ADAPTON: Composable, Demand-Driven Incremental Computation

Abstract

Many researchers have proposed programming languages that support *incremental computation* (IC), which allows programs to be efficiently re-executed after a small change to the input. However, existing implementations of such languages have two important drawbacks. First, recomputation is oblivious to specific demands on the program output; that is, if a program input changes, all dependencies will be recomputed, even if an observer no longer requires certain outputs. Second, programs are made incremental as a unit, with little or no support for reusing results outside of their original context, e.g., when reordered.

To address these problems, we present λ_{ic}^{cdd}, a core calculus that applies a *demand-driven* semantics to incremental computation, tracking changes in a hierarchical fashion in a novel *demanded computation graph*. λ_{ic}^{cdd} also formalizes an explicit separation between inner, incremental computations and outer observers. This combination ensures λ_{ic}^{cdd} programs only recompute computations as demanded by observers, and allows inner computations to be reused more liberally. We present ADAPTON, an OCaml library implementing λ_{ic}^{cdd}. We evaluated ADAPTON on a range of benchmarks, and found that it provides reliable speedups, and in many cases dramatically outperforms state-of-the-art IC approaches.

1. Introduction

Incremental computation (IC), a.k.a. *self-adjusting computation*, is a technique for efficiently recomputing a function after making a small change to its input. In recent years, researchers have shown that for certain algorithms, inputs, and classes of input changes, IC delivers large, even *asymptotic* speed-ups over full reevaluation [5, 6]. IC has been developed in many different settings [11, 17, 19, 32], and has even been used to address open problems, e.g., in computational geometry [7].

IC systems work by recording *traces* of computations and then reusing portions of those traces as inputs change. Unfortunately, prior IC approaches have two major limitations in trace reuse. First, because traditional IC imposes a total ordering on traces (typically, because of reliance on the Dietz-Sleator order-maintenance data structure [9, 15]), several straightforward kinds of reuse are impossible. For example, consider using IC to implement a spreadsheet, so that visible formulae are minimally recomputed as spreadsheet cells are changed, hidden, and shown. Traditional IC omits three common reuse patterns:

- *Sharing*, in which a computation is used in different contexts, e.g., F(A1..A100) appears as a subformula in two different computations or cells. Previous IC systems would recompute the second F(A1..A100) rather than reuse the previous result.
- *Swapping*, in which the order of subcomputations is changed, e.g., changing F(A1..A100) + F(B1..B100) to F(B1..B100) ∗ F(A1..A100). Previous IC systems would recompute F(A1..A100) or F(B1..B100) due to their reliance on a total ordering.
- *Switching*, in which computations are toggled back and forth, e.g., a computation of F(A1..A50) is replaced by F(A51..A100), which is then replaced by F(A1..A100). Previous IC systems would recompute the F(A1..A50) results from scratch, even if that subcomputation could be reused.

A second major problem with prior IC approaches is that they are ill-suited to *interactive* computations, because they are inherently *eager*. When an input value is changed, all values derived from that input are updated. But in many interaction scenarios, users may not need such updates. For example, suppose cell S!B1 on spreadsheet S contains F(T!A1..T!A100); here S!Ai refers to cell Ai in spreadsheet S. Now suppose the user hides S and switches to T to edit A1 and other cells. Then there is no need to update S!B1 until the user switches back to S to display it. Yet standard IC recomputes dependencies on each change, regardless of demand.

In this paper, we introduce ADAPTON, a new IC approach realized in a functional programming language, that addresses the limitations discussed above. The key insight behind ADAPTON is to combine traditional IC-style reuse with a mechanism for memoizing thunks, as in *lazy computation*. In ADAPTON, updates to mutable ref cells signal the potential need for recomputation, but such recomputation is delayed until thunks accessing cells' dependents are force d. Under the hood, both ref s and thunks are implemented almost identically using a *demanded computation graph* (DCG). The DCG captures the *partial* order of which computation's results are used in which other computations. As a result, ADAPTON provides very efficient support of the sharing, swapping, and switching patterns, since partial computations can be reused more effectively. ADAPTON's laziness avoids recomputing undemanded values. (Section 2 gives a high-level overview of ADAPTON.)

We formalize ADAPTON as the core calculus λ_{ic}^{cdd}. Following Levy's call-by-push-value calculus [23], λ_{ic}^{cdd} includes explicit thunk and force primitives, to make laziness apparent in the language, and adds ref, get, and set to model changeable state. A key feature of λ_{ic}^{cdd}, and of ADAPTON, is that it explicitly separates *inner* computations—which may read but not write ref s—from *outer* computations, which can allocate and mutate ref s and thus potentially precipitate change propagation. We should note that, to our knowledge, this clear inner/outer separation is absent from previous treatments of IC. (Section 3 presents λ_{ic}^{cdd}.)

We formalize an incremental semantics for λ_{ic}^{cdd} that captures the notion of *prior knowledge*, which consists of the demanded computation traces of prior computations. This semantics declaratively specifies the process of reusing traces from prior knowledge by (locally) *patching* their inconsistencies. We prove that the patching process is *sound* in that patched results will match what (re)computation from scratch would have produced. (Section 4 presents our incremental semantics.)

We have implemented ADAPTON as an OCaml library (Section 5). We compared ADAPTON's performance against that of a traditional IC system using a range of standard subject programs

from the IC literature, e.g., map, filter, sort, etc. We created micro-benchmarks that invoke these programs with varying levels of demand (e.g., demand a single element vs. all elements) and with varying change patterns (e.g., swapping and switching).

Our results show that ADAPTON greatly outperforms traditional IC for lazy interactions: where traditional IC gets 2× to 20× speedups over naive recomputation, ADAPTON gets 7× to 2000× speedups. For one program (`mergesort`), traditional IC actually incurs a 6.5× *slowdown*, whereas ADAPTON provides a 300× speedup. ADAPTON provides similarly significant speedups on the switching and swapping patterns, whereas traditional IC often incurs substantial (4× to 500×) slowdowns. ADAPTON does not perform as well as traditional IC when all output is demanded—it can be 1.5× to 3.5× slower—but still achieves a significant speedup over naive recomputation.

As a more practical measure of ADAPTON's utility, we developed the ADAPTON SpreadSheet (AS^2), which uses ADAPTON as its recomputation engine. We found that ADAPTON successfully incrementalized the (stateless) specification for formulae evaluation; a pair of simple benchmarks showed speedups of up to 20× compared to naive recomputation. On the same benchmarks classic IC techniques performed poorly, always resulting in a slowdown (up to 100×). (Section 6 presents our experimental results.)

While most prior work on IC requires a total ordering of events, which compromises reuse, Ley-Wild et al. have recently studied *non-monotonic* changes [24, 26]. Non-monotonic IC supports the swapping pattern mentioned above, but does not support sharing or switching. Moreover, non-monotonic IC has never been implemented, and is (in our opinion) far more complicated than ADAPTON. An increasingly popular computational paradigm related to IC is functional-reactive programming (FRP) [13, 14, 21]. FRP provides some reuse of computations under a changing signal, but is more specialized than ADAPTON (and IC in general). We have implemented an FRP library using ADAPTON; for space reasons we relegate discussion of it to our supplemental technical report. (Section 7 compares to IC, non-monotonic IC, and FRP in detail.)

In sum, we believe that ADAPTON offers a compellingly simple, yet general approach for programming incremental, interactive computation.

2. Overview of ADAPTON

We illustrate how ADAPTON works using a simple example inspired by a typical user interaction with a spreadsheet. ADAPTON's programming model is based on an ML-like language with explicit primitives for thunks and mutable state, where changes to the latter eventually propagate to update previous results.

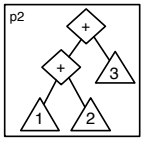
Consider the following toy language for formulae in spreadsheet cells:

```
type cell    = M formula
and  formula = Num of int | Plus of cell × cell
```

Values of type `cell` are formula addresses, i.e., mutable references containing a cell formula. Here the `M` constructor (for "mutable") is equivalent to `ref` type constructor in ML. A formula consists of either an integer (`Num`) or the sum of two other cells (`Plus`).

We begin by building an initial expression tree, illustrated to the right of the code below. Note that, since it is not otherwise indicated, all of this code runs at the *outer layer*, hence it is allowed to allocate new memory:

```
let n1 : cell = ref Num 1 in
let n2 : cell = ref Num 2 in
let n3 : cell = ref Num 3 in
let p1 : cell = ref Plus n1 n2 in
let p2 : cell = ref Plus p1 n3 in ···
```



Given a cell of interest, we can evaluate it using the obvious interpreter with a few extra calls for reuse:

```
eval : cell → U int
eval c = thunk (inner (case get c of
  | Num x       ⇒ x
  | Plus c1 c2 ⇒ force (eval c1) + force (eval c2) ))
```

Here `thunk` creates a suspended computation, which is typed by the `U` constructor, e.g., `U int` is equivalent to `unit → int`. Thunks are demanded by calling `force`, as in the recursive calls to `eval`. The body of the thunk is an expression wrapped in `inner`, indicating that expression occurs at the *inner layer*. Inner layer computations may only read `ref` cells—e.g., via the call to `get` above—and not allocate or change them. This restriction enables inner computations to be incrementally reused. Thus, in this example, each nested call to `eval` is a computation that can be updated and/or its results reused when `ref`s are modified at the outer layer. To illustrate how changes are made and propagated, consider the code shown at the top of Figure 1. Although written as a block of code, we envision the user entering this code a line at a time at an interactive top level. On lines 1 and 2, the user calls `eval` to produce a thunk that, when forced, will compute the values of `p1` and `p2`, respectively. Then on lines 3 and 4, the user forces evaluation and displays the result using a function

```
display : U int → unit   (* force arg and display it *)
```

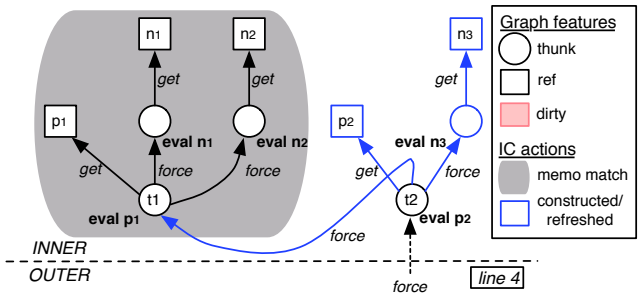
(In this example the user refreshes the display manually, rather than having it update automatically, to illustrate varying demand.) The computation on line 3 evaluates the formula of cell `p1`, which recursively forces the (trivial) evaluation of leaf cells `n1` and `n2`. Next, line 4 computes the value of `p2`. Since `p2` has `p1` as a subexpression, we would like to reuse the prior computation of `p1`. ADAPTON accomplishes this reuse via *demanded computation graphs* (DCGs).

Sharing. In ADAPTON, DCGs operate behind the scenes by recording inner layer computations. Figure 1a shows the DCG after evaluating line 4. Lines 1 and 2 only create thunks and otherwise do no computation, and line 3 essentially creates what is the left side of Figure 1a. We depict DCGs growing from the bottom of the *INNER* line upwards. We use a dotted line to identify operations performed at the outer layer that affect the inner layer graph, and color graph edges when they are touched as the result of that operation; edges are shown in blue when created or refreshed, and pink when they are dirtied. Rather than draw outer layer edges for `n1`, `t1`, `p2`, etc. to their respective nodes, we write the names in the nodes themselves, to avoid clutter.

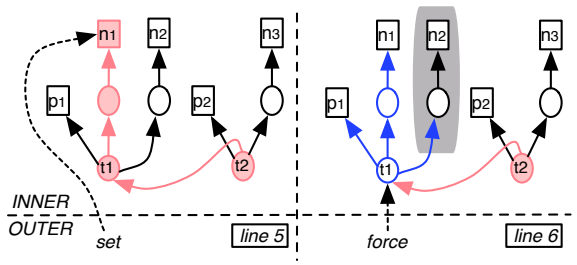
Each node in the DCG corresponds to a reference cell (depicted as a square) or a thunk (depicted as a circle). Edges that target reference cells are produced by `get` operations, and edges that target thunks are produced by `force` operations. Though not shown in the figure, each thunk node records a (suspended) expression and, once forced, its valuation; each reference node records the reference address and its content. Thunk nodes may have outgoing edges, where the leftmost edge was created first and the rightmost last. In the figure, the leftmost edge from `t2` goes to `p2` because `t2` is the result of `eval p2`, which calls `get` on its argument. The rightmost edge corresponds to the forced recursive `eval` call that gets `n3`. One key feature of ADAPTON is that it tries to memo-match previously computed results whenever possible. Here, the middle edge out of `t2` memo-matches the result of `eval p1` previously computed when `t1` was forced on line 3; memo-matched expressions are depicted with a gray background. Prior IC implementations would not be able to reuse `t2`'s result in this manner because they enforce a monolithic, total order of events; in this case we are reusing (i.e., *sharing*) a previous computation in the event history.

Dirtying and lazy change propagation Next, on line 5, the user decides to change the value of leaf `n1`, and on line 6, they de-

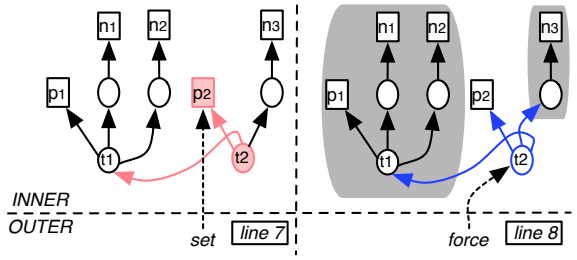
```
1 let t1 : U int = eval p1 in
2 let t2 : U int = eval p2 in
3 display t1;              (* demands (eval p1) *)
4 display t2;              (* memo matches (eval p1) *)
5 set n1 ← Num 5;          (* mutate leaf value *)
6 display t1;              (* does not re-eval p2 *)
7 set p2 ← Plus n3 p1;     (* swaps operand cells *)
8 display t2;              (* memo matches twice *)
```



(a) Sharing



(b) Dirtying and lazy change propagation



(c) Swapping

Figure 1: Spreadsheet example.

mand and display the updated result. Since a ref cell changed, the memoized thunk valuations may be inconsistent, and hence ADAPTON needs to repair the computation. Figure 1b shows how this repair and recomputation is done via the DCG. When the outer layer mutates the ref on line 5, ADAPTON performs *dirtying* to mark nodes and edges that may need recomputation. As shown in the figure, ADAPTON traverses the graph from the mutated node downwards, dirtying the nodes and edges that (transitively) depend on the changed reference cell n1 (viz., the thunks for eval n1 and eval p1). When the outer layer demands a computation on line 6, ADAPTON performs *propagation* by selectively traversing dirty nodes and edges from the bottom upwards and left to right, repairing dirty graph components by reevaluating dirty thunk nodes, which in turn replace their graph components with up-to-date versions; in the figure, the refreshed nodes/edges are in blue. Importantly, this traversal order reflects evaluation (and demand) order, allowing change propagation to lazily avoid repair to components not currently under demand. In this example, line 6 re-demands the first result for t1 but since there is no demand of t2, ADAPTON does not recompute its value.

Swapping. Next, on line 7, the outer layer updates p2 by *swapping* its two subcomponents. This kind of structural change defeats traditional IC reuse, but DCGs support it naturally. In the figure, we can see that the outer layer update on line 7 dirties an additional node and edge. Then line 8 demands the result t2, which initiates propagation to recompute the thunk eval p2. As shown in the figure, propagation is able to memo-match eval p1 (as in the original computation) and eval n3, even though they occur in a different order.

Switching. Finally, suppose the user updates expression p1 but then changes her mind and switches it back:

```
9  set p1 ← Num 4;       display t2;
10 set p1 ← Plus n1 n2;  display t2
```

After line 9 the eval p1 thunk in the graph would be updated to point to a new thunk that evaluates Num 4, and would no longer point to the eval n1 and eval n2 thunks. However, these thunks are still available for reuse, and when the user switches back on line 10, the eval p1 thunk is restored to its original state, memo-matching the previous eval n1 and eval n2 results. We call this pattern *switching* because it switches back to previously computed, but currently inactive, results; once again, traditional IC would fail to achieve reuse in this case.

3. Core calculus

We formalize ADAPTON as λ_{ic}^{cdd}, a core calculus for incremental computation in a setting with lazy evaluation. λ_{ic}^{cdd} is an extension of Levy's call-by-push-value (CBPV) calculus [22], which is a standard variant of the simply-typed lambda calculus with an explicit thunk primitive. It uses thunks as part of a mechanism to syntactically distinguish computations from values, and to make evaluation order syntactically explicit. λ_{ic}^{cdd} adds reference cells to the CBPV core, along with notation for specifying inner- and outer-layer computations.

As there exist standard translations from both call-by-value (CBV) and call-by-name (CBN) into CBPV, we intend λ_{ic}^{cdd} to be in some sense *canonical*, regardless of whether the host language is lazy or eager. We give a translation from a CBV language variant of λ_{ic}^{cdd} in the appendix of the supplemental technical report.

3.1 Syntax, typing and basic semantics for λ_{ic}^{cdd}

The top of Figure 2 gives the formal syntax of λ_{ic}^{cdd}, with new features highlighted. Figure 3 gives λ_{ic}^{cdd}'s type system. As most of the type rules are standard we weave discussion of them into our presentation of the language.

λ_{ic}^{cdd} inherits most of its syntax from CBPV. Terms consist of value terms (written v) and computation terms (written e), which we alternatively call expressions. Types consist of value types (written A, B) and computation types (written C, D). Standard value types consist of those for unit values $()$ (typed by 1), injective values $\mathbf{inj}_i\ v$ (typed as a sum $A + B$), pair values (v_1, v_2) (typed as a product $A \times B$) and thunk values $\mathbf{thunk}\, e$ (typed as a suspended computation $U\ C$).

Standard computation types consist of functions (typed by arrow $A \to C$, and introduced by $\lambda x.e$), and value-producers (typed by connective $F_\ell\ A$, and introduced by $\mathbf{ret}\, v$). These two term forms are special in that they correspond to the two introduction forms for computation types, and also the two *terminal* computation forms, i.e., the possible results of computations.

Values	v	::=	$x \mid () \mid (v_1, v_2) \mid \mathbf{inj}_i\ v \mid \mathbf{thunk}\ e \mid a$
Comps	e	::=	$\lambda x.e \mid e\ v \mid \mathbf{let}\ x \leftarrow e_1\ \mathbf{in}\ e_2 \mid \mathbf{ret}\ v$ $\mid \mathbf{fix} f.e \mid f \mid \mathbf{case}\ (v, x_1.e_1, x_2.e_2)$ $\mid \mathbf{split}\ (v, x_1.x_2.e) \mid \mathbf{force}_\ell\ v \mid \mathbf{inner}\ e$ $\mid \mathbf{ref}\ v \mid \mathbf{get}\ v \mid \mathbf{set}\ v_1 \leftarrow v_2$

Value types	$A,\ B$	::=	$1 \mid A + B \mid A \times B \mid \mathsf{U}\ C \mid \mathsf{M}\ A$
Comp. types	$C,\ D$	::=	$A \to C \mid \mathsf{F}_\ell\ A$
Comp. layers	ℓ	::=	$\mathsf{inner} \mid \mathsf{outer}$
Typing env.	Γ	::=	$\varepsilon \mid \Gamma, x{:}A \mid \Gamma, f{:}C \mid \Gamma, a{:}A$

Store	S	::=	$\varepsilon \mid S, a{:}v$
Terminal comps	$\tilde{e}$	::=	$\lambda x.e \mid \mathbf{ret}\ v$

Figure 2: Values and computations: Term and type syntaxes.

Other standard computation terms consist of function application (eliminates $A \to C$), **let** binding (eliminates $\mathsf{F}_\ell\ A$), fixed point computations (**fix** $f.e$ binds f recursively in its body e), pair splitting (eliminates $A \times B$), case analysis (eliminates $A + B$), and thunk forcing (eliminates $\mathsf{U}\ C$).

Mutable stores and computation layers. The remaining (highlighted) forms are specific to $\lambda^{\mathsf{cdd}}_{\mathsf{ic}}$; they implement *mutable stores* and *computation layers*. Mutable (outer layer) stores S map addresses a to values v. Addresses a are values; they introduce the type connective $\mathsf{M}\ A$, where A is the type of the value that they contain. The forms **ref**, **get** and **set** introduce, access and update store addresses, respectively.

The two layers of a $\lambda^{\mathsf{cdd}}_{\mathsf{ic}}$ program, outer and inner, are ranged over by layer meta variable ℓ. For informing the operational semantics and typing rules, layer annotations attach to force terms (viz., $\mathbf{force}_\ell\ v$) and the type connective for value-producing computations (viz., $\mathsf{F}_\ell\ A$). A term's layer determines how it may interact with the store. Inner layer computations may read from the store, as per the typing rule TyE-Get, while only outer layer computations may also allocate to it and mutate its contents, as enforced by typing rules TyE-Ref and TyE-Set. As per type rule TyE-Inner, inner layer computations e may be used in an outer context by applying the explicit coercion $\mathbf{inner}\ e$; the converse is not permitted. This rule employs the "layer coercion" function $(C)^\ell$, defined in Figure 5, to enforce layer purity in a computation. It is also used to similar purpose in rules TyE-Inner and TyE-Force. The TyE-Inner rule employs the environment transformation $|\Gamma|$, which filters occurrences of recursive variables f from Γ, thus making the outer layer's recursive functions unavailable to the inner layer.

Operational Semantics. The basic reduction semantics for $\lambda^{\mathsf{cdd}}_{\mathsf{ic}}$ proves judgments of the form $S_1 \vdash e \Downarrow S_2; \tilde{e}$, read as "under S_1, computation expression e reduces to terminal $\tilde{e}$, producing store S_2." For space reasons we omit the semantics; they can be found in the appendix of the supplemental technical report. Additionally, they can easily be recovered from the incremental semantics, given in the next section—where the incremental semantics uses a trace, the standard semantics uses a value that corresponds to the last element of the trace.

4. Incremental semantics

In Figure 4, we give the incremental semantics of $\lambda^{\mathsf{cdd}}_{\mathsf{ic}}$. It defines the *reduction to traces* judgment $K; S_1 \vdash e \Downarrow S_2; T$, read as "under prior knowledge K and store S_1, expression e reduces to store S_2 and trace T." We refer to our traces as *demanded computation traces* (DCT) as they record what thunks and suspended expressions a computation has demanded; these are the analogue of the DCG presented in Section 2. Prior knowledge is simply a list of such traces. The first time we evaluate e we will have an empty store and no prior knowledge. As e evaluates, the traces of sub-computations will get added to the prior knowledge K under which subsequent sub-computations are evaluated. If the outer layer mutates the store, this knowledge may be used to support *change propagation*, written $K; S \vdash T_1 \curvearrowright_{\mathsf{prop}} T_2$. The given rules are sound, but non-deterministic and non-algorithmic; a practical, deterministic algorithm is given in Section 5.1.

Prior knowledge	K	::=	$\varepsilon \mid K, T$
Traces	T	::=	$T_1 \cdot T_2 \mid t \mid \tilde{e}$
Trace events	t	::=	$\mathsf{force}^e_{\tilde{e}}[T] \mid \mathsf{get}^a_v$

$\boxed{\mathsf{trm}(T)}$

$$\mathsf{trm}(T_1 \cdot T_2) = \mathsf{trm}(T_2)$$
$$\mathsf{trm}(\mathsf{force}^e_{\tilde{e}}[T]) = \mathsf{trm}(T)$$
$$\mathsf{trm}(\mathsf{get}^a_v) = \mathbf{ret}\ v$$
$$\mathsf{trm}(\tilde{e}) = \tilde{e}$$

Figure 6: Traces and prior knowledge

4.1 Trace structure and propagation semantics

Prior knowledge and traces. Figure 6 defines our notions of prior knowledge and traces. Prior knowledge K consists of a list of traces from prior reductions. Traces T consist of sequences of trace events that end in a terminal expression. Trace events t record demanded computations. Traced events get^a_v record the address a that was read and the value v to which it mapped. Traced events $\mathsf{force}^e_{\tilde{e}}[T]$ record the thunk expression e that was forced, its terminal expression $\tilde{e}$ (i.e., the final term to which e originally reduced), and the trace T that was produced during its evaluation. Thus traces are *hierarchical*: trace events themselves contain traces which are locally consistent—there is no global ordering of all events. This allows change propagation to be more compositional, supporting, e.g., the sharing, switching and swapping patterns shown in Figures 1a to 1c. DCTs are closely related to the DCGs shown and discussed in Section 2; the key difference between the two, as the names suggest, is that DCGs are graphs whereas DCTs are trees. Hence, DCTs do not explicitly represent the sharing found in DCGs, but are easier to define and reason about formally; a shared sub-graph in the DCG corresponds to a a subtree in the DCT that is duplicated several times (one duplicate per use).

Figure 6 also defines $\mathsf{trm}(T)$ as the rightmost element of trace T, i.e., its terminal element, equivalent to $\tilde{e}$ in the normal evaluation judgment. It also defines when prior knowledge is well-formed.

Reduction to traces. Rules Incr-App and Incr-Bind are similar to a standard semantics, except that they use $\mathsf{trm}(T)$ to extract the lambda or return expression, respectively, and they add the trace T_1 from the first sub-expression's evaluation to the prior knowledge available to the second sub-expression. The traces produced from both are concatenated and returned from the entire computation.

Rule Incr-Force produces a force event; notice that the expression e from the thunk annotates the event, along with the trace T and the terminal expression $\tilde{e}$ at its end. Rule Incr-Get similarly produces a get event with the expected annotations. Rules Incr-Term, Incr-Ref, and Incr-Set all return the expected terminal expressions.

Rule Incr-ForceProp performs memoization of inner-layer forces by uses change propagation to repair the memoized trace. Importantly, we do *not* initiate change propagation at a set, and

$\boxed{\Gamma \vdash v : A}$ *(Under Γ, value v has value type A.)*

$$\textsc{TyV-var}\ \frac{\Gamma(x) = A}{\Gamma \vdash x : A} \qquad \textsc{TyV-unit}\ \frac{}{\Gamma \vdash () : 1} \qquad \textsc{TyV-inj}\ \frac{\mathbf{exists}\ i\ \mathbf{in}\ \{1,2\} \qquad \Gamma \vdash v : A_i}{\Gamma \vdash \mathbf{inj}_i\ v : A_1 + A_2}$$

$$\textsc{TyV-pair}\ \frac{\Gamma \vdash v_1 : A_1 \qquad \Gamma \vdash v_2 : A_2}{\Gamma \vdash (v_1, v_2) : A_1 \times A_2} \qquad \textsc{TyV-thunk}\ \frac{\Gamma \vdash e : C}{\Gamma \vdash \mathbf{thunk}\ e : \mathsf{U}\ C} \qquad \textsc{TyV-ref}\ \frac{\Gamma(a) = A}{\Gamma \vdash a : \mathsf{M}\ A}$$

$\boxed{\Gamma \vdash e : C}$ *(Under Γ, expression e has computation type C.)*

$$\textsc{TyE-var}\ \frac{\Gamma(f) = C}{\Gamma \vdash f : C} \qquad \textsc{TyE-lam}\ \frac{\Gamma, x{:}A \vdash e : C}{\Gamma \vdash \lambda x.e : A \to C} \qquad \textsc{TyE-ret}\ \frac{\Gamma \vdash v : A}{\Gamma \vdash \mathbf{ret}\ v : \mathsf{F}_\ell\ A}$$

$$\textsc{TyE-app}\ \frac{\Gamma \vdash e : A \to C \qquad \Gamma \vdash v : A}{\Gamma \vdash e\ v : C} \qquad \textsc{TyE-bind}\ \frac{\Gamma \vdash e_1 : \mathsf{F}_\ell\ A \qquad \Gamma, x{:}A \vdash e_2 : (C)^\ell}{\Gamma \vdash \mathbf{let}\ x \leftarrow e_1\ \mathbf{in}\ e_2 : (C)^\ell}$$

$$\textsc{TyE-case}\ \frac{\mathbf{forall}\ i\ \mathbf{in}\ \{1,2\} \qquad \Gamma \vdash v : A_1 + A_2 \qquad \Gamma, x_i{:}A_i \vdash e_i : C}{\Gamma \vdash \mathbf{case}\ (v, x_1.e_1, x_2.e_2) : C} \qquad \textsc{TyE-split}\ \frac{\Gamma \vdash v : A_1 \times A_2 \qquad \Gamma, x_1{:}A_1, x_2{:}A_2 \vdash e : C}{\Gamma \vdash \mathbf{split}\ (v, x_1.x_2.e) : C}$$

$$\textsc{TyE-fix}\ \frac{\Gamma, f{:}C \vdash e : C}{\Gamma \vdash \mathbf{fix} f.e : C} \qquad \textsc{TyE-force}\ \frac{\Gamma \vdash v : \mathsf{U}\ (C)^\ell}{\Gamma \vdash \mathbf{force}_\ell\ v : (C)^\ell} \qquad \textsc{TyE-inner}\ \frac{|\Gamma| \vdash e : (C)^{\mathsf{inner}}}{\Gamma \vdash \mathbf{inner}\ e : (C)^\ell}$$

$$\textsc{TyE-ref}\ \frac{\Gamma \vdash v : A}{\Gamma \vdash \mathbf{ref}\ v : \mathsf{F}_{\mathsf{outer}}\ \mathsf{M}\ A} \qquad \textsc{TyE-get}\ \frac{\Gamma \vdash v : \mathsf{M}\ A}{\Gamma \vdash \mathbf{get}\ v : \mathsf{F}_\ell\ A} \qquad \textsc{TyE-set}\ \frac{\Gamma \vdash v_1 : \mathsf{M}\ A \qquad \Gamma \vdash v_2 : A}{\Gamma \vdash \mathbf{set}\ v_1 \leftarrow v_2 : \mathsf{F}_{\mathsf{outer}}\ 1}$$

Figure 3: Typing semantics of $\lambda_{\mathsf{ic}}^{\mathsf{cdd}}$

$\boxed{K; S_1 \vdash e \Downarrow S_2; T}$ *(**Reduction** to traces: "Under knowledge K and store S_1, e reduces, yielding S_2 and trace T".)*

$$\textsc{Incr-term}\ \frac{}{K; S \vdash \tilde{e} \Downarrow S; \tilde{e}} \qquad \textsc{Incr-app}\ \frac{\mathsf{trm}(T_1) = \lambda x.e_2 \qquad K; S_1 \vdash e_1 \Downarrow S_2; T_1 \qquad K, T_1; S_2 \vdash e_2[v/x] \Downarrow S_3; T_2}{K; S_1 \vdash e_1\ v \Downarrow S_3; T_1 \cdot T_2}$$

$$\textsc{Incr-bind}\ \frac{\mathsf{trm}(T_1) = \mathbf{ret}\ v \qquad K; S_1 \vdash e_1 \Downarrow S_2; T_1 \qquad K, T_1; S_2 \vdash e_2[v/x] \Downarrow S_3; T_2}{K; S_1 \vdash \mathbf{let}\ x \leftarrow e_1\ \mathbf{in}\ e_2 \Downarrow S_3; T_1 \cdot T_2}$$

$$\textsc{Incr-case}\ \frac{K; S_1 \vdash e_i[v/x_i] \Downarrow S_2; T}{K; S_1 \vdash \mathbf{case}\ (\mathbf{inj}_i\ v, x_1.e_1, x_2.e_2) \Downarrow S_2; T} \qquad \textsc{Incr-split}\ \frac{K; S_1 \vdash e[v_1/x_1][v_2/x_2] \Downarrow S_2; T}{K; S_1 \vdash \mathbf{split}\ ((v_1, v_2), x_1.x_2.e) \Downarrow S_2; T}$$

$$\textsc{Incr-fix}\ \frac{K; S_1 \vdash e[\mathbf{fix} f.e/f] \Downarrow S_2; T}{K; S_1 \vdash \mathbf{fix} f.e \Downarrow S_2; T}$$

$$\textsc{Incr-force}\ \frac{K; S_1 \vdash e \Downarrow S_2; T \qquad \mathsf{trm}(T) = \tilde{e}}{K; S_1 \vdash \mathbf{force}_\ell\ (\mathbf{thunk}\ e) \Downarrow S_2; \mathsf{force}^e_{\tilde{e}}[T]}$$

$$\textsc{Incr-forceProp}\ \frac{\mathsf{force}^e_{\tilde{e}_1}[T_1] \in K \qquad K; S \vdash \mathsf{force}^e_{\tilde{e}_1}[T_1] \curvearrowright_{\mathsf{prop}} \mathsf{force}^e_{\tilde{e}_2}[T_2]}{K; S \vdash \mathbf{force}_{\mathsf{inner}}\ (\mathbf{thunk}\ e) \Downarrow S; \mathsf{force}^e_{\mathsf{trm}(T_2)}[T_2]}$$

$$\textsc{Incr-inner}\ \frac{K; S \vdash e \Downarrow S; T}{K; S \vdash \mathbf{inner}\ e \Downarrow S; T}$$

$$\textsc{Incr-ref}\ \frac{a \notin \mathsf{dom}(S)}{K; S \vdash \mathbf{ref}\ v \Downarrow S, a{:}v; \mathbf{ret}\ a} \qquad \textsc{Incr-get}\ \frac{S_1(a) = v}{K; S_1 \vdash \mathbf{get}\ a \Downarrow S_2; \mathsf{get}^a_v} \qquad \textsc{Incr-set}\ \frac{}{K; S \vdash \mathbf{set}\ a \leftarrow v \Downarrow S, a{:}v; \mathbf{ret}\ ()}$$

Figure 4: Operational semantics of $\lambda_{\mathsf{ic}}^{\mathsf{cdd}}$: Reduction (to traces), propagating incremental changes.

$\boxed{(C)^\ell}$ $(\mathsf{F}_{\ell_1}\ A)^{\ell_2} = \mathsf{F}_{\ell_2}\ A$ $(A \to C)^\ell = A \to (C)^\ell$

$\boxed{\Gamma_1 \vdash \Gamma_2}$ *(Under Γ_1, context Γ_2 is a consistent extension.)*

$$\textsc{Ext-refl}\ \frac{}{\Gamma \vdash \Gamma} \qquad \textsc{Ext-cons}\ \frac{\Gamma_1 \vdash \Gamma_2 \qquad a\ \mathbf{fresh\ for}\ \Gamma_2}{\Gamma_1 \vdash \Gamma_2, a{:}A}$$

$\boxed{\Gamma \vdash S\ \mathsf{wf}}$ *(Under Γ, store S is well-formed.)*

$$\textsc{Swf-emp}\ \frac{}{\Gamma \vdash \varepsilon\ \mathsf{wf}} \qquad \textsc{Swf-cons}\ \frac{\Gamma \vdash S\ \mathsf{wf} \qquad \Gamma \vdash a : \mathsf{M}\ A \qquad \Gamma \vdash v : A}{\Gamma \vdash S, a{:}v\ \mathsf{wf}}$$

Figure 5: Auxiliary typing judgements: Layer coercion, context extension and store typing.

thus we delay change propagation until a computation's result it is actually demanded. Rule INCR-FORCEPROP non-deterministically chooses a prior trace of a force of the same expression e from K (that is, it chooses a *memo match* for the computation e) and recursively switches to the propagating judgement described below. The prior trace to choose as the memo match is the first of two non-deterministic decisions of the incremental semantics; the second concerns the propagating specification, below.

Propagating changes by checking and patching. The change propagation judgment $K; S \vdash T_1 \curvearrowright_{\mathsf{prop}} T_2$ updates a trace T_1 to be T_2 according to knowledge K and the current store S.

$\boxed{K; S \vdash T_1 \curvearrowright_{\mathsf{prop}} T_2}$ ***(Change propagation)***

$$\textsc{Prop-checks}\ \frac{S \vdash T\ \surd}{K; S \vdash T \curvearrowright_{\mathsf{prop}} T} \qquad \textsc{Prop-patch}\ \frac{K; S \vdash e \Downarrow S'; T' \quad T_1\{e : T'\} \overset{\mathsf{patch}}{\rightsquigarrow} T_2 \quad K, T'; S \vdash T_2 \curvearrowright_{\mathsf{prop}} T_3}{K; S \vdash T_1 \curvearrowright_{\mathsf{prop}} T_3}$$

In the base case (rule PROP-CHECKS), there are no changes remaining to propagate through the given trace, which is consistent with the given store, as determined by the *checking judgment* $S \vdash T\ \surd$ (explained shortly). The recursive case (rule PROP-PATCH) arbitrarily chooses an expression e and reduces it to a trace T' under the current store S. (The choice of e is the second non-deterministic decision of this semantic specification.) This new subtrace is patched into the current trace according to the *patching judgment* $T_1\{e : T'\} \overset{\mathsf{patch}}{\rightsquigarrow} T_2$. The patched trace T_2 is processed recursively under prior knowledge expanded to include T', until the trace is ultimately made consistent.

The checking judgement, written $S \vdash T\ \surd$, ensures that a trace is consistent with a store.

CHECK-TRM	$S \vdash \tilde{e}\ \surd$	
CHECK-SEQ	$S \vdash T_1 \cdot T_2\ \surd$	when $S \vdash T_1\ \surd$ and $S \vdash T_2\ \surd$
CHECK-FORCE	$S \vdash \mathsf{force}^{e}_{\tilde{e}}[T]\ \surd$	when $\mathsf{trm}(T) = \tilde{e}$ and $S \vdash T\ \surd$
CHECK-GET	$S \vdash \mathsf{get}^{a}_{v}\ \surd$	when $S(a) = v$

The interesting rules are CHECK-FORCE and CHECK-GET. The first checks that the terminal expression $\tilde{e}$ produced by each force is consistent with the one last observed and recorded in the trace; i.e., it matches the terminal expression $\mathsf{trm}(T)$ of trace T. The second rule checks that the value retrieved from an address a is consistent with the current store.

The patching judgement is written as $\tilde{e}\{e : T\} \overset{\mathsf{patch}}{\rightsquigarrow} T'$:

$\tilde{e}\{e : T_2\} \overset{\mathsf{patch}}{\rightsquigarrow} \tilde{e}$

$(T_1 \cdot T_2)\{e : T_3\} \overset{\mathsf{patch}}{\rightsquigarrow} T_1' \cdot T_2'$ when $T_1\{e : T_3\} \overset{\mathsf{patch}}{\rightsquigarrow} T_1'$, $T_2\{e : T_3\} \overset{\mathsf{patch}}{\rightsquigarrow} T_2'$

$(\mathsf{force}^{e}_{\tilde{e}}[T_1])\{e : T_2\} \overset{\mathsf{patch}}{\rightsquigarrow} \mathsf{force}^{e}_{\tilde{e}}[T_2]$

$(\mathsf{force}^{e_1}_{\tilde{e}}[T_1])\{e_2 : T_2\} \overset{\mathsf{patch}}{\rightsquigarrow} \mathsf{force}^{e_1}_{\tilde{e}}[T_3]$ when $T_1\{e_2 : T_2\} \overset{\mathsf{patch}}{\rightsquigarrow} T_3$, $e_1 \neq e_2$

$(\mathsf{get}^{a}_{v})\{e : T_2\} \overset{\mathsf{patch}}{\rightsquigarrow} \mathsf{get}^{a}_{v}$

Conceptually, patching a DCT T simulatenously replaces all occurrences of a forced thunk's trace with an update-to-date version. The definition is above straightforward: All the rules are congruences except for the first force rule, which performs the actual patching. It substitutes the given trace for the existing trace of the forced expression in question, based on the *syntactic equivalence* of the forced expression e. This means that all force events whose forced computation is e will be updated "all at once," simulating the sharing pattern of DCGs.

In sum, the incremental semantics defined above is a declarative specification for an efficient implementation. Below, we prove that this specification is sound, in the sense that it always yields a result consistent with non-incremental evaluation. In the next section, we give an efficient algorithmic account of change propagation that conforms to this specification.

```
type 'a aref
val aref : 'a → 'a aref
val get : 'a aref → 'a
val set : 'a aref → 'a → unit

type 'a athunk
val force : 'a athunk → 'a
val thunk : (unit → 'a) → 'a athunk
val memo : ('fn → 'arg → 'a) → (('arg → 'a athunk) as 'fn)
```

Figure 7: Basic ADAPTON API

4.2 Meta theory of incremental semantics

The following theorem says that trace-based runs under empty knowledge in the incremental semantics are equivalent to runs in the basic (non-incremental) semantics.

Theorem 4.1 (Equivalence of blind evaluation)**.** $\varepsilon; S_1 \vdash e \Downarrow S_2; T$ *if and only if* $S_1 \vdash e \Downarrow S_2; \tilde{e}$ *where* $\tilde{e} = \mathsf{trm}(T)$

Next, we introduce well-formed knowledge, defined as

$\boxed{\Gamma \vdash K\ \mathsf{wf}}$ *(Under* Γ*, knowledge* K *is well formed.)*

$$\textsc{Kwf-emp}\ \frac{}{\Gamma \vdash \varepsilon\ \mathsf{wf}} \qquad \textsc{Kwf-cons}\ \frac{\Gamma \vdash K\ \mathsf{wf} \quad \Gamma \vdash S_1\ \mathsf{wf} \quad \varepsilon; S_1 \vdash e \Downarrow S_2; T}{\Gamma \vdash K, T\ \mathsf{wf}}$$

We now state and prove that the incremental semantics enjoys subject reduction.

Theorem 4.2 (Subject reduction)**.** *Suppose that* $\Gamma \vdash K$ *wf,* $\Gamma \vdash S_1$ *wf,* $\Gamma \vdash e : C$*, and* $K; S_1 \vdash e \Downarrow S_2; T$ *then there exists* Γ' *such that* $\Gamma \vdash \Gamma'$*,* $\Gamma' \vdash S_2$ *wf, and* $\Gamma' \vdash \mathsf{trm}(T) : C$

Finally, we prove that the incremental semantics is sound: when seeded with (well-formed) prior knowledge, there exists a consistent run in the basic (non-incremental) semantics.

Theorem 4.3 (Soundness)**.** *Suppose that* $\Gamma \vdash K$ *wf and* $\Gamma \vdash S_1$ *wf. Then* $K; S_1 \vdash e \Downarrow S_2; T$ *if and only if* $S_1 \vdash e \Downarrow S_2; \mathsf{trm}(T)$

This theorem establishes that every incremental reduction has a corresponding basic reduction, and vice versa. This correspondance establishes extensional consistency, i.e., the initial and final conditions of the runs are equivalent.

5. OCaml library

We have implemented ADAPTON as an OCaml library implements with the basic API shown in Figure 7. The fundamental data types are aref and athunk, corresponding to $\mathsf{M}\ A$ and $\mathsf{U}\ C$ in $\lambda^{\mathsf{cdd}}_{\mathsf{ic}}$, respectively. The functions operating on aref and athunk are named after their counterparts in $\lambda^{\mathsf{cdd}}_{\mathsf{ic}}$.

The last function, memo, solves a practical implementation issue while fixing the memoization choice left open by rule INCR-FORCEPROP in $\lambda^{\mathsf{cdd}}_{\mathsf{ic}}$ (Figure 4). In $\lambda^{\mathsf{cdd}}_{\mathsf{ic}}$, memoization is based on syntactic equality, and occurs implicitly at **force**, but we cannot perform syntactic equality checks in OCaml. As such, the memo function creates *memoized thunk constructors*, which are unary functions that return athunks. For example, we can define a memoized constructor that computes fibonacci:

```
let memo_fib = memo (fun memo_fib n → if n <= 1 then 1 else
  force (memo_fib (n − 1)) + force (memo_fib (n − 2)));;
print_int (force (memo_fib 10));; (* 89 *)
```

```
1  function dirty(node)
2      foreach edge ∈ node.incomingset do
3          if ¬ edge.dirty then
4              edge.dirty ⟵ true;
5              dirty(edge.source);
6  function propagate(node)
7      foreach edge ∈ node.outgoinglist do
8          if edge.dirty then
9              edge.dirty ⟵ false;
10             if edge.target is athunk then
11                 propagate(edge.target);
12             if edge.label ≠ edge.target.value then
13                 node.outgoinglist ⟵ [];
14                 evaluate(node);
15                 return;
```

Algorithm 1: ADAPTON core pseudocode.

memo takes a function of two arguments (here, memo_fib and n). It returns a constructor (here, also called memo_fib) that, when called with an argument (e.g., 10), first checks a memoization table to see if the constructor was previously called with the same argument. If not, the constructor creates an athunk that, when forced, computes its value by invoking the function with the constructor itself (to make recursive calls) and the argument passed to the constructor (e.g., 10); this athunk is then stored in the memoization table and returned. Otherwise, the constructor returns the same athunk as before. This simple choice of returning the same athunk is equivalent to choosing the most recently patched occurrence of a trace from the prior knowledge in rule INCR-FORCEPROP. To limit the size of memoization tables, we implement them using weak hash tables, relying on OCaml's garbage collector to eventually remove athunks that are no longer reachable.

ADAPTON does not provide an equivalent to inner in λ_{ic}^{cdd}, since OCaml's type system makes it hard to enforce layer separation statically. In ADAPTON, an inner level computation implicitly begins when force is called and ends when the call returns.

5.1 ADAPTON change propagation algorithm

As described in Section 4, λ_{ic}^{cdd} leaves open the choice of an expression e to patch in rule PROP-PATCH, allowing several possible change propagation algorithms. ADAPTON implements an efficient change propagation algorithm that avoids re-evaluating thunks unnecessarily.

ADAPTON operates on an acyclic *demanded computation graph* (DCG), corresponding to demanded computation traces T. Similar to the visualization in Section 2, each node in the graph represents an aref or an athunk, and each directed edge represents a dependency pointing from an athunk to another aref or athunk. The DCG is initially empty at the beginning of the execution. Nodes are added to the DCG whenever a new aref or athunk is created via aref, thunk, or a memo constructor (when memoization misses). Edges are added when an athunk calls get or force; edges are labeled with the value returned by that call. We maintain edges bidirectionally: Each node stores an ordered list of outgoing edges that is appended by each call to get or force, as well as an unordered set of incoming edges. This allows us to traverse the DCG from caller to callee or vice-versa.

As described in Section 2, the key to making ADAPTON efficient is to split change propagation into two phases—*dirtying* and *propagation*. Algorithm 1 lists the pseudocode for these two phases. The dirtying phase occurs when we make calls to set to update inputs to the incremental program. For each call to set, we traverse the DCG starting from the aref backward, marking all traversed edges as "dirty" (lines 1 to 5). Note that unlike Section 2, in our implementation only edges are dirtied; a node is implicitly dirty if any of its outgoing edges are dirty.

The propagation phase occurs when we make calls to force to demand results from the incremental program. For each call to force on an athunk, we perform an inorder traversal starting from the athunk's dirty outgoing edges, re-evaluating athunks as necessary. We check if we need to re-evaluate an athunk by iterating over its dirty outgoing edges in the order they were added (line 7). For each dirty edge, we first clean its dirty flag (line 9). If the target node is an athunk, we recursively check if we need to re-evaluate the target athunk (lines 10 to 11). Then, we compare the value of the target aref or athunk against the label of the outgoing edge (line 12). If the value is inconsistent, we know that at least one input to the athunk has changed, so we need to re-evaluate the athunk (line 14), and need not check its remaining outgoing edges (line 15). In fact, we first remove all its outgoing edges (line 13), since some edges may no longer be relevant due to the changed input; relevant edges will be re-added when get or force is called during re-evaluation. (We store incoming edges in weak hash tables, relying on OCaml's garbage collector to remove irrelevant edges.)

Note that, in both dirtying and propagation, we only traverse an edge if it is clean or dirty, respectively. We can do so because the above procedures maintain an invariant that, if an edge is dirty at the end of a dirtying or propagation phase, then all edges transitively reachable by traversing incoming edges beginning from the source node will also be dirty; dually, if an edge is clean, then all edges transitively reachable by traversing outgoing edges beginning from the target node will also be clean. This greatly improves efficiency by amortizing dirtying costs across consecutive calls to set, and propagation cost across consecutive calls to force.

By traversing the DCG inorder, ADAPTON re-evaluates inconsistent nodes in the same order as a standard non-incremental lazy evaluator would force thunks. Therefore, ADAPTON avoids re-evaluating nodes unnecessarily, such as athunks that were conditionally forced due to certain inputs, but may no longer be forced under updated inputs.

6. Empirical evaluation

This section evaluates ADAPTON's performance against traditional IC on micro benchmarks and larger example modeling a spreadsheet. We find that ADAPTON provides reliable speedups over naive recomputation and often significantly outperforms traditional IC.

6.1 Micro-benchmark

We ran a micro-benchmark to evaluate the effectiveness of ADAPTON in handling incremental programs that are *lazy*, as well as those that use the *swapping* and *switching* patterns. We also evaluate it on incremental *batch* programs, the target of traditional IC where there is no repetition in the input and the entire output is eagerly demanded.

- For *lazy* programs, we include standard list-manipulating benchmarks from the IC literature: filter, map, quicksort, and mergesort. We run each program on a randomly generated list of integers, and then demand the first item from the output list.
- For the *swapping* pattern, we use filter, map, fold applying min and sum, and exptree, an arithmetic expression tree evaluator similar to that in Section 2. We run each program on a randomly generated list of integers (or balanced expression tree) and demand the output. Then we randomly split the input into two

lists (or pick two subtrees at the half the height), swap those parts, and then re-demand the entire output.

- For the *switching* pattern, we wrote two programs, updown1 and updown2, that sort a list of integers in either ascending or descending order depending on another input. updown1 does the obvious thing, sorting the input list in one direction or the other, whereas updown2 first sorts the input list in both directions and then returns the appropriate one. (As we will show in the results, the odd structure of updown2 is necessary to achieve a speed-up in traditional IC.) After sorting an initial list of integers and demanding the first output, we randomly remove an item, randomly insert an item, toggle the sort direction, and then demand the first output again.
- Finally, for *batch* programs, we use the same programs as the *swapping* pattern, but instead of swapping parts of the input, we randomly insert or remove a single list item (or replace a leaf node in an expression tree with a binary node with two leaf nodes or vice-versa) before demanding the entire output.

We measure the time it takes to run ADAPTON in comparison to other variants that implement the same API. First, to compare against prior IC work, we implemented EagerTotalOrder, which uses the traditional, totally ordered, monolithic form of IC (in particular, [2]). Second, as baseline, we compare against standard lazy programs with LazyNonInc, which wraps lazy values and does not provide incremental semantics or memoization. In particular, 'a aref and 'a athunk are simply records containing 'a lazy and a unique ID, set throws an exception (thus requiring the program to recompute the results from scratch), and memo just calls thunk.

We compile the micro-benchmarks using OCaml 4.00.1 and run them on an 8-core, 2.26 GHz Intel Mac Pro with 16 GB of RAM running Mac OS X 10.6.8. We run 2, 4, or 8 programs in parallel, depending on the memory usage of the particular program. For most programs, we choose input sizes of 1e6 items. For quicksort and mergesort, we choose 1e5 items, and for updown1 and updown2, we choose 4e4 items, since these programs use up to 6GB of memory under EagerTotalOrder. We report the average of 8 runs for each program using seeds 1–8 to initialize OCaml's random number generator (to generate the input data, seed hash functions, etc.), and each run consists of 250 change-then-propagate cycles for changes that do not affect the input size. For changes that do affect the input size, we run 250 pairs of cycles, e.g., alternating between removing and inserting a list item, to ensure consistent input size.

In our initial evaluation, we observed that EagerTotalOrder spends a significant portion of time in the garbage collector (sometimes more than 50%), which has also been observed in prior work [3]. To mitigate this issue, we tweak OCaml's garbage collector under EagerTotalOrder, increasing the minor heap size from 2MB to 16MB and major heap increment from 1MB to 32MB.

Results. Table 1 summarizes the speed-up of ADAPTON EagerTotalOrder when performing each *incremental* change-then-propagate computation over LazyNonInc. We also highlight table cells in gray to indicate whether ADAPTON or EagerTotalOrder has a higher speed-up.

We can see that ADAPTON provides a speed-up to all patterns and programs. Also, ADAPTON is faster than EagerTotalOrder for the *lazy*, *swapping*, and *switching* patterns. These results validates the benefits of our approach.

For the *batch* pattern, ADAPTON gets only about half the speed-up of EagerTotalOrder. This is expected, since EagerTotalOrder is optimized for the *batch* pattern, whereas ADAPTON adds overhead that is unnecessary if all outputs are demanded. Interestingly, ADAPTON is faster for fold(min), since single changes are not as likely to affect the result of the min operation as compared to other operations such as sum.

	pattern	input #	ADAPTON speed-up over LazyNonInc	EagerTotalOrder speed-up over LazyNonInc
filter	lazy	1e6	12.8	2.24
map	lazy	1e6	7.80	1.53
quicksort	lazy	1e5	2020	22.9
mergesort	lazy	1e5	336	0.148
filter	swap	1e6	1.99	0.143
map	swap	1e6	2.36	0.248
fold(min)	swap	1e6	472	0.123
fold(sum)	swap	1e6	501	0.128
exptree	swap	1e6	667	10.1
updown1	switch	4e4	22.4	0.00247
updown2	switch	4e4	24.7	4.28
filter	batch	1e6	2.04	4.11
map	batch	1e6	2.21	3.32
fold(min)	batch	1e6	4350	3090
fold(sum)	batch	1e6	1640	4220
exptree	batch	1e6	248	746

Table 1: ADAPTON micro-benchmark results.

Conversely, EagerTotalOrder actually incurs a slowdown over LazyNonInc in many other cases. For *lazy* mergesort, EagerTotalOrder performs badly due to limited memoization between each internal recursion in mergesort. Prior work solved this problem by asking programmers to manually modify mergesort using techniques such as *adaptive memoization* [3] or *keyed allocation* [16]; we are currently investigating alternative approaches.

EagerTotalOrder also incurs slowdowns for *swapping* and *switching*, except for exptree and updown2. Unlike ADAPTON, EagerTotalOrder can only memo-match about half the input on average for changes considered by *swapping* due to its underlying total ordering assumption, and has to recompute the rest.

For updown1 in particular, the structure of the computation trace is such that EagerTotalOrder cannot memo-match any prior computation at all, and has to re-sort the input list every time the flag input is toggled. updown2 works around this limitation by unconditionally sorting the input list in both directions before returning the appropriate one, but this effectively wastes half the computation. In contrast, ADAPTON is equally effective for updown1 and updown2: It is able to memo-match the computation in updown1 regardless of the flag input, and, due to laziness, incurs no cost to unconditionally sort the input list twice in updown2.

Other experiments. In addition to the running times, we also measured the memory consumed by ADAPTON and EagerTotalOrder. ADAPTON uses 3–98% less memory than EagerTotalOrder for the *lazy*, *swapping*, and *switching* patterns, and it uses 103–120% more memory for the *batch* pattern. Our supplemental technical report contains a more detailed table summarizing memory usage as well as other results of our experiment.

Finally, we also ran ADAPTON on quicksort while varying the demand size (recall that in Table 1, one element is demanded for the lazy benchmarks). As expected, the speed-up decreases as demand size increases, but ADAPTON still outperforms EagerTotalOrder when demanding up to 1.8% of the output for quicksort. We also observed that the dirtying cost also increases with demand size. This is due to the interplay between dirtying and propagation phases: As more output is demanded, more edges will be cleaned by the propagation phase, and will have to be dirtied by the dirtying phase.

Our supplemental technical report provides more details of these observations and our elapsed-time experiments.

6.2 AS^2: An experiment in stateless spreadsheet design

As a more realistic case study of ADAPTON, we developed the ADAPTON SpreadSheet (AS^2), which implements basic spreadsheet functionality and uses ADAPTON to handle all change propagation as cells are updated. This is in contrast to conventional spreadsheet implementations, which have custom caching and dependence tracking logic.

In AS^2, spreadsheets are organized into a standard three-dimensional coordinate system of *sheets*, *rows* and *columns*, and each spreadsheet cell contains a formula. The language of formulae extends that of Section 2 with support for cell coordinates, arbitrary-precision rational numbers and binary operations over them, and aggregate functions such as max, min and sum. It also adds a command language for navigation (among sheets, rows and columns), cell mutation and display. For instance, the following script simulates the interaction from Section 2:

```
goto A1; =1; goto A2; =2; goto A3; =3;
goto B1; =(A1 + A2); goto B2; =(A1 + A3);
display B1; display B2; goto A1; =5; display B1;
goto B2; =(A3 + B1); display B2;
```

The explicit state of AS^2 consists simply of a mutable mapping of three-dimensional coordinates to cell formulae. We empirically study different implementations of AS^2 using a test script that simulates a user loading dense spreadsheet (with s sheets, ten rows and ten columns) and making a sequence of c random cell changes while observing the (entire) final sheet s:

```
scramble-all; goto s!a1; print; repeat c{scramble-one; print}
```

The scramble-all command initializes the formulae of each sheet such that sheet 1 holds random constants (uniformly in [0, 10k]), and for $i > 1$, sheet i consists of binary operations (drawn uniformly among $\{+, -, \div, \times\}$) over two random coordinates in sheet $i - 1$. scramble-one changes a randomly chosen cell to a random constant.

Figure 8 shows the performance of this test script. In the left plot, the number of sheets varies, and the number of changes is fixed at ten; in the right plot, the number of sheets is fixed at fifteen, and the number of changes varies. In both plots, we show the relative speedup/slowdown of ADAPTON and EagerTotalOrder to that of naive, stateless evaluation. The left plot shows that as the number of sheets grows, the benefit of ADAPTON increases. In fact, our measurements show that with only four sheets, the performance of the naive approach is overtaken by ADAPTON; the gap widens exponentially for more sheets. By contrast, EagerTotalOrder offers no incremental benefit, and the performance is *always* worse than the naive implementation, resulting in slowdowns that worsen as input sizes grow. We note that the speedups vary, depending on random choices made by both scramble-one and scramble-all.[1] The right plot shows that even for a fixed-sized spreadsheet, as the number of changes grows, the benefit of ADAPTON increases exponentially. As with the left plot, EagerTotalOrder again offers no incremental benefit, and always incurs a slowdown (e.g., at the right edges of each plot, we consistently measure slowdowns of 100x or more).

In both cases (more sheets and more changes), ADAPTON offers significant speedups over the naive stateless evaluator. These performance results makes sense: efficient AS^2 evaluation relies critically on the ability to reuse results multiple times within a computation (the *sharing* pattern). While ADAPTON supports this incremental pattern with ease, EagerTotalOrder fundamentally lacks the ability to do so, and instead only incurs a large performance penalty for its dependence-tracking overhead.

[1] We plot an average of eight randomized runs for each coordinate.

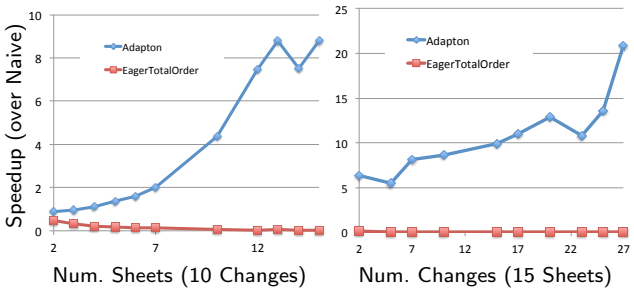


Figure 8: ADAPTON Spreadsheet (AS^2) performance tests.

7. Related Work

Incremental computation. The idea of memoization—improving efficiency by caching and reusing the results of pure computations—dates back to at least the late 1950's [8, 28, 29]. Incremental computation (IC), which has also been studied for decades [1, 20, 27, 30, 31], uses memoization to avoid unnecessary recomputation when input changes. Early IC approaches were promising, but are limited to caching final results.

Self-adjusting computation is a recent approach to IC that uses a special form of memoization that caches and reuses portions of *dynamic dependency graphs* (DDGs) of a computation. These DDGs are generated from conventional-looking programs with general recursion and fine-grained data dependencies [4, 10]. As a result, self-adjusting computation tolerates store-based differences between the pending computation being matched and its potential matches in the memo table; change-propagation repairs any inconsistencies in the matched graph. Researchers later combined these dynamic graphs with a special form of memoization, making the approach even more efficient and broadly applicable [2]. More recently, researchers have studied ways to make self-adjusting programs easier to write and reason about [11, 12, 25] and better performing [18, 19].

ADAPTON is similar to self-adjusting computation in that it applies to a conventional-looking language and tracks dynamic dependencies. However, as discussed in Sections 1 and 2, we make several advances over prior work in the setting of interactive, demand-driven computations. First, we formally characterize the semantics of the inner and outer layers working in concert, whereas all prior work simply ignored the outer layer (which is problematic for modeling interactivity). Second, we offer a compositional model that supports several key incremental patterns—sharing, switching, and swapping. All prior work on self-adjusting computation, which is based on maintaining a single totally ordered view of past computation, simply cannot handle these patterns.

Ley-Wild et al. have recently studied non-monotonic changes (viz., what we call "swapping"), giving a formal semantics and preliminary algorithmic designs [24, 26]. However, these semantics still assume a totally ordered, monolithic trace representation and hence are still of limited use for interactive settings, as discussed in Section 1. For instance, their techniques explicitly assume the absence of sharing (they assume all function invocations have unique arguments), and they do not support laziness, which they leave for future work. Additionally, to our knowledge, their techniques have no corresponding implementations.

Functional reactive programming (FRP). The chief aim of FRP is to provide a declarative means of specifying interactive and/or time-varying behavior [13, 14, 21]. FRP-based proposals share some commonalities with incremental computation; e.g., when an input signal is updated (due to an event like a key press, or simply

the passage of time), dependent computations are updated as well, and this update process may take advantage of memoization.

However, FRP's notion of incremental change is *implicit*, as part of its evaluation model, rather than an explicit mechanism one can program with, as with an IC system like ADAPTON. While it may be possible, it is hard to imagine writing an efficient incremental sorting algorithm using FRP. On the other hand, IC would seem to be an appropriate mechanism for implementing an FRP engine. As such, we have begun to experiment with an IC-based implementation of FRP using ADAPTON's abstractions, and plan further explorations in future work. More details about this FRP library are available in the appendix of our supplemental technical report.

8. Conclusion

Within the context of interactive, demand-driven scenerios, we identify key limitations in prior work on incremental computation. Specically, we show that certain idiomatic patterns naturally arise that result in incremental computations being *shared*, *switched* and *swapped*, each representing in an "edge case" that past work cannot handle efficiently. These limitations are a direct consequence of past works' tacit assumption that the maintained cache enabling incremental reuse is monolithic and totally-ordered.

To overcome these problems, we give a new, more composable approach that naturally expresses lazy (ie., demand-driven) evaluation that uses the notion of a thunk to identify reusable units of computation. This new approach naturally supports the idioms that were previously problematic. We executed this new approach both formally, as a core calculus that we prove is always consistent with full-reevaluation, as well as practically, as an OCaml library (viz., ADAPTON). We evaluated ADAPTON on a range of benchmarks, showing that it provides reliable speedups, and in many cases dramatically outperforms prior state-of-the-art IC approaches.

References

[1] M. Abadi, B. W. Lampson, and J.-J. Lévy. Analysis and caching of dependencies. In *International Conference on Functional Programming*, pages 83–91, 1996.

[2] U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. A library for self-adjusting computation. *Electronic Notes in Theoretical Computer Science*, 148(2), 2006.

[3] U. A. Acar, G. E. Blelloch, M. Blume, and K. Tangwongsan. An experimental analysis of self-adjusting computation. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2006.

[4] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(6):990–1034, 2006.

[5] U. A. Acar, A. Ihler, R. Mettu, and O. Sümer. Adaptive Bayesian inference. In *Neural Information Processing Systems (NIPS)*, 2007.

[6] U. A. Acar, G. E. Blelloch, K. Tangwongsan, and D. Türkoğlu. Robust kinetic convex hulls in 3D. In *Proceedings of the 16th Annual European Symposium on Algorithms*, Sept. 2008.

[7] U. A. Acar, A. Cotter, B. Hudson, and D. Türkoğlu. Dynamic well-spaced point sets. In *Symposium on Computational Geometry*, 2010.

[8] R. Bellman. *Dynamic Programming*. Princeton Univ. Press, 1957.

[9] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *ESA 2002*, pages 152–164. Springer, 2002.

[10] M. Carlsson. Monads for incremental computing. In *International Conference on Functional Programming*, pages 26–35, 2002.

[11] Y. Chen, J. Dunfield, M. A. Hammer, and U. A. Acar. Implicit self-adjusting computation for purely functional programs. In *Int'l Conference on Functional Programming (ICFP '11)*, pages 129–141, Sept. 2011.

[12] Y. Chen, J. Dunfield, and U. A. Acar. Type-directed automatic incrementalization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Jun 2012.

[13] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *In European Symposium on Programming*, pages 294–308, 2006.

[14] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for GUIs. In *PLDI*, 2013.

[15] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th ACM Symposium on Theory of Computing*, pages 365–372, 1987.

[16] M. Hammer and U. A. Acar. Memory management for self-adjusting computation. In *International Symposium on Memory Management*, pages 51–60, 2008.

[17] M. Hammer, U. A. Acar, M. Rajagopalan, and A. Ghuloum. A proposal for parallel self-adjusting computation. In *DAMP '07: Declarative Aspects of Multicore Programming*, 2007.

[18] M. Hammer, G. Neis, Y. Chen, and U. A. Acar. Self-adjusting stack machines. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011.

[19] M. A. Hammer, U. A. Acar, and Y. Chen. CEAL: a C-based language for self-adjusting computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[20] A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *Programming Language Design and Implementation*, pages 311–320, 2000.

[21] N. R. Krishnaswami and N. Benton. A semantic model for graphical user interfaces. In M. M. T. Chakravarty, Z. Hu, and O. Danvy, editors, *ICFP*, pages 45–57. ACM, 2011. ISBN 978-1-4503-0865-6.

[22] P. Levy. Call-by-push-value: A subsuming paradigm. *Typed Lambda Calculi and Applications*, pages 644–644, 1999.

[23] P. B. Levy. *Call-by-push-value: A Functional/imperative Synthesis*, volume 2. Springer, 2003.

[24] R. Ley-Wild. *Programmable Self-Adjusting Computation*. PhD thesis, Computer Science Department, Carnegie Mellon University, Oct. 2010.

[25] R. Ley-Wild, U. A. Acar, and M. Fluet. A cost semantics for self-adjusting computation. In *Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages*, 2009.

[26] R. Ley-Wild, U. A. Acar, and G. E. Blelloch. Non-monotonic self-adjusting computation. In *ESOP*, pages 476–496, 2012.

[27] Y. A. Liu and T. Teitelbaum. Systematic derivation of incremental programs. *Sci. Comput. Program.*, 24(1):1–39, 1995.

[28] J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

[29] D. Michie. "Memo" functions and machine learning. *Nature*, 218: 19–22, 1968.

[30] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Principles of Programming Languages*, pages 315–328, 1989.

[31] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Principles of Programming Languages*, pages 502–510, 1993.

[32] A. Shankar and R. Bodik. DITTO: Automatic incrementalization of data structure invariant checks (in Java). In *Programming Language Design and Implementation*, 2007.