Typed Adapton: Refinement types for nominal memoization

Matthew A. Hammer[1] and Joshua Dunfield[2]

[1] University of Colorado Boulder
matthew.hammer@colorado.edu
[2] University of British Columbia
joshdunf@cs.ubc.ca

Abstract. Nominal memoization combines memoized functional programming with a controlled form of imperative cache effects. By leveraging these imperative effects, nominal memoization can dramatically outperform traditional ("structural") memoization. However, the nominal memoization programming model is error-prone: when the programmer unintentionally misuses names, their incremental program ceases to correspond to its purely functional semantics.
This paper develops a refinement type system for nominal memoization that enforces a program's correspondence with its purely functional semantics. Our type system employs set-indexed types in the style of DML (Xi and Pfenning 1999), extended with polymorphism over kinds and index functions. We prove that our type system enforces the dynamic side conditions required by Hammer et al. (2015). Past work shows that these conditions suffice for realistic uses of nominal memoization while also guaranteeing *from-scratch consistency* of the incremental programs. Furthermore, we propose a decidable form of name-level computations for expressing generic *naming strategies* in type-generic incremental code.

1 Introduction

Memoization is an evaluation technique that records a mapping from computation identities to computation results, enabling low-cost *reuse* of computations' results (Michie 1968; Pugh 1988). For example, the conventional identity of a function application is the function name and its argument.

We use the term *structural memoization* for memoization via this conventional notion of computation identity, where two comptuations are compared by comparing their functions and inputs, conservatively. This comparison is commonly implemented in $\mathsf{O}(1)$ time per pointer comparison by using a variant of hash-consing (Filliâtre and Conchon 2006). Hash-consing is an allocation strategy that hashes and shares identical structures, assigning them identical pointers. This allocation strategy allows memoization implementors to use pointer identity to efficiently test structural identity (e.g., of function inputs).

Nominal memoization uses a notion of *pointer name* identity to identify computations, as well as their inputs and outputs. Hammer et al. (2015) give a library of incremental programs that use first-class names and nominal memoization in otherwise purely-functional data structures and algorithms. Unlike structural memoization, which records immutable facts about functions, nominal memoization is closely tied to change propagation techniques for incremental computation, so that when input changes mutably (from an outside "mutator"), the runtime system can update the output of each computation by selectively revaluating only the directly-affected subcomputations.

In Sec. 2, we give a detailed example that maps a list of input elements to a list of output elements. Hammer et al. (2015) showed that, using an explicit *naming strategy* via first-class names, this `listmap` algorithm can respond to an imperative input change such as element insertion with worst-case $O(1)$ re-executions and fresh allocations, rather than the $O(n)$ required with structural memoization, which rebuilds and rehashes the prefix of the changed output.

The practical performance of nominal memoization stems directly from how the programmer designs a naming strategy. In the context of nominal memoization, a naming strategy is the sub-computation of an otherwise functional program that determines how first-class names in the function's inductive input structure become names and pointer names in the function's inductive output structure. To implement this strategy, programmers instrument an otherwise-functional program, including its input and output, with first-class names and incrementally-changing pointers. Each pointer is a *one-shot*: once written in a (from-scratch) run, and uniquely named, it is not mutated, and no other pointer is given the same name. Instead, changes occur at the *meta level* of the incremental computation, where input changes from the external environment induce output changes via a general-purpose change propagation algorithm.

Instrumenting a functional program with a naming strategy is error-prone, and making a mistake may change the meaning of the program, turning it into an imperative program whose from-scratch behavior no longer corresponds to a purely functional program, and whose incremental behavior may be unsound.

This paper presents a refinement type system for nominal memoization of purely functional programs. Our type system enforces that all programs behave as though they are purely functional, excluding behavior that overwrites prior allocations with later ones. In this work, we limit our focus to the semantics of one-shot store objects; prior work showed how the soundness of nominal memoization, including change propagation, follows directly from this semantic property (Hammer et al. 2015).

Compared with prior work, we extend the expressivity of naming strategies, permitting programs to be generic in how names of output structures are chosen, a pattern we refer to as *name parametricity.* This generality allows programmers to safely author and compose generic library code.

Contributions:

- We develop a type system for a variant of Nominal Adapton (Hammer et al. 2015). We use types to *statically* enforce that nominal memoization behaves

as though the program is purely functional, a condition modeled after the dynamic soundness criteria from Hammer et al. (2015).
In this work, unlike prior work, we only consider non-incremental runs. This focus is justified, since the property of having a functional semantics is a property of non-incremental runs. For functional correctness, we want to reason about incremental runs as non-incremental runs, and to reason about non-incremental runs as purely-functional runs, whose use of the store to name objects is inconsequential. Our type system enforces this latter property, whereas prior work established the former and *merely assumed* the latter for all programs.

– We formalize a type system that permits writing code that is generic in a *naming strategy*. In particular, we show various forms of *name- and name-function parametricity*, and illustrate through examples its importance for expressing nominal memoization in composable library code.
– To encode the necessary invariants on names, our type system uses set-indexed types in the style of DML (Xi and Pfenning 1999), extended with polymorphism over kinds and index functions; the latter was inspired by abstract refinement types (Vazou et al. 2013).

2 Overview: `listmap` example series

In this section, we consider the higher-order function `listmap` that, given a function over integers, maps an input list of integers to an output list in a pointwise fashion. We use this familar function to illustrate two connected activities: (1) augmenting functional programs with naming strategies (to use first-class names and nominal memoization), and (2) generalizing these naming strategies to write composable library functions.

Below, we consider several versions of `listmap` that only differ in their naming strategy: The names given to allocations differ, but nothing else about the algorithm does. In each case, we show the naming strategy in terms of both its program text and type, each of which reflect how names flow from input structures into output structures. In particular, the refinement type structure exposes the naming strategy of each version, and it creates a practical static abstraction for enforcing a global naming strategy with composable parts.

The type below defines incremental lists of integers, ($\mathsf{List}[X;Y]\ \mathsf{Int}$). This type has two conventional constructors, `Nil` and `Cons`, as well as two additional constructors that use the two type indices X and Y. The `Name` constructor permits names from set X to appear in the list sequence; it creates a `Cons`-cell-like pair holding a first-class name from X and the rest of the sequence. The `ref` constructor permits injecting (incrementally changing) references to lists into the list type; these pointers have names drawn from set Y.

$$\begin{array}{rlrl}
\mathtt{Nil}: & \forall X, Y. & & \mathsf{List}[X;Y]\ \mathsf{Int} \\
\mathsf{Cons}: & \forall X, Y. & \mathsf{Int} \rightarrow \mathsf{List}[X;Y]\ \mathsf{Int} \rightarrow & \mathsf{List}[X;Y]\ \mathsf{Int} \\
\mathtt{Name}: & \forall X_1, X_2, Y. & \mathsf{Nm}[X_1] \rightarrow \mathsf{List}[X_2;Y]\ \mathsf{Int} \rightarrow & \mathsf{List}[X_1 \perp X_2;Y]\ \mathsf{Int} \\
\mathsf{Ref}: & \forall X, Y_1, Y_2. & \mathsf{Ref}[Y_1]\ (\mathsf{List}[X;Y_2]\ \mathsf{Int}) \rightarrow & \mathsf{List}[X;Y_1 \perp Y_2]\ \mathsf{Int}
\end{array}$$

For both of these latter forms, the constructor's types enforce that each name (or pointer name) in the list is disjoint from those in the remainder of the list. For instance, we write $X_1 \perp X_2$ for the disjoint union of X_1 and X_2, and similarly for $Y_1 \perp Y_2$. However, the names in X and pointer names in Y may overlap (or even coincide); in particular, we use the type $\mathsf{List}\,[X;X]\;\;\mathsf{Int}$ in the output type of `listmap1`, below.

The `Nil` constructor creates an empty sequence with *any* pointer or name type sets in its resulting type. For practical reasons, we find it helpful to permit types to be *over-approximations* of the names actually used in each instance. In light of this, the type of the constructor for `Nil` makes sense.

In each naming strategy for `listmap`, we decide how to map names from the input list to names and pointers in the output list. Note that we use the names, not the pointer names, of the input list to produce the names and pointers of the output list.

2.1 listmap1: Naive nominal memoization

First, we consider the simplest naming strategy for `listmap`, where we temporarily ignore writing composable libraries. The input list consists of names from set X and pointers from set Y, and we produce an output list whose names and pointers consist of the names from set X. Notice how the function type communicates these relationships (that is, that name set X is the first index of the input list, and both indices of the output list).

```
listmap1 : ∀X,Y : NmSet. (Int → Int) → (List [X;Y]  Int) → (List [X;X]  Int)
listmap1 = λf. fix rec. λl.
   match l with
    Nil        ⇒ Nil,
    Cons(h,t)  ⇒ Cons(f h, rec t)
    Ref(r)     ⇒ rec (get r)
    Name(n,t)  ⇒ Name(n, Ref(memo_ref n [ rec t ]))
```

Turning to the program text for `listmap1`, the `Nil` and `Cons` cases are entirely conventional. The latter two cases handle pointers and names in the list. In the `Ref` case, the programmer observes the contents of the reference cell holding the remainder of the input, and recurs. In the `Name` case, the programmer injects the name `n` into the output list, but also uses this name in the construct `memo_ref` to (deterministically) allocate named a reference cell, and a memoized recursive call on the list tail. We use `memo_ref` to combine these steps into one step that consumes a single name. In fact, this construct decomposes into a special combination of simpler steps (viz., reference allocation, thunk allocation, and thunk forcing). Sec. 3 presents a formalism for these decomposed steps.

Example demanded computation graph (DCG). Suppose that we execute `listmap1` on an input list with two integers, two names n_1 and n_2, and two reference cells p_1 and p_2, as shown in Fig. 1. Recall that `listmap1` employs a naming strategy that uses the names n_1 and n_2 from input list `inp` as the names *and* pointers in output list `out`, shown below the input list, connected by various graph edges. Bold boxes denote references in the input and output lists, and bold circles denote memoized thunks that compute with them. Each bold store object (thunk or reference) is associated with a name.

The two thunks, shown as bold circles named by n_1 and n_2, follow a similar pattern: each has outgoing edges to the `Name` list structure to which they correspond (north- and south-west edges), the reference that they dereference (north-east), the recursive thunk that they force, if any (due west), and the reference that they name and allocate (southeast), named either n_1 or n_2.

The benefit of using names to identify these memoized thunks and their output list references is that imperative $O(1)$ changes to the input structure can be reflected into the DCG with only $O(1)$ changes to its named content. Further, the names also provide a primitive form of *cache eviction*: Memoized results are *overwritten* when names are re-associated with new content. By contrast, structural memoization generally requires rebuilding recursive structures whose content changes; in this case, an $O(1)$ change to `inp` will generally require re-hashing and rebuilding a linear number of output cells, all of which duplicate the content of an existing cell, modulo the $O(1)$ change. This contrast is detailed in prior work (Hammer et al. 2015).

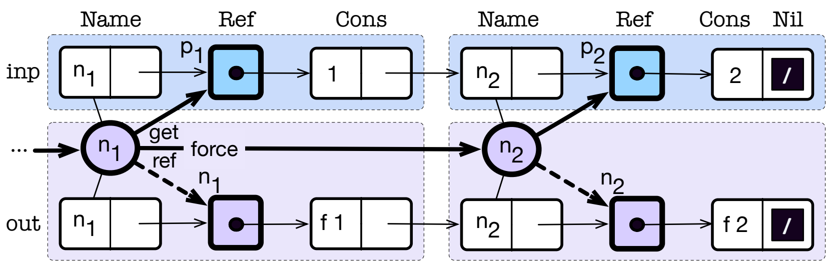

Fig. 1. DCG of `listmap1` on `inp`, a two-integer, two-name input list. Its names are drawn from set $X = \{n_1, n_2, \ldots\}$, and its pointer names are drawn from the set $Y = \{p_1, p_2, \ldots\}$. The two lower purple sections consist of DCG nodes (the four store objects named by n_1 and n_2) that result from two calls to `memo_ref`. The DCG consists of these objects and their relationships, shown as bold directed edges (the unbold directed edges illustrate immutable data; the undirected edges are just for illustration).

Unintended imperative effects: Unintended feedback and churn. Unfortunately, the naming strategy of `listmap1` is not immediately composable in larger programs. Consider the following situations:

1. **Unintended feedback**: If the input pointer names are drawn from the same set as those in the `Name` cells, then the allocations that `listmap1` performs will overwrite these input pointers with output pointers. Note that above, if $X = Y$, this is indeed possible.

2. **Unintended churn**: If the same input list is mapped twice in the same incremental program with two different element map functions, each alternation from one to the other will overwrite the prior with the latter.

    ```
    let xs = listmap1 abs inp
    let ys = listmap1 sq  inp
    ```

    For example, the mapping of `sq` will overwrite the mapping of `abs`, since the output pointers of each mapping coincide.
    Likewise, if another function, e.g., `listfilter`, uses the same naming strategy as `listmap1`, and these functions are composed to create a larger incremental function pipeline, the first function will overwrite the output of the second computation, rather than retaining incrementally-changing versions of both simultaneously.

Both feedback and churn arise because the names equip the program with a dynamic semantics that overlays an *imperative* allocation strategy atop an otherwise *purely functional* algorithm. Generally, as programmers, we want to exploit this imperative allocation strategy to enable efficient incremental computations, via a clever use of imperative overwriting, memoization, and dependency tracking. However, in the cases where the imperative, incremental changes are unintended, they lead to unintended behavior, either in terms of performance, or correctness, or both. The sections below generalize the naming strategy of `listmap1`, overcoming these limitations.

2.2 listmap2: Name parametricity for the output list

Fortunately, *first-class names* allow us to begin addressing both problems described above:

```
listmap2 : ∀X,Y : NmSet. ∀Z : Nm. Nm[Z] → (Int → Int) → (List[X;Y] Int)
           → (List[X; (Z, X)] Int)
listmap2 = λm. λ f. fix rec. λ l.
   match l with ···
    Name(n,t) ⇒ Name(n, Ref(memo_ref (m, n) [ rec t ]))
```

We adapt the original program by adding a new parameter, first-class name `m`, of type `Nm[Z]` and in the `Name` case of the pattern match, we name the output structure by the name pair `(m,n)` instead of just the name `n`. Statically, the index of the output list's reference cells is (Z, X) instead of merely X, allowing us to use different names in set Z to distinguish different, simultaneous uses of the function.

Because it is parametric in the function `f` *as well as* the distinguished name `m`, a calling context may use `listmap` to instantiate different integer mappings, as follows:

```
let x = listmap2 m1 abs inp
let y = listmap2 m2 sq  inp
```

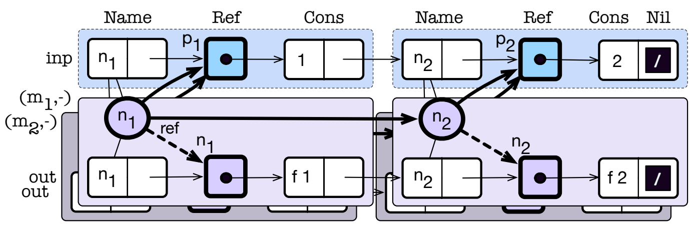

Fig. 2. Using distinct names $m_1 \perp m_2$ to create disjoint namespaces $(m_1, -)$ and $(m_2, -)$

Assuming that the names `m1` $\in Z$ and `m2` $\in Z$ are distinct (or "apart"), which we write as `m1` $\perp$ `m2`, we have that the reference cells and memoized calls for `x` and `y` are distinct from each other, and co-exist without overwriting one another. Fig. 2 illustrates this scenerio graphically, with two disjoint layers of output.

Otherwise, if we had that `m1` $=$ `m2`, the code would exhibit the churn pattern described above, which may not be intended. The type system proposed by this paper rules out this behavior, as well as all other cases of churn or feedback.

2.3 listmap3: Name-function parametricity for the output list

When writing reusable library code, the programmer may want to avoid choosing the specifics of how to name the output list structure, and its memoized computation. For instance, when allocating the reference cell in the `Name` case above, rather than choose to pair the names m and n, she may want to be even more generic by defering even more decisions to the calling context. To express this generality, she parameterizes `listmap` with a name-transforming function `nmf`. She names the reference cells of the output list type by applying `nmf` to the names of the input list.

$$\texttt{listmap3} \;:\; \forall X, Y : \mathsf{NmSet}.\ \forall \texttt{nmf} : \mathsf{Nm} \to_{\mathsf{Nm}} \mathsf{Nm}.\ \ (\mathsf{Nm} \to_{\mathsf{Nm}} \mathsf{Nm})[\texttt{nmf}]$$
$$\to (\mathsf{Int} \to \mathsf{Int}) \to (\mathsf{List}[X;Y]\ \ \mathsf{Int}) \to (\mathsf{List}[X;\ \texttt{nmf}\ \ X\ ]\ \ \mathsf{Int})$$

```
listmap3 l = λnmf. λ f. fix rec. λ l.
   match l with
    ...
    Name(n,t) ⇒ Name(n, Ref (memo_ref (nmf n) [ rec t ]))
```

In terms of the new type features, the name function `nmf` is classified by the index sort $\mathsf{Nm} \to_{\mathsf{Nm}} \mathsf{Nm}$, which says that it maps names to names. (We also lift all name functions to map sets of names point-wise). To express the naming strategy in the program, the user binds the argument variable `nmf` of singleton type $(\mathsf{Nm} \to_{\mathsf{Nm}} \mathsf{Nm})[\texttt{nmf}]$, the type of a name function that coincides with the index-level term `nmf`. In terms of program text, this generalization applies the name function in the `Name` case, rather than pairing names, as in listmap2. Compared to the version above that is parametric in a distinguished name `m`, this version is more general; to recover the first version, let the name function `nmf` be one that pairs its argument with the distinguished name `m`, viz., `λa.(m,a)`, as asserted below:

```
listmap2 m f l = listmap3 (λ a.(m,a)) f l
```

When run on the same list, with two different name functions, the resulting DCG is analogous to that of `listmap2` in Fig. 2.

2.4 listmap4: Naming strategy for higher-order parametricity

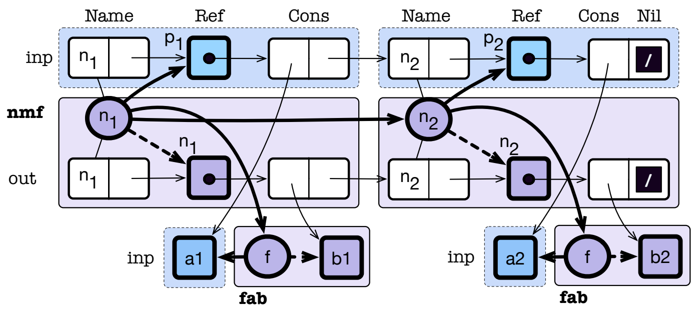

Fig. 3. `listmap4` permits higher order composition, with a non-trivial element mapping function `f`, and output element namespace `fab`. Though drawn as single references and thunks, the DCG of the element mapping `f` could be any generic structure, including the mapping of another inductive data structure.

Suppose we want a map function that works over any type of lists, not just lists of integers. To do so with maximum generality, we may also want the computation to be generic in the naming relationships of the input and output list elements. The programmer achieves this generality by abstracting over several structure-specific choices: The input and output list's element type structure, call it A and B, respectively, as well as the name structure of these types, and their relationship. In particular, we introduce another name function `fab` to abstract over the ways that element map function `f` uses the names in structures of type A to name of the output element structures of type B. To classify the types A and B, we use the kinds $k_A \Rightarrow \star$ and $k_B \Rightarrow \star$, respectively; the index sorts k_A and k_B classify the name-based indices of these types, and the index sort $k_A \rightarrow_{\mathsf{Nm}} k_B$ classifies the name function `fab`, which relates names of type A to names of type B.

$$
\begin{array}{l}
\forall X, Y : \mathsf{NmSet}.\, \forall \mathtt{nmf} : \mathsf{Nm} \rightarrow_{\mathsf{Nm}} \mathsf{Nm}.\\
\forall A : k_A \Rightarrow \star.,\, \forall B : k_B \Rightarrow \star.\;\; \forall \mathtt{fab} : k_A \rightarrow_{\mathsf{Nm}} k_B.\;\; \forall Z : k_A.\\
(\mathsf{Nm} \rightarrow_{\mathsf{Nm}} \mathsf{Nm})\,[\mathtt{nmf}]\\
\quad \rightarrow (\, A\,[Z] \;\rightarrow\; B\,[\mathtt{fab}\ Z]\,)\\
\quad \rightarrow (\mathsf{List}\,[X; Y]\;\; A\,[Z]\,) \rightarrow (\mathsf{List}\,[X; \mathtt{nmf}\ X]\;\; B\,[\mathtt{fab}\ Z]\,)
\end{array}
$$

In the transition from listmap3 to listmap4, neither the program text nor the algorithm change, only the type, given above.

listmap1:	listmap2:	listmap3:	listmap4:
$\forall X, Y : \mathsf{NmSet}.$	$\forall X, Y : \mathsf{NmSet}.$	$\forall X, Y : \mathsf{NmSet}.$	$\forall X, Y : \mathsf{NmSet}.$
	$\forall Z : \mathsf{Nm}.$	$\forall \mathtt{nmf} : \mathsf{Nm} \to_{\mathsf{Nm}} \mathsf{Nm}.$	$\forall \mathtt{nmf} : \mathsf{Nm} \to_{\mathsf{Nm}} \mathsf{Nm}.$
			$\forall A : k_A \Rightarrow \star.$ $\forall B : k_B \Rightarrow \star.$
			$\forall \mathtt{fab} : k_A \to_{\mathsf{Nm}} k_B.$ $\forall Z : k_A.$
		$(\mathsf{Nm} \to_{\mathsf{Nm}} \mathsf{Nm})[\mathtt{nmf}]$	$(\mathsf{Nm} \to_{\mathsf{Nm}} \mathsf{Nm})[\mathtt{nmf}]$
$(\mathsf{Int} \to \mathsf{Int})$	$(\mathsf{Int} \to \mathsf{Int})$	$\to (\mathsf{Int} \to \mathsf{Int})$	$\to A[Z] \to B[\mathtt{fab}\ Z]$
$\to \mathsf{List}\,[X; Y]\ \mathsf{Int}$	$\to \mathsf{List}[X; Y]\ \mathsf{Int}$	$\to \mathsf{List}[X; Y]\ \mathsf{Int}$	$\to \mathsf{List}[X; Y]\ A[Z]$
$\to \mathsf{List}\,[X; X]\ \mathsf{Int}$	$\to \mathsf{List}[X; (Z, X)]\ \mathsf{Int}$	$\to \mathsf{List}[X; \mathtt{nmf}\ X]\ \mathsf{Int}$	$\to \mathsf{List}[X; \mathtt{nmf}\ X]\ B[\mathtt{fab}\ Z]$

Fig. 4. Comparison of types for listmap, including differences.

Fig. 4 summarizes the four versions of listmap that we toured above. Each highlights differences from the prior, with listmap1 indicating the type indices that decorate the "usual" type for listmap : $(\mathsf{Int} \to \mathsf{Int}) \to \mathsf{List\ Int} \to \mathsf{List\ Int}$. Next, listmap2 is parametric in a name, which it pairs with the input names; listmap3 generalizes this pattern to any name function nmf, not just pair-producing ones; listmap4 generalizes the mapping function to relate any two types, A and B, which any naming strategy fab.

2.5 Monadic scopes hide the plumbing of a naming strategy

In the final version of listmap above, we see that being fully parametric in both the type and the naming strategy means introducing a lot of type structure that can be viewed as "plumbing". Specifically, for each name-index function, we must instrument the code to abstract over and apply the name function when the corresponding allocations occur (e.g., as with fab and nmf above). This adds some bookkeeping to both the types and the programs that inhabit them.

In some cases, it is useful to obey a more restrictive *monadic scoping* discipline, which helps hide the plumbing of generic programs in the style of listmap4, above. Specifically, we use the original, simplest version listmap1, along with caller-named scopes that disambiguate different uses of the names in the common input list:

```
let x = scope m1 [ listmap1 abs inp ]
let y = scope m2 [ listmap1 sq  inp ]
```

The resulting computation composes without overwriting itself, analogous to the DCG of listmap2 in Fig. 2. Intuitively, the use of scope merely hides the explicit name function parameter (either m1 or m2, above), threading it behind the scenes until an allocation or memoization point occurs, where the name function maps the given name into a particular *namespace*. In this sense, `scope` controls a monadic piece of state that is implicitly carried throughout a dynamic scope of computation.

Limitation of naming with scopes: No sharing. While permitting economical sizes of code and types, the key disadvantage of `scope` is exactly that it makes subcomputations completely disjoint. In particular, suppose that two distinct lists share common elements, and the programmer wants to map both lists with the same mapping function, sharing the work of mapping elements common to both lists; `listmap4` can express this naming strategy, whereas naming strategies that use `scope` will fail to express any sharing between the two sub-computations.

3 Program Syntax

$$
\begin{array}{lll}
\text{Values} & v ::= & x \mid () \mid (v_1, v_2) \mid \mathtt{inj}_i\, v \mid \mathtt{name}\; n \mid \mathtt{nmfn}\; M \mid \mathtt{ref}\; n \mid \mathtt{thunk}\; n \\
\text{Terminal exprs.} & t ::= & \mathtt{ret}(v) \mid \lambda x.\, e \\
\text{Expressions} & e ::= & t \mid \mathtt{split}(v, x_1.x_2.e) \mid \mathtt{case}(v, x_1.e_1, x_2.e_2) \\
 & & \mid e\; v \mid \mathtt{let}(e_1, x.e_2) \mid \mathtt{thunk}(v, e) \mid \mathtt{force}(v) \mid \mathtt{ref}(v, v) \mid \mathtt{get}(v) \\
 & & \mid \mathtt{scope}(M, e) \mid M\; v
\end{array}
$$

Fig. 5. Syntax of expressions

Fig. 5 gives the grammar of values v and expressions e. We use call-by-push-value (CBPV) conventions in this syntax, and in the type system that follows. There are several reasons for this: CBPV can be interpreted as a "neutral" evaluation order that includes both call-by-value or call-by-name, but prefers neither in its design. Further, since we make the unit of memoization a thunk, and since CBPV makes explicit the creation of thunks and closures, it exposes exactly the structure that we wish to extend as a general-purpose abstraction for incremental computation. In particular, thunks are the means by which we cache results and track dynamic dependencies.

Values v consist of variables, the unit value, pairs, sums, and several special forms (described below).

We separate values from expressions, rather than considering values to be a subset of expressions. Instead, *terminal expressions* t are a subset of expressions. A terminal expression t is either $\mathtt{ret}(v)$—the expression that returns the value v—or a λ. Expressions e include terminal expressions, elimination forms for pairs, sums, and functions (`split`, `case` and $e\,v$, respectively); let-binding (which evaluates e_1 to $\mathtt{ret}(v)$ and substitutes v for x in e_2); introduction (`thunk`) and elimination (`force`) forms for thunks, and introduction (`ref`) and elimination (`get`) forms for pointers (reference cells that hold values).

The special forms of values are names $\mathtt{name}\; n$, name-level functions $\mathtt{nmfn}\; M$, references (pointers), and thunks. References and thunks include a name n; this is not the contents of the reference or thunk, but the name *of* the reference or thunk.

The syntax described so far follows that of prior work on Adapton, including Hammer et al. (2015). We add the notion of a *name function*, which captures the idea of a namespace, as well as operations such as "forking" names. The scope construct controls monadic state for the current name function, adding a function to its function composition for the dynamic extent of its subexpression e. The name function application form, $M\ v$, permits programs to compute with names and name functions that reside within the type indices. Since these name functions always terminate, they do not affect a program's termination behavior.

We do not distinguish syntactically between value pointers (for reference cells) and thunk pointers (for suspended expressions); the store maps pointers to either of these.

Desugaring memo_ref. We can desugar the syntax memo_ref into the following construction of primitive operations: the creation and elimination of a named thunk that allocates a named reference cell. We distinguish the two name uses with the tags 1 and 2:

$$\mathsf{memo\_ref}\ n\ [e] = \mathsf{force}\ \big(\mathsf{thunk}\ n.1\ (\mathsf{let}\, x = e\, \mathsf{in}\, \mathsf{ref}(n.2, x))\big)$$

4 Type System

The structure of our type system is inspired by Dependent ML (Xi and Pfenning 1999; Xi 2007). In DML—unlike a fully dependently typed system—the system is separated into a *program level* and a less-powerful *index level.* The classic DML index domain is integers with linear inequalities, making type-checking decidable. Our index domain includes names, sets of names, and functions over names. Such functions constitute a tiny domain-specific language that is powerful enough to express useful transformations of names, but preserves decidability of type-checking.

In DML, indices usually have no direct computational content. For example, when applying a function on vectors that is indexed by vector length, the length index is not directly manipulated at run time. However, indices do reflect properties of run-time values. The simplest case is that of an indexed *singleton type*, such as $\mathsf{Int}[k]$. Here, the ordinary type Int and the index domain of integers are in one-to-one correspondence; the type $\mathsf{Int}[3]$ has one value, the integer 3.

While indexed singletons work well for the classic index domain of integers, they are less suited to names—at least for our purposes. Unlike integer constraints, where integer literals are common in types—for example, the length of the empty list is 0—literal names are rare in types. Many of the name constraints we need to express look like "given a value of type A whose name in the

$$\begin{array}{lrcl}
\text{Name terms} & M & ::= & n \mid () \mid \lambda a.\, M \mid a \mid M(M) \mid M.1 \mid M.2 \mid (M_1,\, M_2) \\
\text{Index sorts} & \gamma & ::= & \mathsf{Nm} \mid \mathsf{NmSet} \mid 1 \mid \gamma * \gamma \mid \gamma \to_{\mathsf{Nm}} \gamma \\
\text{Names} & n, m, p, q & ::= & \mathtt{root} \mid n.1 \mid n.2 \mid (n_1,\, n_2) \\
\text{Index variables} & a & & \\
\text{Index exprs.} & i, X, Y, Z, R, W & ::= & a \mid \{n\} \mid X \perp Y \mid () \mid (i,\, i) \mid M\,[i]
\end{array}$$

Fig. 6. Syntax of indices, including names

set X, this function produces a value of type B whose name is in the set $f(X)$". A DML-style system can express such constraints, but the types become verbose:

$$\begin{array}{l}\forall a : \mathsf{Nm}.\ \forall X : \mathsf{NmSet}. \\ \quad (a \in X) \supset \big(A\,[a] \to B\,[f(a)]\big)\end{array}$$

The notation is taken from one of DML's descendants, Stardust (Dunfield 2007). The type is read "for all names a and name sets X, such that $a \in X$, given some $A\,[a]$ the function returns $B\,[f(a)]$".

We avoid such locutions by indexing single values by name sets, rather than names. For types of the shape given above, this cuts the number of quantifiers in half, and obviates the $\in$-constraint attached via $\supset$:

$$\forall X : \mathsf{NmSet}.\ A\,[X] \to B\,[f(X)]$$

This type says the same thing as the earlier one, but now the approximations are expressed within the indexing of A and B. Note that f, a function on names, is interpreted pointwise: $f(X) = \{f(x) \mid x \in X\}$.

(Standard singletons will come in handy for index functions on names, where one usually needs to know the specific function.)

For aggregate data structures such as lists, indexing by a name set denotes an *overapproximation* of the names present. That is, the proper DML type

$$\begin{array}{l}\forall Y : \mathsf{NmSet}.\ \forall X : \mathsf{NmSet}. \\ \quad (Y \subseteq X) \supset \big(A\,[Y] \to B\,[f(Y)]\big)\end{array}$$

can be expressed by $\forall X : \mathsf{NmSet}.\ \big(A\,[X] \to B\,[f(X)]\big)$.

Following call-by-push-value (Levy 1999, 2001), we distinguish *value types* from *computation types*. Our computation types will also model effects, such as the allocation of a thunk with a particular name.

4.1 Index Level

Figure 6 gives the syntax of names, name terms (which include simple total computations over names), index sorts (which classify indices), and index expressions. We use several meta-variables for index expressions; by convention, X, Y, Z, R and W are used for sets of names.

Kinds	K	$::=$	$\star \mid \star \Rightarrow K \mid \gamma \Rightarrow K$
Effects	ϵ	$::=$	$\langle W; R\rangle$
Type constructors	d		
Type variables	α		
Value types	A, B	$::=$	$\alpha \mid d \mid A + B \mid A \times B \mid \mathsf{unit} \mid \mathsf{Ref}\,[i]\ A \mid \mathsf{Thk}\,[i]\ E$
			$\mid A\,[i] \mid A\ B \mid \mathsf{Nm}\,[i] \mid (\mathsf{Nm} \rightarrow_{\mathsf{Nm}} \mathsf{Nm})\,[M]$
Computation types	C, D	$::=$	$\mathsf{F}\,A \mid A \rightarrow E$
... with effects	E	$::=$	$C \rhd \epsilon \mid \forall\alpha : K.\, E \mid \forall a : \gamma.\, E$
Typing contexts	Γ	$::=$	$\cdot \mid \Gamma, p : A \mid \Gamma, p : E \mid \Gamma, x : A$

Fig. 7. Syntax of kinds, effects and types

Names Names are the value form of name terms. A name n is either a symbol s, the root name $\mathtt{root}$, the "halves" $n.1$ and $n.2$, or a pair of two names (n_1, n_2).

Name terms Name terms model names and functions over names; they correspond to terms in a λ-calculus extended with a type of names. A name term M is either a name (value) n, the unit name $()$, a function $\lambda a.\, M$ or argument a, application $M_1\ M_2$, the tuple (M_1, M_2), or a "half" ($M.1$ or $M.2$). Name terms do *not* allow recursion or destruction: a name function cannot case-analyze its argument.

We write $M \Downarrow_{\mathsf{M}} V$ for big-step evaluation of name terms; the rules are given in Figure 12.

We write $M \equiv M'$ when name terms M and M' are convertible, that is, applying a series of β-reductions and/or β-expansions to one term results in the other.

Name sets Supposing we give a name to each element of a list. Then the entire list should carry the set of those names. We write $\{n\}$ for the singleton name set, and $X \perp Y$ for a union of two sets X and Y that requires X and Y to be disjoint.

Indices An index i (also written X, Y, ... when the index is a set of names) is either an index-level variable a, a name set $\{n\}$ or $X \perp Y$, the unit index $()$, a pair of indices (i_1, i_2), or a name term M applied to an index $M\,[i]$. For example, if $M = (\lambda a.\, a.1.2)$ then $M\,[\mathsf{root}] = \mathsf{root}.1.2$.

Sorts The index level has its own type system; to reduce confusion, types at this level are called index *sorts.* The sort Nm classifies indices that are single names, and the sort NmSet classifies name sets. To combine index sorts γ_1 and γ_2, use the product sort $\gamma_1 * \gamma_2$. Finally, the sort $\rightarrow_{\mathsf{Nm}}$ classifies index-level functions: $\mathsf{Nm} \rightarrow_{\mathsf{Nm}} \mathsf{Nm}$ takes a name and returns a name.

Entailment We assume an entailment relation $\Gamma \vdash P$, where P is a set-theoretic proposition such as $n \in X$ or $X \subseteq Y$ or $X \perp Y$; the latter is interpreted as $(X \cap Y) = \emptyset$.

We assume that entailment is closed under conversion: for example, if $M(n) \equiv n'$, then $\Gamma \vdash M(n) \in N$ iff $\Gamma \vdash n' \in N$. We also assume that weakening holds: if $\Gamma_1, \Gamma_3 \vdash P$ then $\Gamma_1, \Gamma_2, \Gamma_3 \vdash P$.

4.2 Kinds

$\boxed{\Gamma \vdash M : \gamma}$ Under Γ, name term M has sort γ

$$\dfrac{\Gamma \vdash n \text{ wf-name}}{\Gamma \vdash n : \mathsf{Nm}}\ \text{M-const} \qquad \dfrac{\Gamma, a : \gamma_1 \vdash M : \gamma_2}{\Gamma \vdash (\lambda a.\, M) : (\gamma_1 \rightarrow_{\mathsf{Nm}} \gamma_2)}\ \text{M-abs} \qquad \dfrac{(a : \gamma) \in \Gamma}{\Gamma \vdash a : \gamma}\ \text{M-var}$$

$$\dfrac{\Gamma \vdash M_1 : (\gamma' \rightarrow_{\mathsf{Nm}} \gamma) \qquad \Gamma \vdash M_2 : \gamma'}{\Gamma \vdash (M_1\ M_2) : \gamma}\ \text{M-app} \qquad \dfrac{}{\Gamma \vdash () : 1}\ \text{M-unit}$$

$$\dfrac{\Gamma \vdash M_1 : \gamma_1 \qquad \Gamma \vdash M_2 : \gamma_2}{\Gamma \vdash (M_1, M_2) : (\gamma_1 * \gamma_2)}\ \text{M-pair} \qquad \dfrac{\Gamma \vdash M : \mathsf{Nm}}{\Gamma \vdash M.1 : \mathsf{Nm} \qquad \Gamma \vdash M.2 : \mathsf{Nm}}\ \text{M-append}$$

$\boxed{\Gamma \vdash i : \gamma}$ Under Γ, index i has sort γ

$$\dfrac{(a : \gamma) \in \Gamma}{\Gamma \vdash a : \gamma}\ \text{sort-var} \qquad \dfrac{\Gamma \vdash n : \mathsf{Nm}}{\Gamma \vdash \{n\} : \mathsf{NmSet}}\ \text{sort-singleton}$$

$$\dfrac{\Gamma \vdash X : \mathsf{NmSet} \qquad \Gamma \vdash Y : \mathsf{NmSet}}{\Gamma \vdash (X \perp Y) : \mathsf{NmSet}}\ \text{sort-sep-union} \qquad \dfrac{}{\Gamma \vdash () : 1}\ \text{sort-unit}$$

$$\dfrac{\Gamma \vdash i_1 : \gamma_1 \qquad \Gamma \vdash i_2 : \gamma_2}{\Gamma \vdash (i_1, i_2) : (\gamma_1 * \gamma_2)}\ \text{sort-pair} \qquad \dfrac{\Gamma \vdash M : \gamma_1 \rightarrow_{\mathsf{Nm}} \gamma_2 \qquad \Gamma \vdash i : \gamma_1}{\Gamma \vdash M[i] : \gamma_2}\ \text{sort-apply}$$

$$\dfrac{\Gamma \vdash M : \mathsf{Nm} \rightarrow_{\mathsf{Nm}} \mathsf{Nm} \qquad \Gamma \vdash i : \mathsf{NmSet}}{\Gamma \vdash M[\![i]\!] : \mathsf{NmSet}}\ \text{sort-apply-pointwise}$$

Fig. 8. Sorts statically classify name terms M, and the name indices i that index types

We use a relatively simple system of *kinds* K (Figure 7) to classify the different animals in the type system:

- The kind $\star$ classifies value types, such as unit and $(\mathsf{Thk}[i]\ E)$.
- The kind $\star \Rightarrow K$ classifies type expressions that are parametrized by a type. Such types are called *type constructors* in some languages; for example, list by itself has kind $\star \Rightarrow \star$ (and $\mathsf{list}\ \mathsf{unit}$ has kind $\star$).
- The kind $\gamma \Rightarrow K$ classifies type expressions that are parametrized by an index. For example, the List type constructor from Section 2 takes two name

$\boxed{\Gamma \vdash A : K}$ Under Γ, value type A has kind K

$$\frac{(\alpha : \star) \in \Gamma}{\Gamma \vdash \alpha : \star}\ \text{k-typevar} \qquad \frac{(\mathsf{d} : K) \in \Gamma}{\Gamma \vdash \mathsf{d} : K}\ \text{k-tycon} \qquad \frac{\Gamma \vdash A_1 : \star \qquad \Gamma \vdash A_2 : \star}{\begin{array}{c}\Gamma \vdash (A_1 + A_2) : \star \\ \Gamma \vdash (A_1 \times A_2) : \star\end{array}}\ \text{k-binop}$$

$$\frac{}{\Gamma \vdash \mathsf{unit} : \star}\ \text{k-constant} \qquad \frac{\Gamma \vdash i : \mathsf{NmSet}}{\Gamma \vdash \mathsf{Nm}[i] : \star}\ \text{k-name}$$

$$\frac{\Gamma \vdash i : \mathsf{NmSet} \qquad \Gamma \vdash A : \star}{\Gamma \vdash (\mathsf{Ref}[i]\ A) : \star}\ \text{k-ref} \qquad \frac{\Gamma \vdash i : \mathsf{NmSet} \qquad \Gamma \vdash E \text{ efftype}}{\Gamma \vdash (\mathsf{Thk}[i]\ E) : \star}\ \text{k-thk}$$

$$\frac{\Gamma \vdash A : (\star \Rightarrow K) \qquad \Gamma \vdash B : \star}{\Gamma \vdash (A\ B) : K}\ \text{k-app-type} \qquad \frac{\Gamma \vdash A : (\gamma \Rightarrow K) \qquad \Gamma \vdash i : \gamma}{\Gamma \vdash A[i] : K}\ \text{k-app-index}$$

$\boxed{\Gamma \vdash C \text{ ctype}}$ Under Γ, computation type C is well-formed

$$\frac{\Gamma \vdash A : \star}{\Gamma \vdash (\mathsf{F}\,A) \text{ ctype}}\ \text{ctype-lift} \qquad \frac{\Gamma \vdash A : \star \qquad \Gamma \vdash E \text{ efftype}}{\Gamma \vdash (A \to E) \text{ ctype}}\ \text{ctype-arr}$$

$\boxed{\Gamma \vdash \epsilon \text{ wf-effects}}$ Under Γ, effect specification ϵ is well-formed

$$\frac{\Gamma \vdash W : \mathsf{NmSet} \qquad \Gamma \vdash R : \mathsf{NmSet}}{\Gamma \vdash \langle W; R \rangle \text{ wf-effects}}\ \text{wf-eff}$$

$\boxed{\Gamma \vdash E \text{ efftype}}$ Under Γ, type-with-effects E is well-formed

$$\frac{\Gamma \vdash C \text{ ctype} \qquad \Gamma \vdash \epsilon \text{ wf-effects}}{\Gamma \vdash (C \rhd \epsilon) \text{ efftype}}\ \text{etype-effects}$$

$$\frac{\Gamma, \alpha : K \vdash E \text{ efftype}}{\Gamma \vdash (\forall \alpha : K.\, E) \text{ efftype}}\ \text{etype-poly} \qquad \frac{\Gamma, a : \gamma \vdash E \text{ efftype}}{\Gamma \vdash (\forall a : \gamma.\, E) \text{ efftype}}\ \text{etype-poly-index}$$

Fig. 9. Kinds statically classify types and effects

sets and the type of the list elements, e.g. $\mathsf{List}\,[X;Y]\;\;\mathsf{Int}$. Therefore, List has kind $\mathsf{NmSet} \Rightarrow \big(\mathsf{NmSet} \Rightarrow (\star \Rightarrow \star)\big)$. (The inner kind $(\star \Rightarrow \star)$ is an instance of the $\star \Rightarrow \mathsf{K}$ kind, not the $\gamma \Rightarrow \mathsf{K}$ kind.)

4.3 Effects

Effects are described by $\langle W; R\rangle$, meaning that the associated code may write names in W, and may read names in R.

Effect sequencing is a (meta-level) partial function over a pair of effects: if ϵ_1 then ϵ_2 is defined and equal to ϵ, then ϵ describes the combination of having effects ϵ_1 followed by effects ϵ_2. Sequencing is a partial function because the effects are only valid when (1) the writes of ϵ_1 are disjoint from the writes of ϵ_2, and (2) the reads of ϵ_1 are disjoint from the writes of ϵ_2. Condition (1) holds when each cell or thunk is not written more than once (and therefore has a unique value). Condition (2) holds when each cell or thunk is written before it is read.

A second meta-level partial function, effect coalescing—written "E after ϵ"—combines "clusters" of effects. For example:

$$\big(C \rhd \langle\{n_2\};\emptyset\rangle\big)\ \mathsf{after}\ \langle\{n_1\};\emptyset\rangle \;=\; C \rhd \big(\langle\{n_1\};\emptyset\rangle\ \mathsf{then}\ \langle\{n_2\};\emptyset\rangle\big) \;=\; C \rhd \langle\{n_1,n_2\};\emptyset\rangle$$

4.4 Types

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

$$\frac{}{\Gamma \vdash () : \mathsf{unit}}\ \text{unit} \qquad \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\ \text{var} \qquad \frac{\Gamma \vdash v_1 : A_1 \qquad \Gamma \vdash v_2 : A_2}{\Gamma \vdash (v_1, v_2) : (A_1 \times A_2)}\ \text{pair}$$

$$\frac{\Gamma \vdash n \in X}{\Gamma \vdash (\mathtt{name}\ n) : \mathsf{Nm}\,[X]}\ \text{name} \qquad \frac{\Gamma \vdash M_v : (\mathsf{Nm} \rightarrow_{\mathsf{Nm}} \mathsf{Nm}) \qquad M_v \equiv M}{\Gamma \vdash (\mathtt{nmfn}\ M_v) : (\mathsf{Nm} \rightarrow_{\mathsf{Nm}} \mathsf{Nm})\,[M]}\ \text{namefn}$$

$$\frac{\Gamma \vdash n \in X \qquad \Gamma \vdash n\ \mathsf{has\text{-}store\text{-}type}\ A}{\Gamma \vdash (\mathtt{ref}\ n) : \mathsf{Ref}\,[X]\ A}\ \text{ref} \qquad \frac{\Gamma \vdash n \in X \qquad \Gamma \vdash n\ \mathsf{has\text{-}store\text{-}type}\ E}{\Gamma \vdash (\mathtt{thunk}\ n) : \big(\mathsf{Thk}\,[X]\ E\big)}\ \text{thunk}$$

Fig. 10. Value typing

The value types, written A, B, in Figure 7 include standard sums $+$ and products $\times$, a unit type, the type $\mathsf{Ref}\,[i]\ A$ of references named i containing a value of type A, the type $\mathsf{Thk}\,[i]\ E$ of thunks named i whose contents have type E (see below), the application $A\,[i]$ of a type to an index, the application $A\ B$ of a type A (e.g. a type constructor d) to a type B, the type $\mathsf{Nm}\,[i]$, and a singleton type $(\mathsf{Nm} \rightarrow_{\mathsf{Nm}} \mathsf{Nm})\,[M]$ where M is a function on names.

As usual in call-by-push-value, computation types C and D include a connective F, which "lifts" value types to computation types: $\mathsf{F}\,A$ is the type of

$\boxed{(\epsilon_1\ \mathsf{then}\ \epsilon_2) = \epsilon}$ Effect sequencing

$$\big(\langle W_1; R_1\rangle\ \mathsf{then}\ \langle W_2; R_2\rangle\big) \;=\; \langle W_1 \cup W_2; R_1 \cup R_2\rangle \qquad \text{if } W_1 \perp W_2 \text{ and } R_1 \perp W_2$$

$\boxed{(E\ \mathsf{after}\ \epsilon) = E'}$ Effect coalescing

$$\big((C \rhd \epsilon_2)\ \mathsf{after}\ \epsilon_1\big) \;=\; \big(C \rhd (\epsilon_1\ \mathsf{then}\ \epsilon_2)\big)$$

$$\begin{aligned} (\forall\alpha : K.\, E)\ \mathsf{after}\ \epsilon &= \forall\alpha : K.\, (E\ \mathsf{after}\ \epsilon) \\ (\forall a : \gamma.\, E)\ \mathsf{after}\ \epsilon &= \forall a : \gamma.\, (E\ \mathsf{after}\ \epsilon) \\ (P \supset E)\ \mathsf{after}\ \epsilon &= P \supset (E\ \mathsf{after}\ \epsilon) \end{aligned}$$

$\boxed{\Gamma \vdash^{M} e : E}$ Under Γ, within namespace M, computation e has type-with-effects E.

$$\frac{\Gamma \vdash^{M} e : (C \rhd \epsilon_1) \qquad \epsilon_1 \preceq \epsilon_2}{\Gamma \vdash^{M} e : (C \rhd \epsilon_2)}\ \text{eff-subsume}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{M} v : (A_1 \times A_2) \\ \Gamma, x_1 : A_1, x_2 : A_2 \vdash^{M} e : E\end{array}}{\Gamma \vdash^{M} \mathsf{split}(v, x_1.x_2.e) : E}\ \text{split} \qquad \frac{\Gamma \vdash^{M} v : (A_1 + A_2) \qquad \begin{array}{l}\Gamma, x_1 : A_1 \vdash^{M} e_1 : E \\ \Gamma, x_2 : A_2 \vdash^{M} e_2 : E\end{array}}{\Gamma \vdash^{M} \mathsf{case}(v, x_1.e_1, x_2.e_2) : E}\ \text{case}$$

$$\frac{\Gamma \vdash v : A}{\Gamma \vdash^{M} \mathsf{ret}(v) : \big((F\,A) \rhd \langle\emptyset;\emptyset\rangle\big)}\ \text{ret} \qquad \frac{\Gamma \vdash^{M} e_1 : (F\,A) \rhd \epsilon_1 \qquad \Gamma, x : A \vdash^{M} e_2 : (C \rhd \epsilon_2)}{\Gamma \vdash^{M} \mathsf{let}(e_1, x.e_2) : \big(C \rhd (\epsilon_1\ \mathsf{then}\ \epsilon_2)\big)}\ \text{let}$$

$$\frac{\Gamma, x : A \vdash^{M} e : E}{\Gamma \vdash^{M} (\lambda x.\, e) : \big((A \to E) \rhd \langle\emptyset;\emptyset\rangle\big)}\ \text{lam} \qquad \frac{\Gamma \vdash^{M} e : \big((A \to E) \rhd \epsilon_1\big) \qquad \Gamma \vdash v : A}{\Gamma \vdash^{M} (e\ v) : (E\ \mathsf{after}\ \epsilon_1)}\ \text{app}$$

$$\frac{\Gamma \vdash v : \mathsf{Nm}[X] \qquad \Gamma \vdash^{M} e : E}{\Gamma \vdash^{M} \mathsf{thunk}(v, e) : \big(F\,(\mathsf{Thk}[M(X)]\ E)\big) \rhd \langle M(X);\emptyset\rangle}\ \text{thunk}$$

$$\frac{\Gamma \vdash v : \mathsf{Thk}[X]\ (C \rhd \epsilon)}{\Gamma \vdash^{M} \mathsf{force}(v) : \big(C \rhd (\langle\emptyset;X\rangle\ \mathsf{then}\ \epsilon)\big)}\ \text{force}$$

$$\frac{\Gamma \vdash v_1 : \mathsf{Nm}[X] \qquad \Gamma \vdash v_2 : A}{\Gamma \vdash^{M} \mathsf{ref}(v_1, v_2) : \big(F\,(\mathsf{Ref}[M(X)]\ A)\big) \rhd \langle M(X);\emptyset\rangle}\ \text{ref} \qquad \frac{\Gamma \vdash v : \mathsf{Ref}[X]\ A}{\Gamma \vdash^{M} \mathsf{get}(v) : \big(F\,A\big) \rhd \langle\emptyset;X\rangle}\ \text{get}$$

$$\frac{\begin{array}{l}\Gamma \vdash v_M : (\mathsf{Nm} \to_{\mathsf{Nm}} \mathsf{Nm})[M] \\ \Gamma \vdash v : \mathsf{Nm}[i]\end{array}}{\Gamma \vdash^{N} (v_M\ v) : F\,(\mathsf{Nm}[M(i)]) \rhd \langle\emptyset;\emptyset\rangle}\ \text{name-app} \qquad \frac{\begin{array}{l}\Gamma \vdash N' : \mathsf{Nm} \to_{\mathsf{Nm}} \mathsf{Nm} \\ \Gamma \vdash^{N \circ N'} e : C \rhd \langle W;R\rangle\end{array}}{\Gamma \vdash^{N} \mathsf{scope}(N', e) : C \rhd \langle W;R\rangle}\ \text{scope}$$

$$\frac{\Gamma, a : \gamma \vdash^{M} t : E}{\Gamma \vdash^{M} t : (\forall a : \gamma.\, E)}\ \text{AllIndexIntro} \qquad \frac{\Gamma \vdash^{M} e : (\forall a : \gamma.\, E) \qquad \Gamma \vdash i : \gamma}{\Gamma \vdash^{M} e : [i/a]E}\ \text{AllIndexElim}$$

$$\frac{\Gamma, a : \gamma \vdash^{M} t : E}{\Gamma \vdash^{M} t : (\forall\alpha : K.\, E)}\ \text{AllIntro} \qquad \frac{\Gamma \vdash^{M} e : (\forall\alpha : K.\, E) \qquad \Gamma \vdash A : K}{\Gamma \vdash^{M} e : [A/\alpha]E}\ \text{AllElim}$$

Fig. 11. Computation typing

computations that, when run, return a value of type A. (Call-by-push-value usually has a connective dual to F, written U, that "th**U**nks" a computation type into a value type; in our system, Thk plays the role of U.)

Computation types also include functions, written $\mathsf{A} \to \mathsf{E}$. In standard CBPV, this would be $\mathsf{A} \to \mathsf{C}$, not $\mathsf{A} \to \mathsf{E}$. We separate computation types alone, written C, from computation types with effects, written E; this decision is explained below.

Computation types-with-effects E consist of $\mathsf{C} \rhd \epsilon$, which is the bare computation type C with effects ϵ, as well as universal quantifiers (polymorphism) over types ($\forall \alpha : \mathsf{K}.\, \mathsf{E}$) and indices ($\forall a : \gamma.\, \mathsf{E}$).

Why distinguish computation types from types-with-effects? Can we unify computation types C and types-with-effects E? Not easily. We have two computation types, F and $\to$. For F, the expression being typed could create a thunk, so we must put that effect somewhere in the syntax. For $\to$, applying a function is (per call-by-push-value) just a "push": the function carries no effects of its own (though its codomain may need to have some). However, suppose we force a thunked function of type $\mathsf{A}_1 \to (\mathsf{A}_2 \to \cdots)$ and apply the function (the contents of the thunk) to one argument. In the absence of effects, the result would be a computation of type $\mathsf{A}_2 \to \cdots$, meaning that the computation is waiting for a second argument to be pushed. But, since forcing the thunk has the effect of reading the thunk, we need to track this effect in the result type. So we cannot return $\mathsf{A}_2 \to \cdots$, and must instead put effects around $(\mathsf{A}_2 \to \cdots)$. Thus, we need to associate effects to both F and $\to$, that is, to both computation types.

Now we are faced with a choice: we could (1) extend the syntax of each connective with an effect (written next to the connective), or (2) introduce a "wrapper" that encloses a computation type, either F or $\to$. These seem more or less equally complicated for the present system, but if we enriched the language with more connectives, choice (1) would make the new connectives more complicated, while under choice (2), the complication would already be rolled into the wrapper. We choose (2), and write the wrapper as $\mathsf{C} \rhd \epsilon$, where C is a computation type and ϵ represents effects.

Where should these wrappers live? We could add $\mathsf{C} \rhd \epsilon$ to the grammar of computation types C. But it seems useful to have a clear notion of *the* effect associated with a type. When the effect on the outside of a type is the only effect in the type, as in $(\mathsf{A}_1 \to \mathsf{F}\,\mathsf{A}_2) \rhd \epsilon$, "the" effect has to be ϵ. Alas, types like $(\mathsf{C} \rhd \epsilon_1) \rhd \epsilon_2$ raise awkward questions: does this type mean the computation does ϵ_2 and then ϵ_1, or ϵ_1 and then ϵ_2?

We obtain an unambiguous, singular outer effect by distinguishing types-with-effects E from computation types C. The meta-variables for computation types appear only in the production $\mathsf{E} ::= \mathsf{C} \rhd \epsilon$, making types-with-effects E the "common case" in the grammar. Many of the typing rules follow this pattern, achieving some isolation of effect tracking in the rules.

$$\text{Name term values } V ::= n \mid \lambda a.\, M \mid a \mid () \mid (V, V) \mid V.1 \mid V.2$$

$\boxed{M \Downarrow_{M} V}$ Name term M evaluates to name term value V

$$\frac{}{V \Downarrow_{M} V}\ \text{teval-value} \qquad \frac{M_1 \Downarrow_{M} \lambda a.\, M \qquad M_2 \Downarrow_{M} V_2 \qquad [V_2/a]M \Downarrow_{M} V}{(M_1\, M_2) \Downarrow_{M} V}\ \text{teval-app}$$

$$\frac{M_1 \Downarrow_{M} V_1 \qquad M_2 \Downarrow_{M} V_2}{(M_1, M_2) \Downarrow_{M} (V_1, V_2)}\ \text{teval-tuple} \qquad \frac{M \Downarrow_{M} V}{\begin{array}{l} M.1 \Downarrow_{M} V.1 \\ M.2 \Downarrow_{M} V.2 \end{array}}\ \text{teval-append}$$

Fig. 12. Name term evaluation

5 Dynamics

5.1 Dynamics of name terms

Fig. 12 gives the dynamics for evaluating a name term M into a name term value V. Because name terms lack the ability to perform recursion and pattern-matching, it is easy to see that they always terminate.

5.2 Dynamics of program expressions

Fig. 13 defines graphs, the mutable state that names control dynamically. In this work, we are only interested in showing a correspondence to a purely functional run, so we need not model the cache and dependency-graph structure of prior work. As a result, our "graphs" are merely stores that map pointer names to values and expressions.

Fig. 14 defines the big-step evaluation relation for expressions, relating an initial and final graph, as well as a namespace and "current node" to a program. Because we do not build the dependency graph, the "current node" is not relevant here; the current namespace is used in rules ref and thunk to help name the allocated pointer. The shaded rules (including those two, and rules get and force for accessing their contents) change in the incremental version of the semantics, building and using the dependency graph structure that we omit here, for simplicity.

6 Metatheory: Type Soundness and "Pure" Effects

In this section, we prove that our type system and dynamics agree. Further, we show that the type system enforces a purely functional effect discipline, whose dynamics is codified formally by Def. 1, below. Our main theorem establishes that a well-typed, terminating program produces a terminal computation of the program's type, and that the actual dynamic effects are consistent with a purely functional allocation strategy. Specifically, sequenced writes never overwrite one another.

$$\begin{array}{lll} \text{Graphs } G,H & ::= \varepsilon & \text{empty graph} \\ & \mid G, p{:}v & p \text{ points to value } v \\ & \mid G, p{:}e\,@\,M & p \text{ points to thunk } e \text{ in namespace } M \text{ (no cached result)} \end{array}$$

Fig. 13. Graphs without dependency or cache structure

$\boxed{G_1 \vdash^M_m e \Downarrow G_2; t}$ Under graph G in namespace M at current node m, expression e produces new graph G_2 and result t.

$$\frac{}{G \vdash^M_m t \Downarrow G; t}\ \text{term} \qquad \frac{G_1 \vdash^M_m [v_2/x_2][v_1/x_1]e \Downarrow G_2; e'}{G_1 \vdash^M_m \mathsf{split}((v_1, v_2), x_1.x_2.e) \Downarrow G_2; e'}\ \text{split}$$

$$\frac{G_1 \vdash^M_m [v_i/x_i]e_i \Downarrow G_2; e'}{G_1 \vdash^M_m \mathsf{case}(\mathsf{inj}_i\, v, x_1.e_1, x_2.e_2) \Downarrow G_2; e'}\ \text{case} \qquad \frac{\begin{array}{c} G_1 \vdash^M_m e_1 \Downarrow G_1'; \mathsf{ret}(v) \\ G_1' \vdash^M_m [v/x]e_2 \Downarrow G_2'; e_2' \end{array}}{G_1' \vdash^M_m \mathsf{let}(e_1, x.e_2) \Downarrow G_2'; e_2'}\ \text{let}$$

$$\frac{\begin{array}{c} G_1 \vdash^M_m e_1 \Downarrow G_1'; \lambda x.\, e_2 \\ G_1 \vdash^M_m [v/x]e_2 \Downarrow G_2'; e_2' \end{array}}{G_1' \vdash^M_m e_1\ v \Downarrow G_2'; e_2'}\ \text{app} \qquad \frac{G_1 \vdash^{M_1 \circ M_2}_m e \Downarrow G_2; e'}{G_1 \vdash^{M_1}_m \mathsf{scope}(M_2, e) \Downarrow G_2; e'}\ \text{scope}$$

$$\frac{M_1 \Downarrow_M \lambda a.\, M_2 \qquad [n/a]M_2 \Downarrow_M p}{G \vdash^M_m M_1\ (\mathsf{name}\ n) \Downarrow G; \mathsf{ret}(\mathsf{name}\ p)}\ \text{name-app}$$

$$\frac{(M\, n) \Downarrow_M p \qquad G_1\{p \mapsto e\,@\,M\} = G_2}{G_1 \vdash^M_m \mathsf{thunk}(\mathsf{name}\ n, e) \Downarrow G_2; \mathsf{ret}(\mathsf{thunk}\ p)}\ \text{thunk}$$

$$\frac{(M\, n) \Downarrow_M p \qquad G_1\{p \mapsto v\} = G_2}{G_1 \vdash^M_m \mathsf{ref}(\mathsf{name}\ n, v) \Downarrow G_2; \mathsf{ret}(\mathsf{ref}\ p)}\ \text{ref}$$

$$\frac{\begin{array}{c} \mathsf{exp}(G_1, p) = e \qquad \mathsf{ns}(G_1, p) = M_0 \\ G_1 \vdash^{M_0}_p e \Downarrow G_2; t \end{array}}{G_1 \vdash^M_m \mathsf{force}(\mathsf{thunk}\ p) \Downarrow G_2; t}\ \text{force} \qquad \frac{G(p) = v}{G \vdash^M_m \mathsf{get}(\mathsf{ref}\ p) \Downarrow G; \mathsf{ret}(v)}\ \text{get}$$

Fig. 14. Dynamics for non-incremental evaluation

$$\frac{}{\cdot : \Gamma}\ \text{emp} \qquad \frac{G \vdash \Gamma \qquad \Gamma \vdash v : A \qquad \Gamma(p) = A}{\vdash (G, p : v) : \Gamma}\ \text{ref} \qquad \frac{G \vdash \Gamma \qquad \Gamma \vdash e : E \qquad \Gamma(p) = E}{\vdash (G, p : e) : \Gamma}\ \text{thunk}$$

Fig. 15. Store typing: $G \vdash \Gamma$, read “store G typed by Γ”.

6.1 Pure effects: Read and write sets

$$
\begin{array}{rll}
\mathcal{D}\ \mathsf{by}\ \text{Eval-term}()\ \mathsf{reads}\ \emptyset\ \mathsf{writes}\ \emptyset & & \\
\mathcal{D}\ \mathsf{by}\ \text{Eval-app}(\mathcal{D}_1,\mathcal{D}_2)\ \mathsf{reads}\ \mathsf{R}_1 \cup \mathsf{R}_2\ \mathsf{writes}\ W_1 \perp W_2 & \text{if} & \mathcal{D}_1\ \mathsf{reads}\ \mathsf{R}_1\ \mathsf{writes}\ W_1 \\
& \text{and} & \mathcal{D}_2\ \mathsf{reads}\ \mathsf{R}_2\ \mathsf{writes}\ W_2 \\
\mathcal{D}\ \mathsf{by}\ \text{Eval-bind}(\mathcal{D}_1,\mathcal{D}_2)\ \mathsf{reads}\ \mathsf{R}_1 \cup \mathsf{R}_2\ \mathsf{writes}\ W_1 \perp W_2 & \text{if} & \mathcal{D}_1\ \mathsf{reads}\ \mathsf{R}_1\ \mathsf{writes}\ W_1 \\
& \text{and} & \mathcal{D}_2\ \mathsf{reads}\ \mathsf{R}_2\ \mathsf{writes}\ W_2 \\
\mathcal{D}\ \mathsf{by}\ \text{Eval-scope}(\mathcal{D}_0)\ \mathsf{reads}\ \mathsf{R}\ \mathsf{writes}\ W & \text{if} & \mathcal{D}_0\ \mathsf{reads}\ \mathsf{R}\ \mathsf{writes}\ W
\end{array}
$$

$$
\begin{array}{rll}
\mathcal{D}\ \mathsf{by}\ \mathsf{Eval\text{-}fix}(\mathcal{D}_0)\ \mathsf{reads}\ \mathsf{R}\ \mathsf{writes}\ W & \text{if} & \mathcal{D}_0\ \mathsf{reads}\ \mathsf{R}\ \mathsf{writes}\ W \\
\mathcal{D}\ \mathsf{by}\ \mathsf{Eval\text{-}case}(\mathcal{D}_0)\ \mathsf{reads}\ \mathsf{R}\ \mathsf{writes}\ W & \text{if} & \mathcal{D}_0\ \mathsf{reads}\ \mathsf{R}\ \mathsf{writes}\ W \\
\mathcal{D}\ \mathsf{by}\ \mathsf{Eval\text{-}split}(\mathcal{D}_0)\ \mathsf{reads}\ \mathsf{R}\ \mathsf{writes}\ W & \text{if} & \mathcal{D}_0\ \mathsf{reads}\ \mathsf{R}\ \mathsf{writes}\ W \\
\mathcal{D}\ \mathsf{by}\ \mathsf{Eval\text{-}ref}()\ \mathsf{reads}\ \emptyset\ \mathsf{writes}\ q & \text{where} & e = \mathtt{ref}(\mathtt{name}\ n, v)\ \text{and}\ q = t\ n \\
\mathcal{D}\ \mathsf{by}\ \mathsf{Eval\text{-}thunk}()\ \mathsf{reads}\ \emptyset\ \mathsf{writes}\ q & \text{where} & e = \mathtt{thunk}(\mathtt{name}\ n, e_0)\ \text{and}\ q = t\ n \\
\mathcal{D}\ \mathsf{by}\ \mathsf{Eval\text{-}get}()\ \mathsf{reads}\ q\ \mathsf{writes}\ \emptyset & \text{where} & e = \mathtt{get}(\mathtt{ref}\ q)\ \text{and}\ G(q) = v \\
\mathcal{D}\ \mathsf{by}\ \mathsf{Eval\text{-}force}()\ \mathsf{reads}\ q, \mathsf{R}'\ \mathsf{writes}\ W' & \text{where} & e = \mathtt{force}(\mathtt{thunk}\ q) \\
& \text{and} & \mathcal{D}'\ \mathsf{reads}\ \mathsf{R}'\ \mathsf{writes}\ W' \\
& \text{where} & \mathcal{D}'\ \text{is the derivation that computed}\ t
\end{array}
$$

Fig. 16. Read- and write-sets of a non-incremental evaluation derivation.

Definition 1 (Pure effects). *The pure effect of an evaluation derived by $\mathcal{D}$, written $\mathcal{D}$ reads R writes W, is defined in Figure 16.*

This is a (partial) function over derivations. We call these effects "pure" since sibling sub-derivations must have disjoint write sets.

We write "$\mathcal{D}$ by *Rulename (Dlist)* reads R writes W" to mean that rule *Rulename* concludes $\mathcal{D}$ and has subderivations *Dlist*.

For example, $\mathcal{D}$ by Eval-scope($\mathcal{D}_0$) reads R writes W provided that $\mathcal{D}$ reads R writes W, where $\mathcal{D}_0$ derives the only premise of Eval-scope.

6.2 Lemmas

Property 1 (Weakening of entailment). *If* $\Gamma_1, \Gamma_3 \vdash P$ *then* $\Gamma_1, \Gamma_2, \Gamma_3 \vdash P$.

Lemma 1 (Index-level weakening).

1. *If* $\Gamma \vdash M : \gamma$ *then* $\Gamma, \Gamma' \vdash M : \gamma$.
2. *If* $\Gamma \vdash i : \gamma$ *then* $\Gamma, \Gamma' \vdash i : \gamma$.
3. *If* $\Gamma \vdash A : K$ *then* $\Gamma, \Gamma' \vdash A : K$.

Proof. By induction on the given derivation. □

Lemma 2 (Weakening).

1. *If* $\Gamma \vdash e : A$ *then* $\Gamma, \Gamma' \vdash e : A$.

2. *If* $\Gamma \vdash^{M} e : C$ *then* $\Gamma, \Gamma' \vdash^{M} e : C$.

Proof. By induction on the given derivation, using Property 1 (for example, in the case for the value typing rule 'name') and Lemma 1 (Index-level weakening) (for example, in the case for the computation typing rule 'AllIndexElim'). □

Lemma 3 (Substitution)**.**

1. *If* $\Gamma \vdash v : A$ *and* $\Gamma, x : A \vdash e : C$ *then* $\Gamma \vdash \big([v/x]e\big) : C$.
2. *If* $\Gamma \vdash v : A$ *and* $\Gamma, x : A \vdash v' : B$ *then* $\Gamma \vdash \big([v/x]v'\big) : B$.

Proof. By mutual induction on the derivation typing e (in part 1) or v' (in part 2). □

Lemma 4 (Canonical Forms)**.** *If* $\Gamma \vdash v : A$, *then*

1. *If* $A = 1$	*then* $v = ()$.	
2. *If* $A = (B_1 \times B_2)$	*then* $v = (v_1, v_2)$.	
3. *If* $A = (B_1 + B_2)$	*then* $v = \mathtt{inj}_i\, v$	*where* $i \in \{1, 2\}$.
4. *If* $A = (\mathsf{Nm}\,[X])$	*then* $v = \mathtt{name}\; n$	*where* $\Gamma \vdash n \in X$.
5. *If* $A = (\mathsf{Ref}\,[X]\; A_0)$	*then* $v = \mathtt{ref}\; n$	*where* $\Gamma \vdash n \in X$.
6. *If* $A = (\mathsf{Thk}\,[X]\; E)$	*then* $v = \mathtt{thunk}\; n$	*where* $\Gamma \vdash n \in X$.
7. *If* $A = (\mathsf{Nm} \to_{\mathsf{Nm}} \mathsf{Nm})\,[M]$	*then* $v = \mathtt{nmfn}\; M_v$	*where* $M \equiv (\lambda a.\, M')$ *and* $\cdot \vdash (\lambda a.\, M') : (\mathsf{Nm} \to_{\mathsf{Nm}} \mathsf{Nm})$ *and* $M_v \equiv M$.

Proof. In each part, exactly one value typing rule is applicable, so the result follows by inversion. □

Lemma 5 (Application and membership commute)**.** *If* $\Gamma \vdash n \in i$ *and* $p \equiv M(n)$ *then* $\Gamma \vdash p \in M(i)$.

Proof. The set $M(i)$ consists of all elements of i, but mapped by function M. The name p is convertible to the name $M(n)$. Since $n \in i$, we have that p is in the M-mapping of i, which is $M(i)$. □

6.3 Main theorem

In the statement of the theorem, "::" is read "derives", so that $\mathcal{S}$ and $\mathcal{D}$ are the given typing and evaluation derivations. We write $\langle W', R' \rangle \preceq \langle W; R \rangle$ to mean that $W' \subseteq W$ and $R' \subseteq R$.

Theorem 1 (Subject Reduction).
If $\Gamma_1 \vdash M : \mathsf{Nm} \to_{\mathsf{Nm}} \mathsf{Nm}$
and $\mathcal{S} :: \Gamma_1 \vdash^{M} e : C \rhd \langle W; R \rangle$
and $\vdash G_1 : \Gamma_1$
and $\mathcal{D} :: G_1 \vdash^{M}_{m} e \Downarrow G_2; t$
then there exists $\Gamma_2 \supseteq \Gamma_1$ *such that* $\vdash G_2 : \Gamma_2$ *and* $\Gamma_2 \vdash t : C \rhd \langle \emptyset; \emptyset \rangle$
and $\mathcal{D}$ reads $R_{\mathcal{D}}$ writes $W_{\mathcal{D}}$ *and* $\langle W_{\mathcal{D}}; R_{\mathcal{D}} \rangle \preceq \langle W; R \rangle$.

For proofs, see Appendix A.

7 Related Work

DML (Xi and Pfenning 1999; Xi 2007) is an influential system of limited dependent types or *indexed* types. Inspired by Freeman and Pfenning (1991), who created a system in which datasort refinements were clearly separated from ordinary types, DML separates the "weak" index level of typing from ordinary typing; the dynamic semantics ignores the index level.

Motivated in part by the perceived burden of type annotations in DML, liquid types (Rondon et al. 2008; Vazou et al. 2013) deploy machinery to infer more types. These systems also provide more flexibility: types are not indexed by fixed tuples of indices.

To our knowledge, Gifford and Lucassen (1986) were the first to express effects within (or alongside) types. Since then, a variety of systems with this power have been developed. A full accounting of this area is beyond the scope of this paper; for an overview, see Henglein et al. (2005). We briefly discuss a type system for regions (Tofte and Talpin 1997), in which allocation is central. Regions organize subsets of data, so that they can be deallocated together. The type system tracks each block's region, which in turn requires effects on types: for example, a function whose effect is to return a block within a given region. Our type system shares region typing's emphasis on allocation, but we differ in how we treat the names of allocated objects. First, names in our system are fine-grained, in contrast to giving all the objects in a region the same designation. Second, names have structure (for example, the names `root.1.1` and `root.1.2` share a prefix), which allows programmers to deterministically assign names.

Techniques for general-purpose incremental computation. Incremental algorithms, variously called *online algorithms* and *dynamic algorithms*, are expressly designed to be run repeatedly on changing inputs. Through careful language design, modern incremental computation (IC) abstractions elevate implementation questions, such as "how does this particular change pattern affect a particular incremental state of the system?" into more general questions, answered through special programming abstractions. These IC abstractions identify changing data and reusable sub-computations (Acar et al. 2008; Hammer et al. 2009, 2014, 2015; Mitschke et al. 2014), and they allow the programmer to relate the expression of an incremental algorithm to the ordinary version of the algorithm that operates over fixed, unchanging input: The gap between these two programs is witnessed by the special abstractions offered by the incremental language. Through careful algorithm and run-time system design, these IC abstractions admit a fast change-propagation implementation. In particular, after an initial run of the program, as the input changes dynamically, change propagation provides a general (provably sound) approach for recomputing the affected output (Acar et al. 2006; Acar and Ley-Wild 2009; Hammer et al. 2015). Further, incremental computation can deliver *asymptotic* speedups (Acar et al. 2007, 2008; Sümer et al. 2011; Chen et al. 2012). These IC abstractions exist in many languages (Shankar and Bodik 2007; Hammer and Acar 2008; Hammer et al. 2009; Chen et al. 2014).

Functional reactive programming. Incremental computation and reactive programming (especially functional reactive programming or FRP) share common elements: both attempt to respond to outside changes and their implementations often both employ dependence graphs to model dependencies in a program that change over time (Cooper and Krishnamurthi 2006; Krishnaswami and Benton 2011; Krishnaswami 2013; Czaplicki and Chong 2013). In a sketch of future work below, we hope to marry the feedback that is unique to FRP with the incremental data structures and algorithms that are unique to IC.

8 Conclusion and Future Work

We motivate the need for generic naming strategies in programs that use nominal memoization. To fill this need, we define a refinement type system that gives practical static approximations of these naming strategies. We prove that our type system enforces that well-typed programs that use nominal memoization always correspond with a purely functional program. Meanwhile, prior work shows that these programs can dramatically outperform non-incremental programs as well as those using traditional memoization.

Future work: Meta-level programs. The entire point of incremental computation (IC) is to *update* input with *changes*, and then propagate these changes (efficiently) into a *changed* output. Hence, imperative updates are fundamental to IC. Consequently, future work should follow the direction of Hammer et al. (2014) and give explicit type-based annotations that permit such imperative behavior in explicit locations. We discuss feedback in further detail, below.

Future work: Functional reactive programming with explicit names. The effect patterns of feedback and churn, described as problems in Sec. 2, may also be viewed as desirable patterns, especially in contexts such as functional reactive programming (FRP). By definition, feedback occurs when a program overwrites prior allocations with new data, and thus these overwrite effects violate a purely functional allocation strategy. However, we can view this allocation strategy as corresponding to an operational view of FRP with an explicit store where controlled (annotated) feedback may occur safely.

The type system presented here suggests that we can go further than modeling FRP in isolation, and (potentially) marry the controlled feedback of FRP with nominal memoization, and thus, with general-purpose incremental computation. In particular, future work may explore explicit programming annotations for marking intended places and names where feedback occurs:

$$\frac{\Gamma \vdash e : \mathsf{LoopComp}\ A \rhd \langle R \perp F; W \cup F; F\rangle}{\Gamma \vdash \mathsf{loop}\ e\ \mathsf{over}\ F : \mathsf{F}\,A \rhd \langle R \perp F; W \cup F; \emptyset\rangle}\ \text{feedback-loop}$$

The statics of this rule says that sub-expression e has type $\mathsf{LoopComp}\ A \rhd \epsilon$, which is a computation that produces the following (recursive) sum type:

$$\big(\mathsf{LoopComp}\ A \rhd \epsilon\big) = \big(\mathsf{F}\,(A + \mathsf{Thk}\,[X]\ (\mathsf{LoopComp}_X\ A) \rhd \epsilon) \rhd \epsilon\big)$$

Statically, the expression e will either produce a value of type A, or a thunk that produces more thunks (and perhaps possibly a value) in the future.

The dynamics of this rule would re-run expression e until it produces its value (if ever) and in so doing, the write effects of expression e would be free to *overwrite* the reads of e, creating a feedback loop. In the rule above, the annotation $\cdots$ over F makes the over-written set of names F explicit in the program. To statically distinguish delayed, feedback writes from ordinary (immediate) writes, we may track three (not two) name sets in each effect ϵ: the read set, the write set, and the set of feedback writes, here F. Operationally, the dynamic semantics treats feedback writes specially, by delaying them until the $\mathsf{LoopComp}$ fully completes an iteration. This proposal corresponds closely with prior work on synchronous, discrete-time FRP (Krishnaswami and Benton 2011).

Future work: Bidirectional type system. Drawing closer to an implementation, we intend to derive a bidirectional version of the type system, and prove that it corresponds to our declarative type system. A key challenge in implementing the bidirectional system would be to handle constraints over names, name terms and name sets.

Bibliography

Umut A. Acar and Ruy Ley-Wild. Self-adjusting computation with Delta ML. In *Advanced Functional Programming*. Springer, 2009.

Umut A. Acar, Guy E. Blelloch, Matthias Blume, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2006.

Umut A. Acar, Alexander Ihler, Ramgopal Mettu, and Özgür Sümer. Adaptive Bayesian inference. In *Neural Information Processing Systems (NIPS)*, 2007.

Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative self-adjusting computation. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages*, 2008.

Yan Chen, Joshua Dunfield, and Umut A. Acar. Type-directed automatic incrementalization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

Yan Chen, Joshua Dunfield, Matthew A. Hammer, and Umut A. Acar. Implicit self-adjusting computation for purely functional programs. *J. Functional Programming*, 24(1):56–112, 2014.

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

Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. In *PLDI*, 2013.

Joshua Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, 2007. CMU-CS-07-129.

Jean-Christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In *Proceedings of the 2006 Workshop on ML*, pages 12–19. ACM, 2006.

Tim Freeman and Frank Pfenning. Refinement types for ML. In *Programming Language Design and Implementation*, pages 268–277, 1991.

David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on LISP and Functional Programming*, pages 28–38. ACM Press, 1986.

Matthew A. Hammer and Umut A. Acar. Memory management for self-adjusting computation. In *International Symposium on Memory Management*, pages 51–60, 2008.

Matthew A. Hammer, Umut A. Acar, and Yan Chen. CEAL: a C-based language for self-adjusting computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

Matthew A. Hammer, Yit Phang Khoo, Michael Hicks, and Jeffrey S. Foster. Adapton: Composable, demand-driven incremental computation. In *PLDI*, 2014.

Matthew A. Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael Hicks, and David Van Horn. Incremental computation with names. In *OOPSLA*, 2015.

Fritz Henglein, Henning Makholm, and Henning Niss. Effect types and region-based memory management. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 3, pages 87–135. MIT Press, 2005.

Neelakantan R. Krishnaswami. Higher-order functional reactive programming without spacetime leaks. In *ICFP*, 2013.

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

Paul Blain Levy. Call-by-push-value: A subsuming paradigm. In *Typed Lambda Calculi and Applications*, pages 228–243. Springer, 1999.

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

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

Ralf Mitschke, Sebastian Erdweg, Mirko Köhler, Mira Mezini, and Guido Salvaneschi. i3QL: Language-integrated live data views. *OOPSLA*, 2014.

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

Patrick Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *Programming Language Design and Implementation*, pages 159–169, 2008.

Ajeet Shankar and Rastislav Bodik. DITTO: Automatic incrementalization of data structure invariant checks (in Java). In *Programming Language Design and Implementation*, 2007.

Özgür Sümer, Umut A. Acar, Alexander Ihler, and Ramgopal Mettu. Adaptive exact inference in graphical models. *Journal of Machine Learning*, 8:180–186, 2011. To appear.

Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. Abstract refinement types. In *European Symp. on Programming*, pages 209–228, 2013.

Hongwei Xi. Dependent ML: An approach to practical programming with dependent types. *J. Functional Programming*, 17(2):215–286, 2007.

Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Principles of Programming Languages*, pages 214–227, 1999.

A Omitted Proofs

In each case of the proof, we write "☞" to the left of each goal, as we prove it.

Theorem 1 (Subject Reduction).
If $\Gamma_1 \vdash M : \mathsf{Nm} \rightarrow_{\mathsf{Nm}} \mathsf{Nm}$
and $\mathcal{S} :: \Gamma_1 \vdash^{M} e : C \rhd \langle W; R\rangle$
and $\vdash G_1 : \Gamma_1$
and $\mathcal{D} :: G_1 \vdash^{M}_{m} e \Downarrow G_2; t$
then there exists $\Gamma_2 \supseteq \Gamma_1$ *such that* $\vdash G_2 : \Gamma_2$ *and* $\Gamma_2 \vdash t : C \rhd \langle \emptyset; \emptyset\rangle$
and $\mathcal{D}$ reads $R_{\mathcal{D}}$ writes $W_{\mathcal{D}}$ *and* $\langle W_{\mathcal{D}}; R_{\mathcal{D}}\rangle \preceq \langle W; R\rangle$.

Proof. By induction on the typing derivation $\mathcal{S}$.

– **Case** $$\dfrac{\Gamma_1 \vdash v : A}{\Gamma_1 \vdash^{M} \mathtt{ret}(v) : \big((\mathsf{F}\,A) \rhd \langle \emptyset; \emptyset\rangle\big)}\ \text{ret}$$

	$(e = t)$ and $(G_1 = G_2)$	Given
	$(R_{\mathcal{D}} = W_{\mathcal{D}} = R = W = \emptyset)$	″
	$(\Gamma_2 = \Gamma_1)$	Suppose
☞	$\vdash G_2 : \Gamma_2$	by above equalities
☞	$\Gamma_2 \vdash t : C \rhd \langle \emptyset, \emptyset\rangle$	″
☞	$\mathcal{D}$ reads $R_{\mathcal{D}}$ writes $W_{\mathcal{D}}$	By Def. 1
☞	$\langle W_{\mathcal{D}}; R_{\mathcal{D}}\rangle \preceq \langle W; R\rangle$	All are empty

– **Case** $$\dfrac{\Gamma_1 \vdash v : \mathsf{Ref}\,[X]\ A}{\Gamma_1 \vdash^{M} \mathtt{get}(v) : \big(\mathsf{F}\,A\big) \rhd \langle \emptyset; X\rangle}\ \text{get}$$

$(W = \emptyset)$ and $(R = X)$	Given
$\Gamma_1 \vdash v : \mathsf{Ref}\,[X]\ A$	Given
$\exists p.\ (v = \mathtt{ref}\,p)$	Lemma 4 (Canonical Forms)
$\Gamma_1 \vdash p \in X$	″
$\Gamma_1(p) = A$	By inversion of value typing
$\exists v_p.\ G_1(p) = v_p$	Inversion on $\vdash G_1 : \Gamma_1$
$\Gamma_1 \vdash v_p : A$	″

	$(\Gamma_2 = \Gamma_1)$ and $(t = \mathtt{ret}(v_p))$	Suppose
	$(R_{\mathcal{D}} = \{p\})$ and $(W_{\mathcal{D}} = \emptyset = W)$	″
☞	$\vdash G_2 : \Gamma_2$	By above equalities
☞	$\Gamma_2 \vdash t : C \rhd \langle \emptyset, \emptyset\rangle$	″

☞	$\mathcal{D}$ reads $R_{\mathcal{D}}$ writes $W_{\mathcal{D}}$	By Def. 1
☞	$\langle W_{\mathcal{D}}; R_{\mathcal{D}}\rangle \preceq \langle W; R\rangle$	By above equality $W_{\mathcal{D}} = W = \emptyset$,
		… and inequality for $(R_{\mathcal{D}} = \{p\}) \subseteq (X = R)$.

– **Case** $$\dfrac{\Gamma_1 \vdash v : \mathsf{Thk}\,[X]\ (C \rhd \epsilon)}{\Gamma_1 \vdash^{M} \mathtt{force}(v) : \big(C \rhd (\langle \emptyset; X\rangle \ \mathsf{then}\ \epsilon)\big)}\ \text{force}$$

	$(W = \emptyset)$ and $(R = X)$	Given
	$\Gamma_1 \vdash v : \mathsf{Thk}\,[X]\ (C \rhd \epsilon)$	Given
	$\exists p.\ (v = \mathtt{thunk}\ p)$	Lemma 4 (Canonical Forms)
	$\Gamma_1 \vdash p \in X$	$''$
	$\Gamma_1(p) = (C \rhd \epsilon)$	By inversion of value typing
	$\exists e_p.\ G_1(p) = e_p$	Inversion on $\vdash G_1 : \Gamma_1$
$\mathcal{S}_0 ::$	$\Gamma_1 \vdash e_p : (C \rhd \epsilon)$	$''$

$\mathcal{D}_0 ::$	$G_1 \vdash^{M}_{m} e_p \Downarrow G_2; t$	Inversion of $\mathcal{D}$
☞	$\vdash G_2 : \Gamma_2$	By i.h. on $\mathcal{S}_0$ and $\mathcal{D}_0$
☞	$\Gamma_2 \vdash t : C \rhd \langle \emptyset, \emptyset \rangle$	$''$
	$\mathcal{D}_0$ reads $R_{\mathcal{D}}$ writes $W_{\mathcal{D}}$	$''$
	$\langle W_{\mathcal{D}_0}; R_{\mathcal{D}_0} \rangle \preceq \langle W; R \rangle$	$''$
☞	$\mathcal{D}$ reads $R_{\mathcal{D}_0}$ writes $W_{\mathcal{D}_0}$	By Def. 1
☞	$\langle W_{\mathcal{D}_0}; R_{\mathcal{D}_0} \rangle \preceq \langle W; R \rangle$	by above equality $W_{\mathcal{D}} = W = \emptyset$,

... and inequality $(R_{\mathcal{D}} = \{p\}) \subseteq (X = R)$.

– **Case**
$$\frac{\Gamma_1 \vdash M' : \mathsf{Nm} \rightarrow_{\mathsf{Nm}} \mathsf{Nm} \qquad \Gamma_1 \vdash^{M \circ M'} e_0 : C \rhd \langle W; R \rangle}{\Gamma_1 \vdash^{M} \mathsf{scope}(M', e_0) : C \rhd \langle W; R \rangle}\ \text{scope}$$

$\mathcal{S}_0 ::$	$\Gamma_1 \vdash^{M \circ M'} e_0 : C \rhd \langle W; R \rangle$	Subderivation 2 of $\mathcal{S}$
$\mathcal{D} ::$	$G_1 \vdash^{M}_{m} \mathsf{scope}(M', e_0) \Downarrow G_2; t$	Given
$\mathcal{D}_0 ::$	$G_1 \vdash^{M \circ M'}_{m} e_0 \Downarrow G_2; t$	By inversion (scope)

$\Gamma_1 \vdash M : \mathsf{Nm} \rightarrow_{\mathsf{Nm}} \mathsf{Nm}$	Assumption
$\Gamma_1 \vdash M' : \mathsf{Nm} \rightarrow_{\mathsf{Nm}} \mathsf{Nm}$	Subderivation 1 of $\mathcal{S}$
$\Gamma_1, x : \mathsf{Nm} \vdash M'\,x : \mathsf{Nm}$	By rule $\mathsf{t\text{-}app}$
$\Gamma_1, x : \mathsf{Nm} \vdash M\,(M'\,x) : \mathsf{Nm}$	By rule $\mathsf{t\text{-}app}$
$\Gamma_1 \vdash \lambda x.\, M\,(M'\,x) : \mathsf{Nm} \rightarrow_{\mathsf{Nm}} \mathsf{Nm}$	By rule $\mathsf{t\text{-}abs}$
$\Gamma_1 \vdash (M \circ M') : \mathsf{Nm} \rightarrow_{\mathsf{Nm}} \mathsf{Nm}$	By definition of $M \circ M'$

☞	$\vdash G_2 : \Gamma_2$	By i.h. on $\mathcal{S}_0$
☞	$\Gamma_2 \vdash t : C \rhd \langle \emptyset; \emptyset \rangle$	$''$
	$\mathcal{D}_0$ reads $R_{\mathcal{D}_0}$ writes $W_{\mathcal{D}_0}$	$''$
	$\langle W_{\mathcal{D}_0}; R_{\mathcal{D}_0} \rangle \preceq \langle W; R \rangle$	$''$
☞	$\mathcal{D}$ reads $R_{\mathcal{D}}$ writes $W_{\mathcal{D}}$	By Def. 1
	$\langle W_{\mathcal{D}}; R_{\mathcal{D}} \rangle = \langle W_{\mathcal{D}_0}; R_{\mathcal{D}_0} \rangle$	$''$
☞	$\langle W_{\mathcal{D}}; R_{\mathcal{D}} \rangle \preceq \langle W; R \rangle$	By above equalities

– **Case**
$$\frac{\Gamma_1 \vdash v : \mathsf{Nm}\,[X] \qquad \Gamma_1 \vdash e : E}{\Gamma_1 \vdash^{M} \mathtt{thunk}(v_1, v_2) : \mathsf{F}\,(\mathsf{Thk}\,[M(X)]\ E) \rhd \langle M(X); \emptyset \rangle}\ \text{thunk}$$

$C = \mathsf{F}\,(\mathsf{Thk}\,[M(X)]\ E)$ and $R = \emptyset$ and $W = M(X)$	Given from $\mathcal{S}$
$\Gamma_1 \vdash v : \mathsf{Nm}\,[X]$	Subderivation
$(v = \mathtt{name}\ n)$ and $(n \in X)$	By Lemma 4
$M\ n \Downarrow p$ and $R_{\mathcal{D}} = \emptyset$ and $W_{\mathcal{D}} = \{p\}$	Given from $\mathcal{D}$
$G_2 = (G_1, p : e)$	$''$

	$\Gamma_2 = (\Gamma_1, p : \mathsf{Thk}\,[p]\ E)$	Suppose
☞	$\vdash G_2 : \Gamma_2$	By rule (Fig. 15)
	$\Gamma_2(p) = E$	By inversion of value typing

	$\Gamma_2 \vdash \mathtt{ref}\,p : \mathsf{Ref}\,[p]\ E$	By rule thunk
☞	$\Gamma_2 \vdash^M \mathtt{ret(thunk}\ p\mathtt{)} : \mathsf{F}\,(\mathsf{Thk}\,[p]\ A) \rhd \langle\emptyset;\emptyset\rangle$	By rule ret
☞	$\mathcal{D}$ reads $R_{\mathcal{D}}$ writes $W_{\mathcal{D}}$ and $W_{\mathcal{D}} = \{p\}$	By Def. 1

	$n \in X$	Above
	$M(n) \in M(X)$	Name term application is pointwise
	$M(n) \in W$	By above equality
	$M(n) = p$	
	$\{p\} \subseteq W$	By set theory
☞	$\langle W_{\mathcal{D}}; R_{\mathcal{D}}\rangle \preceq \langle W; R\rangle$	

– **Case** $$\frac{\Gamma_1 \vdash v_1 : \mathsf{Nm}\,[X] \qquad \Gamma_1 \vdash v_2 : A}{\Gamma_1 \vdash^M \mathtt{ref(}v_1, v_2\mathtt{)} : \mathsf{F}\,(\mathsf{Ref}\,[M(X)]\ A) \rhd \langle M(X);\emptyset\rangle}\ \text{ref}$$

$C = \mathsf{F}\,(\mathsf{Ref}\,[M(X)]\ A)$ and $R = \emptyset$ and $W = M(X)$ Given from $\mathcal{S}$

$\Gamma_1 \vdash v_1 : \mathsf{Nm}\,[X]$	Subderivation
$(v_1 = \mathtt{name}\ n)$ and $(n \in X)$	Lemma 4 (Canonical Forms)
$M\ n \Downarrow p$ and $R_{\mathcal{D}} = \emptyset$ and $W_{\mathcal{D}} = \{p\}$	Given from $\mathcal{D}$
$G_2 = (G_1, p : v_2)$	$''$

	$\Gamma_2 = (\Gamma_1, p : \mathsf{Ref}\,[p]\ A)$	Suppose
☞	$\vdash G_2 : \Gamma_2$	By rule (Fig. 15)
	$\Gamma_2(p) = A$	By inversion of value typing
	$\Gamma_2 \vdash \mathtt{ref}\,p : \mathsf{Ref}\,[p]\ A$	By rule ref
☞	$\Gamma_2 \vdash^M \mathtt{ret(ref}\ p\mathtt{)} : \mathtt{ret(}\mathsf{Ref}\,[p]\ A\mathtt{)} \rhd \langle\emptyset;\emptyset\rangle$	By rule ret
☞	$\mathcal{D}$ reads $R_{\mathcal{D}}$ writes $W_{\mathcal{D}}$ and $W_{\mathcal{D}} = \{p\}$	By Def. 1

	$n \in X$	Above
	$M(n) \in M(X)$	Name term application is pointwise
	$M(n) \in W$	By above equality
	$M(n) = p$	
	$\{p\} \subseteq W$	By set theory
☞	$\langle W_{\mathcal{D}}; R_{\mathcal{D}}\rangle \preceq \langle W; R\rangle$	

– **Case** $$\frac{\Gamma_1 \vdash^M e_1 : (\mathsf{F}\,A \rhd \epsilon_1) \qquad \Gamma_1, x : A \vdash^M e_2 : (C \rhd \epsilon_2)}{\Gamma_1 \vdash^M \mathtt{let(}e_1, x.e_2\mathtt{)} : C \rhd (\epsilon_1\ \mathsf{then}\ \epsilon_2)}\ \text{let}$$

	$\vdash G_1 : \Gamma_1$	Given
$\mathcal{S}_1$ ::	$\Gamma_1 \vdash^M e_1 : \mathsf{F}\,A \rhd \epsilon_1$	Subderivation 1 of $\mathcal{S}$
$\mathcal{D}_1$ ::	$G_1 \vdash^M_m e_1 \Downarrow G_{12}; t_1$	Subderivation 1 of $\mathcal{D}$
	exists $\Gamma_{12} \supseteq \Gamma_1$ such that $G_{12} : \Gamma_{12}$	By i.h. on $\mathcal{S}_1$
	$\Gamma_{12} \vdash t_1 : \mathsf{F}\,A \rhd \langle\emptyset;\emptyset\rangle$	$''$
	$\mathcal{D}_1$ reads $R_{\mathcal{D}_1}$ writes $W_{\mathcal{D}_1}$	$''$
	$\langle W_{\mathcal{D}_1}; R_{\mathcal{D}_1}\rangle \preceq \epsilon_1$	$''$
	$\langle W_{\mathcal{D}_1}; R_{\mathcal{D}_1}\rangle \preceq \langle W_1, R_1\rangle$	$''$

	$\Gamma_{12} \vdash v : A$	inversion of typing rule ret, for terminal computation t_1
$\mathcal{S}_2$ ::	$\Gamma_1, x : A \vdash^{M} e_2 : C \rhd \epsilon_2$	Subderivation 2 of $\mathcal{S}$
	$\Gamma_{12}, x : A \vdash^{M} e_2 : C \rhd \epsilon_2$	Lemma 2 (Weakening)
	$\Gamma_{12} \vdash^{M} [v/x]e_2 : C \rhd \epsilon_2$	Lemma 3 (Substitution)
$\mathcal{D}_2$ ::	$G_{12} \vdash^{M}_{m} [v/x]e_2 \Downarrow G_2; t_2$	Subderivation 2 of $\mathcal{D}$
	exists $\Gamma_2 \supseteq \Gamma_{12} \supseteq \Gamma_1$ such that	By i.h. on $\mathcal{S}_2$
☞	$\vdash G_2 : \Gamma_2$	$''$
☞	$\Gamma_2 \vdash^{M} t_2 : C \rhd \langle\emptyset;\emptyset\rangle$	$''$
	$\mathcal{D}_2$ reads $R_{\mathcal{D}_2}$ writes $W_{\mathcal{D}_2}$	$''$
	$\langle W_{\mathcal{D}_2}; R_{\mathcal{D}_2}\rangle \preceq \epsilon_2$	$''$
	$\langle W_{\mathcal{D}_2}; R_{\mathcal{D}_2}\rangle \preceq \langle W_2, R_2\rangle$	$''$

	$W_1 \perp W_2$ and $R_1 \perp W_2$	Definition of ϵ_1 then ϵ_2
	$W_{\mathcal{D}_1} \perp W_{\mathcal{D}_2}$ and $R_{\mathcal{D}_1} \perp W_{\mathcal{D}_2}$	$W_{\mathcal{D}_1} \subseteq W_1$; $W_{\mathcal{D}_2} \subseteq W_2$; $R_{\mathcal{D}_1} \subseteq R_1$
	$W_{\mathcal{D}} = W_{\mathcal{D}_1} \perp W_{\mathcal{D}_2}$	By Def. 1
	$R_{\mathcal{D}} = R_{\mathcal{D}_1} \cup (R_{\mathcal{D}_2} - W_{\mathcal{D}_1})$	$''$
☞	$\mathcal{D}$ reads $R_{\mathcal{D}}$ writes $W_{\mathcal{D}}$	$''$
☞	$\langle W_{\mathcal{D}}, R_{\mathcal{D}}\rangle \preceq \langle W, R\rangle$	Since $W_{\mathcal{D}} \subseteq W$ and $R_{\mathcal{D}} \subseteq R$

– **Case**

$$\frac{\Gamma \vdash^{M} e : \big((A \to E) \rhd \epsilon_1\big) \qquad \Gamma \vdash v : A}{\Gamma \vdash^{M} (e\ v) : (E\ \mathsf{after}\ \epsilon_1)}\ \text{app}$$

Similar to the case for let.

– **Case**

$$\frac{\Gamma \vdash^{M} v : (A_1 \times A_2) \qquad \Gamma, x_1 : A_1, x_2 : A_2 \vdash^{M} e : E}{\Gamma \vdash^{M} \mathsf{split}(v, x_1.x_2.e) : E}\ \text{split}$$

Similar to the case for let, using Lemma 4 (Canonical Forms).

– **Case**

$$\frac{\Gamma \vdash^{M} v : (A_1 + A_2) \qquad \begin{array}{c}\Gamma, x_1 : A_1 \vdash^{M} e_1 : E \\ \Gamma, x_2 : A_2 \vdash^{M} e_2 : E\end{array}}{\Gamma \vdash^{M} \mathsf{case}(v, x_1.e_1, x_2.e_2) : E}\ \text{case}$$

Similar to the case for let, using Lemma 4 (Canonical Forms).

– **Case**

$$\frac{\begin{array}{l}\Gamma_1 \vdash v_M : (\mathsf{Nm} \to_{\mathsf{Nm}} \mathsf{Nm})\,[M] \\ \Gamma_1 \vdash v : \mathsf{Nm}\,[i]\end{array}}{\Gamma_1 \vdash (v_M\ v) : \mathsf{F}\,(\mathsf{Nm}\,[M(i)])\rhd \langle\emptyset;\emptyset\rangle}\ \text{name-app}$$

$\Gamma_1 \vdash v_M : (\mathsf{Nm} \to_{\mathsf{Nm}} \mathsf{Nm})\,[M]$	Given
$v_M = \mathbf{nmfn}\ M_v$	Lemma 4 (Canonical Forms)
$M \equiv (\lambda a.\, M')$	$''$
$\cdot \vdash \lambda a.\, M' : (\mathsf{Nm} \to_{\mathsf{Nm}} \mathsf{Nm})$	$''$
$M_v \equiv M$	$''$

	$\Gamma_1 \vdash v : \mathsf{Nm}[i]$	Given
	$v = \mathtt{name}\ n$	Lemma 4 (Canonical Forms)
	$\Gamma \vdash n \in i$	$''$
	$M \Downarrow_{\mathsf{M}} (\lambda a.\, M')$	By inversion on $\mathcal{D}$ (name-app)
	$[n/a]M' \Downarrow_{\mathsf{M}} p$	$''$
	$p \equiv [n/a]M'$	By a property of $\Downarrow_{\mathsf{M}}$
	$\equiv (\lambda a.\, M')(n)$	By a property of $\equiv$
	$\equiv M(n)$	By a property of $\equiv$
	$(\Gamma_2 = \Gamma_1), (G_2 = G_1)$	Suppose
☞	$\vdash G_2 : \Gamma_2$	By above equalities and $G_1 \vdash \Gamma_1$
	$\Gamma_1 \vdash n \in i$	Above
	$p \equiv M(n)$	Above
	$\Gamma_1 \vdash p \in M(i)$	By Lemma 5
	$\Gamma_1 \vdash \mathtt{name}\ p : \mathsf{Nm}[M(i)]$	By rule name
☞	$\Gamma_1 \vdash \mathtt{ret(name\ p)} : \mathsf{F}\,(\mathsf{Nm}[M(i)]) \rhd \langle\emptyset;\emptyset\rangle$	By rule ret
☞	$\mathcal{D}$ by Eval-name-app reads $\emptyset$ writes $\emptyset$	By Def. 1
☞	$(R_{\mathcal{D}} = R = \emptyset), (W_{\mathcal{D}} = W = \emptyset)$	By above equalities

– **Case** $$\dfrac{\Gamma_1, a:\gamma \vdash^{M} t : E}{\Gamma_1 \vdash^{M} t : (\forall a:\gamma.\, E)}\ \text{AllIndexIntro}$$

	$\mathcal{S}_0 :: \Gamma_1, a:\gamma \vdash^{M} t : E$	Subderivation
	$\mathcal{D}_0 ::\ G_1 \vdash^{M}_{m} e \Downarrow G_2; t$	Subderivation
	$\exists \Gamma_2 \subseteq \Gamma_1$	By i.h.
☞	$\vdash G_2 : \Gamma_2$	$''$
	$\mathcal{D}_0$ reads $R_{\mathcal{D}_0}$ writes $W_{\mathcal{D}_0}$	$''$
	$\Gamma_2 \vdash t : E$	$''$
	$\langle R_{\mathcal{D}_0}; W_{\mathcal{D}_0}\rangle \preceq \langle R; W\rangle$	$''$
☞	$\Gamma_2 \vdash^{M} t : (\forall a:\gamma.\, E)$	By typing rule
☞	$\mathcal{D}$ reads $R_{\mathcal{D}}$ writes $W_{\mathcal{D}}$	By Def. 1
☞	$\langle R_{\mathcal{D}}; W_{\mathcal{D}}\rangle \preceq \langle R; W\rangle$	By set theory

– **Case** $$\dfrac{\Gamma_1 \vdash^{M} e : (\forall a:\gamma.\, E) \qquad \Gamma_1 \vdash i : \gamma}{\Gamma_1 \vdash^{M} e : [i/a]E}\ \text{AllIndexElim}$$

	$\mathcal{S}_0 ::\ \Gamma_1 \vdash^{M} e : (\forall a:\gamma.\, E)$	Subderivation
	$\mathcal{D}_0 ::\ G_1 \vdash^{M}_{m} e \Downarrow G_2; t$	Subderivation
	$\exists \Gamma_2 \subseteq \Gamma_1$	By i.h.
☞	$\vdash G_2 : \Gamma_2$	$''$
	$\mathcal{D}_0$ reads $R_{\mathcal{D}_0}$ writes $W_{\mathcal{D}_0}$	$''$
	$\Gamma_2 \vdash t : (\forall a:\gamma.\, E)$	$''$
	$\langle R_{\mathcal{D}_0}; W_{\mathcal{D}_0}\rangle \preceq \langle R; W\rangle$	$''$

	$\Gamma_1 \vdash i : \gamma$	Subderivation
	$\Gamma_2 \vdash i : \gamma$	By weakening
☞	$\Gamma_2 \vdash^{M} t : [i/a]$	By typing rule
☞	$\mathcal{D}$ reads $R_{\mathcal{D}}$ writes $W_{\mathcal{D}}$	By Def. 1
☞	$\langle R_{\mathcal{D}}; W_{\mathcal{D}} \rangle \preceq \langle R; W \rangle$	By set theory

– **Case** $\dfrac{\Gamma, a : \gamma \vdash^{M} t : E}{\Gamma \vdash^{M} t : (\forall \alpha : K.\, E)}$ AllIntro

Similar to the AllIndexIntro case.

– **Case** $\dfrac{\Gamma \vdash^{M} e : (\forall \alpha : K.\, E) \qquad \Gamma \vdash A : K}{\Gamma \vdash^{M} e : [A/\alpha]E}$ AllElim

Similar to the AllIndexElim case. □