Differential Execution with Lexical Tracing

SEBASTIAN ERDWEG, KIT, Germany
RUNQING XU, KIT, Germany
MO BITAR, KIT, Germany

Incremental computing promises large speed-ups after small input edits. Yet, most incrementality approaches merely skip unchanged work and recompute the remaining sub-computations, even when the inputs change only slightly. Differential execution avoids this by propagating data changes (i.e., deltas), and prior work has shown how to develop a provably correct differential big-step semantics. Unfortunately, that semantics must still replay the original computation at every step, squandering much of the potential gain of incrementalization. While the semantics clearly needs caching to avoid recomputations, a sound and efficient caching discipline is challenging. First, each execution step must be uniquely identified; second, the identifier must remain stable even when the preceding control flow changes. To this end, we develop *lexical tracing*, which identifies execution steps through their path in the derivation tree of the big-step semantics. We then extend differential execution with lexical tracing and caching to deliver, for the first time, a formally verified, asymptotically efficient account of differential execution for imperative languages. In particular, we developed a novel mechanized theory of cache stability for lexical traces and their semantic rules, which was essential in proving the differential caching semantics correct and complete in Rocq.

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

Additional Key Words and Phrases: incremental computing, operational semantics, mechanized metatheory

ACM Reference Format:
Sebastian Erdweg, Runqing Xu, and Mo Bitar. 2026. Differential Execution with Lexical Tracing. 1, 1 (February 2026), 26 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Incremental computations save execution time and energy by reacting to input changes. Most approaches to incremental computing rely on *selective recomputation* [Ramalingam and Reps 1993]: They rerun subcomputations that transitively depend on the changed input, but skip subcomputations whose inputs are unchanged. For example, build systems apply selective recomputation to react to file changes [Konat et al. 2018], and self-adjusting computations popularized selective recomputation in the PL community [Acar et al. 2008; Hammer et al. 2014]. Unfortunately, selective recomputation has one severe problem: Whether an input change is large or small, the computation has to be rerun and cannot exploit algebraic properties of the changed data. For example, if we compute `max [1,2,3,4,5]` and reverse the input list `max [5,4,3,2,1]`, selective recomputation has to do a full rerun, wasting time and energy.

An alternative approach to incremental computing is *differential execution* [Kumar et al. 2025], where changes are processed using a dedicated *differential semantics*. Consider an imperative language with a big-step base semantics $s, st \Rightarrow st'$. The corresponding differential semantics

Authors' Contact Information: Sebastian Erdweg, KIT, Germany; Runqing Xu, KIT, Germany; Mo Bitar, KIT, Germany.

ACM XXXX-XXXX/2026/2-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

$s, \Delta st \Rightarrow_\Delta \Delta st'$ executes s relative to a store of value changes and tries to invest minimal effort while being correct: The resulting differential store $\Delta st'$ must be such that $s, st \oplus \Delta st \Rightarrow st' \oplus \Delta st'$, which is know as from-scratch consistency. Differential execution provides efficient incrementality because it propagates value changes to primitive operators, which can exploit algebraic properties of the underlying data types. For example, consider a simple program with free variable `xs`, and an initial and incremental run:

```
# program uses free variable xs                  # initial run          # incremental run
# which must be bound in initial store           xs = [1,2,3,4,5]       Δxs = reverse
m := max xs                                      m = 5                  Δm = noc
x := head xs                                     x = 1                  Δx = +4
if x == m:                                       → false                → true
    ys := tail xs
else:
    ys := xs                                     ys = [1,2,3,4,5]       Δys = drop 1 ∘ reverse
```

In the initial run of the program, we set `xs = [1,2,3,4,5]` and then compute its maximum and head element before selecting the else branch. More interesting is the incremental run using differential execution, where we set `Δxs` to be the reverse of `xs`. This information is stored in the differential store Δst and retrieved whenever we access variable `xs`. In particular, `max xs` yields `noc` (no change) when `xs` is reversed, because the order of list elements does not affect the maximum. The primitive differential maximum operator can exploit this property because it knows *how* its input changed. The `head` operator needs to compare the old head element to the new head element, and finds that it increased by 4. And if the equation comparator remembers the difference between the old `x` and `m` (which was `-4`), it can quickly determine that the new head element is equal to the unchanged maximum. Hence, the differential run will choose the then branch, whose result differs from the original `ys` by first reversing it and then dropping one element. We can then compositionally feed this change of `ys` to subsequent computations to continue differential execution.

Differential execution can yield asymptotic speedups compared to a from-scratch computation. However, we skipped over a crucial detail: caching. Caching is needed in differential execution for two reasons. First, the primitive operators often need to remember information about past executions. We saw this in the `head` and equality operators, which needed to know the past head element and the difference of the previously compared values. And even though `max` could react to `reverse` without prior information, inserting a list element requires `max` to compare the new element to the previous maximum. Xu and Erdweg [2026] provide a detailed account of stateful differential operators, yet they do not embed them in a language and do not explain how the cache can be managed during execution. Second, control structures need to remember information about the past execution state. In our example, the conditional performed a branch switch in the incremental run: We switched to the then branch, whose body we ran in the state that was valid before the original conditional. Similarly, the differential execution of loops requires cached information. Indeed, differential execution without caching is pointless.

While caching is crucial for the correctness and efficiency of differential execution, there is no formalization of it yet. In this paper, we formalize the caching behavior of differential execution by solving two key technical challenges. The first challenge is how to identify cache entries in a way that every operator and control structure obtains their distinct entry, which is critical for correctness. The second challenge is how to ensure cache entries remain as stable as possible when processing changes, which is critical for efficiency. To this end, we develop the theory of lexical traces, which uniquely identify points in the program execution in a stable fashion, as we explain in the next section. We use lexical traces to develop a tracing semantics for the original and differential execution, where a cache is propagated in store-passing style. We then provide a new cache-aware

correctness criterion for differential execution:

$$s, st, ch_0 \mathbin{@} tr \Rightarrow st', ch_1 \rightarrow$$
$$s, \Delta st, ch_1 \mathbin{@} tr \Rightarrow_\Delta \Delta st', ch_2 \rightarrow$$
$$s, st \oplus \Delta st, ch_0 \mathbin{@} tr \Rightarrow st' \oplus \Delta st', ch_2$$

Essentially, this is from scratch consistency: If we rerun s with the patched input store $st \oplus \Delta st$, we get the same result as if we had patched the output store st' using the differential result $\Delta st'$. However, we include lexical traces tr and caches ch to model efficient differential execution. To prove correctness, then, we will also need to reason about cache equivalence and cache stability extensively. We formalized lexical tracing and differential caching execution for a small imperative language in Rocq and provide a mechanized proof of its correctness and completeness.

In summary, we make the following contributions:

- We present lexical tracing, an approach to uniquely identify points in the program execution that are stable under subcomputation changes.
- We develop caching semantics for the original and differential execution using lexical traces.
- We develop a theory of local cache equality and cache stability in Rocq, which is elemental for proving the correctness and completeness of the caching differential semantics.
- We implement a differential interpreter based on our principled caching discipline and demonstrate that its performance is on par with a prior ad-hoc implementation.

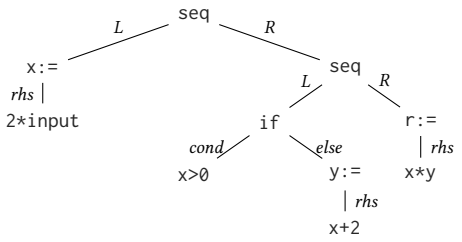
2 Incremental Caching and Lexical Tracing

Differential execution must cache information from the previous run to react to input changes correctly and efficiently. For example, the differential multiplication operator is an instructive example, even though it may be faster to simply recompute multiplications. When we change the operands of a multiplication $x{*}y$ with $\Delta x = +a$ and $\Delta y = +b$, the output change is $+(a{*}y{+}a{*}b{+}x{*}b)$, which we can derive if we remember x and y. However, different instances of the same operator must each remember their own operands. For example, the cache may not confuse the multiplications in `s1` and `s5`:

```
# program with              # initial run / cache      # incremental run / cache
# free variable input       input = -2                Δinput = +3
s1:  x := 2 * input         x = -4 / (2, -2)          Δx = +2*3 = +6 / (2, 1)
s2:  if x > 0:              → false                   → true
s3:     y := x + 1                                    Δy = +5
     else:
s4:     y := x + 2          y = -2
s5:  r := x * y             r = 8 / (-4, -2)          Δr = +(6*-2 + 6*5 + 5*-4) = -2 / (2, 3)
```

Say, we run this program with `input = -2` originally. The multiplication in s_1 must remember its operands `2` and `-2`, whereas the multiplication in s_5 must remember `-4` and `-2`. Only then will both multiplications be able to react to changes to its respective operands. When we increase `input` by `+3`, the operands in s_1 change by `+0` and `+3`. Since the old left-operand was `2`, the multiplication result increases by `+2*3=+6` from `-4` to `2`. Subsequently, this leads to changes of the operands in s_5 by `+6` and `+5`. Since the cached operands are `-4` and `-2`, the result changes by `-2` following the formula from above. Note that we also updated cache entries in the incremental run, so that we can process subsequent changes if needed.

We need to uniquely identify operators during the execution of the program. Unfortunately, the position of the operator is insufficient, because operators can occur in loops and in recursive functions, yet each application of an operator needs a distinct cache entry. One way to identify operator applications is by using the program trace that led to the operator's execution. For example,



Examples of lexical traces and the nodes they resolve to:

$\mathsf{seq_L} \cdot \mathsf{assign_{rhs}} \quad \mapsto \quad$ `2 * input`
$\mathsf{seq_R} \cdot \mathsf{seq_L} \cdot \mathsf{if_{cond}} \quad \mapsto \quad$ `x > 0`
$\mathsf{seq_R} \cdot \mathsf{seq_L} \cdot \mathsf{if_{else}} \quad \mapsto \quad$ `y := x+2`
$\mathsf{seq_R} \cdot \mathsf{seq_R} \cdot \mathsf{assign_{rhs}} \quad \mapsto \quad$ `x * y`

Fig. 1. An execution tree gives rise to lexical traces for each individual node.

in the initial run, the multiplication in s_5 is reached through trace $s_1 \cdot s_2 \cdot s_4 \cdot s_5$. Indeed, this trace clearly distinguishes the operators in s_1 and s_5. However, the trace lacks stability and leads to a huge number of cache misses. For example, in the incremental run, we reach s_5 through $s_1 \cdot s_2 \cdot s_3 \cdot s_5$. Since the two traces are different, we would not find the cache entry and would need to perform a recomputation for the second execution of s_5. Indeed, any change to the control flow preceding a statement will affect the statement's trace. Clearly, this fails our objectives for incremental computing. We propose lexical tracing to regain trace stability.

2.1 Execution Trees and Lexical Tracing

Traditional tracing records the sequence of executed statements. The key idea of lexical tracing is to instead record the path in the big-step derivation tree that leads from the root to a given statement. For our previous example, the following derivation tree models the initial run in a big-step semantics:

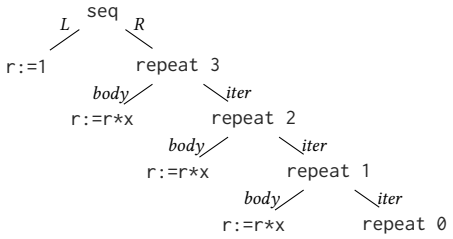
$$\dfrac{\dfrac{\texttt{2*input}, st_0 \Longrightarrow \texttt{-4}}{\texttt{x:=2*input}, st_0 \Longrightarrow st_1} \qquad \dfrac{\texttt{x>0}, st_1 \Longrightarrow \texttt{false} \qquad \dfrac{\texttt{x+2}, st_1 \Longrightarrow \texttt{-2}}{\texttt{y:=x+2}, st_1 \Longrightarrow st_2} \qquad \dfrac{\texttt{x*y}, st_2 \Longrightarrow \texttt{8}}{\texttt{r:=x*y}, st_2 \Longrightarrow st_3}}{\texttt{if (x>0) (y:=x+1) (y:=x+2)}, st_1 \Longrightarrow st_2}}{\texttt{if (x>0) (y:=x+1) (y:=x+2); r:=x*y}, st_1 \Longrightarrow st_3}}{\texttt{x:=2*input; if (x>0) (y:=x+1) (y:=x+2); r:=x*y}, st_0 \Longrightarrow st_3}$$

where $st_1 = st_0; \texttt{x=-4} \qquad st_2 = st_1; \texttt{y = -2} \qquad st_3 = st_2; \texttt{r = 8}$

While we can extract lexical traces from the derivation, it contains a lot of information irrelevant to tracing. Therefore, it is instructive to simplify the derivation tree into what we call an execution tree: A tree that records the execution's structure but ignores the results (here: the values and stores). We represent execution trees in a style similar to abstract syntax trees, with the starting point at the top. However, the execution tree is a projection of the derivation tree and thus only contains the code that was reached during execution.

For our example, we show the execution tree of the initial run in Figure 1. Note that the execution tree only includes the else branch but omits the unreached then branch. The execution tree makes it easy to uniquely identify executed statements and expressions through their path from the root node. For our example, we show a few paths in Figure 1 with the nodes they resolve to. These are the *lexical traces* of the program.

Lexical traces are partial: They do not record the full history of execution, but only identify the current statement or expression. This is exactly what we need when using lexical traces for caching. Specifically, the key advantage of lexical traces is that they are stable with respect to the execution of non-overlapping subcomputations. For example, the trace $\mathsf{seq_R} \cdot \mathsf{seq_R}$ for the assignment of `r` is independent of the execution of the preceding statements at $\mathsf{seq_L}$ and $\mathsf{seq_R} \cdot \mathsf{seq_L}$. Even in the



Examples of lexical traces and the nodes they resolve to:

$\mathsf{seq_R} \cdot \mathsf{rep_{body}} \quad \mapsto \quad$ `r := r * x`
$\mathsf{seq_R} \cdot \mathsf{rep_{iter}} \cdot \mathsf{rep_{body}} \quad \mapsto \quad$ `r := r * x`
$\mathsf{seq_R} \cdot \mathsf{rep_{iter}}^{k} \quad \mapsto \quad$ `repeat (3 − k)`

Fig. 2. The execution tree for a loop witnesses the loop unfolding as done by the big-step semantics.

incremental run, when the execution of the if statement at $\mathsf{seq_R} \cdot \mathsf{seq_L}$ changes from the else to the then branch, the trace $\mathsf{seq_R} \cdot \mathsf{seq_R}$ still identifies the assignment of `r`. This stability of lexical traces is elemental for using them to identify nodes in a cache.

Before moving on, we should take a brief moment to consider lexical traces for programs that have loops. When a loop body executes multiple times, we need to distinguish operator applications in each iteration. Lexical traces provide just that: When we record the path in the big-step derivation or execution tree, we obtain distinct lexical traces for each loop iteration. For example, we can compute the third power of `x` using `r := 1; repeat 3 (r := r * x)`, which produces the execution tree shown in Figure 2. The execution tree shows the unfolded loop, providing distinct lexical traces to reach each execution of the loop body. This way, we can cache separate entries for all three applications of the multiplication operator in `r := r * x`. Here and in the remainder of this paper we write step^k for the k-fold repetition of step.

2.2 A Semantics with Lexical Tracing

Before turning to differential execution and its caching, we want to show how lexical tracing can be incorporated into a language's semantics. To this end, we provide a big-step semantics for a simple imperative language with lexical tracing. We will later add caching to the semantics, where we use the lexical traces to identify cache entries.

We show the syntax and semantics of the simple imperative language IMP in Figure 3. IMP features mutable variables, conditional statements, and repeat loops (rep e s). Note that we employ an auxiliary syntactic form $\mathsf{rep}^n\ s$ for loops with n remaining iterations (cf. Section 3). We use this information later as part of our caching strategy for differential execution. A lexical trace tr is a sequence of trace links l, describing paths in the implicit execution tree. The semantics of expressions $e, st \mathbin{@} tr \Rightarrow v$ and statements $s, st \mathbin{@} tr \Rightarrow st'$ expects the current trace tr as an additional input. Other than that, the semantics is completely standard. The only novelty is that the semantics extends the trace as it recurses. Thus, the current trace tr always explains how execution proceeded from the root of the program to the current expression or statement.

Let us go through a few exemplary rules. For binary operators $e_1\ \mathsf{op}\ e_2$, we evaluate e_1 under $tr \cdot \mathsf{op}_L$ and e_2 under $tr \cdot \mathsf{op}_R$. The same happens for sequential statements $s_1\,;s_2$. For conditionals if $e\ s_1\ s_2$, we evaluate e under $tr \cdot \mathsf{if_{cond}}$ and then decide which branch to execute. The branch selection is evident in the sub-trace used recursively. For loops rep e s, we evaluate e to determine how many iterations n are necessary. Then we execute $\mathsf{rep}^n\ s$, which repeats the body of the loop n times using $\mathsf{rep_{iter}}$ to distinguish loop iterations in the trace.

We will later reason about traces to prove that independent subcomputations leave their respective cache entries intact. A key property that we will use is trace independence.

Definition 2.1 (Trace independence $tr_1 \# tr_2$). Lexical traces tr_1 and tr_2 are independent $tr_1 \# tr_2$ if neither trace is a prefix of the other: $tr_1 \cdot tr \neq tr_2$ and $tr_1 \neq tr_2 \cdot tr$ for all tr.

$$
\begin{array}{lll}
\text{(values)} & v & ::= \mathsf{true} \mid \mathsf{false} \mid n \\
\text{(expressions)} & e & ::= v \mid x \mid e + e \mid e * e \mid e > e \\
\text{(statements)} & s & ::= \mathsf{skip} \mid x := e \mid s\,; s \mid \mathsf{if}\ e\ s\ s \mid \mathsf{rep}\ e\ s \mid \mathsf{rep}^n\ s \\
\text{(stores)} & st & \in x \to v \\
\text{(trace links)} & l & ::= +_{\mathsf{L}} \mid +_{\mathsf{R}} \mid *_{\mathsf{L}} \mid *_{\mathsf{R}} \mid >_{\mathsf{L}} \mid >_{\mathsf{R}} \\
 & & \quad \mid \mathsf{assign}_{\mathsf{rhs}} \mid \mathsf{seq}_{\mathsf{L}} \mid \mathsf{seq}_{\mathsf{R}} \mid \mathsf{if}_{\mathsf{cond}} \mid \mathsf{if}_{\mathsf{then}} \mid \mathsf{if}_{\mathsf{else}} \\
 & & \quad \mid \mathsf{rep}_{\mathsf{count}} \mid \mathsf{rep}_{\mathsf{aux}} \mid \mathsf{rep}_{\mathsf{body}} \mid \mathsf{rep}_{\mathsf{iter}} \\
\text{(lexical traces)} & tr & ::= l \cdots l
\end{array}
$$

$$
\frac{}{x, st\ @\ tr \Rightarrow st\ x} \qquad \frac{e_1, st\ @\ tr \cdot \mathsf{op}_{\mathsf{L}} \Rightarrow v_1 \qquad e_2, st\ @\ tr \cdot \mathsf{op}_{\mathsf{R}} \Rightarrow v_2}{e_1\ \mathsf{op}\ e_2, st\ @\ tr \Rightarrow v_1\ \mathsf{op}\ v_2}
$$

$$
\frac{}{\mathsf{skip}, st\ @\ tr \Rightarrow st} \qquad \frac{e, st\ @\ tr \cdot \mathsf{assign}_{\mathsf{rhs}} \Rightarrow v}{x := e, st\ @\ tr \Rightarrow st[x \mapsto v]} \qquad \frac{\begin{array}{l} s_1, st\ @\ tr \cdot \mathsf{seq}_{\mathsf{L}} \Rightarrow st_1 \\ s_2, st_1\ @\ tr \cdot \mathsf{seq}_{\mathsf{R}} \Rightarrow st_2 \end{array}}{s_1\,; s_2, st\ @\ tr \Rightarrow st_2}
$$

$$
\frac{\begin{array}{l} e, st\ @\ tr \cdot \mathsf{if}_{\mathsf{cond}} \Rightarrow \mathsf{true} \\ s_1, st\ @\ tr \cdot \mathsf{if}_{\mathsf{then}} \Rightarrow st_1 \end{array}}{\mathsf{if}\ e\ s_1\ s_2, st\ @\ tr \Rightarrow st_1} \qquad \frac{\begin{array}{l} e, st\ @\ tr \cdot \mathsf{if}_{\mathsf{cond}} \Rightarrow \mathsf{false} \\ s_2, st\ @\ tr \cdot \mathsf{if}_{\mathsf{else}} \Rightarrow st_2 \end{array}}{\mathsf{if}\ e\ s_1\ s_2, st\ @\ tr \Rightarrow st_2}
$$

$$
\frac{\begin{array}{l} e, st\ @\ tr \cdot \mathsf{rep}_{\mathsf{count}} \Rightarrow n \\ \mathsf{rep}^n\ s, st\ @\ tr \cdot \mathsf{rep}_{\mathsf{aux}} \Rightarrow st' \end{array}}{\mathsf{rep}\ e\ s, st\ @\ tr \Rightarrow st'} \qquad \frac{}{\mathsf{rep}^0\ s, st\ @\ tr \Rightarrow st} \qquad \frac{\begin{array}{l} s, st\ @\ tr \cdot \mathsf{rep}_{\mathsf{body}} \Rightarrow st' \\ \mathsf{rep}^n\ s, st'\ @\ tr \cdot \mathsf{rep}_{\mathsf{iter}} \Rightarrow st'' \end{array}}{\mathsf{rep}^{n+1}\ s, st\ @\ tr \Rightarrow st''}
$$

Fig. 3. Big-step semantics for IMP expressions and statements with lexical tracing.

For example, we have $\mathsf{seq}_{\mathsf{R}} \cdot \mathsf{seq}_{\mathsf{L}} \,\#\, \mathsf{seq}_{\mathsf{R}} \cdot \mathsf{seq}_{\mathsf{R}}$ in our running example from Figure 1. We will later see that this ensure that the (differential) execution of the if statement at $\mathsf{seq}_{\mathsf{R}} \cdot \mathsf{seq}_{\mathsf{L}}$ cannot affect the cached entries for the assignment at $\mathsf{seq}_{\mathsf{R}} \cdot \mathsf{seq}_{\mathsf{R}}$. Similarly, for the repeat example from Figure 2, we have $\mathsf{seq}_{\mathsf{R}} \cdot \mathsf{rep}_{\mathsf{iter}}{}^k \cdot \mathsf{rep}_{\mathsf{body}} \,\#\, \mathsf{seq}_{\mathsf{R}} \cdot \mathsf{rep}_{\mathsf{iter}}{}^m \cdot \mathsf{rep}_{\mathsf{body}}$ for any $k \neq m$. This ensures that each loop iteration can use the cache safely. We are now ready to formalize the differential semantics of IMP with caching.

3 Differential Semantics with Caching

Differential semantics was recently proposed as a new approach for incremental computing [Kumar et al. 2025]. A differential semantics executes a program and processes input changes to produce output changes. For example, the differential semantics for IMP expressions should take the form $e, \Delta st \Rightarrow_\Delta \Delta v$, where $\Delta st : var \to \Delta v$ maps variables to their value change. However, we cannot possibly implement a differential semantics of this form, because we need information about the original run. To illustrate, consider again the multiplication expression and its standard semantics:

$$
\frac{e_1, st \Rightarrow m \qquad e_2, st \Rightarrow n}{e_1 * e_2, st \Rightarrow m * n}
$$

If we attempt a differential semantics for multiplication without caching, we necessarily fail:

$$
\frac{e_1, \Delta st \Rightarrow_\Delta +a \qquad e_2, \Delta st \Rightarrow_\Delta +b}{e_1 * e_2, \Delta st \Rightarrow_\Delta +(a * n + a * b + m * b)}
$$

$$\frac{}{n, \Delta st, ch \mathbin{@} tr \Rightarrow_\Delta +0, ch}\ (\Delta\textsc{Num}) \qquad \frac{b \in \{\mathsf{true}, \mathsf{false}\}}{b, \Delta st, ch \mathbin{@} tr \Rightarrow_\Delta \mathsf{noc}, ch}\ (\Delta\textsc{Bool})$$

$$\frac{}{x, \Delta st, ch \mathbin{@} tr \Rightarrow_\Delta \Delta st\, x, ch}\ (\Delta\textsc{Var}) \qquad \frac{\begin{array}{l} e_1, \Delta st, ch_0 \mathbin{@} tr \cdot +_\mathsf{L} \Rightarrow_\Delta +a, ch_1 \\ e_2, \Delta st, ch_1 \mathbin{@} tr \cdot +_\mathsf{R} \Rightarrow_\Delta +b, ch_2 \end{array}}{e_1 + e_2, \Delta st, ch_0 \mathbin{@} tr \Rightarrow_\Delta +(a + b), ch_2}\ (\Delta\textsc{Add})$$

$$\frac{\begin{array}{ll} e_1, \Delta st, ch_0 \mathbin{@} tr \cdot *_\mathsf{L} \Rightarrow_\Delta +a, ch_1 & ch_0\ tr = \mathsf{mul}\, m\, n \\ e_2, \Delta st, ch_1 \mathbin{@} tr \cdot *_\mathsf{R} \Rightarrow_\Delta +b, ch_2 & ch_3 = ch_2[tr \mapsto \mathsf{mul}\, (m + a)\, (n + b)] \end{array}}{e_1 * e_2, \Delta st, ch_0 \mathbin{@} tr \Rightarrow_\Delta +(a * n + a * b + m * b), ch_3}\ (\Delta\textsc{Mul})$$

$$\frac{\begin{array}{ll} e_1, \Delta st, ch_0 \mathbin{@} tr \cdot >_\mathsf{L} \Rightarrow_\Delta +a, ch_1 & ch_0\ tr = \mathsf{gt}\, d \\ e_2, \Delta st, ch_1 \mathbin{@} tr \cdot >_\mathsf{R} \Rightarrow_\Delta +b, ch_2 & ch_3 = ch_2[tr \mapsto \mathsf{gt}\, (d + a - b)] \end{array}}{e_1 > e_2, \Delta st, ch_0 \mathbin{@} tr \Rightarrow_\Delta (d + a - b > 0) \ominus (d > 0), ch_3}\ (\Delta\textsc{Gt})$$

Fig. 4. The differential caching semantics for IMP expressions with threaded caches ch.

This differentially evaluates e_1 and e_2 to changes $+a$ and $+b$. The problem is that we cannot describe how the multiplication result changes without referring to the previous operands m and n. In the original introduction of differential semantics, Kumar et al. [2025] sidestepped this issue by providing the original store to the differential semantics. Their rule for multiplication reads:

$$\frac{\begin{array}{ll} e_1, st \Rightarrow m & e_2, st \Rightarrow n \\ e_1, \Delta st \Rightarrow_\Delta +a & e_2, \Delta st \Rightarrow_\Delta +b \end{array}}{e_1 * e_2, st, \Delta st \Rightarrow_\Delta +(a * n + a * b + m * b)}$$

This rule is correct and executable, but very inefficient because it reevaluates e_1 and e_2 given the original store st. Indeed, almost all reduction rules of Kumar et al. [2025] have to do some recomputation, because their semantics lacks caching. But caching is difficult to incorporate: The caching semantics is complex, it requires stable cache keys, and reasoning about cache correctness requires a lot of proof engineering. In the remainder of this section, we present our differential caching semantics based on lexical tracing.

3.1 Differential Caching Semantics for IMP Expressions

The differential caching semantics for IMP expressions takes the form

$$e, \Delta st, ch \mathbin{@} tr \Rightarrow_\Delta \Delta v, ch'.$$

That is, it takes an expression, a differential store, a cache ch, and a trace tr, and it produces a change value Δv and a possibly updated cache ch'. Caches $ch : tr \to \eta$ map traces to cache entries η, which is a variant type for all different kinds of cache entries required by IMP. For IMP expressions, we need variants mul and gt to support differential multiplication and greater-than.

We show the differential caching semantics in Figure 4, which uses the same tracing as in Section 2.2. The differential evaluation of numeric and Boolean constants yields no change. For variables, (ΔVar) looks up the change in the differential store $\Delta st \in x \to \Delta v$. Additions evaluate as usual, but (ΔAdd) makes sure to propagate the cache in store-passing style. The remaining two rules are the interesting ones, because we need caching for multiplication and greater-than.

The differential evaluation of multiplications is handled by (ΔMul). As usual, (ΔMul) recursively evaluates the operands e_1 and e_2 to obtain their changes $+a$ and $+b$. But to produce the result

change $+(a * n + a * b + m * b)$, we also need to know the value of the original operands m and n. This is where the cache comes into play: We retrieve m and n from the cache, which must provide a mul entry for the multiplication expression $e_1 * e_2$. We use the lexical trace tr to identify $e_1 * e_2$ and distinguish it from other multiplication expressions in the program. Finally, we must also update the cache to prepare for any subsequent input changes that may occur. To this end, we store the updated operands in ch_3 and yield it as part of the output.

The differential evaluation of grater-than expressions in (ΔGT) has a similar form, but differs in its details. First, for differential grater-than, it suffices to remember the integer distance $d = m - n$ between the original operands, rather than remembering the operands themselves. Second, we update the cache entry to reflect the distance of the changed operands: $(m + a) - (n + b) = (m - n) + (a - b) = d + a - b$. Third, we compute the output change by diffing the Boolean values $new \ominus old$, which yields a Boolean change noc or neg. For example, $5 > 3$ induces $d = 2$, and $d > 0$ is true. If we now get operand changes $a = +1$ and $b = +4$, then $d + a - b = -1$ and the new result is false. Differential evaluation then yields the result change false $\ominus$ true = neg.

The careful reader may have noticed that the differential evaluation is stuck unless the input cache is already initialized. Therefore, we need a specialized initialization semantics, which behaves like the standard big-step semantics, but also initializes the cache for subsequent differential runs. Indeed, this initialization semantics is equivalent to the tracing from Figure 3, but additionally threads a cache ch in store-passing style. Moreover, when the differential semantics requires cache entries, the initializing semantics must provide them. For expressions, the evaluation rules for multiplication and greater-than expressions must initialize the cache:

$$\frac{\begin{array}{l} e_1, st, ch_0 \mathbin{@} tr \cdot *_{\mathsf{L}} \Rightarrow m, ch_1 \\ e_2, st, ch_1 \mathbin{@} tr \cdot *_{\mathsf{R}} \Rightarrow n, ch_2 \\ ch_3 = ch_2[tr \mapsto \mathsf{mul}\ m\ n] \end{array}}{e_1 * e_2, st, ch_0 \mathbin{@} tr \Rightarrow m * n, ch_3}\ (\textsc{mul}) \qquad \frac{\begin{array}{l} e_1, st, ch_0 \mathbin{@} tr \cdot >_{\mathsf{L}} \Rightarrow m, ch_1 \\ e_2, st, ch_1 \mathbin{@} tr \cdot >_{\mathsf{R}} \Rightarrow n, ch_2 \\ ch_3 = ch_2[tr \mapsto \mathrm{gt}\ (m - n)] \end{array}}{e_1 > e_2, st, ch_0 \mathbin{@} tr \Rightarrow (m - n > 0), ch_3}\ (\textsc{gt})$$

We will later prove in Section 4 that differential evaluation of IMP expressions is complete and correct. Completeness ensures that differential evaluation succeeds whenever the initial evaluation succeeds. In particular, the initial evaluation provides cache entries whenever the differential evaluation requires them. Correctness means that differential evaluation yields results that match a from-scratch re-evaluation of the program with updated inputs. In particular, this guarantees that the initial evaluation provides correct cache entries. However, the actual proofs are complex and require theories of cache equivalence and cache stability; we dedicate Section 4 to the metatheory.

3.2 Differential Caching Semantics for Assignments and Conditionals

The differential semantics for statements is more involved, because control structures are inherently difficult to handle incrementally. The fundamental problem is *branch switching*: When an input change triggers the execution of new code that was not originally executed. In particular, when an input change flips the condition of an if statement, we must switch from one branch to the other. Here, differential execution is out of options and the only way forward is a from-scratch execution of the new code. This requires caching: We must remember the original execution state so that we can restore it for the from-scratch execution of the new branch. A similar problem occurs for loops, where we may have to execute more or less iterations, which also requires caching.

We show the differential caching semantics for statements in Figure 5. The differential execution judgment takes the form

$$s, \Delta st, ch \mathbin{@} tr \Rightarrow_{\Delta} \Delta st', ch'.$$

$$\frac{}{\mathsf{skip}, \Delta st, ch \mathbin{@} tr \Rightarrow_\Delta \Delta st, ch}\ (\Delta\textsc{Skip}) \qquad \frac{e, \Delta st, ch \mathbin{@} tr \cdot \mathsf{assign}_{\mathsf{rhs}} \Rightarrow_\Delta \Delta v, ch'}{x := e, \Delta st, ch \mathbin{@} tr \Rightarrow_\Delta \Delta st[x \mapsto \Delta v], ch'}\ (\Delta\textsc{Assign})$$

$$\frac{\begin{array}{l} s_1, \Delta st, ch \mathbin{@} tr \cdot \mathsf{seq}_\mathsf{L} \Rightarrow_\Delta \Delta st_1, ch_1 \\ s_2, \Delta st_1, ch_1 \mathbin{@} tr \cdot \mathsf{seq}_\mathsf{R} \Rightarrow_\Delta \Delta st_2, ch_2 \end{array}}{s_1 ; s_2, \Delta st, ch \mathbin{@} tr \Rightarrow_\Delta \Delta st_2, ch_2}\ (\Delta\textsc{Seq})$$

$$\frac{\begin{array}{l} ch_0\ tr = \mathsf{if}\ \mathsf{true}\ st\ st' \\ e, \Delta st, ch_0 \mathbin{@} tr \cdot \mathsf{if}_\mathsf{cond} \Rightarrow_\Delta \mathsf{noc}, ch_1 \\ s_1, \Delta st, ch_1 \mathbin{@} tr \cdot \mathsf{if}_\mathsf{then} \Rightarrow_\Delta \Delta st', ch_2 \\ ch_3 = ch_2[tr \mapsto \mathsf{if}\ \mathsf{true}\ (st \oplus \Delta st)\ (st' \oplus \Delta st')] \end{array}}{\mathsf{if}\ e\ s_1\ s_2, \Delta st, ch_0 \mathbin{@} tr \Rightarrow_\Delta \Delta st', ch_3}\ (\Delta\textsc{If-tt})$$

$$\frac{\begin{array}{l} ch_0\ tr = \mathsf{if}\ \mathsf{false}\ st\ st' \\ e, \Delta st, ch_0 \mathbin{@} tr \cdot \mathsf{if}_\mathsf{cond} \Rightarrow_\Delta \mathsf{noc}, ch_1 \\ s_2, \Delta st, ch_1 \mathbin{@} tr \cdot \mathsf{if}_\mathsf{else} \Rightarrow_\Delta \Delta st', ch_2 \\ ch_3 = ch_2[tr \mapsto \mathsf{if}\ \mathsf{false}\ (st \oplus \Delta st)\ (st' \oplus \Delta st')] \end{array}}{\mathsf{if}\ e\ s_1\ s_2, \Delta st, ch_0 \mathbin{@} tr \Rightarrow_\Delta \Delta st', ch_3}\ (\Delta\textsc{If-ff})$$

$$\frac{\begin{array}{l} ch_0\ tr = \mathsf{if}\ \mathsf{true}\ st\ st' \\ e, \Delta st, ch_0 \mathbin{@} tr \cdot \mathsf{if}_\mathsf{cond} \Rightarrow_\Delta \mathsf{neg}, ch_1 \\ s_2, st \oplus \Delta st, ch_1 \mathbin{@} tr \cdot \mathsf{if}_\mathsf{else} \Rightarrow st'', ch_2 \\ \Delta st' = st'' \ominus st' \\ ch_3 = ch_2[tr \mapsto \mathsf{if}\ \mathsf{false}\ (st \oplus \Delta st)\ st''] \end{array}}{\mathsf{if}\ e\ s_1\ s_2, \Delta st, ch_0 \mathbin{@} tr \Rightarrow_\Delta \Delta st', ch_3}\ (\Delta\textsc{If-tf})$$

$$\frac{\begin{array}{l} ch_0\ tr = \mathsf{if}\ \mathsf{false}\ st\ st' \\ e, \Delta st, ch_0 \mathbin{@} tr \cdot \mathsf{if}_\mathsf{cond} \Rightarrow_\Delta \mathsf{neg}, ch_1 \\ s_1, st \oplus \Delta st, ch_1 \mathbin{@} tr \cdot \mathsf{if}_\mathsf{then} \Rightarrow st'', ch_2 \\ \Delta st' = st'' \ominus st' \\ ch_3 = ch_2[tr \mapsto \mathsf{if}\ \mathsf{true}\ (st \oplus \Delta st)\ st''] \end{array}}{\mathsf{if}\ e\ s_1\ s_2, \Delta st, ch_0 \mathbin{@} tr \Rightarrow_\Delta \Delta st', ch_3}\ (\Delta\textsc{If-ft})$$

Fig. 5. The differential semantics for IMP statements uses the cache for conditionals. We highlight key case distinctions and mark premises that rely on from-scratch execution.

For each statement and differential store, the semantics computes a differential store that describes how the original output changes. Skip statements have no effect on the output and hence do not change it. For assignments $x := e$, (ΔAssign) uses differential evaluation of expressions to determine how e changes, and then puts this change in the differential store. For example, when x is increased by one, $y := 2 * x$ differentially evaluates independent of the original store:

$$y := 2 * x, (x \mapsto +1), ch \mathbin{@} tr \Rightarrow_\Delta (x \mapsto +1, y \mapsto +2), ch'.$$

That is, when x increases by 1, y increases by 2. When statements occur in sequence, rule (ΔSeq) executes them subsequently and threads the differential store and cache as usual. Indeed, the first three rules do not involve conditional control structures and are simple adaptions from the tracing semantics of IMP from Section 2.2. The next rules are more interesting.

For if statements, we have to distinguish whether a branch switch occurs or not. For example, when x changes from true to false in (if x $(z := y + 1)$ $(z := y * 2)$), we must switch from the then to the else branch. To this end, we must remember the store before the if statement to execute the else branch from scratch. Moreover, to compute how the else-branch result differs from the previous then-branch result, we also need to remember the store after the if statement, so we can compare the two resulting stores. Our differential caching semantics makes this behavior explicit and, of course, also updates the cache to keep it up-to-date.

In fact, we have four rules to deal with if statements, depending on which branch was originally executed and whether a branch switch occurs. The cache entry (if b st st') indicates whether the condition was previously b = true or b = false, and which store we had before (st) and after (st') executing the original if statement. When the differential evaluation of the condition yields $e, \ldots \Rightarrow_\Delta$ noc (no change), we remain at the same branch as before. For b = true, rule (ΔIf-tt)

remains at the then branch and uses differential execution of s_1 to obtain the output change $\Delta st'$. Conversely, for $b = \text{false}$, rule (ΔIF-FF) remains at the else branch and uses differential execution of s_2. But before we can yield the output change, we must restore the consistency of the cache by updating the store before and after the execution of the if statement. We need them when a branch switch occurs.

A branch switch occurs when the differential evaluation of the condition yields $e, \ldots \Rightarrow_\Delta \text{neg}$ (negate). In rule (ΔIF-TF), we switch from $b = \text{true}$ to false, whereas (ΔIF-FT) switches the other way around. As differential execution switches from one branch to the other, we have to execute the new branch from scratch, because differential execution requires a previous initializing execution. Indeed, rule (ΔIF-TF) executes else branch s_2 with original store st from scratch, and (ΔIF-FT) executes s_1 from scratch, as highlighted in Figure 5. But rather than a differential store, this yields an output store st''. Therefore, we diff ($\ominus$) the new store st'' and the original output store st' to obtain a differential store, which describes the observable changes of the if statement. Lastly, we update the cache, making sure to flip the condition b and update the stores.

The differential semantics requires the cache to be initialized for conditionals. Therefore, the initial run of the program must prepare the cache, which is straightforward:

$$\frac{e, st, ch_0 \mathbin{@} tr \cdot \text{if}_{\text{cond}} \Rightarrow \text{true}, ch_1 \quad s_1, st, ch_1 \mathbin{@} tr \cdot \text{if}_{\text{then}} \Rightarrow st_1, ch_2 \quad ch_3 = ch_2[tr \mapsto \text{if true } st\ st_1]}{\text{if } e\ s_1\ s_2, st, ch_0 \mathbin{@} tr \Rightarrow st_1, ch_3} \qquad \frac{e, st, ch_0 \mathbin{@} tr \cdot \text{if}_{\text{cond}} \Rightarrow \text{false}, ch_1 \quad s_2, st, ch_1 \mathbin{@} tr \cdot \text{if}_{\text{else}} \Rightarrow st_2, ch_2 \quad ch_3 = ch_2[tr \mapsto \text{if false } st\ st_2]}{\text{if } e\ s_1\ s_2, st, ch_0 \mathbin{@} tr \Rightarrow st_2, ch_3}$$

The following program demonstrates the differential execution of conditionals:

```
# free variable n
if n > 0
    r := 5 * n
else:
    r := 2 * n
```

$n = 3$

$ch = \{\ \text{if}_{\text{cond}} \mapsto \text{gt } 3,$
$\quad \text{if}_{\text{then}} \cdot \text{assign}_{\text{rhs}} \mapsto \text{mul } 5\ 3$
$\quad \varepsilon \mapsto \text{if true } \{n \mapsto 3\}\ \{n \mapsto 3, r \mapsto 15\}\ \}$

$\Delta n = -5$

$ch' = \{\ \text{if}_{\text{cond}} \mapsto \text{gt } (-2),$
$\quad \text{if}_{\text{then}} \cdot \text{assign}_{\text{rhs}} \mapsto \text{mul } 5\ 3$
$\quad \text{if}_{\text{else}} \cdot \text{assign}_{\text{rhs}} \mapsto \text{mul } 2\ (-2)$
$\quad \varepsilon \mapsto \text{if false } \{n \mapsto -2\}\ \{n \mapsto -2, r \mapsto -4\}$

The initial run uses $n = 3$. We show the cache ch after the initial run, which contains three entries: (i) gt for the if condition, which is true, (ii) mul for the multiplication in the then branch, and (iii) if with the stores before and after the conditional. Now, consider we make a change $\Delta st = \{\Delta n \mapsto -5\}$ and run it through the differential semantics. The differential semantics updates the cached entry for gt and negates the if condition. Hence, we trigger (ΔIF-TF) and re-execute the else branch using the cached store, which introduces a new cache entry for its own multiplication at $\text{if}_{\text{else}} \cdot \text{assign}_{\text{rhs}}$. Finally, we update the cache and diff the resulting store $\{r \mapsto (-4)\}$ with the previous store $\{r \mapsto 15\}$ to deduce $\Delta r = -19$. Note that we currently do not clear cache entries, but could have removed the entry for $\text{if}_{\text{then}} \cdot \text{assign}_{\text{rhs}}$ after the branch switch.

3.3 Differential Caching Semantics for Loops

For loops, there are various possible caching strategies that affect the performance/memory trade-off of incremental computing. For this paper, we chose a memory-economical caching strategy that is efficient when the number of iterations stays the same or grows, but inefficient for loop rollbacks. Specifically, we only cache the store from before and after the loop, but no intermediate stores from individual loop iterations. Therefore, all cache handling occurs for the (rep e s) construct,

$$\frac{}{\mathsf{rep}^0\ s, \Delta st, ch \mathbin{@} tr \Rightarrow_\Delta \Delta st, ch}\ (\Delta\textsc{rep-0}) \qquad \frac{s, \Delta st_0, ch_0 \mathbin{@} tr \cdot \mathsf{rep}_{\mathsf{body}} \Rightarrow_\Delta \Delta st_1, ch_1 \quad \mathsf{rep}^n\ s, \Delta st_1, ch_1 \mathbin{@} tr \cdot \mathsf{rep}_{\mathsf{iter}} \Rightarrow_\Delta \Delta st_2, ch_2}{\mathsf{rep}^{n+1}\ s, \Delta st_0, ch_0 \mathbin{@} tr \Rightarrow_\Delta \Delta st_2, ch_2}\ (\Delta\textsc{rep+1})$$

$$\frac{\begin{array}{l} ch_0\ tr = \mathsf{rep}\ n\ st\ st' \\ e, \Delta st, ch_0 \mathbin{@} tr \cdot \mathsf{rep}_{\mathsf{count}} \Rightarrow_\Delta \pm 0, ch_1 \\ \mathsf{rep}^n\ s, \Delta st, ch_1 \mathbin{@} tr \cdot \mathsf{rep}_{\mathsf{aux}} \Rightarrow_\Delta \Delta st', ch_2 \\ ch_3 = ch_2[tr \mapsto \mathsf{rep}\ n\ (st \oplus \Delta st)\ (st' \oplus \Delta st')] \end{array}}{\mathsf{rep}\ e\ s, \Delta st, ch_0 \mathbin{@} tr \Rightarrow_\Delta \Delta st', ch_3}\ (\Delta\textsc{rep-noc}) \qquad \frac{\begin{array}{l} ch_0\ tr = \mathsf{rep}\ n\ st\ st' \\ e, \Delta st, ch_0 \mathbin{@} tr \cdot \mathsf{rep}_{\mathsf{count}} \Rightarrow_\Delta -m, ch_1 \\ \mathsf{rep}^{n-m}\ s, st \oplus \Delta st, ch_1 \mathbin{@} tr \cdot \mathsf{rep}_{\mathsf{aux}} \Rightarrow st'', ch_2 \\ \Delta st' = st'' \ominus st' \\ ch_3 = ch_2[tr \mapsto \mathsf{rep}\ (n-m)\ (st \oplus \Delta st)\ st''] \end{array}}{\mathsf{rep}\ e\ s, \Delta st, ch_0 \mathbin{@} tr \Rightarrow_\Delta \Delta st', ch_3}\ (\Delta\textsc{rep-sub})$$

$$\frac{\begin{array}{l} ch_0\ tr = \mathsf{rep}\ n\ st\ st' \\ e, \Delta st, ch_0 \mathbin{@} tr \cdot \mathsf{rep}_{\mathsf{count}} \Rightarrow_\Delta +m, ch_1 \\ \mathsf{rep}^n\ s, \Delta st, ch_1 \mathbin{@} tr \cdot \mathsf{rep}_{\mathsf{aux}} \Rightarrow_\Delta \Delta st', ch_2 \\ \mathsf{rep}^m\ s, (st' \oplus \Delta st'), ch_2 \mathbin{@} tr \cdot \mathsf{rep}_{\mathsf{aux}} \cdot {\mathsf{rep}_{\mathsf{iter}}}^n \Rightarrow st'', ch_3 \\ \Delta st' = st'' \ominus st' \\ ch_4 = ch_3[tr \mapsto \mathsf{rep}\ (n+m)\ (st \oplus \Delta st)\ st''] \end{array}}{\mathsf{rep}\ e\ s, \Delta st, ch_0 \mathbin{@} tr \Rightarrow_\Delta \Delta st', ch_4}\ (\Delta\textsc{rep-add})$$

Fig. 6. For loops, the differential semantics inspects how the iteration count changes.

whereas the differential semantics of $(\mathsf{rep}^n\ s)$ does not interact with the cache. Indeed, as shown in Figure 6, rules (ΔREP-0) and (ΔREP+1) bear no surprises.

When the iteration count does not change (ΔREP-NOC), the control flow of the program remains the same. Therefore, it is sufficient to differentially execute $(\mathsf{rep}^n\ s)$ to propagate the changes. Note that the number of iterations n done by the original loop is retrieved from the cache. When instead the number of iterations is reduced (ΔREP-SUB), we execute the loop from scratch. This is the only option, because we cannot execute loop bodies backward and chose to not cache intermediate results. We obtain the resulting store change by diffing the original (and cached) result st' with the new result st''.

Finally, when the iteration count grows (ΔREP-ADD), we compute the output change in two steps. First, we propagate the input change through the original n iterations incrementally. Second, we execute the m new iterations from scratch. Since $(\mathsf{rep}^m\ s)$ is a continuation of $(\mathsf{rep}^n\ s)$, we have to produce a trace that simulates this behavior by n-fold repetition of $\mathsf{rep}_{\mathsf{iter}}$. Only then will the two subcomputations have isolated cache entries, which is required for correctness. When $m \ll n$, (ΔREP-ADD) yields significant speedups compared to a full recomputation [Kumar et al. 2025].

For example, consider the following program that computes the factorial of n:

```
# free variable n
i := 2; r := 1
repeat n - 1:
    r := r * i
    i := i + 1
```

$n = 3$

$$ch = \{\ \mathsf{seq_R} \cdot \mathsf{seq_R} \cdot \mathsf{rep_{aux}} \cdot \mathsf{rep_{body}} \mapsto \mathsf{mul}\ 1\ 2,\ \mathsf{seq_R} \cdot \mathsf{seq_R} \cdot \mathsf{rep_{aux}} \cdot \mathsf{rep_{iter}}^1 \cdot \mathsf{rep_{body}} \mapsto \mathsf{mul}\ 2\ 3,\ \mathsf{seq_R} \cdot \mathsf{seq_R} \mapsto \mathsf{rep}\ 2\ \{n \mapsto 3, i \mapsto 2, r \mapsto 1\}\ \{n \mapsto 3, i \mapsto 4, r \mapsto 6\}\ \}$$

$\Delta n = +2$

$$ch' = \{\ \mathsf{seq_R} \cdot \mathsf{seq_R} \cdot \mathsf{rep_{aux}} \cdot \mathsf{rep_{body}} \mapsto \mathsf{mul}\ 1\ 2,\ \mathsf{seq_R} \cdot \mathsf{seq_R} \cdot \mathsf{rep_{aux}} \cdot \mathsf{rep_{iter}}^1 \cdot \mathsf{rep_{body}} \mapsto \mathsf{mul}\ 2\ 3,\ \mathsf{seq_R} \cdot \mathsf{seq_R} \cdot \mathsf{rep_{aux}} \cdot \mathsf{rep_{iter}}^2 \cdot \mathsf{rep_{body}} \mapsto \mathsf{mul}\ 6\ 4,\ \mathsf{seq_R} \cdot \mathsf{seq_R} \cdot \mathsf{rep_{aux}} \cdot \mathsf{rep_{iter}}^3 \cdot \mathsf{rep_{body}} \mapsto \mathsf{mul}\ 24\ 5,\ \mathsf{seq_R} \cdot \mathsf{seq_R} \mapsto \mathsf{rep}\ 4\ \{n \mapsto 5, i \mapsto 2, r \mapsto 1\}\ \{n \mapsto 5, i \mapsto 6, r \mapsto 120\}\ \}$$

We start with $n = 3$, which yields $r = 6$ after two iterations. After the initial run, the cache contains three entries: a distinct entry for each iteration's multiplication, and a single entry to cache the result of the loop. Now, consider we make a change $\Delta st = \{\Delta n \mapsto +2\}$. Since the loop's counter expression $n - 1$ evaluates to $+2$ under Δst, we trigger (ΔRep-add). This rule first propagates Δst through the original 2 iterations, which yields $\Delta st' = \Delta st$, that is, there is no further change. Indeed, it is safe to skip the original iterations because the changed Δn does not occur in the body of the loop (Kumar et al. [2025] call this the short-circuiting optimization). Next, we run the 2 new iterations in $st' \oplus \Delta st' = \{n \mapsto 5, i \mapsto 4, r \mapsto 6\}$, which yields $st'' = \{n \mapsto 5, i \mapsto 6, r \mapsto 120\}$ and inserts two additional cache entries for multiplications. Lastly, we update the cache entry for the loop and derive the output change $st'' \ominus st' = \{i \mapsto +2, r \mapsto +114\}$.

This concludes our differential caching semantics. We have explained the key ideas of lexical tracing and incremental caching with a focus on illustration and intuition. The next section studies the theory of our semantics and proves it correct.

4 Cache Equivalence, Cache Stability, and Differential Correctness

We want to prove in Rocq that the differential semantics yields valid and correct changes. This ensures we can safely use the differential semantics in place of a recomputation. Given the original execution $s, st, ch_0 \Rightarrow st', ch_1$ and a subsequent differential execution $s, \Delta st, ch_1 \Rightarrow \Delta st', ch_2$, we must prove:

Validity The output change must be valid for the original output $st' \Vdash \Delta st'$, which ensures patching $st' \oplus \Delta st'$ succeeds. For example, $\{l \mapsto [2,4,6]\} \Vdash \{l \mapsto \mathsf{drop}\ 1\}$ holds, whereas $\{l \mapsto [2,4,6]\} \Vdash \{l \mapsto \mathsf{drop}\ 5\}$ does not hold, because the list only contains 3 elements.

Correctness The output change st' must reflect the difference between the original computation and a from-scratch recomputation. That is, we require $s, st \oplus \Delta st, ch_0 \Rightarrow_\Delta st' \oplus \Delta st', ch_2$.

Completeness Lastly, we show that the differential semantics can react to any valid input change and never gets stuck. That is, $s, \Delta st, ch_1 \Rightarrow_\Delta \Delta st', ch_2$ exists whenever $s, st, ch_0 \Rightarrow st', ch_1$. In particular, the cache is always well-shaped to satisfy the cache requirements of differential execution.

Unfortunately, proving these properties is not simple at all. A key challenge is that we need to reason about the original and the differential execution together, and the original output cache becomes the differential input cache. Consider the case for sequences $s_1 ; s_2$ and its rules:

$$\frac{s_1, st, ch_0 \mathbin{@} tr \cdot \mathsf{seq_L} \Rightarrow st_a, ch_a \quad s_2, st_a, ch_a \mathbin{@} tr \cdot \mathsf{seq_R} \Rightarrow st', ch_1}{s_1 ; s_2, st, ch_0 \mathbin{@} tr \Rightarrow st', ch_1}\ (\textsc{seq}) \qquad \frac{s_1, \Delta st, ch_1 \mathbin{@} tr \cdot \mathsf{seq_L} \Rightarrow_\Delta \Delta st_b, ch_b \quad s_2, \Delta st_b, ch_b \mathbin{@} tr \cdot \mathsf{seq_R} \Rightarrow_\Delta \Delta st', ch_2}{s_1 ; s_2, \Delta st, ch_1 \mathbin{@} tr \Rightarrow_\Delta \Delta st', ch_2}\ (\Delta\textsc{seq})$$

The problem is that we cannot use the induction hypothesis for s_1 or s_2. To use an induction hypothesis, the output cache of the initial computation $s_1, st, ch_0 \mathbin{@} tr \cdot \mathsf{seq_L} \Rightarrow st_a, ch_a$ needs to

$$\frac{}{e \succ \varepsilon} \qquad \frac{e_1 \succ tr}{e_1 \,\mathsf{op}\, e_2 \succ \mathsf{op}_\mathsf{L} \cdot tr} \qquad \frac{e_2 \succ tr}{e_1 \,\mathsf{op}\, e_2 \succ \mathsf{op}_\mathsf{R} \cdot tr}$$

$$\frac{}{s \succ \varepsilon} \qquad \frac{e \succ tr}{x := e \succ \mathsf{assign}_\mathsf{rhs} \cdot tr} \qquad \frac{s_1 \succ tr}{s_1\,;s_2 \succ \mathsf{seq}_\mathsf{L} \cdot tr} \qquad \frac{s_2 \succ tr}{s_1\,;s_2 \succ \mathsf{seq}_\mathsf{R} \cdot tr}$$

$$\frac{e \succ tr}{\mathsf{if}\ e\ s_1\ s_2 \succ \mathsf{if}_\mathsf{cond} \cdot tr} \qquad \frac{s_1 \succ tr}{\mathsf{if}\ e\ s_1\ s_2 \succ \mathsf{if}_\mathsf{then} \cdot tr} \qquad \frac{s_2 \succ tr}{\mathsf{if}\ e\ s_1\ s_2 \succ \mathsf{if}_\mathsf{else} \cdot tr}$$

$$\frac{e \succ tr}{\mathsf{rep}\ e\ s \succ \mathsf{rep}_\mathsf{count} \cdot tr} \qquad \frac{\mathsf{rep}^n\ s \succ tr}{\mathsf{rep}\ e\ s \succ \mathsf{rep}_\mathsf{aux} \cdot tr} \qquad \frac{s \succ tr}{\mathsf{rep}^n\ s \succ \mathsf{rep}_\mathsf{body} \cdot tr} \qquad \frac{\mathsf{rep}^n\ s \succ tr}{\mathsf{rep}^n\ s \succ \mathsf{rep}_\mathsf{iter} \cdot tr}$$

Fig. 7. Trace reachability $s \succ tr$ enumerates all possible trace suffixes that can occur during the execution of s.

be the same as the input cache of the differential computation $s_1, \Delta st, ch_1 \mathbin{@} tr \cdot \mathsf{seq}_\mathsf{L} \Rightarrow_\Delta \Delta st_b, ch_b$. However, $ch_a = ch_1$ does not hold in general.

For example, consider $x := 3\,; y := 2 * x$. The original execution of $x := 3$ does not cache anything, hence $ch_a = \{\,\}$. However, the subsequent execution of $y := 2 * x$ caches the multiplication, hence $ch_1 = \{\mathsf{seq}_\mathsf{R} \mapsto \mathsf{mul}\,2\,3\}$. Since the differential computation starts with ch_1, we have $ch_a \neq ch_1$, which shows why we cannot use the induction hypothesis for s_1. But, there is a way forward: When executing s_1, we only access the cache for traces starting with seq_L, whereas when executing s_2, we only access the cache for traces starting with seq_R. Therefore, ch_1 is equal to ch_a for all traces starting with seq_L. Indeed, since $tr \cdot \mathsf{seq}_\mathsf{L} \neq tr \cdot \mathsf{seq}_\mathsf{R}$, the executions of s_1 and s_2 operate on independent parts of the cache. We need to incorporate this observation into our theory.

Note. Except for trace reachability, we only show the metatheory for IMP statements and their execution. All theorems we show have an analogous counterpart for the expressions of IMP. Of course, our Rocq formalization contains proofs for all of the metatheory: for IMP expressions and statements. However, since the metatheory for expressions does not reveal any additional insight, we decided to omit it from the presentation. We also generally assume all programs and stores are well-typed, that is, we only consider executions where $\Gamma \vdash st$ and $\Gamma \vdash s\ \mathsf{ok}$ for some context Γ.

4.1 Cache Equivalence and Trace Reachability

To strengthen the induction hypothesis, we need to generalize the properties we are proving. Specifically, we may not request that the cache produced by the initializing execution is used as input for the differential execution. Rather, we only require these caches to be *observably equivalent*. Two caches are observably equivalent if they provide equal cache entries for all *reachable traces*.

Definition 4.1 (Trace reachability). Given a statement s, only reachable traces $s \succ tr$ can occur during the execution of s. We define the set of reachable traces inductively in Figure 7.

To define cache equivalence, we additionally consider the trace tr_0 that records the execution from the program's main entry point to the current statement s. Two caches are equivalent for executing s if they have identical entries for all traces $(tr_0 \cdot tr)$, where tr is reachable from s:

Definition 4.2 (Cache equivalence). Given a statement s and a trace tr_0, caches ch_1 and ch_2 are observably equivalent $s \mathbin{@} tr_0 \vdash ch_1 \sim ch_2$ if for all $s \succ tr$, we have $ch_1(tr_0 \cdot tr) = ch_2(tr_0 \cdot tr)$.

We formalized trace reachability and cache equivalence in Rocq and proved a few interesting properties. First, we show that we indeed defined an equivalence relation, which is trivial to prove.

Lemma 4.3. For any s and tr, cache equivalence $s \mathbin{@} tr \vdash ch_1 \sim ch_2$ is an equivalence relation.

A more interesting property is that cache updates are sometimes irrelevant. Specifically, a cache update at tr' is irrelevant for any cache equivalence rooted at tr if tr is not a prefix of tr'. This is because cache equivalence at tr only concerns cache entries for reachable continuations of tr, which does not include tr'.

Lemma 4.4. For any tr not prefix of tr', $s \mathbin{@} tr \vdash ch_1 \sim ch_2$ if and only if $s \mathbin{@} tr \vdash ch_1 \sim ch_2[tr' \mapsto e]$.

Proof. Since $tr \cdot tr_0 \neq tr'$ for any tr_0, the entry for tr' is irrelevant for cache equivalence. □

Lastly, it is interesting to note that trace reachability over-approximates the actual traces inhabited by the execution. This is observable for conditionals, where both branches are reachable independent of the condition. For loops, this is even more remarkable, because trace reachability and cache equivalence make no prediction about the number of iterations a loop may take.

Lemma 4.5. For any m and n, if $\mathrm{rep}^m\ s \mathbin{@} tr \vdash ch_1 \sim ch_2$, then $\mathrm{rep}^n\ s \mathbin{@} tr \vdash ch_1 \sim ch_2$.

Proof. By induction on tr, $\mathrm{rep}^m\ s \succ tr$ implies $\mathrm{rep}^n\ s \succ tr$. It follows that the iteration count is irrelevant for cache equivalence, because the set of relevant traces is independent of the count. □

Cache equivalence forms the basis for reasoning about the differential execution inductively. But before we can prove validity, correctness, and completeness of differential execution, we first must observe and verify a number of key properties that connect execution and cache equivalence.

4.2 Cache Stability and Cache Invariance

We introduced lexical traces for two reasons: First, to ensure we can uniquely identify language constructs that need caching during the execution. For IMP, this was multiplication, comparisons, conditionals, and loops. The theorems in Section 4.3 ensure we succeeded.

The second reason we introduced lexical traces was to make cache entries stable during differential execution. For example, in $s_1\,;s_2$, changes to the control flow in s_1 do not proliferate the lexical traces that occur in s_2. Specifically, we argued that independent subcomputations do not affect each others' cache entries. Recall from Definition 2.1 that two traces are independent $tr \# tr'$ if neither is a prefix of the other, which means tr and tr' root independent sub-computations. We thus formulate cache stability for the initializing execution.

Theorem 4.6 (Initial cache stability). Given independent traces $tr \# tr'$, an execution rooted at tr does not affect cache entries rooted at tr': If $s, st, ch \mathbin{@} tr \Rightarrow st', ch'$, then $s' \mathbin{@} tr' \vdash ch \sim ch'$.

Proof. By induction over the execution relation, using Lemmas 4.3 and 4.4. Note that s' is indeed independent of s, because the root traces are sufficient to distinguish the cache accesses. □

The proof of this theorem is quite straightforward actually, although we programmed a number of tactics in Rocq to improve automation. Unfortunately, cache stability leaves one special case: When tr is not a prefix of tr', but tr' may be a prefix of tr. In this case, we cannot rely on the reachable traces used in cache equivalence, because that could include trace continuations shared by tr and tr'. But, we can guarantee cache stability for the specific trace tr', which we need later.

Lemma 4.7. For any tr not prefix of tr', when $s, st, ch \mathbin{@} tr \Rightarrow st', ch'$, then $ch\ tr' = ch'\ tr'$.

Proof. By induction over the execution relation, using Lemma 4.3. □

With this, we can prove an important property about cache equivalence. Namely, equivalent caches lead to equivalent initializing executions, where the resulting stores are identical and the resulting caches are again equivalent. In other words, the initial execution is invariant to equivalent caches.

Theorem 4.8 (Cache invariance). Given caches such that $s \mathbin{@} tr_0 \vdash ch_1 \sim ch_2$. For any st, if $s, st, ch_1 \mathbin{@} tr_0 \Rightarrow st_1, ch_1'$ and $s, st, ch_2 \mathbin{@} tr_0 \Rightarrow st_2, ch_2'$, then $st_1 = st_2$ and $s \mathbin{@} tr_0 \vdash ch_1' \sim ch_2'$.

Proof. By induction over one of the execution relations, using Theorem 4.6 and all of the lemmas from above. □

The proof of cache invariance is rather involved, even though we wrote tactics that generalize repeating proof patterns. But this property is elemental to proving differential correctness when branch switching occurs, where it ensures that the from-scratch execution and the branch-switch execution yield the same stores, even though their caches differ.

4.3 Differential Validity, Correctness, and Completeness

So far, we have reasoned about cache equivalence and have shown the initializing execution is well-behaved with respect to it. It is time to validate that the differential execution is also well-behaved. We start by proving that differential execution adheres to cache stability.

Theorem 4.9 (Differential cache stability). Given independent traces $tr \# tr'$, differential execution rooted at tr does not affect cache entries rooted at tr'. That is, if $s, \Delta st, ch \mathbin{@} tr \Rightarrow_\Delta \Delta st', ch'$, then $s' \mathbin{@} tr' \vdash ch \sim ch'$.

Proof. By induction over the differential execution relation, using Theorem 4.6 and Lemmas 4.3 and 4.4. Again, s' is independent of s, because the root traces are sufficient to distinguish the cache accesses. □

We now have the necessary building blocks to tackle our main theorems. Differential validity ensures that the changes produced by differential execution can be patched. Indeed, patching $st' \oplus \Delta st'$ is only defined when $st' \Vdash \Delta st'$.

Theorem 4.10 (Differential validity). Given a valid input change $st \Vdash \Delta st$ and equivalent caches $s \mathbin{@} tr \vdash ch_1 \sim ch_2$. If the initializing execution $s, st, ch_0 \mathbin{@} tr \Rightarrow st', ch_1$ is followed by a differential execution $s, \Delta st, ch_2 \mathbin{@} tr \Rightarrow_\Delta \Delta st', ch_3$, then $st' \Vdash \Delta st'$.

Proof. By induction over the differential execution relation. Note that $st' \Vdash \Delta st'$ is defined pointwise and holds if for all x, $st'\ x \Vdash \Delta st'\ x$. For assignments, we hence rely on the validity of differential evaluation of the assignment's right-hand side. For branch switching, we exploit that diffing produces valid changes: $st_2 \Vdash st_1 \ominus st_2$. □

Now that we know the differential result is valid, we can investigate the correctness of the resulting change store $\Delta st'$. Specifically, we compare the patched result $st' \oplus \Delta st'$ to a store st'', which has been recomputed from scratch for the patched input.

Theorem 4.11 (Differential correctness). Given a valid input change $st \Vdash \Delta st$ and equivalent caches $s \mathbin{@} tr \vdash ch_1 \sim ch_2$. If the initializing execution $s, st, ch_0 \mathbin{@} tr \Rightarrow st', ch_1$ is followed by a differential execution $s, \Delta st, ch_2 \mathbin{@} tr \Rightarrow_\Delta \Delta st', ch_3$, then $st' \oplus \Delta st'$ yields the same result as a from-scratch execution. That is, for ch_4 equivalent to the initial cache $s \mathbin{@} tr \vdash ch_0 \sim ch_4$, then any from-scratch execution on the changed input $s, st \oplus \Delta st, ch_4 \mathbin{@} tr \Rightarrow st'', ch_5$ yields $st'' = st' \oplus \Delta st'$.

Proof. By induction over the differential execution relation. The most complex case is (ΔRep-add), where we need to reason about $\mathsf{rep}^{n+m}\ s$ and show that we can split this into $\mathsf{rep}^{n}\ s$ followed by $\mathsf{rep}^{m}\ s$. Moreover, we must prove that the starting trace $tr \cdot \mathsf{rep}_{\mathsf{aux}} \cdot {\mathsf{rep}_{\mathsf{iter}}}^{n}$ of $\mathsf{rep}^{m}\ s$ indeed corresponds to the trace used by the from-scratch execution, which allows us to derive the necessary cache equivalences. □

Differential correctness guarantees that differential execution yields the same result as a from-scratch execution. However, correctness does not ensure that differential execution actually finds an output change. For example, a differential execution relation that is always stuck would satisfy the above correctness criterion. We need a completeness theorem to ensure that differential execution must deliver an output change.

Theorem 4.12 (Differential completeness). Given a valid input change $st \Vdash \Delta st$ and equivalent caches $s \mathbin{@} tr \vdash ch_1 \sim ch_2$ as well as $s \mathbin{@} tr \vdash ch_0 \sim ch_4$. If the initializing execution $s, st, ch_0 \mathbin{@} tr \Rightarrow st', ch_1$ and the from-scratch execution $s, st \oplus \Delta st, ch_4 \mathbin{@} tr \Rightarrow st'', ch_5$ succeed, then the differential execution succeeds $s, \Delta st, ch_2 \mathbin{@} tr \Rightarrow_\Delta \Delta st', ch_3$ for some $\Delta st'$ and ch_3.

Proof. By induction on the initializing execution relation. For each initializing execution rule, we need to find differential execution rules that can fire, which ensures the cache requirements of the differential semantics are always satisfied. Note that the premise about from-scratch execution is actually only needed when executions can diverge, which is not the case for IMP. □

With respect to termination, our completeness theorem states that the differential execution must terminate whenever both the original and the patched execution terminate. This is a very strong property. In case either original or patched execution diverge, it is likely that the differential execution will do the same, but this is not covered by our theory. Together, correctness and completeness ensure we can use differential execution as a drop-in replacement for from-scratch executions.

4.4 Summary

We have stated and verified key properties about the metatheory of differential execution. The most important results are differential validity, correctness, and completeness, which together ensure we can use differential execution as a drop-in replacement for a from-scratch execution. To verify these properties, we had to develop a theory of local cache equivalence and prove that execution only affects cache entries local to the executed sub-program. Indeed, execution is invariant under equivalent caches, which is crucial for reasoning about branch switching within differential execution.

We have formalized the complete metatheory in the Rocq proof assistant. The mechanized metatheory comprises about 3600 SLoC and includes 146 lemmas and 26 tactics (small and large). We will submit the Rocq code for artifact evaluation and make it open source.

5 Extending the Expressivity of IMP

Our imperative language IMP, in its current form, cannot express general-purpose algorithms because it lacks syntax constructs for recursion and while loops. We now extend the language's syntax and semantics to incorporate these essential features.

5.1 Differential Execution of Recursive Functions

To support recursive functions, we extend the IMP syntax with constructs for function definitions and calls, as detailed in Figure 8. A function definition takes the form $(\mathsf{def}\ f(x) = s; e)$, which specifies the function's formal parameter x, its body (a statement s), and its return expression e. A function call is a statement of the form $(x := \mathsf{call}\ f\ e)$, which invokes function f with the argument expression e and assigns the result to the variable x. To trace the a function call, we also introduce three corresponding trace links: $\mathsf{call}_{\mathsf{arg}}$ for argument evaluation, $\mathsf{call}_{\mathsf{body}}$ for the execution of the function body, and $\mathsf{call}_{\mathsf{exit}}$ for the evaluation of the return expression.

$$
\begin{array}{lll}
\text{(function definitions)} & d & ::= \mathsf{def}\ f(x) = s; e \\
\text{(statements)} & s & ::= \ldots \mid x := \mathsf{call}\ f\ e \\
\text{(enviroments)} & env & \in f \to d \\
\text{(trace links)} & l & ::= \ldots \mid \mathsf{call}_{\mathsf{arg}} \mid \mathsf{call}_{\mathsf{body}} \mid \mathsf{call}_{\mathsf{exit}}
\end{array}
$$

$$
\frac{\begin{array}{ll} env\ f = \mathsf{def}\ f(y) = s; r & e, st, ch_0 \mathbin{@} tr \cdot \mathsf{call}_{\mathsf{arg}} \Rightarrow v_e, ch_1 \\ s, st_{empty}[y \mapsto v_e], ch_1, env \mathbin{@} tr \cdot \mathsf{call}_{\mathsf{body}} \Rightarrow st_1, ch_2 & r, st_1, ch_2 \mathbin{@} tr \cdot \mathsf{call}_{\mathsf{exit}} \Rightarrow v_r, ch_3 \end{array}}{x := \mathsf{call}\ f\ e, st, ch_0, env \mathbin{@} tr \Rightarrow st[x \mapsto v_r], ch_3} \quad (\textsc{call})
$$

$$
\frac{\begin{array}{l} env\ f = \mathsf{def}\ f(y) = s; r \quad e, \Delta st, ch_0 \mathbin{@} tr \cdot \mathsf{call}_{\mathsf{arg}} \Rightarrow_\Delta \Delta v_e, ch_1 \\ s, \Delta st_{empty}[y \mapsto \Delta v_e], ch_1, env \mathbin{@} tr \cdot \mathsf{call}_{\mathsf{body}} \Rightarrow_\Delta \Delta st_1, ch_2 \\ r, \Delta st_1, ch_2 \mathbin{@} tr \cdot \mathsf{call}_{\mathsf{exit}} \Rightarrow_\Delta \Delta v_r, ch_3 \end{array}}{x := \mathsf{call}\ f\ e, \Delta st, ch_0, env \mathbin{@} tr \Rightarrow_\Delta \Delta st[x \mapsto \Delta v_r], ch_3} \quad (\Delta\textsc{call})
$$

Fig. 8. Big-step semantics and differential semantics for function calls and the new function environment.

With the additions of functions, the structure of the big-step and differential semantics is extended. We introduce a function environment *env* to map function names to definitions, as highlighted in Figure 8. While all semantics rules in Figure 3 must be updated to carry this environment, we omit them for brevity, with the understanding that *env* is simply propagated where it is not actively used. The (CALL) rule implements a call-by-value semantics where functions operate in a local scope. It evaluates the argument e, then executes the function's body s in a fresh store st_{empty} containing only the parameter binding. The function's final return value is then assigned to x in the original store st, ensuring the call has no other side effects on the caller's variables. Rule (ΔCALL) is structurally analogous to (CALL), but it threads a differential store and a cache through the execution. This allows previously computed results stored in the cache to be reused in subsequent computations.

Since function bodies can contain arbitrary statements, this extension also makes IMP Turing-complete. We have formally proven that differential validity (Theorem 4.10), correctness (Theorem 4.11), and completeness (Theorem 4.12) hold for the recursive semantics in Rocq.

5.2 Differential Execution of While Loops

IMP supports repeat loops, where the number of iterations is known before the loop starts. This was useful for the differential semantics, because it allowed us to predict if we need the same, more, or less iterations after an input change. In particular, our differential semantics for repeat loops bailed out when the number of iterations had shrunk (ΔREP-SUB), which required knowledge of the new number of iterations ahead of time. However, it is also possible to support generic while loops, as we will show here. Note that while loops are *not* part of our mechanized metatheory.

While loops continue to iterate until the while condition evaluates to `false`. Hence, we cannot predict the number of iterations a priori. The best we can do is to start iterating the loop to propagate the changes using differential execution until either the original or the new iterations stop, meaning their control flow diverges. Now, we have three scenarios: The differential execution of the loop may terminate earlier, later, or after the same number of iterations as the original execution. However, since we already committed to a differential execution of the loop, it would be wasteful to bail now and to start a from-scratch execution from the beginning. Instead, we choose to cache the current store at each loop iteration.

$$\frac{\mathsf{while}^0\ e\ s, st, ch_0\ @\ tr \cdot \mathsf{while}_{\mathsf{aux}} \Rightarrow st', ch_1, i}{\mathsf{while}\ e\ s, st, ch_0\ @\ tr \Rightarrow st', ch_1[tr \mapsto \mathsf{count}\ i], 0} \quad (\textsc{while})$$

$$\frac{e, st, ch_0\ @\ tr \cdot \mathsf{while}_{\mathsf{cond}} \Rightarrow \mathsf{false}, ch_1}{\mathsf{while}^n\ e\ s, st, ch_0\ @\ tr \Rightarrow st, ch_1[tr \mapsto \mathsf{step}\ st], n} \quad (\textsc{while-false})$$

$$\frac{\begin{array}{l} e, st, ch_0\ @\ tr \cdot \mathsf{while}_{\mathsf{cond}} \Rightarrow \mathsf{true}, ch_1 \\ s, st, ch_1[tr \mapsto \mathsf{step}\ st]\ @\ tr \cdot \mathsf{while}_{\mathsf{body}} \Rightarrow st', ch_2, i_0 \\ \mathsf{while}^{n+1}\ e\ s, st', ch_2\ @\ tr \cdot \mathsf{while}_{\mathsf{iter}} \Rightarrow st'', ch_3, i \end{array}}{\mathsf{while}^n\ e\ s, st, ch_0\ @\ tr \Rightarrow st'', ch_3, i} \quad (\textsc{while-true})$$

$$\frac{ch_0\ tr = \mathsf{count}\ i \qquad \mathsf{while}^i\ e\ s, \Delta st, ch_0\ @\ tr \cdot \mathsf{while}_{\mathsf{aux}} \Rightarrow_\Delta \Delta st', ch_1, \Delta i}{\mathsf{while}\ e\ s, \Delta st, ch_0\ @\ tr \Rightarrow_\Delta \Delta st', ch_1[tr \mapsto \mathsf{count}\ i \oplus \Delta i], +0} \quad (\Delta\textsc{while})$$

$$\frac{e, \Delta st, ch_0\ @\ tr \cdot \mathsf{while}_{\mathsf{cond}} \Rightarrow_\Delta \boxed{\mathsf{noc}}, ch_1}{\mathsf{while}^{\boxed{0}}\ e\ s, \Delta st, ch_0\ @\ tr \Rightarrow_\Delta \Delta st, ch_1, +0} \quad (\Delta\textsc{while-both-stop})$$

$$\frac{\begin{array}{l} e, \Delta st, ch_0\ @\ tr \cdot \mathsf{while}_{\mathsf{cond}} \Rightarrow_\Delta \boxed{\mathsf{noc}}, ch_1 \\ s, \Delta st, ch_1\ @\ tr \cdot \mathsf{while}_{\mathsf{body}} \Rightarrow_\Delta \Delta st', ch_2, \Delta i_0 \\ \mathsf{while}^n\ e\ s, \Delta st', ch_2\ @\ tr \cdot \mathsf{while}_{\mathsf{iter}} \Rightarrow_\Delta \Delta st'', ch_3, \Delta i \end{array}}{\mathsf{while}^{\boxed{n+1}}\ e\ s, \Delta st, ch_0\ @\ tr \Rightarrow_\Delta \Delta st'', ch_3, \Delta i} \quad (\Delta\textsc{while-both-step})$$

$$\frac{\begin{array}{l} e, \Delta st, ch_0\ @\ tr \cdot \mathsf{while}_{\mathsf{cond}} \Rightarrow_\Delta \boxed{\mathsf{neg}}, ch_1 \\ ch_0\ tr = \mathsf{step}\ st \qquad \mathsf{while}^0\ e\ s, st \oplus \Delta st, ch_1\ @\ tr \Rightarrow_\Delta st', ch_2, i \end{array}}{\mathsf{while}^{\boxed{0}}\ e\ s, \Delta st, ch_0\ @\ tr \Rightarrow_\Delta st' \ominus st, ch_2, +i} \quad (\Delta\textsc{while-run-longer})$$

$$\frac{\begin{array}{l} e, \Delta st, ch_0\ @\ tr \cdot \mathsf{while}_{\mathsf{cond}} \Rightarrow_\Delta \boxed{\mathsf{neg}}, ch_1 \\ ch_0\ tr = \mathsf{step}\ st \qquad ch_0\ (tr +\!\!+ \mathsf{while}_{\mathsf{iter}}{}^{n+1}) = \mathsf{step}\ st' \qquad ch_2 = ch_1[tr \mapsto st \oplus \Delta st] \end{array}}{\mathsf{while}^{\boxed{n+1}}\ e\ s, \Delta st, ch_0\ @\ tr \Rightarrow_\Delta (st \oplus \Delta st) \ominus st', ch_2, -(n+1)} \quad (\Delta\textsc{while-stop-early})$$

Fig. 9. Big-step and differential semantics for while loops and the new iteration count.

We introduce the following syntax to describe the semantics of while loops and their lexical traces.

$$\begin{array}{lll} \text{(statements)} & s & ::= \ldots \mid \mathsf{while}\ e\ s \mid \mathsf{while}^n\ e\ s \\ \text{(trace links)} & l & ::= \ldots \mid \mathsf{while}_{\mathsf{aux}} \mid \mathsf{while}_{\mathsf{cond}} \mid \mathsf{while}_{\mathsf{body}} \mid \mathsf{while}_{\mathsf{iter}} \end{array}$$

Besides the standard while loop $\mathsf{while}\ e\ s$, we also add an auxiliary construct $\mathsf{while}^n\ e\ s$. In the non-differential semantics, $\mathsf{while}^n\ e\ s$ counts upward how many iterations n of the loop have *already happened*. But in the differential semantics, $\mathsf{while}^n\ e\ s$ counts downward how many iterations n are left before the original execution stopped. This allows us to detect which execution stops a loop earlier.

We show the big-step semantics of while loops in Figure 9. Rule (WHILE) initializes the iteration at $\text{while}^0\ e\ s$ and retrieves the final iteration count i from the sub-computation. Indeed, as highlighted, we had to extend the big-step relation to produce the iteration count of $\text{while}^n\ e\ s$ loops, and the differential relation to compute how the iteration count changes.[1] The sole purpose of these counts is to allow caching at the end of the loop in (WHILE). Rule (WHILE-FALSE) stops the loop when the condition evaluates to false, and caches the current store. Rule (WHILE-TRUE) caches the store, runs the body, and continues the loop with an incremented iteration count.

Figure 9 also shows the differential execution rules for while loops. Rule (ΔWHILE) retrieves the original iteration count i from the cache and initializes the differential iteration at $\text{while}^i\ e\ s$. Once the iteration is finished, we update the cached iteration count using the iteration change Δi. The next two rules (ΔWHILE-BOTH-STOP) and (ΔWHILE-BOTH-STEP) handle the cases, where original and differential execution agree on the iteration length: The evaluation of the condition e is $\cdot$. When we finished all original iterations and reach $\text{while}^0\ e\ s$ in (ΔWHILE-BOTH-STOP), the iteration count remains the same. When instead there are original iteration left $\text{while}^{n+1}\ e\ s$ and the condition is unchanged (i.e., it is still true), we differentially run the body s once and continue with one less original iterations $\text{while}^n\ e\ s$.

But what if the iteration counts of the original and the differential execution disagree. Rule (ΔWHILE-RUN-LONGER) fires when we reach $\text{while}^0\ e\ s$ (all original iterations are finished) but the condition's values is negated. This means the original condition was false, but now it is true. Hence, we must continue running the loop longer than before. To this end, we start a from-scratch execution after the original iterations, and mark the extra iterations i as a change to the iteration count. Rule (ΔWHILE-STOP-EARLY) handles the opposite case: We reach $\text{while}^{n+1}\ e\ s$ (there are more original iterations) but again the condition's values is negated. This means the original condition was true, but now it is false, and we must stop the loop earlier than before. Here we can use a small trick: We retrieve the store at the end of the original loop by calculating the associated lexical trace $tr +\!\!+ \text{while}_{\text{iter}}{}^{n+1}$. This way, we can compute how the current up-to-date store $st \oplus \Delta st$ differs from the original store at the end of the loop st'. Lastly, we mark the surplus iterations $n+1$ as a negative iteration count change.

In summary, the differential semantics of while loops uses a clever trick: In the initial semantics, $\text{while}^i\ e\ s$ counts how many iterations have already been done, that is, i is growing. In contrast, in the differential semantics, $\text{while}^i\ e\ s$ counts how many original iterations we can still match with the differential run, that is, i is shrinking. This allows for an efficient handling of while loops during differential execution. We have formalized the semantics of while loops in Rocq and have implemented and tested it in Scala. However, we were not able to prove the metatheoretical properties of differential execution for while loops. The problem is that we need to relate the differential run to the initial run in the validity and correctness theorems, but the iteration schemes misalign: The initial semantics moves from $\text{while}^i\ e\ s$ to $\text{while}^{i+1}\ e\ s$, whereas the differential semantics moves from $\text{while}^{i+1}\ e\ s$ to $\text{while}^i\ e\ s$. This makes it difficult to find a sufficiently strong induction hypothesis. It may be possible to design a less optimized semantics for while loops that avoids this problem, but we leave this for future work.

6 Implementation and Evaluation

We implemented the initializing and differential semantics of IMP as interpreters in Scala. The implementation follows the semantics very closely except for a few optimizations.

[1] We could have introduced an auxiliary reduction relation, but opted for this simpler version encoding where all other constructs yield a dummy iteration count 0.

```
enum Step:
  case Return; case SeqL; case SeqR(n: Int)
  case RepCount; case RepAux; case RepBody; case Iter(n: Int)
  ...
enum Trace:
  case Root
  case TraceStep(step: Step, tr: Trace)
  override val hashCode: Int = MurmurHash3.productHash(this)
  def push(s: Step): Trace = TraceStep(s, this)
  def pushIter(): Trace = this match
    case TraceStep(Step.Iter(n), tail) => TraceStep(Step.Iter(n + 1), tail)
    case _ => TraceStep(Step.Iter(1), this)
  def pushSeqR(): Trace = this match
    case TraceStep(Step.SeqR(n), tail) => TraceStep(Step.SeqR(n + 1), tail)
    case _ => TraceStep(Step.SeqR(1), this)
```

Fig. 10. Lexical traces in Scala. We compress subsequent occurrences of rep_{iter} and seq_{R}.

6.1 Optimizations in Implementing Differential Execution

The cache is passed around throughout the semantics, but we actually only ever access the latest version of the cache. Therefore, we can use a single shared cache with destructive updates instead. The implementation of the interpreters then follows a simple architecture:

```
class IncrementalInterpreter(prog: Program):
    val cache: Cache = new Cache
    val initialInterpreter = new InitialInterpreter(prog, cache)
    val differentialInterpreter = new DifferentialInterpreter(prog, cache, initialInterpreter)

    def runInitial(args: Seq[Value]): Value = initialInterpreter.run(args)
    def runDifferential(args: Seq[Value]): Value = differentialInterpreter.run(args)
```

Note that both interpreters receive access to the mutable cache. Moreover, the differential interpreter obtains a reference to the initial interpreter, as required by the semantics.

Traces are used as keys in the cache. We show our Scala encoding of lexical traces in Figure 10. As traces can grow long, the time required to compute hash values matters. We applied two simple optimizations to reduce this time. First, we compress traces: When multiple `SeqR` steps occur in sequence, we represent them as a single `SeqR(n)` in our implementation. We do the same for sequences of `RepIter`. Second, we cache the hash for each trace, such that the hash of `step · tr` can be computed in constant time independent of the size of `tr`. We believe that further trace compressions and more compact trace representations are possible, but have not investigated this yet.

6.2 Performance Evaluation

We proposed a principled approach to caching for differential execution: lexical traces that provide unique and stable access to the cache. Our approach improves over prior work on differential execution [Kumar et al. 2025] in that the caching behavior is formally specified and part of the mechanized theory. In contrast, Kumar et al. [2025] used extensive recomputation in their semantics and only considered caching in their implementation, which masks the effective caching semantics. But does our principled approach to caching have a negative impact on the initial or differential performance of the interpreter? To answer this question, we repeated the experiment of Kumar et al. [2025] using their and our implementation of a differential interpreter for IMP.

```
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
                }
                v = v + 1 }
            u = u + 1 }}}
```

Fig. 11. We evaluate the differential interpreters on the Bellman-Ford algorithm implemented in IMP.

We evaluate the initialization time and the incremental update time of the differential interpreters and compare them to a non-incremental baseline interpreter. We evaluate the interpreters on an IMP program that implements the Bellman-Ford algorithm: a single-source shortest path algorithm that supports negative edges and runs in cubic time. We took the code of the IMP program from Kumar et al. [2025] and reproduce it in Figure 11. The Bellman-Ford implementation uses a global mutable `table` to obtain the initial graph, where `table[u,v]` provides the edge weight between nodes `u` and `v`, both of which are identified by positive numbers. The algorithm then performs dynamic programming: At any point, `table[0,v]` contains the shortest path from the source node to `v` found so far.

We have extended our IMP interpreter to support the mutable `table` needed by Bellman-Ford. Semantically, `table` behaves like a store, but it uses index pairs `(u,v)` rather than variable names. In particular, the differential interpreter accepts table changes, which we use to encode changes to the initial graph given to Bellman-Ford. We adopt the initial graph and the graph changes used by Kumar et al. [2025]: The initial graph has $N \in \{10, 20, \ldots, 150\}$ nodes that form a single cycle, which means each edge is important for the final result. We then introduce new edges with high weight that do not affect the shortest paths of the source node, so that these changes are amenable to efficient incrementalization.

We measure the wall-clock running time of the baseline interpreter, the initial and differential interpreter by Kumar et al. [2025], and our new initial and differential interpreter using lexical tracing. We repeat each measurement 10 times and report the peak performance. We ran all measurements on an Apple M2 chip with 24GB memory running OSX 15.5. The interpreters are implemented in Scala 3, executed with OpenJDK 19.0.1 that uses up to 16 GB of memory for the heap. The benchmarking itself is done using the JMH framework.

We show the results in Figure 12. On the left-hand side, we see the initialization time in relation to the non-incremental baseline interpreter. We can see that differential interpretation has a considerable initialization overhead, mostly because it needs to populate the cache. However, our principled approach based on lexical traces only takes slightly longer compared to the prior work by

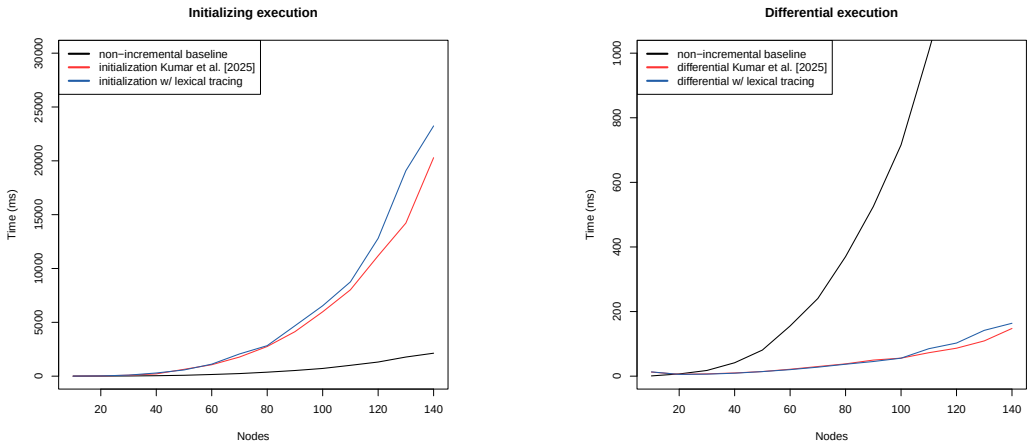


Fig. 12. Performance results. Left: Initialization time. Right: Differential updating time.

Kumar et al. [2025]. On the right-hand side, we see the incremental updating time of the differential interpreters in comparison to a from-scratch computation using the baseline interpreter. Here, we can see that (i) differential interpretation indeed provides asymptotic speedups and (ii) our principled approach does not introduce relevant performance penalties. We conclude that lexical tracing enables provably correct caching for differential execution while retaining the asymptotic performance advantages of differential execution.

6.3 Discussion: Where Do Incremental Performance Benefits Come From?

The main goal of incremental computing is to provide an asymptotic speedup over from-scratch recomputation. As the reader may have noticed, the operators used in our benchmark are simple arithmetic operators with a differential semantics that is not more efficient than its standard counterpart. Therefore, the performance benefits mainly stem from the incremental loop behavior and the rules for short-circuiting introduced by Kumar et al. [2025]. Short-circuiting allows us to skip the differential execution of statements that are not reading any changed variable.

For incremental loops, when the iteration count is unchanged, the loop only runs the differential semantics, yielding a speedup over the full recomputation. When the iteration count increases, only the additional iterations are executed with the standard semantics. On the other hand, when the iteration count decreases, we currently have to run all iterations with the standard semantics. However, this can be mitigated by caching the store after each iteration instead of only caching it at the end of the loop. This introduces a space-time tradeoff that we plan to investigate in our future work. Caching every n-th iteration as a checkpoint instead of caching every iteration is also a possible solution that requires working out efficient heuristics.

Although the differential version of the arithmetic operators is not more efficient than the standard one, falling back to their standard version during differential execution requires a transition to standard execution and then back to the differential one. Such a transition requires caching the operands and the result to be able to patch the operands with their delta values, execute the operator and diff the result afterwards to insert its delta value into the delta store. The overhead of such a transition is higher than the small overhead of the differential operator.

Nevertheless, more complex data structures like linked lists have differential operators with asymptotic speedups. As a simple example, consider the length operator on lists. As elements are

inserted or removed from a list, a differential length operator can compute the length change in constant time $O(1)$, whereas a recomputation takes linear time $O(n)$. Since incorporating such operators would divert from our primary focus on caching and its correctness, we leave this to future and related work [Xu and Erdweg 2026].

7 Related Work

Prior work by Kumar et al. [2025] focuses on building a correct foundation for differential execution. They introduce a differential semantics with the properties of completeness, output validity, and from-scratch consistency. They also develop some optimization rules that significantly speed up the differential execution. Their semantics, however, recomputes the original store at every step, leaving caching as future work. In this paper, we build on the established correctness and provide efficiency by introducing provably correct caching, which is essential for gaining any performance out of incremental computing and for preserving the correctness of the differential semantics. With incorrect caching, the differential semantics can get stuck because of missing entries or, even worse, can lead to invalid results that are hard to debug, for example when outdated cache entries are used from previous runs. Finally, our idea behind lexical tracing is general enough to implement incremental caching for other languages, which will be the case in our future work.

Xu and Erdweg [2026] complement the work on differential execution by providing a theory for primitive differential operators that can be embedded into the differential semantics. Their differential operators are stateful: Each operator can define its own internal state paired with a state invariant. A stateful differential operator is correct if it maintains its state invariant and, given a valid state, it produces a from-scratch consistent output change. With our technique of lexical tracing, we can embed stateful differential operators into the differential semantics to associate each operator application with a unique cache key. The cache then includes the internal state of each differential operator application, enabling sound reuse of the operator state across different executions.

Dynamic-algorithm research maintains problem-specific data structures that let an update of size δ be handled in poly($|\delta|$) time—e.g. fully dynamic connectivity and minimal spanning trees [Holm et al. 2001], dynamic planar convex hulls [Brodal and Jacob 2002], and decremental shortest paths. These solutions achieve optimal asymptotic bounds for their narrow domains, yet demand bespoke invariants and deep expertise. Our work follows a complementary strategy: by embedding incrementality into the language semantics itself, ordinary imperative code inherits efficient updates "for free," avoiding the engineering effort of crafting special-purpose dynamic data structures.

Memoization systems such as lazy functional runtimes or memoization libraries reuse results at the granularity of pure function calls but collapse in the face of mutation [Abadi et al. 1996]. Self-adjusting computation (SAC) generalises memoization with dynamic dependency graphs and change propagation [Acar 2005; Acar et al. 2008, 2002]. Extensions add traceable data types [Acar et al. 2010] and demand-driven evaluation (Adapton) [Hammer et al. 2014]. While SAC excels when control flow stays unchanged, branching or loop-bound edits often trigger whole-graph rebuilding. Our differential semantics bypasses dependency graphs altogether: change functions derived from the operational rules propagate updates directly, and lexical traces yield stable cache keys even when control flow diverges.

Memoization and self-adjusting computations fundamentally are based on selective recomputation [Ramalingam and Reps 1993]: a sub-computation is recomputed whenever any of its inputs changes in the slightest. Many approaches realize selective recomputing through dirt flagging: first mark all transitively affected inputs as dirty, then recompute all sub-computations with marked inputs. Selective-recomputation also underlies practical build systems and incremental compilers

(e.g., make, Bazel, many IDE services), which use dependency graphs or timestamp comparisons to decide what to rebuild. Our semantics generalises those heuristics to *fine-grained*, language-level keys: each operation's lexical trace plays the role of a build target, and cache entries record exactly the information needed to patch results after edits. And, of course, we perform differential computing and handle data changes (i.e., deltas) rather than changed data.

Semantics-based transformational incrementalization (STI) systematically rewrites a program f into an incremental variant f' able to compute $f(x+\delta x)$ from $f(x)$ and δx [Liu 2024; Liu et al. 1998]. STI embeds explicit *cache*, *prune*, and *discover* phases that (i) choose intermediate results worth retaining, (ii) discard entries no longer relevant, and (iii) introduce auxiliary state when it reduces update cost. Indeed, STI materializes the new full output in a single pass as an instance of selective recomputing, while we compute how the output changes, which composes with other differential approaches. Moreover, STI is applied at *compile time* and requires the developer to regenerate f' whenever significant structural changes occur, while our run-time differential semantics needs only the original program and adapts automatically through lexical traces.

Like STI, the incremental lambda calculus (ILC) also uses compile-time rewriting. However, ILC performs a type-directed source transformation of the original program to produce a derivative program [Cai et al. 2014; Giarrusso et al. 2019]. Like our approach and unlike STI, ILC's derivative program operates on changes: the derivative program maps input changes to output changes, given the original input. Since ILC derivatives also require the original input, the original input has to be patched at the end of each change-processing round to prepare for the next round. Moreover, the generated code often needs subsequent optimizations that have to be incorporated into the program generator. We instead execute the *same source* under an alternative semantics that uses caching: caching limits recomputed inputs to those rules that really need it and optimization opportunities live in the evaluator, not in extra generated code. Finally, ILC supports higher-order pure code, but does not explain how to handle imperative programming features.

Relational and data-flow systems employ delta rules to maintain materialized views [Blakeley et al. 1986; Gupta and Mumick 1995]. Differential Datalog [Motik et al. 2015] and Differential Dataflow [McSherry et al. 2013] extend the idea to recursive and streaming settings with recursive aggregation [Szabó et al. 2018, 2021]. These approaches are domain-specific and only handle relational changes: inserted and deleted database tuples. We provide a general-purpose differential programming semantics and we mechanize its correctness and completeness theorems.

8 Conclusion

Starting from a big-step operational semantics, we (1) derived change-propagation rules that translate input deltas into output deltas, (2) proved from-scratch consistency and completeness in Rocq, and (3) implemented the semantics in an interpreter capable of transparently accelerating ordinary imperative programs. Specifically, this work defines a formally verified caching semantics in which every execution step is given a stable, unique key—its lexical trace—and shows that these traces are sufficient to guarantee from-scratch consistency under arbitrary control-flow variation. In proving correctness and completeness of the differential semantics, we established a novel theory about cache stability for lexical traces and their semantic rules. The resulting differential caching semantics is easy to implement, extensible, and yields the expected asymptotic improvements on incremental runs. This work forms the basis for the development of differential semantics for richer languages, where caching is an integral part of the formalization and correctness guarantees.

Acknowledgements

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

References

Martin 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*. 83–91.

Umut A. Acar. 2005. *Self-Adjusting Computation.* Ph. D. Dissertation. Carnegie Mellon University.

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

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

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

John 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*. 61–71.

Gerth Stølting Brodal and Riko Jacob. 2002. Dynamic Planar Convex Hull. In *43rd Symposium on Foundations of Computer Science (FOCS 2002), 16-19 November 2002, Vancouver, BC, Canada, Proceedings*. IEEE Computer Society, 617–626. doi:10.1109/SFCS.2002.1181985

Yan Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. 2014. 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*. 145–155.

Paolo G. Giarrusso, Yann Régis-Gianas, and Philipp Schuster. 2019. Incremental Lambda-Calculus in Cache-Transfer Style: Static Memoization by Program Transformation. In *Proceedings of the 28th European Symposium on Programming*. 553–580.

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

Michael A. Hammer, Kyle Y. J. Phang, Michael Hicks, and Jeffrey S. Foster. 2014. Adapton: Composable, Demand-Driven Incremental Computation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 156–166.

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

Gabriël Konat, Sebastian Erdweg, and Eelco Visser. 2018. Scalable incremental building with dynamic task dependencies. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. ACM, 76–86.

Prashant Kumar, André Pacak, and Sebastian Erdweg. 2025. Incremental Computing by Differential Execution. In *39th European Conference on Object-Oriented Programming, ECOOP 2025, June 30 to July 2, 2025, Bergen, Norway (LIPIcs, Vol. 333)*, Jonathan Aldrich and Alexandra Silva (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 20:1–20:24. doi:10.4230/LIPICS.ECOOP.2025.20

Yanhong A. Liu. 2024. Incremental Computation: What Is the Essence? (Invited Contribution). In *Proceedings of the 2024 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation, PEPM 2024, London, UK, 16 January 2024*, Gabriele Keller and Meng Wang (Eds.). ACM, 39–52. doi:10.1145/3635800.3637447

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

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

Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. 2015. Incremental Update of Datalog Materialisation: The Backward/Forward Algorithm. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*. 1560–1568.

Ganesan Ramalingam and Thomas Reps. 1993. A Categorized Bibliography on Incremental Computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 502–510.

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*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1–15. doi:10.1145/3453483.3454026

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