma_f(S)$. Indeed, we can define a differential operator by setting $\Delta\sigma_f(\mathit{ins}\, S) = \mathit{ins}\, \sigma_f(S)$ and $\Delta\sigma_f(\mathit{del}\, S) = \mathit{del}\, \sigma_f(S)$. This differential operator is stateless because it only takes the change as input but does not require other information. It takes time linear in the size of S, but is independent of the size of R, which is an asymptotic improvement. This corresponds to the definition of incremental selection in real-world database systems.

Now, let us investigate if this differential operator can be integrated into existing approaches to incremental computing. Frameworks that perform selective recomputing such as Adapton [Hammer et al. 2014] do not have a notion of change; they only have a notion of changed (yes/no). Therefore, it is not possible to integrate differential operators that read and produce changes. The best we can do is to wrap our differential operator in a function that takes the old input R and new input R':

$$\sigma'_f(R, R') = \sigma_f(R) \oplus \Delta\sigma_f(R' \ominus R).$$

But this incurs a considerable overhead due to diffing $\ominus$ and patching $\oplus$.[1] Alternatively, we can forgo differential computing and use selective recomputing for operators as well:

$$\sigma'_f(R, R') \quad = \quad \text{if } R = R' \text{ then } \sigma_f(R) \text{ else } \sigma_f(R').$$

That is, we do not use a differential operator at all, but selectively rerun σ_f on R' when it differs from the previous input R. However, the comparison of the inputs takes time and even a slight change of the input results in a complete recomputation. Therefore, both these integrations take time linear in $|R|$, which is unacceptable for an efficient incremental computation.

Let us consider the Incremental Lambda Calculus (ILC) [Cai et al. 2014] instead. The ILC framework derives differential programs in a type-driven manner. Specifically, since our selection operator σ_f has type $\mathsf{Rel} \to \mathsf{Rel}$, the differential operator $\Delta\sigma_f$ has type $\mathsf{Rel} \to \Delta\mathsf{Rel} \to \Delta\mathsf{Rel}$ in ILC. That is, the differential operator obtains two inputs: the original input and its change. This typing discipline is elemental to ILC, because it transforms the rest of the program in accordance to this interface. And it is also quite flexible: we can embed our differential operator in ILC by ignoring the first argument:

$$\lambda R.\ \ \lambda \Delta R.\ \ \Delta\sigma_f(\Delta R) \quad : \quad \mathsf{Rel} \to \Delta\mathsf{Rel} \to \Delta\mathsf{Rel}.$$

However, we pay a price for the unused parameter: Code that uses our differential operator has to compute the original input. In particular, most applications of incremental computing will change the input multiple times in sequence $\Delta R_1, \ldots, \Delta R_n$ after an initial input R_0. For each change, we then have to compute and store $R_n = R_{n-1} \oplus \Delta R_n$, even though this data is never used. While it may be possible to rely on lazy evaluation or other optimizations to eliminate the overhead, it would be better to avoid this overhead altogether. Besides, the original input is often insufficient for stateful differential operators as the following example shows.

2.2 Join – a stateful differential operator

The binary join operator $P \bowtie R$ combines relations P and R: For each tuple in P, the join operator finds all matching tuples in R and concatenates them. The exact matching criteria varies for different

[1] Diffing and patching can be optimized. For example, Incr_map uses immutable data structures with substructural sharing to reduce this overhead [Jane Street and OCaml community 2024].

kinds of joins (natural, equi, etc.), but does not matter for our discussion. When new tuples are added $R \oplus ins\, S$, we can update the result of the join incrementally because

$$P \bowtie (R \oplus ins\, S) = P \bowtie R \;\oplus\; ins\, (P \bowtie S).$$

That is, we can derive the join result for the new tuples S separately from the join of the original inputs. However, we need extra information: We need to remember the tuples in P to match against the new tuples in S. Then we can define $P \bowtie_\Delta ins\, S = ins\, (P \bowtie S)$ and $P \bowtie_\Delta del\, S = del\, (P \bowtie S)$ to react to changes of R. Analogously, we can define ${}_\Delta\!\bowtie$ to react to changes of P. These differential operators ${}_\Delta\!\bowtie$ and $\bowtie_\Delta$ are stateful, because in order to process the changes of one relation they need to remember the contents of the other relation.

The ILC framework supports such differential join operators gracefully. The original join operator has type $\mathsf{Rel}_P \times \mathsf{Rel}_R \to \mathsf{Rel}_{P \cdot R}$. That is, it takes a pair of relations and yields a relation that concatenates tuples from P and R. In the ILC framework, we can embed the differential join operators as follows:

$$\begin{aligned}
\lambda P.\, \lambda \Delta R.\quad P \bowtie_\Delta \Delta R \quad &: \quad \mathsf{Rel}_P \to \Delta\mathsf{Rel}_R \to \Delta\mathsf{Rel}_{P \cdot R} \\
\lambda R.\, \lambda \Delta P.\quad \Delta P \;{}_\Delta\!\bowtie\; R \quad &: \quad \mathsf{Rel}_R \to \Delta\mathsf{Rel}_P \to \Delta\mathsf{Rel}_{P \cdot R} \\
\lambda\,(P, R).\, \lambda\,(\Delta P, \Delta R).\;\; (P \bowtie_\Delta \Delta R) +\!\!+ (\Delta P \;{}_\Delta\!\bowtie\; (R \oplus \Delta R)) \quad &: \quad \mathsf{Rel}_P \times \mathsf{Rel}_R \to \\
&\qquad\quad \Delta(\mathsf{Rel}_P \times \mathsf{Rel}_R) \to \Delta\mathsf{Rel}_{P \cdot R}
\end{aligned}$$

The last function illustrates the full differential join operator that accepts changes to both input relations. To this end, we concatenate (++) the changes produced by ${}_\Delta\!\bowtie$ and $\bowtie_\Delta$ (formal details follow later). It looks like the ILC framework is expressive enough to capture stateful differential operators. But there is a severe limitation.

The ILC framework supports exactly one kind of state for differential operators: the original input. Unfortunately, this precludes a truly efficient implementation of the differential join operators. An efficient differential join operator $\bowtie_\Delta$ needs to maintain an index Ix^P into the original relation P such that it can perform constant-time lookups when tuples are inserted or removed from the other relation R. That is, instead of $P \bowtie_\Delta ins\, S = ins\, (P \bowtie S)$, we need to compute $P \bowtie_\Delta ins\, S = ins\, (Ix^P \bowtie S)$. This improves the asymptotic running time from $P \cdot S$ to $m \cdot S$, where $m \ll P$ is the number of matches for a tuple in the index. Indeed, this is the version of the differential join operator that real-world database systems implement.

The ILC framework does not allow differential operators to define and maintain user-defined state. It is also not clear what the formal specification of such stateful operators should look like and how to reason about them. Our paper answers these questions and introduces a framework for differential operators with user-defined state. We then use our framework to realize a number of existing and novel differential operators for strings and relations that define custom states to improve their incremental efficiency.

3 A Formal Structure of Changes

Differential operators receive and produce changes of data. That is, changes are first-class values. In order to build a solid theory for differential operators, we must first define the formal structure of changes.

3.1 Change structures

A change Δx is a value that describes how data is modified. Conceptually, the semantics of a change is to transform data $[\![\Delta x]\!] : \mathsf{T} \to \mathsf{T}$. This justifies the conventional patching notation $x \oplus \Delta x$, which we define as $[\![\Delta x]\!]\, x$. Unfortunately, this simple design is not well-founded because $[\![\Delta x]\!]$ is often not a total function.

For example, consider changes of the natural number *inc k* and *dec k*, which increment and decrement by k, respectively. Intuitively, we would define $[\![dec\ k]\!] n = n - k$, but what if $n < k$? Of course, we could require some default behavior, but (i) this breaks certain algebraic properties such change commutativity and (ii) similar issues arise for many other change descriptors. For example, we also get undefined behavior for removing an element from a bag that does not exist, modifying a list at an index that does not exist, and so on. Instead of relying on default behavior, we require the definition of changes to explicitly declare for which data a change is valid, and the semantics must be total for valid changes.

We formalize changes and differential operators in Rocq and present a slightly prettified version of the Rocq code in the paper. We model changes as a Rocq structure (think: record type):

```
Structure change T : Type := {
  ΔT : Type ;
  vc : ΔT → T → Prop ;
  patch : ∀ (Δt : ΔT) (t : T), vc Δt t → T
}.
Notation "⟦ Δx ⟧" := (patch Δx).
Notation "t ⊕ Δt" := (patch Δt t _).
```

A `change` for type `T` consists of the type of change values `ΔT`, the valid-change relation `vc` between `ΔT` and `T`, and the `patch` function. The `patch` function has three parameters:[2] a change `Δt`, a value `t` to be patched, and proof of type `vc Δt t` to witness that the change is valid for the given value. The result of patching is an updated value of type `T`. Lastly, we define some notation for convenience. In particular, we write `t⊕Δt` to let Rocq infer the validity proof.[3]

For example, we can define the change structure for natural numbers and bags as follows:

```
Inductive nat_change : Type :=
  | nat_inc : nat → nat_change
  | nat_dec : nat → nat_change.
Definition natc : change nat := {|
  ΔT := nat_change;
  vc Δn n := match Δn with
    | nat_inc k ⇒ True
    | nat_dec k ⇒ k ≤ n
    end;
  patch Δn n _ := match Δn with
    | nat_inc k ⇒ n + k
    | nat_dec k ⇒ n - k
    end
|}.
```

```
Variable T : Type.
Inductive bag_change : Type :=
  | bag_ins : bag T → bag_change
  | bag_del : bag T → bag_change.
Definition bagc : change (bag T) := {|
  ΔT := bag_change;
  vc Δb b := match Δb with
    | bag_ins s ⇒ True
    | bag_del s ⇒ s ⊆ b
    end;
  patch Δb b _ := match Δb with
    | bag_ins s ⇒ union_all b s
    | bag_del s ⇒ diff_all b s
    end
|}.
```

A nat change is an increment or decrement, increments are always valid, but decrements are only valid if the patched number is large enough. Under these conditions, patch is well-defined and does not need to rely on default behavior of subtraction. Similarly, a bag change restricts validity of deletions to those tuples that are actually in the bag.

Our formalization of changes incorporates some subtle design decisions that we want to elaborate on. In particular, readers familiar with the ILC framework may recall their change structure $(T, \Delta t, \oplus, \ominus)$. There are two important differences to our change structure. First, ILC uses a dependent type Δt for changes, where $dt \in \Delta t$ implies dt is a valid change for t. The problem with this

[2]Note that `∀(x:X), T` represents a dependent function type in Rocq. That is, it behaves like a function type `X→ T`, but `T` may vary with the argument value `x`.

[3]In the paper, we freely write `t⊕Δt` even when Rocq fails to infer the validity proof `vc Δt t` automatically. In those places, our implementation provides the witness explicitly.

design is that we need the original value t to even formulate the type signature of a differential operator. That is, instead of $\Delta f : \Delta X \to \Delta Y$, we would have to specify $\Delta f : (x : X) \to \Delta x \to \Delta(f(x))$. That is, the input change is valid for the original input and the resulting change is valid for the original function result. This may make sense for the ILC, where the original input is considered available state anyways, but we do not want to carry that burden. Therefore, we decided to externalize the validity property into its own definition `vc` and to prove change validity separately.

The second difference is that ILC includes a difference operation $\ominus : (x : T) \to (y : T) \to \Delta y$, which computes the change between two values. We omit $\ominus$ because (i) it is not unique, (ii) does not always exist, and ultimately (iii) it is not necessary. A difference operation is not unique for complex data types such as strings, where various diffing algorithms exist. For other data types, a difference operation does not even exist. For example, the bag difference of $\{1, 2\}$ and $\{2, 3\}$ cannot be described using `bag_change` from above, because it requires both an insertion and a deletion simultaneously. Rather than inflating change structures to be more expressive, we add higher-order change structures that compose more primitive changes (see below). Finally, the difference operator is not needed for the definition of differential operators. Instead, it is mainly relevant to produce the initial change for incremental computing, and as a backup when incremental computing fails to construct the resulting change precisely. To this end, we provide an interface for diffing operations that can be implemented optionally:

```
Structure difference T : Type := {
  C : change T;
  diff : T → T → ΔT C    where "new ⊖ old" := (diff old new);
  valid_diff : forall t1 t2, vc (t2 ⊖ t1) t1
  patch_diff : forall t1 t2, t1 ⊕ (t2 ⊖ t1) = t2
}.
```

A difference operation for a given type `T` requires a change structure `C` for `T`, the diffing implementation (`ΔT C` accesses the change type `ΔT` in `C`), and two properties: The change `t2⊖t1` is valid for `t1` and it indeed patches `t1` to `t2`. We use the difference operator in one of our larger case studies.

3.2 Higher-order change structures

So far, we have considered atomic changes that patch data in a single step. But we can also construct complex changes that (i) change compound data types or (ii) change the same data multiple times. We can generically describe such complex changes in higher-order change structures.

A higher-order change structure is parametric in one or more underlying change structures. This is useful for describing changes of compound data types. For example, consider a pair `A * B`, where we may change only `A`, only `B`, or both. We can define a generic change structure for such pairs given change structures for `A` and `B`:

```
Variable (A : Type) (B : Type).
Variable (CA : change A) (CB : change B).
Inductive pair_change : Type :=
  | pair_fst  : ΔT CA → pair_change
  | pair_snd  : ΔT CB → pair_change
  | pair_both : ΔT CA → ΔT CB → pair_change.
Definition pairc : change (A * B) := {|
  ΔT := pair_change;
  vc c p := match c with
    | pair_fst ca ⇒ vc ca (fst p)
    | pair_snd cb ⇒ vc cb (snd p)
    | pair_both ca cb ⇒ vc ca (fst p) ∧ vc cb (snd p)
    end;
  patch c p _ := match c with
```

```
    | pair_fst ca ⇒ (fst p ⊕ ca, snd p)
    | pair_snd cb ⇒ (fst p, snd p ⊕ cb)
    | pair_both ca cb ⇒ (fst p ⊕ ca, snd p ⊕ cb)
    end
|}.
Arguments pairc {A B}.  (* infer the first two type arguments *)
```

The change structure for pairs delegates to the change structures `CA` and `CB`. Note that `ΔT CA` accesses the field `ΔT` of `CA`, which is the change descriptor type. Pair changes play a significant role for differential operators with more than one input. For example, the differential binary join operator from Section 2 reads changes from `pairc (bagc A) (bagc B)`.

Another common scenario is the construction of change sequences, where a single piece of data is changed multiple times. We have seen such a scenario in Section 2: The join operator yields multiple result changes when both input relations change simultaneously. We represent change sequences as lists of changes from a given change structure `C`:

```
Variable T : Type.
Variable C : change T.
Definition seq_change := list (ΔT C).
Inductive seq_vc : seq_change → T → Prop :=
  | seq_vc_nil : forall t, seq_vc nil t
  | seq_vc_cons : forall Δhd Δtl t,
    vc Δhd t →
    seq_vc Δtl (t ⊕ Δhd) →
    seq_vc (Δhd :: Δtl) t.
Program Fixpoint seq_patch Δt t (vc : seq_vc Δt t) : T :=
  match Δt with
  | nil ⇒ t
  | Δhd::Δtl⇒ seq_patch Δtl (t ⊕ Δhd) _
  end.
Definition seqc : change T := {|ΔT := seq_change; vc := seq_vc; patch := seq_patch|}.
Arguments seqc {T}.  (* infer the first type argument *)
```

A change sequence is valid if each contain change can be applied in order. In particular, `seq_vc_cons` requires that `Δtl` is valid for `t⊕Δhd`, that is, after applying change `Δhd`. Function `seq_patch` exploits this property to patch the individual changes, one after the other. The resulting change structure `seqc` represents sequential changes of `T` for any underlying `C` that also changes `T`. Indeed, `seqc C` and `C` both have type `change T`: A differential operator can freely choose whether to describe a single change or a sequence of changes. For example, the differential join operator from Section 2 produces changes from `seqc (bagc (A * B))`, which allows it to concatenate changes. Moreover, any differential operator can be applied to sequences of changes transparently, a property which is often missing in prior work.

4 Stateless Differential Operators

With our formal model of changes in place, we are ready to formalize differential operators. We start with stateless differential operators like the relational selection operator from Section 2. Conceptually, stateless differential operators are a special case of stateful operators. However, it is more efficient to leave out the state when it is not needed. Moreover, it is instructive to study the simpler stateless case first.

We model differential operators in Rocq using modules. Hence, we define module types to list the required definitions and properties that each differential operator must provide. In a module type, `Parameter` declarations denote required definitions and their type, whereas `Axiom` declarations

denote required properties. Stateless differential operators are defined through the following module type:

```
Module Type StatelessDiffOp.
  Parameter (A : Type) (B : Type).
  Parameter f : A → B.

  Parameter (CA : change A) (CB : change B).
  Parameter Δf : ΔT CA → ΔT CB.

  Axiom inc_valid   : ∀ x Δx, vc Δx x → vc (Δf Δx) (f x).
  Axiom inc_correct : ∀ x Δx, vc Δx x → f (x ⊕ Δx) = f x ⊕ Δf Δx.
End StatelessDiffOp.
```

A stateless differential operator incrementalizes a given function `f` from `A` to `B`. Therefore, the differential operator `Δf` receives changes from a change structure `CA` and yields changes from a change structure `CB`. Any given definition of `Δf` must satisfy two properties. First, for any valid input change, the output change produced by `Δf` must be valid with respect to the original output `f x`. That is, it must be possible to use this change and patch `f x`. Second, when we do patch the previous result `f x ⊕ Δf Δx`, we must obtain the same result produced by a recomputation `f (x ⊕ Δx)`. This is the correctness property of stateless differential operators, sometimes called from-scratch consistency in the literature [Hammer et al. 2015].

We can now cast the selection operator as a stateless differential operator:

```
Module SelectOp <: StatelessDiffOp.
  Parameter T : Type.
  Parameter cond : T → bool.
  Definition A := bag T.   Definition B := bag T.
  Definition f (x : bag T) : bag T := select cond x.

  Definition CA := bagc T. Definition CB := bagc T.
  Definition Δf (Δx : bag_change T) : bag_change T :=
    match Δx with
    | bag_union b ⇒ bag_union (select cond b)
    | bag_diff  b ⇒ bag_diff (select cond b)
    end.

  (* omitted: lemmas that prove inc_valid and inc_correct *)
End SelectOp.
```

The differential select operator is parametric in the type `T` contained in the bag and the predicate `cond`. It implements module type `StatelessDiffOp` and hence must define all declared parameters and prove all declared properties of the module type. We set the operator's input and output types `A` and `B` to `bag T`, and define `f` to be the regular non-incremental selection on bags. For `CA` and `CB`, we can select any change structure of type `change (bag T)`; here we simply use `bagc T`. We can then define the differential operator `Δf` to convert input changes into output changes. Finally, we have to prove that our definitions satisfy `inc_valid` and `inc_correct`, otherwise the module fails to satisfy its module type and is rejected by Rocq.

Before turning to stateful differential operators, we want to emphasize the flexibility of our approach. In our framework, many change structures of type `change T` co-exist, and we can select the best fit. We can use this to solve an understudied issue in prior work: how an incremental computation reacts to a sequence of input changes. For example, the ILC framework requires we compute $x \oplus \Delta x_1$ after each change Δx_1, because this is the basis for processing the next change Δx_2. In our framework, we can generically lift a differential operator to sequences of changes:

```
Module Type StatefulDiffOp.
  Parameter (A : Type) (B : Type).
  Parameter (CA : change A) (CB : change B).
  Parameter f : A → B.

  (* state declarations *)
  Parameter ST : Type.
  Parameter init : A → ST.
  Parameter vs : ΔT CA → ST → Prop.
  Parameter Δst : ∀ (Δx : ΔT CA) (st : ST), vs Δx st → ST.

  (* state invariant and properties *)
  Parameter inv : A → ST → Prop.
  Axiom inv_init : ∀ x, inv x (init x).
  Axiom inv_step : ∀ x Δx st,  inv x st → vc Δx x → inv (x ⊕ Δx) (Δst Δx st _).
  Axiom state_valid : ∀ x Δx st,  inv x st → vc Δx x → vs Δx st.

  (* diff op declaration and properties *)
  Parameter Δf : ∀ (Δx : ΔT CA) (st : ST), vs Δx st → ΔT CB.
  Axiom inc_valid   : ∀ x Δx st, inv x st → vc Δx x → vc (Δf Δx st _) (f x).
  Axiom inc_correct : ∀ x Δx st, inv x st → vc Δx x →
                                f (x ⊕ Δx) = f x ⊕ Δf Δx st _.
End StatefulDiffOp.
```

Fig. 1. Signature of the stateful differential operator.

```
Module SeqStatelessDiffOp (op : StatelessDiffOp) <: StatelessDiffOp.
  Definition A := op.A.          Definition B := op.B.          Definition f := op.f.
  Definition CA := seqc op.CA.   Definition CB := seqc op.CB.
  Fixpoint Δf (xs : list (ΔT op.CA)) : list (ΔT op.CB) :=
    match xs with
    | nil ⇒ nil
    | Δhd :: Δtl ⇒ op.Δf Δhd :: Δf Δtl
    end.
  (* omitted: lemmas that prove inc_valid and inc_correct *)
End SeqStatelessDiffOp.
```

This differential operator takes another operator `op` as input. It then defines a differential operator with the same original function `f : A → B`. However, the change structures `CA` and `CB` are different: We accept and produce a sequence of changes. This is proof that any stateless differential operator can be used to process sequences of changes correctly.

5 Stateful Differential Operators

Stateless differential operators are the exception, not the rule. Indeed, many differential operators must maintain state in order to react to input changes efficiently. A simple example of a stateful differential operator is the differential square operation. When the original input n changes by k, we can compute the updated output as $(n + k)^2 = n^2 + 2nk + k^2$. Relative to the old output n^2, the output change is $2nk + k^2$. To compute that change, we need to know not only the increment k but also the previous input n. Stateful differential operators can keep track of such information.

5.1 A formal model for stateful differential operators

We model stateful differential operators as module types in Rocq. Figure 1 shows the module type, which we explain in this subsection.

Each stateful differential operator can declare its own state type `ST`. Indeed, what type of state to use becomes a primary design consideration for stateful differential operators. A well-designed differential operator must carefully balance the effort required for maintaining its state with the benefit the state brings when reacting to input changes. Ideally, the maintenance of the state has a constant overhead while enabling asymptotic speedups for input changes. While the differential square operation promises little benefit, the differential join operator from Section 2 leads to asymptotic speedups.

Besides declaring their state type `ST`, stateful differential operators must provide two functions that produce state. Initially, the state is a projection of the original input as computed by function `init`. Then, with each input change, the state gets updated by function `Δst` to support the processing of subsequent changes. This way, for any sequence of previously seen input changes $x \oplus \Delta x_0 \oplus \Delta x_1 \oplus \Delta x_2$, we obtain the current state `Δst Δx₂ (Δst Δx₁ (Δst Δx₀ (init x)))`. However, not all states are compatible with all input changes. Therefore, we also declare a valid-state property `vs` that restricts the applicability of `Δst` to valid changes `Δx`.

In order to reason about the current state, stateful differential operators can declare a state invariant `inv`. The state invariant `inv x st` relates the current state `st` to the current input `x`.[4] We require that the initial state satisfies the invariant (`inv_init`) and `Δst` maintains the invariant for any valid input change (`inv_step`). Moreover, the invariant must be sufficient to ensure each valid input change is also valid for the current state (`state_valid`).

Finally, we declare a differential operator `Δf`. Compared to its stateless counterpart, the stateful `Δf` obtains not only an input change `Δx`, but also the current state `st`, which must be compatible with `Δx`. From this information, `Δf` must compute the corresponding output change according to change structure `CB`. The differential operator `Δf` has to adhere to validity and correctness like before, but may assume that the invariant holds for the current state.

In the remainder of this section, we illustrate the formal model of stateful differential operators through two examples. Besides illustrating the definition, the examples also help clarify how exactly our model generalizes the ILC framework.

5.2 Stateful differential operators subsume ILC operators

Operators from the Incremental Lambda Calculus (ILC) always use the current input as their state. We can describe this behavior as a special case of our stateful differential operators. We first define a specialized version `InputDiffOp` of `StatefulDiffOp`, where `Δf` takes an input change and the current input (instead of the current state), adapt axioms `inc_valid` and `inc_correct` accordingly (not shown). We then define a functor `InputStateDiffOp` that implements a `StatefulDiffOp` given the simpler `InputDiffOp`:

```
Module Type InputDiffOp.
  (* excerpt from StatefullDiffOp: Parameters A, B, CA, CB, f.  Axioms inc_valid,
    inc_correct. *)
  Parameter Δf : ∀ (Δx : ΔT CA) (x : A), vc Δx x → ΔT CB.
End InputDiffOp.
Module InputStateDiffOp (op : InputDiffOp) <: StatefulDiffOp.
  (* delegates to op: Definitions A, B, CA, CB, f, Δf.*)
  Definition ST : Type := A.
  Definition init : A → ST := id.
  Definition vs : ΔT CA → ST → Prop := vc.
  Definition Δst : forall Δx st, vs Δx st → ST := patch.
  Definition inv : A → ST → Prop := eq.
```

[4] Importantly, the current input only occurs in the formalization of the invariant, but not in the differential computation `Δf`, which only interacts with the state. Thus, the state type determines the efficiency of the differential operator.

```
  (* omitted: proofs of the following three lemmas *)
  Lemma inv_init : ∀ x, inv x (init x).
  Lemma inv_step : ∀ x Δx st,  inv x st → vc Δx x → inv (x ⊕ Δx) (Δst Δx st _).
  Lemma state_valid : ∀ x Δx st,  inv x st → vc Δx x → vs Δx st.
  (* Lemmas inc_valid and inc_correct proved by op.inc_valid and op.inc_correct. *)
End InputStateDiffOp.
```

In `InputStateDiffOp`, we define the state type `ST` to be identical to the input type `A`: The initial state is the same as the initial input, hence we use the identity function to compute it. An input change is valid for the current state if it is valid for the current input, and the next state is computed by patching the current input. We maintain a simple invariant, namely that the state is equal to the current input. This is sufficient to prove `inv_init`, `inv_step`, and `state_valid` independent of the given `InputDiffOp`. Hence, it suffices to implement `InputDiffOp` and to prove `inc_valid` and `inc_correct` when using the current input as current state. These are the same properties required by the ILC framework. For example, we can define a differential square operator as an `InputDiffOp`:

```
Module SquareInputOp <: InputDiffOp.
  Definition A := nat.   Definition B := nat.
  Definition CA := natc.  Definition CB := natc.
  Definition f x := x * x.
  Definition Δf (Δx : ΔT CA) (x : nat) (_ : vc Δx x) : ΔT CB :=
    match Δx with
    | nat_add k ⇒ nat_add (2 * k * x + k * k)
    | nat_minus k ⇒ nat_minus (2 * x * k - k * k)
    end.
  (* omitted: lemmas inc_valid and inc_correct *)
End SquareInputOp.
Module SquareOp := InputStateDiffOp(SquareInputOp).
```

The last line applies functor `InputStateDiffOp` to construct a provably correct stateful differential operator `SquareOp` from `SquareInputOp`. However, as explained in Section 2, efficient differential operators often need to maintain custom state beyond the current input. Differential operators with custom state are not expressible in the ILC, as we detail in related work (Section 9).

5.3 Using custom state in differential operators

Our model of differential operator extends the state of the art by allowing operators to define custom state. This can have crucial effects on the operator's incremental performance, as was the case for the differential join operator from Section 2. In this section, we present a simpler example first: An efficient differential operator for trimming right-bound whitespace from a string. Usually, when a string is updated, trimming the right-bound whitespace takes linear time, since it requires a full scan of the string. But, our differential operator can compute an output change in constant time by keeping track of two pieces of information in its state: (i) how much right-bound whitespace exists in the current string and (ii) whether are all characters of the string whitespace.

We show the definition of `TrimRightOp` in Figure 2. The state is a pair of `nat` and `bool`, initialized by function `init`, which scans the initial string once. Subsequently, we can maintain the state in constant time using function `Δst`. The definition of `Δst` is not too complicated, but there is one important corner case: When a string is all whitespace, a whitespace prepend is equivalent to a whitespace append, and hence we must increment the whitespace count. Such corner cases are easy to miss without a formal correctness proof; lemma `inv_step` would have caught any mistake here. Also note that the trim-right operator has a simple invariant `st = init s`, which works because the state has a canonical representation captured by `init`. In general, `Δst` may construct states different from `init` as long as both satisfy the invariant.

```
Module TrimRightOp <: StatefulDiffOp.
  Definition A := string.    Definition B := string.
  Definition CA := stringc.  Definition CB := seqc stringc.
  Definition f s := trim_right s.

  (* number of right-bound whitespaces and if the string is only whitespace *)
  Definition ST : Type := nat * bool.
  Fixpoint init (s : string) : ST :=
    match s with
    | nil ⇒ (0, true)
    | c :: tl ⇒ let (ws, all_ws) := init tl in
      if all_ws
      then (if c =? " " then (S ws, true) else (ws, false))
      else (ws, false)
    end.
  Definition inv (s : string) (st : ST) := st = init s.
  Definition Δst (Δt : ΔT CA) (st : ST) (_ : vs Δt st) : ST :=
    let '(ws, all_ws) := st in
    match Δt with
    | str_prepend c ⇒
      if c =? " " then
        if all_ws then (S ws, all_ws) else (ws, all_ws)
      else
        (ws, false)
    | str_append c ⇒ if c =? " " then (S ws, all_ws) else (0, false)
    end.
  Definition Δf (Δt : ΔT CA) (st : ST) (_ : vs Δt st) : ΔT CB :=
    let '(ws, all_ws) := st in
    match Δt with
    | str_prepend c  ⇒
      if c =? " " then
        if all_ws then [] else [str_prepend " "]
      else
        [str_prepend c]
    | str_append c ⇒
      if c =? " " then
        []
      else
        (list_mul ws (str_append " ")) ++ [str_append c]
    end.
  (* omitted all lemmas. inv_step and inc_correct are the key properties here. *)
End TrimRightOp.
```

Fig. 2. A stateful differential operator for right trimming a string, using a custom state for asymptotic speedup.

We compute the output change with function `Δf`, which has to observe the same corner case. Moreover, when appending a non-whitespace character, we also have to append all previously trimmed whitespace. For example, `Δf (str_append "d") (init "abc␣␣")` must compute `[str_append "␣"; str_append "␣"; str_append "d"]`, where we use `"␣"` to denote a single whitespace character. Hence, patching `f "abc␣␣" = "abc"` yields the expected result `"abc␣␣d"`, as required by `inc_correct`.

To the best of our knowledge, a differential trim-right operator was not known before this paper, let alone a formal proof of its correctness. Our framework of stateful differential operators opens the door for an extensive exploration of such operators, for strings and other data types. We envision

to collect such operators in a standard library for incremental computing that will form the basis for efficient incrementality in many different domains.

6 Case Studies

To demonstrate that our model of stateful differential operators is expressive and scales to real-world needs, we use it to model two of the most complex differential operators known: database join and user-defined aggregation. The differential join operator is at the heart of live view maintenance in database systems in practice, but we present the first framework for differential operators that can capture and validate its behavior. Differential aggregation with user-defined functions was discovered more recently [Szabó et al. 2018] and is used in incremental Datalog systems, where it enables the incremental whole-program data-flow analysis of large software projects [Szabó et al. 2021]. We faithfully cast these operations as stateful differential operators and prove them correct. We then move on to develop novel differential string operators, many of which require custom state.

6.1 Differential database join

As elaborated in Section 2.2, efficient differential join operations need to maintain an index of the two input relations. Note that many different join variants exist in the literature, which differ in how to select matching tuples. We chose to implement an equi-join operator that combines tuples based on key extraction. That is, given a `bag L` and a `bag R`, we extract keys of type `K` using extraction functions `lk : L→ K` and `rk : R→ K`. The join operation then yields a `bag (L*R)` where tuples are composed that mapped to the same key. We applied our framework of stateful differential operators to model this differential join operator as shown in Figure 3 and proved it correct in Rocq.

First, let us inspect the used change structures. Since join is a binary operator, `CA` uses pair changes over changes of the respective bags. The output changes `CB` are sequences of bag changes, which becomes relevant when both input relations change simultaneously. For example, if the left relation grows and the right relation shrinks, we may need to insert some tuples and remove some tuples from the join output.

The state `ST` of `JoinOp` consists of two indices: one for each input relation. Each index is a dictionary that maps keys to a bag of values. Indices have to satisfy some properties, which are part of the invariant described below. Initially, we obtain the state of `JoinOp` using a helper function `build_ix` that converts a bag into an index. This only has to be done once; afterwards `Δst` will update the state as needed. We can undo this conversion using the function `as_bag`, which makes it easy to formulate the validity proposition `vs` for states. Note that validity only occurs as part of the formalization, but is not needed during execution. Hence, the overhead of `as_bag` is irrelevant.

When a change arrives, `Δst` updates the indices using the parametric helper function `Δix`, which can operate on either index. When new tuples are inserted (resp. removed), we insert (resp. remove) the corresponding entries from the index. This is necessary to make sure that the indices always represent the current input relations faithfully, as required by the invariant. The invariant `inv` uses helper function `inv_ix` to restrict both indices. Each index needs to be well-formed and contain exactly those elements that occur in the corresponding input relation (including their support counts). An index is well-formed if it is a finite map and each tuple that occurs is indexed by its key. The proofs of `inv_init` and `inv_step` (not shown) guarantee that `JoinOp` maintains this invariant.

Finally, we can use the indices to compute the output change of `JoinOp`. We use two helper functions `Δf_l` and `Δf_r` that process changes of the left and right input relation, respectively. When tuples are inserted into the left input relation, `Δf_l` uses the right index to find the matching tuples in the right relation. To this end, we define a generic helper function `join_ix` that takes an index and a bag and combines their elements. Specifically, for each tuple `x` in the bag, we find

```
Module JoinOp <: StatefulDiffOp.
  Parameter (L : Type) (R : Type).
  Parameter (K : Type) (lk : L → K) (rk : R → K) (EqK : EqDec K).

  Definition A := bag L * bag R.            Definition B := bag (L * R).
  Definition CA := pairc (bagc L) (bagc R). Definition CB := seqc (bagc (L * R)).
  Definition f x := join lk rk (fst x) (snd x).

  Definition ix K V := dict K (bag V).
  Definition ST := ix K L * ix K R.
  Definition init x := (build_ix lk (fst x), build_ix rk (snd x)).
  Definition vs Δx st := vc Δx (as_bag (fst st), as_bag (snd st)).

  (* Insert/remove elements from the index to keep in sync with input bags. *)
  Definition Δix {C} (k : C → K) (Δx : bag_change C) (st : ix K C) : ix K C :=
    match Δx with
    | bag_union b ⇒ ix_union_bag k st b
    | bag_diff b ⇒ ix_diff_bag k st b
    end.
  Definition Δst Δx st (_ : vs Δx st) : ST :=
    match Δx with
    | pair_fst Δbl ⇒ (Δix lk Δbl (fst st), snd st)
    | pair_snd Δbr ⇒ (fst st, Δix rk Δbr (snd st))
    | pair_both Δbl Δbr ⇒ (Δix lk Δbl (fst st), Δix rk Δbr (snd st))
    end.

  (* Each index is well-formed and contains the same elements as the input bag. *)
  Definition inv_ix {C} (k : C → K) (b : bag C) (st : ix K C) :=
    wellformed_ix st ∧ forall x, bag_support b x = ix_support st (k x) x.
  Definition inv x st : Prop :=
    inv_ix lk (fst x) (fst st) ∧ inv_ix rk (snd x) (snd st).

  (* For each x in b, do an index lookup and combine the resulting tuples. *)
  Definition join_ix {C1 C2 C3} (k : C1 → K) (st : ix K C1) (b : bag C2)
                                (f : bag C1 → bag C2 → bag C3) : bag C3 :=
    flat_map (fun x⇒
      f (ix_lookup st (k x)) (singleton_bag x)
    ) b.
  Definition Δf_l (Δbl : bag_change L) (st : ix K R) : bag_change (L * R) :=
    match Δbl with
    | bag_union b ⇒ bag_union (join_ix lk st b (fun b a ⇒ product a b))
    | bag_diff b ⇒ bag_diff (join_ix lk st b (fun b a ⇒ product a b))
    end.
  Definition Δf_r (Δbr : bag_change R) (st : ix K L) : bag_change (L * R) :=
    match Δbr with
    | bag_union b ⇒ bag_union (join_ix rk st b (fun a b ⇒ product a b))
    | bag_diff b ⇒ bag_diff (join_ix rk st b (fun a b ⇒ product a b))
    end.
  Definition Δf Δx st (_ : vs Δx st) :=
    match Δx with
    | pair_fst Δbl ⇒ [Δf_l Δbl (snd st)]
    | pair_snd Δbr ⇒ [Δf_r Δbr (fst st)]
    | pair_both Δbl Δbr ⇒ [Δf_l Δbl (snd st); Δf_r Δbr (Δix lk Δbl (fst st))]
    end.
End JoinOp.
```

Fig. 3. Modeling Δjoin as stateful differential operator that takes indices as state.

all indexed tuples with the same key `k x`. We then combine them with `x` using function `f`, whose purpose is to ensure the tuples are combined in the right order to yield a `bag (L*R)` (note the swapped parameters names in `Δf_l` and `Δf_r`). The differential computation `Δf` delegates input changes to `Δf_l` and `Δf_r` as needed. However, there is a pitfall that is easy to miss: When both relations change simultaneously (`pair_both`), we have to process the second change using an updated index to account for the first change. Concretely, we call `Δix` to update the index passed to `Δf_r` to account for `Δbl`. Otherwise, the correctness proof would have failed.

We have verified that `JoinOp` is a correct stateful differential operator in Rocq. Technically, we had to define a variant of the module type `StatefulDiffOp` that permits the use of setoids. In Rocq, setoids are used for data types that are compared modulo equivalence rather than equality. This is necessary because our bag data type does not have a canonical representation. Hence, we defined `StatefulDiffOpSetoid` which is identical to `StatefulDiffOp` except for `inc_correct`, where it uses equivalence `==` rather than equality `=` to compare the recomputation result to the updated computation result. We elided these details in the paper for simplicity.

In summary, we modeled an efficient differential join operator using our framework and proved it correct. The differential join operator uses a complex state, protected through a sophisticated invariant. While the correctness of this operator was never in doubt, it is reaffirming that our framework is expressive enough to describe and reason about a differential operator used in real-world applications.

6.2 Differential user-defined aggregation

Datalog is a logic programming language that computes relations based on inference rules. Datalog is well-known for its support for scalable source-code analysis [Bravenboer and Smaragdakis 2009], and the use of user-defined lattice-based aggregation [Madsen et al. 2016]. Some Datalog engines also support incremental computing [Ryzhyk and Budiu 2019; Szabó et al. 2016], where changes to input relations trigger updates of output relations. However, user-defined lattice-based aggregation poses a challenge for incremental Datalog engines, because naively deleting an aggregand would require a rescan of the remaining aggregands. To this end, Szabó et al. [2018] proposed:

> "We build a balanced search tree from the aggregands. At each node, we store additionally the aggregate result of all aggregands at or below that node. The final aggregate result is available at the root node. Upon insertion or deletion, we proceed with the usual search tree manipulation. Then we locally recompute the aggregate results of affected nodes and their ancestors in the tree. [...] This way we can incrementally update aggregate results in $O(\log N)$ steps."

They integrate this strategy in their Datalog system IncA, where it helped perform incremental analysis of real-world software projects [Szabó et al. 2021]. However, their paper does not provide any further detail about the implementation or correctness of this strategy.

We investigate if it is possible to express above strategy as a stateful differential operator in our framework. In particular, we formalize a data structure called *aggregate tree* that implements the strategy by Szabó et al. [2018]. We then use this aggregate tree as custom state in a differential operator to incrementalize aggregation on bags for join-style operations.

We show an excerpt of our aggregate tree formalization in Figure 4. Aggregate trees are parametric in type `E`, strict total order `<E`, operation `join`, which must be associative and commutative. An aggregate tree `AT` is either empty or consists of an inner node `Node` with two subtrees `l` and `r`. Inner nodes store an element, where `count` describes how many time `elem` was inserted. The special feature of aggregate trees is that they also store the `result` of applying `join` to all elements in the subtree. We often use the smart constructor `node`, which initializes `result` appropriately.

```
Module AggregateTree (Ord : STRICT_TOTAL_ORDER) (Join : JOIN(Ord)).
Import Ord. (* Import element type E and comparison functions <E and =E.  *)
Import Join. (* Import join : E → E → E *)
Inductive AT : Type :=  (* Aggregate tree is a binary tree with aggregate result *)
  Empty : AT | Node (elem : P) (count : nat) (res : E) (l r : AT) : AT.
Definition result (t : AT) : E := (* extract result or bottom *)
  match t with Empty ⇒ Ord.bottom | Node _ _ res _ _ ⇒ res end.
(* Join the same element multiple times *)
Fixpoint elem_join (elem : E) (count : nat) : E := match count with
  | 0 ⇒ bottom
  | 1 ⇒ elem
  | S count ⇒ join (elem_join elem count) elem
  end.
Definition node (elem : E) (count : nat) (l r : AT) : AT := (* make node w/ join *)
  Node elem count (join (join (result l) (result r)) (elem_join elem count)) l r.
(* Invariant 1: Valid aggregate trees are binary search trees *)
Fixpoint BST (t : AT) : Prop := match t with
 | Empty ⇒ True
 | Node elem count res l r ⇒
     BST l ∧ BST r ∧ AT_gt elem l ∧ count > 0 ∧ AT_lt elem r
 end.
(* Invariant 2: Valid aggregate trees maintain the substree's join result *)
Fixpoint AggT (t : AT) : Prop := match t with
 | Empty ⇒ True
 | Node elem count res l r ⇒
     AggT l ∧ AggT r ∧ res = join elem (join (result l) (result r))
 end.
Fixpoint insert (t : AT) (x : E) : AT := match t with
 | Empty ⇒ Node x 1 x Empty Empty
 | Node elem c res l r ⇒
   if x =E elem then node elem (c + 1) res l r
   else if x <E elem then node elem c (insert l x) r else node elem c l (insert r x)
 end.
Fixpoint delete_min (t : AT) : option (A * nat * AT) := ...
Definition delete_root (t : AT) : AT := ...
Fixpoint delete (t : AT) (x : E) : AT := match t with
 | Empty ⇒ t
 | Node elem c res l r ⇒
   if x =E elem then
     if 1 <? c then node elem (c - 1) res l r
     else delete_root t
   else if x <E elem then node elem c (delete l x) r else node elem c l (delete r x)
 end.
Lemma BST_insert : forall x t, BST t → BST (insert t x).
Lemma AggT_insert : forall x t, AggT t → AggT (insert t x).
Lemma BST_delete : forall x t, BST t → BST (delete t x).
Lemma AggT_delete : forall x t, AggT t → AggT (delete t x).
End AggregateTree.
```

Fig. 4. An aggregate tree is a binary search tree that maintains the subtree's aggregation result at each node.

Aggregate trees must adhere to two invariants. First, each aggregate tree must be a binary search tree as specified by property BST. Szabó et al. [2018] also required the tree to be balanced to avoid degenerate trees to affect performance, we omitted this detail from our formalization. Second, the aggregate result stored in a tree must be computed correctly. It is this invariant which ensures we

```
Module AggregationOp (Ord : STRICT_TOTAL_ORDER) (Join : JOIN Ord)
       <: StatefulDiffOp.
  Module Agg := AggregateTree Ord Join.
  Import Ord Join Agg.
  Definition A := bag E.   Definition B := E.
  Definition f x := fold_right join bottom x.
  Parameter DiffE : difference E. (* A difference structure for type E. *)
  Definition CA := bagc E. Definition CB := DiffE.(C). (* change structure for E. *)
  Definition ST : Type := AT.
  Definition init x := fold_left insert Empty x.
  Definition vs dt1 st : Prop := ...
  Definition Δst (Δx : bag_change E) st (_ : vs Δt st) : ST :=
    match Δx with
    | bag_union b ⇒ fold_left insert st b
    | bag_diff b ⇒ fold_left delete st b
    end.
  Definition inv (x : bag E) st : Prop :=
    BST st ∧ AggT st ∧ forall (e : E), bag_count_occ x e = tree_count_occ st e.
  Definition Δf Δt st (H : vs Δt st) : ΔT CB :=
    let st' := Δst Δt st H in (result st') ⊖ (result st).
End AggregationOp.
```

Fig. 5. Modeling incremental user-defined aggregation as a stateful differential operator.

can read the correct aggregation result from the root of tree. Our differential operator will enforce both invariants, and we have proven that they are maintained correctly.

We define functions to add and remove elements from an aggregate tree. Function `insert` navigates the binary search tree until it either (i) finds the element already exists or (ii) reaches an empty tree. If the element exists, we increment its count and update the aggregation result at the current node in accordance with the updated count. If the element does not exist, we insert it and update the join results along the spine. In total, `insert` takes time logarithmic in the size of the (balanced) tree.

Deletion is more involved, because we may need to remove nodes from the middle of the tree in a way that restores both invariants. Specifically, when `delete` finds the element to be removed has a count of 1, the node must be removed. To remove the root of `(Node elem c res l r)`, we implement the common strategy of finding and removing the minimal (i.e, left-most) leaf in `r` and making the leaf element the new root element. This maintains the binary search tree invariant. To maintain the aggregate result invariant, we must recompute the aggregation result along the complete spine from the minimal leaf to the root of the tree, which takes logarithmic time as desired.

Finally, we can define a differential aggregation operator that uses the aggregate tree as its state as shown in Figure 5. The aggregation operator takes a `bag E` and yields its join result of type `E`. Incremental Datalog engines process bag changes as represented by our change structure `bagc`. However, they do not handle fine-grained changes of individual elements, and our aggregate trees compute the updated join result rather than how it changed. There are different ways of integrating updated results into our framework. We demonstrate the most general approach: We require a difference structure `DiffE` on values of type `E` and use its internal change structure. This design works for any data type that implements the `difference` interface, without having a differential join operator. Alternatively, we could have used a trivial change structure that replaces the current element by a new one:

```
Definition repl_change A : change A := {| ΔT:=A; vc _ _:=True; patch new _ _:=new |}.
```

We use an aggregate tree as the state of the differential operator, which we initialize and maintain using `insert` and `delete`. Through `inv`, we enforce the invariants of the aggregate tree's and additionally guarantee that it contains the exact same elements as the current input bag. The differential computation `Δf` then becomes easy: We read the old and the new aggregation results from the root of the tree, and use the diffing operation ⊖ to compute the corresponding change. Our correctness proof establishes that the aggregation tree indeed is a valid implementation strategy for supporting incremental aggregation over bags.

6.3 String differential operators

In the previous case study, we showed how to formalize existing differential operators from relational algebra and Datalog as stateful differential operators. But our framework also forms the basis for discovering novel differential operators. To this end, in our second case study, we develop a series of novel and efficient differential operators for strings and prove them correct.

First, let us reconsider the change structure for strings. In Section 5.3, we introduced the trim-right operator to illustrate stateful differential operators. However, for simplicity, we only considered two basic changes: prepend and append of individual characters. In general, string changes should support changes at any index of the string, and they should support the insertion and deletion of entire substrings. The following change structure encodes these kind of changes and their validity:

```
Inductive str_change := ins (i : nat) (s : string) | del (i : nat) (n : nat).
Definition stringc : Change string := {|
  ΔT := str_change;
  vc Δs s := match Δs with
    | ins i _ ⇒ i <= length s | del i n ⇒ i + n <= length s
    end;
  patch Δs s _ := match Δs with
    | ins i ss ⇒ substring s 0 i ++ ss ++ substring s i (length s)
    | del i n ⇒ substring s 0 i ++ substring s (i + n) (length s)
    end
|}.
```

Based on this change structure, we have developed and verified 7 differential string operators, of which 3 are stateless and 4 are stateful. Here, we summarize our findings.

Stateless operators. We developed differential operators `length`, `toUpperCase`, and `toLowerCase`. These operators do not require state to process input changes. For example, `length` yields `nat_inc (length s)` for `ins i s` and `nat_dec n` for `del i n`. And since `toUpperCase` maps over the string, it yields `ins i (toUpperCase s)` for `ins i s`, whereas deletions are simply forwarded to the output. Hence, the three stateless operators take time linear in the size of the change, but independent of the original input. Since the orignal operations require linear time in the size of the full input, our differential operators provide an asymptotic speedup.

Stateful operators. We developed stateful differential operators `isEmpty`, `concat`, `indexOf`, and `lastIndexOf`. These operators use different kinds of states:

isEmpty uses `nat` as state, which represents the length of the current string. The operator yields `bool_neg` when the length flips from zero to positive or vice versa. In general, this operator does not speedup the original computation, which takes constant time.

concat uses `nat` as state, which represents the length of the left input string. The operator uses the state to shift any manipulation of the right input string by the length of the left input string. Differential `concat` takes time linear in the size of the change (to compute the length of the inserted string), whereas the non-incremental `concat` takes time linear in the size of

the left string (to perform list concatenation). Hence, we obtain asymptotic speedups when the change size is significantly smaller than the input size.

indexOf uses `string * option nat` as state, which represents the previous input and the previously found index. We use the previous index to determine whether a change can affect the search result. When inserting before the previously fond index, we check if the inserted substring contains the character we search, which may lead to an updated index. This takes time linear in the size of the change. However, when deleting a substring that contains the previously found index, we have to search the remaining string for another occurrence, which takes time linear the input size. Hence, this operator's asymptotic performance varies.

lastIndexOf also uses `string * option nat` as state and behaves analogously to `indexOf`. In particular, the differential performance is the same as for `indexOf`: Many changes can be handled in time linear in the size of the change, but deleting a substring that contains the previously found index triggers a search for the second-to-last occurrence. However, the non-incremental performance of `lastIndexOf` is worse than `indexOf` in practical scenarios, because we always have to search until the end of the string.[5] Hence, the potential for incremental speedups is larger here, as our evaluation will show.

All of these stateful differential are new discoveries, and the custom state is elemental in providing efficient incremental performance. We believe that our framework opens the door for the community-driven development of a complete standard library of differential operators.

7 Building Incremental Computations from Differential Operators

So far, this paper focused on the theoretical aspects of (stateful) differential operators: how to specify their behavior, how to model them, and how to prove them correct. In this section, we show that our differential operators give rise to practical incremental computations. Specifically, we demonstrate the following:

(1) We can express differential operators in general-purpose languages. To this end, we extract OCaml code from the Rocq formalization of the differential operators from relational algebra and strings.
(2) We can compose differential operators into incremental computations. To this end, we encapsulate stateful differential operators and provide an API for composing them.
(3) The resulting incremental computations yield significant speedups when processing input changes. To this end, we construct and exercise a few relational queries and string operations built from differential operators.

7.1 Extracting differential operators into OCaml

The Rocq Prover supports the extraction of programs to different target languages via plugins [Letouzey 2008], retaining the functional properties of the code. We used this facility to extract OCaml implementations of a few differential operators that we previously modeled and verified. Specifically, we obtain one OCaml module for each differential operator and the functions have the same signature as in Rocq, albeit without dependent types. In particular, the extracted stateful differential operators remain in a purely functional style, where `Δst` returns the updated state. As noted before, our formalization omitted the rebalancing of the aggregation tree. We implemented AVL-style rebalancing in OCaml after extraction to avoid degenerated aggregation trees [Szabó et al. 2018]. Other than that, we reuse the data structure of bags, join indices, and strings from our Rocq formalization, which are list-based and not efficient.

[5] We assume a list-based representation of strings, which is typical in functional programming languages.

We implement an API for building incremental computations from differential operators. Each incremental computation maps input changes `da` to output changes `db`, as captured by the following interface:

```
module type IC = sig type da ; type db ; val  ic : da → db end
```

For stateless differential operators, it is easy to implement this interface by setting `ic = Δf`. But for stateful operators, we must encapsulate their internal state like this:

```
module Product (P : ProductParams) : IC = struct
  type da = (P.t1 bag_change, P.t2 bag_change) pair_change
  type db = (P.t1 * P.t2) bag_change list
  let state := ref (init P.init_input)
  let ic da =
    let res = Δf da !state in
    state := Δst da !state; res
end
```

Here, `P` captures the parameters of the differential operator, including the types of the input relations and their initial value `init_input`. The latter is used to initialize a mutable reference cell named `state`. In OCaml, `!state` reads the content of the reference cell and `state := v; w` writes `v` before returning `w`. We can now implement `ic` by running `Δf` on the old state before updating the state using `Δst`. The same pattern applies to all stateful differential operators: we encapsulate their internal state and hide it behind interface `IC`.

To construct interesting incremental computations, we must also be able to compose differential operators that implement `IC`. To this end, we provide module functors for the unary and binary composition of incremental computations:

```
module Seq (F : IC) (G : IC with type db = F.da) : IC = struct
  type da = G.da
  type db = F.db
  let ic da = F.ic (G.ic da)
end

module Binary (F : IC) (G : IC)
              (H : IC with type da = (F.db, G.db) pair_change) : IC = struct
  type da = (F.da, G.da) pair_change
  type db = H.db
  let ic da = match da with
  | Pair_fst da1 → H.ic (Pair_fst (F.ic da1))
  | Pair_snd da2 → H.ic (Pair_snd (G.ic da2))
  | Pair_both (da1, da2) → H.ic (Pair_both (F.ic da1, G.ic da2))
end

module Map (F : IC) = struct
  type da = F.da list
  type db = F.db list
  let rec dop da = match da with [] → [] | x :: xs → (Op.dop x) :: (dop xs)
end
```

A sequential composition `Seq` feeds the output changes of `G` to `F` as input changes. The sharing constraint `with type` asserts that the output type `G.db` conforms with `F.da`. The implementation then becomes very simple and represents functional composition `F ∘ G`. We can use `Seq` to apply unary differential operators like `select` and `aggregate`. A `Binary` composition is more complex: It feeds the output changes of two computations `F` and `G` to a single computation `H`, which must accept a `pair_change` as input. This construction corresponds to `fun (x1,x2) → H (F x1) (G x2)`. We use `Binary` to apply binary differential operators like `product` and `join`. Lastly, `Map` transforms

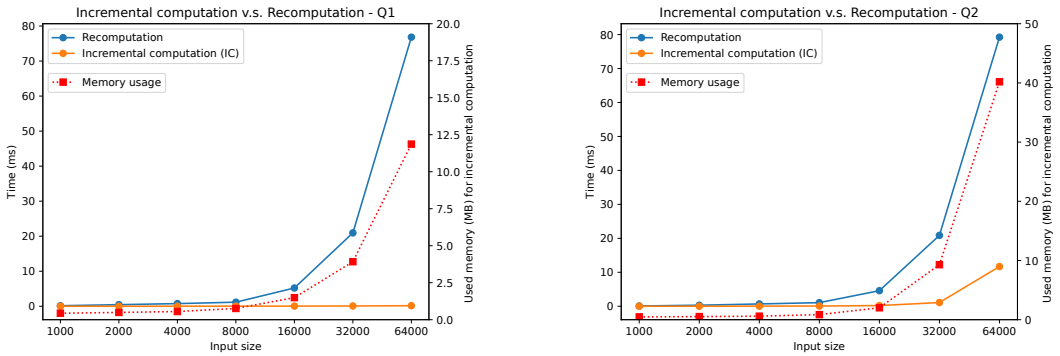


Fig. 6. Execution time of incremental computation and recomputation for Q1 and Q2

an incremental computation to handle sequence of input changes. The `Map` functor is useful for composing `F` and `G` when `F.da = G.db list`, so that `Seq(Map(F))(G)` becomes valid.

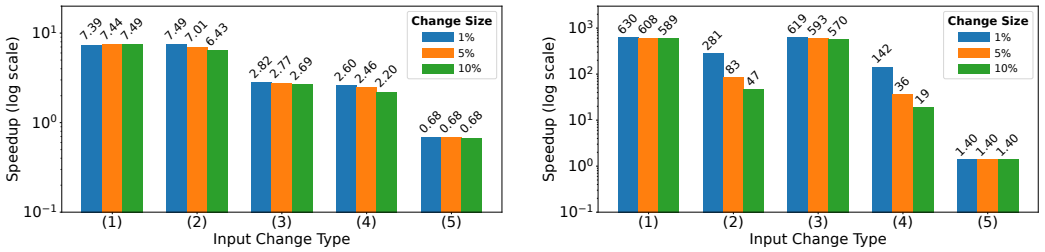
7.2 Evaluation on relational queries

We have used the extracted operators and our API to construct a few relational queries and performed benchmarking. This is to show that our framework is well-suited for developing useful differential operators that can power realistic incremental computations. The concrete operators we used are known from database systems and do not require a dedicated performance evaluation. We performed our benchmarking on a machine with an Apple M2 chip at 3.49 GHz with 24 GB of RAM running 64-bit OSX 15.3.2 and OCaml 5.1.1. We consider two incremental queries `Q1` and `Q2` in our evaluation:

```
module S0 = SelectDiffOp(struct type a = int; let cond x = x mod 10 = 0 end)
module S1 = SelectDiffOp(struct type a = int; let cond x = x mod 5 = 0 end)
module J = JoinOp(struct ...; let lk x = x mod 500;; let rk y = y mod 1000 end)
module Q1 = Binary(S0)(S1)(J)    (* S_0 ⋈ S1 *)
module A = AggregationOp(struct ...; let lt = pair_lt; let join = pair_max end)
module Q2 = Seq(Map(A))(Q1)      (* Aggregate_pair_max Q1 *)
```

The first query `Q1` joins the results of two selection operators `S0` and `S1`. `S0` selects numbers divisible by 10 from its input, whereas `S1` selects numbers divisible by 5. We feed the results of the selections into the join `J` using the `Binary` functor. The join merges inputs whose remainder is equal modulo 500 and 1000, respectively During the execution, `Q1` takes a change of type `(int bag_change, int bag_change) pair_change` as input, and returns a change of type `(int * int) bag_change list`. The second query `Q2` further performs an aggregation on top of the results of `Q1`. The aggregation operator can be used with any user-defined function that is associative, commutative, and order-preserving. To illustrate, our aggregation `A` takes pairs of numbers as input and computes their maximum using a lexicographic ordering. As `Q1` returns a list of bag changes, but `AggregateOp` expects a single bag change, we apply the `Map` functor before using `Seq` to connect it with `Q1`.

We measured the incremental and non-incremental execution time of `Q1` and `Q2`, along with the memory usage for incremental computation, across initial database sizes ranging from 1 000 to 64 000 tuples. We use half of the initial tuples as input for `S0` and the other half for `S1`. All inputs are consecutive numbers, such that the result of the join operator `J` grows superlinearly with the number of inputs. We can observe the consequential non-incremental recomputation times in Figure 6 for both `Q1` and `Q2`.



(a) Differential speedups before optimization.

(b) Differential speedups after optimization.

Fig. 7. Differential string operators deliver tremendous speedups compared to a from-scratch computation.

To measure the incremental performance, we synthesized input changes that insert or remove random numbers. Each synthesized change affects 1% of the current tuples, evenly split between `S0` and `S1`. For example, when the initial database contains 1 000 numbers, we might generate a change `Pair_both (Bag_union [748;...;752], Bag_diff [777;...;781])`. We ensure to only remove existing elements and to only insert new elements, which will trigger more computations in the incrementalization. Moreover, since the cost of computing different changes are different, we average the execution time of three runs. We can observe that both `Q1` and `Q2` bring asymptotic speedups compared with a recomputation. For example, when the input size is 64k, the average execution time of `Q1` is 0.16ms, while recomputation requires 76.8ms. The incremental performance of `Q2` is getting slower for larger inputs, which may be due to our use of inefficient data structures for bags and join indices.

7.3 Evaluation on string operations

Our second benchmarking experiment targets string operations, aiming to demonstrate that stateful differential operators enable efficient incremental computation for new domains. We consider a simple string-processing function that combines four string operators:

```
let f (s1, s2) = lastIndexOf('A', concat (toLowerCase s1, toUpperCase s2))
```

Note that the differential operators of `concat` and `lastIndexOf` are stateful, whereas the other two are stateless. We choose this function for the evaluation because it highlights an important property of differential updating: different types of changes can have significantly different impact on the incremental performance. We build a differential version of `f` using our API:

```
module M = Binary(ToLowerCase)(ToUpperCase)(Concat)
module F = Seq(Map(LastIndexOf(struct let c = 'A' end)))(M)
```

Here, `M` builds the incremental computation `concat(toLowerCase(s1), toUpperCase(s2))`, whose output is a list of string changes. This is necessary because when both operands of `concat` are modified, the resulting change to the output cannot, in general, be represented by a single change. We forward the concat changes to `LastIndexOf`, finding the last occurrence of `'A'`.

In our evaluation, we consider five types of changes to the input of `f`:

(1) Delete a substring in `s1`. Expected incremental behavior: constant time processing.
(2) Insert a substring in `s1`. Expected incremental behavior: linear time in the size of the inserted string, which `toLowerCase` must map over and for which `concat` computes the length.
(3) Delete a substring in `s2` before the last occurrence of `'A'`. Expected incremental behavior: constant time processing.

(4) Insert a substring in `s2` before the last `'A'`. Expected incremental behavior: linear time in the size of the inserted string, which `toUpperCase` must map over.
(5) Delete a substring containing the last `'A'` in `s2`. Expected incremental behavior: linear time in the size of the original input, because `lastIndexOf` must search for the next occurrence.

We evaluate the differential computation using two paragraphs of text as input for `s1` (1425 characters) and `s2` (570 characters). Then, for each kind of change, we consider different change sizes relative to the input string: 1% (20 characters), 5% (100 characters), and 10% (200 characters), where inserted characters are randomly generated. We repeat each measurement 3000 times, eliminate outliers, and compute the average speedup relative to a from-scratch computation. We show the results in Figure 7a, which deviates from what we expected. For example, we expected change (1) to provide constant time updates and the time for change (2) should be dependent on the change size. Further investigation revealed that there is a performance bottleneck in `lastIndexOf`.

Our stateful differential operators define separate functions Δf and Δst to update their output and state, which is nice for reasoning about them. However, for `lastIndexOf`, this leads to redundant computations. Therefore, we merged Δf and Δst in the extracted Ocaml code to eliminate this redundancy, which improves performance considerably. But we noted another performance improvement, which we implemented: `lastIndexOf` maintains the current input as part of its state. However, it only requires this part of the state in case a recomputation becomes necessary. There is a generic optimization we can use: Collect the input changes and only patch them once the current input is needed. That is, we modified the state of `lastIndexOf` to `string * string_change list * option nat` and delay patching until necessary. With this optimization, we repeated the measurements and show the results in Figure 7b, which demonstrates significant speedups that match our expectations. Interestingly, even when we delete the last occurrence of `'A'` to force `lastIndexOf` to recompute from scratch, the incremental computation still outperforms recomputation because the other operators maintain their incremental performance advantage.

In summary, we have shown that it is possible to implement stateful differential operators in general-purpose languages, that stateful differential operators can be composed to describe complex incremental computations, and that these computations achieve the asymptotic speedups promised by incremental computing.

8 Discussion

While our framework provides a comprehensive interface for implementing efficient differential operators, how to design them effectively remains an open question. Drawing from the insights we learned from designing the string operators shown in Section 6.3, we outline the challenges programmers face and offer some strategies for the design process.

The efficiency of a differential operator hinges on its smart use of state to cache relevant information. To design a stateful operator, we must balance three factors: the reaction time of `Δf`, the update time of `Δst`, and the memory required to maintain the state. In our experience, it is usually best to start defining the change-processing function `Δf` and to observe what information is necessary to implement it. And there are reusable design patterns. For example, when an operator only requires an up-to-date state for some changes but not others, it is often more efficient to delay the state updating until necessary, as we have done for `lastIndexOf` in Section 7.3.

Moreover, programmers should be aware that differential operators are error-prone. Most bugs we encountered were due to the incorrect or missing handling of corner cases in the definition of `Δf` and `Δst`. Fixing these bugs sometimes involves revising the state and the state updating, which then requires fixes in `Δst`, and so on until we reach a fixed point. This iterative design loop makes formal verification with Rocq impractical for day-to-day development. The challenge is

not just using the proof assistant, but re-proving correctness after each design change. Therefore, we propose an alternative for programmers: using our framework's correctness specification as a foundation for property-based testing, which provides a more efficient way to validate operators during development, before proving them correct.

Finally, we want to discuss the impact of the change structure on the design of differential operators. So far, we have not developed a comprehensive algebraic structure for changes. Consequently, implementing certain differential operators requires extra effort on the part of the user. For example, suppose we want to implement a differential operator for `fst : A * B → A`, which maps a pair to its first element. When a change to a pair `(a, b)` only modifies the second element, implementing `Δfst` must yield a "no change" of type `ΔA`, which is not part of the elemental change structure we provided. Instead, we defined additional type classes such as `no_change` and `difference`, which introduce additional part of the algebra of changes. With this, `Δfst` can be implemented using an extra type class constraint `[no_change A ΔA]`, which then provides a zero change `noc : ΔA` such that `a ⊕ noc = a` for all `a : A`. Moreover, compared to the change algebra introduced in the ILC framework [Cai et al. 2014; Giarrusso 2020], our work does not support function changes yet, which poses a problem when programmers want to define differential operators for higher-order functions. For future work, we plan to investigate richer algebraic structures over changes and how incorporate function changes.

9 Related Work

In this work, we proposed a framework for developing and verifying stateful differential operators. This design differs from most prior work which either neglect the algebraic properties of changes or lack support for using custom states to accelerate incrementalization.

We categorize the literature of incremental computing into two approaches: differential updating and selective recomputation. Differential updating approaches translate any change of a computation's input to change of the computation's output by chaining differential operators. Selective recomputation approaches try to reuse the previous computation results by analyzing the program structure. Our framework belong to the differential updating approach, but to the best of our knowledge, we give the first formal specification of differential operators. In the remainder of this section, we discuss the related work based on this distinction.

9.1 Differential updating

Incremental view maintenance. Blakeley et al. first proposed change-processing operators to update views in relational database [Blakeley et al. 1986]. The change-processing operators only perform necessary updates to maintain the views. Their approach was later refined to efficiently support duplicates in bags [Chaudhuri et al. 1995; Griffin and Libkin 1995; Gupta et al. 1993] and aggregation [Ramakrishnan et al. 1994]. DBToaster [Koch et al. 2014] introduced higher-order deltas that further speed up change processing for certain expensive queries.

DBSP [Budiu et al. 2023] gives a formal model that unifies streaming computation and incremental view maintenance. DBSP builds incremental computations by composing primitive operators and supports stateful operators such as equi-join, which share some similarities with our framework. Nevertheless, we find DBSP and our framework differ in both approach and scope. DBSP targets database queries rather than general-purpose computations and provides a fixed set of built-in primitive operators. In particular, DBSP abstracts away the implementation and internal state management of stateful operators, as long as they correctly translate input changes into output changes. In contrast, our work provides a framework for building custom stateful operators for general-purpose computation, enabling explicit reasoning about their states. Furthermore, although DBSP allows supplying auxiliary information via a delayed feedback loop, emulating our stateful

differential operators within DBSP remains impossible. This is because DBSP operates purely on stream of changes, while our operators require access to the full and up-to-date state. Moreover, DBSP only allows changes that form commutative groups. As we have shown with string changes, this constraint is too strong to model data changes beyond relational algebra.

Incremental Datalog. Differential updating in relational algebra has enabled the development of incremental solvers for Datalog. Datalog resembles relational algebra in some senses, but it allows expressing recursions from its basic language constructs [Green et al. 2013]. This induces difficulty for incrementalization, since tracking the dependencies across the mutual recursive Datalog rules is tricky. Gupta et al. proposed the first algorithm DRed [Gupta and Mumick 1995] for incrementalizing recursive Datalog programs. Differential dataflow [McSherry et al. 2013; Ryzhyk and Budiu 2019] is an alternative for DRed that sometimes provides better incrementality by tracking the recursion depth for each tuple. IncA [Klopp et al. 2024; Szabó et al. 2018, 2016] is a Datalog engine that supports efficient incremental recursive aggregation. We can observe that most related work of differential updating limits to the relational algebra. Our framework provides a foundation for developing differential operators in other domains, as witnessed by our differential string operators.

Incremental lambda calculus (ILC). ILC [Cai et al. 2014] extends the simply-typed lambda calculus (STLC) with change types and present a framework for static differentiation of STLC programs. Static differentiation refers to computing a derivative Δf from the original function f at compile time that satisfies the equation $f(x \oplus \Delta x) = f(x) \oplus \Delta f(x, \Delta x)$. They define basic rules for differentiating closed STLC terms like constants, variables, abstraction, and application. But the translated programs are usually not efficient, as the differential rules overlook the algebraic properties of the functions and changes, such as those in relational algebra operations. Hence, they also allow users to provide differential plugins to integrate handcrafted derivatives. The main difference between their work and ours is that their derivatives, Δf, must carry and can only use the original input to compute the output change. This design limits efficiency and flexibility when derivatives are stateless or require custom states, as demonstrated in our work.

Technically, ILC provides a program transformation that converts normal programs into incremental programs. During transformation, each primitive operator $f : A \to B$ is replaced with a differential operator $\Delta f : A \to \Delta A \to \Delta B$ that must be manually designed. Since Δf must take the previous input x as argument for each change Δx, one issue is that the current x has to be recomputed for all operators that occur in a computation. To this end, Giarrusso et al. [2019] propose a systematic caching strategy to reuse previously computed inputs. Technically, they use the same form of differential operators as the original ILC work, the only difference is their operators can also obtain the cache, but the cache is from previous sub-computations, rather than previous change-processing rounds. Thus, although this eliminates some of the accidental overhead, it does not reduce the inherent loss of efficiency: The transformed program is only efficient if Δf requires the current input exactly, but neither stateless nor custom-state operators are well-supported. For example, our `trim`, `join`, and `aggregate` operators perform asymptotically better than the best possible differential operator in ILC.

We also considered if it is possible to integrate stateful differential operators into ILC. The only option seems to be to incorporate the differential state in the non-incremental operator already $f : (A, ST) \to B$, changing existing call sites from $f(x)$ to $f(x, init\ x)$. ILC would then derive $\Delta f : (A, ST) \to (\Delta A, \Delta ST) \to \Delta B$ and transform $f(x, init\ x)$ to $\Delta f(x, init\ x)(\Delta x, \Delta init\ x\ \Delta x)$. We can now set $\Delta init\ x\ \Delta x = \Delta x$ and $st \oplus \Delta x = \Delta st\ st\ \Delta x$ to update the current state correctly. Then, $\Delta f\ (x, st)\ (\Delta x, \Delta x') = \Delta f_{stateful}\ st\ \Delta x$ should yield the correct result when calling one of our stateful differential operators $\Delta f_{stateful}$ (we have not formalized this). However, there is a lot of overhead

in the derived program, and it is not clear how easily this can be eliminated. We leave a deeper investigation for future work.

Giarrusso's Ph.D. thesis [Giarrusso 2020] presents a more comprehensive theory of change algebra than any previous work on ILC. Specifically, they define a notion of *basic change structure*. It only consists of a value set V, a change set ΔV, and a tenary relation $dv \triangleright v_1 \hookrightarrow v_2$, which denotes $dv : \Delta V$ is a valid change from value $v_1 : V$ to $v_2 : V$. It is different from the original ILC framework as the change set ΔV does not depend on a specific value $v \in V$, as well as the diffing operation $\ominus$ is removed. Our change structure builds on a similar idea but we explicitly separate the validity relation from the patching operation $\oplus$. While their ternary relation may offer better guidance for specification, this separation allows us to treat patching as a computable function rather than as part of a logical relation. Giarrusso's thesis also explores other topics, such as equational reasoning on changes, which is important for optimizing differential operators.

9.2 Selective recomputing

The simplest form of selective recomputing is function memoization [Abadi et al. 1996; Heydon et al. 2000; Liu et al. 1998], where arguments and calls are stored in a cache. Before invoking a function, memoization checks the cache to see if the result is known, otherwise it performs recomputation. This is not efficient since sometimes not all the intermediate results depend on the input changes.

More advanced selective recomputing approaches track control and data dependencies between subcomputation. They construct and maintain a dependency graph that connect the producer of a value to its consumer. One example is static dependency graph [Demers et al. 1981; Erdweg et al. 2015a], which is solely based on the program's computation and independent of the input. Approaches based on static dependency graphs are not expressive because the dependency between data can be changed by different inputs.

Approaches based on dynamic dependency graphs avoids these problems by tracking the data dependencies at runtime. Acar et al. [Acar 2005; Acar et al. 2006] originally proposed self-adjusting computation (SAC), which builds dynamic dependency graphs to incrementalize programs. SAC traces dependencies by storing changable data in special cells that record read and write activities. A dependency is built when the cell is visited, and updating the cell triggers recomputation of dependent values. SAC yields outstanding performance on a wide range of tasks [Acar et al. 2009], including quicksort and convex-hull computations. The Adapton framework further improves the efficiency of SAC by modeling pull-based demand [Hammer et al. 2014] and granularity of nodes [Hammer et al. 2015] on the dependency graph. Compared with our work, the problem of SAC and other selective recomputing approaches is they have no notion of how values change, they only have a notion of changed (yes/no). Consequently, they cannot use the algebraic properties of changes to incrementalize program.

10 Conclusion

In this work, we proposed a formal framework for developing and verifying efficient differential operators. Notably, our framework supports modeling stateful differential operators that maintain information between incoming changes, which is the common characteristics of efficient differential operators. We formalized our framework in Rocq, and demonstrated its expressiveness by modeling both existing complex differential operators in relational algebra and novel differential string operators. Our framework provides a solid foundation for developing differential operators. An immediate future work is to investigate the optimization of differential operators. For example, if the input to a differential operator is a sequence of primitive changes, it is not efficient to handle each change individually if the primitive changes can be compressed. Ultimately, we want to establish a standard library of differential operators for incremental computing.

Acknowledgements

We thank the anonymous reviewers for their effort and helpful suggestions. This work has been funded by the European Research Council (ERC) under the European Union's Horizon 2023 (Grant Agreement ID 101125325).

References

Martín Abadi, Butler W. Lampson, and Jean-Jacques Lévy. 1996. Analysis and Caching of Dependencies. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP 1996, Philadelphia, Pennsylvania, USA, May 24-26, 1996*, Robert Harper and Richard L. Wexelblat (Eds.). ACM, 83–91. doi:10.1145/232627.232638

Umut A Acar. 2005. *Self-adjusting computation.* Carnegie Mellon University.

Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. 2009. An experimental analysis of self-adjusting computation. *ACM Trans. Program. Lang. Syst.* 32, 1 (2009), 3:1–3:53. doi:10.1145/1596527.1596530

Umut A. Acar, Guy E. Blelloch, and Robert Harper. 2006. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.* 28, 6 (2006), 990–1034. doi:10.1145/1186632.1186634

José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. 1986. Efficiently Updating Materialized Views. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 28-30, 1986*, Carlo Zaniolo (Ed.). ACM Press, 61–71. doi:10.1145/16894.16861

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*. ACM, 243–262.

Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. 2023. DBSP: Automatic Incremental View Maintenance for Rich Query Languages. *Proc. VLDB Endow.* 16, 7 (2023), 1601–1614. doi:10.14778/3587136.3587137

Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. 2014. A theory of changes for higher-order languages: incrementalizing λ-calculi by static differentiation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 145–155. doi:10.1145/2594291.2594304

Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. 1995. Optimizing Queries with Materialized Views. In *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, Philip S. Yu and Arbee L. P. Chen (Eds.). IEEE Computer Society, 190–200. doi:10.1109/ICDE.1995.380392

Alan J. Demers, Thomas W. Reps, and Tim Teitelbaum. 1981. Incremental Evaluation for Attribute Grammars with Application to Syntax-Directed Editors. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages, Williamsburg, Virginia, USA, January 1981*, John White, Richard J. Lipton, and Patricia C. Goldberg (Eds.). ACM Press, 105–116. doi:10.1145/567532.567544

Sebastian Erdweg, Oliver Bracevac, Edlira Kuci, Matthias Krebs, and Mira Mezini. 2015a. A co-contextual formulation of type rules and its application to incremental type checking. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 880–897. doi:10.1145/2814270.2814277

Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. 2015b. A sound and optimal incremental build system with dynamic dependencies. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 89–106. doi:10.1145/2814270.2814316

Paolo G. Giarrusso. 2020. *Optimizing and Incrementalizing Higher-order Collection Queries by AST Transformation.* Ph. D. Dissertation. University of Tübingen, Germany. https://publikationen.uni-tuebingen.de/xmlui/handle/10900/97998/

Paolo G. Giarrusso, Yann Régis-Gianas, and Philipp Schuster. 2019. Incremental λ-Calculus in Cache-Transfer Style. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11423)*, Luís Caires (Ed.). Springer, 553–580. doi:10.1007/978-3-030-17184-1_20

Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. 2013. Datalog and Recursive Query Processing. *Found. Trends Databases* 5, 2 (2013), 105–195. doi:10.1561/1900000017

Timothy Griffin and Leonid Libkin. 1995. Incremental Maintenance of Views with Duplicates. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, Michael J. Carey and Donovan A. Schneider (Eds.). ACM Press, 328–339. doi:10.1145/223784.223849

Ashish Gupta and Inderpal Singh Mumick. 1995. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Eng. Bull.* 18, 2 (1995), 3–18. http://sites.computer.org/debull/95JUN-CD.pdf

Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. 1993. Maintaining Views Incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*, Peter

Buneman and Sushil Jajodia (Eds.). ACM Press, 157–166. doi:10.1145/170035.170066

Matthew A. Hammer, Jana Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael W. Hicks, and David Van Horn. 2015. Incremental computation with names. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. ACM, 748–766.

Matthew A. Hammer, Yit Phang Khoo, Michael Hicks, and Jeffrey S. Foster. 2014. Adapton: composable, demand-driven incremental computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 156–166. doi:10.1145/2594291.2594324

Allan Heydon, Roy Levin, and Yuan Yu. 2000. Caching function calls using precise dependencies. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, Monica S. Lam (Ed.). ACM, 311–320. doi:10.1145/349299.349341

Jane Street and OCaml community. 2024. Incr_map — OCaml Package Documentation (v0.17.0). https://ocaml.org/p/incr_map/latest/doc/incr_map/Incr_map/index.html. Accessed: 2025-10-23.

David Klopp, Sebastian Erdweg, and André Pacak. 2024. A Typed Multi-level Datalog IR and Its Compiler Framework. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 327 (2024), 29 pages.

Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. 2014. DBToaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.* 23, 2 (2014), 253–278. doi:10.1007/S00778-013-0348-4

Pierre Letouzey. 2008. Extraction in Coq: An Overview. In *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings (Lecture Notes in Computer Science, Vol. 5028)*, Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe (Eds.). Springer, 359–369.

Yanhong A. Liu, Scott D. Stoller, and Tim Teitelbaum. 1998. Static Caching for Incremental Computation. *ACM Trans. Program. Lang. Syst.* 20, 3 (1998), 546–585. doi:10.1145/291889.291895

Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. 2016. From Datalog to flix: a declarative language for fixed points on lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. ACM, 194–208.

Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *Sixth Biennial Conference on Innovative Data Systems Research, CIDR 2013, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf

André Pacak, Tamás Szabó, and Sebastian Erdweg. 2022. Incremental Processing of Structured Data in Datalog. In *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2022, Auckland, New Zealand, December 6-7, 2022*, Bernhard Scholz and Yukiyoshi Kameyama (Eds.). ACM, 20–32. doi:10.1145/3564719.3568686

Raghu Ramakrishnan, Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. 1994. Efficient Incremental Evaluation of Queries with Aggregation. In *Logic Programming, Proceedings of the 1994 International Symposium, Ithaca, New York, USA, November 13-17, 1994*, Maurice Bruynooghe (Ed.). MIT Press, 204–218.

Leonid Ryzhyk and Mihai Budiu. 2019. Differential Datalog. In *Datalog 2.0 2019 - 3rd International Workshop on the Resurgence of Datalog in Academia and Industry co-located with the 15th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2019) at the Philadelphia Logic Week 2019, Philadelphia, PA (USA), June 4-5, 2019 (CEUR Workshop Proceedings, Vol. 2368)*. CEUR-WS.org, 56–67.

Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing lattice-based program analyses in Datalog. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 139:1–139:29. doi:10.1145/3276509

Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. 2021. Incremental whole-program analysis in Datalog with lattices. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. ACM, 1–15.

Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: a DSL for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. ACM, 320–331.

Received 2025-07-10; accepted 2025-11-06