Adapton: Composable, Demand-Driven Incremental Computation

CS-TR-5027 — July 12, 2013

Matthew A. Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S. Foster
University of Maryland, College Park, USA

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 composed more freely. We describe an algorithm for implementing λ_{ic}^{cdd} efficiently, and we present ADAPTON, a library for writing λ_{ic}^{cdd}-style programs in OCaml. We evaluated ADAPTON on a range of benchmarks, and found that it provides reliable speedups, and in many cases dramatically outperforms prior state-of-the-art IC approaches.

1. Introduction

Incremental computation (IC), also sometimes referred to as *self-adjusting computation*, is a technique for efficiently recomputing a function after making a small change to its input. A good application of IC is a spreadsheet. A user enters a column of data I, defines a function F over it (e.g., sorting), and stores the result in another column O. Later, when the user changes I (e.g., by inserting a cell), the spreadsheet will update O. Rather than re-sort the entire column, we could use IC to perform *change propagation*, which only performs an incremental amount of work to bring O up to date. For certain algorithms (even involved ones [5, 6]), certain inputs, and certain classes of input changes, IC delivers large, even *asymptotic* speed-ups over full reevaluation. IC has been developed in many different settings [12, 17, 19, 31], and has even been used to address open problems, e.g., in computational geometry [7].

Unfortunately, existing approaches to IC do not perform well when interactions with a program are unpredictable. To see the problem, we give an example, but first we introduce some terminology. IC systems stratify a computation into two distinct layers. The *inner layer* performs a computation whose inputs may later change. Under the hood, a *trace* of its dynamic dependencies is implicitly recorded and maintained (for efficiency, the trace may be represented as a graph). The *outer layer* actually changes these inputs and decides what to do with the (automatically updated) inner-layer outputs, i.e., in the context of the broader application. The problem arises when the outer layer would like to orchestrate inner layer computations based on dynamic information.

To see the issue, suppose there are two inner-layer computations, $\mathsf{F}(\mathsf{I})$ and $\mathsf{G}(\mathsf{I})$, and the application only ever displays the results of one or the other. For example, perhaps $\mathsf{F}(\mathsf{I})$ is on one spreadsheet pane, while $\mathsf{G}(\mathsf{I})$ is on another, and a flag P determines which pane is currently visible. There are two ways we could structure this computation. Option (A) is to define $\mathsf{F}(\mathsf{I})$ and $\mathsf{G}(\mathsf{I})$ as two inner-layer computations, and make the decision about what to display entirely at the outer layer. In this case, when the outer layer changes I, change propagation will update both $\mathsf{F}(\mathsf{I})$ and $\mathsf{G}(\mathsf{I})$, even though only one of them is actually displayed. Option (B) is to create one inner layer computation that performs *either* $\mathsf{F}(\mathsf{I})$ or $\mathsf{G}(\mathsf{I})$ based on a flag P, now also a changeable input. When I is updated, one of $\mathsf{F}(\mathsf{I})$ or $\mathsf{G}(\mathsf{I})$ is updated as usual. But when P is toggled, the prior work computing one of $\mathsf{F}(\mathsf{I})$ or $\mathsf{G}(\mathsf{I})$ is discarded. Thus, under many potential scenarios there is a lost opportunity to reuse work, e.g., if the user displays $\mathsf{F}(\mathsf{I})$, toggles to $\mathsf{G}(\mathsf{I})$, and then toggles back to $\mathsf{F}(\mathsf{I})$, the last will be recomputed from scratch. The underlying issue derives from the use of the Dietz-Sleator order maintenance data structure to represent the trace [10, 15]. This approach requires a *totally ordered*, *monolithic view* of inner layer computations as change propagation updates a trace to look just as it would had the computation been performed for the first time.

This monolithic view also conspires to prevent other useful compositions of inner and outer layer computations. A slight variation of the above scenario computes $\mathsf{X} = \mathsf{F}(\mathsf{I})$ unconditionally, and then depending on the flag P conditionally computes $\mathsf{G}(\mathsf{X})$. For technical reasons again related to Dietz-Sleator, Option (A) of putting the two function calls in separate inner layer computations, with the outer layer connecting them by a conditional on P, is not even permitted. Once again, this is dissatisfying because putting both in the same inner layer computation results in each change to P discarding work that might be fruitfully reused.

In this paper, we propose a new way of implementing IC that we call *Composable, Demand-driven Incremental Computation* ($\mathrm{CD^2IC}$), which addresses these problems toward the goal of efficiently handling interactive incremental computations. $\mathrm{CD^2IC}$'s centerpiece is a change propagation algorithm that takes advantage of *lazy evaluation*. Lazy evaluation is a highly compositional (and highly programmable) technique for expressing computational demand as a first-class concern: It allows programmers to delay computations in a suspended form (as “thunks”) until they are demanded (“forced”) by some outside observer. Just as lazy evaluation does not compute thunks that are not forced, our *demand-driven change propagation* ($\mathrm{D^2CP}$) algorithm performs no work until forced to; it even avoids recomputing results that were *previously* forced until they are forced again. As such, we can naturally employ Option (A) for the first example above, and change propagation will only take place for the demanded computation.

To implement D^{2}CP we use a novel form of execution trace we call the *demanded computation trace* (DCT), which in practice we represent as a graph (the DCG). Traced events are demanded computations, i.e., which thunks have been forced and which input (reference) cells have been read. Each force event references a subtrace of the events produced by running its corresponding thunk. When an input changes, these events will become inconsistent, but no changes are propagated immediately. Rather, when a thunk e is forced, the CD^{2}IC engine sees if it has been executed before, and attempts to reuse its result after making it consistent (via change propagation), if needed. Focusing change propagation on thunks makes prior computations more composable and reusable. For example, repeated executions of the same thunk will be reused, as with standard memoization, even within the same execution. Moreover, because trace elements are self-contained, and not globally ordered, each can be freely reordered and composed. For example, we can do things like map a function over a linked list, swap the front half and the back half of the list, and change propagation will update the result in a constant amount of work rather than recompute (at least) half of the list. Because our representation is not monolithic, we can also naturally intersperse inner and outer layer computations, e.g., to be able to employ Option (A) in the second example above.

We make several contributions in this paper. First, we formalize CD^{2}IC as the core calculus λ_{ic}^{cdd} (Section 3), which has two key features. Following Levy's call-by-push-value calculus [22], λ_{ic}^{cdd} includes explicit thunk and force primitives, to make laziness apparent in the language. In addition, λ_{ic}^{cdd} defines a notion of mutable store, employing a simple type system to enforce its correct usage for IC: inner layer computations may only read the store, and thus are locally pure, while outer layer computations may also update it.

We formalize D^{2}CP as an incremental semantics for λ_{ic}^{cdd} (Section 4). The semantics formalizes 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) repairing, or *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. We also give an algorithmic presentation of patching (Section 5), which makes the order of patching steps deterministic. We prove that the algorithm is sound with respect to the declarative semantics.

We have implemented CD^{2}IC in ADAPTON, an OCaml library for incremental computation (Section 6). ADAPTON provides a simple API for writing incremental programs and uses an efficient bidirectional graph-based data structure to realize the DCG and the D^{2}CP algorithm. Section 7 presents an empirical evaluation showing ADAPTON performs well for a number of interesting patterns that arise in applications of interactive incremental computations. For some patterns, ADAPTON is far superior to prior implementations of incremental computations, to which we compare our approach in Section 8. ADAPTON is publicly available.

2. Overview

This section illustrates our approach to composable, demand-driven incremental computation using a simple example, inspired by the idea of a user interacting with cells in a spreadsheet. Our programming model is based on an ML-like language with explicit primitives for thunks and mutable state, where changes to the latter may (eventually) propagate to update previously computed results. As

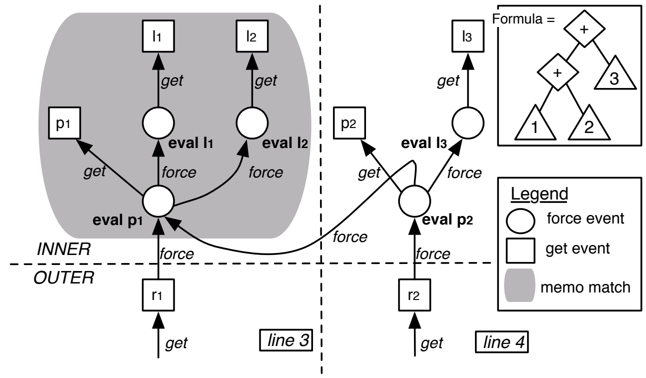

Figure 1: **Sharing:** Traces generated for lines 3 and 4.

usual, thunks are suspended computations, treated as values. We use the type connective U for typing thunks, whose introduction and elimination forms, respectively, correspond to the thunk and force keywords, illustrated below.

In addition, we have an outer layer that may create special reference cells for expressing incremental change; these mutable cells combine the features of ordinary reference cells and thunks. The reference cells are created, accessed and updated by the primitives ref, get and set, respectively, and typed by the M connective. Inner layer computations use get to access mutable state; the outer layer uses ref and set to create and mutate this state.

Now suppose that we have the following (toy) language for the formulae in spreadsheet cells:

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

Values of type cell are formula addresses, i.e., mutable references containing a cell formula. A formula either consists of a literal value (Leaf), or the sum of two other cell values (Plus). At the outer layer, we build an initial expression tree as follows (shown at the upper right of Figure 1):

```
let l1 : cell = ref Leaf 1 in
let l2 : cell = ref Leaf 2 in
let l3 : cell = ref Leaf 3 in
let p1 : cell = ref Plus l1 l2 in
let p2 : cell = ref Plus p1 l3 in ···
```

Given a cell of interest, we can evaluate it as follows:

```
eval : cell → int
eval c = force thunk( case (get c) of
  | Leaf x      ⇒ x
  | Plus c1 c2 ⇒ (eval c1) + (eval c2)
 (* end thunk *) )
```

This code corresponds to the obvious interpreter modulo the use of force, thunk and get. As we explain below, the role of these primitives here is not laziness (indeed, the introduced thunk is immediately forced); rather, the evaluator uses thunks to demarcate reusable work in future computations, to avoid its reevaluation. (Of course, thunks *can* be used for lazy computation as usual; we just do not use them in this way here.)

Now suppose we have a function display, whose behavior is to demand that a given reference cell be computed, and to display the result of this (integer-valued) computation to the user.

```
display : M int → unit
```

Finally, suppose that the user performs the following actions:

```
1 let r1 : M int = ref (inner eval p1) in
2 let r2 : M int = ref (inner eval p2) in
3 display r1;                (* demands (eval p1) *)
```

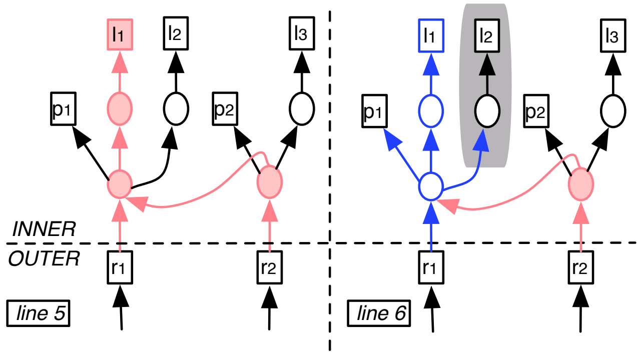

Figure 2: **Switching:** Traces generated for lines 5 and 6.

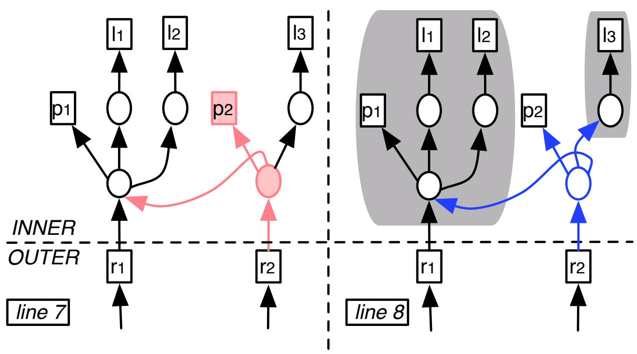

Figure 3: **Swapping:** Traces generated for lines 7 and 8.

```
4 display r2;                    (* memo matches (eval p1) *)
5 set l1 ← thunk(Leaf 5);        (* mutate leaf value *)
6 display r1;                    (* does not re-eval p2 *)
7 set p2 ← thunk(Plus l3 p1);    (* swaps operand cells *)
8 display r2;                    (* memo matches twice *)
```

Though shown as a straight-line script above, we imagine that the user issues commands to update cells and display results interactively. In line 1, the user creates a *suspended evaluation* of the formula of cell p1. Due to the inner keyword, this evaluation, when forced, will occur at the inner layer, and will thus have the opportunity to benefit from incrementality. That the computation is not performed eagerly illustrates how ref is non-standard in our language: The contents of reference cells are not necessarily values, and generally consist of suspended computations.[1] Line 2 is analogous to line 1: It creates a suspended, inner-layer computation that evaluates cell p2.

In lines 3 and 4, the user forces evaluation and displays the results. The computation in line 3 evaluates the formula of cell p1, which recursively forces the (trivial) evaluation of leaf cells l1 and l2. As we explain below, the computation in line 4 benefits from memoization: Since cell p2 has a formula that contains cell p1, it can simply reuse the result computed in the line 3. In line 5, the user decides to change the value of leaf l1, and (in line 6), they demand and display the updated result.

Demanded computation graphs. Behind the scenes, supporting incremental reuse relies on maintaining special dynamic records of inner layer computations. We call these dynamic records *demanded computation graphs*, or simply *graphs* for short. Figure 1 shows the maintained graph after evaluating line 3 (the left side) and then line 4 (the right side, which shares elements from the left side) in the listing above. (Lines 1 and 2 only allocate ref cells but otherwise do no computation, so we elide their graphs.) We depict graphs growing from the bottom upwards; we use a dotted line to distinguish operations performed at the inner layer from those at the outer layer.

The graph consists of the following structure. Each graph node corresponds to a reference cell (depicted as a square) or a thunk (depicted as a circle). Edges that target reference cells correspond to get operations, and edges that target thunks correspond to force operations. Locally, the outgoing edges of each node are ordered (depicted from left to right); however, edges and nodes do not have a globally total ordering, but instead only the partial ordering that corresponds directly to the graph structure. Nodes carry additional information which is not depicted for purposes of readability: Each thunk node records a (suspended) expression and, once forced, its valuation; each reference node records the reference address and its content (a suspended expression). Additionally, nodes and edges carry a *dirty flag* that indicates one of two (mutually exclusive) states: clean or dirty. We depict dirtiness with pink highlighting.

Programs intersperse computation with incremental reuse that is triggered by memo-matching previously generated graphs. We describe how to memo-match inconsistent graphs below. Initially, there are no inconsistencies, and memo-matching can immediately reuse previously computed results. We see this in Figure 1 for the part of the graph created for line 4, which memo-matches the computation of eval p1 already computed line 3, depicted with the gray background. This sort of reuse is disallowed in implementations of IC that enforce a monolithic, total order of events. For our approach memo-matching can occur not only *within*, but also *between* otherwise distinct inner layer computations, as is the case here. We generally refer to this pattern as *sharing*.

Demand-driven change propagation. When a memo-matched graph contains inconsistencies under the current store, reuse requires repair. Under the hood, the incremental behavior of a program actually consists of two distinct phases. Each phase processes the maintained graph: when the outer layer mutates reference cells, the *dirtying phase* sets the dirty flag of certain nodes and edges; when the outer layer re-forces a thunk already present in the graph, the *propagate phase* traverses the graph, 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.

Figure 2 depicts that after executing line 5 the dirtying phase traverses the graph from the top downwards, dirtying the nodes and edges that (transitively) depend on the changed reference cell l1 (viz., the thunks for eval l1 and eval p1). Then after executing line 6, the outer layer re-demands the first result r1, which in turn initiates propagation. This phase selectively traverses the dirty nodes and edges in the opposite direction, from the bottom upwards; it does not traverse clean edges or dirty edges that are not reachable from the demanded node. This is depicted on the right hand side of the figure by coloring the traversed edges in blue. Notably, neither the thunk eval p2 nor its dependencies are processed because they have not been demanded. We generally refer to this pattern as *switching* (of demand). This sort of demand-driven propagation is not implemented by prior work on IC.

In line 7, as depicted in Figure 3, the outer layer updates p2, which consequently dirties an additional node and edge. Line 8 demands the result r2 be redisplayed, which initiates another propagate phase that recomputes the thunk eval p2, but, as shown by the gray highlights in the figure, is able to memo-match two subcomponents, i.e., the graphs of eval p1 (as in the original com-

[1] This design choice is not fundemental; rather, it simply brings thunks and reference cells into a close correspondance, in terms of their typing and evaluation semantics (Section 3).

putation) and `eval l3`. Following memo-matching, these matched graph components swap their order, relative to the original computation. The reuse here is permitted to swap memo-matched components as needed since, unlike past work on incremental computation, CD^{2}IC does not enforce a globally total ordering of components. We generally refer to this pattern as *swapping*.

The next three sections formalize CD^{2}IC, and the following two describe our implementation and evaluation.

3. Core calculus

This section presents 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 make evaluation order syntactically explicit.[2] λ_{ic}^{cdd} adds reference cells to the CBPV core, along with notation for specifying inner- and outer-layer computations—inner layer computations may only read reference cells, while outer layer computations may change them and thus potentially precipitate change propagation.

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.

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

Figure 4 gives formal syntax of λ_{ic}^{cdd}, with new features highlighted. Figure 5 gives λ_{ic}^{cdd}'s type system. Figure 6 gives its basic evaluation relation as a big-step semantics, which we refer to as basic-λ_{ic}^{cdd}. In this semantics, we capture only non-incremental behavior; we formalize the incremental semantics later (Section 4) and use the basic-λ_{ic}^{cdd} semantics as its formal specification.

Standard CBPV elements. 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 $\mathsf{U}\ C$).

Standard computation types consist of functions (typed by arrow $A \rightarrow C$, and introduced by $\lambda x.e$), and value-producers (typed by connective $\mathsf{F}_\ell\ A$, 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 as per the big-step semantics in Figure 6.

Other standard computation terms consist of function application (eliminates $A \rightarrow C$), **let** binding (eliminates $\mathsf{F}_\ell\ A$), fixed point computations ($\mathbf{fix} f.e$ 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 λ_{ic}^{cdd}; they implement *mutable stores* and *computation layers*. Mutable (outer layer) stores S map addresses a to expressions e. Addresses a are values; they introduce the type connective $\mathsf{M}\ C$, where C is the type of the computation that they contain. The forms **ref**, **get** and **set** introduce, access

[2] The *"push"* in *"call-by-push-value"* merely refers to stack-based discipline for passing arguments from caller to callee.

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

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

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

Figure 4: Values and computations: Term and type syntaxes.

and update store addresses, respectively. It is somewhat unusual for stores to contain computations rather than values, but doing so creates pleasing symmetry between references and thunks, which we can see from the typing and operational semantics (though mapping addresses to values would create no difficulties).

The two layers of a λ_{ic}^{cdd} 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" auxiliary (total) function $(C)^\ell$ over computation types C to enforce layer purity in a computation; it is defined in Figure 7. 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.

3.2 Meta theory of basic λ_{ic}^{cdd}

We show that the λ_{ic}^{cdd} type system and the basic reduction semantics enjoy subject reduction. Judgments $\Gamma \vdash S_1\ \mathsf{ok}$ and $\Gamma \vdash \Gamma'$ used below are defined in Figure 7.

Theorem 3.1 (Subject reduction)**.** *Suppose that $\Gamma \vdash S_1$ ok, $\Gamma \vdash e : C$, and $S_1 \vdash e \Downarrow^n S_2; \tilde{e}$ then there exists Γ' such that $\Gamma \vdash \Gamma'$, $\Gamma' \vdash S_2$ ok, and $\Gamma' \vdash \tilde{e} : C$*

An analogous result for a small-step semantics, which we omit for space reasons, establishes that the type system guarantees progress. We show that the inner layer does not mutate the outer layer store (recall that the inner layer's only store effect is read-only access via **get**), and always that it yields deterministic results:

Theorem 3.2 (Inner purity)**.** *Suppose that $\Gamma \vdash e : (C)^{inner}$ and $S_1 \vdash e \Downarrow^n S_2; \tilde{e}$ then $S_1 = S_2$.*

Theorem 3.3 (Inner determinism)**.** *Suppose that $\Gamma \vdash e : (C)^{inner}$, $S_1 \vdash e \Downarrow^n S_2; \tilde{e}_2$, and $S_1 \vdash e \Downarrow^n S_3; \tilde{e}_3$ then $S_2 = S_3$ and $\tilde{e}_2 = \tilde{e}_3$.*

4. Incremental semantics

In Figure 9, we give the incremental semantics of λ_{ic}^{cdd}. It defines the *reduction to traces* judgment $K; S_1 \vdash e \Downarrow^n S_2; T$, which is the

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

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

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

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

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

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

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

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

$$\textsc{TyE-inner}\ \frac{|\Gamma| \vdash e : (C)^{\mathsf{inner}}}{\Gamma \vdash \textbf{inner}\ e : (C)^{\mathsf{outer}}} \qquad \textsc{TyE-ref}\ \frac{\Gamma \vdash e : C}{\Gamma \vdash \textbf{ref}\ e : \mathsf{F}_{\mathsf{outer}}\ \mathsf{M}\ C} \qquad \textsc{TyE-get}\ \frac{\Gamma \vdash v : \mathsf{M}\ C}{\Gamma \vdash \textbf{get}\ v : C}$$

$$\textsc{TyE-set}\ \frac{\Gamma \vdash v_1 : \mathsf{M}\ C \quad \Gamma \vdash v_2 : \mathsf{U}\ C}{\Gamma \vdash \textbf{set}\ v_1 \leftarrow v_2 : \mathsf{F}_{\mathsf{outer}}\ 1}$$

Figure 5: Typing semantics of $\lambda_{\mathsf{ic}}^{\mathsf{cdd}}$

$\boxed{S_1 \vdash e \Downarrow^n S_2; \tilde{e}}$ ***(Basic reduction)***

$$\textsc{Eval-term}\ \frac{}{S \vdash \tilde{e} \Downarrow^0 S; \tilde{e}} \qquad \textsc{Eval-app}\ \frac{S_1 \vdash e_1 \Downarrow^n S_2; \lambda x.e_2 \quad S_2 \vdash e_2[v/x] \Downarrow^m S_3; \tilde{e}}{S_1 \vdash e_1\ v \Downarrow^{n+m} S_3; \tilde{e}}$$

$$\textsc{Eval-bind}\ \frac{S_1 \vdash e_1 \Downarrow^n S_2; \textbf{ret}\ v \quad S_2 \vdash e_2[v/x] \Downarrow^m S_3; \tilde{e}}{S_1 \vdash \textbf{let}\ x \leftarrow e_1\ \textbf{in}\ e_2 \Downarrow^{n+m} S_3; \tilde{e}} \qquad \textsc{Eval-case}\ \frac{\textbf{exists } i \textbf{ in } \{1,2\} \quad S_1 \vdash e_i[v/x_i] \Downarrow^n S_2; \tilde{e}}{S_1 \vdash \textbf{case}\ (\textbf{inj}_i\ v, x_1.e_1, x_2.e_2) \Downarrow^n S_2; \tilde{e}}$$

$$\textsc{Eval-split}\ \frac{S_1 \vdash e[v_1/x_1][v_2/x_2] \Downarrow^n S_2; \tilde{e}}{S_1 \vdash \textbf{split}\ ((v_1, v_2), x_1.x_2.e) \Downarrow^n S_2; \tilde{e}} \qquad \textsc{Eval-fix}\ \frac{S_1 \vdash e[\textbf{fix} f.e/f] \Downarrow^n S_2; \tilde{e}}{S_1 \vdash \textbf{fix} f.e \Downarrow^n S_2; \tilde{e}}$$

$$\textsc{Eval-force}\ \frac{S_1 \vdash e \Downarrow^n S_2; \tilde{e}}{S_1 \vdash \textbf{force}_\ell\ (\textbf{thunk}\ e) \Downarrow^{n+1} S_2; \tilde{e}} \qquad \textsc{Eval-inner}\ \frac{S \vdash e \Downarrow^n S; \tilde{e}}{S \vdash \textbf{inner}\ e \Downarrow^n S; \tilde{e}}$$

$$\textsc{Eval-ref}\ \frac{a \notin \mathsf{dom}(S)}{S \vdash \textbf{ref}\ e \Downarrow^0 S, a{:}e; \textbf{ret}\ a} \qquad \textsc{Eval-get}\ \frac{S_1(a) = e \quad S_1 \vdash e \Downarrow^n S_2; \tilde{e}}{S_1 \vdash \textbf{get}\ a \Downarrow^{n+1} S_2; \tilde{e}} \qquad \textsc{Eval-set}\ \frac{}{S \vdash \textbf{set}\ a \leftarrow \textbf{thunk}\ e \Downarrow^0 S, a{:}e; \textbf{ret}\ ()}$$

Figure 6: Basic reduction semantics of $\lambda_{\mathsf{ic}}^{\mathsf{cdd}}$ (i.e., non-incremental evaluation).

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

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

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

$\boxed{\Gamma \vdash S\ \mathsf{ok}}$ *(Under Γ, store S types okay.)*

$$\textsc{Sok-emp}\ \frac{}{\Gamma \vdash \varepsilon\ \mathsf{ok}} \qquad \textsc{Sok-cons}\ \frac{\Gamma \vdash S\ \mathsf{ok} \quad \Gamma \vdash a : \mathsf{M}\ C \quad \Gamma \vdash e : C}{\Gamma \vdash S, a{:}e\ \mathsf{ok}}$$

Figure 7: Auxiliary typing judgements: Layer coercion, context extension and store typing.

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

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

$$\mathsf{trm}(T_1 \cdot T_2) = \mathsf{trm}(T_2)$$
$$\mathsf{trm}(\mathsf{force}^{e}_{\tilde{e}}[T]) = \mathsf{trm}(T)$$
$$\mathsf{trm}(\mathsf{get}^{a}_{e}[T]) = \mathsf{trm}(T)$$
$$\mathsf{trm}(\tilde{e}) = \tilde{e}$$

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

$$\text{KWF-EMP}\ \frac{}{\Gamma \vdash \varepsilon\ \mathsf{wf}} \qquad \text{KWF-CONS}\ \frac{\Gamma \vdash K\ \mathsf{wf} \quad \Gamma \vdash S_1\ \mathsf{ok} \quad \varepsilon; S_1 \vdash e \Downarrow^{n} S_2; T}{\Gamma \vdash K, T\ \mathsf{wf}}$$

Figure 8: Traces and prior knowledge

analogue to normal evaluation $S_1 \vdash e \Downarrow^{n} S_2; \tilde{e}$ from Figure 6. The reduction to traces judgment says that, 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. 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 subcomputations are evaluated. If the outer layer mutates the store, this knowledge may be used to support *demand-driven change propagation* (D²CP), written $K; S \vdash T_1 \curvearrowright^{n}_{\mathsf{prop}} T_2$. These judgements are instrumented with analytical costs (denoted by n, m and variants) to count the steps performed, as in the basic reduction semantics. The given rules are sound, but non-deterministic and nonalgorithmic; a deterministic algorithm is given in Section 5.

4.1 Trace structure and propagation semantics

We begin by giving the syntax and intuition behind our notions of prior knowledge and traces, and then describe the semantics of change propagation.

Prior knowledge and traces. Figure 8 defines our notions of prior knowledge and traces. Prior knowledge (written K) consists of a list of traces from prior reductions. Traces (written T) consist of sequences of trace events that end in a terminal expression. Trace events (written t) record demanded computations. Traced force events, written $\mathsf{force}^{e}_{\tilde{e}}[T]$, 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. Traced get events $\mathsf{get}^{a}_{e}[T]$ record the address a that was read, the expression e to which it mapped, and the trace T that was produced when e was subsequently evaluated. 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 1 to 3, respectively.

Figure 8 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. Most of the rules of the reduction to traces judgment at the top of Figure 9 are adaptations of the basic reduction semantics (Figure 6).

Rules INCR-APP and INCR-BIND are similar to their basic counterparts, 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.

Change propagation is initiated in rule INCR-FORCEPROP at an inner-layer force; importantly, we do *not* initiate change propagation at a set, and 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 and recursively switches to the propagating judgement described below. The prior trace to choose 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 $K; S \vdash T_1 \curvearrowright^{n}_{\mathsf{prop}} T_2$ updates a trace T_1 to be T_2 according to knowledge K and the current store S. 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 an arbitrarily chosen store S. (This 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 ensures that a trace is consistent. 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 expression gotten from an address a is consistent with the current store.

The patching judgement is straightforward. All the rules are congruences except for rule PATCH-TRUPDATE, which actually performs the 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.

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)**.**
$\varepsilon; S_1 \vdash e \Downarrow^{n} S_2; T$ *if and only if* $S_1 \vdash e \Downarrow^{n} S_2; \tilde{e}$ *where* $\tilde{e} = \mathit{trm}(T)$

We prove that the incremental semantics enjoys subject reduction.

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

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.

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

$$\textsc{Incr-term}\quad \frac{}{K; S \vdash \tilde{e} \Downarrow^{0} S; \tilde{e}}$$

$$\textsc{Incr-app}\quad \frac{\mathsf{trm}(T_1) = \lambda x.e_2 \qquad K; S_1 \vdash e_1 \Downarrow^{n} S_2; T_1 \qquad K, T_1; S_2 \vdash e_2[v/x] \Downarrow^{m} S_3; T_2}{K; S_1 \vdash e_1\ v \Downarrow^{n+m} S_3; T_1 \cdot T_2}$$

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

$$\textsc{Incr-case}\quad \frac{\mathbf{exists}\ i\ \mathbf{in}\, \{1, 2\} \qquad K; S_1 \vdash e_i[v/x_i] \Downarrow^{n} S_2; T}{K; S_1 \vdash \mathbf{case}\,(\mathbf{inj}_i\ v, x_1.e_1, x_2.e_2) \Downarrow^{n} S_2; T}$$

$$\textsc{Incr-split}\quad \frac{K; S_1 \vdash e[v_1/x_1][v_2/x_2] \Downarrow^{n} S_2; T}{K; S_1 \vdash \mathbf{split}\,((v_1, v_2), x_1.x_2.e) \Downarrow^{m} S_2; T}$$

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

$$\textsc{Incr-force}\quad \frac{K; S_1 \vdash e \Downarrow^{n} S_2; T \qquad \mathsf{trm}(T) = \tilde{e}}{K; S_1 \vdash \mathbf{force}_{\ell}\ (\mathbf{thunk}\, e) \Downarrow^{n+1} S_2; \mathsf{force}^{e}_{\tilde{e}}[T]}$$

$$\textsc{Incr-forceProp}\quad \frac{\mathsf{force}^{e}_{\tilde{e}_1}[T_1] \in K \qquad K; S \vdash \mathsf{force}^{e}_{\tilde{e}_1}[T_1] \curvearrowright^{n}_{\mathsf{prop}} \mathsf{force}^{e}_{\tilde{e}_2}[T_2]}{K; S \vdash \mathbf{force}_{\mathsf{inner}}\ (\mathbf{thunk}\, e) \Downarrow^{n+1} S; \mathsf{force}^{e}_{\mathsf{trm}(T_2)}[T_2]}$$

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

$$\textsc{Incr-ref}\quad \frac{a \notin \mathsf{dom}(S)}{K; S \vdash \mathbf{ref}\ e \Downarrow^{0} S, a{:}e; \mathbf{ret}\ a}$$

$$\textsc{Incr-get}\quad \frac{S_1(a) = e \qquad K; S_1 \vdash e \Downarrow^{n} S_2; T}{K; S_1 \vdash \mathbf{get}\ a \Downarrow^{n+1} S_2; \mathsf{get}^{a}_{e}[T]}$$

$$\textsc{Incr-set}\quad \frac{}{K; S \vdash \mathbf{set}\ a \leftarrow \mathbf{thunk}\, e \Downarrow^{0} S, a{:}e; \mathbf{ret}\,()}$$

$\boxed{K; S \vdash T_1 \curvearrowright^{\mathfrak{n}}_{\mathsf{prop}} T_2}$ ***(Change propagation**: "Under K, changes in store S propagate through trace T_1 in $\mathfrak{n}$ steps, yielding trace T_2".)*

$$\textsc{Prop-checks}\quad \frac{S \vdash T \surd}{K; S \vdash T \curvearrowright^{0}_{\mathsf{prop}} T}$$

$$\textsc{Prop-patch}\quad \frac{K; S \vdash e \Downarrow^{n} S'; T' \qquad T_1\{e : T'\} \overset{\mathsf{patch}}{\rightsquigarrow} T_2 \qquad K, T'; S \vdash T_2 \curvearrowright^{m}_{\mathsf{prop}} T_3}{K; S \vdash T_1 \curvearrowright^{n+m}_{\mathsf{prop}} T_3}$$

Figure 9: Incremental semantics of $\lambda^{\mathsf{cdd}}_{\mathsf{ic}}$: Reduction (to traces), propagating changes.

$\boxed{S \vdash T \surd}$ ***(Checking**: "Under store S, trace T checks as consistent.")*

$$\textsc{Check-term}\quad \frac{}{S \vdash \tilde{e} \surd}$$

$$\textsc{Check-seq}\quad \frac{S \vdash T_1 \surd \qquad S \vdash T_2 \surd}{S \vdash T_1 \cdot T_2 \surd}$$

$$\textsc{Check-force}\quad \frac{\mathsf{trm}(T) = \tilde{e} \qquad S \vdash T \surd}{S \vdash \mathsf{force}^{e}_{\tilde{e}}[T] \surd}$$

$$\textsc{Check-get}\quad \frac{S(a) = e \qquad S \vdash T \surd}{S \vdash \mathsf{get}^{a}_{e}[T] \surd}$$

$\boxed{T_1\{e : T_2\} \overset{\mathsf{patch}}{\rightsquigarrow} T_3}$ ***(Patching**: "Patching trace T_1 at forces of e with replacement trace T_2 yields trace T_3".)*

$$\textsc{Patch-term}\quad \frac{}{\tilde{e}\{e : T_2\} \overset{\mathsf{patch}}{\rightsquigarrow} \tilde{e}}$$

$$\textsc{Patch-seq}\quad \frac{T_1\{e : T_3\} \overset{\mathsf{patch}}{\rightsquigarrow} T_1' \qquad T_2\{e : T_3\} \overset{\mathsf{patch}}{\rightsquigarrow} T_2'}{(T_1 \cdot T_2)\{e : T_3\} \overset{\mathsf{patch}}{\rightsquigarrow} T_1' \cdot T_2'}$$

$$\textsc{Patch-update}\quad \frac{}{(\mathsf{force}^{e}_{\tilde{e}}[T_1])\{e : T_2\} \overset{\mathsf{patch}}{\rightsquigarrow} \mathsf{force}^{e}_{\tilde{e}}[T_2]}$$

$$\textsc{Patch-force}\quad \frac{e_1 \neq e_2 \qquad T_1\{e_2 : T_2\} \overset{\mathsf{patch}}{\rightsquigarrow} T_3}{(\mathsf{force}^{e_1}_{\tilde{e}}[T_1])\{e_2 : T_2\} \overset{\mathsf{patch}}{\rightsquigarrow} \mathsf{force}^{e_1}_{\tilde{e}}[T_3]}$$

$$\textsc{Patch-get}\quad \frac{T_1\{e_2 : T_2\} \overset{\mathsf{patch}}{\rightsquigarrow} T_1'}{(\mathsf{get}^{a}_{e_1}[T_1])\{e_2 : T_2\} \overset{\mathsf{patch}}{\rightsquigarrow} \mathsf{get}^{a}_{e_1}[T_1']}$$

Figure 10: Incremental semantics of $\lambda^{\mathsf{cdd}}_{\mathsf{ic}}$: Patching traces with substituted sub-traces, and checking traces for consistency.

$\boxed{S; e \vdash T \overset{\text{sched}}{\rightsquigarrow} p}$ ***(Schedule**: Under S and e, p is next in T.)*

$$\textsc{Sched-term}\quad \frac{}{S; e \vdash \tilde{e} \overset{\text{sched}}{\rightsquigarrow} \circ} \qquad \textsc{Sched-seq1}\quad \frac{S; e_1 \vdash T_1 \overset{\text{sched}}{\rightsquigarrow} e_2}{S; e_1 \vdash T_1 {\cdot} T_2 \overset{\text{sched}}{\rightsquigarrow} e_2} \qquad \textsc{Sched-seq2}\quad \frac{S; e \vdash T_1 \overset{\text{sched}}{\rightsquigarrow} \circ \quad S; e \vdash T_2 \overset{\text{sched}}{\rightsquigarrow} p}{S; e \vdash T_1 {\cdot} T_2 \overset{\text{sched}}{\rightsquigarrow} p}$$

$$\textsc{Sched-patchForce}\quad \frac{\mathsf{trm}(T) \neq \tilde{e}}{S; e_1 \vdash \mathsf{force}^{e_2}_{\tilde{e}}[T] \overset{\text{sched}}{\rightsquigarrow} e_1} \qquad \textsc{Sched-forceOk}\quad \frac{\mathsf{trm}(T) = \tilde{e} \quad S; e_2 \vdash T \overset{\text{sched}}{\rightsquigarrow} p}{S; e_1 \vdash \mathsf{force}^{e_2}_{\tilde{e}}[T] \overset{\text{sched}}{\rightsquigarrow} p}$$

$$\textsc{Sched-patchGet}\quad \frac{S(a) \neq e_2}{S; e_1 \vdash \mathsf{get}^{a}_{e_2}[T] \overset{\text{sched}}{\rightsquigarrow} e_1} \qquad \textsc{Sched-getOk}\quad \frac{S(a) = e_2 \quad S; e_1 \vdash T \overset{\text{sched}}{\rightsquigarrow} p}{S; e_1 \vdash \mathsf{get}^{a}_{e_2}[T] \overset{\text{sched}}{\rightsquigarrow} p}$$

$\boxed{K; S \vdash T_1 \curvearrowright^{n}_{\mathsf{alg}} T_2}$ ***(Algorithm**: determinisitically patches T_1 into T_2.)*

$$\textsc{Alg-done}\quad \frac{S; e \vdash T \overset{\text{sched}}{\rightsquigarrow} \circ}{K; S \vdash T \curvearrowright^{0}_{\mathsf{alg}} T} \qquad \textsc{Alg-patchStep}\quad \frac{\begin{array}{c} S; e_1 \vdash T_1 \overset{\text{sched}}{\rightsquigarrow} e_2 \\ K; S \vdash e_2 \Downarrow^{n} S; T' \\ (\mathsf{force}^{e_1}_{\tilde{e}_1}[T_1])\{e_2 : T'\} \overset{\text{patch}}{\rightsquigarrow} T_2 \\ K, T'; S \vdash T_2 \curvearrowright^{m}_{\mathsf{alg}} T_3 \end{array}}{K; S \vdash \mathsf{force}^{e_1}_{\tilde{e}_1}[T_1] \curvearrowright^{n+m}_{\mathsf{alg}} T_3}$$

Figure 11: A deterministic algorithm for patching.

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

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. Algorithmic patching

The incremental semantics given in the previous section is sound, but not yet an algorithm. In this section, we address one of two sources of nondeterminism in the semantics: the arbitrary choices for ordering patching steps in rule PROP-PATCH. In general, some orders are preferable to others: Some patching steps may fix inconsistencies in sub-traces that ultimately are not relevant under the current store. The algorithmic semantics given here addresses this problem by giving a deterministic order to patching, such that no unnecessary patching steps occur. The other source of nondeterminism in the semantics arises from rule INCR-FORCEPROP. The specification allows an arbitrary choice of trace to patch from prior knowledge; in general, many instances of a forced expression e may exist there, and it is difficult to know, a priori, which trace will be most efficient to patch under the current store. We address this problem in our implementation, given in Section 6.

Figure 11 defines $K; S \vdash T_1 \curvearrowright^{n}_{\mathsf{alg}} T_2$, the algorithmic propagation judgment, which employs an explicit, deterministic patching order. This order is determined by the *scheduling judgment* $S; e \vdash T \overset{\text{sched}}{\rightsquigarrow} p$, where e is the nearest enclosing think, and p is either some thunk e' or else is $\circ$ if trace T is consistent.

```
type 'a thunk
val force : 'a thunk → 'a
val update : 'a thunk → 'a → unit
val make_const : 'a → 'a thunk
val make_thunk : (unit → 'a) → 'a thunk
val memo : (module HashedType with type t = 'arg)
        → ('fn → 'arg → 'a) → (('arg → 'a thunk) as 'fn)
```

Figure 12: Basic ADAPTON API

$$\text{optional thunk} \quad p \quad ::= \quad \circ \mid e$$

The scheduling judgement chooses p based on the inconsistencies of the trace T and store S. Rule SCHED-TERM is the base case; rule SCHED-SEQ1 and rule SCHED-SEQ2 handle the sequencing cases where there is a thunk to patch is in the left sub-trace, or not, respectively. For forces and gets, the respective rules check that the force or get event is consistent. If so, rules SCHED-FORCEOK and SCHED-GETOK recursively check the trace enclosed within those events. rule SCHED-FORCEOK additionally places the thunk associated with the enclosed trace as e to the left of the turnstile, since that is the nearest enclosing thunk for that trace. Otherwise, if the force or get event is inconsistent, rules SCHED-PATCHFORCE and SCHED-PATCHGET schedule the nearest enclosing thunk to patch.

Algorithmic change propagation $K; S \vdash T_1 \curvearrowright^{n}_{\mathsf{alg}} T_2$ closely resembles the nondeterministic specification $K; S \vdash T_1 \curvearrowright^{n}_{\mathsf{prop}} T_2$ from Section 4. It gives rules for successively scheduling a thunk and patching the associated trace T_1 into T_2. We note that the root of the incoming and outgoing trace always consists of a force event (i.e., $\mathsf{force}^{e}_{\tilde{e}_1}[T_1]$); this invariant is preserved by the propagation algorithm, and its use in rule INCR-FORCEPROP, which is the only evaluation rule that uses this judgement as a premise. The rule ALG-PATCHSTEP does all the work of the algorithm: it schedules the next thunk e_2 to patch, recomputes this thunk yielding a new trace T', patches in the new trace for occurrences of e in the current trace ($\mathsf{force}^{e}_{\tilde{e}_1}[T_1]$), and repeats this process, which terminates with rule ALG-DONE. The premise to rule ALG-DONE is justified by the following lemma, which relates this base case to the specification semantics:

Lemma 5.1 (No scheduled thunk implies check)**.**
$S; e \vdash T \overset{\textit{sched}}{\rightsquigarrow} \circ$ *implies* $S \vdash T \surd$

The result below says that our algorithmic semantics is sound with respect to the specification semantics.

Theorem 5.2 (Algorithmic semantics is sound)**.**
If $K; S \vdash T_1 \curvearrowright^{n}_{\mathit{alg}} T_2$ *then* $K; S \vdash T_1 \curvearrowright^{n}_{\mathit{prop}} T_2$

6. ADAPTON: An OCaml Library for Incremental Computation

We have developed an implementation of CD^{2}IC in an OCaml library named ADAPTON. Incremental program are written using the LazyBidirectional module which provides the simple API shown in Figure 12. The fundamental data type is thunk, which subsumes the roles of both references $\mathsf{M}\ C$ and thunks $\mathsf{U}\ C$ in $\lambda^{\mathsf{cdd}}_{\mathsf{ic}}$. The force function retrieves the value of a thunk, computing or updating its value as necessary, similar to **get** or **force** in $\lambda^{\mathsf{cdd}}_{\mathsf{ic}}$. The update function replaces the value in a thunk, like **set** in $\lambda^{\mathsf{cdd}}_{\mathsf{ic}}$. The make_const and make_thunk functions construct thunks from a value or a nullary function, respectively. Finally, the memo function (whose use is discussed in more detail shortly) creates a memoized constructor from a unary function, given a HashedType module that describes the type of the argument (used internally to create memoization hash tables). The thunks constructed by make_thunk or by constructors created by memo do not contain a value at first. A thunk's value is computed by calling the given function when it is forced, and then the resulting value is cached in the thunk. These constructors are the counterparts to **thunk** in λ_{ic}^{cdd}.

We find it convenient to unify references and thunks in ADAPTON, since **get** and **force** as well as **ref** and **thunk** are symmetrical operations with the same extensional semantics. Furthermore, OCaml's type system does not allow us to easily enforce layer separation statically. Thus, we chose this unified API to make it simpler to program with ADAPTON. In ADAPTON, an inner level computation begins when force is called and ends when the call returns.

There is one semantic difference between λ_{ic}^{cdd} and LazyBidirectional: in λ_{ic}^{cdd}, memoization occurs at **force**, whereas in LazyBidirectional, memoization occurs when calling a memoized constructor created by memo. As an OCaml library, ADAPTON cannot compare two expressions for syntactic equality after variable substitution, unlike λ_{ic}^{cdd}. Instead, we use memo to manually identify free variables of an expression as function arguments, and use the values of those arguments for memoization, i.e., we represent thunk expressions as functions and free variables of the expression as function arguments. When a constructor created by memo is called, we first check a memoization table to see if the constructor was previously called with the same argument. If so, we return the same thunk as before; otherwise, we create a new thunk, store it in the memoization table, and return it. This design choice is equivalent to deterministically choosing the most recently patched occurrence of a trace from the prior knowledge in rule INCR-FORCEPROP of the incremental semantics. To limit the size of memoization tables, we implement them using weak hash tables, relying on OCaml's garbage collector to eventually remove thunks that are no longer reachable.

In addition to LazyBidirectional, ADAPTON also provides EagerTotalOrder, which implements a totally ordered, monolithic form of incremental computation as described in prior work (in particular, [3]). There are also two modules, EagerNonInc and LazyNonInc, which are simply wrappers around regular and lazy values, respectively, that do not provide incremental semantics or memoization. All four modules implement the same API in Figure 12 to make them easily interchangeable; thus it is straightforward to compare the same program under different evaluation and incremental semantics using ADAPTON.

6.1 LazyBidirectional **Implementation**

The LazyBidirectional module implements λ_{ic}^{cdd} using an efficient graph-based representation to realize the algorithmic patching semantics in Section 5. The core underlying data structure of LazyBidirectional is the acyclic *demanded computation graph* (DCG), corresponding to traces T. Similar to the visualization in Section 2, each node in the graph represents a thunk, and each directed edge represents a dependency pointing from the thunk calling force to the forced thunk.

The graph is initially empty at the beginning of the execution of an incremental program, and is built and maintained dynamically as the program executes. Nodes are added when make_const or make_thunk is called. Nodes are also added when a memo constructor is called and a new thunk is created, i.e., when memoization misses. Edges are added when force is called, and are labeled with the value returned by that call. We maintain edges bidirectionally: each node stores both an ordered list of outgoing edges that is appended by each call to force, and an unordered set of incoming edges. This allows us to traverse the graph from caller to callee or vice-versa.

As described in Section 2, LazyBidirectional takes advantage of the bidirectional nature of the graph in the two phases. The *dirtying phase* occurs when we update the inputs to the incremental program, i.e., when we make (consecutive) calls to update. In this phase, we record calls to update in the graph by starting from the updated thunk and traversing incoming edges backward through calling thunks up to thunks with no incoming edges, marking all traversed edges as "dirty." These dirty edges indicate intermediate values in the computation that may potentially change because of the updated inputs, i.e., they induce a sub-graph of thunks that may need to be re-evaluated.

The *propagate phase* performs D^2CP, beginning with a call force on a thunk that contains dirty outgoing edges, i.e., that may be potentially affected by an updated input. Then, we perform patching using a generalized inorder traversal of dirty outgoing edges, re-evaluating thunks if necessary. We check if we need to re-evaluate the forced thunk by traversing its dirty outgoing edges in the order they were added: for each edge, we first clean its dirty flag, then check if the target thunk contains dirty outgoing edges. If so, we recursively check if we need to re-evaluate the target thunk; otherwise, we compare the value of the target thunk against the label of the outgoing edge. If the value is inconsistent, then we know that at least one of its inputs has changed, so we skip its remaining outgoing edges and immediately re-evaluate the thunk. Before doing so, we first remove all its outgoing edges, since some of the edges may no longer be relevant due to the changed input; relevant edges will be re-added when the re-evaluation of the thunk calls force (we store incoming edges in a weak hash table, relying on OCaml's garbage collector to remove irrelevant edges). If all the values of outgoing edges are consistent, we need not re-evaluate the thunk since no inputs have changed.

The above procedure checks and re-evaluates thunks in the same order as described in Section 5, but in an optimized manner. First, the above procedure immediately re-evaluates thunks as necessary while traversing the graph, whereas the formalism schedules thunks to patch by repeatedly restarting the traversal from the initially forced thunk. Second, the above procedure only traverses the sub-graph induced by dirty edges, which can be much smaller than the entire graph if the number of updated inputs is small.

One possible concern is the cost of the dirtying phase. However, we observe that above procedure maintains an invariant that, if an edge is dirty at the end of a dirtying or propagate phase, then all edges transitively reachable by traversing incoming edges beginning from the source thunk will also be dirty. Thus, we need not continue the dirtying phase past dirty edges, in effect amortizing the dirtying cost across consecutive calls to update. Dually, if an edge is clean, then all edges transitively reachable by outgoing edges beginning from the target thunk will also be clean, which amortizes change propagation cost across consecutive calls to force.

7. Empirical Evaluation

We ran micro-benchmarks to evaluate the effectiveness of λ_{ic}^{cdd} in handling several different interactive computing patterns:

- *lazy*, which demands only a small portion of the output (just one item), and makes only simple changes to the input (e.g., adding or removing a list item or a tree node);
- *batch*, which demands the entire output, and makes only simple changes as with *lazy*;
- *swap*, which also demands the entire output, but changes the input by swapping two portions of the input (e.g., swapping two halves of a list or two subtrees);
- and *switch*, which chooses between two computations to apply to a main input based on another flag input, then demands a small portion of the output (just one item), and toggles the flag input while making simple changes to the main input.

	pattern	input #	demand #	EagerNonInc time (s)	EagerNonInc mem (MB)	LazyNonInc time (s)	LazyNonInc mem (MB)	LazyBidirectional from-scratch overhead Eager	LazyBidirectional from-scratch overhead Lazy	LazyBidirectional from-scratch mem (MB)	LazyBidirectional incremental speed-up Eager	LazyBidirectional incremental speed-up Lazy	LazyBidirectional incremental mem (MB)	EagerTotalOrder from-scratch overhead Eager	EagerTotalOrder from-scratch overhead Lazy	EagerTotalOrder from-scratch mem (MB)	EagerTotalOrder incremental speed-up Eager	EagerTotalOrder incremental speed-up Lazy	EagerTotalOrder incremental mem (MB)
filter	lazy	1e6	1	0.863	184	0.0000116	96.7	0.000035	2.63	264	951000	12.8	264	12.8	953000	773	166000	2.24	773
map		1e6		1.35	184	0.00000685	96.7	0.0000184	3.61	264	1540000	7.80	264	8.54	1680000	773	301000	1.53	773
quicksort		1e5		0.793	35.0	0.0741	18.6	2.23	23.8	161	21600	2020	162	53.3	570	2610	245	22.9	2680
mergesort		1e5		1.04	49.2	0.346	50.8	5.72	17.2	380	1010	336	395	20.1	60.3	1390	0.443	0.148	4900
filter	batch	1e6	all	1.05	155	0.629	157	12.4	20.7	915	3.39	2.04	1580	11.5	19.1	773	6.84	4.11	1410
map		1e6		1.79	232	1.20	232	7.22	10.8	934	3.31	2.21	1600	7.35	11.0	806	4.97	3.32	1540
fold(min)		1e6		2.43	240	1.43	179	18.3	31.2	1610	7400	4350	1610	9.08	15.5	1410	5260	3090	1440
fold(sum)		1e6		2.44	255	1.48	180	18.4	30.5	1630	2700	1640	1630	9.22	15.2	1440	6970	4220	1440
exptree		1e6		0.143	153	0.308	152	613	285	1780	116	248	1780	129	60.1	1480	347	746	1480
filter	swap	1e6	all	0.868	155	0.502	157	12.0	20.7	913	3.43	1.99	1580	11.0	19.1	773	0.247	0.143	2710
map		1e6		1.42	232	0.894	232	6.63	10.5	934	3.75	2.36	1600	7.35	11.7	806	0.394	0.248	2900
fold(min)		1e6		1.92	240	1.04	179	19.6	36.3	1610	872	472	1620	9.49	17.5	1420	0.226	0.123	6070
fold(sum)		1e6		1.97	255	1.11	180	19.1	34.0	1630	888	501	1640	9.60	17.0	1440	0.228	0.128	6090
exptree		1e6		0.145	153	0.307	152	590	278	1780	315	667	1780	128	60.6	1480	4.78	10.1	1810
updown1	switch	4e4	1	0.198	15.2	0.0328	8.63	3.60	21.8	73.4	135	22.4	121	70.9	429	987	0.015	0.00247	3710
updown2		4e4		0.409	19.3	0.0326	8.63	1.74	21.9	73.4	309	24.7	119	72.9	915	1930	53.7	4.28	2120

Legend overhead: *NonInc time / from-scratch time speed-up: *NonInc time / incremental time mem: OCaml's GC maximum heap size

Table 1: ADAPTON micro-benchmark results.

The *lazy*, *swap*, and *switch* patterns are some of the patterns described in Section 2 that motivated our work, whereas prior work only addressed the *batch* pattern.

We implement each micro-benchmark by writing several incremental programs using each ADAPTON module, and measure the time and memory these programs take to run either from scratch or incrementally. For *lazy*, we include standard list-manipulating benchmark programs from the incremental computing literature—filter, map, quicksort, and mergesort—and demand only one item from the output list. For *batch*, we demand the entire output of filter and map as well as the two other list programs—fold applying min and sum—and exptree, an arithmetic expression tree evaluator similar to that in Section 2. For *swap*, we use the same programs as *batch*. Finally, for *switch*, we write two programs, updown1 and updown2, both returning a list sorted in either ascending or descending order depending on the value of another input: updown1 is the straightforward implementation that sorts the input list in one direction or the other, whereas updown2 first sorts the input list in both directions, then returns the appropriate one.

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 much more memory under certain ADAPTON modules. We report the average of 8 runs for each program using random seeds 1–8, and each run consists of 250 change-then-propagate cycles or 500 paired cycles (for the list programs, removing then re-inserting an item; for updown1 and updown2, sorting in each direction).

In our initial evaluation, we observed that EagerTotalOrder spends a significant portion of time in the garbage collector, well over half the time for some programs. This issue that has been reported in prior work [4]. To mitigate this issue, we tweak OCaml's garbage collector to use a minor heap size of 16MB and major heap increment of 32MB for EagerTotalOrder.

7.1 Micro-benchmark Results

Table 1 shows our results. For EagerNonInc and LazyNonInc, we report the wall-clock time and maximum heap size, as reported by OCaml's garbage collector, that it takes to run the program. For LazyBidirectional and EagerTotalOrder, instead of reporting wall-clock time, we report *overhead*, the time it takes to run the initial *from-scratch* computation relative to EagerNonInc and LazyNonInc, and *speed-up*, the time it takes to run each *incremental* change-then-propagate cycle, also relative to EagerNonInc and LazyNonInc. We also highlight table cells in gray to indicate whether LazyBidirectional or EagerTotalOrder has a higher speed-up or uses less memory when performing an incremental computation.

We can see that LazyBidirectional is faster and uses less memory than EagerTotalOrder for the *lazy*, *swap*, and *switch* patterns, which are some of the patterns that motivated this work. As a sanity check for *lazy*, we note that LazyBidirectional is over a million times faster than EagerNonInc for map since only one out of a million input item needs to be processed, and similarly for filter. It is also quite effective for quicksort and mergesort. Note that mergesort actually incurs a slowdown under EagerTotalOrder, and also under LazyBidirectional if more output elements were demanded. This is due to limited memoization between each internal recursion in mergesort. Prior work required programmers to manually modify mergesort to use techniques such as *adaptive memoization* [4] or *keyed allocation* [16] to improve its incremental performance. We are currently looking into an alternative technique for λ_{ic}^{cdd} that employs the concept of *functional dependencies* [8] from databases to systematically identify better memoization opportunities.

For the *batch* pattern, LazyBidirectional does not perform as well as EagerTotalOrder—at about half the speed. We expected this to be the case since EagerTotalOrder is optimized for the *batch* pattern, and because LazyBidirectional involves a dirtying phase in addition to a change propagation phase to perform an incremental computation (as described in Section 6.1), rather than just a change propagation phase in EagerTotalOrder. In the *batch* pattern, the dirtying phase becomes an unnecessary cost as all outputs are demanded. Nonetheless, for all programs LazyBidirectional provides a speed-up that can still be quite significant in some cases. Interestingly, LazyBidirectional is faster for fold(min), since changes are not as likely to affect the result of the min operation as compared to other operations such as sum.

LazyBidirectional is much better behaved than EagerTotalOrder in that LazyBidirectional provides a speed-up to all patterns and programs. In contrast, EagerTotalOrder actually incurs slowdowns in

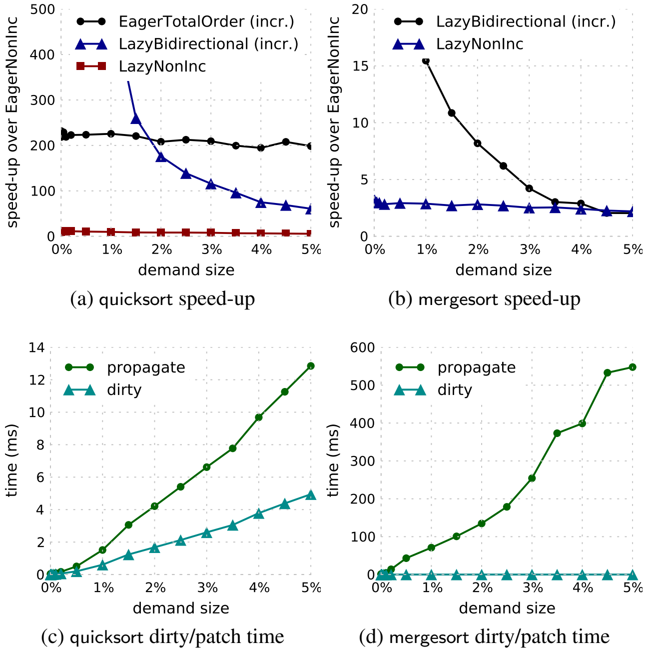

Figure 13: Speed-up and dirtying/patching time over varying demand sizes for input size of 100,000

swap and *switch*, except for exptree and updown2. Due to its underlying total ordering assumption, EagerTotalOrder can only memo-match about half the input on average for changes considered by *swap*. It 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 and uses twice as much memory. In contrast, LazyBidirectional 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 almost no cost to unconditionally sort the input list twice in updown2.

7.2 Evaluating performances over varying demand

LazyBidirectional performs best for small demand size, but the cost of D^2CP gradually becomes more significant as demand size increases. We quantify this effect for quicksort and mergesort using the same procedure as for the *lazy* pattern in the previous section, measuring the impact of increasing demand. We use EagerNonInc as the baseline and vary the demand size from one element to 5% of the output. For comparison, we also measure the speed-up of EagerTotalOrder as well as the speed-up of LazyNonInc over EagerNonInc. Since LazyBidirectional contains two different phases that update the DCG—a dirtying phase when the input list is updated and a propagating phase to repair inconsistencies—we additionally measure the time spent in each phase to understand their relative costs.

The results are shown in Figure 13. In Figure 13a we see that the speed-up of LazyBidirectional decreases as demand size increases, whereas the speed-up of EagerTotalOrder is relatively constant across demand size (though there is a minor cost to take each additional element from the output list). LazyBidirectional becomes slower than EagerTotalOrder when demanding more than about 1.8% of the output. However, the speed-up of LazyBidirectional remains greater than both EagerNonInc and LazyNonInc even when demanding 5% of the output, thus, there is still an advantage to using LazyBidirectional, if not as much as EagerTotalOrder.

The speed-up for mergesort is lower but still significant, as shown in Figure 13b. We omit EagerTotalOrder from this plot because it incurs a slowdown, as explained in Section 7.1. The plot shows that LazyBidirectional becomes slower than LazyNonInc when demanding more than 4% of the output.

Figures 13c and 13d shows how much time is spent in the dirtying and propagation phases of LazyBidirectional. As expected, propagation time increases with demand size, as more computation has to be performed to compute the output. Also, dirtying is less costly than propagation, since it does not perform any computation on thunks; dirtying is significantly less costly for mergesort relative to propagation as more thunks are re-evaluated than in quicksort. Interestingly, however, dirtying time increases with demand size. This is due to the interaction between the amortization of the dirtying and propagation phases described at the end of Section 6.1. For a set of input changes, each consecutive change has to dirty fewer edges as more edges become dirty in the graph. However, as demand size increases, more dirty edges will be cleaned by propagation, resulting in more dirtying work for the next set of input changes.

8. Related Work

Incremental computation via memoization. Memoization, also called *function caching* [1, 20, 26, 29], improves the efficiency of any purely functional program wherein functions are applied repeatedly to equivalent arguments. In fact, this idea dates back to at least the late 1950's [9, 27, 28]. Self-adjusting computation, discussed below, uses a special form of memoization that caches and reuses portions of *dynamic dependency graphs* of a computation, as opposed to simply caching their final results.

Our memoization technique is related to that of self-adjusting computation, in that CD^2IC uses memoization to cache dependency graphs. As in self-adjusting computation, and unlike earlier purely-functional memoization approaches, CD^2IC 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.

Incremental computation via dependence graphs. Incremental computation has been studied by programming language researchers for decades; we refer the reader to a categorized bibliography of early work [30]. Most techniques maintain some representation of data dependencies as graphs. Self-adjusting computation adapts the dependence graphs of earlier techniques, introducing *dynamic dependence graphs* (DDGs), which are generated from conventional-looking programs with general recursion and fine-grained data dependencies [2, 11]. Later, researchers combined these dynamic graphs with a special form of memoization, making the approach even more efficient and broadly applicable [3]. More recently, researchers have studied ways to make self-adjusting programs easier to write and reason about [12, 13, 24], as well as more performant, empirically [18, 19].

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 handled poorly or not at all in prior work. These patterns consist of the following: *sharing*, where distinct inner computations share dependency graph components; *switching*, where outer layer demand drives what computations are incrementally updated; and *swapping*, where the inputs and/or computation steps interchange their position, relative to some prior demand. Past work based on maintaining a single totally-ordered view of past computation (e.g., all work on self-adjusting computation) simply cannot handle these patterns.

Ley-Wild et al. have recently studied non-monotonic changes (viz., what we call "swapping") and lazy evaluation, giving a formal semantics and algorithmic designs [23, 25]. 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. To our knowledge, these extensions have no corresponding implementations.

Functional reactive programming. The chief aim of FRP is to provide a declarative means of specifying interactive and/or time-varying behavior. Some FRP-based proposals share some commonalities with incremental computation (e.g., [14, 21]). By virtue of its declarative nature, FRP makes incremental change implicit, rather than explicit: it hides mutation and incremental change beneath abstractions for *streams*, which capture time-varying data.

By contrast, CD^{2}IC explicitly exposes the inner/outer dichotomy, and allows the outer layer to perform arbitrary store mutations. Unlike incremental computation broadly, FRP is not chiefly concerned with asymptotic trends or fine-grained incremental dependencies. Interesting future work may involve investigating how FRP applications can benefit from CD^{2}IC techniques, and how CD^{2}IC can benefit from the higher-level abstractions proposed by researchers studying FRP.

9. 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 on 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.

Acknowledgments

This research was supported in part by NSF CCF-0915978.

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, and R. Harper. Adaptive functional programming. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, pages 247–259, 2002.

[3] 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.

[4] 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.

[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] W. W. Armstrong. Dependency structures of data base relationships. In *IFIP Congress*, pages 580–583, 1974.

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

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

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

[12] 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.

[13] 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. to appear.

[14] 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.

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

[24] 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.

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

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

[27] 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.

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

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

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

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

A. From CBV into CBPV

This section presents a call-by-value (CBV) variant of $\lambda_{\mathsf{ic}}^{\mathsf{cdd}}$ that shares a close correspondance to the CBPV variant presented in the main body of the paper. In particular, we present syntax, typing and reduction rules in a CBV style, which are otherwise analogous to the CBPV presented in Section 3. We connect the CBV variant presented here to the CBPV variant of $\lambda_{\mathsf{ic}}^{\mathsf{cdd}}$ via a type-directed translation. We show that this translation is preserved by the basic, non-incremental reduction semantics of both variants: If a CBV term M translates to a CBPV e, and if M reduces to a value V, then e reduces to a CBPV terminal $\tilde{e}$ to which value V translates.

CBV syntax. Unlike CBPV syntax, CBV syntax does not syntactically separate values and expressions; rather, values are a syntactic subclass of expressions. Second, CBV treats lambdas as values. Finally, since the CBV type syntax is missing an analogue to $\mathsf{F}_\ell\ A$, which carries a layer annotation in $\lambda_{\mathsf{ic}}^{\mathsf{cdd}}$, we instead affix this layer annotation to the type connectives arrow ($\tau_1 \xrightarrow{\ell} \tau_2$), thunk ($\mathsf{U}^{\ell}\ \tau_1$) and reference cells ($\mathsf{M}^{\ell}\ \tau_1$). The other differences to the syntax and typing rules reflect these key distinctions:

$$
\begin{array}{llll}
\text{CBV values} & \mathsf{V} & ::= & () \mid \lambda x.\mathsf{M} \mid \mathbf{inj}_i\ \mathsf{V} \mid (\mathsf{V}_1,\mathsf{V}_2) \mid \mathbf{thunk}\ \mathsf{M} \\
& & & \mid a \\
\text{CBV terms} & \mathsf{M} & ::= & \mathsf{V} \mid x \mid \mathsf{M}_1\,\mathsf{M}_2 \mid \mathbf{let}\,x \leftarrow \mathsf{M}_1\,\mathbf{in}\,\mathsf{M}_2 \\
& & & \mid \mathbf{fix}\,f.\mathsf{M} \mid \mathbf{inj}_i\ \mathsf{M} \\
& & & \mid \mathbf{case}\,(\mathsf{M}, x_1.\mathsf{M}_1, x_2.\mathsf{M}_2) \mid (\mathsf{M}_1,\mathsf{M}_2) \\
& & & \mid \mathbf{fst}\,\mathsf{M} \mid \mathbf{snd}\,\mathsf{M} \mid \mathbf{force}_{\ell}\ \mathsf{M} \mid \mathbf{inner}\ \mathsf{M} \\
& & & \mid \mathbf{ref}\,\mathsf{M} \mid \mathbf{get}\,\mathsf{M} \mid \mathbf{set}\,\mathsf{M}_1 \leftarrow \mathsf{M}_2 \\
\text{CBV types} & \tau & ::= & 1 \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 \xrightarrow{\ell} \tau_2 \mid \mathsf{U}^{\ell}\ \tau \\
& & & \mid \mathsf{M}^{\ell}\ \tau \\
\text{CBV typing env.} & \mathsf{G} & ::= & \varepsilon \mid \mathsf{G}, x:\tau \mid \mathsf{G}, f:\tau \mid \mathsf{G}, a:\tau \\
\text{CBV store} & \dot{\mathsf{S}} & ::= & \varepsilon \mid \dot{\mathsf{S}}, a{:}\mathsf{M}
\end{array}
$$

CBV typing. Figure 14 gives the judgement for typing CBV terms under a typing context G and layer ℓ.

CBV big-step semantics. Figure 15 gives the judgement for big-step evaluation of CBV terms under a store $\dot{\mathsf{S}}$.

CBV type-directed translation. Figure 16 gives the judgement for translating CBV types into corresponding CBPV types. Figure 17 gives the judgement for translating CBV terms into corresponding CBPV value terms. Figure 18 gives the judgement for translating CBV terms into corresponding CBPV value terms. Figure 19 gives the judgement for translating CBV stores into corresponding CBPV stores.

Meta theory. We show several simple results:

Theorem A.1 (CBV subject reduction)**.**
Suppose that:

- $\mathsf{G}_1 \vdash \dot{\mathsf{S}}_1$
- $\mathsf{G}_1 \vdash_\ell \mathsf{M} : \tau$
- $\dot{\mathsf{S}}_1 \vdash \mathsf{M} \Downarrow \dot{\mathsf{S}}_2; \mathsf{V}$

then there exists G_2 *such that* $\mathsf{G}_1 \vdash \mathsf{G}_2$ *and*

- $\mathsf{G}_2 \vdash \dot{\mathsf{S}}_2$
- $\mathsf{G}_2 \vdash_\ell \mathsf{V} : \tau$

Theorem A.2 (CBV typing implies CBPV translation)**.**

- *If* $\mathsf{G} \vdash_\ell \mathsf{M} : \tau$ *then* $\mathsf{G} \vdash_\ell \mathsf{M} : \tau \overset{comp}{\rightsquigarrow} \Gamma \vdash e : C$
- *If* $\mathsf{G} \vdash_\ell \mathsf{V} : \tau$ *then* $\mathsf{G} \vdash \mathsf{V} : \tau \overset{val}{\rightsquigarrow} (\Gamma \vdash v : A)$

Theorem A.3 (CBPV translation and CBV reduction commute)**.**
Suppose that:

- $\mathsf{G} \vdash \dot{\mathsf{S}}_1 \rightsquigarrow \Gamma \vdash S_1$
- $\mathsf{G} \vdash_\ell \mathsf{M}_1 : \tau \overset{comp}{\rightsquigarrow} \Gamma \vdash e : C$
- $\dot{\mathsf{S}}_1 \vdash \mathsf{M}_1 \Downarrow \dot{\mathsf{S}}_2; \mathsf{V}$

then there exists extended contexts Γ' *and* G' *with* $\mathsf{G} \vdash \mathsf{G}'$ *and* $\Gamma \vdash \Gamma'$ *such that:*

- $\mathsf{G}' \vdash \dot{\mathsf{S}}_2 \rightsquigarrow \Gamma' \vdash S_2$
- $\mathsf{G}' \vdash_\ell \mathsf{V} : \tau \overset{comp}{\rightsquigarrow} \Gamma' \vdash \tilde{e} : C$
- $S_1 \vdash e \Downarrow^n S_2; \tilde{e}$

$\boxed{G \vdash \dot{S}}$ *(Under G, the store $\dot{S}$ is well-typed)*

CBVTyS-EMP
$$\frac{}{G \vdash \varepsilon}$$

CBVTyS-ADDR
$$\frac{G \vdash \dot{S} \qquad G(a) = M^{\ell}\,\tau \qquad G \vdash_{\ell} M : \tau}{G \vdash \dot{S}, a{:}M}$$

$\boxed{G \vdash_{\ell} M : \tau}$ *(Under G at ℓ, the term M has type τ)*

CBVTy-VAR
$$\frac{G(x) = \tau}{G \vdash_{\ell} x : \tau}$$

CBVTy-UNIT
$$\frac{}{G \vdash_{\ell} () : 1}$$

CBVTy-ABS
$$\frac{G, x : \tau_1 \vdash_{\ell_2} M : \tau_2}{G \vdash_{\ell_1} \lambda x.M : \tau_1 \xrightarrow{\ell_2} \tau_2}$$

CBVTy-APP
$$\frac{G \vdash_{\ell} M_1 : \tau_1 \xrightarrow{\ell} \tau_2 \qquad G \vdash_{\ell} M_2 : \tau_1}{G \vdash_{\ell} M_1\, M_2 : \tau_2}$$

CBVTy-LET
$$\frac{G \vdash_{\ell} M_1 : \tau_1 \qquad G, x : \tau_1 \vdash_{\ell} M_2 : \tau_2}{G \vdash_{\ell} \textbf{let}\, x \leftarrow M_1 \,\textbf{in}\, M_2 : \tau_2}$$

CBVTy-FIX
$$\frac{G, f : \tau \vdash_{\ell} M : \tau}{G \vdash_{\ell} \textbf{fix}\, f.M : \tau}$$

CBVTy-INJ
$$\frac{\textbf{exists}\, i \,\textbf{in}\, \{1,2\} \qquad G \vdash_{\ell} M : \tau_i}{G \vdash_{\ell} \textbf{inj}_i\, M : \tau_1 + \tau_2}$$

CBVTy-CASE
$$\frac{G \vdash_{\ell} M : \tau_1 + \tau_2 \qquad G, x_1 : \tau_1 \vdash_{\ell} M_1 : \tau_3 \qquad G, x_2 : \tau_2 \vdash_{\ell} M_2 : \tau_3}{G \vdash_{\ell} \textbf{case}\,(M, x_1.M_1, x_2.M_2) : \tau_3}$$

CBVTy-PAIR
$$\frac{G \vdash_{\ell} M_1 : \tau_1 \qquad G \vdash_{\ell} M_2 : \tau_2}{G \vdash_{\ell} (M_1, M_2) : \tau_1 \times \tau_2}$$

CBVTy-FST
$$\frac{G \vdash_{\ell} M : \tau_1 \times \tau_2}{G \vdash_{\ell} \textbf{fst}\, M : \tau_1}$$

CBVTy-SND
$$\frac{G \vdash_{\ell} M : \tau_1 \times \tau_2}{G \vdash_{\ell} \textbf{snd}\, M : \tau_2}$$

CBVTy-THUNK
$$\frac{G \vdash_{\ell_2} M : \tau}{G \vdash_{\ell_1} \textbf{thunk}\, M : U^{\ell_2}\,\tau}$$

CBVTy-FORCE
$$\frac{G \vdash_{\ell} M : U^{\ell}\,\tau}{G \vdash_{\ell} \textbf{force}_{\ell}\, M : \tau}$$

CBVTy-INNER
$$\frac{|G| \vdash_{\mathsf{inner}} M : \tau}{G \vdash_{\mathsf{outer}} \textbf{inner}\, M : \tau}$$

CBVTy-ADDR
$$\frac{G(a) = M^{\ell_2}\,\tau}{G \vdash_{\ell_1} a : M^{\ell_2}\,\tau}$$

CBVTy-REF
$$\frac{G \vdash_{\ell} M : \tau}{G \vdash_{\mathsf{outer}} \textbf{ref}\, M : M^{\ell}\,\tau}$$

CBVTy-GET
$$\frac{G \vdash_{\ell} M : M^{\ell}\,\tau}{G \vdash_{\ell} \textbf{get}\, M : \tau}$$

CBVTy-SET
$$\frac{G \vdash_{\mathsf{outer}} M_1 : M^{\ell_2}\,\tau \qquad G \vdash_{\mathsf{outer}} M_2 : U^{\ell_2}\,\tau}{G \vdash_{\mathsf{outer}} \textbf{set}\, M_1 {\leftarrow} M_2 : 1}$$

Figure 14: CBV typing semantics.

$\boxed{\dot{S}_1 \vdash M \Downarrow \dot{S}_2; V}$ *(Under $\dot{S}_1$, the term M reduces, yielding store $\dot{S}_2$ and value V)*

CBVEval-VAL
$$\frac{}{\dot{S} \vdash V \Downarrow \dot{S}; V}$$

CBVEval-APP
$$\frac{\dot{S}_1 \vdash M_1 \Downarrow \dot{S}_2; \lambda x.M \qquad \dot{S}_3 \vdash M_2 \Downarrow \dot{S}_4; V_2 \qquad \dot{S}_5 \vdash M[V_2/x] \Downarrow \dot{S}_6; V_3}{\dot{S}_1 \vdash M_1\, M_2 \Downarrow \dot{S}_7; V_3}$$

CBVEval-LET
$$\frac{\dot{S}_1 \vdash M_1 \Downarrow \dot{S}_2; V_1 \qquad \dot{S}_2 \vdash M_2[V_1/x] \Downarrow \dot{S}_3; V_2}{\dot{S}_1 \vdash \textbf{let}\, x \leftarrow M_1 \,\textbf{in}\, M_2 \Downarrow \dot{S}_3; V_2}$$

CBVEval-FIX
$$\frac{\dot{S}_1 \vdash M[\textbf{fix}\, f.M/f] \Downarrow \dot{S}_2; V}{\dot{S}_1 \vdash \textbf{fix}\, f.M \Downarrow \dot{S}_2; V}$$

CBVEval-INJ
$$\frac{\dot{S}_1 \vdash M \Downarrow \dot{S}_2; V}{\dot{S}_1 \vdash \textbf{inj}_i\, M \Downarrow \dot{S}_2; \textbf{inj}_i\, V}$$

CBVEval-CASE
$$\frac{\textbf{exists}\, i \,\textbf{in}\, \{1,2\} \qquad \dot{S}_1 \vdash M \Downarrow \dot{S}_2; \textbf{inj}_i\, V_1 \qquad \dot{S}_2 \vdash M_i[V_1/x_i] \Downarrow \dot{S}_3; V_2}{\dot{S}_1 \vdash \textbf{case}\,(M, x_1.M_1, x_2.M_2) \Downarrow \dot{S}_3; V_2}$$

CBVEval-PAIR
$$\frac{\dot{S}_1 \vdash M_1 \Downarrow \dot{S}_2; V_1 \qquad \dot{S}_2 \vdash M_2 \Downarrow \dot{S}_3; V_2}{\dot{S}_1 \vdash (M_1, M_2) \Downarrow \dot{S}_3; (V_1, V_2)}$$

CBVEval-FST
$$\frac{\dot{S}_1 \vdash M \Downarrow \dot{S}_2; (V_1, V_2)}{\dot{S}_1 \vdash \textbf{fst}\, M \Downarrow \dot{S}_2; V_1}$$

CBVEval-SND
$$\frac{\dot{S}_1 \vdash M \Downarrow \dot{S}_2; (V_1, V_2)}{\dot{S}_1 \vdash \textbf{snd}\, M \Downarrow \dot{S}_2; V_2}$$

CBVEval-INNER
$$\frac{\dot{S}_1 \vdash M \Downarrow \dot{S}_2; V}{\dot{S}_1 \vdash \textbf{inner}\, M \Downarrow \dot{S}_2; V}$$

CBVEval-FORCE
$$\frac{\dot{S}_1 \vdash M \Downarrow \dot{S}_2; \textbf{thunk}\, M' \qquad \dot{S}_2 \vdash M' \Downarrow \dot{S}_3; V}{\dot{S}_1 \vdash \textbf{force}_{\ell}\, M \Downarrow \dot{S}_3; V}$$

CBVEval-REF
$$\frac{a \notin \mathsf{dom}(\dot{S})}{\dot{S} \vdash \textbf{ref}\, M \Downarrow \dot{S}, a{:}M; a}$$

CBVEval-GET
$$\frac{\dot{S}_1 \vdash M \Downarrow \dot{S}_2; a \qquad \dot{S}_2(a) = M' \qquad \dot{S}_2 \vdash M' \Downarrow \dot{S}_3; V}{\dot{S}_1 \vdash \textbf{get}\, M \Downarrow \dot{S}_2; V}$$

CBVEval-SET
$$\frac{\dot{S}_1 \vdash M_1 \Downarrow \dot{S}_2; a \qquad \dot{S}_2 \vdash M_2 \Downarrow \dot{S}_3; \textbf{thunk}\, M}{\dot{S}_1 \vdash \textbf{set}\, M_1 {\leftarrow} M_2 \Downarrow \dot{S}_3, a{:}M; ()}$$

Figure 15: CBV big-step evaluation semantics.

$\boxed{\mathsf{G} \overset{\text{ctxt}}{\rightsquigarrow} \Gamma}$ *(The CBV typing context* G *translates to the CBPV typing context* Γ*)*

$$\text{CBV-TrCtxt-emp}\quad \frac{}{\varepsilon \overset{\text{ctxt}}{\rightsquigarrow} \varepsilon}$$

$$\text{CBV-TrCtxt-varVal}\quad \frac{\mathsf{G} \overset{\text{ctxt}}{\rightsquigarrow} \Gamma \qquad \tau \overset{\text{val}}{\rightsquigarrow} A}{\mathsf{G}, x : \tau \overset{\text{ctxt}}{\rightsquigarrow} \Gamma, x : A}$$

$$\text{CBV-TrCtxt-varFix}\quad \frac{\mathsf{G} \overset{\text{ctxt}}{\rightsquigarrow} \Gamma \qquad \tau \overset{\text{comp}}{\rightsquigarrow} C}{\mathsf{G}, f : \tau \overset{\text{ctxt}}{\rightsquigarrow} \Gamma, f : C}$$

$$\text{CBV-TrCtxt-addr}\quad \frac{\mathsf{G} \overset{\text{ctxt}}{\rightsquigarrow} \Gamma \qquad \tau \overset{\text{comp}}{\rightsquigarrow} (C)^{\ell}}{\mathsf{G}, a : \mathsf{M}^{\ell}\,\tau \overset{\text{ctxt}}{\rightsquigarrow} \Gamma, a : (C)^{\ell}}$$

$\boxed{\tau \overset{\text{val}}{\rightsquigarrow} A}$ *(The CBV type* τ *translates to the CBPV value type* A*)*

$$\text{CBV-TrValTy-unit}\quad \frac{}{1 \overset{\text{val}}{\rightsquigarrow} 1}$$

$$\text{CBV-TrValTy-thunk}\quad \frac{\tau \overset{\text{comp}}{\rightsquigarrow} (C)^{\ell}}{\mathsf{U}^{\ell}\,\tau \overset{\text{val}}{\rightsquigarrow} \mathsf{U}\,C}$$

$$\text{CBV-TrValTy-ref}\quad \frac{\tau \overset{\text{comp}}{\rightsquigarrow} (C)^{\ell}}{\mathsf{M}^{\ell}\,\tau \overset{\text{val}}{\rightsquigarrow} \mathsf{M}\,C}$$

$$\text{CBV-TrValTy-sum}\quad \frac{\tau_1 \overset{\text{val}}{\rightsquigarrow} A_1 \qquad \tau_2 \overset{\text{val}}{\rightsquigarrow} A_2}{\tau_1 + \tau_2 \overset{\text{val}}{\rightsquigarrow} A_1 + A_2}$$

$$\text{CBV-TrValTy-prod}\quad \frac{\tau_1 \overset{\text{val}}{\rightsquigarrow} A_1 \qquad \tau_2 \overset{\text{val}}{\rightsquigarrow} A_2}{\tau_1 \times \tau_2 \overset{\text{val}}{\rightsquigarrow} A_1 \times A_2}$$

$$\text{CBV-TrValTy-arrow}\quad \frac{\tau_1 \overset{\ell}{\rightarrow} \tau_2 \overset{\text{comp}}{\rightsquigarrow} (C)^{\ell}}{\tau_1 \overset{\ell}{\rightarrow} \tau_2 \overset{\text{val}}{\rightsquigarrow} \mathsf{U}\,(C)^{\ell}}$$

$\boxed{\tau \overset{\text{comp}}{\rightsquigarrow} C}$ *(The CBV type* τ *translates to the CBPV computation type* C*)*

$$\text{CBV-TrCompTy-arrow}\quad \frac{\tau_1 \overset{\text{val}}{\rightsquigarrow} A \qquad \tau_2 \overset{\text{comp}}{\rightsquigarrow} (C)^{\ell}}{\tau_1 \overset{\ell}{\rightarrow} \tau_2 \overset{\text{comp}}{\rightsquigarrow} A \rightarrow (C)^{\ell}}$$

$$\text{CBV-TrCompTy-free}\quad \frac{\tau \overset{\text{val}}{\rightsquigarrow} A}{\tau \overset{\text{comp}}{\rightsquigarrow} \mathsf{F}_{\ell}\,A}$$

Figure 16: CBV types into CBPV types.

$\boxed{\mathsf{G} \vdash \mathsf{M} : \tau \overset{\text{val}}{\rightsquigarrow} (\Gamma \vdash v : A)}$ *(Under* G*, the CBV term* M *translates to CBPV value term* v*)*

$$\text{CBV-TrValTm-varVal}\quad \frac{\tau \overset{\text{val}}{\rightsquigarrow} A}{\mathsf{G} \vdash x : \tau \overset{\text{val}}{\rightsquigarrow} (\Gamma \vdash x : A)}$$

$$\text{CBV-TrValTm-unit}\quad \frac{}{\mathsf{G} \vdash () : 1 \overset{\text{val}}{\rightsquigarrow} (\Gamma \vdash () : 1)}$$

$$\text{CBV-TrValTm-inj1}\quad \frac{\begin{array}{c}\textbf{forall}\ j\ \textbf{in}\ \{1,2\}, \tau_j \overset{\text{val}}{\rightsquigarrow} A_j \\ \textbf{exists}\ i\ \textbf{in}\ \{1,2\} \\ \mathsf{G} \vdash \mathsf{M} : \tau_i \overset{\text{val}}{\rightsquigarrow} (\Gamma \vdash v : A_i)\end{array}}{\mathsf{G} \vdash \textbf{inj}_i\ \mathsf{M} : \tau_1 + \tau_2 \overset{\text{val}}{\rightsquigarrow} (\Gamma \vdash \textbf{inj}_i\ v : A_1 + A_2)}$$

$$\text{CBV-TrValTm-pair}\quad \frac{\begin{array}{c}\textbf{forall}\ i\ \textbf{in}\ \{1,2\} \\ \mathsf{G} \vdash \mathsf{M}_1 : \tau_i \overset{\text{val}}{\rightsquigarrow} (\Gamma \vdash v : A_i)\end{array}}{\mathsf{G} \vdash (\mathsf{M}_1, \mathsf{M}_2) : \tau_1 \times \tau_2 \overset{\text{val}}{\rightsquigarrow} (\Gamma \vdash \textbf{inj}_1\ v : A_1 \times A_2)}$$

$$\text{CBV-TrValTm-abs}\quad \frac{\mathsf{G}, x : \tau_1 \vdash_{\ell} \mathsf{M} : \tau_2 \overset{\text{comp}}{\rightsquigarrow} \Gamma, x : A \vdash e : C}{\mathsf{G} \vdash \lambda x.\mathsf{M} : \tau_1 \overset{\ell}{\rightarrow} \tau_2 \overset{\text{val}}{\rightsquigarrow} (\Gamma \vdash \textbf{thunk}\ \lambda x.e : \mathsf{U}\,(A \rightarrow C))}$$

$$\text{CBV-TrValTm-thunk}\quad \frac{\mathsf{G} \vdash_{\ell} \mathsf{M} : \tau \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash e : C}{\mathsf{G} \vdash \textbf{thunk}\ \mathsf{M} : \mathsf{U}^{\ell}\,\tau \overset{\text{val}}{\rightsquigarrow} (\Gamma \vdash \textbf{thunk}\ e : \mathsf{U}\,C)}$$

$$\text{CBV-TrValTm-addr}\quad \frac{\begin{array}{c}\tau \overset{\text{comp}}{\rightsquigarrow} C \\ \mathsf{G}(a) = \mathsf{M}^{\ell}\,\tau \qquad \Gamma(a) = C\end{array}}{\mathsf{G} \vdash a : \mathsf{M}^{\ell}\,\tau \overset{\text{val}}{\rightsquigarrow} (\Gamma \vdash a : \mathsf{M}\,C)}$$

Figure 17: CBV terms as CBPV value terms.

$\boxed{G \vdash_\ell M : \tau \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash e : C}$ *(Under G and ℓ, the CBV term M translates to CBPV computation term e)*

CBV-TrCompTm-abs
$$\frac{G, x : \tau_1 \vdash_\ell M : \tau_2 \overset{\text{comp}}{\rightsquigarrow} \Gamma, x : A \vdash e : C}{G \vdash_\ell \lambda x.M : \tau_1 \xrightarrow{\ell} \tau_2 \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash \lambda x.e : A \to C}$$

CBV-TrCompTm-app
$$\frac{G \vdash_\ell M_2 : \tau_1 \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash e_2 : F_\ell\ A \qquad G \vdash_\ell M_1 : \tau_1 \xrightarrow{\ell} \tau_2 \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash e_1 : A \to C}{G \vdash_\ell M_1\, M_2 : \tau_2 \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash \textbf{let}\, x \leftarrow e_2\, \textbf{in}\, e_1\, x : C}$$

CBV-TrCompTm-let
$$\frac{G \vdash_\ell M_1 : \tau_1 \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash e_1 : F_\ell\ A \qquad G, x : \tau_1 \vdash_\ell M_2 : \tau_2 \overset{\text{comp}}{\rightsquigarrow} \Gamma, x : A \vdash e_2 : C}{G \vdash_\ell \textbf{let}\, x \leftarrow M_1\, \textbf{in}\, M_2 : \tau_2 \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash \textbf{let}\, x \leftarrow e_1\, \textbf{in}\, e_2 : C}$$

CBV-TrCompTm-ret
$$\frac{G \vdash V : \tau \overset{\text{val}}{\rightsquigarrow} (\Gamma \vdash v : A)}{G \vdash_\ell V : \tau \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash \textbf{ret}\, v : F_\ell\ A}$$

CBV-TrCompTm-fix
$$\frac{G, f : \tau \vdash_\ell M : \tau \overset{\text{comp}}{\rightsquigarrow} \Gamma, f : C \vdash e : C}{G \vdash_\ell \textbf{fix}\, f.M : \tau \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash \textbf{fix}\, f.e : C}$$

CBV-TrCompTm-varFix
$$\frac{\tau \overset{\text{comp}}{\rightsquigarrow} C}{G \vdash_\ell f : \tau \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash f : C}$$

CBV-TrCompTm-inj
$$\frac{\textbf{forall}\, j\, \textbf{in}\, \{1,2\}, \tau_j \overset{\text{val}}{\rightsquigarrow} A_j \qquad \textbf{exists}\, i\, \textbf{in}\, \{1,2\} \qquad G \vdash_\ell M : \tau \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash e : F_\ell\ A_i}{G \vdash_\ell \textbf{inj}_i\, M : \tau_1 + \tau_2 \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash \textbf{let}\, x \leftarrow e\, \textbf{in}\, \textbf{ret}\, \textbf{inj}_i\, x : F_\ell\ (A_1 + A_2)}$$

CBV-TrCompTm-case
$$\frac{\textbf{forall}\, i\, \textbf{in}\, \{1,2\} \qquad G \vdash_\ell M : \tau \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash e : F_\ell\ (A_1 + A_2) \qquad G, x : \tau_i \vdash_\ell M : \tau_3 \overset{\text{comp}}{\rightsquigarrow} \Gamma, x : A_i \vdash e_i : C}{G \vdash_\ell \textbf{case}\, (M, x_1.M_1, x_2.M_2) : \tau_3 \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash \textbf{let}\, x \leftarrow e\, \textbf{in}\, \textbf{case}\, (x, x_1.e_1, x_2.e_2) : C}$$

CBV-TrCompTm-pair
$$\frac{\textbf{forall}\, i\, \textbf{in}\, \{1,2\} \qquad G \vdash_\ell M_i : \tau_i \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash e_i : F_\ell\ A_i}{G \vdash_\ell (M_1, M_2) : \tau_1 \times \tau_2 \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash \textbf{let}\, x_1 \leftarrow e_1\, \textbf{in}\, \textbf{let}\, x_2 \leftarrow e_2\, \textbf{in}\, \textbf{ret}\, (x_1, x_2) : F_\ell\ (A_1 \times A_2)}$$

CBV-TrCompTm-fst
$$\frac{G \vdash_\ell M : \tau_1 \times \tau_2 \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash e : F_\ell\ (A_1 \times A_2)}{G \vdash_\ell \textbf{fst}\, M : \tau_1 \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash \textbf{let}\, x \leftarrow e\, \textbf{in}\, \textbf{split}\, (x, y_1.y_2.\textbf{ret}\, y_1) : F_\ell\ A_1}$$

CBV-TrCompTm-snd
$$\frac{G \vdash_\ell M : \tau_1 \times \tau_2 \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash e : F_\ell\ (A_1 \times A_2)}{G \vdash_\ell \textbf{snd}\, M : \tau_2 \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash \textbf{let}\, x \leftarrow e\, \textbf{in}\, \textbf{split}\, (x, y_1.y_2.\textbf{ret}\, y_2) : F_\ell\ A_2}$$

CBV-TrCompTm-thunk
$$\frac{G \vdash_{\ell_2} M : \tau \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash e : C}{G \vdash_{\ell_1} \textbf{thunk}\, M : U^{\ell_2}\ \tau \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash \textbf{ret}\, (\textbf{thunk}\, e) : F_{\ell_1}\ U\ C}$$

CBV-TrCompTm-force
$$\frac{G \vdash_\ell M : U^{\ell}\ \tau \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash e : F_\ell\ U\ C}{G \vdash_\ell \textbf{force}_\ell\ M : \tau \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash \textbf{let}\, x \leftarrow e\, \textbf{in}\, \textbf{force}_\ell\ x : C}$$

CBV-TrCompTm-ref
$$\frac{G \vdash_\ell M : \tau \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash e : C}{G \vdash_{\text{outer}} \textbf{ref}\, M : 1 \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash \textbf{ref}\, e : F_{\text{outer}}\ 1}$$

CBV-TrCompTm-get
$$\frac{G \vdash_\ell M : M^\ell\ \tau \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash e : F_\ell\ (M\ C)}{G \vdash_\ell \textbf{get}\, M : \tau \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash \textbf{let}\, x \leftarrow e\, \textbf{in}\, \textbf{get}\, x : C}$$

CBV-TrCompTm-set
$$\frac{G \vdash_{\text{outer}} M_2 : U^{\ell}\ \tau \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash e_1 : F_{\text{outer}}\ (U\ C) \qquad G \vdash_{\text{outer}} M_1 : M^\ell\ \tau \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash e_2 : F_{\text{outer}}\ (M\ C)}{G \vdash_{\text{outer}} \textbf{set}\, M_1 \leftarrow M_2 : 1 \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash \textbf{let}\, x_1 \leftarrow e_1\, \textbf{in}\, \textbf{let}\, x_2 \leftarrow e_2\, \textbf{in}\, \textbf{set}\, x_1 \leftarrow x_2 : F_{\text{outer}}\ 1}$$

Figure 18: CBV terms as CBPV computation terms.

$\boxed{G \vdash \dot{S} \rightsquigarrow \Gamma \vdash S}$ *(Under G, the CBV store Ṡ translates to CBPV store S)*

CBV-TrStore-emp
$$\frac{}{G \vdash \varepsilon \rightsquigarrow \Gamma \vdash \varepsilon}$$

CBV-TrStore-cons
$$\frac{G \vdash \dot{S} \rightsquigarrow \Gamma \vdash S \qquad G(a) = M^\ell\ \tau \qquad G \vdash_\ell M : \tau \overset{\text{comp}}{\rightsquigarrow} \Gamma \vdash e : C}{G \vdash \dot{S}, a{:}M \rightsquigarrow \Gamma \vdash S, a{:}e}$$

Figure 19: CBV stores as CBPV stores.