Incremental Computing by Differential Execution

Prashant Kumar
JGU Mainz, Germany

André Pacak
JGU Mainz, Germany

Sebastian Erdweg
JGU Mainz, Germany

Abstract

Incremental computing offers the potential for significant performance gains by efficiently updating computations in response to changing data. However, traditional approaches are either problem-specific or use an inefficient all-or-nothing strategy of rerunning affected computations entirely. This paper presents differential semantics, a novel approach that directly embeds the propagation of changes into the semantics of a general-purpose programming language. Given a precise description of input changes, differential semantics rules define how these changes are tracked and propagated through core language constructs like assignments, conditionals, and loops to produce corresponding output changes. We formalize differential semantics and verify key properties, including correctness, using the Rocq proof assistant. We also develop and formally prove a set of optimizations, particularly for loop handling, that enable asymptotic performance improvements. An implementation of the semantics as a differential interpreter achieves order-of-magnitude speedups over recomputation on the Bellman-Ford shortest path algorithm.

2012 ACM Subject Classification Theory of computation → Operational semantics; Software and its engineering → Formal language definitions; Software and its engineering → Formal methods; Software and its engineering → Incremental compilers

Keywords and phrases Incremental computing, differential semantics, programming language design, formal verification, big-step semantics

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Funding This work is supported by ERC grant AutoInc (`https://doi.org/10.3030/101125325`).

1 Introduction

The field of incremental computing aims to develop programming abstractions for describing incremental computations: Computations that react to changes of their inputs efficiently. Most existing approaches follow the idea of selective recomputing [22]: They track which subcomputations are affected by an input change transitively and rerun these subcomputations when their up-to-date result is needed [5, 13]. However, this is an all-or-nothing approach: If a subcomputation is affected in any way, it has to be rerun. For example, consider a grammar checker that processes a large text document represented as a character string. Any change to the text will trigger significant recomputations. Better incremental computing must exploit *how* the input was changed, not just *if* it was changed.

In this paper, we propose incremental computing through differential execution. Here and in the remainder of this paper, we use *differential* to mean "dealing with differences". Hence, differential execution means to execute a program given a description of *how* its inputs are changed. We can exploit these change descriptions in two ways. First, we can use dedicated execution rules that react to input changes directly. For example, when the iteration count of a loop is increased, it is sometimes sufficient to execute the extra loop iterations without repeating the previous ones. Second, we can integrate change-processing language primitives,

which are often asymptotically faster than their non-incremental counterpart. For example, we can integrate differential operators from relational algebra, as known from live view maintenance in database systems. This paper focuses on the first aspect: How to provide differential execution rules that exploit how inputs have changed.

We study differential execution based on the big-step operational semantics of a small imperative language that features variable assignments, loops, and conditionals. We explain how to incrementalize these features by developing a differential big-step semantics that reacts to changes in the initial variable assignment. Specifically, while $\sigma \vdash s \Rightarrow \sigma'$ executes statement s in store σ, we define a differential semantics $\sigma, \Delta\sigma \vdash s \Rightarrow_\Delta \Delta\sigma'$ that executes s based on the store change $\Delta\sigma$. We have implemented our formal model of differential execution in Rocq and use it to reason about the differential behavior of programs. First, we prove that the differential semantics does not get stuck and yields the same result as a recomputation from scratch:

$$\sigma \vdash s \Rightarrow \sigma' \ \wedge \ \sigma, \Delta\sigma \vdash s \Rightarrow_\Delta \Delta\sigma' \quad \rightarrow \quad \sigma \oplus \Delta\sigma \vdash s \Rightarrow \sigma' \oplus \Delta\sigma'.$$

And second, we can use the formal semantics to justify a number of important optimizations. For example, we prove that the differential semantics may skip statements that do not read changed variables from $\Delta\sigma$, and we develop provably correct optimizations for the differential execution of loops and conditionals. Compared to prior work on the incremental lambda calculus [9], which also processes change descriptions, we are the first to consider imperative language features and use a completely different methodology: differential semantics instead of type-driven program derivation targeting a standard semantics.

While this paper focuses on the formal development, we also want to demonstrate that the differential semantics enables efficient incremental computations. Unfortunately, a direct implementation of the differential semantics in an interpreter is inefficient because the semantics lacks caching of previously computed information. In particular, the differential semantics $\sigma, \Delta\sigma \vdash s \Rightarrow_\Delta \Delta\sigma'$ expects the original store σ as input, which has to be computed for each statement unless we cache it. Therefore, we develop a differential interpreter based on the differential semantics, but incorporating caching and some verified optimizations. We demonstrate the efficiency of our differential interpreter empirically based on an imperative implementation of the Bellman-Ford shortest-path algorithm. Given changes to the initial edge weights, our differential interpreter computes updates in the shortest paths orders of magnitude faster than a recomputation could. The differential interpreter is implemented in Scala 3 and available open-source.[1]

In summary, this paper makes the following contributions:

- We define a static typing discipline for changes of a language's values, including typed patching and diffing (Section 3).
- We introduce the approach of differential execution based on the differential big-step semantics of an imperative language. We prove that the differential semantics preserves types, makes progress, and yields the correct result (Section 4).
- We propose, develop, and verify optimizations for differential execution that skip unaffected statements and improve the running time of loops and conditionals asymptotically (Section 5).
- We implement an efficient differential interpreter based on our semantics (Section 6) and use it to incrementalize the Bellman-Ford algorithm, where it yields order-of-magnitude speedups (Section 7).

[1] `https://gitlab.rlp.net/plmz/artifacts/autoinc-interp-implementation-ecoop25`

2 Differential Execution: Goals and Challenges

An incremental computation $\Delta f(\Delta x) = \Delta y$ reacts to input changes Δx and derives output changes Δy. How can we automatically provide Δf given the original function f?

In this work, we assume f is given through its source code s, such that $f(x) = y$ when $x \vdash s \Rightarrow y$. Our goal is to define a differential semantics $\sigma, x, \Delta x \vdash s \Rightarrow_\Delta \Delta y$ that can be used to process changes Δx of the program input. Note that only the input x can change; the source code s is not subject to change in this work. We can thus define $\Delta f(x, \Delta x) = \Delta y$ when $\sigma, x, \Delta x \vdash s \Rightarrow_\Delta \Delta y$, since we will see that the differential semantics is deterministic. The challenge of course is to define the differential semantics such that it is not only correct but also has efficient incremental performance when reacting to input changes. In this section, we motivate our approach through examples.

2.1 Differential execution for epidemiological models

In this paper, we explore differential execution for a statement-based imperative programming language called Imp. Imp features numeric and Boolean values that can be used in assignments, conditionals, and repeat loops, which iterate a loop body a fixed number of times

$$\begin{array}{lrcl} \text{(Statements)} & s & ::= & x := e \mid s; s \mid \textbf{if}\ e\ s\ s \mid \textbf{repeat}\ e\ s \\ \text{(Expressions)} & e & ::= & v \mid x \mid e + e \mid e - e \mid e * e \mid e^e \mid e > e \\ \text{(Stores)} & \sigma & & \end{array}$$

We use a big-step reduction semantics $\sigma \vdash s \Rightarrow \sigma'$ that takes a program s and an initial store σ as input, and yields an output store σ'. We can use this language to compute epidemiological models.

Epidemiological models are essential tools for understanding the spread of infectious diseases and informing public health interventions [15]. For example, consider a simplified model that simulates disease spread over a fixed number of n days:

```
# free variables n, I, and β must be bound in the initial store
α = 0.04
S = 0
repeat n:            # n is the number of days for which we model the disease spread
    S = S + I        # S is the cumulative infection count, which grows by I each day
    β = β + α        # β is the disease transmissibility, which grows by α each day
    I = I * 2.718^β  # I is the infected population, which grows exponentially based on β
```

We can now predict how a disease spreads in $n = 6$ days given an initial infectious population $I = 10$ and assuming a transmissibility of $\beta = 0.3$ Figure 1a shows the content of the store σ_i after iteration i of the loop (we omit `n` and α because they are constant). This constitutes the initial, non-incremental run of the program. Note that σ_0 shows the initial state of the model and $\sigma_{out} = \sigma_6$ is the final prediction.

When epidemiologists revise the initial parameters of the model, we want to update our prediction efficiently. In the following, we discuss three change scenarios. First, consider we want to change the initial infected population from 10 to 15. The differential store $\Delta\sigma_0$ in Figure 1b models this change. Our goal is to determine how the final prediction changes, that is, what is $\Delta\sigma_{out}$. To this end, we propose the use of differential execution: Essentially, we rerun the program but only consider and compute changes. Starting from $\Delta\sigma_0$, differential execution of the loop body computes $\Delta\sigma_1$, showing how S and I are affected by the initial change of I. Differential execution continues to iterate the loop body until we reach $\Delta\sigma_6 = \Delta\sigma_{out}$. We observe that 5 additional infections on day 0 lead to 68 additional infections on day 6.

$\sigma_0 = \{S \mapsto 0,\ I \mapsto 10,\ \beta \mapsto 0.3\}$
$\sigma_1 = \{S \mapsto 10,\ I \mapsto 14,\ \beta \mapsto 0.34\}$
$\sigma_2 = \{S \mapsto 24,\ I \mapsto 20,\ \beta \mapsto 0.38\}$
$\sigma_3 = \{S \mapsto 44,\ I \mapsto 30,\ \beta \mapsto 0.42\}$
$\sigma_4 = \{S \mapsto 74,\ I \mapsto 47,\ \beta \mapsto 0.46\}$
$\sigma_5 = \{S \mapsto 121,\ I \mapsto 77,\ \beta \mapsto 0.50\}$
$\sigma_{out} = \{S \mapsto 198, I \mapsto 132, \beta \mapsto 0.54\}$

(a) Stores σ_i after i iterations.

$\Delta\sigma_0 = \{I \mapsto \textbf{inc}\ 5\}$
$\Delta\sigma_1 = \{S \mapsto \textbf{inc}\ 5,\ I \mapsto \textbf{inc}\ 7\}$
$\Delta\sigma_2 = \{S \mapsto \textbf{inc}\ 12,\ I \mapsto \textbf{inc}\ 10\}$
$\Delta\sigma_3 = \{S \mapsto \textbf{inc}\ 22,\ I \mapsto \textbf{inc}\ 15\}$
$\Delta\sigma_4 = \{S \mapsto \textbf{inc}\ 37,\ I \mapsto \textbf{inc}\ 24\}$
$\Delta\sigma_5 = \{S \mapsto \textbf{inc}\ 61,\ I \mapsto \textbf{inc}\ 40\}$
$\Delta\sigma_{out} = \{S \mapsto \textbf{inc}\ 101,\ I \mapsto \textbf{inc}\ 68\}$

(b) Differential stores for $I \mapsto \textbf{inc}\ 5$.

$\Delta\sigma_{0\ldots6} = \{n \mapsto \textbf{inc}\ 3\}$
$\sigma_7 = \{S \mapsto 499,\ I \mapsto 357,\ \beta \mapsto 0.58\}$
$\sigma_8 = \{S \mapsto 856,\ I \mapsto 663,\ \beta \mapsto 0.62\}$
$\sigma_{out} = \{S \mapsto 1519,\ I \mapsto 1282,\ \beta \mapsto 0.66\}$
$\Delta\sigma_{out} = \{S \mapsto \textbf{inc}\ 1321,\ I \mapsto \textbf{inc}\ 1150,$
$\beta \mapsto \textbf{inc}\ 0.12,\ n \mapsto \textbf{inc}\ 3\}$

(c) Differential stores for $n \mapsto \textbf{inc}\ 3$.

$\Delta\sigma_{0\ldots4} = \{n \mapsto \textbf{dec}\ 5\}$
$\sigma_{out} = \{S \mapsto 111,\ I \mapsto 71,\ \beta \mapsto 0.46\}$
$\Delta\sigma_{out} = \{S \mapsto \textbf{dec}\ 1408,\ I \mapsto \textbf{dec}\ 1211,$
$\beta \mapsto \textbf{dec}\ 0.2,\ n \mapsto \textbf{dec}\ 5\}$

(d) Differential stores for $n \mapsto \textbf{dec}\ 5$.

Figure 1 Original and differential stores during execution of the epidemiological model.

Providing the correct differential result is easy: Reexecute the program and compute the outputs' difference. The challenge for differential execution is to compute output changes efficiently by avoiding recomputations. For example, an increase of I in $\Delta\sigma_0$ does not affect β, which is why β does not occur in Figure 1b. Indeed, as we prove in Section 5, statements that do not read changed variables can be skipped during differential execution. It is shortcuts like this that we need to discover to make differential execution efficient.

For example, consider epidemiologists now want to know what happens after 9 instead of 6 days. We encode this change in $\Delta\sigma_0$ of Figure 1c. Note that we assume the prior change of I remains in place; changes occur in sequence and have to be undone explicitly if so desired. How can differential execution handle the change of n efficiently? In general, it is necessary to repeat each loop iteration to ensure input changes are propagated correctly. However, in our case, n only determines the number of iterations, but it is not used inside the loop body. We prove in Section 5 that in this case we can skip all previous iterations and directly continue with the new ones. Since iterations 7, 8, and 9 have not been executed before, we have to compute the resulting stores from scratch. The final result $\Delta\sigma_{out}$ is then defined by $\sigma_9 \ominus \sigma_6$. In general, when the original loop count n is increased by k, we only need to perform k instead of $n + k$ iterations, which yields significant speedups when $k \ll n$.

Lastly, consider epidemiologists want to rollback n from 9 to 4. This is a case where incremental computing has to trade-off memory and running time. If we still have σ_4 cached, we can compute $\Delta\sigma_{out}$ by $\sigma_4 \ominus \sigma_9$. Otherwise, we may have to do more recomputation. In our study of a formal semantics for differential execution, we ignore all concerns regarding caching. Later, in Section 6, we discuss how we implemented the formal semantics in an efficient interpreter and what caching strategies we considered.

2.2 Branch switching and worst-case complexity

A particular challenge for all kinds of incremental computing is what we call *branch switching*. A branch switch occurs when a change to the input data alters the control flow, causing a different branch of a conditional to be executed compared to the original run. For a simple example, consider this extreme case:

```
# main function with free variable x, which must be bound in the initial store
```

```
if x < 0:
    foo()
else:
    bar()
```

When `x` changes from negative to positive, the entire behavior of the program shifts: Instead of running `foo`, we end up running `bar`. This has two important consequences. First, the worst-case complexity of differential execution is at best equal to regular execution. Second, we must retain (or be able to reconstruct) the original state σ that is valid at the beginning of a branch. This state is necessary when a branch switch occurs, as input for executing the other branch. This indicates a lower bound on the memory necessary for differential execution if we want to avoid reconstructing previously seen states.

Does this mean that all hope is lost? No, not at all. Many conditionals do not amount to a reconfiguration of the program, but have a rather limited scope. For example, an improved epidemiological model may consider emergency measures, which are active when infections exceed a threshold T of 40 patients:

```
α = 0.04; γ = 0.02; S = 0; T = 40
repeat n:
    if I > T:           # do infectious cases exceed threshold T?
        β = β - γ       # emergency measures activated, transmissibility shrinks
    else:
        β = β + α       # no emergency measures, transmissibility grows
    S = S + I; I = I * 2.718^β
```

When we change the input variable I, this can now affect the control flow and lead to branch switching. However, the cost of re-executing one of the branches is limited in this case, and only results in different subsequent changes to β. One interesting issue that persists is that branch switching requires diffing between the original and the new state to determine the subsequent changes. In Section 5, we present an optimization that avoids this cost when branches execute similar code.

3 Values and Changes

While regular program execution operates on values, differential execution operates on changes. That is, changes are first-class objects in differential execution. Since the definition of changes is language-specific, we first introduce general rules for how to design well-behaved changes for a statically typed base language. We then define specific changes for our imperative language Imp.

3.1 Change validity, patching, and diffing

Changes must support two fundamental operations: *diffing* to produce changes from values ($new \ominus old$), and *patching* to apply changes to values ($base \oplus change$). Patching and diffing should act as inverses: $v_1 \oplus (v_2 \ominus v_1) = v_2$ and $(v \oplus \Delta v) \ominus v = \Delta v$. These laws are elemental for differential execution as they justify optimizations that avoid redundant work.

However, handling changes correctly is subtle and it is easy to break the second property. For example, consider $v = 2$ and $\Delta v = \mathbf{dec}\ 5$ for the naturals with $2 \oplus \mathbf{dec}\ 5 = 0$. Then,

$$(v \oplus \Delta v) \ominus v = 0 \ominus 2 = \mathbf{dec}\ 2 \neq \mathbf{dec}\ 5 = \Delta v.$$

The problem is that we tried to patch $v = 2$ with $\Delta v = \mathbf{dec}\ 5$, which is invalid because $2 - 5$ does not yield a natural number, and defaulting to 0 contradicts our inversion properties.

The same problem occurs for other data types, such as strings, sets, and lists. For example, we get (“ab” $\oplus$ **drop** 5) $\ominus$ “ab” = “” $\ominus$ “ab” = **drop** 2 $\neq$ **drop** 5. Again, we tried to apply a patch that has an undefined behavior, and defaulting to the empty string breaks our inversion laws. Therefore, we follow the incremental lambda calculus [9] and restrict patching to valid changes. But rather than using Coq's dependent typing, we rely on a dedicated typing judgment $v \vdash \Delta v$. Patching and diffing then must satisfy the following properties:

$$\frac{v : \tau \qquad v \vdash \Delta v}{v \oplus \Delta v : \tau}\ \text{T-Patch} \qquad\qquad \frac{v_1 : \tau \qquad v_2 : \tau}{v_1 \vdash v_2 \ominus v_1}\ \text{V-Diff}$$

Patching is only defined when $v \vdash \Delta v$ and yields a value of the same type as v. Diffing is defined for any v_1 and v_2 of the same type, and it yields a valid change that can be applied to v_1. When defining differential execution, it is our responsibility to define changes Δv, validity $v \vdash \Delta v$, patching $\oplus$, and diffing $\ominus$ such that these typing rules hold.

On top of these typing rules for patching and diffing, we can now stipulate the inversion properties we require.

Property 3.1 (Patch-Diff Inversion)**.** *For any well-typed values $v_1 : \tau$ and $v_2 : \tau$, patching inverts diffing: $v_1 \oplus (v_2 \ominus v_1) = v_2$.*

Property 3.2 (Diff-Patch Inversion)**.** *For any value v and valid change $v \vdash \Delta v$, diffing inverts patching: $(v \oplus \Delta v) \ominus v \equiv \Delta v$.*

Note how the latter property uses equivalence $\equiv$ on changes rather than equality. This is due to the fact that changes often have non-unique representations and diffing might choose any equivalent representation. For example, **inc** 0 and **dec** 0 are equivalent but not equal. In general, $\Delta v_1 \equiv \Delta v_2$ iff for all v, $v \vdash \Delta v_1 \leftrightarrow v \vdash \Delta v_2$ and $v \oplus \Delta v_1 = v \oplus \Delta v_2$.

The above properties justify our use of **noc**, representing no change. Specifically, for any v, we have $v \oplus (v \ominus v) = v$ and therefore $\mathbf{noc} \equiv v \ominus v$. In practice, it is usually easy to decide if a given Δv is equivalent to **noc**.

3.2 Numeric and Boolean changes

Our imperative language Imp uses unsigned integers and Boolean values. We define changes for each type:

$$\begin{array}{ll} \text{(value changes)} & \Delta v ::= \Delta n \mid \Delta b \\ \text{(numeric changes)} & \Delta n ::= \mathbf{noc}_{\mathbb{N}} \mid \mathbf{inc}\ k \mid \mathbf{dec}\ k \\ \text{(Boolean changes)} & \Delta b ::= \mathbf{noc}_{\mathbb{B}} \mid \mathbf{neg} \end{array}$$

Numeric changes consist of no change ($\mathbf{noc}_{\mathbb{N}}$), increasing (**inc** k), and decreasing (**dec** k), where k is any natural number. Boolean changes consist of no change ($\mathbf{noc}_{\mathbb{B}}$) and negation (**neg**). Changes are only valid for an appropriately typed value, and **dec** k is only valid for numbers larger or equal to k:

$$\frac{v : \mathsf{Bool}}{v \vdash \mathbf{noc}_{\mathbb{B}}} \quad \frac{v : \mathsf{Bool}}{v \vdash \mathbf{neg}} \quad \frac{v : \mathsf{Num}}{v \vdash \mathbf{noc}_{\mathbb{N}}} \quad \frac{v : \mathsf{Num}}{v \vdash \mathbf{inc}\ k} \quad \frac{v : \mathsf{Num} \qquad v \geq k}{v \vdash \mathbf{dec}\ k}$$

Whenever $v \vdash \Delta v$, we can patch v while retaining its type as required by T-Patch. Moreover, differencing ensures $v_1 \vdash v_2 \ominus v_1$ as required by V-Diff.

$$\begin{array}{ll} n \oplus \mathbf{noc}_{\mathbb{N}} = n & n \ominus n = \mathbf{noc}_{\mathbb{N}} \\ n \oplus \mathbf{inc}\ k = n + k & n \ominus m = \mathbf{inc}\ (n - m), \text{ if } n > m \\ n \oplus \mathbf{dec}\ k = n - k & n \ominus m = \mathbf{dec}\ (m - n), \text{ if } m > n \\ b \oplus \mathbf{noc}_{\mathbb{B}} = b & b \ominus b = \mathbf{noc}_{\mathbb{B}} \\ b \oplus \mathbf{neg} = \neg b & b \ominus c = \mathbf{neg}, \text{ if } b \neq c \end{array}$$

We have modeled these changes and their operations in Rocq, where we proved they satisfy the respective typing rules as well as patch-diff and diff-patch inversion.

3.3 Lifting to stores

We have established how patching and diffing operate on individual values, but our imperative programs range over variables that map to values. Therefore, we need to lift changes and their operations from single values to entire stores $\sigma : x \to v$ that map variables to values. To this end, we define differential stores $\Delta\sigma : x \to \Delta v$, which map variables to value changes. Differential stores must satisfy the same properties as differential values: well-typed patching and diffing according to T-Patch and V-Diff, such that patch-diff and diff-patch inversion hold.

We start off by requiring stores to be well-typed given a store type $\Gamma : x \to \tau$. We define store typing $\Gamma \vdash \sigma$ and differential store validity $\sigma \vdash \Delta\sigma$ pointwise:

$$\frac{\forall x.\ \sigma(x) : \Gamma(x)}{\Gamma \vdash \sigma}\ \text{T-Store} \qquad \frac{\forall x.\ \sigma(x) \vdash \Delta\sigma(x)}{\sigma \vdash \Delta\sigma}\ \text{V-Store}$$

Then the following definitions of patching and diffing satisfy T-Patch and V-Diff:

$$\begin{aligned} \sigma \ \oplus \Delta\sigma &= \lambda x.\ \sigma(x) \oplus \Delta\sigma(x) \\ \sigma_1 \ominus \sigma_2 &= \lambda x.\ \sigma_1(x) \ominus \sigma_2(x) \end{aligned}$$

Patch-diff inversion holds because

$$\sigma_1 \oplus (\sigma_2 \ominus \sigma_1) \ = \ \lambda x.\ \sigma_1(x) \oplus (\sigma_2(x) \ominus \sigma_1(x)) \ = \ \lambda x.\ \sigma_2(x) \ = \ \sigma_2$$

and conversely diff-patch inversion is due to

$$(\sigma \oplus \Delta\sigma) \ominus \sigma \ = \ \lambda x.\ (\sigma(x) \oplus \Delta\sigma(x)) \ominus \sigma(x) \ \equiv \ \lambda x.\ \Delta\sigma(x) \ = \ \Delta\sigma(x).$$

Again, we have modeled differential stores in Rocq and proved the above properties formally there. This shows how to lift changes from values to stores, which provides the foundation for our differential semantics. The Rocq formalization is provided at

4 Differential Semantics: A Foundation for Incremental Computing

Incremental computing is all about eliminating redundant work in the face of input changes. In the pursuit of efficiency, incremental computing tries to cut as many corners as possible while retaining correctness. Our differential style of incremental computing is particularly susceptible to correctness errors, because we need to capture the precise difference of each computation output. In this section, we introduce the idea of differential semantics to capture the incremental behavior of a program. Specifically, we develop a differential semantics for the imperative language Imp from Section 2 with expressions and statements.

A differential semantics establishes a sound baseline for incremental computing, but it is not usually optimal. Instead, we separately introduce optimizations for the efficient incremental execution of programs and prove them correct with respect to the baseline differential semantics. To support the modular definition and verification of optimizations, we formulate the differential semantics compositionally for each language construct. But first, let us review the standard big-step semantics of our imperative language Imp.

$$\frac{}{\sigma \vdash \textbf{true} \Rightarrow \textbf{true}} \qquad \frac{}{\sigma \vdash \textbf{false} \Rightarrow \textbf{false}} \qquad \frac{}{\sigma \vdash n \Rightarrow n}$$

$$\frac{}{\sigma \vdash x \Rightarrow \sigma(x)} \qquad \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_1 \odot e_2 \Rightarrow v_1 \odot v_2}$$

$$\frac{\sigma \vdash e \Rightarrow v}{\sigma \vdash x := e \Rightarrow [x \mapsto v]\sigma} \qquad \frac{\sigma \vdash s_1 \Rightarrow \sigma' \qquad \sigma' \vdash s_2 \Rightarrow \sigma''}{\sigma \vdash s_1; s_2 \Rightarrow \sigma''}$$

$$\frac{\sigma \vdash e \Rightarrow \textbf{true} \qquad \sigma \vdash s_1 \Rightarrow \sigma'}{\sigma \vdash \textbf{if}\ e\ s_1\ s_2 \Rightarrow \sigma'} \qquad \frac{\sigma \vdash e \Rightarrow \textbf{false} \qquad \sigma \vdash s_2 \Rightarrow \sigma'}{\sigma \vdash \textbf{if}\ e\ s_1\ s_2 \Rightarrow \sigma'}$$

$$\frac{\sigma \vdash e \Rightarrow 0}{\sigma \vdash \textbf{repeat}\ e\ s \Rightarrow \sigma} \qquad \frac{\sigma \vdash e \Rightarrow n+1 \qquad \sigma \vdash \textbf{repeat}\ n\ s \Rightarrow \sigma' \qquad \sigma' \vdash s \Rightarrow \sigma''}{\sigma \vdash \textbf{repeat}\ e\ s \Rightarrow \sigma''}$$

Figure 2 Standard big-step reduction semantics for Imp expressions and statements.

4.1 Standard big-step semantics of Imp

We already introduced the syntax of Imp expressions, statements, and stores in Section 2. Figure 2 shows the standard semantics for Imp using two judgments. For expressions, we write $\sigma \vdash e \Rightarrow v$ to denote that expression e evaluates to value v under store σ. Expressions can read but cannot change the content of the store. The last reduction rule for expressions handles binary expressions $e_1 \odot e_2$. Imp supports a few binary operators $\odot \in \{+, -, *, >, \mathsf{exp}\}$, and we assume that $v_1 \odot v_2$ follows the standard arithmetic interpretation of these operators.

For statements, we write $\sigma \vdash s \Rightarrow \sigma'$ to denote that statement s executes under store σ, yielding a possibly updated store σ'. In these rules, the notation $[x \mapsto v]\sigma$ represents the store obtained from σ by updating the mapping of variable x to value v, while leaving all other mappings unchanged. For repeat loops, we skip the body when the iteration count is zero. Otherwise, we recursively evaluate the loop with a decremented count, followed by running the body. We opted for this left-recursive loop unrolling to simplify some of the proofs.

4.2 Differential semantics for Imp expressions

A differential semantics describes how a program's output changes in reaction to input changes. The inputs of our Imp programs are stores σ. We have described store changes $\Delta\sigma$ in Subsection 3.3. The output of an Imp program is also a store, but we first only consider Imp expressions. An Imp expression computes a value v, hence the differential semantics for Imp expressions computes a value change Δv. To this end, we define a differential reduction relation $\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \Delta v$ to compute how e's output changes:

$$\frac{e \in \{\textbf{true}, \textbf{false}\}}{\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \textbf{noc}_\mathbb{B}}\ \text{D-Bool} \qquad \frac{}{\sigma, \Delta\sigma \vdash n \Rightarrow_\Delta \textbf{noc}_\mathbb{N}}\ \text{D-Num} \qquad \frac{}{\sigma, \Delta\sigma \vdash x \Rightarrow_\Delta \Delta\sigma(x)}\ \text{D-Var}$$

$$\frac{\sigma, \Delta\sigma \vdash e_1 \Rightarrow_\Delta \Delta v_1 \qquad \sigma, \Delta\sigma \vdash e_2 \Rightarrow_\Delta \Delta v_2 \qquad (\sigma \vdash e_1 \Rightarrow v_1) \qquad (\sigma \vdash e_2 \Rightarrow v_2)}{\sigma, \Delta\sigma \vdash e_1 \odot e_2 \Rightarrow_\Delta \odot_\Delta(v_1, v_2, \Delta v_1, \Delta v_2)}\ \text{D-BinOp}$$

The first rules handle an expression e that is a Boolean or numeric constant. The value of constant expressions can not change, hence the output is a no change (**noc**). A variable x changes in accordance with $\Delta\sigma$. In particular, when an input is changed, reading from that input yields its change, which can then be propagated.

The last rule handles binary operators $\odot$ and delegates their differential behavior to dedicated functions $\odot_\Delta$. We allow these functions to use the original input values (v_1, v_2) and their changes (Δv_1, Δv_2). For example, for a differential multiplication $m * (n \oplus \textbf{inc}\ k)$, we need to know the original arguments to compute the output change **inc** $(m * k)$. While our implementation supports various operators, in this paper and the Rocq formalization we focus on establishing the core differential semantics. A complete treatment of differential operators is outside the scope of this paper.

Note that we required the original store σ during the differential execution of expressions only to recover the original arguments of binary operators. In places like these, we can expect an actual implementation to cache the previous evaluation result of each operand where needed. We highlight opportunities for caching in the reduction rules by putting preconditions in parentheses (*cached*) and slightly shading their background.

4.3 A differential semantics for IMP statements

We define the differential semantics for statements using the relation

$$\sigma, \Delta\sigma \vdash s \Rightarrow_\Delta \Delta\sigma'$$

which executes statement s under differential store $\Delta\sigma$ and original store σ to yield an updated differential store $\Delta\sigma'$. The inference rules are shown in Figure 3.

Rule D-ASSIGN evaluates the assigned expressions differentially and then updates the differential store. The store update is a destructive update, overwriting the previous change assigned to x. Together with rule D-SEQ, this realizes change propagation behavior. For example, assume a program $x := a; y := x$ with $\Delta\sigma(a) = \Delta v$. Rule D-SEQ runs the assignments in order. The first assignment sets $\Delta\sigma(x) = \Delta v$ because $\sigma, \Delta\sigma \vdash a \Rightarrow_\Delta \Delta v$. Then the second assignment sets $\Delta\sigma(y) = \Delta v$ because $\sigma', \Delta\sigma' \vdash x \Rightarrow_\Delta \Delta v$ Note that D-SEQ also runs the first statement non-incrementally, but only to obtain σ', which is needed as input for s_2. The highlighting of this and other preconditions in Figure 3 indicates that the precondition's result can be retrieved from a cache in an actual implementation.

For conditional statements, we generally distinguish two situations: branch constant and branch switching. A conditional behaves branch-constant if the value of the condition does not change, handled by rules D-IF$_{\text{tt}}$ and D-IF$_{\text{ff}}$. These rules simply propagate changes to the relevant branch that was selected originally. Rules D-IF$_{\text{tf}}$ and D-IF$_{\text{ft}}$ handle branch switching, which happens when the condition's value changes. If the original value of the condition was **true** and now becomes **false**, then rule D-IF$_{\text{tf}}$ fires and performs branch switching as introduced in Subsection 2.2. Originally, we have executed s_1, but now we need to run s_2. Therefore, we execute s_2 from scratch and compute how its output differs from s_1. Rule D-IF$_{\text{ft}}$ is analogous and handles the case when the condition switches from **false** to **true**.

Finally, we provide four rules to handle repeat loops. The differential semantics for **repeat** e s depends crucially on how the iteration count changes. When the differential evaluation of e yields $\textbf{noc}_{\mathbb{N}}$, the iteration count remains the same as in the original execution. In this case, we replay the loop but only do change propagation. If the iteration count is zero ($\sigma \vdash e \Rightarrow 0$), we simply return the input differential store unchanged, as no iterations need to be replayed. For a positive count ($\sigma \vdash e \Rightarrow n + 1$), we unroll the loop recursively

$$\frac{\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \Delta v}{\sigma, \Delta\sigma \vdash x := e \Rightarrow_\Delta [x \mapsto \Delta v]\Delta\sigma}\ \text{D-Assign}$$

$$\frac{\sigma, \Delta\sigma \vdash s_1 \Rightarrow_\Delta \Delta\sigma' \quad (\,\sigma \vdash s_1 \Rightarrow \sigma'\,) \quad \sigma', \Delta\sigma' \vdash s_2 \Rightarrow_\Delta \Delta\sigma''}{\sigma, \Delta\sigma \vdash s_1; s_2 \Rightarrow_\Delta \Delta\sigma''}\ \text{D-Seq}$$

$$\frac{\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \mathbf{noc}_\mathbb{B} \quad (\,\sigma \vdash e \Rightarrow \mathbf{true}\,) \quad \sigma, \Delta\sigma \vdash s_1 \Rightarrow_\Delta \Delta\sigma'}{\sigma, \Delta\sigma \vdash \mathbf{if}\ e\ s_1\ s_2 \Rightarrow_\Delta \Delta\sigma'}\ \text{D-IF}_{\text{tt}} \qquad \frac{\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \mathbf{noc}_\mathbb{B} \quad (\,\sigma \vdash e \Rightarrow \mathbf{false}\,) \quad \sigma, \Delta\sigma \vdash s_2 \Rightarrow_\Delta \Delta\sigma'}{\sigma, \Delta\sigma \vdash \mathbf{if}\ e\ s_1\ s_2 \Rightarrow_\Delta \Delta\sigma'}\ \text{D-IF}_{\text{ff}}$$

$$\frac{\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \mathbf{neg} \quad (\,\sigma \vdash e \Rightarrow \mathbf{true}\,) \quad \sigma \oplus \Delta\sigma \vdash s_2 \Rightarrow \sigma_2 \quad (\,\sigma \vdash s_1 \Rightarrow \sigma_1\,)}{\sigma, \Delta\sigma \vdash \mathbf{if}\ e\ s_1\ s_2 \Rightarrow_\Delta \sigma_2 \ominus \sigma_1}\ \text{D-IF}_{\text{tf}} \qquad \frac{\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \mathbf{neg} \quad (\,\sigma \vdash e \Rightarrow \mathbf{false}\,) \quad \sigma \oplus \Delta\sigma \vdash s_1 \Rightarrow \sigma_1 \quad (\,\sigma \vdash s_2 \Rightarrow \sigma_2\,)}{\sigma, \Delta\sigma \vdash \mathbf{if}\ e\ s_1\ s_2 \Rightarrow_\Delta \sigma_1 \ominus \sigma_2}\ \text{D-IF}_{\text{ft}}$$

$$\frac{\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \mathbf{noc}_\mathbb{N} \quad (\,\sigma \vdash e \Rightarrow 0\,)}{\sigma, \Delta\sigma \vdash \mathbf{repeat}\ e\ s \Rightarrow_\Delta \Delta\sigma}\ \text{D-Rep-0}$$

$$\frac{\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \mathbf{noc}_\mathbb{N} \quad (\,\sigma \vdash e \Rightarrow n+1\,) \quad \sigma, \Delta\sigma \vdash \mathbf{repeat}\ n\ s \Rightarrow_\Delta \Delta\sigma' \quad (\,\sigma \vdash \mathbf{repeat}\ n\ s \Rightarrow \sigma'\,) \quad \sigma', \Delta\sigma' \vdash s \Rightarrow_\Delta \Delta\sigma''}{\sigma, \Delta\sigma \vdash \mathbf{repeat}\ e\ s \Rightarrow_\Delta \Delta\sigma''}\ \text{D-Rep+1}$$

$$\frac{\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \mathbf{dec}\ k \quad (\,\sigma \vdash e \Rightarrow n\,) \quad \sigma, \Delta\sigma \vdash \mathbf{repeat}\ (n-k)\ s \Rightarrow_\Delta \Delta\sigma' \quad (\,\sigma \vdash \mathbf{repeat}\ (n-k)\ s \Rightarrow \sigma'\,) \quad (\,\sigma \vdash \mathbf{repeat}\ n\ s \Rightarrow \sigma''\,)}{\sigma, \Delta\sigma \vdash \mathbf{repeat}\ e\ s \Rightarrow_\Delta (\sigma' \oplus \Delta\sigma') \ominus \sigma''}\ \text{D-Rep-Dec}$$

$$\frac{\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \mathbf{inc}\ k \quad (\,\sigma \vdash e \Rightarrow n\,) \quad \sigma, \Delta\sigma \vdash \mathbf{repeat}\ n\ s \Rightarrow_\Delta \Delta\sigma' \quad (\,\sigma \vdash \mathbf{repeat}\ n\ s \Rightarrow \sigma'\,) \quad \sigma' \oplus \Delta\sigma' \vdash \mathbf{repeat}\ k\ s \Rightarrow \sigma''}{\sigma, \Delta\sigma \vdash \mathbf{repeat}\ e\ s \Rightarrow_\Delta \sigma'' \ominus \sigma'}\ \text{D-Rep-Inc}$$

Figure 3 Differential semantics for Imp statements. Marked (preconditions) can be cached.

followed by a differential run of the loop body s. The resulting $\Delta\sigma''$ captures the cumulative effect of differentially executing all iterations.

When the iteration count is decreased ($\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \mathbf{dec}\ k$), the process is more complicated. First, we propagate the changes for $(n-k)$ iterations to obtain $\Delta\sigma'$. Then we retrieve the original output after $(n-k)$ iterations σ' and patch it with $\Delta\sigma'$. This represents the updated output after $(n-k)$ iterations, but we need to know how it changed compared to n iterations. To this end, we compute the difference with σ'', which is the original output after n iterations.

Lastly, when the iteration count grows ($\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \mathbf{inc}\ k$), we use a two-phase strategy. First, we execute the original n iterations differentially to obtain $\Delta\sigma'$. However, from here on out, we are dealing with iterations that have not occurred before. Therefore, in the second phase, we execute the additional k iterations non-incrementally. To this end, we compute the updated output after n iterations using $\Delta\sigma'$ and feed it as input for the remaining k

iterations. Finally, we compute the difference with σ', which was the original output.

Example. Consider the following program and stores: repeat x $\{y = y + z;\ z = z + 1\}$ with initial store $\sigma = \{x \mapsto 5,\ y \mapsto 7,\ z \mapsto 3\}$. The original output is $\sigma' = \{x \mapsto 5,\ y \mapsto 32,\ z \mapsto 8\}$. Now consider we apply the following input changes $\Delta\sigma = \{x \mapsto \textbf{inc}\ 2,\ y \mapsto \textbf{inc}\ 3,\ z \mapsto \textbf{inc}\ 5\}$. The example demonstrates how differential execution handles both loop counter changes (x) and variable changes (y, z). Specifically, here rule D-Rep-Inc is leading the execution and proceeds in the following steps:

1. Execute original 5 iterations differentially: $\Delta\sigma' = \{x \mapsto \textbf{inc}\ 2,\ y \mapsto \textbf{inc}\ 28,\ z \mapsto \textbf{inc}\ 5\}$.
2. Compute patched state after 5 iterations: $\sigma' \oplus \Delta\sigma' = \{x \mapsto 7,\ y \mapsto 60,\ z \mapsto 13\}$.
3. Execute new iterations (k = 2) from $\sigma' \oplus \Delta\sigma'$ to obtain $\sigma'' = \{x \mapsto 7,\ y \mapsto 87,\ z \mapsto 15\}$.
4. Compute differential output: $\sigma'' \ominus \sigma' = \{x \mapsto \textbf{inc}\ 2,\ y \mapsto \textbf{inc}\ 55,\ z \mapsto \textbf{inc}\ 7\}$.

4.4 Properties of differential semantics

Our differential semantics aims to replace recomputation with change-based execution. That is, when the original input σ is changed to $\sigma' = \sigma \oplus \Delta\sigma$, the differential semantics must be able to process $\Delta\sigma$ directly. For this to work, we need three guarantees:

Completeness: Whenever a program can execute under the original σ and the updated σ', then the differential semantics must derive an output change for $\Delta\sigma$.

Output validity: The output change must be valid for the original output.

From-scratch consistency Patching the original output with the output changes is consistent with a recomputation starting with σ'.

That is: We can obtain changes using the differential semantics, the changes can be used to patch the original output, and the patched output is correct. We have formalized these properties in Rocq and have a mechanized proof that shows our differential semantics satisfies them. Here, we only present the formal formulation of the properties.

In general, we only make claims about well-typed programs $\Gamma \vdash s$. The typing relation for statements and expressions is standard: variables are global and cannot change type. With this, we can formulate the properties of differential semantics.

▶ **Theorem 1** (Completeness). *Given a well-typed program $\Gamma \vdash s$, well-typed store $\Gamma \vdash \sigma$, and valid differential store $\sigma \vdash \Delta\sigma$. Whenever $\sigma \vdash s \Rightarrow \theta$ and $\sigma \oplus \Delta\sigma \vdash s \Rightarrow \theta'$ for arbitrary θ and θ', then $\sigma, \Delta\sigma \vdash s \Rightarrow_\Delta \Delta\sigma'$ for some $\Delta\sigma'$.*

Proof sketch. By induction on the typing derivation of statement s. For base cases like Assign, we directly construct differential derivations. For compound statements like Seq and If, we apply induction hypotheses to substatements and compose their results. The critical case is Repeat, where we handle changes to the iteration count by carefully tracking execution states for each iteration. ◀

Completeness ensures we find output changes $\Delta\sigma'$. Note that for our Imp language, the original and update execution always succeeds for well-typed programs and stores, but richer languages may include run-time errors and divergence For Imp, completeness ensures that we have defined enough reduction rules to handle all well-typed execution states.

Validity guarantees that differential execution produces changes that are compatible with the values they will be patched with. This ensures we can safely apply all changes produced during execution.

▶ **Theorem 2** (Output validity). *Given a well-typed program $\Gamma \vdash s$, well-typed store $\Gamma \vdash \sigma$, and valid differential store $\sigma \vdash \Delta\sigma$. Whenever $\sigma \vdash s \Rightarrow \sigma'$ and $\sigma, \Delta\sigma \vdash s \Rightarrow_\Delta \Delta\sigma'$, then $\sigma' \vdash \Delta\sigma'$ is valid for the original output.*

Proof sketch. By induction on the differential semantics derivation, showing each rule preserves validity when input changes are valid. Critical cases include branch switching and loop optimizations, where the type system ensures changes remain compatible with their target values. ◀

From-scratch consistency is our key correctness property—it proves that differential execution produces the same final results as re-running the program with changed inputs. This means users can trust that differential execution is just an optimization that does not affect program behavior.

▶ **Theorem 3** (From-Scratch Consistency). *Given a well-typed program* $\Gamma \vdash s$*, well-typed store* $\Gamma \vdash \sigma$*, and valid differential store* $\sigma \vdash \Delta\sigma$*. Whenever* $\sigma \vdash s \Rightarrow \sigma'$ *and* $\sigma, \Delta\sigma \vdash s \Rightarrow_\Delta \Delta\sigma'$*, then* $\sigma \oplus \Delta\sigma \vdash s \Rightarrow \sigma' \oplus \Delta\sigma'$*.*

Proof sketch. By induction on differential semantics, proving patched original outputs match recomputation results. The key insight relies on our change operations satisfying algebraic properties like patch-diff and diff-patch inversion, ensuring consistency across all language constructs. ◀

We have formally verified all three theorems using the Rocq proof assistant. The complete mechanized proofs are available in the repository referenced in Section 3.

4.5 Mechanized Rocq formalization

We formalized the differential semantics of Imp in Rocq and proved completeness, output validity, and from-scratch consistency. The Rocq code is available open-source.[2] The Rocq formalization consists of approximately 1,450 lines of code and was important in shaping our theory of changes. A key challenge emerged when proving the diff-patch inversion property (Property 3.2) introduced in Section 3. While standard equality was sufficient for proving the patch-diff property (Property 3.1), we discovered it was inadequate for the diff-patch case. We needed a custom equivalence relation because multiple differential values (noc, incr 0, decr 0) can represent the same change semantically

The consistency proof (Theorem 3) was particularly challenging, especially for the increment case of repeat statements. The complexity of this case revealed that our initial induction hypothesis wasn't strong enough — an observation we only gained after successfully proving all other cases. This led to a reformulation of the theorem with a stronger induction hypothesis.

In our opinion, for the number of results we proved, our Rocq code is reasonably compact. This was achieved through careful proof automation: we developed custom tactics for common proof patterns and registered them as hints for Rocq's eauto system. This automation strategy significantly reduced the proof size.

5 Optimizations as Alternative Reduction Rules

To enhance the efficiency of our differential semantics, we introduce several optimizations which allow us to avoid unnecessary computations during execution. We present them as alternative reduction rules within our differential semantics. Each optimization we present is a shortcut: it lets us compute the same result more efficiently. To ensure these shortcuts are

[2] `https://gitlab.rlp.net/plmz/artifacts/autoinc-interp-formalization-ecoop25`

correct, we must prove they are *admissible*. This means that whenever we use an optimization rule to derive a result, we can also derive the same result (perhaps less efficiently) using only the standard rules from Figure 3. Formally, we define admissibility of a rule as follows.

▶ **Definition 4** (Rule Admissibility). *An optimization rule R with premises P and conclusion $\sigma, \Delta\sigma \vdash s \Rightarrow_\Delta \Delta\sigma'$ is admissible if for any well-typed statement s, whenever P holds, there exists a derivation D using only standard rules that concludes $\sigma, \Delta\sigma \vdash s \Rightarrow_\Delta \Delta\sigma'$.*

Admissibility ensures optimizations are behavior-preserving shortcuts. In the following sections, we present several optimizations. We have proved the admissibility for each of these rules in Rocq, working only with well-typed programs since well-typedness is essential for ensuring the soundness of our optimizations. We categorize the various optimizations into three main types: *short-circuiting optimizations*, *loop optimizations*, and *branch switching optimizations*, and discuss each in detail.

5.1 Short-Circuiting Rules

Short-circuiting rules enable bypassing differential computations entirely under certain conditions. We begin with a special case where all variables of a store map to no change. Then we generalize this rule to handle programs where only referenced variables are unchanged.

$$\frac{\Gamma \vdash s \qquad \sigma \vdash s \Rightarrow \sigma' \qquad \forall x.\ \Delta\sigma(x) = \mathbf{noc}}{\sigma, \Delta\sigma \vdash s \Rightarrow_\Delta \Delta\sigma}\ \varepsilon\text{-Store}$$

The ε-Store rule shows that when every variable in the differential store maps to **noc**, differential execution preserves the input differential store unchanged. This optimization is useful because when no variables have changed at all, we can completely bypass differential execution, avoiding differential computations entirely. The rule is fundamentally sound since a program operating on unchanged inputs must produce unchanged outputs. The premise $\sigma \vdash s \Rightarrow \sigma'$ in the rule establishes that s executes successfully in the standard semantics, which is necessary for proving the rule's admissibility.

Consider the following program:

```
x = y + z
w = v * 2
```

With the differential store $\Delta\sigma = \{x \mapsto \mathbf{noc}, y \mapsto \mathbf{noc}, z \mapsto \mathbf{noc}, w \mapsto \mathbf{noc}, v \mapsto \mathbf{noc}\}$, the rule lets us skip differential execution entirely.

To handle more realistic scenarios where a differential store might contain changes to many variables but a specific program fragment only references unchanged variables, we need to precisely characterize when a fragment is unaffected by changes. We do this through the No-Change relation.

▶ **Definition 5** (No-Change Relation). *The no-change relation for expressions $\Gamma, \Delta\sigma \vdash^0_e e$ and statements $\Gamma, \Delta\sigma \vdash^0_s s$ indicates that under typing context Γ and differential store $\Delta\sigma$, all variables referenced in e or s map to $\mathbf{noc}_\tau$ of their respective type τ in Γ. This relation is defined inductively by the rules in Figure 4.*

Using this relation, we formulate our second optimization rule as follows:

$$\frac{\Gamma, \Delta\sigma \vdash^0_s s \qquad \Gamma \vdash s}{\sigma, \Delta\sigma \vdash s \Rightarrow_\Delta \Delta\sigma}\ \text{No-Change}$$

$$\frac{}{\Gamma, \Delta\sigma \vdash_e^0 \ \textbf{true}} \qquad \frac{}{\Gamma, \Delta\sigma \vdash_e^0 \ \textbf{false}} \qquad \frac{}{\Gamma, \Delta\sigma \vdash_e^0 \ n} \qquad \frac{\tau = \Gamma\ x \quad \Delta\sigma(x) = \textbf{noc}_\tau}{\Gamma, \Delta\sigma \vdash_e^0 \ x}$$

$$\frac{\tau = \Gamma\ x \quad \Delta\sigma(x) = \textbf{noc}_\tau \quad \Gamma, \Delta\sigma \vdash_e^0 \ e}{\Gamma, \Delta\sigma \vdash_s^0 \ x := e} \qquad \frac{\Gamma, \Delta\sigma \vdash_s^0 \ s_1 \quad \Gamma, \Delta\sigma \vdash_s^0 \ s_2}{\Gamma, \Delta\sigma \vdash_s^0 \ s_1; s_2}$$

$$\frac{\Gamma, \Delta\sigma \vdash_e^0 \ e \quad \Gamma, \Delta\sigma \vdash_s^0 \ s_1 \quad \Gamma, \Delta\sigma \vdash_s^0 \ s_2}{\Gamma, \Delta\sigma \vdash_s^0 \ \textbf{if}\ e\ s_1\ s_2} \qquad \frac{\Gamma, \Delta\sigma \vdash_e^0 \ e \quad \Gamma, \Delta\sigma \vdash_s^0 \ s}{\Gamma, \Delta\sigma \vdash_s^0 \ \textbf{repeat}\ e\ s}$$

Figure 4 Inference rules for expressions and statements experiencing no change under $\Delta\sigma$.

The No-Change rule generalizes the ε-Store rule: when no variables referenced in s are affected by changes ($\Gamma, \Delta\sigma \vdash_s^0 \ s$), any differential execution of s must preserve the input differential store. This optimization is particularly valuable as it allows us to skip computations even when some variables in the program have changed, as long as they aren't referenced in the code being optimized.

Consider our earlier program fragment in a larger context:

```
x = y + z
w = v * 2
# ... later code using 'a' ...
```

With differential store $\Delta\sigma = \{x \mapsto \textbf{noc}, y \mapsto \textbf{noc}, z \mapsto \textbf{noc}, w \mapsto \textbf{noc}, v \mapsto \textbf{noc}, a \mapsto \textbf{inc}\ 5\}$, the No-Change rule lets us skip the first two assignments even though a changes elsewhere in the program. This local reasoning about variable usage avoids wasteful expression evaluation - we don't compute changes that would inevitably be **noc**.

5.2 Loop Optimization Rules

Loop constructs often dominate execution time in incremental computations. We present three optimization rules for loops, each handling a different scenario where full recomputation can be avoided. We start with loop idempotence, then present rules for handling changes to iteration counts.

$$\frac{\begin{array}{ll} & n \geq 0 \\ \Gamma \vdash \sigma & \sigma \vdash e \Rightarrow n \\ \sigma \vdash \Delta\sigma & \sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \textbf{noc} \qquad \sigma \vdash s \Rightarrow \sigma \qquad \sigma, \Delta\sigma \vdash s \Rightarrow_\Delta \Delta\sigma \end{array}}{\sigma, \Delta\sigma \vdash \textbf{repeat}\ e\ s \Rightarrow_\Delta \Delta\sigma}\ \textsc{Loop-Idemp}$$

The Loop-Idemp rule provides a significant optimization opportunity: when we can prove that a loop body preserves both the store and differential store in a single iteration and there is no change in the number of loop iterations, we can completely skip executing the remaining iterations. This is valuable because without this rule, we would need to execute each iteration even though we know the final result would be unchanged.

Consider a loop with operations that cancel out:

```
repeat n:
    x = (x * 1) + (y - y)    # x remains unchanged through identities
    y = (y + x) - x          # y remains unchanged through inverse operations
```

When $\Delta\sigma = \{x \mapsto \textbf{noc}, y \mapsto \textbf{noc}\}$, we can skip all iterations since each iteration preserves the store through these reversing computations. While this example is deliberately simple, the Loop-Idemp rule becomes particularly powerful with richer language features. For instance, in a language with arrays and objects, this rule would recognize loops that appear to modify data structures but actually preserve their state, allowing us to skip such loops entirely when we can prove that their operations are self-cancelling.

Our two most powerful loop optimizations address scenarios where the loop count changes, but the loop body remains unaffected by changes in the differential store. That is, all variables in the loop are assigned **noc** in the input differential store. This optimization reduces computational effort by focusing only on the additional iterations introduced by the change in the loop count, thereby avoiding redundant execution of the unchanged loop body.

$$\frac{\begin{array}{lll}\Gamma, \Delta\sigma \vdash^0_s\ s & & \sigma \vdash \textbf{repeat}\ e\ s \Rightarrow \sigma' \\ \Gamma \vdash s & \sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \textbf{inc}\ m & \sigma' \vdash \textbf{repeat}\ m\ s \Rightarrow \sigma''\end{array}}{\sigma, \Delta\sigma \vdash \textbf{repeat}\ e\ s \Rightarrow_\Delta (\sigma'' \oplus \Delta\sigma) \ominus \sigma'}\ \textsc{Loop-Incr}$$

The Loop-Incr rule handles increases in iteration count. When the loop count increases by m iterations and the loop body references no changed variables ($\Gamma, \Delta\sigma \vdash^0_s\ s$), we can optimize by doing the following.

- Reusing the result σ' of the original execution
- Computing only the additional m iterations from σ' to get σ''
- Computing the differential results as $(\sigma'' \oplus \Delta\sigma) \ominus \sigma'$

The differential output captures how the loop's output differs after running the additional iterations. With cached results from the original execution, we only compute the m new iterations: $\sigma' \vdash \textbf{repeat}\ m\ s \Rightarrow \sigma''$ - precisely the work that must be done. This optimization avoids redoing unchanged iterations while still performing the genuinely new computation. However, when any variable read in the loop body changes (for example, if an array being summed changes), we cannot apply this optimization. This restriction is necessary - changes to variables in the loop body would produce different results in every iteration, invalidating our assumption that earlier iterations remain unchanged and in process invalidating the cached result σ'.

Consider the following example where we accumulate a sum with a variable count:

```
sum = 0
i = 0
repeat n:  # n increases from 5 to 7
  sum = sum + i
  i = i + 1
```

With $\Delta\sigma = \{n \mapsto \textbf{inc}\ 2, \texttt{sum} \mapsto \textbf{noc}, i \mapsto \textbf{noc}\}$, both `sum` and `i` are unchanged in the input store, allowing us to safely reuse cached results. After the original 5 iterations, we compute just 2 more iterations with values we haven't seen before. If instead we had $\Delta\sigma = \{n \mapsto \textbf{inc}\ 2, i \mapsto \textbf{inc}\ 1\}$, the change to `i` would affect every iteration's computation, forcing us to recompute all iterations with the new initial value.

When all premises remain as in the increment case, but with $\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \textbf{dec}\ m$ replacing the increment, we obtain an elegant formulation for the decrement case as shown below.

$$\frac{\begin{array}{lll}\Gamma, \Delta\sigma \vdash^0_s\ s & \sigma \vdash e \Rightarrow n & \sigma \vdash \textbf{repeat}\ e\ s \Rightarrow \sigma' \\ \Gamma \vdash s & \sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \textbf{dec}\ m & \sigma \vdash \textbf{repeat}\ (n - m)\ s \Rightarrow \sigma''\end{array}}{\sigma, \Delta\sigma \vdash \textbf{repeat}\ e\ s \Rightarrow_\Delta (\sigma'' \oplus \Delta\sigma) \ominus \sigma'}\ \textsc{Loop-Decr}$$

For decreased iteration counts, the Loop-Decr rule is even more efficient than the increment case. If we have cached the original computation, we have access to both the original output σ' and the intermediate store σ'' after the $(n-m)^{th}$ iteration. Then, we can compute the final result directly with just a patch and a diff operation, without any loop execution. This makes the decrement case even more efficient than the increment case, where we still needed to perform m additional iterations. Again, this optimization requires $\Gamma, \Delta\sigma \vdash^0_s s$, as changes to loop body variables would invalidate our cached states.

Using the same summation example but with n decreasing from 5 to 3 and $\Delta\sigma = \{n \mapsto \textbf{dec}\ 2, \texttt{sum} \mapsto \textbf{noc}, i \mapsto \textbf{noc}\}$, we don't need to compute anything new from the normal execution. We can directly use our cached state after 3 iterations to compute the change.

In both the increment and the decrement case, we see that caching the original computation is important for efficiency. We discuss various strategies for caching computations in Section 6.1.

5.3 Branch Switching Rules

One of the main challenges in differential execution is handling branch switches efficiently, as we typically need to execute both branches and compare their stores. Consider a common pattern in imperative programs, where an if-statement contains only assignments to the same variable in both branches:

```
if (y) {
    x = z + 1
} else {
    x = z - 1
}
```

With branch switching, the standard differential semantics would compute the result by diffing the resulting stores corresponding to the two different branches that are taken. However, since only variable x changes between branches, we can update just one entry instead of the entire store. This targeted update of x based on the expressions' difference provides orders of magnitude in speedup over store-wide diffing, especially pronounced for large stores. We develop two optimizations here based on this insight.

$$\frac{\begin{array}{lll} & & \Gamma \vdash \textbf{if}\ e\ (x := e_1)\ (x := e_2) \\ & \sigma \vdash e \Rightarrow \textbf{true} & \sigma \vdash e_1 \Rightarrow v_1 \\ \Gamma \vdash \sigma & \sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \Delta v & \sigma \oplus \Delta\sigma \vdash e_2 \Rightarrow v_2 \\ \sigma \vdash \Delta\sigma & \textbf{true} \oplus \Delta v = \textbf{false} & \sigma, \Delta\sigma \vdash \textbf{if}\ e\ (x := e_1)\ (x := e_2) \Rightarrow_\Delta \Delta\sigma' \end{array}}{\Delta\sigma' \equiv [x \mapsto v_2 \ominus v_1]\Delta\sigma}\ \textsc{If-T-F}$$

The If-T-F rule handles the case where a condition switches from true to false. The rule requires both stores to be well-typed and the conditional statement itself to be well-typed. When the conditional expression e switches from true to false and both branches are assignments to the same variable x, we can directly compute the change to x as $v_2 \ominus v_1$, where v_1 is e_1's value in the original store and v_2 is e_2's value in the patched store.

For our example, suppose y was initially **true** and we have $\Delta\sigma = \{y \mapsto \textbf{neg}, x \mapsto \textbf{noc}, z \mapsto \textbf{noc}\}$ causing a branch switch. Instead of recomputing the entire store, we only need to update the entry corresponding to x as $(z-1) \ominus (z+1) = \textbf{dec}\ 2$. However, if branches modified different variables (e.g., `x = z + 1` vs `y = z - 1`), this optimization cannot apply - the changes aren't localized to a single variable.

$$\dfrac{\begin{array}{lll} & & \Gamma \vdash \textbf{if}\ e\ (x := e_1)\ (x := e_2) \\ & \sigma \vdash e \Rightarrow \textbf{false} & \sigma \vdash e_2 \Rightarrow v_2 \\ \Gamma \vdash \sigma & \sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \Delta v & \sigma \oplus \Delta\sigma \vdash e_1 \Rightarrow v_1 \\ \sigma \vdash \Delta\sigma & \textbf{false} \oplus \Delta v = \textbf{true} & \sigma, \Delta\sigma \vdash \textbf{if}\ e\ (x := e_1)\ (x := e_2) \Rightarrow_\Delta \Delta\sigma' \end{array}}{\Delta\sigma' \equiv [x \mapsto v_1 \ominus v_2]\Delta\sigma}\ \textsc{If-F-T}$$

The If-F-T rule handles the symmetric case where the condition switches from false to true. Its structure mirrors the true-to-false case, but computes $v_1 \ominus v_2$ since we are switching to the first branch.

This optimization significantly improves performance when stores are large, but changes are localized. In our Bellman-Ford implementation (Figure 5), this optimization avoids diffing the entire shortest-path table on each branch switch within the inner loop's conditional, leading to substantial speedups as shown in Section 7.

6 Implementing a Differential Interpreter

In this section, we describe the implementation of a differential interpreter based on the differential semantics. The implementation is written in Scala 3, is open source and available at `https://gitlab.rlp.net/plmz/artifacts/autoinc-interp-implementation-ecoop25`.

6.1 Caching to avoid Re-Computation

The differential semantics requires access to previous execution states when handling control flow constructs. Consider an if statement **if** $e\ s_1\ s_2$ where branch switching occurs, that is, when the condition's value changes from true to false or vice versa. We reiterate the relevant rule D-IF$_{\text{tf}}$ from Figure 3, highlighting in gray the computations that can be cached from the previous execution:

$$\dfrac{\sigma, \Delta\sigma \vdash e \Rightarrow_\Delta \textbf{neg} \qquad (\,\sigma \vdash e \Rightarrow \textbf{true}\,) \qquad \sigma \oplus \Delta\sigma \vdash s_2 \Rightarrow \sigma_2 \qquad (\,\sigma \vdash s_1 \Rightarrow \sigma_1\,)}{\sigma, \Delta\sigma \vdash \textbf{if}\ e\ s_1\ s_2 \Rightarrow_\Delta \sigma_2 \ominus \sigma_1}\ \text{D-IF}_{\text{tf}}$$

Applying this rule requires performing new computations while accessing cached results from the previous execution. Specifically, we must differentially evaluate the condition to get **neg** and execute the new branch s_2, while we can reuse the original condition evaluation result and the result of executing s_1. A similar requirement exists for repeat statements, which need stores from previous iterations to enable differential execution. This dependency on previous states explains why our differential reduction relation $\sigma, \Delta\sigma \vdash s \Rightarrow_\Delta \Delta\sigma'$ takes the previous store σ as input.

Re-computing previous results and propagating old states throughout differential execution incurs significant performance overhead. To address this, we develop an *initializing interpreter* that caches execution states. However, uniquely identifying cached states requires more than just statement identifiers, particularly for loops. Consider **repeat** 5 s where s executes five times—each iteration operates on a different store and thus requires its own cached state.

We solve this through path-sensitive identifiers that precisely track loop nesting structure. A path P starts at root R (nesting level 0) and records each enclosing loop's iteration count. For example, path $R, 2, 1$ indicates execution within the first iteration of an inner loop, nested within the second iteration of an outer loop. Our cache maps each execution point to its corresponding program state: **cache** : **UID** $\times$ **P** $\mapsto$ **S**, where UID uniquely identifies

statements in the program, P captures the loop nesting context through iteration counts, and S represents the program state we need to cache.

Our caching strategy for conditional statements preserves three key pieces: the store before evaluating the condition, the condition's evaluated result, and the store after executing the chosen branch. This enables direct access to previously computed states, eliminating redundant recomputation during differential execution.

For repeat statements, we implement two strategies: a basic strategy that caches the initial store, iteration count, and final store, and an enhanced strategy that additionally caches intermediate states after each iteration. The enhanced strategy enables efficient implementation of the decrement optimization (Section 5.2) by providing direct access to any intermediate store from the original execution, eliminating the need for additional computation.

6.2 Additional Language-Features

We have implemented a differential interpreter that supports more constructs than present in the formalization. In addition to integer and boolean values, we also support string values. In the same vein, the language supports more than just the addition operator. It can call arbitrary operators based on an operator interface which requires an `init` function and an `inc` function:

```
trait Operator[T, U]:
  def init(x: T): U
  def inc(dx: Change[T]): Change[U]
```

The type `Change[T]` marks a change of value type `T`. Note that operators themselves can keep track of local state that is required for implementing efficient incremental operators. Additionally, the language enables first-order function calls of user-defined top-level functions. Each function ranges over its own local store.

One major difference is that the language supports a mutable 2-dimensional table ranging over integers which is global to all functions. To this end, the language supports a table-write statement $\mathbf{table}[i, j] = e$ and a table-read expression $\mathbf{table}[i, j]$. Note that we currently assume that the indices cannot change between the initialization and a differential run. This is a restriction we want to address in the future work.

6.3 Applying Optimizations

To enable efficient differential execution, we implemented various optimizations. We describe optimizations in the formalization as overlapping rules, where we decide manually which rule to apply. Our implementation always applies the optimization rule whenever possible. One could imagine that there are multiple applicable optimization rules. This leads to *optimization strategies* which select the optimization rule to apply at a given point at execution time. We want to investigate optimization strategies in future work.

We implemented the short-circuiting optimizations from Section 5.1 by preemptively computing read/written variables within statements and expressions. Because our language supports a 2-dimensional global table, this analysis extends to tracking table cell accesses. Building on the branch switching optimization from Section 5.3, we handle table operations efficiently:

```
if (c) {
  table[i, j] = e1
} else {
  table[i, j] = e2
}
```

```
def bellmanFord(numNodes: Int, src: Int): Unit {
  decl i: Int, u: Int, v: Int, w: Int
  i = 1
  repeat numNodes {
    table[0, i] = Int.Max
    i = i + 1
  }
  table[0, src] = 0
  repeat numNodes - 1 {
    u = 1
    repeat numNodes {
      v = 1
      repeat numNodes {
        w = table[u, v]
        if (w != 0 && table[0, v] > table[0, u] + w) {
          table[0, v] = table[0, u] + w
        } else {
          skip
        }
        v = v + 1
      }
      u = u + 1
    }
  }
}
```

Figure 5 Bellman-Ford Algorithm implemented in our language

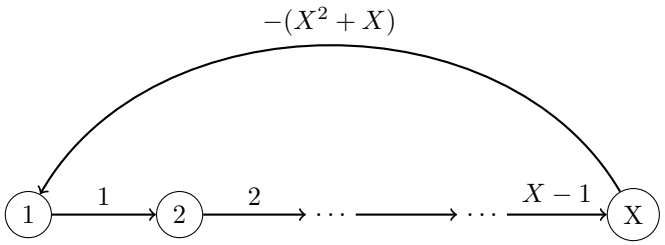
When branches differ only in their expressions (e_1 vs e_2), we can avoid expensive table diffing operations by directly computing $\Delta t[(i,j) \mapsto v_2 \ominus v_1]$ where $\sigma \vdash e1 \Rightarrow v_1$ and $\sigma \oplus \Delta\sigma \vdash e2 \Rightarrow v_2$.

7 Case Study: Shortest-Path with Bellman-Ford Algorithm

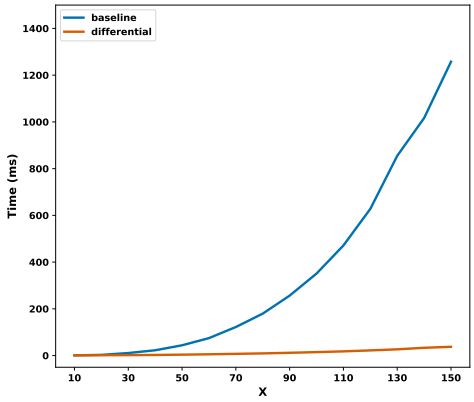
To evaluate how precise change descriptions enable efficient incremental computation of dynamic algorithms, we implemented the Bellman-Ford single-source shortest-path algorithm [6] in our language (Figure 5). Our implementation represents the graph using an adjacency matrix stored in a global 2-dimensional table, with shortest path weights from source node `src` to node i stored at indices $(0, i)$.

7.1 Setup

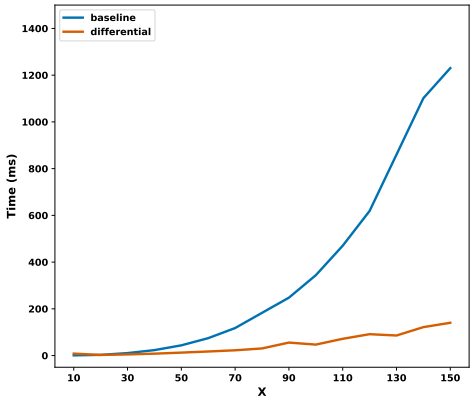
To evaluate our differential interpreter, we designed a graph structure that exhibits interesting incremental behavior. The graph is a cycle where we can configure its size by setting the number of nodes $X \in \{10, 30, \ldots, 130, 150\}$:



The weight of the back edge $X \to 1$ is chosen to introduce new shortest paths, ensuring that small input changes can trigger significant output changes. We compute shortest paths starting from node 1 for different values of X and evaluate two types of changes:



(a) Differential vs Baseline - Change (i)



(b) Differential vs Baseline - Change (ii)

Figure 6 Running times of Differential Interpretation and Normal Interpretation. Differential execution achieves significant speedups by avoiding redundant work.

(i) Adding an edge $1 \to (X-1)$ with weight 10^9. This introduces an alternative path that does not affect the shortest path solution.

(ii) Adding six edges forming cross-connections: $1 \to (X-1)$, $2 \to (X-2)$, $6 \to (X-2)$, $(X-5) \to 6$, $(X-3) \to 3$, and $(X-1) \to 2$, each with weight 10^9. Despite creating multiple new paths, these additions do not affect the shortest path solution.

We conducted experiments on an Apple M4 Pro chip with 24GB memory running 64-bit OSX 15.1.1 and OpenJDK 23.0.1. Using the JMH benchmarking framework[3], we performed 5 warmup runs followed by 10 measurement runs for each configuration.

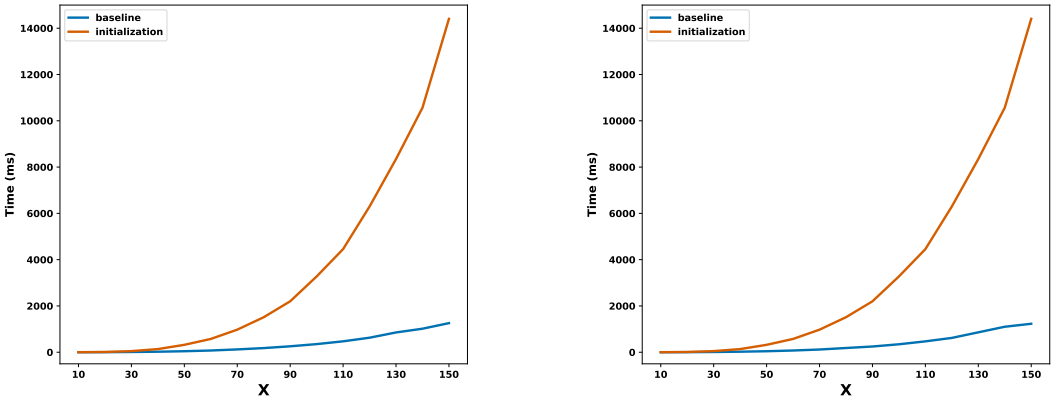
7.2 Results

In Figure 6, we compare the running times of differential interpretation against normal interpretation on the patched input. On the left we show the running times for change (i) and on the right we show them for change (ii). The differential interpreter outperforms normal interpretation for change (i). While Bellman-Ford has a time complexity of $O(X^3)$, the differential interpreter can skip most iterations because we employ an optimization that skips loop iterations when the changed table entries are not affected by the iteration. Since change (i) modifies just a single table entry without introducing new shortest paths, we can skip almost all iterations, yielding significant running time savings. The same benefits apply to change (ii), though to a lesser degree as fewer iterations are skippable.

A second critical optimization avoids diffing when branch switches occur, significantly reducing the differential interpreter's running time. Without this optimization, all branch switches for the inner if statement within the nested repeat loops would require diffing the old and new result tables, incurring an $O(X^2)$ cost. Instead, we apply constant-time updates to the current delta table.

To evaluate stability over extended use, we executed 40 consecutive changes (Change (ii)) on graphs of varying sizes (10-150 nodes). We measured performance at five points in this sequence (1^{st}, 10^{th}, 20^{th}, 30^{th}, and 40^{th} change), with 5 warmup and 10 measurement

[3] `https://github.com/openjdk/jmh`



(a) Initialization vs Baseline - Change (i)

(b) Initialization vs Baseline - Change (ii)

Figure 7 Running times of Initializing Interpretation and Normal Interpretation. The initial run incurs substantial overhead due to caching.

runs per point. After filtering outliers using the interquartile range (IQR) method, our analysis confirms that execution times remain consistent across the change sequence for each graph size. The Coefficient of Variation (CV), which measures relative variability as standard deviation divided by mean, is remarkably low (average 8%, maximum 13%), indicating that our differential approach maintains stable performance throughout extended use without degradation. While execution times scale with graph size as expected, the consistent performance within each size category demonstrates that repeated incremental updates do not compromise efficiency, further justifying the one-time initialization cost.

In Figure 7 we compare the running times of the initializing interpreter to normal interpretation on the initial input. On the left we show the running times for change (i) and on the right for change (ii). The graphs show that initialization requires significantly more time than normal interpretation. Normal interpretation takes roughly 1.5s for processing the initial graph where $X = 150$. In contrast, the initialization interpreter requires roughly 15s, a 10x increase. This overhead comes from caching states by inserting entries into hash tables. This overhead is an implementation artifact that we believe can be substantially reduced through engineering improvements and smarter caching strategies.

These results demonstrate that differential interpretation of dynamic algorithms like Bellman-Ford is feasible and can achieve order-of-magnitude speedups, provided we have: 1) optimizations that reduce work by skipping unaffected program fragments, enabled by precise change descriptors, and 2) efficient caching strategies to minimize initialization overhead. Additionally, reducing initialization time remains an important area for improvement.

8 Related Work

Incremental computation has been extensively studied, with various approaches aiming to efficiently update outputs in response to changes in inputs. In this section, we discuss work related to our approach, namely dynamic algorithms, incremental computing by selective recomputing, incremental computing by differential updating, and the incremental lambda calculus.

Dynamic algorithms are designed to handle arbitrary changes to inputs by maintaining data structures that can be efficiently updated [19]. They achieve efficiency through carefully

designed update operations that maintain complex invariants specific to each problem domain, as demonstrated in problems like dynamic planar convex hulls [8, 20] and dynamic minimum spanning trees [14, 16]. While these algorithms can offer optimal incremental performance for specific problems, their development requires significant effort and domain expertise [22].

Our approach differs by embedding incrementality into the language semantics rather than relying on specialized data structures. Where dynamic algorithms require manual development of update operations for each problem, our differential semantics provides a systematic approach to deriving incremental behavior for programs written in our imperative language. This language-based approach demonstrates how incremental computation can be treated as a fundamental programming language feature rather than a specialized algorithmic concern.

Selective recomputing techniques aim to reuse previous results by identifying and rerunning only the affected parts of a computation when inputs change [10, 22]. Early methods like function memoization [21, 1] cache function calls and reuse results, but they struggle with programs that have internal state or complex control flow.

Self-adjusting computation [5, 4], representing the state-of-the-art in selective recomputing, introduces sophisticated dependency tracking using dynamic dependency graphs to identify precisely which computations need to be redone. Their change propagation algorithm then selectively recomputes only the affected portions of these dependency graphs. This approach has been enhanced to improve efficiency through data structure-level dependency tracking [3] and extended to handle imperative features [2], but maintains the core mechanism of dependency tracking. Despite these advances, the fundamental dependency tracking approach and its inherent disadvantages remain unchanged. *Adapton* [13] extends this line of work with demand-driven evaluation and better support for cyclic dependencies, but still faces similar overhead challenges from maintaining complex runtime structures.

Where selective recomputation focuses on identifying which computations to redo through dependency information, our differential semantics takes a fundamentally different approach by directly modeling how changes propagate through program constructs. This direct modeling enables more efficient handling of control flow changes: instead of rebuilding and traversing complex dependency graphs when loop bounds or conditions change, our semantics directly computes the precise differential effect of the modified control flow. The key distinction is that while self-adjusting computation and its extensions determine *what* to recompute, our differential semantics specifies *how* changes transform program states through the language semantics itself.

The incremental lambda calculus (ILC) [9, 11] provides a theoretical foundation for incrementalizing purely functional programs through type-directed program transformation. For each function f, it derives a derivative function f' that computes output changes from input changes, executing these derivative programs using standard semantics.

Our approach differs fundamentally from ILC by providing a differential semantics that directly executes programs on changes without requiring program transformation. This distinction has important practical implications - optimizing our differential semantics requires only modifying semantic rules, while ILC must optimize its program transformation process to generate more efficient derivative code. Another obvious difference between our approaches is that while ILC provides powerful techniques for incrementalizing higher-order functional programs, our differential semantics offers direct support for imperative features.

Differential updating originated in databases to maintain materialized views efficiently [7]. For each relational operator, they derive rules that translate changes to input relations into changes to output relations, avoiding full recomputation. View maintenance systems formalize

these as delta rules [12] with algebraic properties that ensure correctness: a delta rule applied to input changes must produce the same result as recomputing with updated inputs. This approach has been particularly successful for Datalog, where differential evaluation enables efficient incremental maintenance of recursive views [18].

Our work shares the fundamental insight of deriving change-propagation rules but applies it to programming language constructs rather than relational operators. Like database approaches, we ensure correctness through algebraic properties, but our properties focus on language-level operations rather than relational algebra. This enables incrementalization of general-purpose programs while maintaining the mathematical rigor of differential database techniques.

Differential dataflow [17] extends dataflow systems to support incremental updates through data provenance tracking. While effective for data-parallel computations, it requires programs to be expressed in a restricted dataflow model. Our approach provides similar benefits for general imperative programs through language-level change tracking, making incremental computation accessible to mainstream programming.

9 Conclusion

This paper presents differential semantics, which exploits precise change information rather than using all-or-nothing selective recomputing. Rather than treating incrementality as a separate concern, we embed it directly in the programming language through semantic rules that precisely track and propagate changes. Our key contributions include a type-safe theory of changes that ensures semantic consistency, a differential semantics for an imperative language that handles both data and control flow changes, and a set of verified optimizations that make this approach practical. Through our implementation as a differential interpreter, validated on the Bellman-Ford algorithm, we demonstrate that our approach enables order-of-magnitude speedups compared to recomputation. These results establish that principled language design can make incremental computation both automatic and efficient. Future work could extend these foundations to richer language features like functions and recursion, moving toward general-purpose languages where incrementality is the default rather than a specialized capability.

References

1 Martín Abadi, Butler Lampson, and Jean-Jacques Lévy. Analysis and caching of dependencies. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 83–91, 1996.

2 U. A. Acar, A. Ahmed, and M. Blume. Imperative self-adjusting computation. *SIGPLAN Not.*, 43(1):309–322, 2008.

3 U. A. Acar, G. Blelloch, R. Ley-Wild, K. Tangwongsan, and D. Turkoglu. Traceable data types for self-adjusting computation. *SIGPLAN Not.*, 45(6):483–496, 2010.

4 Umut A. Acar. *Self-Adjusting Computation*. PhD thesis, Carnegie Mellon University, 2005.

5 Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 247–259, 2002.

6 Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

7 John A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. Efficiently updating materialized views. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 61–71, 1986.

8 Gerth Stølting Brodal and Riko Jacob. Dynamic planar convex hull. In *Proceedings of the 43rd Annual Symposium on Foundations of Computer Science*, pages 617–626, 2002.

9 Yan Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. A theory of changes for higher-order languages: Incrementalizing lambda-calculi by static differentiation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 145–155, 2014.

10 Allan Demers, Thomas Reps, and Tim Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 105–116, 1981.

11 Paolo G. Giarrusso, Yann Régis-Gianas, and Philipp Schuster. Incremental lambda-calculus in cache-transfer style: Static memoization by program transformation. In *Proceedings of the 28th European Symposium on Programming*, pages 553–580, 2019.

12 Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 157–166, 1993.

13 Michael A. Hammer, Kyle Y. J. Phang, Michael Hicks, and Jeffrey S. Foster. Adapton: Composable, demand-driven incremental computation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–166, 2014.

14 Monika Rauch Henzinger and Valerie King. Algorithmic aspects of dynamic graph connectivity. *Algorithmica*, 22(1–2):119–146, 1998.

15 Herbert W. Hethcote. The mathematics of infectious diseases. *SIAM Review*, 42(4):599–653, 2000.

16 Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48:723–760, 2001. URL: `https://api.semanticscholar.org/CorpusID:7273552`.

17 Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research*, pages 1–11, 2013.

18 Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. Incremental update of datalog materialisation: The backward/forward algorithm. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 1560–1568, 2015.

19 Mark H. Overmars. *The Design of Dynamic Data Structures*. Springer-Verlag, 1983.

20 Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23(2):166–204, 1981.

21 William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 315–328, 1989.

22 Ganesan Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 502–510, 1993.