Typed Adapton:
Refinement Types for Nominal Memoization of Purely Functional Incremental Programs

Matthew A. Hammer
University of Colorado Boulder
matthew.hammer@colorado.edu

Joshua Dunfield
University of British Columbia
joshdunf@cs.ubc.ca

October 1, 2016

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 with a purely functional specification.

This paper develops a refinement type system for nominal memoization that enforces a program's correspondence with a purely functional specification. 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 proposed by Hammer et al. (2015a). Past work shows that these conditions suffice to write useful examples of nominal memoization while also guaranteeing *from-scratch consistency* of the incremental programs. These features contribute to its overall goal of expressing generic *naming strategies* in type-generic incremental code. In particular, we show various forms of *namespace parametricity* and illustrate through these examples its importance for expressing nominal memoization in library code. We also show how extensions to our type system can permit controlled forms of naming strategies that encode incremental *churn* and *feedback*. We speculate that certain feedback and churn patterns constitute naming strategies that encode existing forms of functional reactive computation as a mode of use of nominal memoization.

1 Introduction

Memoization is an evaluation technique that records a mapping from computation identities to computation results (e.g., the identity of a function application may consist of a function name and an argument); the goal of memoization is to *reuse* computations' results and avoid recomputing them redundantly.

Structural memoization is defined by comparing the input and output structures of memoized computations based on their *structural identity*, which is commonly implemented in $O(1)$ time per pointer comparison by using a variant of hash-consing. 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 as a practical method for efficiently testing structural identity.

Nominal memoization uses a notion of *pointer name* identity to identify both computations and data structures (input and output structures). Unlike structural memoization, the users of nominal memoization provide explicit, first-class names to each allocation site, and in turn, these names orchestrate an explicit, deterministic *naming strategy*, with direct implications for incremental computations.

By convention, programs that employ nominal memoizaton compute with inductive input and output structures that contain these first-class names, as well as pointers named by them. The first-class names to determine, for each recursive function, how the individual components of an input structure are related, structurally, to the components of its output structure. In Sec. 2, we give a detailed example of this style, in terms of mapping a list of input elements to a list of output elements. Prior work showed for this example that using names, we can respond to an imperative input change, such as element insertion, with worst-case $\mathsf{O}(1)$ re-executions and fresh allocations, rather than the $\mathsf{O}(n)$ required with structural memoization. That work also showed other examples where names permit an otherwise purely functional specification to use nominal memoization to respond to input changes efficiently.

The practical performance of nominal memoization stems directly from how the programmer designs a naming strategy. To implement it, they instrument an otherwise functional program with names that *explicitly relate input and output structures*. This instrumentation is error-prone, and making a mistake may change the meaning of the program, rendering it into an imperative program whose behavior no longer corresponds to a purely functional program.

This paper presents a refinement type system for nominal memoization of purely functional programs. Specifically, the type system enforces that all programs behave as though they are purely functional, excluding behavior that overwrites prior allocations with later ones. Meanwhile, the type system provides a way to statically specify the meaning of a program that uses nominal memoization, and thus, a verifiable way to safely compose incremental programs from generic library code. In particular, we illustrate how generic naming strategies can be encoded statically as various forms of name parametricity.

Contributions:

- We develop a type system for a variant of Nominal Adapton (Hammer et al. 2015a). 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. 2015a).

  In this work, unlike prior work, we only consider non-incremental runs.
- 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 these 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

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. 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 explicates 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 a sub-list for the rest of the sequence. The `ref` constructor permits injecting (incrementally changing) references holding sub-lists into the list type; these pointers have names drawn from set Y.

$$\begin{array}{rcrrrl}
\mathtt{Nil} & : & \forall X, Y. & & & \mathsf{List}\,[X;Y]\ \mathsf{Int} \\
\mathtt{Cons} & : & \forall X, Y. & \mathsf{Int} \rightarrow & \mathsf{List}\,[X;Y]\ \mathsf{Int} \rightarrow & \mathsf{List}\,[X;Y]\ \mathsf{Int} \\
\mathtt{Name} & : & \forall X_1, X_2, Y. & \mathsf{Nm}\,[X_1] \rightarrow & \mathsf{List}\,[X_2;Y]\ \mathsf{Int} \rightarrow & \mathsf{List}\,[X_1 \perp X_2;Y]\ \mathsf{Int} \\
\mathtt{Ref} & : & \forall X, Y_1, Y_2. & \multicolumn{2}{r}{\mathsf{Ref}\,[Y_1]\ (\mathsf{List}\,[X;Y_2]\ \mathsf{Int}) \rightarrow} & \mathsf{List}\,[X;Y_1 \perp Y_2]\ \mathsf{Int}
\end{array}$$

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.

2.1 listmap1: Naive nominal memoization

First, we consider the simplest naming strategy for listmap, where input and output names coincide; statically, both are drawn from an arbitrary name set X. Meanwhile, the names of the input reference cells are drawn from a name set Y. Notice how the output type list type communicates that the output names and pointer names are drawn from the same set X, the name set of the input list (that is, its first type index).

$$\begin{array}{l} \forall X, Y : \mathbf{NmSet}. \\ \longrightarrow (\mathsf{Int} \longrightarrow \mathsf{Int}) \\ \longrightarrow \mathsf{List}\,[X;Y]\ \mathsf{Int} \\ \longrightarrow \mathsf{List}\,[X;X]\ \mathsf{Int} \end{array}$$

```
listmap1 = lambda f. fix listmap_rec. lambda l.
   match l with
    Nil       => Nil,
    Cons(h,t) => Cons(f h, listmap_rec t)
    Ref(r)    => listmap_rec (get r)
    Name(n,t) => Name(n, Ref(memo_ref n [ listmap_rec t ]))
```

Turning to the program text for listmap1, the Nil and Cons cases below are conventional. 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 to 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.

Unintended imperative effects: Unintended feedback and churn. Conceptually, the program above captures the idea of mapping a list, with several additional nominal steps in the Name case: the program maps input names to output names, output reference cells, and a corresponding memoized recursive call; all are identified with the same name. Conceptually, these are the right steps; but, on closer inspection, however, two problems arise quickly with this naive naming strategy:

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.

2.2 listmap2: Name parametricity for the output list

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

$$\begin{array}{l} \forall X, Y : \mathbf{NmSet}.\ \forall Z : \mathbf{Nm}. \\ \mathsf{Nm}\,[Z] \\ \to (\mathsf{Int} \to \mathsf{Int}) \\ \to \mathsf{List}\,[X; Y]\ \mathsf{Int} \\ \to \mathsf{List}\,[X; (Z,\ X)]\ \mathsf{Int} \end{array}$$

```
listmap2 = lambda m. lambda f. fix listmap_rec. lambda l.
   match l with
    ...
    Name(n,t) => Name(n, Ref(memo_ref (m,n) [ listmap_rec t ]))
```

We adapt the original program by adding a new parameter, first-class name `m`, and in the `Name` case, 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
```

Assuming that the names `m1` $\in Z$ and `m2` $\in Z$ are disjoint (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. 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, except in places that the programmer designates explicitly with certain annotations.

As a result, in all other places the type system enforces that no unintended overwriting occurs. Further, we can prove that a simple incremental change propagation algorithm always produces results that are consistent with a purely-functional execution; without this condition, it is easy to show simple counter-examples, such as those given above.

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 `fnm`. She names the reference cells of the output list type by applying `fnm` to the names of the input list. In terms of program text, this generalization universally quantifies over the name function `fnm`, and it applies this function in the `Name` case.

$$\begin{array}{l}
\forall X, Y : \mathbf{NmSet}. \\
\forall \mathtt{fnm} : \mathbf{Nm} \rightarrow_{\mathbf{Nm}} \mathbf{Nm}. \\
(\mathbf{Nm} \rightarrow_{\mathbf{Nm}} \mathbf{Nm})\,[\mathtt{fnm}] \\
\rightarrow (\mathsf{Int} \rightarrow \mathsf{Int}) \\
\rightarrow \mathsf{List}\,[X; Y]\ \mathsf{Int} \\
\rightarrow \mathsf{List}\,[X; \mathtt{fnm}\ X\,]\ \mathsf{Int}
\end{array}$$

```
listmap3 l = lambda fnm. lambda f. fix listmap_rec. lambda l.
   match l with
    ...
    Name(n,t) => Name(n, Ref (memo_ref (fnm n) [ listmap_rec t ]))
```

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 `fnm` be one that pairs its argument with the distinguished name `m`, viz., λ`a.(m,a)`, as asserted below:

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

2.4 listmap4: Naming strategy for higher-order parametricity

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 $k_A \rightarrow_{\mathbf{Nm}} k_B$ classifies the name function `fab`, which relates names of type A to names of type B.

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

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

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 namespace of the output list 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 fnm 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, simplist 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 ]
```

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 peice of state that is implicitly carried throughout a dynamic scope of computation.

Limitations of disjoint dynamic scopes: No sharing. While permitting economical sizes of code and types, the disadvantages of scope relate to that of monads generally: Sometimes the "plumbing" hidden by a monadic computation cannot express the generic composition pattern that the programmer intends. For instance, differently-scoped sub-computations will never share any allocations or sub-computations since, by definition, their allocated names will be disjoint. Meanwhile, the programmer may want to permit (safe) forms of sharing within their naming strategy.

For instance, suppose two lists share common elements and we wish to map both of them with the same element-mapping function. In this case, we may use listmap4 to share the allocation of mapped elements, but not the allocation of list cells (which could do strange things, and generally would not preserve the structure of the input lists). Meanwhile, using listmap1 and different scopes would fail to share the allocation of mapped elements, making the two sub-computations completely disjoint.

We should note that the type system presented here does not permit any sharing, though we imagine that with special annotations, we could make such effects explicit, and rely on the human programmer to check these uses for safety. As future work, we sketch another annotation-based approach in Sec. 8 for expressing controlled forms of feedback.

3 Program Syntax

Fig. 1 gives the abstract syntax for expressions e and values v. 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 corresponds with 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 explicates the creation of all closures and thunks, it also exposes exactly the structure that we wish to extend as a general-purpose abstraction for incremental computation. In particular, thunks are the unit by which we cache results and track dynamic dependencies. Finally, focusing on thunks helps us model a demand-driven incremental computation, e.g., incremental, demand-driven sorting algorithms.

Expressions consist of elimination forms for pairs and sums (split and case, respectively); intro and elimination forms for producing values (ret and let, respectively); intro and elimination forms for function types (lambda and application, respectively); intro and elimination forms for thunks, and intro and elimination forms for value pointers (reference cells that hold values).

The syntax in these figures closely follows that of prior work on Adapton, including Hammer et al. (2015a). We generalize that work with the notion of a *name function*, which generalizes the idea of a namespace and also operations over names such as "forking". The construct scope 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 permits programs to compute with names and name functions that reside within the type indicies. Since these name functions always terminate, they do not affect a program's termination behavior.

Values	v ::=	x	
		$\mid ()$	
		$\mid (v_1, v_2)$	introduce product
		$\mid \mathtt{inj}_i\ v$	introduce sum
		$\mid \mathtt{name}\ n$	the name n, as a value
		$\mid \mathtt{ref}\ n$	ref cell at location n
		$\mid \mathtt{thunk}\ n$	thunk at location n
Terminal expressions	t ::=	$\mathtt{ret}\,(v)$	return the value v
		$\mid \lambda x.\, e$	ordinary λ
Expressions	e ::=	t	terminal expression (above)
		$\mid \mathtt{split}\,(v, x_1.x_2.e)$	eliminate product
		$\mid \mathtt{case}\,(v, x_1.e_1, x_2.e_2)$	eliminate sum
		$\mid \mathtt{let}\,(e_1, x.e_2)$	
		$\mid e\ v$	ordinary function application
		$\mid \mathtt{ref}(v, v)$	allocate a ref cell
		$\mid \mathtt{thunk}\,(v, e)$	create a thunk
		$\mid \mathtt{force}\,(v)$	force a thunk
		$\mid \mathtt{get}\,(v)$	get the contents of a ref cell
		$\mid \mathtt{scope}\,(M, e)$	enter a namespace
		$\mid M\ v$	name function M applied to name v

Figure 1 Syntax of expressions

The syntax for values consists of the unit value, pairs of values, tagged injected values (one of the two "halves" of a sum type), first-class names, value pointers and thunk pointers.

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 possibilities.

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:

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

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.

Kinds	K	$::=$	$\star$	
			$\mid \star \Rightarrow K$	type argument (binder space)
			$\mid \gamma \Rightarrow K$	index argument (binder space)
Name terms	M	$::=$	n	name
			$\mid ()$	unit
			$\mid \lambda a.\, M$	abstraction
			$\mid a$	variable
			$\mid M(M)$	application
			$\mid M.1$	append 1 to name
			$\mid M.2$	append 2 to name
			$\mid (M_1, M_2)$	tuple
Index sorts	γ	$::=$	$\mathbf{Nm}$	name
			$\mid \mathbf{NmSet}$	name set sort
			$\mid \mathbf{1}$	unit index sort
			$\mid \gamma * \gamma$	product index sort
			$\mid \gamma \rightarrow_{\mathbf{Nm}} \gamma$	index-level function
Symbols	s			
Names	n, m, p, q	$::=$	s	symbol
			$\mid \mathtt{root}$	root name
			$\mid n.1 \mid n.2$	append bit (disjunctive names)
			$\mid (n_1, n_2)$	pairing (conjunctive names)
Index variables	a			
Index expressions	i, X, Y, Z, R, W	$::=$	a	
			$\mid \{n\}$	singleton set
			$\mid X \perp Y$	separating union
			$\mid () \mid (i, i) \mid M\,[i]$	

Figure 2 Syntax for indices

Effects	ϵ	$::=$	$\langle W; R\rangle$	
Type constructors	d			
Type variables	α			
Value types	A, B	$::=$	α	type variable
			$\mid d$	type constructor
			$\mid A + B$	sum
			$\mid A \times B$	product
			$\mid \mathsf{unit}$	
			$\mid \mathsf{Ref}\,[i]\ A$	named reference cell
			$\mid \mathsf{Thk}\,[i]\ E$	named thunk (with effects)
			$\mid A\,[i]$	application of type to index
			$\mid A\ B$	application of type constructor to type
			$\mid \mathsf{Nm}\,[i]$	name type (name contained in name set i)
			$\mid (\mathbf{Nm} \rightarrow_{\mathbf{Nm}} \mathbf{Nm})\,[M]$	name function type (singleton)
Computation types	C, D	$::=$	$\mathbf{F}\,A$	li**F**t
			$\mid A \rightarrow E$	function
...with effects	E	$::=$	$C \rhd \epsilon$	effects
			$\mid \forall\alpha : K.\,E$	type polymorphism
			$\mid \forall a : \gamma.\,E$	index polymorphism
Typing contexts	Γ	$::=$	$\cdot$	
			$\mid \Gamma, p : A$	ref pointer
			$\mid \Gamma, p : E$	thunk pointer
			$\mid \Gamma, x : A$	value variable

Figure 3 Syntax for types

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

$$\frac{\Gamma \vdash n \text{ wf-name}}{\Gamma \vdash n : \mathbf{Nm}}\ \text{M-const} \qquad \frac{\Gamma, a : \gamma_1 \vdash M : \gamma_2}{\Gamma \vdash (\lambda a.\,M) : (\gamma_1 \rightarrow_{\mathbf{Nm}} \gamma_2)}\ \text{M-abs} \qquad \frac{(a : \gamma) \in \Gamma}{\Gamma \vdash a : \gamma}\ \text{M-var} \qquad \frac{\begin{array}{c}\Gamma \vdash M_1 : (\gamma' \rightarrow_{\mathbf{Nm}} \gamma)\\ \Gamma \vdash M_2 : \gamma'\end{array}}{\Gamma \vdash (M_1\ M_2) : \gamma}\ \text{M-app}$$

$$\frac{}{\Gamma \vdash () : \mathbf{1}}\ \text{M-unit} \qquad \frac{\begin{array}{c}\Gamma \vdash M_1 : \gamma_1\\ \Gamma \vdash M_2 : \gamma_2\end{array}}{\Gamma \vdash (M_1, M_2) : (\gamma_1 * \gamma_2)}\ \text{M-pair} \qquad \frac{\Gamma \vdash M : \mathbf{Nm}}{\begin{array}{c}\Gamma \vdash M.1 : \mathbf{Nm}\\ \Gamma \vdash M.2 : \mathbf{Nm}\end{array}}\ \text{M-append}$$

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

$$\frac{(a : \gamma) \in \Gamma}{\Gamma \vdash a : \gamma}\ \text{sort-var} \qquad \frac{\Gamma \vdash n : \mathbf{Nm}}{\Gamma \vdash \{n\} : \mathbf{NmSet}}\ \text{sort-singleton} \qquad \frac{\Gamma \vdash X : \mathbf{NmSet} \qquad \Gamma \vdash Y : \mathbf{NmSet}}{\Gamma \vdash (X \perp Y) : \mathbf{NmSet}}\ \text{sort-sep-union}$$

$$\frac{}{\Gamma \vdash () : \mathbf{1}}\ \text{sort-unit} \qquad \frac{\Gamma \vdash i_1 : \gamma_1 \qquad \Gamma \vdash i_2 : \gamma_2}{\Gamma \vdash (i_1, i_2) : (\gamma_1 * \gamma_2)}\ \text{sort-pair} \qquad \frac{\Gamma \vdash M : \gamma_1 \rightarrow_{\mathbf{Nm}} \gamma_2 \qquad \Gamma \vdash i : \gamma_1}{\Gamma \vdash M\,[i]\ : \gamma_2}\ \text{sort-apply-function}$$

$$\frac{\Gamma \vdash M : \mathbf{Nm} \rightarrow_{\mathbf{Nm}} \mathbf{Nm} \qquad \Gamma \vdash i : \mathbf{NmSet}}{\Gamma \vdash M\,[i]\ : \mathbf{NmSet}}\ \text{sort-apply-function-pointwise}$$

Figure 4 Sorts statically classify both name terms M, as well as the name indices i that index types

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

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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash C\ \text{ctype} \qquad \Gamma \vdash \epsilon\ \text{wf-effects}}{\Gamma \vdash (C \rhd \epsilon)\ \text{efftype}}\ \text{etype-effects} \qquad \frac{\Gamma, \alpha : K \vdash E\ \text{efftype}}{\Gamma \vdash (\forall \alpha : K.\, E)\ \text{efftype}}\ \text{etype-poly}$$

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

Figure 5 **Kinds statically classify types and effects**

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

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

$$\frac{\Gamma \vdash n \in X}{\Gamma \vdash (\mathsf{name}\ n) : \mathsf{Nm}\,[X]}\ \text{name} \qquad \frac{\Gamma \vdash M_v : (\mathbf{Nm} \rightarrow_{\mathbf{Nm}} \mathbf{Nm}) \qquad M_v =_\beta M}{\Gamma \vdash (\mathsf{nmtm}\ M_v) : (\mathbf{Nm} \rightarrow_{\mathbf{Nm}} \mathbf{Nm})\,[M]}\ \text{nametm}$$

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

Figure 6 **Value typing**

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

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

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

$$\big((C \rhd \epsilon_2)\ \mathsf{after}\ \epsilon_1\big) \;=\; \big(C \rhd (\epsilon_1\ \mathsf{then}\ \epsilon_2)\big) \qquad \begin{aligned} \big((\forall\alpha : K.\, E)\ \mathsf{after}\ \epsilon\big) &= \forall\alpha : K.\,(E\ \mathsf{after}\ \epsilon) \\ \big((\forall a : \gamma.\, E)\ \mathsf{after}\ \epsilon\big) &= \forall a : \gamma.\,(E\ \mathsf{after}\ \epsilon) \\ \big((P \supset E)\ \mathsf{after}\ \epsilon\big) &= P \supset (E\ \mathsf{after}\ \epsilon) \end{aligned}$$

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

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

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

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

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

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

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

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

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

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

Figure 7 Computation typing

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 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:

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

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 $\supset$:

$$\begin{array}{l}\forall X : \textbf{NmSet}. \\ \quad A\,[X] \to B\,[f(X)]\end{array}$$

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

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

can be expressed by

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

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

We use several meta-variables for index expressions. By convention, X, Y, Z, R and W are sets of names; i is any index (perhaps a pair of indices).

4.1.1 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).

4.1.2 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.

4.1.3 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.

4.1.4 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 $M[\mathtt{root}] = \mathtt{root}.1.2$.

4.1.5 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 $\rightarrow_{\mathbf{Nm}}$ classifies index-level functions: $\mathbf{Nm} \rightarrow_{\mathbf{Nm}} \mathbf{Nm}$ takes a name and returns a name.

4.2 Kinds

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

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

Coalescing goes under quantifiers:

$$\big(\forall\alpha : \star.\, C \rhd \langle\emptyset; \emptyset\rangle\big) \text{ after } \langle\emptyset; \{n_3\}\rangle \;=\; \forall\alpha : \star.\, C \rhd \big(\langle\emptyset; \emptyset\rangle \text{ after } \langle\emptyset; \{n_3\}\rangle\big) \;=\; \forall\alpha : \star.\, \big(C \rhd \langle\emptyset; \{n_3\}\rangle\big)$$

4.4 Types

The value types, written A, B, in Figure 3 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 $(\mathbf{Nm} \rightarrow_{\mathbf{Nm}} \mathbf{Nm})[M]$ where M is a function on names.

As usual in call-by-push-value, computation types C and D include a connective $\mathbf{F}$, which "lifts" value types to computation types: $\mathbf{F}\,A$ is the type of computations that, when run, return a value of type A. (Call-by-push-value usually has a connective dual to $\mathbf{F}$, written $\mathbf{U}$, that "th**U**nks" a computation type into a value type; in our system, Thk plays the role of $\mathbf{U}$.)

Computation types also include functions, written $A \rightarrow E$. In standard CBPV, this would be $A \rightarrow C$, not $A \rightarrow E$. 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 $C \rhd \epsilon$, which is the bare computation type C with effects ϵ, as well as universal quantifiers (polymorphism) over types $(\forall\alpha : K.\ E)$ and indices $(\forall a : \gamma.\ E)$.

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, $\mathbf{F}$ and $\rightarrow$. For $\mathbf{F}$, the expression being typed could create a thunk, so we must put that effect somewhere in the syntax. For $\rightarrow$, 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 $A_1 \rightarrow (A_2 \rightarrow \cdots)$ 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 $A_2 \rightarrow \cdots$, meaning that the computation is waiting for a second argument to be pushed. But, since the act of forcing the thunk has the effect of reading the thunk, we need to track this effect in the result type. So we cannot return $A_2 \rightarrow \cdots$, and must instead put effects around $(A_2 \rightarrow \cdots)$. Thus, we need to associate effects to both $\mathbf{F}$ and $\rightarrow$, 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 $\mathbf{F}$ or $\rightarrow$. 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 $C \rhd \epsilon$, where C is a computation type and ϵ represents effects.

Where should these wrappers live? We could add $C \rhd \epsilon$ 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 $(A_1 \rightarrow \mathbf{F}\,A_2) \rhd \epsilon$, "the" effect has to be ϵ. Alas, types like $(C \rhd \epsilon_1) \rhd \epsilon_2$ 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 $E ::= C \rhd \epsilon$, 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.

5 Dynamics

5.1 Dynamics of name terms

Fig. 8 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.

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

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

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

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

Figure 8 Name term evaluation

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

Figure 9 Graphs without dependency or cache structure

$\boxed{G_1 \vdash^M_m e \Downarrow G_2; e'}$ Under graph G, namespace M and current node m, e produces G_2 and e'.

$$\frac{}{G \vdash^M_m t \Downarrow G; t}\ \text{term} \qquad \frac{G_1 \vdash^M_m [v_2/x_2][v_1/x_1]e \Downarrow G_2; e'}{G_1 \vdash^M_m \mathsf{split}((v_1, v_2), x_1.x_2.e) \Downarrow G_2; e'}\ \text{split} \qquad \frac{G_1 \vdash^M_m [v_i/x_i]e_i \Downarrow G_2; e'}{G_1 \vdash^M_m \mathsf{case}(\mathsf{inj}_i\, v, x_1.e_1, x_2.e_2) \Downarrow G_2; e'}\ \text{case}$$

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

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

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

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

Figure 10 Dynamics for non-incremental evaluation

5.2 Dynamics of program expressions

Fig. 9 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. 10 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 Statics 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. 6.1, 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.

6.1 Store typing

$$\dfrac{}{\cdot : \Gamma}\,\text{emp} \qquad \dfrac{G \vdash \Gamma \qquad \begin{array}{l}\Gamma \vdash v : A\\ \Gamma \vdash p \text{ has-store-type } A\end{array}}{\vdash (G, p : v) : \Gamma}\,\text{ref} \qquad \dfrac{G \vdash \Gamma \qquad \begin{array}{l}\Gamma \vdash e : E\\ \Gamma \vdash p \text{ has-store-type } E\end{array}}{\vdash (G, p : e) : \Gamma}\,\text{thunk}$$

Figure 11 Store typing

6.2 Read and Write Sets

Join and Merge Operations. 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 $p \in \mathsf{dom}(H_1)$ and $p \in \mathsf{dom}(H_2)$, then $H_1(p) = H_2(p)$.

Definition 6.1 (Reads/writes). *The effect of an evaluation derived by $\mathcal{D}$, written $\mathcal{D}$ reads R writes W, is defined in Figure 12.*

This is a function over derivations. 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, Eval-scope($\mathcal{D}_0$) by R reads W writes provided that $\mathcal{D}_0$ reads R writes W where $\mathcal{D}_0$ derives the only premise of scope.

6.3 Lemmas

Lemma 6.2 (Weakening). *If* $\Gamma \vdash e : C$ *then* $\Gamma, \Gamma' \vdash e : C$.

Proof. By induction on the derivation of $\Gamma \vdash e : C$. □

Lemma 6.3 (Substitution).

$$\begin{array}{rll}
\mathcal{D} \text{ by Eval-term}() \text{ reads } \emptyset \text{ writes } \emptyset & & \\
\mathcal{D} \text{ by Eval-app}(\mathcal{D}_1, \mathcal{D}_2) \text{ reads } R_1 \cup R_2 \text{ writes } W_1 \perp W_2 & \text{if} & \mathcal{D}_1 \text{ reads } R_1 \text{ writes } W_1 \\
& \text{and} & \mathcal{D}_2 \text{ reads } R_2 \text{ writes } W_2 \\
\mathcal{D} \text{ by Eval-bind}(\mathcal{D}_1, \mathcal{D}_2) \text{ reads } R_1 \cup R_2 \text{ writes } W_1 \perp W_2 & \text{if} & \mathcal{D}_1 \text{ reads } R_1 \text{ writes } W_1 \\
& \text{and} & \mathcal{D}_2 \text{ reads } R_2 \text{ writes } W_2 \\
\mathcal{D} \text{ by Eval-scope}(\mathcal{D}_0) \text{ reads } R \text{ writes } W & \text{if} & \mathcal{D}_0 \text{ reads } R \text{ writes } W
\end{array}$$

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

Figure 12 **Read- and write-sets of a non-incremental evaluation derivation.**

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 6.4 (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* $A = (\mathbf{Nm} \to_{\mathbf{Nm}} \mathbf{Nm})\,[M]$ *then* $v = \mathtt{nmtm}\ M_v$ *where* $M =_\beta (\lambda a.\, M')$ *and* $\cdot \vdash (\lambda a.\, M') : (\mathbf{Nm} \to_{\mathbf{Nm}} \mathbf{Nm})$ *and* $M_v =_\beta M$.

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

Lemma 6.5 (Application and membership commute). *If* $\Gamma \vdash n \in i$ *and* $p = M(n)$ *then* $\Gamma \vdash p \in M(i)$.

Proof. The set $M(i)$ consists of all elements of i, but mapped by function M. Meanwhile, the name p is precisely the name n mapped by M, and since we know that n is in i, we have that p is in the M-mapping of i, $M(i)$. □

6.4 Main proof

Theorem 6.6 (Subject Reduction for Reference Semantics). *If*

- $\Gamma_1 \vdash M : \mathbf{Nm} \to_{\mathbf{Nm}} \mathbf{Nm}$
- $\mathcal{S} :: \Gamma_1 \vdash^M e : C \rhd \langle W; R \rangle$

- $\vdash G_1 : \Gamma_1$
- $\mathcal{D} :: G_1 \vdash^{M}_{m} e \Downarrow G_2; t$

there exists $\Gamma_2 \supseteq \Gamma_1$ *such that*

- $\vdash G_2 : \Gamma_2$
- $\Gamma_2 \vdash t : C \rhd \langle \emptyset; \emptyset \rangle$
- $\mathcal{D}$ *reads* $R_{\mathcal{D}}$ *writes* $W_{\mathcal{D}}$
- $\langle W_{\mathcal{D}}; R_{\mathcal{D}} \rangle \preceq \langle W; R \rangle$

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

- **Case** $$\frac{\Gamma \vdash v : A}{\Gamma \vdash^{M} \mathtt{ret}(v) : \big((\mathbf{F}\, A) \rhd \langle \emptyset; \emptyset \rangle\big)}\ \text{ret}$$

	$(e = t)$ 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 the corresponding rule in Def. 6.1
☞	$\langle W_{\mathcal{D}}; R_{\mathcal{D}} \rangle \preceq \langle W; R \rangle$	All are empty

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

	$(W = \emptyset)$ and $(R = X)$	Given
	$\Gamma_1 \vdash v : \mathsf{Ref}\,[X]\ A$	Given
	$\exists p.\ (v = \mathtt{ref}\ p)$	Lemma 6.4 (Canonical Forms)
	$\Gamma_1 \vdash p \in X$	″
	$\Gamma_1 \vdash p\ \mathsf{has\text{-}store\text{-}type}\ A$	By inversion of value typing rule
	$\exists v_p.\ G_1(p) = v_p$	Inversion of $\vdash G_1 : \Gamma_1$ with p $\mathsf{has\text{-}store\text{-}type}$ A
	$\Gamma_1 \vdash v_p : A$	″
	$(\Gamma_2 = \Gamma_1)$ 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 the corresponding rule in Def. 6.1
☞	$\langle W_{\mathcal{D}}; R_{\mathcal{D}} \rangle \preceq \langle W; R \rangle$	by above equality $W_{\mathcal{D}} = W = \emptyset$, . . . and inequality for $(R_{\mathcal{D}} = \{p\}) \subseteq (X = R)$.

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

	$(W = \emptyset)$ and $(R = X)$	Given
	$\Gamma_1 \vdash v : \mathsf{Thk}\,[X]\ E$	Given
	$\exists p.\ (v = \mathtt{thunk}\ p)$	Lemma 6.4 (Canonical Forms)
	$\Gamma_1 \vdash p \in X$	″
	$\Gamma_1 \vdash p$ has-store-type E	By inversion of value typing rule
	$\exists e_p.\ G_1(p) = e_p$	Inversion of $\vdash G_1 : \Gamma_1$ with p has-store-type E
$\mathcal{S}_0$ ::	$\Gamma_1 \vdash e_p : C \rhd \epsilon$	″
$\mathcal{D}_0$ ::	$G_1 \vdash^{M}_{m} e_p \Downarrow G_2; t$	Inversion of $\mathcal{D}$
☞	$\vdash G_2 : \Gamma_2$	By IH 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 the corresponding rule in Def. 6.1
☞	$\langle W_{\mathcal{D}_0}; R_{\mathcal{D}_0} \rangle \preceq \langle W; R \rangle$	by above equality $W_{\mathcal{D}} = W = \emptyset$,
		. . . and inequality for $(R_{\mathcal{D}} = \{p\}) \subseteq (X = R)$.

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

$\mathcal{S}_0$ ::	$\Gamma \vdash^{M \circ M'} e_0 : C \rhd \langle W; R \rangle$	Subderivation 2 of $\mathcal{S}$
$\mathcal{D}$ ::	$G_1 \vdash^{M}_{m} \mathtt{scope}(M', e_0) \Downarrow G_2; t$	Given
$\mathcal{D}_0$ ::	$G_1 \vdash^{M \circ M'}_{m} e_0 \Downarrow G_2; t$	By inversion (scope)
	$\Gamma_1 \vdash M : \mathbf{Nm} \to_{\mathbf{Nm}} \mathbf{Nm}$	Assumption
	$\Gamma_1 \vdash M' : \mathbf{Nm} \to_{\mathbf{Nm}} \mathbf{Nm}$	Subderivation 1 of $\mathcal{S}$
	$\Gamma_1, x : \mathbf{Nm} \vdash M'\,x : \mathbf{Nm}$	By rule t-app
	$\Gamma_1, x : \mathbf{Nm} \vdash M\,(M'\,x) : \mathbf{Nm}$	By rule t-app
	$\Gamma_1 \vdash \lambda x.\, M\,(M'\,x) : \mathbf{Nm} \to_{\mathbf{Nm}} \mathbf{Nm}$	By rule t-abs
	$\Gamma_1 \vdash M \circ M' : \mathbf{Nm} \to_{\mathbf{Nm}} \mathbf{Nm}$	By definition of $M \circ M'$
☞	$\vdash G_2 : \Gamma_2$	By IH 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 the corresponding rule in Def. 6.1
	$\langle W_{\mathcal{D}}; R_{\mathcal{D}} \rangle = \langle W_{\mathcal{D}_0}; R_{\mathcal{D}_0} \rangle$	″
☞	$\langle W_{\mathcal{D}}; R_{\mathcal{D}} \rangle \preceq \langle W; R \rangle$	By above equalities

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

	$C = \mathbf{F}\,(\mathsf{Thk}\,[M(X)]\ E)$ and $R = \emptyset$ and $W = M(X)$	Given from $\mathcal{S}$
	$\Gamma_1 \vdash v : \mathsf{Nm}\,[X]$	Subderivation
	$(v = \mathtt{name}\ n)$ and $(n \in X)$	Lemma 6.4 (Canonical Forms)
	$M\ n \Downarrow p$ and $R_{\mathcal{D}} = \emptyset$ and $W_{\mathcal{D}} = \{p\}$	Given from $\mathcal{D}$
	$G_2 = (G_1, p : e)$	″
	$\Gamma_2 = (\Gamma_1, p : \mathsf{Thk}\,[p]\ E)$	Suppose
☞	$\vdash G_2 : \Gamma_2$	By application of store typing rule (Fig. 11)
	$\Gamma_2 \vdash p$ has-store-type E	By inversion of value typing rule
	$\Gamma_2 \vdash \mathsf{ref}\ p : \mathsf{Ref}\,[p]\ E$	By rule thunk
☞	$\Gamma_2 \vdash^{M} \mathtt{ret}(\mathtt{thunk}\ p) : \mathtt{ret}(\mathsf{Thk}\,[p]\ A) \rhd \langle\emptyset;\emptyset\rangle$	By rule ret
☞	$\mathcal{D}$ reads $R_{\mathcal{D}}$ writes $W_{\mathcal{D}}$ and $W_{\mathcal{D}} = \{p\}$	By the corresponding rule in Def. 6.1
	$n \in X$	Above
	$M(n) \in M(X)$	By “definition of name term application”
	$M(n) \in W$	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** $$\frac{\Gamma_1 \vdash v_1 : \mathsf{Nm}\,[X] \qquad \Gamma_1 \vdash v_2 : A}{\Gamma_1 \vdash^{M} \mathtt{ref}(v_1, v_2) : \mathbf{F}\,(\mathsf{Ref}\,[M(X)]\ A) \rhd \langle M(X); \emptyset\rangle}\ \text{ref}$$

	$C = \mathbf{F}\,(\mathsf{Ref}\,[M(X)]\ A)$ 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 6.4 (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 application of store typing rule (Fig. 11)
	$\Gamma_2 \vdash p$ has-store-type A	By inversion of value typing rule
	$\Gamma_2 \vdash \mathsf{ref}\ p : \mathsf{Ref}\,[p]\ A$	By rule ref
☞	$\Gamma_2 \vdash^{M} \mathtt{ret}(\mathsf{ref}\ p) : \mathtt{ret}(\mathsf{Ref}\,[p]\ A) \rhd \langle\emptyset;\emptyset\rangle$	By rule ret
☞	$\mathcal{D}$ reads $R_{\mathcal{D}}$ writes $W_{\mathcal{D}}$ and $W_{\mathcal{D}} = \{p\}$	By the corresponding rule in Def. 6.1
	$n \in X$	Above
	$M(n) \in M(X)$	By “definition of name term application”
	$M(n) \in W$	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** $$\frac{\Gamma_1 \vdash^{M} e_1 : (\mathbf{F}\,A \rhd \epsilon_1) \qquad \Gamma_1, x : A \vdash^{M} e_2 : (C \rhd \epsilon_2)}{\Gamma_1 \vdash^{M} \mathtt{let}(e_1, x.e_2) : C \rhd (\epsilon_1\ \mathsf{then}\ \epsilon_2)}\ \text{let}$$

	$\vdash G_1 : \Gamma_1$	given
$\mathcal{S}_1$ ::	$\Gamma_1 \vdash^M e_1 : \mathbf{F}\,A \rhd \epsilon_1$	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 IH on $\mathcal{S}_1$
	$\Gamma_{12} \vdash t_1 : \mathbf{F}\,A \rhd \langle\emptyset;\emptyset\rangle$	″
	$\mathcal{D}_1$ 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 6.2 (Weakening)
	$\Gamma_{12} \vdash^M [v/x]e_2 : C \rhd \epsilon_2$	Lemma 6.3 (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 IH 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 $W_{\mathcal{D}_2}$	″
	$\langle W_{\mathcal{D}_2}; R_{\mathcal{D}_2}\rangle \preceq \epsilon_2$	″
	$\langle W_{\mathcal{D}_2}; R_{\mathcal{D}_2}\rangle \preceq \langle W_2, R_2\rangle$	″

	$W_1 \perp W_2, R_1 \perp W_2$	Definition of ϵ_1 then ϵ_2
	$W_{\mathcal{D}_1} \perp W_{\mathcal{D}_2}, R_{\mathcal{D}_1} \perp W_{\mathcal{D}_2}$	Since $W_{\mathcal{D}_1} \subseteq W_1$, $W_{\mathcal{D}_2} \subseteq W_2$ and $R_{\mathcal{D}_1} \subseteq R_1$
	$W_{\mathcal{D}} = W_{\mathcal{D}_1} \perp W_{\mathcal{D}_2}$	By the corresponding rule in Def. 6.1
	$R_{\mathcal{D}} = R_{\mathcal{D}_1} \cup (R_{\mathcal{D}_2} - W_{\mathcal{D}_1})$	″
☞	$\mathcal{D}$ 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** $$\frac{\Gamma \vdash^M e : \big((A \to E) \rhd \epsilon_1\big) \qquad \Gamma \vdash v : A}{\Gamma \vdash^M (e\ v) : (E \;\mathsf{after}\; \epsilon_1)}\ \text{app}$$
  (Analogous to case for let).

- **Case** $$\frac{\Gamma \vdash^M v : (A_1 \times A_2) \qquad \Gamma, x_1 : A_1, x_2 : A_2 \vdash^M e : E}{\Gamma \vdash^M \mathsf{split}\,(v, x_1.x_2.e) : E}\ \text{split}$$
  (Analogous to case for let, with the addition of canonical forms).

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

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

	$\Gamma_1 \vdash v_M : (\mathbf{Nm} \rightarrow_{\mathbf{Nm}} \mathbf{Nm})\,[M]$	Given
	$\quad v_M = \mathtt{nmtm}\ M_v$	Lemma 6.4 (Canonical Forms)
	$\quad M =_\beta (\lambda a.\, M')$	″
	$\quad \cdot \vdash \lambda a.\, M' : (\mathbf{Nm} \rightarrow_{\mathbf{Nm}} \mathbf{Nm})$	″
	$\quad M_v =_\beta M$	″
	$\Gamma_1 \vdash v : \mathsf{Nm}\,[i]$	Given
	$\quad v = \mathtt{name}\ n$	Lemma 6.4 (Canonical Forms)
	$\quad \Gamma \vdash n \in i$	″
	$\quad M \Downarrow_M (\lambda a.\, M')$	By inversion on $\mathcal{D}$ (name-app)
	$\quad [n/a]M' \Downarrow_M p$	″
	$\quad (\Gamma_2 = \Gamma_1), (G_2 = G_1)$	Suppose
☞	$\quad \vdash G_2 : \Gamma_2$	By above equalities and $G_1 \vdash \Gamma_1$
	$\Gamma_1 \vdash n \in i$	Above
	$\Gamma_1 \vdash p \in M(i)$	Lemma 6.5 (Application and membership commute)
		. . . with $p = M(n)$
	$\Gamma_1 \vdash \mathtt{name}\ p : \mathsf{Nm}\,[M(i)]\,)$	By rule name
☞	$\Gamma_1 \vdash \mathtt{ret}(\mathtt{name}\ p) : \mathbf{F}\,(\mathsf{Nm}\,[M(i)]\,) \rhd \langle\emptyset;\emptyset\rangle$	By rule ret
☞	$\quad \mathcal{D}$ by Eval-name-app reads $\emptyset$ writes $\emptyset$	By the corresponding rule in Def. 6.1
☞	$\quad (\mathsf{R}_{\mathcal{D}} = \mathsf{R} = \emptyset), (W_{\mathcal{D}} = W = \emptyset)$	By above equalities

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

	$\mathcal{S}_0 :: \Gamma_1, a : \gamma \vdash^M t : E$	subderivation
	$\mathcal{D}_0 :: \quad G_1 \vdash^M_m e \Downarrow G_2; t$	subderivation
	$\quad \exists \Gamma_2 \subseteq \Gamma_1$	By i.h.
☞	$\qquad \vdash G_2 : \Gamma_2$	″
	$\qquad \mathcal{D}_0$ reads $\mathsf{R}_{\mathcal{D}_0}$ writes $W_{\mathcal{D}_0}$	″
	$\qquad \Gamma_2 \vdash t : E$	″
	$\qquad \langle \mathsf{R}_{\mathcal{D}_0}; W_{\mathcal{D}_0} \rangle \preceq \langle \mathsf{R}; W \rangle$	″
☞	$\quad \Gamma_2 \vdash^M t : (\forall a : \gamma.\, E)$	By typing rule
☞	$\qquad \mathcal{D}$ reads $\mathsf{R}_{\mathcal{D}}$ writes $W_{\mathcal{D}}$	By the corresponding rule in Def. 6.1
☞	$\qquad \langle \mathsf{R}_{\mathcal{D}}; W_{\mathcal{D}} \rangle \preceq \langle \mathsf{R}; W \rangle$	By set theory

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

	$\mathcal{S}_0 :: \quad \Gamma_1 \vdash^{M} e : (\forall a : \gamma.\, E)$	subderivation
	$\mathcal{D}_0 :: \quad G_1 \vdash^{M}_{m} e \Downarrow G_2; t$	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
	$\Gamma_2 \vdash i : \gamma$	weakening
☞	$\Gamma_2 \vdash^{M} t : [i/a]$	By typing rule
☞	$\mathcal{D}$ reads $R_{\mathcal{D}}$ writes $W_{\mathcal{D}}$	By the corresponding rule in Def. 6.1
☞	$\langle R_{\mathcal{D}}; W_{\mathcal{D}} \rangle \preceq \langle R; W \rangle$	By set theory

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

  (analogous to case for AllIndexIntro.)

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

  (analogous to case for AllIndexElim.)

□

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 annotations. 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 these systems is beyond the scope of this report; for an overview of some of them, 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: each allocated object is uniquely named. Second, names have structure and are related by that structure: the names root.1.1 and root.1.2 are not arbitrary distinct names (because they share a prefix).

Techniques for general-purpose incremental computation. A computation is *incremental* if repeating it with a changed input is faster than from-scratch recomputation. Incremental computation (IC, for short) is ubiquitous in computing.

Across computer science, many have studied *incremental algorithms* (variously called *online algorithms* and *dynamic algorithms*), where the traditional fixed-input computing paradigm is replaced with one where programs run repeatedly on changing inputs. Typically, researchers study each such problem in isolation, and they exploit domain-specific structure to justify a special incremental algorithm for the given domain. For example, search algorithms in robotics are incremental versions of standard search algorithms, and motion simulation algorithms can be viewed as incremental versions of standard computational geometry algorithms (e.g., see Agarwal et al. (1999, 2001); Alexandron et al. (2005); Basch (1999); Basch et al. (2004), and see Agarwal et al. (2002) for a survey).

In contrast to solving online problems in a domain-specific, one-off fashion, the programming languages community offers several threads of research that attempt to offer *general-purpose programming language abstractions*, so that the language (not the program) abstracts over the incremental aspect of the desired program behavior Ramalingam and Reps (1993, 1996); Demers et al. (1981); Reps and Teitelbaum (1988); Liu and Teitelbaum (1995); Liu et al. (1998); Acar (2009); Guo and Engler (2011). These incremental languages are aware of incremental change, and through specially-designed abstractions, they help the programmer avoid thinking about changes directly. Instead, she thinks about how to apply the abstractions, which are general purpose.

Through careful language design, modern incremental abstractions elevate implementation questions such as *"how does this particular change pattern affect a particular incremental state of the system?"* into simpler, more general questions, answered with special programming abstractions (e.g., via special annotations). These abstractions identify changing data and reusable sub-computations (Acar et al. 2008a; Hammer et al. 2015b, 2014, 2009; Mitschke et al. 2014), and they allow the programmer to relate the expression of incremental algorithm with 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 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. 2015b). Further, researchers have shown that IC can deliver *asymptotic* speedups (Acar et al. 2007; Hammer et al. 2007; Acar et al. 2008c,b,a; Acar 2009; Sümer et al. 2011), and has even addressed open problems (Acar et al. 2010). These IC abstractions exist in many languages (Shankar and Bodik 2007; Hammer et al. 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; 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

In this report, we motivate the need for generic naming strategies in programs that use nominal memoization. We define a refinement type system that gives practical static approximations of these 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 fundemental to IC. To address this fact, 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 proram 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:

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

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:

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

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 explicates the over-written set of names F. 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 and name sets.

8.1 Acknowledgments

This material is based in part upon work supported by a gift from Mozilla, and support from the National Science Foundation under grant number CCF-1619282. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of Mozilla or the National Science Foundation.

References

Umut A. Acar. Self-adjusting computation (an overview). In *Proceedings of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 2009.

Umut A. Acar and Ruy Ley-Wild. Self-adjusting computation with Delta ML. In *Advanced Functional Programming*. 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*, 2008a.

Umut A. Acar, Guy E. Blelloch, Kanat Tangwongsan, and Duru Türkoğlu. Robust kinetic convex hulls in 3D. In *Proceedings of the 16th Annual European Symposium on Algorithms*, September 2008b.

Umut A. Acar, Alexander Ihler, Ramgopal Mettu, and Özgür Sümer. Adaptive inference on general graphical models. In *Uncertainty in Artificial Intelligence (UAI)*, 2008c.

Umut A. Acar, Andrew Cotter, Benoît Hudson, and Duru Türkoğlu. Dynamic well-spaced point sets. In *Symposium on Computational Geometry*, 2010.

Pankaj K. Agarwal, Julien Basch, Mark de Berg, Leonidas J. Guibas, and John Hershberger. Lower bounds for kinetic planar subdivisions. In *SCG '99: Proceedings of the 15th Annual Symposium on Computational Geometry*, pages 247–254. ACM Press, 1999.

Pankaj K. Agarwal, Leonidas J. Guibas, John Hershberger, and Eric Veach. Maintaining the extent of a moving set of points. *Discrete and Computational Geometry*, 26(3):353–374, 2001.

Pankaj K. Agarwal, Leonidas J. Guibas, Herbert Edelsbrunner, Jeff Erickson, Michael Isard, Sariel Har-Peled, John Hershberger, Christian Jensen, Lydia Kavraki, Patrice Koehl, Ming Lin, Dinesh Manocha, Dimitris Metaxas, Brian Mirtich, David Mount, S. Muthukrishnan, Dinesh Pai, Elisha Sacks, Jack Snoeyink, Subhash Suri, and Ouri Wolfson. Algorithmic issues in modeling motion. *ACM Comput. Surv.*, 34(4):550–572, 2002.

Giora Alexandron, Haim Kaplan, and Micha Sharir. Kinetic and dynamic data structures for convex hulls and upper envelopes. In *9th Workshop on Algorithms and Data Structures (WADS). Lecture Notes in Computer Science*, volume 3608, pages 269—281, aug 2005.

Julien Basch. *Kinetic Data Structures*. PhD thesis, Department of Computer Science, Stanford University, June 1999.

Julien Basch, Jeff Erickson, Leonidas J. Guibas, John Hershberger, and Li Zhang. Kinetic collision detection between two simple polygons. *Computational Geometry*, 27(3):211–235, 2004.

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.

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

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

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, 1986.

Philip J. Guo and Dawson Engler. Using automatic persistent memoization to facilitate data analysis scripting. In *2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 287–297, 2011.

Matthew Hammer, Umut A. Acar, Mohan Rajagopalan, and Anwar Ghuloum. A proposal for parallel self-adjusting computation. In *DAMP '07: Declarative Aspects of Multicore Programming*, 2007.

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*, 2015a.

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

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 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.

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

Yanhong A. Liu, Scott Stoller, and Tim Teitelbaum. Static caching for incremental computation. *ACM Transactions on Programming Languages and Systems*, 20(3):546–585, 1998.

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

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

G. Ramalingam and T. Reps. On the computational complexity of dynamic graph algorithms. *Theoretical Computer Science*, 158(1–2):233–277, 1996.

T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language Based Editors*. Springer-Verlag, 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.