Incremental Computation with Names

Abstract

Over the past thirty years, there has been significant progress in developing general-purpose, language-based approaches to *incremental computation*, which aims to efficiently update the result of a computation when an input is changed. A key design challenge in such approaches is how to provide efficient incremental support for a broad range of programs. In this paper, we argue that first-class *names* are a critical linguistic feature for efficient incremental computation. Names identify computations to be reused across differing runs of a program, and making them first class gives programmers a high level of control over reuse. We demonstrate the benefits of names by presenting NOMINAL ADAPTON, an ML-like language for incremental computation with names. We describe how to use NOMINAL ADAPTON to efficiently incrementalize several standard programming patterns—including maps, folds, and unfolds—and show how to build efficient, incremental probabilistic trees and tries. Since NOMINAL ADAPTON's implementation is subtle, we formalize it as a core calculus and prove it is *from-scratch consistent*, meaning it always produces the same answer as simply re-running the computation. Finally, we demonstrate that NOMINAL ADAPTON can provide large speedups over both from-scratch computation and ADAPTON, a previous state-of-the-art incremental system.

1. Introduction

Memoization is a widely used technique to speed up running time by caching and reusing prior results (Michie 1968). The idea is simple—the first time we call $f(\vec{x})$, we store the result r in a *memo table* mapping $\vec{x}$ to r. Then on subsequent calls $f(\vec{y})$, we can return r immediately if $\vec{x}$ and $\vec{y}$ match exactly. *Incremental computation* (IC) (Pugh 1988) takes this idea a step further, aiming to reuse prior computation even if there is a small change in the input. For example, IC may be able to use a prior computation $f(\vec{x}, a)$ in a computation $f(\vec{x}, b)$, even if $a \neq b$. A key design choice in IC and memoization systems in general is how to decide whether a computation is "the same" as a prior computation so that a memoized result can be reused.

Prior work on IC such as that of Pugh (1988), Reps (1982b) and ADAPTON (Hammer et al. 2014) have focused on a *structural* approach to memoization where a computation is considered "the same" if it depends on identical data, where primitives are compared by value and pointers are compared by address.

This approach to matching is effective, but it has a major downside: the pointer identity of addresses is fragile and non-deterministic. Thus, during recomputation, the address of a reference may change coincidentally, leading to missed opportunities for reuse and thereby reducing performance. This reduction is exacerbated when spurious changes cascade—a single change to a pointer identity can cause subsequent computations built on top of it to also change their pointer identities. Since each newly allocated computation requires a new slot in the memo table, these cases of missed reuse also increase memory usage.

In this paper, we present NOMINAL ADAPTON, an incremental programming language that gives programmers more control over the recomputation matching process. In NOMINAL ADAPTON, references can be given a *name* by the programmer. When performing memoization, references are compared by this name, rather than by address. NOMINAL ADAPTON provides facilities to generate names *deterministically*, based on previously generated names.

As an example, suppose we map a function over a list `in`, producing a new list `out`. Then we update `in` to insert a new element `x` into `in`, and wish to produce an updated output list `out'`, reusing as much of the previous computation as possible. Many prior approaches to IC will successfully reuse the computation that maps the list following `x`, but will fail to reuse the prefix that precedes it due to cascading disequalities. In NOMINAL ADAPTON, the programmer avoids this problem by deterministically naming references in the output list based on the corresponding element in the input list. As such, the entire prefix will match, saving both time and space. Section 2 presents this example in detail, giving an overview of the NOMINAL ADAPTON programming model and how it improves performance compared to prior structural approaches.

NOMINAL ADAPTON allows the programmer to say when two references are morally "the same," but the programmer can make mistakes. As hinted at above, the best use of names is to "connect" an input and an output computation by giving them related names, where the lack of a named connection implies no dependence, and thus no need for recomputation. A programmer could choose names badly, either suggesting a dependence where none exists or occluding a non-dependence. Both cases hamper reuse. For-

tunately, as we discuss in Section 3, there are simple design patterns for using names that avoid these problems. We discuss ones we have developed for lists, trees, and common computation patterns over them. We also propose a fundamental data structure for probabilistically balanced trees that works in a variety of applications.

A worse mistake the programmer could make would be to use the same name for two distinct references; this is problematic because it would cause a result from one use to match when the other use should match instead. NOMINAL ADAPTON uses an efficient run-time check to abort if this ever happens. To avoid contorted programming patterns that might otherwise result, NOMINAL ADAPTON includes first-class *namespaces*—the same name can be reused as long as each use is in a separate namespace. For example, we can safely map different functions over the same list by wrapping the computations in separate namespaces. We have formalized NOMINAL ADAPTON in a core calculus λ_{NomA} and proved its incremental recomputation is *from-scratch consistent*, meaning it produces the same answer as would a recomputation from scratch. As such, mistakes from the programmer will never produce incorrect results. Section 4 presents our formalism and theorem.

We implemented NOMINAL ADAPTON in OCaml as an extension to ADAPTON (Section 5). We evaluated our implementation by comparing it to ADAPTON on a set of subject programs commonly evaluated in the IC literature, including map, filter, reduce, reverse, median, mergesort, and quickhull. As an even more involved example, we implemented an interpreter for an imperative programming language (IMP with arrays), showing that interpreted programs enjoy incrementality by virtue of using NOMINAL ADAPTON as the meta-language. Across our benchmarks, we find that ADAPTON is nearly always slower than NOMINAL ADAPTON (sometimes *orders of magnitude* slower), and is sometimes orders of magnitude slower than from-scratch computation. By contrast, NOMINAL ADAPTON uniformly enjoys speedups over from-scratch computation (up to $10900\times$) as well as classic ADAPTON (up to $21000\times$). (Section 6 describes our experiments.)

The idea of names has come up in prior incremental computation systems, but only in an informal way. For example, Acar and Ley-Wild (2009) includes a paragraph describing the idea of named references (there called "keys") in the DeltaML implementation of SAC. To our knowledge, our work is the first to formalize a notion of named computations in IC and prove their usage correct. We are also the first to empirically evaluate the costs and benefits of programmer-named references and thunks. Finally, the notion of first-class namespaces, with the same determinization benefits as named thunks and references, is also new. (Section 7 discusses SAC and other related work in more detail.)

2. Overview

In this section we present NOMINAL ADAPTON and its programming model, illustrating how names can be used to improve opportunities for reuse. We start by introducing ADAPTON's approach to incremental computation, highlighting how NOMINAL ADAPTON extends its programming model with support for names. Next we use an example, mapping over a list, to show how names can be used to improve incremental performance.

2.1 ADAPTON and NOMINAL ADAPTON

ADAPTON aims to reuse prior computations as much as possible after a change to the input. ADAPTON achieves this by memoizing a function call's arguments and results, reusing memoized results when the arguments match. In this section, we write $\mathsf{memo}(e)$ to indicate that the programmer wishes e to be memoized.[1]

ADAPTON provides mutable references: ref e allocates a memory location p which it initializes to (the result of evaluating) e, and ! p retrieves the contents of that cell. Changes to inputs are expressed via reference cell mutations; ADAPTON propagates the effect of such changes to update previous results. Like many approaches to incremental computation, ADAPTON distinguishes two layers of computation. Computations in the *inner layer* are incremental, but can only read and allocate references, while computations in the *outer layer* can change reference values (necessitating change propagation for the affected inner-layer computations) but are not themselves incremental. This works by having the initial incremental run produce a *demanded computation graph* (DCG), which stores values of memoized computations and tracks dependencies between those computations and references. Changes to mutable state "dirty" this graph, and change propagation "cleans" it, making its results consistent.

NOMINAL ADAPTON extends ADAPTON's programming model with the notion of a *name*. Programmers can give a name when creating a reference cell; this name will be used for memoization. At a high level, the goal is to localize changes by using names to relate input structures to corresponding output structures while isolating independent sub-structures, thus improving reuse. Importantly, NOMINAL ADAPTON provides facilities for deterministically creating names, and for using names correctly, so that programmers can safely exploit opportunities for reuse.

2.2 Incremental Computation in ADAPTON

As a running example, consider incrementalizing a program that maps over a list's elements. To support this, we define a list data structure that allows the tail to be imperatively modified by the outer context:

[1] Programmers actually have more flexibility thanks to ADAPTON's support for laziness, but laziness is orthogonal to names, which we focus on in this section. We discuss laziness in Section 4.

```
type 'a list = Nil | Cons of 'a * ('a list) ref
let rec map f xs = memo( match xs with
  | Nil → Nil
  | Cons (x, xs) → Cons (f x, ref (map f !xs)) )
```

This is a standard map function, except for two twists: the function body is memoized via memo, and the input and output Cons tails are reference cells. The use of memo here records function calls to map, identifying prior calls using the function f and input list xs. In turn, xs is either Nil or is identified by a value of type 'a and a reference cell. Hence, reusing the identity of references is critical to reusing calls to map via memo. Now we can create a list (in the outer layer) and map over it (in the inner layer):

```
let l3 = Cons(3, ref Nil)  (* l3 = [3] *)
let l1 = Cons(1, ref l3)   (* l1 = [1; 3] *)
let l0 = Cons(0, ref l1)   (* l0 = [0; 1; 3] *)
let m0 = (* inner start *) map f l0 (* inner end *)
   (* m0 = [f 0; f 1; f 3] *)
```

Suppose we change the input to map by inserting an element:

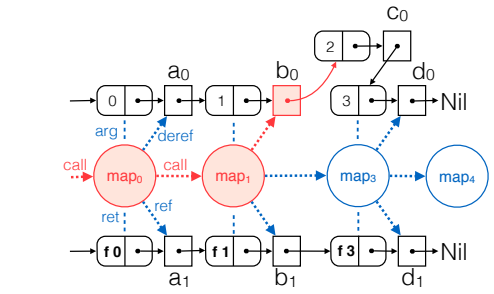
```
(tl l1) := Cons(2, ref l3)  (* l0 = [0; 1; 2; 3] *)
```

Here, tl returns the tail of its list argument. After this change, m0 will be updated to [f 0; f 1; f 2; f 3]. In principle, computing m0 should only require applying f 2 and inserting the result into the original output. However, ADAPTON performs much more work for the above code. Specifically, ADAPTON will recompute f 0 and f 1, and in fact were the change in the middle of a longer list it would recompute the *entire* prefix of the list before the change. In contrast, NOMINAL ADAPTON will only redo the minimal amount of work.

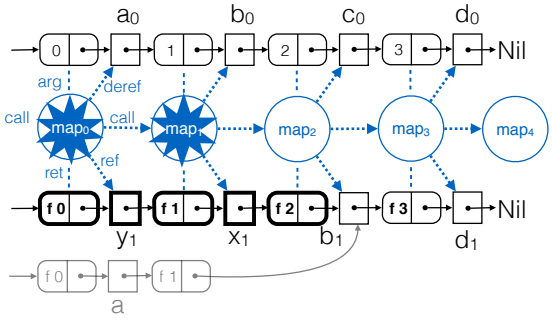
To understand why, consider Figure 1a, which illustrates what happens after the list update. In this figure, the initial input and output lists are shown in black at the top and the bottom of the figure, respectively. The middle of the figure shows the *demanded computation graph* (DCG), which records each recursive call of map and its dynamic dependencies. Here, nodes map_0, map_1, map_3 and map_4 correspond to the four calls to map. For each call, the DCG records the arguments, the result, and the computation's effects. Here, the effects are: dereferencing a pointer; making a recursive call; and allocating a ref cell in the output list. We label the arrows/lines of the first node only, to avoid clutter; the same pattern holds for map_1 and map_3.

In the input and output, the tail of each Cons cell (a rounded box) consists of a reference (a square box). The input list is labeled a_0, b_0, d_0, and the output list is labeled a_1, b_1, d_1. In ADAPTON, these labels are chosen based on *content*, meaning that if a program tries to allocate a reference with value v and some reference already holds v, the existing reference will be *shared*. If no such reference exists, a fresh label is chosen *arbitrarily*.

After the list is updated, ADAPTON dirties all the computations that transitively depend on the changed reference



(a) ADAPTON, after insertion update.



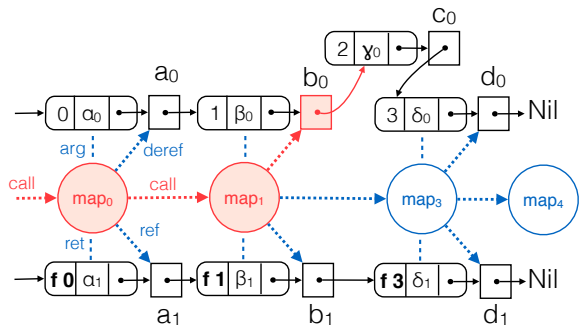
(b) ADAPTON, after change propagation.

Figure 1: Incremental computation of map in ADAPTON.

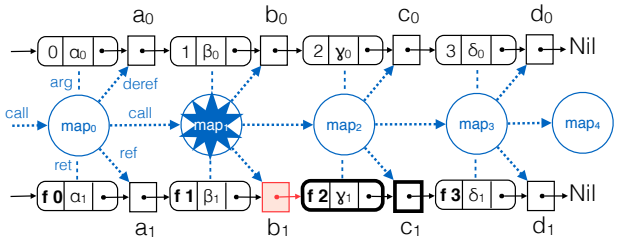
cell, b_0. The dirtied elements are shaded in red. Dirtying is how ADAPTON knows that previously memoized results cannot be reused safely without further processing. ADAPTON processes dirty parts of the DCG into clean computations on demand, when they are demanded by the outer program. To do so, it either re-executes the dirty computations, or verifies that they are unaffected by the original set of changes.

Figure 1b shows the result of recomputing the output following the insertion change, using this mechanism. When the outer program recomputes map f l0 (shown as map_0), ADAPTON will clean (either recompute or reuse) the dirty nodes of the DCG. First, it re-executes computation map_1, because it is the first dirty computation to be affected by the changed reference cell. We indicate re-executed computations with stars in their DCG nodes. Upon re-execution, map_1 dereferences b_0 and calls map on the inserted Cons cell holding 2. This new call map_2 calls f 2 (not shown), dereferences c_0, calls map_3's computation map f $(\mathsf{Cons}(3,d_0))$, allocates b_1 to hold its result and returns $\mathsf{Cons}(\mathsf{f}\ 3,\ d_1)$.

The recomputation of map_2 exploits two instances of reuse. First, when it calls map f $(\mathsf{Cons}(3,d_0))$, ADAPTON reuses this portion of the DCG and the result it computed in the first run. ADAPTON knows that the prior result of map f $(\mathsf{Cons}(3,d_0)$ is unchanged because map_3 is not dirty. Notice that even if the tail of the list were much longer, the prior computation of map_3 could still be reused, since the



(a) NOMINAL ADAPTON, after insertion update.



(b) NOMINAL ADAPTON, after change propagation.

Figure 2: IC of map with NOMINAL ADAPTON.

insertion doesn't affect it. Second, when map_2 allocates the reference cell to hold the result of map f (Cons(3,d_0), i.e. Cons(f 3, d_1), it reuses and shares the existing reference cell b_1 that already holds this content.

So far, ADAPTON has successfully reused sub-computations, but consider what happens next. When the call map_2 completes, map_1 resumes control and allocates a reference cell to hold Cons(f 2, b_1). Since no reference exists with this content, it allocates a fresh reference x_1. New references are shown in bold. The computation then returns Cons(f 1, x_1), which does not match its prior return value Cons(f 1, b_1). Since this return value has changed from the prior run, ADAPTON re-runs map_1's caller, map_0. This consists of re-running f 0, reusing the (just recomputed, hence no longer dirty) computation map_1, and allocating a reference to hold its result Cons(f 1, x_1). Again, no reference exists yet with this content (x_1 is a fresh tail pointer), so ADAPTON allocates a fresh cell y_1. Finally, the call to map_0 completes, returning a new list prefix with the *same content* (f 0 and f 1) as in the first run, but with new reference cell identities (y_1 and x_1).

In this example, small changes cascade into larger changes because ADAPTON identifies reference cells *structurally* based on their contents. Thus, the entire prefix of the output list before the insertion is reallocated and recomputed, which is much more work than should be necessary.

2.3 The Nominal Approach

We can solve the problems with structural reuse by giving the programmer explicit *names* to control reuse. In this particular case, we aim to avoid re-computing map_0 and any preceding computations. In particular, we wish to recompute only map_1 (since it reads a changed pointer) and to compute map_2, the mapping for the inserted Cons cell.

The first step is to augment mutable lists with names provided by NOMINAL ADAPTON:

```
type list = Nil | Cons of int * name * (list ref)
```

Globally fresh names are generated either non-deterministically via new, or from an existing name via fork. In particular, fork n returns a pair of distinct names based on the name n with the property that it always returns the *same* pair of names given the same name n. In this way, the inner layer can *deterministically* generate additional names from a given one to enable better reuse. Finally, when the programmer allocates a reference cell, she explicitly indicates which name to use, e.g. ref(n,Cons(0,Nil)) instead of ref(Cons(0,Nil)).

Now, when the list is created, the outer layer calls new to generate fresh, globally distinct names for each Cons cell:

```
let l3 = Cons(3, new, ref(new, Nil))  (* l3 = [3] *)
let l1 = Cons(1, new, ref(new, l3 ))  (* l1 = [1; 3] *)
let l0 = Cons(0, new, ref(new, l1 ))  (* l0 = [0; 1; 3] *)
```

When the inner layer computes with the list, it uses the names in each Cons cell to indicate dependencies between the inputs and outputs of the computation. In particular, we rewrite map as follows:

```
let rec map f xs = memo( match xs with
  | Nil → r
  | Cons(x, nm, xs) →
    let nm1, nm2 = fork nm in
      Cons(f x, nm1, ref (nm2, map f (!xs))) )
```

Unlike the outer program, which chooses reference names in an unpredictable way, the inner program uses fork to relate the names and references in the output list to the *names* in the input list.

Now consider applying this function and making the same change as above:

```
let m0 = map f l0 in
(tl l1) := Cons(2, new, ref l3)
```

Figure 2 shows what happens. The initial picture in Figure 2a is similar to the structural case in Figure 1a, except the input and output lists additionally contain names $\alpha_0, \beta_0, \gamma_0$ and δ_0. The first part of the recomputation is the same: NOMINAL ADAPTON re-computes map_1, which reads the mutated reference b_0. In turn, it recomputes map_2, which reuses the call to map_3 to compute Cons(f 2, γ_1, c_1). The recomputation of map_2 returns a different value than in the prior run, with the new head value f 2.

At this point, the critical difference occurs. Even though the result of map_2 is distinct from any list in the prior run, the

call map_1 allocates the *same* ref cell b_0 as before, because the name it uses for this allocation, β_1, is the same as before. In the figure, fork $\beta_0 \mapsto (\beta_1,\beta_2)$, where β_1 becomes the name in the output list and $\mathsf{b}_1 = \mathsf{ref}\ \beta_2$ identifies the reference cell in its tail. NOMINAL ADAPTON dirties the reference b_1, to ensure that any dependent computations will be cleaned before their results are reused. Due to this reuse, the result of the call map_1 is identical to its prior result: The value of f 1 is unaffected, and the tail pointer b_1 was reused exactly. Next, ADAPTON examines map_0 and all prior calls. Because the return value of map_1 did not change, NOMINAL ADAPTON simply marks the DCG node for its caller, map_0, as clean; no more re-evaluation occurs. This cleaning step breaks the cascade of changes that occurred under ADAPTON. Prior computations are now clean, because they only depend on clean nodes.

As a result of this difference in behavior, NOMINAL ADAPTON is able to reuse all but two calls to function f for an insertion at any index i, while ADAPTON will generally re-execute all $\mathsf{i}-1$ calls to function f that precede the inserted cell. Moreover, ADAPTON allocates a new copy of the output prefix (from 0 to i), while NOMINAL ADAPTON reuses all prior allocations. Our experiments (Section 6) confirm that these differences make ADAPTON over $10\times$ slower than NOMINAL ADAPTON, even for medium-sized lists (10k elements) and cheap instances of f (integer arithmetic).

2.4 Namespaces

Suppose we want to map an input list twice:

```
let ys = map f input_list
let zs = map g input_list
```

Recall that in the Cons case of `map` we use each name it contains. But since we perform two calls to `map`, we use each name in the input twice, which leads to *nominal ambiguity*.

Avoiding such ambiguity is necessary for correctness: if later in the program, we require ys is still the map of f over input_list, this will not hold, since the names in input_list are overwritten by map g input_list on the following line.

The solution is to create distinct *namespaces* for distinct functions (`f` versus `g`). A modified version of map using namespaces would be written thus:

```
let map' n h xs = nest(ns(n), map h xs)
```

The code nest(s,e) performs the nested computation e in namespace s, and the code ns(n) creates a namespace from a given name n. Just as with references, we must be careful about how namespaces correspond across different incremental runs, and thus we seed a namespace with a given name. Now, distinct callers can safely call map' with distinct names:

```
let n1, n2 = (new, new) in
let xs = map' n1 f input_list in
let ys = map' n2 g input_list in
```

The result is that each name in the input list is used only once per namespace: Names in map f will be associated with the first namespace (named by n1), and names in map g will associate with the second namespace (named by n2).

In sum, the use of names allows the programmer to control (1) how mutable reference names are chosen the first time, and (2) how to selectively reuse and overwrite these references to account for incremental input changes from the outer layer. These names are transferred from input to output through the data structures that they help identify (the input and output lists here), by the programs that process them (such as map). Sometimes we want to use the same name more than once, in different program contexts (e.g., map f · versus map g ·); we distinguish these program contexts using namespaces.

3. Programming with Names

While NOMINAL ADAPTON's names are a powerful tool for improving incremental reuse, they create more work for the programmer. In our experience so far, effective name reuse follows easy-to-understand patterns. Section 3.1 shows how to augment standard data structures—lists and trees—and operations over them—maps, folds, and unfolds—to incorporate names in a way that supports effective reuse. Section 3.2 describes *probabilistic tries*, a nominal data structure we developed that efficiently implements incremental maps and sets. Finally, Section 3.3 describes our implementation of an incremental IMP interpreter that takes advantage of these data structures to support incremental evaluation of its imperative input programs. The benefits of these patterns and structures are measured precisely in Section 6.

3.1 General Programming Patterns

Practical functional programs use a wide variety of programming patterns; three particularly popular ones are mapping, folding, and unfolding. We consider them here in the context of lists and trees.

Mapping. Maps traverse a list (as in Section 2) or tree and produce an output structure that has a one-to-one correspondence with the input structure. We have already seen how to incrementalize mapping by associating a name with each element of the input list and using fork to derive a corresponding name for each element in the output list, thereby avoiding spurious recomputation of whole list prefixes on a change.

Folding. Folds traverse a list or tree and reduce subcomputations to provide a final result. Examples are summing list elements or finding the minimum element in a tree.

If we implement folding in a straightforward way in NOMINAL ADAPTON, the resulting program tends to perform poorly. The problem is that every step in a list-based reduction uses an accumulator or result that induces a global dependency on all prior steps—i.e., every step depends on

the *entire prefix* or the *entire suffix* of the sequence, meaning that any change therein necessitates recomputing the step.

The solution is to use an approach from parallel programming: Use trees to structure the input data, rather than lists, to permit expressing independence between sub-problems. Consider the following code, which defines a type tree for trees of integers:

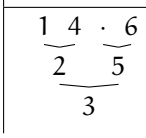
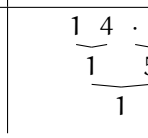
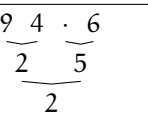
```
type tree = Leaf | Bin of name * int * (tree ref) * (tree ref)
```

Like lists, these trees use refs to permit their recursive structure to change incrementally, and each tree node includes a name. We can reduce over this tree in standard functional style:

```
let tree_min tree = memo(match tree with
  | Leaf         → max_int
  | Bin(n,x,l,r) → min3(x, tree_min(!l), tree_min(!r)))
```

If we later update the tree and recompute, we can reuse subtree minimum computations, because the names are stable in the tree.

Below, we show the original input tree alongside two illustrations (also depicted as trees) of which element of each subtree is the minimum element, before and after the replacement of element 1 with the new element 9:

Original tree	Minimums (pre-change)	Minimums (post-change)
		

Notice that while the first element 1 changed to 9, this only affects the minimum result along one path in the tree: the path from the root to the changed element. In contrast, if we folded the sequence naively as a list, all the intermediate computations of the minimum could be affected by a change to the initial element (or final element, depending on the fold direction). By contrast, the balanced tree structure (with expected logarithmic depth) overcomes this problem by better isolating independent sub-computations.

Pleasingly, as first shown by Pugh and Teitelbaum (1989), we can efficiently build a tree probabilistically from an input list, and thus transfer the benefits of incremental tree reuse to list-style computations. Building such a tree is an example of unfolding, described next.

Unfolding. Unfolds iteratively generate lists or trees using a "step function" with internal state. As just mentioned, one example is building a balanced tree from a list. Unfortunately, if we implement unfolding in a straightforward way, incremental computation suffers. In particular, we want similar lists (related by small edits) to lead to similar trees, with many common subtrees; meanwhile, textbook algorithms for building balanced trees, such as AVL trees and splay trees, are too sensitive to changes to individual list elements.

The solution to this problem is to construct a *probabilistically balanced tree*, with expected $O(\log n)$ height for input list n. The height of each element in the resulting tree is determined by a function that counts the number of trailing zero bits in a hash of the given integer.

For example, if input elements $[a, b, c, d, e, f]$ have heights $[0, 1, 0, 2, 1, 0]$, respectively, then our tree-construction function will produce the binary tree in Figure 3.

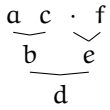


Figure 3: Example tree

Pugh and Teitelbaum (1989) showed that this procedure induces a probabilistically balanced tree, with similar lists inducing similar" trees, as desired. Further, each distinct list of elements maps to exactly one tree structure. This property is useful in NOMINAL ADAPTON, since a canonical structure is more likely to be reused than one that can exhibit more structural variation. While past work has considered incremental computations over such balanced trees, in this work we find that the *construction* of the tree from an mutable, changing sequence can also be efficiently incrementalized (Pugh's work focused only on a *pure* outer program).

3.2 Probabilistic Tries

Inspired by probabilistic trees, we developed efficient, incremental *probabilistic tries*, which use a different naming pattern in which certain names are *external* to the data structure.

We define tries as binary trees whose nodes hold a name and two children (in reference cells), and whose leaves store data. Here we use integers for simplicity, but in general nodes would hold arbitrary data (e.g., for maps, they would hold key-value pairs):

```
type trie =
    Nil | Leaf of int | Bin of name * (trie ref) * (trie ref)
```

The key idea of a probabilistic trie is to use a bit string to identify two things at once: the element stored in the structure (via its hash) and the path to retrieve that element, if it is present. To keep it simple, the code below assumes that all data elements have a unique hash, and that the input trie is complete, meaning that all paths are defined and either terminate in a Nil or a Leaf. Our actual implementation of tries makes neither assumption.

The first operation of a trie is find, which returns either Some data element or None, depending on whether data with the given hash (a list of bools) is present in the trie.

```
val find : trie → bool list → int option
let rec find trie bits =
  match bits, trie with
  | [], Leaf(x) → Some x
  | [], Nil     → None
  | true::bits,  Bin(_,left,right) → find left  bits
  | false::bits, Bin(_,left,right) → find right bits
```

The other operation on tries is extend n t b d which, given an input trie t, a data element d and its hash b, produces a new trie with d added to it:

```
val extend : name → trie → bool list → int → trie
let rec extend nm trie bits data = memo (
  match bits, trie with
  | [],           (Leaf _ | Nil)    → Leaf(x)
  | false::bits, Bin(_,left,right) → (* symmetric to below *)
  | true::bits,  Bin(_,left,right) →
      let n1, n2, n3, n4 = fork nm in
      let right' = extend n1 (!right) bits data in
        Bin(n2, ref(n3, !left), ref(n4, right'))  )
```

Critically, the first argument to extend is an externally provided nm that is used to derive names (using fork) for each ref in the new path. Thus, the identity of the trie returned by extend n t b d only depends directly on the name n and the inserted data, and *not* on the names or other content of the input trie t.

Any incremental program that sequences multiple trie extensions makes critical use of this independence (e.g., the interpreter discussed below). To see how, consider two incremental runs of such a program with two similar sequences of extensions, $[1,2,3,4,5,6]$ versus $[2,3,4,5,6]$, with the same sequence of five names for common elements $[2,\ldots,6]$. Using the name-based extension above, the tries in both runs will use exactly the same reference cells. By contrast, the structural approach will build entirely new tries in the second run, since the second sequence is missing the leading 1 (a different initial structure). Similarly, using the names *in the trie* to extend it will also fail here, since it will effectively identify each extension by a global count, which in this case shifts by one for every extension in the second run. Using external names for each extension overcomes both of these problematic behaviors, and gives maximal reuse.

3.3 Interpreter for IMP

Finally, as a challenge problem, we used NOMINAL ADAPTON to build an interpreter for IMP (Winskel 1993), a simple imperative programming language. Since our interpreter is incremental, we can efficiently recompute an interpreted program's output after a change to the program code itself. While Adapton requires incrementalized computations to be fully functional, implementing a purely functional interpreter for IMP allows the imperative object-language to inherit the incrementality of the meta-language.

The core of our interpreter is a simple, big-step eval function that recursively evaluates an IMP command in some store (that is, a heap) and environment, returning the final store and environment.

Commands include while loops, sequences, conditions, assignments (of arithmetic and boolean expressions) to variables, and array operations (allocation, reading, and writing). All program values are either booleans or integers. As in C, an integer also doubles as a "pointer" to the store. The interpreter uses finite maps to represent its environment of type env (a mapping from variables to ints) and its store of array content (a mapping from ints to ints).

For incremental efficiency, the interpreter makes critical use of the programming patterns we have seen so far. We use probabilistic tries to represent the finite maps for stores and environments. Each variable assignment or array update in IMP is implemented as a call to extend for the appropriate trie. Names have two uses inside the interpreter. Each call to extend requires a name, as does each recursive call to eval. Classic ADAPTON would identify each with the full structural content of its input. For the IMP language, we require far less information to disambiguate one program state from another.

Consider first the IMP language without while loops. Each subcommand is interpreted at most once; as such, each program state and each path created in the environment or store can be identified with the particular program position of the subcommand being interpreted.

With the addition of while loops, we may interpret a single program position multiple times. To disambiguate these, we thread loop counts through the interpreter represented as a list of integers. The loop count $[3,4]$, for example, tells us we are inside two while loops: the third iteration through the inner loop, within the fourth iteration through the outer loop. We extend the NOMINAL ADAPTON API to allow creating names not just by forking, but also by adding in a count—here, names are created using the loop count paired with the name of the program position. As such, the names sufficiently distinguish recursive calls to the interpreter and paths inside the environment and store.

```
val eval : nm * int list * store * env * cmd → store * env
```

In our experiments, we show that the combination of this naming strategy and probabalistically balanced tries yields an efficient incremental interpreter. Moreover, the interpreter's design provides evidence that NOMINAL ADAPTON's programming patterns are *compositional*, allowing us to separately choose how to use names for different parts of the program.

4. Formal Development

The interaction between memoization, names, and the demanded computation graph is subtle. For this reason, we have distilled NOMINAL ADAPTON to a core calculus called λ_{NomA}, which represents the essence of the NOMINAL ADAPTON implementation and formalizes the key algorithms behind incremental computation with names. We prove the fundamental correctness property of incremental computation, dubbed *from-scratch consistency*, which states that incrementally evaluating a program produces the same answer as re-evaluating it from scratch. This theorem also establishes that (mis)use of names cannot interfere with the meaning of programs; while a poor use of names may have

Values	v	::=	$x \mid (v_1, v_2) \mid \mathbf{inj}_i\ v$
		$\mid$	$\mathbf{ref}\ p \mid \mathbf{thk}\ p \mid \mathbf{nm}\ k \mid \mathbf{ns}\ \omega$
Results	t	::=	$\lambda x.e \mid \mathbf{ret}\ v$
Computations	e	::=	$t \mid e\ v \mid f \mid \mathbf{fix}\ f.e \mid \mathbf{let}\ x \leftarrow e_1\ \mathbf{in}\ e_2$
		$\mid$	$\mathbf{case}\ (v, x_1.e_1, x_2.e_2) \mid \mathbf{split}\ (v, x_1.x_2.e)$
		$\mid$	$\mathbf{ref}(v_1, v_2) \mid \mathbf{get}\ (v)$
		$\mid$	$\mathbf{thunk}(v, e) \mid \mathbf{force}\ (v)$
		$\mid$	$\mathbf{fork}(v) \mid \mathbf{ns}\ (v, x.e) \mid \mathbf{nest}(v, e_1, x.e_2)$
Pointers	p, q	::=	$k@\omega \mid \mathsf{root}$
Namespaces	ω, μ	::=	$\top \mid \omega.k$
Names	k	::=	$\bullet \mid k{\cdot}1 \mid k{\cdot}2$
Graphs	G	::=	$\varepsilon \mid G, p{:}v \mid G, p{:}e \mid G, p{:}(e, t)$
		$\mid$	$G, (p, a, b, q)$
Actions	a	::=	$\mathbf{alloc}\ v \mid \mathbf{alloc}\ e \mid \mathbf{obs}\ v \mid \mathbf{obs}\ t$
Statuses	b	::=	$\mathsf{clean} \mid \mathsf{dirty}$

Figure 4: Syntax of values, computations and graphs

negative impact on performance, it will not cause a program to compute the wrong result.

We sketch the theory and consistency result here, and we refer the reader to the appendix for full details.

Syntax. The syntax of λ_{NomA} is defined in the style of the call-by-push-value (CBPV) calculus (Levy 1999), a standard variant of the lambda calculus with an explicit thunk mechanism. The syntax of the language is given at the top of Figure 4. The non-highlighted features are standard, and the highlighted forms are new to λ_{NomA}.

NOMINAL ADAPTON follows ADAPTON, supporting *demand-driven* incremental computation using a lazy programming model. In ADAPTON, programmers can write $\mathsf{thunk}(e)$ to create a suspended computation, or *thunk*. The thunk v's value is computed only when it is *forced*, using syntax $\mathsf{force}(v)$. Thunks also serve as ADAPTON's (and NOMINAL ADAPTON's) unit of incremental reuse: if we want to reuse a computation, we must make a thunk out of it. The syntax $\mathsf{memo}(\cdots)$ we used earlier is shorthand for $\mathsf{force}(\mathsf{thunk}(n, \cdots))$ (for a fresh name n),[2] which introduces a thunk that the program immediately forces, eliminating laziness, but supporting memoization.

Graphs, pointers, and names. Graphs G are defined at the bottom of Figure 4. In the semantics they represent the mutable store (references), memo tables (which cache thunk results), and the DCG. Element $p{:}v$ says that pointer p's current value is v. Element $p{:}e$ says that pointer p is the name of thunk e. Element $p{:}(e, t)$ says that p is the name of thunk e with a previously computed result t attached. Element (p, a, b, q) is a DCG edge indicating that the thunk pointed to by p depends on node q, where q could either name another thunk or a reference cell. Dependency edges also reflect the *action* a that produced the edge and the edge's *status* b (whether it is clean or dirty).

[2] Notice that NOMINAL ADAPTON thunks are also named, which provides greater control over reuse.

Pointers in λ_{NomA} are represented as pairs $k@\omega$,[3] where $\mathbf{nm}\ k$ was the name given as the first argument in a call to **thunk** or **ref**, and ω was the namespace in which the call to **thunk** or **ref** took place. This namespace ω is either $\top$ for the top-level, or some $\mu.k_1$ as set by a call to **nest**. For the latter case, the program would first construct a value $\mathbf{ns}\ \mu.k_1$ by calling **ns** with first argument $\mathbf{nm}\ k_1$ while in namespace μ. Notice that pointers and namespaces have similar structure, and similar assurances of determinism when creating named thunks, references, and namespaces. Finally, names k consist of the root name $\bullet$, while other names are created by "forking" existing names: invoking $\mathbf{fork}(\mathbf{nm}\ k)$ produces names $\mathbf{nm}\ k{\cdot}1$ and $\mathbf{nm}\ k{\cdot}2$.

Semantics. We define a big-step operational semantics with judgments $G_1 \vdash^p_\omega e \Downarrow G_2; t$, which states that under input graph G_1 and within namespace ω, evaluating the computation e produces output graph G_2 and result t, where e's evaluation was triggered by a previous force of thunk p. Rules for standard language features (pairs, sums, functions, **fix**, and **let**) straightforwardly adapt the standard rules, ignoring p and ω and "threading through" input and output graphs. For example, applying **case** to a sum $\mathbf{inj}_i\ v$ substitutes v for x_i in the appropriate case arm:

$$\frac{G_1 \vdash^p_\omega [v/x_i]e_i \Downarrow G_2; t}{G_1 \vdash^p_\omega \mathbf{case}\ (\mathbf{inj}_i\ v, x_1.e_1, x_2.e_2) \Downarrow G_2; t}\ \textsf{Eval-case}$$

Incremental computation arises by making G_1 a modification of a previously produced graph G_2, and then re-running e. A legal modification involves replacing references $p{:}v$ with $p{:}v'$ and dirtying all edges along paths to p in the DCG: $\mathsf{dirty\text{-}paths\text{-}in}(G_1, p)$ is the same as G_1 but with edges on paths to p marked dirty.

The DCG is constructed during evaluation. The main rule for creating a thunk is Eval-thunkDirty:

$$\frac{\begin{array}{l} q = k@\omega \\ G_1\{q{\mapsto}e\} = G_2 \qquad \mathsf{dirty\text{-}paths\text{-}in}(G_2, q) = G_3 \end{array}}{G_1 \vdash^p_\omega \mathbf{thunk}(\mathbf{nm}\ k, e) \Downarrow G_3, (p, \mathbf{alloc}\ e, \mathsf{clean}, q); \mathbf{ret}\ (\mathbf{thk}\ q)}$$

This rule converts computation e into a thunk by generating a pointer q from the provided name k, which couples the name with the current namespace ω. The output graph is updated to map q to e. If q happens to be in the graph already, all paths to it will be dirtied. Finally, the rule adds edge $(p, \mathbf{alloc}\ e, \mathsf{clean}, q)$ to the output graph, indicating that the currently evaluating thunk p depends on q and is currently clean. Similarly, the rules for $\mathbf{ref}(\mathbf{nm}\ k, v)$, which allocates a reference $\mathbf{ref}\ q$ and initializes it to v, and $\mathbf{get}\ (\mathbf{ref}\ q)$, which retrieves q's value, add dependency edges from p to q.

[3] The pointer root is needed to represent the top-level "thunk" in the semantics, but will never be mapped to an actual value or expression.

The most interesting rules are the following two:

$$\frac{G(q) = (e, t) \qquad \text{all-clean-out}(G, q)}{G \vdash^{p}_{\omega} \textbf{force}\,(\textbf{thk}\,q) \Downarrow G, (p, \textbf{obs}\,t, \text{clean}, q); t}\ \text{Eval-forceClean}$$

$$\frac{\begin{array}{l}\exp(G_1, q) = e' \\ \text{del-edges-out}(G_1\{q{\mapsto}e'\}, q) = G_1' \\ G_1' \vdash^{q}_{\text{namespace}(q)} e' \Downarrow G_2; t' \\ \quad G_2\{q{\mapsto}(e', t')\} = G_2' \\ \quad \text{all-clean-out}(G_2', q) \\ \quad G_2' \vdash^{p}_{\omega} \textbf{force}\,(\textbf{thk}\,p_0) \Downarrow G_3; t\end{array}}{G_1 \vdash^{p}_{\omega} \textbf{force}\,(\textbf{thk}\,p_0) \Downarrow G_3; t}\ \text{Eval-computeDep}$$

The first rule, Eval-forceClean, performs memoization: Given q pointing to (e, t), if q's outgoing edges are clean then its cached result t is consistent, and t can be reused, i.e., returned immediately without reevaluating e.

The second rule is used to force thunks for the first time, or to perform selective recomputation to the point that a cached result can be reused. Its first premise $\exp(G_1, q) = e'$ nondeterministically chooses some thunk q whose suspended expression is e' (whether or not q also has a cached result). Its second premise $\text{del-edges-out}(G_1\{q{\mapsto}e'\}, q) = G_1'$ updates G_1 so that q points to e' (removing q's cached result, if any), and deletes outward edges of q. We need to delete the outward edges before evaluating e' because they represent what a *previous* evaluation of e' depended on. The third premise recomputes q's expression e', with q as the current thunk and q's namespace component as the current namespace. In the fourth premise, the recomputed result t' is cached, resulting in graph G_2'.

The final premise is the same as the conclusion, but under the graph G_2' containing the result of evaluating e'. In deriving this premise we may again apply Eval-computeDep to "fix up" other nodes of the graph, but will eventually end up with the thunk q chosen in the first premise being p_0 itself. In this case, the last premise of Eval-computeDep will be derived by Eval-forceClean (with q instantiated to p_0).

We skipped the fifth premise $\text{all-clean-out}(G_2', q)$, which demands that all outgoing edges from q in the updated graph are clean. This consistency check ensures that the program has not used the same name for two different thunks or references, e.g., by calling $\textbf{thunk}(\textbf{nm}\,k, e_1)$ and later $\textbf{thunk}(\textbf{nm}\,k, e_2)$ in the same namespace ω. If this happens, the graph will first map $k@\omega$ to e_1 but will later map it to e_2. Without this check, a computation q that depends on both e_1 and e_2 could be incorrect, because (re-)computing one of them might use cached values that were due to the other. Fortunately, this potential inconsistency is detected by all-clean-out: When a recomputation of q results in $k@\omega$ being mapped to a different value, all *existing* paths into $k@\omega$ are dirtied (by the last premise of Eval-thunkDirty above). Since q is one of the dependents, it will detect that fact and can signal an error.

The full set of evaluation rules is presented and explained in the supplementary material in Appendix B.

```
type name
val new : unit → name
val fork : name → name * name

type ('arg,'res) mfn
type ('arg,'res) mbody = ('arg,'res) mfn → 'arg → 'res
val mk_mfn : name → ('arg,'res) mbody → ('arg,'res) mfn
val call : ('arg,'res) mfn → ('arg → 'res)

type 'res athunk
val thunk : ('arg,'res) mfn → name → 'arg → 'res athunk
val force : 'res athunk → 'res

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

Figure 5: Basic NOMINAL ADAPTON API

From-scratch consistency. We show that an incremental computation modeled by our evaluation rules has a corresponding non-incremental computation: given an incremental evaluation of e that produced t, a corresponding non-incremental evaluation (using a far simpler set of rules) also produces t. Moreover, the values and expressions in the incremental output graph match those in the graph produced by the non-incremental evaluation.

Eliding some details and generalizations, the from-scratch consistency result is:

Theorem. *If incremental* $\mathcal{D}_i$ *derives* $G_1 \vdash^{p}_{\omega} e \Downarrow G_2; t$, *then a non-incremental* $\mathcal{D}_{ni}$ *derives* $\lfloor G_1 \rfloor_{P_1} \vdash^{p}_{\omega} e \Downarrow \lfloor G_2 \rfloor_{P_2}; t$ *where* $\lfloor G_1 \rfloor_{P_1} \subseteq \lfloor G_2 \rfloor_{P_2}$ *and* $P_2 = P_1 \cup \text{dom}(W)$.

Here, W is the set of pointers that $\mathcal{D}_i$ may allocate. The restriction function $\lfloor G_i \rfloor_{P_i}$ drops all edges from G_i and keeps only nodes in the set P_i. It also removes any cached results t. The set P_i corresponds to the nodes in G_i that are present at this point in the non-incremental derivation, which may differ from the incremental derivation since Eval-computeDep need not compute dependencies in left-to-right order.

The full statement, along with definitions of W, the restriction function, and lemmas, is in the supplementary material as Theorem C.13.

5. Implementation

We implemented NOMINAL ADAPTON as an OCaml library. In this section, we describe its programming interface, data structures, and algorithms. (Additional detail about memory management appears in Appendix A). The code for NOMINAL ADAPTON is freely available.

5.1 Programming Interface

Figure 5 shows the basic NOMINAL ADAPTON API. Two of the data types, name and aref, correspond exactly to names and references in Sections 2 and 4. The other data types, mfn and athunk, work a little differently, due to limitations of OCaml: In OCaml, we cannot type a general-purpose memo table (containing thunks with non-uniform types), nor can we examine a thunk's "arguments" (that is, the values of the variables in a closure's environment).[4]

To overcome these limitations, our implementation creates a tight coupling between namespaces and memoized functions. The function mk_mfn k f takes a name k and a function f and returns a *memoized function* mfn. The function f must have type ('arg,'res) mbody, i.e., it takes an mfn and an 'arg as arguments, and produces a 'res. (The mfn is for recursive calls; see the example below.)

Later on, we call thunk m k arg to create a thunk of type athunk from the memoized function m, with thunk name k (relative to m's namespace) and argument arg. The code for the thunk will be whatever function m was created from. In other words, in our implementation, only function calls can be memoized (not arbitrary expressions), and each set of thunks that share the same function body also share the same namespace.

Using this API, we can rewrite the map code from Section 2.3 as follows:

```
let map f map_f_name =
  let mfn = mk_mfn map_f_name (fun mfn list →
    match list with
    | Nil → Nil
    | Cons(hd, n, tl_ref) →
      let n1, n2 = fork n in
      let tl = get tl_ref in
      Cons(f hd, aref(n1, force(thunk mfn n2 tl))) )
  in fun list → call mfn list
```

The code above differs from the earlier version in that the programmer uses mk_mfn with the name map_f_name to create a memo table in a fresh namespace. Moreover, memoization happens directly on the recursive call, by introducing a thunk (and immediately forcing it).

5.2 Implementing Reuse

Much of the implementation of NOMINAL ADAPTON remains unchanged from classic ADAPTON. Specifically, both systems use DCGs to represent dependency information among nodes representing thunks and refs, and both systems traverse their DCGs to dirty dependencies and to later reuse (and repair) partially inconsistent graph components. These steps were described in Sections 2 and 4, and were detailed further in Hammer et al. (2014).

[4] Recall from the start of the previous section that memoized calls are implemented as thunks in NOMINAL ADAPTON.

The key differences between NOMINAL ADAPTON and Classic ADAPTON have to do with memo tables and thunks.

Memo Tables and Thunks. NOMINAL ADAPTON memo tables are implemented as maps from names to DCG nodes, which contain the thunk they represent. When creating memo tables with mk_mfn, the programmer supplies a name and an mbody;. Using the name, the library checks for an existing table (i.e., a namespace). If none exists, it creates an empty table, registers it globally, and returns it as an mfn. If a table exists, then the library checks that the given mbody is (physically) equal to the mbody component of the existing mfn; it issues a run-time error if not.

When the program invokes thunk, it provides an mfn, name, and argument. The library checks the mfn's memo table for an existing node with the provided name. If none exists, it registers a fresh node with the given name in the memo table and adds an allocation edge to it from the current node (which is set whenever a thunk is forced).

If a node with the same name already exists, the library checks whether the argument is equal to the current one. If equal, then the thunk previously associated with the name is the same as the new thunk, so the library reuses the node, returning it as an athunk and adding an allocation edge to the DCG. If not equal, then the name has been allocated for a different thunk either in a prior run, or in *this run*. The latter case is an error that we detect and signal. To distinguish these two cases, we traverse the current *force stack*, which consists of those nodes being forced. For each such node, we check if any of their outgoing edges is dirty, which indicates the double-use error. Since the force stack is typically shallow, performing this check is relatively inexpensive. In the former case, the library needs to reset the state associated with the name: It clears any prior cached result, dirties any incoming edges (transitively), mutates the argument stored in the node to be the new one, and adds an allocation edge. Later, when and if this thunk is forced, the system will run it. Further, because of the dirtying traversal, any nodes that (transitively or directly) forced this changed node are also candidates for reevaluation.

Names. A name in NOMINAL ADAPTON is implemented as a kind of list, as follows:

```
type name = Bullet | One of int * name | Two of int * name
```

Ignoring the int part, this is a direct implementation of λ_{NomA}'s notion of names. The int part is a hash of the next element in the list (but not beyond it), to speed up disequality checks—if two One or Two elements do not share the same hash field they cannot be equal; if they do, we must compare their tails (because of hash collisions). Thus, at worst, establishing equality is linear in the length of the name, but we can short circuit a full traversal in many cases. We note that in our applications, the size of names is either a constant, or it is proportional to the *depth* (not total size) of the DCG,

which is usually sublinear (e.g., logarithmic) in the current input's total size.

6. Experimental Results

This section evaluates NOMINAL ADAPTON's performance against ADAPTON and from-scratch recomputation.[5] We find that NOMINAL ADAPTON is nearly always faster than ADAPTON, which is sometimes *orders of magnitude slower* than from-scratch computation. NOMINAL ADAPTON always enjoys speedups, and sometimes very dramatic ones (up to 10900×).

6.1 Experimental Setup

Our experiments measure the time taken to recompute the output of a program after a change to the input, for a variety of different sorts of changes. We compare NOMINAL ADAPTON against classic ADAPTON and from-scratch computation on the changed input; the latter avoids all IC-related overhead and therefore represents the best from-scratch time possible.

We evaluate two kinds of subject programs. The first set is drawn from the IC literature on SAC and Adapton (Hammer et al. 2011, 2009, 2014). These consist of standard list processing programs (eager and lazy) filter, (eager and lazy) map, reduce(min), reduce(sum), reverse, median, and a list-based mergesort algorithm, each operating over randomly generated lists. These aim to represent key primitives that are likely to arise in standard functional programs, and use the patterns discussed in Section 3.1. We also consider an implementation of quickhull (Barber et al. 1996), a divide-and-conquer method for computing the convex hull of a set of points in a plane. Convex hull has a number of applications including pattern recognition, abstract interpretation, computational geometry, and statistics.

We also evaluate an incremental IMP interpreter, as discussed in Section 3.3, measuring its performance on a variety of different IMP programs. fact iteratively computes the factorial of an integer. intlog;fact evaluates the sequence of computing an integer logarithm followed by factorial. array max allocates, initializes, and destructively computes the maximum value in an array. matrix mult allocates, initializes, and multiplies two square matrices (implemented as arrays of arrays of integers). These IMP programs exhibit imperative behavior not otherwise incrementalizable, except as programs evaluated by a purely functional, big-step interpreter implemented in an incremental meta-language.

All programs are compiled using OCaml 4.01.0 and run on an 8-core, 2.26 GHz Intel Mac Pro with 16 GB of RAM running Mac OS X 10.6.8.

[5] Hammer et al. (2014) report that for interactive, lazy usage patterns, ADAPTON substantially outperforms another state-of-the-art incremental technique, *self-adjusting computation* (SAC), which sometimes can incur significant slowdowns. We do not compare directly against SAC here.

Batch-mode comparison (*"demand all"*)					
Program	n	**Edit**	FS (ms)	A (×)	NA (×)
eager filter	1e4	insert	21	0.178	1.29
		delete	21	0.257	1.39
		replace	21	0.108	1.27
eager map	1e4	insert	21.6	0.0803	1.02
		delete	21.6	0.0920	1.01
		replace	21.6	0.0841	1.09
min	1e5	insert	424	2790	2980
		delete	424	4450	4720
		replace	424	1850	2310
sum	1e5	insert	421	785	833
		delete	421	1140	1230
		replace	421	727	733
reverse	1e5	insert	197	0.0404	1.23
		delete	197	0.764	1.19
		replace	197	0.0404	1.23
median	1e4	insert	3010	0.747	127
		delete	3010	192	115
		replace	3010	0.755	148
mergesort	1e4	insert	267	0.212	12.0
		delete	267	11.0	10.1
		replace	267	0.205	10.5
quickhull	1e4	insert	853	0.0256*	3.78
		delete	853	0.0270*	4.11
		replace	853	0.0378*	3.86

(a) Speedups of batch-mode experiments.

Demand-driven comparison (*"demand one"*)					
Program	n	**Edit**	FS (ms)	A (×)	NA (×)
lazy filter	1e5	insert	0.016	3.79	3.55
		delete	0.016	18.1	16.3
		replace	0.016	3.55	3.20
lazy map	1e5	insert	0.016	4.08	3.79
		delete	0.016	18.1	20.4
		replace	0.016	3.71	3.62
reverse	1e5	insert	188	0.067	2130
		delete	188	50.8	4540
		replace	188	0.068	2360
mergesort	1e4	insert	63.4	96.3	369
		delete	63.4	111	752
		replace	63.4	86.2	336
quickhull	1e4	insert	509	0.0628*	5.30
		delete	509	0.0571*	5.52
		replace	509	0.0856*	5.23

(b) Speedups of demand-driven experiments.

Table 1: List benchmarks.

6.2 List-based experiments

Table 1 contains the results of our list experiments. For each program (leftmost column), we consider a randomly generated input of size n and three kinds of edits to it: *insert*, *delete*, and *replace*. For the first, we insert an element in the list; for the second, we delete the inserted element; for the last, we delete an element and then re-insert an element with a new value. Rather than consider only one edit position, we consider ten positions in the input list, spaced evenly (1/10 through the list, 2/10 through the list, etc.), and perform the edit at those positions, computing the average time across

all ten edits. We report the median of seven trials of this experiment.

The table reports the time to perform recomputation from scratch, in milliseconds, in column FS, and then the speed-up (or slow-down) factor compared to the from-scratch time for both ADAPTON, in column A, and NOMINAL ADAPTON, in column NA. Table 1a considers the case when *all* of the program's output is demanded, whereas Table 1b considers the case when only one element of the output is demanded, thus measuring the benefits of both nominal and classic ADAPTON in a lazy setting. (Note that in the lazy setting, FS sometimes also avoids complete recomputation.)

Results: Demand all. Table 1a focuses on benchmarks where *all* of the output is demanded, or when there is only a single output value (sum and minimum). In these cases, several patterns emerge in the results.

First, for eager map (Section 2) and eager filter, NOMINAL ADAPTON gets modest speedup and breaks even, respectively, while ADAPTON gets slowdowns of one to two orders of magnitude. As Section 2 explains, ADAPTON recomputes and reallocates a linear number of output elements for each $O(1)$ input change (insertion, deletion or replacement). By contrast, NOMINAL ADAPTON need not rebuild the prefix of the output lists.

Next, the benchmarks minimum and sum use the probabilistically balanced trees from Section 3.1 to do an incremental fold where in expectation, only a logarithmic number of intermediate computations are affected by a small change. Due to this construction, both ADAPTON and NOMINAL ADAPTON get large speedups over from-scratch computation (up to 4720×). NOMINAL ADAPTON tends to get slightly more speedup, since its use of names leads to less tree rebuilding. This is similar to, but not as asymptotically deep as, the eager map example ($O(\log n)$ here versus $O(n)$ above).

The next four benchmarks (reverse, mergesort, median, quickhull) show marked contrasts between the times for NOMINAL ADAPTON and ADAPTON: In all cases, NOMINAL ADAPTON gets a speedup (from about 4× to 148×), whereas ADAPTON nearly always gets a slowdown. Two exceptions are the deletion changes that revert a prior insertion. In these cases, ADAPTON reuses the original cache information that it duplicates (at great expense) after the insertion. ADAPTON gets no speedup for quickhull, our most complex benchmark in this table. By contrast, NOMINAL ADAPTON performs updates *orders of magnitude* faster than ADAPTON and gets a speedup over from-scratch; the stars ($*$) indicate that we ran quickhull at one tenth of the listed input size for ADAPTON, because otherwise it used too much memory due to having large memo tables but little reuse from them.

Results: Demand one. Table 1b focuses on benchmarks where *one* (of many possible) outputs are demanded. In these cases, two patterns emerge. First, on simple lazy list benchmarks map and filter, ADAPTON and NOMINAL ADAPTON perform roughly the same, with ADAPTON getting slightly higher speed-ups than ADAPTON. These cases are good fits for ADAPTON's model, and names only add overhead.

Second, on more involved list benchmarks (reverse, mergesort and quickhull), NOMINAL ADAPTON delivers greater speedups (from 5× to 4540×) than ADAPTON, which often delivers slowdowns. Two exceptions are mergesort, where ADAPTON delivers speedups, but is still up to 6.7× slower than NOMINAL ADAPTON, and the delete changes, which as in the table above are fast because of spurious duplication in the insert change.

In sum, NOMINAL ADAPTON consistently delivers speedups for small changes, while ADAPTON does so to a lesser extent, and much less reliably.

Batch-mode comparison (*"demand all"*)					
Program	n	**Edit**	FS (ms)	A (×)	NA (×)
fact	5e3	repl	945	0.520	10900
		swap1	947	2410	4740
		swap2	955	4740	6590
		ext	847	0.464	0.926
intlog;fact	2^{30}, 5e3	swap	849	0.413	3.18
array	2^{10}	repl1	191	0.323	6.52
max		repl2	191	0.310	7.62
matrix	20x20	swap1	4500	0.617	1.31
mult		swap2	4500	0.756	1.17
	25x25	ext	6100	1.50	1.55

Table 2: Speedups of IMP interpreter experiments.

6.3 Interpreter Experiments

We tested the incremental behavior of the IMP interpreter with three basic forms of edits to the input programs: replacing values (*replace*), swapping sub-expressions (*swap*), and increasing the size of the input (*ext*). These experiments all take the following form: evaluate an expression, mutate the expression, then reevaluate.

- For fact, *repl* mutates the value of an unused variable; *swap1* reverses the order of two assignments at the start of the program; *swap2* reverses the order of two assignments at the end; and *ext* increases the size of the input.
- For intlog;fact, *swap* swaps the two subprograms.
- For array max, *repl1* replaces a value at the start of the array and *repl2* moves a value from the start to the end of the array.
- For matrix mult, *swap1* reverses the order of the initial assignments of the outer arrays of the input matrices; *swap2* reverses the order of the while loops that initialize the inner arrays of the input matrices; and finally *ext* extends the dimensions of the input arrays.

Results. Table 2 summarizes the results, presented the same way as the list benchmarks. We can see that NOMINAL ADAPTON provides a speedup over from-scratch computation in all but one case, and can provide dramatic speedups. In addition, NOMINAL ADAPTON consistently outperforms classic ADAPTON, in some cases providing a speedup where ADAPTON incurs a (sizeable) slowdown.

The fact program's *repl* experiment shows significant performance improvement due to names. Classic ADAPTON dirties each intermediary environment and is forced to recompute. With the naming strategy outlined in Section 3.3, the environment is identified without regard to the particular values inserted. Future computations that depend on the environment, but not the changed value in particular, are reused. The fact *swap* experiments show significant speedup for both classic ADAPTON and NOMINAL ADAPTON, because the trie map representation remains unchanged regardless of order of the assignments.

The remaining results fall into two categories. The edits made to intlog;fact, array max, and matrix mult's *swap*s show speedups between 1.17-7.62× with NOMINAL ADAPTON while classic ADAPTON exhibits a slowdown, due to spending much of its time creating and evaluating new nodes in the DCG. NOMINAL ADAPTON, on the other hand, spends its time walking the already-present nodes and reusing many (from 25% to as much as 99%) of them, with the added benefit of far better memory performance.

The last category includes the *ext* tests for fact (increasing the input value) and matrix mult (extending the dimensions of the input matrices). Such changes have pervasive effects on the rest of the computation and are a challenge to incremental reuse. Extension for matrix multiplication shows a modest speedup over the from-scratch running time for both nominal and classic ADAPTON. NOMINAL ADAPTON is able to reuse a third of the nodes created during the original run, while classic ADAPTON is not able to reuse any. Increasing the value of the input to factorial causes similar behavior, though the single, short loop prevents the amount of reuse to overcome the from-scratch time.

7. Related Work

Here we survey past approaches to incremental computation, organizing our discussion into three categories: static approaches, dynamic approaches, and specialized approaches.

Static approaches to IC. These approaches transform programs to derive a second program that can process "deltas"; the derived program takes as input the last (full) output and the representation of an input change, and produces (the representation of) the next output change. This program derivation is performed a priori, before any dynamic changes are issued. As such, static approaches have the advantage of not requiring dynamic space or time overhead, but also carry disadvantages that stem from not being dynamic in nature: They cannot handle programs with general recursion, and cannot take advantage of cached intermediate results, since by design, there are none (Cai et al. 2014; Liu and Teitelbaum 1995).

Dynamic approaches to IC. In contrast to static approaches, dynamic approaches attempt to trade space for time savings. A variety of dynamic approaches to IC have been proposed. Most early approaches fall into one of two camps: they either perform function caching of pure programs (Bellman 1957; McCarthy 1963; Michie 1968; Pugh 1988), or they support input mutation and employ some form of dynamic dependency graphs. However, the programming model advanced by earlier work on dependence graphs lacked features like general recursion and dynamic allocation, instead restricting programs to those expressible as *attribute grammars* (a language of declarative constraints over tree structures) (Demers et al. 1981; Reps 1982a,b; Vogt et al. 1991).

Some recent general-purpose approaches to dynamic IC (SAC and ADAPTON) support general-purpose input structures and general recursion; internally, they use a notion of memoization to find and reuse portions of existing dependency graphs. As described in Section 1, SAC and ADAPTON differ greatly in the programming model they support (SAC is eager/batch-oriented whereas ADAPTON is demand-driven) and in how they represent dependency graphs. Consequently, they have different performance characteristics, with ADAPTON excelling at demand-driven and interactive settings, and SAC doing better in non-interactive, batch-oriented settings (Hammer et al. 2014).

As mentioned in Section 1, the presence of dynamic memory allocation in SAC poses a reuse problem due to "fresh" object identities, and thus benefits from a mechanism to deterministically match up identities from prior runs. Various past work on SAC addresses this problem in some form (Acar and Ley-Wild 2009; Acar et al. 2004, 2006; Hammer and Acar 2008) describing how to use "hints" or "keys." The reuse problem in NOMINAL ADAPTON is more general in nature than in SAC, and thus requires a very different solution. For example, NOMINAL ADAPTON's DCG and more general memo tables do not impose SAC's total ordering of events, admitting more opportunities for reuse, but complicating the issue of assuring that names are not used more than once within a run. The use of thunks, which also need names, adds a further layer of complication. This paper addresses name reuse in this (more general) IC setting. Further, we address other naming issues, such as how to generate new names from existing ones (via fork) and how to determinize memo table creation (via named namespaces).

Though they do not attempt to address name-reuse issues, the recent work of Chen et al. (2014) reduces the (often large) space overhead of dependence tracking, which is pervasive in both SAC and in ADAPTON. In particular, they propose coarsening the granularity of dependence tracking, and they report massive improvements (orders of magnitude) in space savings as a result. We believe that their approach ("probabilistic chunking") is immediately applicable to our setting, as well as to classic ADAPTON.

Specialized approaches to IC. Some recent approaches to IC are not general-purpose, but exploit domain-specific structure to handle input changes efficiently. DITTO incrementally checks invariants in Java programs (but is limited to

invariant checking) (Shankar and Bodík 2007). i3QL incrementally repairs database views (queries) when the underlying data changes due to insertions and removals of table rows (Mitschke et al. 2014).

Finally, reactive programming (especially functional reactive programming or FRP) shares some elements with incremental computation: both paradigms offer programming models for systems that strive to efficiently react to "outside changes"; internally, they use graph representations to model dependencies in a program that change over time (Cooper and Krishnamurthi 2006; Czaplicki and Chong 2013; Krishnaswami and Benton 2011). However, the chief aim of FRP is to provide a declarative means of specifying programs whose values are time-dependent (stored in signals), whereas the chief aim of IC is to provide time savings for small input changes (stored in special references). The different scope and and programming model of FRP makes it hard to imagine using it to write an efficient incremental sorting algorithm, though it may be possible. On the other hand, IC would seem to be an appropriate mechanism for implementing an FRP engine, though the exact nature of this connection remains unclear.

8. Conclusion

We presented NOMINAL ADAPTON, a general-purpose system for incremental computation in which programmers may use *names* and *namespaces* to achieve better incremental performance than the typical structural approach. We have formalized NOMINAL ADAPTON's algorithms and proved them correct. We have implemented a variety of data structures and benchmark programs in NOMINAL ADAPTON, and performance experiments show that compared to class ADAPTON, which lacks names, NOMINAL ADAPTON enjoys uniformly better performance, sometimes achieving dramatic, orders-of-magnitude speedups over from-scratch computation when ADAPTON would suffer significant slowdowns.

References

U. A. Acar and R. Ley-Wild. Self-adjusting computation with Delta ML. In *Advanced Functional Programming*. 2009.

U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive memoization. Technical Report CMU-CS-03-208, Carnegie Mellon University, Nov. 2004.

U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. A library for self-adjusting computation. *ENTCS*, 148 (2), 2006.

C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4), Dec. 1996.

R. Bellman. *Dynamic Programming*. Princeton Univ. Press, 1957.

Y. Cai, P. G. Giarrusso, T. Rendel, and K. Ostermann. A theory of changes for higher-order languages: Incrementalizing λ-calculi by static differentiation. In *PLDI*, 2014.

Y. Chen, U. A. Acar, and K. Tangwongsan. Functional programming for dynamic and large data with self-adjusting computation. In *ICFP*, 2014.

G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *ESOP*, 2006.

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

A. Demers, T. Reps, and T. Teitelbaum. Incremental evaluation of attribute grammars with application to syntax-directed editors. In *POPL*, 1981.

M. Hammer and U. A. Acar. Memory management for self-adjusting computation. In *ISMM*, 2008.

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.

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.

M. A. Hammer, K. Y. Phang, M. Hicks, and J. S. Foster. Adapton: Composable, demand-driven incremental computation. In *PLDI*, 2014.

N. R. Krishnaswami and N. Benton. A semantic model for graphical user interfaces. In *ICFP*, 2011.

P. B. Levy. Call-by-push-value: A subsuming paradigm. In *TLCA*, 1999.

P. B. Levy. *Call-By-Push-Value*. PhD thesis, Queen Mary and Westfield College, University of London, 2001.

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

J. McCarthy. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, 1963.

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

R. Mitschke, S. Erdweg, M. Köhler, M. Mezini, and G. Salvaneschi. i3QL: Language-integrated live data views. In *OOPSLA*, 2014.

W. Pugh. *Incremental Computation via Function Caching*. PhD thesis, Cornell University, 1988.

W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *POPL*, 1989.

T. Reps. *Generating Language-Based Environments*. PhD thesis, Cornell University, Aug. 1982a.

T. Reps. Optimal-time incremental semantic analysis for syntax-directed editors. In *POPL*, 1982b.

A. Shankar and R. Bodík. DITTO: Automatic incrementalization of data structure invariant checks (in Java). In *PLDI*, 2007.

H. Vogt, D. Swierstra, and M. Kuiper. Efficient incremental evaluation of higher order attribute grammars. In *PLILP*, 1991.

G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

Supplement to "Incremental Computation with Names"

This supplementary material contains additional details about how NOMINAL ADAPTON manages space, in Appendix A. The full set of incremental evaluation rules, along with a reference semantics of non-incremental rules, appears in Appendix B. Our full from-scratch consistency result, along with the definitions and lemmas it uses, appears in Appendix C.

A. Space management

In a long-running program, memo tables could grow without bound. NOMINAL ADAPTON helps reduce table sizes, as we have already seen, but we still need a mechanism to clean out the tables when space becomes limited.

A natural idea is to implement a memo table's mapping from name to DCG node with a *weak reference*, so that if the table is the only reference to the node, the node can be garbage collected when the system is short on space. This is not quite enough, though, because to implement dirtying, DCG edges are bidirectional. To see why, consider Figure **??**(a). When step (1a) modifies the list node, it needs a way to locate the incoming edge it dirties in (1b). Alloc and force edges, like those from $\alpha{\cdot}1$ to $\beta{\cdot}1$, are also bidirectional. So even after the list node $\alpha\,a$ is deleted, the thunk $\alpha{\cdot}1$ is alive because $\beta{\cdot}1$ is alive. To solve this problem, we must also make back edges weak.

Unfortunately, this approach (implemented in the original ADAPTON) is unsound. ADAPTON supports an interactive pattern called *swapping*, which allows an end user to change some input elements but later restore them (e.g., when exploring "what-if" scenarios in a spreadsheet). In the example, it would be reasonable for the user to restore $\alpha\,a$ after having removed it. When this happens (and the output is re-demanded) the memo table entry for the thunk at $\alpha{\cdot}1$ can actually be safely reused. Pathologically, during the time that $\alpha\,a$ is removed from the list (and $\alpha{\cdot}1$ is not pointed to by root) the garbage collector could remove *some* of the DCG, but not all of it. For example, it could null out the weak back edges from $\beta{\cdot}1$ to $\alpha{\cdot}1$ because the $\alpha{\cdot}1$ thunk is only reachable by weak references. But if the node is then restored to the list, and $\beta{\cdot}1$ is subsequently changed, the back edge will be gone, and we will not mark $\alpha{\cdot}1$ as inconsistent.

To fix the GC problem, we still use weak references for back edges, but use strong references for memo table entries, so that from the GC's point of view, all DCG nodes are always reachable. To implement safe space reclamation, we also implement reference counting of DCG nodes, where the counts reflect the number of strong edges reaching a node. When DCG edges are deleted, the reference counts of target nodes are decremented. Nodes that reach zero are not immediately collected; this allows thunks to be "resurrected" by the swapping pattern. Instead, we provide a flush operation for memo tables that deletes the strong mapping edge for all nodes with a count of zero, which means they are no longer reachable by the main program, as is the case for the $\alpha{\cdot}1$ thunk in Figure **??**(a) after the change. Deletion is transitive: removing the node decrements the counts of nodes it points to, which may cause them to be deleted.

An interesting question is how to decide when to invoke flush; this is the system's *eviction policy*. One obvious choice is to flush when the system starts to run short of memory (based on a signal from the GC), which matches the intended effect of the unsound weak reference-based approach. But placing the eviction policy under the program's control opens other possibilities, e.g., the programmer could invoke flush when it is semantically clear that thunks cannot be restored. We leave to future work a further exploration of sensible eviction policies.

B. Theory of λ_{NomA}

B.1 Syntax

Figures 6 and 7 reprise Figure 4, with descriptions for easy reference. Here, we briefly explain the standard constructs from call-by-push-value, which weren't discussed in Section 4.

Call-by-push-value. CBPV distinguishes values, results (*terminal computations*), and computations. A computation e can be turned into a value by thunking it via $\mathbf{thunk}(\nu, e)$ (the first argument, the name ν, is peculiar to Nominal Adapton; ordinary CBPV does not explicitly name thunks). Conversely, a value v can play the role of a (terminal) computation t via $\mathbf{ret}\,v$; terminal computations (called results in our paper) t are a subclass of computations e.

Standard constructs. In addition to CBPV-specific machinery (thunks, **force**, **ret**), we have several standard λ-calculus constructs:

- Functions $\lambda x.e$ are terminal computations; $e\,v$ evaluates e to a function $\lambda x.e'$ and substitutes v for e'. Note that the function argument in $e\,v$ is a value, not a computation.
- Let-expressions $\mathbf{let}\,x \leftarrow e_1\,\mathbf{in}\,e_2$ evaluate the computation e_1 first. The usual λ-calculus application $e_1\,e_2$ can be simulated by $\mathbf{let}\,x \leftarrow e_2\,\mathbf{in}\,(e_1\,x)$.
- Fixed points $\mathbf{fix}\,f.e$ are computations, and so are fixed point variables f.

- Given an injection into a disjoint union, $\mathbf{inj}_i\ v$, the **case** computation form eliminates the sum and computes the corresponding e_i branch, with v substituted for x_i.
- Given a pair (v_1, v_2), the computation $\mathbf{split}\,(v, x_1.x_2.e)$ computes e, first substituting v_1 for x_1, and v_2 for x_2.

Further reading. For more on the (non-nominal, non-incremental) formulations of CBPV, including discussion of CBPV's *value types* and *computation types*, see Levy (1999, 2001).

B.2 Evaluation rules

Figure 8 shows two systems of evaluation rules: an *incremental system* and a *non-incremental* or *reference* system. The incremental system models Nominal Adapton programs that transform a graph whose nodes are store locations (values and thunks) and whose edges represent dependencies (an edge from p to q means that q depends on p); the reference system models call-by-push-value programs under a plain store (a graph with no edges).

Rules above the double horizontal line in Figure 8 do not manipulate the graph, and are present in both the incremental and reference systems. The shaded rules, to the left of vertical double lines, are non-incremental rules that never create edges and do not cache results. The rules to their right create edges, store cached results, and recompute results that have become invalid.

Shared rules. Some of the rules at the top of Figure 8 are derived from standard CBPV rules:

- Given a terminal computation (such as $\mathbf{ret}\,v$), Eval-term returns it along with the input graph.
- Eval-app evaluates a function e_1 to get a terminal computation $\lambda x.e_2$ and substitutes the argument v for x, threading through the graph: evaluating e_1 produces G_2, which is given as input to the second premise, resulting in output graph G_3.
- Eval-fix substitutes the fixed-point computation for its own variable.
- Eval-bind follows the pattern of Eval-app, replacing x with a value v from the terminal computation $\mathbf{ret}\,v$.
- Eval-case substitutes the value v injected into a sum for the pattern variable x_i (where $i \in \{1, 2\}$).
- Eval-split decomposes a pair (v_1, v_2) and substitutes each v_i for x_i.

The last three shared rules are not standard: they deal with names and namespaces.

Pointers	$p, q ::=$	$\mathsf{root} \mid k@\omega$
Names	$k ::=$	$\bullet \mid k{\cdot}1 \mid k{\cdot}2$
Namespaces	$\omega, \mu ::=$	$\top \mid \omega.k$

Values

$v ::= x \mid (v_1, v_2)$	
$\mid \mathbf{inj}_i\ v$	
$\mid \mathbf{nm}\ k$	name
$\mid \mathbf{ref}\ p$	pointer to value (reference cell)
$\mid \mathbf{thk}\ p$	pointer to computation
$\mid \mathbf{ns}\ \omega$	namespace identifier

Results

$t ::= \lambda x.e \mid \mathbf{ret}\,v$	

Computations

$e ::= t \mid e\,v \mid f \mid \mathbf{fix}\,f.e$	
$\mid \mathbf{let}\,x \leftarrow e_1\ \mathbf{in}\ e_2$	
$\mid \mathbf{case}\,(v, x_1.e_1, x_2.e_2)$	
$\mid \mathbf{split}\,(v, x_1.x_2.e)$	
$\mid \mathbf{thunk}(v, e)$	create thunk e at name v
$\mid \mathbf{force}\,(v)$	force thunk that v points to
$\mid \mathbf{fork}(v)$	fork name v into two halves $v{\cdot}1$, $v{\cdot}2$
$\mid \mathbf{ref}(v_1, v_2)$	allocate v_2 at name v_1
$\mid \mathbf{get}\,(v)$	get value stored at pointer v
$\mid \mathbf{ns}\,(v, x.e)$	bind x to a new namespace v
$\mid \mathbf{nest}(v, e_1, x.e_2)$	evaluate e in namespace v and bind x to the result

Figure 6: Syntax

Graphs

$G, H ::=$	ε	empty graph
	$\mid G, p{:}v$	p points to value v
	$\mid G, p{:}e$	p points to thunk e (with no cached result)
	$\mid G, p{:}(e, t)$	p points to thunk e with cached result t
	$\mid G, (p, a, b, q)$	p depends on q due to action a, with status b

Edge actions

$a ::=$	For edge (p, a, b, q), the computation at p...
$\mathbf{alloc}\,v$	...created reference v at q
$\mid \mathbf{alloc}\,e$	...created thunk e at q
$\mid \mathbf{obs}\,v$	...read q's value, which was v
$\mid \mathbf{obs}\,t$	...forced thunk q, which returned t

Edge statuses

$b ::= \mathsf{clean}$	value or computation at sink is out of date
$\mid \mathsf{dirty}$	value or computation at sink is up to date

Figure 7: Graphs

- Eval-fork splits a name k into children $k{\cdot}1$ and $k{\cdot}2$. Once forked, the name k should not be used to allocate a new reference or thunk, nor should it be forked again.
- Running in namespace ω, Eval-namespace makes a new namespace $\omega.k$ and substitutes it for x in the body e.
- Running in namespace ω, Eval-nest runs e_1 in a different namespace μ and then returns to ω to run e_2, with x replaced by the result of running e_1.

Non-incremental rules. These rules cover allocation and use of references and thunks. Like the incremental rules (discussed below), they use names and namespaces; however, they do not cache the results of thunks. We discuss the non-incremental rules first, because they are simpler and provide a kind of skeleton for the incremental rules.

- Eval-refPlain checks that the pointer described by $q = k@\omega$ is fresh ($q \notin \mathsf{dom}(G_1)$), adds a node q with contents v to the graph ($G_1\{q{\mapsto}v\} = G_2$), and returns a reference **ref** q.
- Eval-thunkPlain is similar to Eval-refPlain, but creates a node with a suspended computation e instead of a value.
- Eval-getPlain returns the contents of the pointer q.
- Eval-forcePlain extracts the computation stored in a thunk ($G_1(q) = e$) and evaluates it under the namespace of q (that is, if $q = k@\mu$, it evaluates it under μ).

Incremental rules. Each non-incremental rule corresponds to one or more incremental rules: the incremental semantics is influenced by the graph edges, which are not present in the non-incremental system. For example, Eval-refPlain is replaced by Eval-refDirty and Eval-refClean.

These rules use some predicates and operations, such as dirty-paths-in, that we explain informally as we describe the rules; they are fully defined in Figure 9.

- Like Eval-refPlain, Eval-refDirty creates a node q with value v: $G_1\{q{\mapsto}v\} = G_2$. Unlike Eval-refPlain, Eval-refDirty does not check that q is not in the graph: if we are recomputing, q may already exist. So $G_1\{q{\mapsto}v\} = G_2$ either creates q pointing to v, or updates q by replacing its value with v. It then marks the edges along all paths into q as dirty $\mathsf{dirty\text{-}paths\text{-}in}(G_2, q) = G_3$; these are the paths from nodes that depend on q.
- Eval-refClean can be applied only during recomputation, and only when $G(q) = v$. That is, we are evaluating **ref**(**nm** k, v) and allocating the same value as the previous run. Since the values are the same, we need not mark any dependency edges as dirty, but we do add an edge to remember that p depends on q.
- Eval-thunkDirty corresponds exactly to Eval-refDirty, but for thunks rather than values.
- Eval-thunkClean corresponds to Eval-refClean and does not change the contents of q. Note that the condition $(\mathsf{exp}(G, q)) = e$ applies whether or not q includes a cached result. If a cached result is present, that is, $G(q) = (e, t)$, it remains in the output graph.
- Eval-getClean is the same as Eval-getPlain, except that it adds an edge representing the dependency created by reading the contents of q.

The last three incremental rules, which in some sense collectively correspond to Eval-forcePlain, are the most interesting: they model evaluation and reevaluation of thunked computations. Forcing a thunk that has not been previously computed, an operation that involves one rule in the non-incremental system (Eval-forcePlain), involves at least two rules in the incremental system: Eval-computeDep and Eval-forceClean.

- Eval-computeDep applies when $e =$ **force** (**thk** p_0). Here, we select some node q (in practice, a node that p_0 depends on) whose contents are e'.

  Then we add e' to G_1 with $G_1\{q{\mapsto}e'\}$. If G_1 already had some cached value t_0, that is, $G_1(q) = (e', t_0)$, this operation removes it.

  Next, we delete outward edges of q, resulting in a graph G_1'. We need to delete the outward edges before evaluating e' because they represent what the *previous* evaluation of e' depended on.

  After evaluating e' in $G_1' \vdash^{q}_{\mathsf{namespace}(q)} e' \Downarrow G_2; t'$, with q as the current thunk and q's namespace component as the current namespace, we move to the right-hand stack of premises. First, we cache the result t' with $G_2\{q{\mapsto}(e', t')\} = G_2'$. Second, we check that all outward edges of q are clean ($\mathsf{all\text{-}clean\text{-}out}(G_2', q)$). An outward edge represents a dependency of q on some other node, say q'. If such an edge were dirty, we would have a circular dependency: we have just computed e', but now need to recompute something e' depends on, which would in turn require computing e' again! This check should

always succeed in a correctly written Nominal Adapton program, but we cannot yet statically verify it, so the implementation does it dynamically.

In the last premise, we continue to evaluate **force** (**thk** p_0).

Note that in the case $q = p_0$—where the "dependency" being computed is p_0 itself—the last premise could be derived by Eval-forceClean, which looks up the result just computed by $G_1' \vdash^{q}_{\mathsf{namespace}(q)} e' \Downarrow G_2; t'$.

- Eval-scrubEdge replaces a dirty edge $(q_1, a, \mathsf{dirty}, q_2)$ with a clean edge $(q_1, a, \mathsf{clean}, q_2)$. First, it checks that all edges out from q_2 are clean; this means that the contents of q_2 are up-to-date. Next, it checks that the action a that represents q_1's dependency on q_2 is consistent with the contents of q_2. For example, if q_2 points to a thunk with a cached result (e_2, t_2) and $a =$ **obs** t, then the "consistent-action" premise checks that the currently cached result t_2 matches the result t that was previously used by q_1.
- Eval-forceClean returns the cached result t of a thunk, adding an edge denoting that p has observed the result of q to be t.

The rules vs. the implementation. Our rules are not intended to be an "instruction manual" for building an implementation; rather, they are intended to model our implementation. To keep the rules simple, we underspecify a number of aspects of the implementation, some of which are quite important in practice. For example, when recomputing an allocation in the case when the allocated value is equal to the previously allocated value ($G(q) = v$), either Eval-refDirty or Eval-refClean applies. However, in this situation the implementation always follows the behavior of Eval-refClean, since that is the choice that avoids unnecessarily marking edges as dirty and causing more recomputation.

More subtly, the order in which dirty subcomputations are recomputed is underspecified in the rules, as is the timing of Eval-scrubEdge, which can be applied to dirty edges with no particular connection to the p_0 mentioned in the subject expression **force** (**thk** p_0).

B.3 Graph predicates and transforms

Figure 9 specifies predicates and operations on graphs, used in the rules in Figure 8.

C. Metatheory of λ_{NomA}

C.1 Overview

Our main formal result in this paper, *from-scratch consistency*, states that given an evaluation derivation corresponding to an incremental computation, we can construct a derivation corresponding to a non-incremental computation that returns the same result and a corresponding graph. That is, the incremental computation is consistent with a computation in the simpler non-incremental system.

To properly state the consistency result, we need to define what it means for a graph to be well-formed (Section C.2), relate the incremental computation's DCG to the non-incremental computation's store (Section C.3), describe the sets of nodes read and written by an evaluation derivation (Section C.4), and prove a store weakening lemma (Section C.6). The consistency result itself is stated and proved in Section C.7.

C.2 Graph well-formedness

The judgment $G \vdash H$ wf is read "H is a well-formed subset of G". It implies that H is a linearization of a subset of G: within H, information sources appear to the left of information sinks, dependency edges point to the left, and information flows to the right. Consequently, H is a dag.

As a convenient shorthand, we write G wf for $G \vdash G$ wf, threading the entire graph G through the rules deriving $G \vdash H$ wf. The well-formedness rules in Figure 10 decompose the right-hand graph, and work as follows:

- For values (Grwf-val) and thunks with no cached result (Grwf-thunk), the rules only check the correspondence between G (the entire graph) and H (the subgraph).
- For thunks with a cached result, Grwf-thunkCache examines the outgoing edges. If they are all clean, then it checks that evaluating e *again* would not change the graph at all: $\mathcal{D} :: G \vdash^{p}_{\omega} e \Downarrow G; t$. If one or more edges are dirty, it checks that all incoming edges are dirty.
- Grwf-dirtyEdge checks that all edges flowing into a dirty edge are dirty: given edges from p to q and from q to r, where the edge from q to r is dirty (meaning that q depends on r and r needs to be recomputed), the edge from p to q should also be dirty. Otherwise, we would think we could reuse the result in p, even though p (transitively) depends on r.

$\boxed{G_1 \vdash^p_\omega e \Downarrow G_2; t}$ Under graph G_1, expression e, evaluating as part of thunk p in namespace ω, yields G_2 and t.

$$\frac{}{G \vdash^p_\omega t \Downarrow G; t}\ \textsf{Eval-term} \qquad \frac{G_1 \vdash^p_\omega e_1 \Downarrow G_2; \lambda x.e_2 \quad G_2 \vdash^p_\omega [v/x]e_2 \Downarrow G_3; t}{G_1 \vdash^p_\omega e_1\, v \Downarrow G_3; t}\ \textsf{Eval-app}$$

$$\frac{G_1 \vdash^p_\omega [(\mathbf{fix}\, f.e)/f]e \Downarrow G_2; t}{G_1 \vdash^p_\omega \mathbf{fix}\, f.e \Downarrow G_2; t}\ \textsf{Eval-fix} \qquad \frac{G_1 \vdash^p_\omega e_1 \Downarrow G_2; \mathbf{ret}\, v \quad G_2 \vdash^p_\omega [v/x]e_2 \Downarrow G_3; t}{G_1 \vdash^p_\omega \mathbf{let}\, x \leftarrow e_1\, \mathbf{in}\, e_2 \Downarrow G_3; t}\ \textsf{Eval-bind}$$

$$\frac{G_1 \vdash^p_\omega [v/x_i]e_i \Downarrow G_2; t}{G_1 \vdash^p_\omega \mathbf{case}\, (\mathbf{inj}_i\, v, x_1.e_1, x_2.e_2) \Downarrow G_2; t}\ \textsf{Eval-case} \qquad \frac{G_1 \vdash^p_\omega [v_1/x_1][v_2/x_2]e \Downarrow G_2; t}{G_1 \vdash^p_\omega \mathbf{split}\, ((v_1, v_2), x_1.x_2.e) \Downarrow G_2; t}\ \textsf{Eval-split}$$

$$\frac{}{G \vdash^p_\omega \mathbf{fork}(\mathbf{nm}\, k) \Downarrow G; \mathbf{ret}\, (\mathbf{nm}\, k{\cdot}1, \mathbf{nm}\, k{\cdot}2)}\ \textsf{Eval-fork}$$

$$\frac{G_1 \vdash^p_\omega [\mathbf{ns}\, \omega.k/x]e \Downarrow G_2; t}{G_1 \vdash^p_\omega \mathbf{ns}\, (\mathbf{nm}\, k, x.e) \Downarrow G_2; t}\ \textsf{Eval-namespace} \qquad \frac{G_1 \vdash^p_\mu e_1 \Downarrow G_2; \mathbf{ret}\, v \qquad G_2 \vdash^p_\omega [v/x]e_2 \Downarrow G_3; t}{G_1 \vdash^p_\omega \mathbf{nest}(\mathbf{ns}\, \mu, e_1, x.e_2) \Downarrow G_3; t}\ \textsf{Eval-nest}$$

$$\frac{q = k@\omega \qquad q \notin \mathrm{dom}(G_1) \qquad G_1\{q \mapsto v\} = G_2}{G_1 \vdash^p_\omega \mathbf{ref}(\mathbf{nm}\, k, v) \Downarrow G_2; \mathbf{ret}\, \mathbf{ref}\, q}\ \textsf{Eval-refPlain}$$

$$\frac{q = k@\omega \qquad G_1\{q \mapsto v\} = G_2 \qquad \textsf{dirty-paths-in}(G_2, q) = G_3}{G_1 \vdash^p_\omega \mathbf{ref}(\mathbf{nm}\, k, v) \Downarrow G_3, (p, \mathbf{alloc}\, v, \mathrm{clean}, q); \mathbf{ret}\, \mathbf{ref}\, q}\ \textsf{Eval-refDirty}$$

$$\frac{q = k@\omega \qquad G(q) = v}{G \vdash^p_\omega \mathbf{ref}(\mathbf{nm}\, k, v) \Downarrow G, (p, \mathbf{alloc}\, v, \mathrm{clean}, q); \mathbf{ret}\, \mathbf{ref}\, q}\ \textsf{Eval-refClean}$$

$$\frac{q = k@\omega \qquad q \notin \mathrm{dom}(G_1) \qquad G_1\{q \mapsto e\} = G_2}{G_1 \vdash^p_\omega \mathbf{thunk}(\mathbf{nm}\, k, e) \Downarrow G_2; \mathbf{ret}\, (\mathbf{thk}\, q)}\ \textsf{Eval-thunkPlain}$$

$$\frac{q = k@\omega \qquad G_1\{q \mapsto e\} = G_2 \qquad \textsf{dirty-paths-in}(G_2, q) = G_3}{G_1 \vdash^p_\omega \mathbf{thunk}(\mathbf{nm}\, k, e) \Downarrow G_3, (p, \mathbf{alloc}\, e, \mathrm{clean}, q); \mathbf{ret}\, (\mathbf{thk}\, q)}\ \textsf{Eval-thunkDirty}$$

$$\frac{q = k@\omega \qquad \exp(G, q) = e}{G \vdash^p_\omega \mathbf{thunk}(\mathbf{nm}\, k, e) \Downarrow G, (p, \mathbf{alloc}\, e, \mathrm{clean}, q); \mathbf{ret}\, (\mathbf{thk}\, q)}\ \textsf{Eval-thunkClean}$$

$$\frac{G(q) = v}{G \vdash^p_\omega \mathbf{get}\, (\mathbf{ref}\, q) \Downarrow G; \mathbf{ret}\, v}\ \textsf{Eval-getPlain} \qquad \frac{G(q) = v}{G \vdash^p_\omega \mathbf{get}\, (\mathbf{ref}\, q) \Downarrow G, (p, \mathbf{obs}\, v, \mathrm{clean}, q); \mathbf{ret}\, v}\ \textsf{Eval-getClean}$$

$$\frac{G_1(q) = e \qquad G_1 \vdash^q_{\mathsf{namespace}(q)} e \Downarrow G_2; t}{G_1 \vdash^p_\omega \mathbf{force}\, (\mathbf{thk}\, q) \Downarrow G_2; t}\ \textsf{Eval-forcePlain}$$

$$\frac{G(q) = (e, t) \qquad \textsf{all-clean-out}(G, q)}{G \vdash^p_\omega \mathbf{force}\, (\mathbf{thk}\, q) \Downarrow G, (p, \mathbf{obs}\, t, \mathrm{clean}, q); t}\ \textsf{Eval-forceClean}$$

$$\frac{\begin{array}{l}\textsf{all-clean-out}((G_1, G_2), q_2) \\ \textsf{consistent-action}((G_1, G_2), a, q_2) \\ G_1, (q_1, a, \mathrm{clean}, q_2), G_2 \vdash^p_\omega \mathbf{force}\, (\mathbf{thk}\, p_0) \Downarrow G_3; t\end{array}}{G_1, (q_1, a, \mathrm{dirty}, q_2), G_2 \vdash^p_\omega \mathbf{force}\, (\mathbf{thk}\, p_0) \Downarrow G_3; t}\ \textsf{Eval-scrubEdge}$$

$$\frac{\begin{array}{ll}\exp(G_1, q) = e' & G_2\{q \mapsto (e', t')\} = G_2' \\ \textsf{del-edges-out}(G_1\{q \mapsto e'\}, q) = G_1' & \textsf{all-clean-out}(G_2', q) \\ G_1' \vdash^q_{\mathsf{namespace}(q)} e' \Downarrow G_2; t' & G_2' \vdash^p_\omega \mathbf{force}\, (\mathbf{thk}\, p_0) \Downarrow G_3; t\end{array}}{G_1 \vdash^p_\omega \mathbf{force}\, (\mathbf{thk}\, p_0) \Downarrow G_3; t}\ \textsf{Eval-computeDep}$$

Figure 8: Evaluation semantics of λ_{NomA}; vertical bars separate non-incremental rules (left, shaded) from incremental rules (right)

Conversely, if an edge from p to q is clean (Grwf-cleanEdge), then all edges out from q must be clean (otherwise we would contradict the "transitive dirtiness" just described). Moreover, the action a stored in the edge from p to q must be consistent with the contents of q.

C.3 From graphs to stores: the restriction function

To relate the graph associated with an incremental evaluation to the store associated with a non-incremental evaluation, we define a function $\lfloor G \rfloor_P$ that restricts G to a set of pointers P, drops cached thunk results*, and erases all edges**:

$\mathsf{exp}(G, p) = e$
  if $G(p) = e$ or $G(p) = (e, t)$

$G\{p \mapsto v\} = G'$
  where, if $p \notin \mathsf{dom}(G)$, then $G' = (G, p{:}v)$
  otherwise, if $G = (G_1, p{:}v', G_2)$ then $G' = (G_1, p{:}v, G_2)$

$G\{p \mapsto e\} = G'$
  where, if $p \notin \mathsf{dom}(G)$, then $G' = (G, p{:}e)$
  otherwise, if $G = (G_1, p{:}e', G_2)$ or $G = (G_1, p{:}(e', t'), G_2)$ then $G' = (G_1, p{:}e, G_2)$

$G\{p \mapsto (e, t)\} = G'$
  where, if $p \notin \mathsf{dom}(G)$, then $G' = (G, p{:}(e, t))$
  otherwise, if $G = (G_1, p{:}e', G_2)$ or $G = (G_1, p{:}(e', t'), G_2)$ then $G' = (G_1, p{:}(e, t), G_2)$

$\mathsf{all\text{-}clean\text{-}out}(G, p) \equiv$
  $\forall (p, a, b, q) \in G.\ (b = \mathsf{clean})$

$\big(\mathsf{del\text{-}edges\text{-}out}(G_1, p) = G_2\big) \equiv$
  $\big(\mathsf{nodes}(G_1) = \mathsf{nodes}(G_2)\big)$
  and $\forall q \neq p.\ \big((q, a, b, q') \in G_1\big) \Rightarrow \big((q, a, b, q') \in G_2\big)$
  and $\nexists a, b, q.\ (p, a, b, q) \in G_2$

$\big(\mathsf{dirty\text{-}paths\text{-}in}(G_1, p) = G_2\big) \equiv$
  $\big(\mathsf{nodes}(G_1) = \mathsf{nodes}(G_2)\big)$
  and $\forall q_1, q_2.\ \big((q_1, a, b', q_2) \in G_2\big) \Rightarrow \big((q_1, a, b, q_2) \in G_1\big)$
  and $\forall q_1, q_2.\ \big((q_1, a, b, q_2) \in G_1\big) \Rightarrow$
    there exists $(q_1, a, b', q_2) \in G_2$ such that
      if $\mathsf{path}(q_2, p, G_1)$ then $(b' = \mathsf{dirty})$ else $(b' = b)$

$\mathsf{path}(p, q, G) \equiv$
    $\big((p, a, b, q) \in G\big)$
  $\vee\ \big(\exists p'.\ ((p, a, b, p') \in G) \wedge \mathsf{path}(p', q, G)\big)$

$\mathsf{consistent\text{-}action}(G, a, q)$:
  $\mathsf{consistent\text{-}action}(G, \mathbf{obs}\, v, q)$ when $G(q) = v$
  $\mathsf{consistent\text{-}action}(G, \mathbf{obs}\, t, q)$ when $G(q) = (e, t)$
  $\mathsf{consistent\text{-}action}(G, \mathbf{alloc}\, v, q)$ when $G(q) = v$
  $\mathsf{consistent\text{-}action}(G, \mathbf{alloc}\, e, q)$ when $G(q) = e$

Figure 9: Graph predicates and transformations

Definition C.1 (Restriction)**.**

$$
\begin{array}{rrcll}
 & \lfloor \varepsilon \rfloor_P & = & \varepsilon & \\
 & \lfloor G, p{:}v \rfloor_P & = & \lfloor G \rfloor_P, p{:}v & \text{if } p \in P \\
 & \lfloor G, p{:}v \rfloor_P & = & \lfloor G \rfloor_P & \text{if } p \notin P \\
 & \lfloor G, p{:}e \rfloor_P & = & \lfloor G \rfloor_P, p{:}e & \text{if } p \in P \\
 & \lfloor G, p{:}e \rfloor_P & = & \lfloor G \rfloor_P & \text{if } p \notin P \\
* & \lfloor G, p{:}(e, t) \rfloor_P & = & \lfloor G \rfloor_P, p{:}e & \text{if } p \in P \\
 & \lfloor G, p{:}(e, t) \rfloor_P & = & \lfloor G \rfloor_P & \text{if } p \notin P \\
** & \lfloor G, (p, a, b, q) \rfloor_P & = & \lfloor G \rfloor_P &
\end{array}
$$

C.4 Read and write sets

Join and merge operations. To specify the read and write sets, we use a "separating join" $H_1 * H_2$ on graphs: $H_1 * H_2 = (H_1, H_2)$ if $\mathsf{dom}(H_1) \perp \mathsf{dom}(H_2)$, and undefined otherwise.

$\boxed{G \vdash H \text{ wf}}$

$$\frac{}{G \vdash \varepsilon \text{ wf}}\ \textsf{Grwf-emp} \qquad \frac{G \vdash H \text{ wf} \qquad G(p) = e}{G \vdash H, p{:}v \text{ wf}}\ \textsf{Grwf-val} \qquad \frac{G \vdash H \text{ wf} \qquad G(p) = e}{G \vdash H, p{:}e \text{ wf}}\ \textsf{Grwf-thunk}$$

$$\frac{G \vdash H \text{ wf} \qquad \begin{array}{l} G(p) = (e, t) \\ \text{if all-clean-out}(G, p) \text{ then } (\mathcal{D} :: G \vdash^{p}_{\omega} e \Downarrow G; t) \\ \text{if not all-clean-out}(G, p) \text{ then all-dirty-in}(G, p) \end{array}}{G \vdash H, p{:}(e, t) \text{ wf}}\ \textsf{Grwf-thunkCache}$$

$$\frac{G \vdash H \text{ wf} \qquad \begin{array}{l} (p, a, \textsf{dirty}, q) \in G \\ q \in \text{dom}(H) \\ \text{if } (p_0, a_0, b_0, p) \in G \text{ then } b_0 = \textsf{dirty} \end{array}}{G \vdash H, (p, a, \textsf{dirty}, q) \text{ wf}}\ \textsf{Grwf-dirtyEdge}$$

$$\frac{G \vdash H \text{ wf} \qquad \begin{array}{l} (p, a, \textsf{clean}, q) \in G \\ \text{consistent-action}(H, a, q) \\ \text{all-clean-out}(G, q) \end{array}}{G \vdash H, (p, a, \textsf{clean}, q) \text{ wf}}\ \textsf{Grwf-cleanEdge}$$

Figure 10: Graph well-formedness rules

We also define a "merge" $H_1 \cup H_2$ that *is* defined for subgraphs with overlapping domains, provided H_1 and H_2 are consistent with each other. That is, if $p \in \text{dom}(H_1)$ and $p \in \text{dom}(H_2)$, then $H_1(p) = H_2(p)$.

Definition C.2 (Reads/writes)**.** The effect of an evaluation derived by $\mathcal{D}$, written $\mathcal{D}$ reads R writes W, is defined in Figure 11.

This is a function over derivations. We write "$\mathcal{D}$ by $\mathcal{R}(\vec{\mathcal{D}})$ reads R writes W" to mean that rule $\mathcal{R}$ concludes $\mathcal{D}$ and has subderivations $\vec{\mathcal{D}}$. For example, $\mathcal{D}$ by Eval-fix($\mathcal{D}_0$) reads R writes W provided that $\mathcal{D}_0$ reads R writes W where $\mathcal{D}_0$ derives the only premise of Eval-fix.

In the Eval-forceClean case, we refer back to the derivation that (most recently) computed the thunk being forced (that is, the first subderivation of Eval-computeDep). A completely formal definition would take as input a mapping from pointers q to sets R′ and W', return this mapping as output, and modify the mapping in the Eval-computeDep case.

Agreement. We want to express a result (Lemma C.5 (Respect for write-set)) that evaluation only affects pointers in the write set W, leaving the contents of other pointers alone, so we define what it means to leave pointers alone:

Definition C.3 (Agreement on a pointer)**.** Graphs G_1 and G_2 *agree on* p iff either $\text{exp}(G_1, p) = \text{exp}(G_2, p)$, or $p{:}v \in G_1$ and $p{:}v \in G_2$.

Definition C.4 (Agreement on a set of pointers)**.** Graphs G_1 and G_2 *agree on* a set P of pointers iff G_1 and G_2 agree on each $p \in P$.

Lemma C.5 (Respect for write-set)**.**
If $\mathcal{D} :: G \vdash^{p}_{\omega} e \Downarrow G'; t$ *where* $\mathcal{D}$ *is incremental and* $\mathcal{D}$ reads R writes W
then G' *agrees with* G *on* $\text{dom}(G) - \text{dom}(W)$.

Proof. By a straightforward induction on $\mathcal{D}$, referring to Definition C.2.

In the Eval-forceClean case, use the fact that G' differs from G only in the addition of an edge, which does not affect agreement. □

C.5 Satisfactory derivations

In our main result (Theorem C.13), we will assume that all input and output graphs appearing within a derivation are well-formed, and that the read- and write-sets are defined:

Definition C.6 (Locally satisfactory)**.**
A derivation $\mathcal{D} :: G_1 \vdash^{p}_{\omega} e \Downarrow G_2; t$ is *locally satisfactory* if and only if

$$\mathcal{D} \text{ by Eval-term}() \text{ reads } \varepsilon \text{ writes } \varepsilon$$

$$\mathcal{D} \text{ by Eval-app}(\mathcal{D}_1, \mathcal{D}_2) \text{ reads } R_1 \cup (R_2 - W_1) \text{ writes } W_1 * W_2 \quad \text{if} \quad \mathcal{D}_1 \text{ reads } R_1 \text{ writes } W_1 \text{ and } \mathcal{D}_2 \text{ reads } R_2 \text{ writes } W_2$$

$$\mathcal{D} \text{ by Eval-bind}(\mathcal{D}_1, \mathcal{D}_2) \text{ reads } R_1 \cup (R_2 - W_1) \text{ writes } W_1 * W_2 \quad \text{if} \quad \mathcal{D}_1 \text{ reads } R_1 \text{ writes } W_1 \text{ and } \mathcal{D}_2 \text{ reads } R_2 \text{ writes } W_2$$

$$\mathcal{D} \text{ by Eval-nest}(\mathcal{D}_1, \mathcal{D}_2) \text{ reads } R_1 \cup (R_2 - W_1) \text{ writes } W_1 * W_2 \quad \text{if} \quad \mathcal{D}_1 \text{ reads } R_1 \text{ writes } W_1 \text{ and } \mathcal{D}_2 \text{ reads } R_2 \text{ writes } W_2$$

$$\mathcal{D} \text{ by Eval-computeDep}(\mathcal{D}_1, \mathcal{D}_2) \text{ reads } R_1 \cup R_2 \text{ writes } W_2 \quad \text{if} \quad \mathcal{D}_1 \text{ reads } R_1 \text{ writes } W_1 \text{ and } \mathcal{D}_2 \text{ reads } R_2 \text{ writes } W_2 \text{ and } \mathrm{dom}(W_1) \subseteq \mathrm{dom}(W_2)$$

$$\mathcal{D} \text{ by Eval-fix}(\mathcal{D}_0) \text{ reads } R \text{ writes } W \quad \text{if} \quad \mathcal{D}_0 \text{ reads } R \text{ writes } W$$

$$\mathcal{D} \text{ by Eval-case}(\mathcal{D}_0) \text{ reads } R \text{ writes } W \quad \text{if} \quad \mathcal{D}_0 \text{ reads } R \text{ writes } W$$

$$\mathcal{D} \text{ by Eval-split}(\mathcal{D}_0) \text{ reads } R \text{ writes } W \quad \text{if} \quad \mathcal{D}_0 \text{ reads } R \text{ writes } W$$

$$\mathcal{D} \text{ by Eval-namespace}(\mathcal{D}_0) \text{ reads } R \text{ writes } W \quad \text{if} \quad \mathcal{D}_0 \text{ reads } R \text{ writes } W$$

$$\mathcal{D} \text{ by Eval-fork}() \text{ reads } \varepsilon \text{ writes } \varepsilon$$

$$\mathcal{D} \text{ by Eval-refDirty}() \text{ reads } \varepsilon \text{ writes } q{:}v \quad \text{where } e = \mathbf{ref}(\mathbf{nm}\, k, v) \text{ and } q = k@\omega$$

$$\mathcal{D} \text{ by Eval-refClean}() \text{ reads } \varepsilon \text{ writes } q{:}v \quad \text{where } e = \mathbf{ref}(\mathbf{nm}\, k, v) \text{ and } q = k@\omega$$

$$\mathcal{D} \text{ by Eval-thunkDirty}() \text{ reads } \varepsilon \text{ writes } q{:}e_0 \quad \text{where } e = \mathbf{thunk}(\mathbf{nm}\, k, e_0)$$

$$\mathcal{D} \text{ by Eval-thunkClean}() \text{ reads } \varepsilon \text{ writes } q{:}G(q) \quad \text{where } e = \mathbf{thunk}(\mathbf{nm}\, k, e_0)$$

$$\mathcal{D} \text{ by Eval-getClean}() \text{ reads } q{:}v \text{ writes } \varepsilon \quad \text{where } e = \mathbf{get}\,(\mathbf{ref}\, q) \text{ and } q = k@\omega \text{ and } G(q) = v$$

$$\mathcal{D} \text{ by Eval-forceClean}() \text{ reads } R', q{:}(e, t) \text{ writes } W' \quad \text{where } e = \mathbf{force}\,(\mathbf{thk}\, q) \text{ and } \mathcal{D}' \text{ reads } R' \text{ writes } W' \text{ where } \mathcal{D}' \text{ is the derivation that computed } t \text{ (see text)}$$

$$\mathcal{D} \text{ by Eval-scrubEdge}(\mathcal{D}_0) \text{ reads } R \text{ writes } W \quad \text{if} \quad \mathcal{D}_0 \text{ reads } R \text{ writes } W$$

Figure 11: Read- and write-sets of a derivation

(1) G_1 wf and G_2 wf
(2) $\mathcal{D}$ reads R writes W is defined

Definition C.7 (Globally satisfactory)**.**
An evaluation derivation $\mathcal{D} :: H_1 \vdash^{p}_{\omega} e \Downarrow H_2; t$ is *globally satisfactory*, written $\mathcal{D}$ satisfactory, if and only if $\mathcal{D}$ is locally satisfactory and all its subderivations are locally satisfactory.

C.6 Weakening

The main result needs to construct a non-incremental derivation in a different order from the given incremental derivation. In particular, the first evaluation $\mathcal{D}_1$ done in Eval-computeDep will be done "later" in the non-incremental derivation. Since it is done later, the graph may have new material G', and we need a weakening lemma to move from a non-incremental evaluation of $\mathcal{D}_1$ (obtained through the induction hypothesis) to a non-incremental evaluation over the larger graph.

Lemma C.8 (Weakening (non-incremental))**.**
If $\mathcal{D} :: G_1 \vdash^{p}_{\omega} e \Downarrow G_2; t$ *and* $\mathcal{D}$ *is non-incremental*
and G' *is disjoint from* G_1 *and* G_2
then $\mathcal{D}' :: G_1, G' \vdash^{p}_{\omega} e \Downarrow G_2, G'; t$ *where* $\mathcal{D}'$ *is non-incremental.*

Proof. By induction on $\mathcal{D}$.

- **Cases** Eval-term, Eval-app, Eval-fix, Eval-bind, Eval-case, Eval-split, Eval-fork, Eval-namespace, Eval-nest: These rules do not manipulate the graph, so just use the i.h. on each subderivation, then apply the same rule.
- **Case** Eval-refPlain: Since $G_1\{q \mapsto v\} = G_2$, we have $q \in \mathrm{dom}(G_2)$.
  It is given that $\mathrm{dom}(G') \perp \mathrm{dom}(G_2)$. Therefore $q \notin \mathrm{dom}(G_1, G')$.

By definition, $(G_1, G')\{q \mapsto v\} = (G_2, G')$.
Apply Eval-refPlain.

- **Case** Eval-thunkPlain: Similar to the Eval-refPlain case.
- **Case** Eval-forcePlain: We have $\mathsf{exp}(G_1, q) = e_0$. Therefore $\mathsf{exp}(G_1, G', q) = e_0$. Use the i.h. and apply Eval-forcePlain.
- **Case** Eval-getPlain: Similar to the Eval-forcePlain case. □

C.7 Main result: From-scratch consistency

At the highest level, the main result (Theorem C.13) says:

First approximation
If $H_1 \vdash^{p}_{\omega} e \Downarrow H_2; t$ by an incremental derivation,
then $H_1' \vdash^{p}_{\omega} e \Downarrow H_2'; t$, where H_1' is a non-incremental version of H_1 and H_2' is a non-incremental version of H_2.

Using the restriction function from Section C.3, we can refine this statement:

Second approximation
If $H_1 \vdash^{p}_{\omega} e \Downarrow H_2; t$ by an incremental derivation and $P_1 \subseteq \mathsf{dom}(H_1)$,
then $\lfloor H_1 \rfloor_{P_1} \vdash^{p}_{\omega} e \Downarrow \lfloor H_2 \rfloor_{P_2}; t$ by a non-incremental derivation, for some P_2 such that $P_1 \subseteq P_2$.

Here, the pointer set P_1 gives the scope of the non-incremental input graph $\lfloor H_1 \rfloor_{P_1}$, and we construct P_2 describing the non-incremental output graph $\lfloor H_2 \rfloor_{P_2}$.

We further refine this statement by involving the derivation's read- and write-sets: the read set R must be contained in P_1, the write set W must be disjoint from P_1 (written $\mathsf{dom}(W) \perp P_1$), and P_2 must be exactly P_1 plus W. In the non-incremental semantics, the store should grow monotonically, so we will also show $\lfloor H_1 \rfloor_{P_1} \subseteq \lfloor H_2 \rfloor_{P_2}$:

Third approximation
If $H_1 \vdash^{p}_{\omega} e \Downarrow H_2; t$ by an incremental derivation $\mathcal{D}$ with $\mathcal{D}$ reads R writes W,
and $P_1 \subseteq \mathsf{dom}(H_1)$ such that $\mathsf{dom}(R) \subseteq P_1$ and $\mathsf{dom}(W) \perp P_1$
then $\lfloor H_1 \rfloor_{P_1} \vdash^{p}_{\omega} e \Downarrow \lfloor H_2 \rfloor_{P_2}; t$
where $\lfloor H_1 \rfloor_{P_1} \subseteq \lfloor H_2 \rfloor_{P_2}$ and $P_2 = P_1 \cup \mathsf{dom}(W)$.

Even this refinement is not quite enough, because the incremental system can perform computations in a different order from the non-incremental system. Specifically, the Eval-computeDep rule carries out a subcomputation first, then continues with a larger computation that depends on the subcomputation. The subcomputation does not fit into the non-incremental derivation at that point; non-incrementally, the subcomputation is performed when it is demanded by the larger computation. Thus, we can't just apply the induction hypothesis on the subcomputation.

However, the subcomputation is "saved" in its (incremental) output graph. So we incorporate an invariant that all thunks with cached results in the graph "are consistent", that is, they satisfy a property similar to the overall consistency result. When this is the case, we say that the graph is from-scratch consistent. The main result, then, will assume that the input graph is from-scratch consistent, and show that the output graph remains from-scratch consistent. Since the graph can grow in the interval between the subcomputation of Eval-computeDep and the point where the subcomputation is demanded, we require that the output graph G_2 of the saved derivation does not contradict the larger, newer graph H. This is part (4) in the next definition. (We number the parts from (i) to (ii) and then from (1) so that they mostly match similar parts in the main result.)

Definition C.9 (From-scratch consistency of a derivation)**.**
A derivation $\mathcal{D}_i :: G_1 \vdash^{p}_{\omega} e \Downarrow G_2; t$ is
from-scratch consistent for $P_q \subseteq \mathsf{dom}(G_1)$ *up to* H if and only if

(i) $\mathcal{D}_i$ satisfactory where $\mathcal{D}_i$ reads R writes W
(ii) $\mathsf{dom}(R) \subseteq P_q$ and $\mathsf{dom}(W) \perp P_q$
(1) there exists a non-incremental derivation $\mathcal{D}_{ni} :: \lfloor G_1 \rfloor_{P_q} \vdash^{p}_{\omega} e \Downarrow \lfloor G_2 \rfloor_{P_2}; t$
(2) $\lfloor G_1 \rfloor_{P_q} \subseteq \lfloor G_2 \rfloor_{P_2}$
(3) $P_2 = P_q \cup \mathsf{dom}(W)$
(4) $\lfloor G_2 \rfloor_{P_2} \subseteq \lfloor H \rfloor_{\mathsf{dom}(H)}$

Definition C.10 (From-scratch consistency of graphs)**.**
A graph H is *from-scratch consistent*
if, for all $q \in \mathsf{dom}(H)$ such that $H(q) = (e, t)$,

there exists $\mathcal{D}_q :: G_1 \vdash^p_\omega e \Downarrow G_2; t$
and $P_q \subseteq \mathsf{dom}(G_1)$
such that $\mathcal{D}_q$ is from-scratch consistent (Definition C.9) for P_q up to H.

The proof of the main result must maintain that the graph is from-scratch consistent as the graph becomes larger, for which Lemma C.12 (Consistent graph extension) is useful.

Lemma C.11 (Consistent extension)**.**
If $\mathcal{D}_q$ is from-scratch consistent for P_q up to H
and $H \subseteq H'$
then $\mathcal{D}_q$ is from-scratch consistent for P_q up to H'.

Proof. Only part (4) of Definition C.9 involves the "up to" part of from-scratch consistency, so we already have (i)–(iii) and (1)–(3).

We have (4) $\lfloor G_2 \rfloor_{P_2} \subseteq \lfloor H \rfloor_{\mathsf{dom}(H)}$. Using our assumptions, $\lfloor H \rfloor_{\mathsf{dom}(H)} \subseteq \lfloor H' \rfloor_{\mathsf{dom}(H')}$.
Therefore (4) $\lfloor G_2 \rfloor_{P_2} \subseteq \lfloor H' \rfloor_{\mathsf{dom}(H')}$. □

Lemma C.12 (Consistent graph extension)**.**
If H is from-scratch consistent
and $H \subseteq H'$
and, for all $q \in \mathsf{dom}(H') - \mathsf{dom}(H)$ such that $H'(q) = (e, t)$,
there exists $\mathcal{D}_q :: G_1 \vdash^p_\omega e \Downarrow G_2; t$ and $P_q \subseteq \mathsf{dom}(G_1)$
such that $\mathcal{D}_q$ is from-scratch consistent (Definition C.9)
for P_q up to H',
then H' is from-scratch consistent.

Proof. Use Lemma C.11 (Consistent extension) on each "old" pointer with a cached result in H, then apply the definitions for each "new" pointer with a cached result in $\mathsf{dom}(H') - \mathsf{dom}(H)$. □

At last, we can state and prove the main result, which corresponds to the "third approximation" above, plus the invariant that the graph is from-scratch consistent (parts (iii) and (4)).

We present most of the proof in a line-by-line style, with the judgment or proposition being derived in the left column, and its justification in the right column. In each case, we need to show four different things (1)–(4), some of which are obtained midway through the case, so we highlight these with "(1) ☞ ", and so on.

Theorem C.13 (From-scratch consistency)**.**
Given an incremental $\mathcal{D}_i :: H_1 \vdash^p_\omega e \Downarrow H_2; t$ where

(i) $\mathcal{D}_i$ satisfactory where $\mathcal{D}$ reads R writes W
(ii) a set of pointers $P_1 \subseteq \mathsf{dom}(H_1)$ is such that $\mathsf{dom}(R) \subseteq P_1$ and $\mathsf{dom}(W) \perp P_1$
(iii) H_1 is from-scratch consistent (Definition C.10)

then

(1) there exists a non-incremental $\mathcal{D}_{ni} :: \lfloor H_1 \rfloor_{P_1} \vdash^p_\omega e \Downarrow \lfloor H_2 \rfloor_{P_2}; t$
(2) $\lfloor H_1 \rfloor_{P_1} \subseteq \lfloor H_2 \rfloor_{P_2}$
(3) $P_2 = P_1 \cup \mathsf{dom}(W)$
(4) H_2 is from-scratch consistent (Definition C.10).

Proof. By induction on $\mathcal{D}_i :: H_1 \vdash^p_\omega e \Downarrow H_2; t$.

- **Case** Eval-term: By Definition C.2, $W = \varepsilon$, so let $P_2 = P_1$.
  (1) Apply Eval-term.
  (2) We have $H_1 = H_2$ and $P_1 = P_2$ so $\lfloor H_1 \rfloor_{P_1} = \lfloor H_2 \rfloor_{P_2}$.
  (3) It follows from $\mathsf{dom}(W) = \emptyset$ and $P_2 = P_1$ that $P_2 = P_1 \cup \mathsf{dom}(W)$.
  (4) It is given (iii) that H_1 is from-scratch consistent.
  We have $H_1 = H_2$, so H_2 is from-scratch consistent.

- **Case** Eval-fork: Similar to the Eval-term case.

• **Case** $\dfrac{H_1(q) = v}{H_1 \vdash^p_\omega \mathbf{get}\,(\mathbf{ref}\,q) \Downarrow H_1, (p, \mathbf{obs}\,v, \mathsf{clean}, q); \mathbf{ret}\,v}$ Eval-getClean

	$H_2 = (H_1, (p, \mathbf{obs}\,v, \mathsf{clean}, q))$	Given
	$e = \mathbf{get}\,(\mathbf{ref}\,q)$	″
	$H_1(q) = v$	Premise
	$R = q{:}v$	By Definition C.2
	$\mathsf{dom}(R) \subseteq P_1$	Given (iii)
	$\{q\} \subseteq P_1$	$R = q{:}v$
	$\lfloor H_1 \rfloor_{P_1}(q) = v$	By Definition C.1
	$\lfloor H_1 \rfloor_{P_1} \vdash^p_\omega e \Downarrow \lfloor H_1 \rfloor_{P_1}; \mathbf{ret}\,v$	By rule Eval-getPlain
	Let P_2 be P_1.	
	$\lfloor H_1 \rfloor_{P_1} = \lfloor H_2 \rfloor_{P_1}$	By def. of restriction
(1) ☞	$\lfloor H_1 \rfloor_{P_1} \vdash^p_\omega e \Downarrow \lfloor H_2 \rfloor_{P_1}; \mathbf{ret}\,v$	By above equality
	$W = \varepsilon$	By Definition C.2
(2) ☞	$\lfloor H_1 \rfloor_{P_1} \subseteq \lfloor H_2 \rfloor_{P_1}$	If $=$ then $\subseteq$
(3) ☞	$P_2 = P_1 \cup \mathsf{dom}(\varepsilon)$	$P_2 = P_1$
	H_1 from-scratch consistent	Given (iii)
(4) ☞	H_2 from-scratch consistent	H_2 differs from H_1 only in its edges

• **Case** $\dfrac{q = k@\omega \qquad H_1\{q{\mapsto}e_0\} = G_2 \qquad \mathsf{dirty\text{-}paths\text{-}in}(G_2, q) = G_3}{H_1 \vdash^p_\omega \mathbf{thunk}(\mathbf{nm}\,k, e_0) \Downarrow G_3, (p, \mathbf{alloc}\,e_0, \mathsf{clean}, q); \mathbf{ret}\,(\mathbf{thk}\,q)}$ Eval-thunkDirty

	$H_2 = (G_3, (p, \mathbf{alloc}\,e_0, \mathsf{clean}, q))$	Given
	$e = \mathbf{thunk}(\mathbf{nm}\,k, e_0)$	Given
	$q = k@\omega$	Given
	$R = \varepsilon$	By Definition C.2
	$W = q{:}e_0$	″
	$\mathsf{dom}(W) \perp P_1$	Given
	$q \notin P_1$	
	$q \notin \mathsf{dom}(\lfloor H_1 \rfloor_{P_1})$	From Definition C.1
	$\lfloor H_1 \rfloor_{P_1} \vdash^p_\omega \mathbf{thunk}(\mathbf{nm}\,k, e_0) \Downarrow \lfloor H_1 \rfloor_{P_1}\{q{\mapsto}e_0\}; \mathbf{ret}\,(\mathbf{thk}\,q)$	By rule Eval-thunkPlain
(3) ☞	Let P_2 be $P_1 \cup \{q\}$.	
	$\lfloor H_1 \rfloor_{P_1}\{q{\mapsto}e_0\} = \lfloor H_1\{q{\mapsto}e_0\} \rfloor_{P_1}$	$q \notin P_1$
	$= \lfloor \mathsf{dirty\text{-}paths\text{-}in}(H_1\{q{\mapsto}e_0\}, q) \rfloor_{P_1}$	$q \notin P_1$
	$= \lfloor \mathsf{dirty\text{-}paths\text{-}in}(H_1\{q{\mapsto}e_0\}, q), (p, \mathbf{alloc}\,e_0, \mathsf{clean}, q) \rfloor_{P_1}$	From Definition C.1
(1) ☞	$\lfloor H_1 \rfloor_{P_1} \vdash^p_\omega \mathbf{thunk}(\mathbf{nm}\,k, e_0) \Downarrow \lfloor H_2 \rfloor_{P_1}\{q{\mapsto}e_0\}; \mathbf{ret}\,(\mathbf{thk}\,q)$	By above equalities
	$\lfloor H_1 \rfloor_{P_1} \subseteq \lfloor H_1\{q{\mapsto}e_0\} \rfloor_{P_2}$	Immediate
	$\lfloor H_1 \rfloor_{P_1} \subseteq \lfloor G_2 \rfloor_{P_2}$	By Definition C.1
	$\lfloor H_1 \rfloor_{P_1} \subseteq \lfloor G_3 \rfloor_{P_2}$	Restriction ignores edges
(2) ☞	$\lfloor H_1 \rfloor_{P_1} \subseteq \lfloor H_2 \rfloor_{P_2}$	Restriction ignores edges
	H_1 from-scratch consistent	Given (iii)
(4) ☞	H_2 from-scratch consistent	By Lemma C.12 (Consistent graph extension)

• **Case** $\dfrac{q = k@\omega \qquad \mathsf{exp}(H_1, q) = e_0}{H_1 \vdash^p_\omega \mathbf{thunk}(\mathbf{nm}\,k, e_0) \Downarrow H_1, (p, \mathbf{alloc}\,e_0, \mathsf{clean}, q); \mathbf{ret}\,(\mathbf{thk}\,q)}$ Eval-thunkClean

$H_2 = (H_1, (p, \mathbf{alloc}\, e_0, \mathsf{clean}, q))$ Given
$e = \mathbf{thunk}(\mathbf{nm}\, k, e_0)$ Given
$q = k@\omega$ Given
$W = q{:}e_0$ By Definition C.2
$\mathsf{dom}(W) \perp P_1$ Given
$q \notin P_1$
$q \notin \mathsf{dom}(\lfloor H_1 \rfloor_{P_1})$ From Definition C.1
$\lfloor H_1 \rfloor_{P_1} \vdash^{p}_{\omega} \mathbf{thunk}(\mathbf{nm}\, k, e_0) \Downarrow \lfloor H_1 \rfloor_{P_1}\{q \mapsto e_0\}; \mathbf{ret}\, (\mathbf{thk}\, q)$ By rule Eval-thunkPlain
$\lfloor H_1 \rfloor_{P_1}\{q \mapsto e_0\} = \lfloor H_1, (p, \mathbf{alloc}\, e_0, \mathsf{clean}, q) \rfloor_{P_1}$ From Definition C.1 and $\exp(H_1, q) = e_0$
☞ (1) $\lfloor H_1 \rfloor_{P_1} \vdash^{p}_{\omega} \mathbf{thunk}(\mathbf{nm}\, k, e_0) \Downarrow \lfloor H_2 \rfloor_{P_1}\{q \mapsto e_0\}; \mathbf{ret}\, (\mathbf{thk}\, q)$ By above equalities

Since $H_2 = H_1, (p, \mathbf{alloc}\, e_0, \mathsf{clean}, q)$, parts (2)–(4) are straightforward.

- **Case** Eval-refDirty: Similar to the Eval-thunkDirty case.

- **Case** Eval-refClean: Similar to the Eval-thunkClean case.

- **Case** $\dfrac{H_1 \vdash^{p}_{\omega} e_1 \Downarrow H'; \mathbf{ret}\, v \qquad H' \vdash^{p}_{\omega} [v/x]e_2 \Downarrow H_2; t}{H_1 \vdash^{p}_{\omega} \mathbf{let}\, x \leftarrow e_1\, \mathbf{in}\, e_2 \Downarrow H_2; t}$ Eval-bind

$\mathcal{D}_1 :: H_1 \vdash^{p}_{\omega} e_1 \Downarrow H'; \mathbf{ret}\, v$ Subderivation
(i) $\mathcal{D}_1$ satisfactory By Definition C.7
$R = (R_1 \cup (R_2 - W_1))$ By Definition C.2
$W = W_1 * W_2$ ″
$\mathsf{dom}(R_1 \cup (R_2 - W_1)) \subseteq P_1$ Given
$\mathsf{dom}(W_1 * W_2) \perp P_1$ Given
$\mathcal{D}_1$ reads R_1 writes W_1 ″
$\mathcal{D}_2$ reads R_2 writes W_2 ″
(ii) $\mathsf{dom}(R_1) \subseteq P_1$ $\mathsf{dom}(R_1 \cup (R_2 - W_1)) \subseteq P_1$
(ii) $\mathsf{dom}(W_2) \perp P_1$ $\mathsf{dom}(W_1 * W_2) \perp P_1$
$\lfloor H_1 \rfloor_{P_1} \vdash^{p}_{\omega} e_1 \Downarrow \lfloor H' \rfloor_{P'}; \mathbf{ret}\, v$ By i.h.
$\lfloor H_1 \rfloor_{P_1} \subseteq \lfloor H' \rfloor_{P'}$ ″
$P' = P_1 \cup \mathsf{dom}(W_1)$ ″
H' from-scratch consistent ″

$\mathcal{D}_2 :: H' \vdash^{p}_{\omega} [v/x]e_2 \Downarrow H_2; t$ Subderivation
(i) $\mathcal{D}_2$ satisfactory By Definition C.7
$\mathsf{dom}(R_1 \cup (R_2 - W_1)) \subseteq P_1$ Given
$\mathsf{dom}(R_2 - W_1) \subseteq P_1$
$\mathsf{dom}(R_2) \subseteq P_1 \cup \mathsf{dom}(W_1)$
(ii) $\mathsf{dom}(R_2) \subseteq P'$ $P' = P_1 \cup \mathsf{dom}(W_1)$

	$\mathsf{dom}(W_1 * W_2) \perp P_1$	Given	
	$\mathsf{dom}(W_1) \perp \mathsf{dom}(W_2)$	From def. of $*$	
	$\mathsf{dom}(W_2) \perp P_1 \cup \mathsf{dom}(W_1)$		
(ii)	$\mathsf{dom}(W_2) \perp P'$	$P' = P_1 \cup \mathsf{dom}(W_1)$	
	$\lfloor H' \rfloor_{P'} \vdash^{p}_{\omega} [v/x]e_2 \Downarrow \lfloor H_2 \rfloor_{P_2}; t$		By i.h.
	$\lfloor H' \rfloor_{P'} \subseteq \lfloor H_2 \rfloor_{P_2}$		$''$
	$P_2 = P' \cup \mathsf{dom}(W_2)$		$''$
(4) ☞	H_2 from-scratch consistent		$''$
(2) ☞	$\lfloor H_1 \rfloor_{P_1} \subseteq \lfloor H_2 \rfloor_{P_2}$		By transitivity of $\subseteq$
	$P_2 = P_1 \cup \mathsf{dom}(W_1) \cup \mathsf{dom}(W_2)$		By $P' = P_1 \cup \mathsf{dom}(W_1)$
(3) ☞	$P_2 = P_1 \cup \mathsf{dom}(W)$		By $W = W_1 * W_2$
	$\lfloor H_1 \rfloor_{P_1} \vdash^{p}_{\omega} e_1 \Downarrow \lfloor H' \rfloor_{P'}; \mathbf{ret}\, v$		Above
	$\lfloor H' \rfloor_{P'} \vdash^{p}_{\omega} [v/x]e_2 \Downarrow \lfloor H_2 \rfloor_{P_2}; t$		Above
(1) ☞	$\lfloor H_1 \rfloor_{P_1} \vdash^{p}_{\omega} \mathbf{let}\, x \leftarrow e_1 \,\mathbf{in}\, e_2 \Downarrow \lfloor H_2 \rfloor_{P_2}; t$		By rule Eval-bind

- **Case** Eval-app: Similar to the Eval-bind case.
- **Case** Eval-nest: Similar to the Eval-bind case.
- **Case** Eval-fix: The input and output graphs of the subderivation match those of the conclusion, as do the read and write sets according to Definition C.2. Thus, we can just use the i.h. and apply Eval-fix.
- **Case** Eval-case: Similar to the Eval-fix case.
- **Case** Eval-split: Similar to the Eval-case case.
- **Case** Eval-namespace: Similar to the Eval-fix case.

- **Case**
$$\frac{\begin{array}{ll}\mathsf{exp}(H_1, q) = e' & G_2\{q \mapsto (e', t')\} = G_2' \\ \mathsf{del\text{-}edges\text{-}out}(H_1\{q \mapsto e'\}, q) = G_1' & \mathsf{all\text{-}clean\text{-}out}(G_2', q) \\ \mathcal{D}_1 :: G_1' \vdash^{q}_{\mathsf{namespace}(q)} e' \Downarrow G_2; t' & \mathcal{D}_2 :: G_2' \vdash^{p}_{\omega} \mathbf{force}\,(\mathbf{thk}\, p_0) \Downarrow H_2; t\end{array}}{H_1 \vdash^{p}_{\omega} \mathbf{force}\,(\mathbf{thk}\, p_0) \Downarrow H_2; t}\ \textsf{Eval-computeDep}$$

$e = \mathbf{force}\,(\mathbf{thk}\, p_0)$	Given
$R = (R_1 \cup R_2)$	By Definition C.2
$W = W_2$	$''$
$\mathcal{D}_1$ reads R_1 writes W_1	$''$
$\mathcal{D}_2$ reads R_2 writes W	$''$
$\mathsf{dom}(W_1) \subseteq \mathsf{dom}(W_2)$	$''$

We don't immediately need to apply the i.h. to $\mathcal{D}_1$, because that computation will be done later in the reference derivation. But we do need to apply the i.h. to $\mathcal{D}_2 :: G_2' \vdash^{p}_{\omega} e \Downarrow H_2; t$. So we need to show part (iii) of the statement, which says that each *cached* computation in the input graph is consistent with respect to an earlier version of the graph.
Since we're adding such a computation e' in G_2', which is the input graph of $\mathcal{D}_2$, we have to show that the computation of e' (by $\mathcal{D}_1$) is consistent, which means applying the i.h. to $\mathcal{D}_1$.

(iii)	$\mathsf{dom}(R_1) \subseteq P_1$	$\mathsf{dom}(R) \subseteq P_1$
(iii)	$\mathsf{dom}(W_1) \perp P_1$	$\mathsf{dom}(W) \perp P_1$
	H_1 from-scratch consistent	Given (iii)
	G_1' from-scratch consistent	G_1' differs from H_1 only in its edges (note that q points to e' but has no cached result)
	$\mathcal{D}_1 :: G_1' \vdash^{q}_{\mathsf{namespace}(q)} e' \Downarrow G_2; t'$	Subderivation
(1)	$\lfloor G_1' \rfloor_{P_1} \vdash^{q}_{\mathsf{namespace}(q)} e' \Downarrow \lfloor G_2 \rfloor_{P_1'}; t'$	By i.h.
(2)	$\lfloor G_1' \rfloor_{P_1} \subseteq \lfloor G_2 \rfloor_{P_1'}$	$''$
(3)	$P_1' = P_1 \cup \mathsf{dom}(W_1)$	$''$
(4)	G_2 from-scratch consistent	$''$
	$\lfloor G_2 \rfloor_{P_1'} \subseteq \lfloor G_2 \rfloor_{P_1'}$	If $=$ then $\subseteq$
	G_2' from-scratch consistent	By Lemma C.12 (Consistent graph extension)

Having "stowed away" the consistency of e', we can move on to $\mathcal{D}_2$.

	$\mathcal{D}_2 :: G_2' \vdash^{p}_{\omega} e \Downarrow H_2; t$	Subderivation
(i)	$\mathcal{D}_2$ satisfactory	$\mathcal{D}_2$ is a subderivation of $\mathcal{D}_i$
	$\mathsf{dom}(R) \subseteq P_1$ and $\mathsf{dom}(W) \perp P_1$	Given
(ii)	$\mathsf{dom}(R_2) \subseteq P_1$ and $\mathsf{dom}(W_2) \perp P_1$	Using above equalities
(iii)	H_1 from-scratch consistent	$''$
(1)	$\lfloor G_2' \rfloor_{P_1} \vdash^{p}_{\omega} e \Downarrow \lfloor H_2 \rfloor_{P_2}; t$	By i.h.
(2)	$\lfloor G_2' \rfloor_{P_1} \subseteq \lfloor H_2 \rfloor_{P_2}$	$''$
☞ (3)	$P_2 = P_1 \cup \mathsf{dom}(W)$	$''$
☞ (4)	H_2 from-scratch consistent	$''$

$\lfloor H_1 \rfloor_{P_1}$	
$= \lfloor H_1\{q \mapsto e'\} \rfloor_{P_1}$	Follows from $\mathsf{exp}(H_1, q) = e'$
$= \lfloor \mathsf{del\text{-}edges\text{-}out}(H_1\{q \mapsto e'\}, q) \rfloor_{P_1}$	Definition C.1 ignores edges
$= \lfloor G_1' \rfloor_{P_1}$	By above equality

We need to show $\lfloor G_1' \rfloor_{P_1} = \lfloor G_2 \rfloor_{P_1}$. That is, evaluating e'—which will be done *inside* the reference derivation's version of $\mathcal{D}_2$—doesn't change anything in P_1.
Fortunately, we know that $\mathsf{dom}(W) \perp P_1$ and $\mathsf{dom}(W_1) \subseteq \mathsf{dom}(W)$. Therefore $\mathsf{dom}(W_1) \perp P_1$.
By Lemma C.5 (Respect for write-set), G_2 agrees with G_1' on $\mathsf{dom}(G_1') - \mathsf{dom}(W_1)$. Since W_1 is disjoint from P_1, we have that G_2 agrees with G_1' on P_1.
Therefore $\lfloor G_1' \rfloor_{P_1} = \lfloor G_2 \rfloor_{P_1}$.
Now we'll show that $\lfloor G_2 \rfloor_{P_1} = \lfloor G_2' \rfloor_{P_1}$, that is, $\lfloor G_2 \rfloor_{P_1} = \lfloor G_2\{q \mapsto (e', t')\} \rfloor_{P_1}$.

- If $q \notin P_1$ then this follows easily from Definition C.1.
- Otherwise, $q \in P_1$. We have $\mathsf{exp}(G_1', q) = e'$ and therefore $\mathsf{exp}(G_2, q) = e'$, so updating G_2 with q pointing to e' doesn't change the restriction.

	$\lfloor H_1 \rfloor_{P_1} = \lfloor G_2' \rfloor_{P_1}$	Shown above
☞ (1)	$\lfloor H_1 \rfloor_{P_1} \vdash^{p}_{\omega} e \Downarrow \lfloor H_2 \rfloor_{P_2}; t$	By above equality
☞ (2)	$\lfloor H_1 \rfloor_{P_1} \subseteq \lfloor H_2 \rfloor_{P_2}$	$''$

- **Case** $\dfrac{H_1(q) = (e, t) \qquad \mathsf{all\text{-}clean\text{-}out}(H_1, q)}{H_1 \vdash^{p}_{\omega} \mathbf{force}\,(\mathbf{thk}\,q) \Downarrow H_1, (p, \mathbf{obs}\,t, \mathsf{clean}, q); t}\ \mathsf{Eval\text{-}forceClean}$

$H_2 = (H_1, (p, \mathbf{obs}\, t, \mathsf{clean}, q))$ Given

$H_1(q) = (e, t)$ Premise
H_1 from-scratch consistent over P_1 Given (iii)
$\mathcal{D}_q$ satisfactory Definition C.9 (i)
$\mathsf{dom}(R_q) \subseteq P_q$ ″ (ii)
$\mathsf{dom}(W_q) \perp P_q$ ″ (ii)
$R = (R_q, q{:}(e, t))$ By Definition C.2 for Eval-forceClean ($\mathcal{D}' = \mathcal{D}_q$)
$W = W_q$ ″

To show that $\mathcal{D}_q$ is consistent, we use assumption (iii) that H_1 is from-scratch consistent. We have $q{:}(e, t)$ in H_1. By Definition C.10), $\mathcal{D}_q :: G_q \vdash^{p_q}_{\omega_q} e \Downarrow G'_q; t$ is from-scratch consistent for some $P_q \subseteq \mathsf{dom}(G_q)$ up to H_1.
Now we turn to Definition C.9.

$\lfloor G_q \rfloor_{P_q} \vdash^{p_q}_{\omega_q} e \Downarrow \lfloor G'_q \rfloor_{P_q \cup \mathsf{dom}(W_q)}; t$ By Definition C.9 (1)
$\lfloor G_q \rfloor_{P_q} \subseteq \lfloor G'_q \rfloor_{P_q \cup \mathsf{dom}(W_q)}$ By Definition C.9 (2)
$\lfloor G'_q \rfloor_{P_q \cup \mathsf{dom}(W_q)} \subseteq \lfloor H_1 \rfloor_{P_1}$ By Definition C.9 (4)

$\lfloor H_1 \rfloor_{P_1} \vdash^{p_q}_{\omega_q} e \Downarrow \lfloor H_1 \rfloor_{P_1 \cup \mathsf{dom}(W_q)}; t$ By Lemma C.8 (Weakening (non-incremental))

Let P_2 be $P_1 \cup \mathsf{dom}(W_q)$.
$\lfloor H_1 \rfloor_{P_1} \vdash^{p}_{\omega} e \Downarrow \lfloor H_1 \rfloor_{P_2}; t$ By above equality
$\lfloor H_1 \rfloor_{P_1} \vdash^{p}_{\omega} \mathbf{force}\,(\mathbf{thk}\, q) \Downarrow \lfloor H_1 \rfloor_{P_2}; t$ By rule Eval-forcePlain
$\lfloor H_1 \rfloor_{P_1} \vdash^{p}_{\omega} \mathbf{force}\,(\mathbf{thk}\, q) \Downarrow \lfloor H_1, (p, \mathbf{obs}\, t, \mathsf{clean}, q) \rfloor_{P_2}; t$ By Definition C.1
(1) ☞ $\lfloor H_1 \rfloor_{P_1} \vdash^{p}_{\omega} \mathbf{force}\,(\mathbf{thk}\, q) \Downarrow \lfloor H_2 \rfloor_{P_2}; t$ By above equality
$\lfloor H_1 \rfloor_{P_1} \subseteq \lfloor H_1 \rfloor_{P_2}$ By a property of Definition C.1
(2) ☞ $\lfloor H_1 \rfloor_{P_1} \subseteq \lfloor H_2 \rfloor_{P_2}$ By a property of Definition C.1
(3) ☞ $P_2 = P_1 \cup \mathsf{dom}(W)$ $W = W_q$
(4) ☞ H_2 from-scratch consistent H_2 differs from H_1 only in its edges

- **Case**
$$\frac{\begin{array}{l}\mathsf{all\text{-}clean\text{-}out}((G_1, G_2), q_2) \\ \mathsf{consistent\text{-}action}((G_1, G_2), a, q_2) \\ G_1, (q_1, a, \mathsf{clean}, q_2), G_2 \vdash^{p}_{\omega} \mathbf{force}\,(\mathbf{thk}\, p_0) \Downarrow H_2; t\end{array}}{G_1, (q_1, a, \mathsf{dirty}, q_2), G_2 \vdash^{p}_{\omega} \mathbf{force}\,(\mathbf{thk}\, p_0) \Downarrow H_2; t}\ \textsf{Eval-scrubEdge}$$

$e = \mathbf{force}\,(\mathbf{thk}\, p_0)$ Given
$H_1 = (G_1, (q_1, a, \mathsf{dirty}, q_2), G_2)$ Given
Let G' be $G_1, (q_1, a, \mathsf{clean}, q_2), G_2$.
$\lfloor G'_1 \rfloor_{P_1} \vdash^{p}_{\omega} e \Downarrow \lfloor H_2 \rfloor_{P_2}; t$ By i.h.
$\lfloor G'_1 \rfloor_{P_1} \subseteq \lfloor H_2 \rfloor_{P_2}$ ″
(3) ☞ $P_2 = P_1 \cup \mathsf{dom}(W)$ ″
(4) ☞ H_2 from-scratch consistent ″
$\lfloor H_1 \rfloor_{P_1} = \lfloor G' \rfloor_{P_1}$ Definition C.1 ignores edges
(1) ☞ $\lfloor H_1 \rfloor_{P_1} \vdash^{p}_{\omega} e \Downarrow \lfloor H_2 \rfloor_{P_2}; t$ By above equality
(2) ☞ $\lfloor H_1 \rfloor_{P_1} \subseteq \lfloor H_2 \rfloor_{P_2}$ By above equality □