Fungi: A typed, functional language for programs that dynamically name their own cached dependency graphs

MATTHEW A. HAMMER, University of Colorado Boulder
JOSHUA DUNFIELD, University of British Columbia
KYLE HEADLEY, University of Colorado Boulder
MONAL NARASIMHAMURTHY, University of Colorado Boulder
DIMITRIOS J. ECONOMOU, University of Colorado Boulder

Many programming language techniques for incremental computation employ programmer-specified *names* for cached information. At runtime, each name identifies a "cache location" for a dynamic data value or a sub-computation; in sum, these cache location choices guide change propagation and incremental (re)execution.

We call a cache location name *precise* when it identifies at most one value or subcomputation; we call all other names *imprecise*, or *ambiguous*. At a minimum, cache location names must be precise to ensure that change propagation works correctly; yet, reasoning statically about names in incremental programs remains an open problem.

As a first step, this paper defines and solves the *precise name problem*, where we verify that incremental programs with explicit names use them precisely. We formalize and implement our approach in the form of Fungi, a typed, functional language for programs that name their own cached dependency graphs. In particular, we give a core calculus for Fungi with a refinement type and effect system that we prove sound: every well-typed Fungi program uses names precisely. We also demonstrate that this type system is expressive by verifying example programs that compute over efficient representations of incremental sequences and sets. Beyond verifying these programs, our type system also describes their dynamic naming strategies, e.g., for library documentation purposes. Finally, we give a (sound) bidirectional variant of our proposed type and effects system, and a closely related prototype implementation of Fungi in Rust, as a deeply embedded DSL.

1 INTRODUCTION

In many computing scenarios, the result of running a program is not merely a result, but also *a cached dependency graph of how the program computed it*. For instance, language-based incremental computation techniques strive to improve the asymptotic time complexity of programs using dynamic dependency graphs that provide dynamic structure for *function call caching*, *cache invalidation* and *change propagation* [Acar 2005; Acar et al. 2006a,b, 2009; Acar and Ley-Wild 2009; Hammer and Acar 2008; Hammer et al. 2009; Ley-Wild et al. 2008; Chen et al. 2012].

Relatedly, the implementation techniques that underly some functional reactive programming (FRP) systems also build and maintain dynamic dependency graphs [Cooper and Krishnamurthi 2006; Krishnaswami and Benton 2011; Krishnaswami 2013; Czaplicki and Chong 2013]. Meanwhile, software build systems and other large-scale, batch-processing systems often cache their results, if not the dependency graph that produced them [Bhatotia et al. 2011, 2015; Erdweg et al. 2015b].

In all of these systems, running a program builds (and then helps maintain) a cache of intermediate results and often, a *dynamic dependency graph*, which represents the steps of its "most recent" execution. In fact, this graph undergoes incremental re-execution, where its existence permits the system to soundly avoid recomputing the work of *unaffected* subcomputations.

Authors' addresses: Matthew A. Hammer, University of Colorado Boulder, Department of Computer Science; Joshua Dunfield, University of British Columbia, Department of Computer Science; Kyle Headley, University of Colorado Boulder, Department of Computer Science; Monal Narasimhamurthy, University of Colorado Boulder, Department of Computer Science; Dimitrios J. Economou, University of Colorado Boulder, Department of Computer Science.

2018. XXXX-XXXX/2018/5-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In the most general variants of these systems, the programming model demands that incremental algorithms employ explicit *names*, where each name identifies a cache location for dynamic data (a dynamic pointer allocation), or a cache location for dynamic sub-computations, e.g., the arguments, dependencies, and results of a recursive function call [Hammer et al. 2015].

Through an incremental algorithm's input data structures, these names enter the program's dynamic data flow, where they mix with dynamic computations, perhaps getting combined with one another, dynamically; eventually, the program *uses* each name at a dynamic allocation site within the computation. Specifically, each dynamic allocation forms a *uniquely named* node in the program's (implicitly-built) dependency graph, which caches function results and the dynamic dependency edges (runtime effects) that relate the cached data and subcomputations.

As described above, (dynamically computed) names afford the program explicit control over naming its own dependency graph nodes. Further, each cache location name n simultaneously describes both *cache allocation* (as above) and *cache eviction*, where the former corresponds with the "functional view" of the program (as it were running from-scratch), and the latter behavior corresponds with the "incremental view" of the program, from within its *imperative*, incremental runtime system. In particular, names guide incremental cache invalidation, where the incremental runtime system follows the program's naming choices to *re-associate* an existing name with new content (either new data, or a new subcomputation). This eviction removes prior results and cached dependency edges in the graph, replacing them with updated versions, related in part or *not at all* with the prior content.

This control comes at a cost: Explicit allocation names are prone to misuse, where each such misuse might undermine type safety, change propagation soundness, or change propagation efficiency, depending on the system (Sec. 9 discusses past work in depth).

In this paper, we present Fungi, a typed, functional language for programs that name their own cached dynamic dependency graphs. Fungi's type system defines a verification problem, *the precise name problem*, for programs that compute with explicit allocation names. Specifically, the type system attempts to prove that for all possible inputs, in every execution, each name allocated by the program is *precise*, and not ambiguous. For an evaluation derivation $\mathcal{D}$, we say that an allocated pointer name is precise when it has at most one definition in $\mathcal{D}$, and otherwise we say that a name is ambiguous (imprecise) when it identifies two or more data structures and/or sub-computations.

Across *distinct, successive* incremental re-evaluations of a Fungi program, with a series of incrementally related evaluation derivations $\mathcal{D}_1, \mathcal{D}_2, \ldots$, each name n may be associated with different values, when and if this content changes incrementally (e.g., first value v_1, then later, v_2 such that $v_1 \neq v_2$). In these situations, name n is still precise exactly when, *in each execution* $\mathcal{D}_i$, the Fungi's effects are consistent with purely functional execution, and in particular, these effects associate at most one unique value v_i with the pointer named n.

In summary, Fungi's type and effect system *statically* reasons about one execution at a time (not incremental re-executions) and proves that *dynamically*, in each such execution, each name is used uniquely, if at all. Before outlining our approach and contributions, we give background on incremental programming with names, with examples.

1.1 Background: Incremental computation with names

Incremental programs employ explicit cache location names (1) to choose cached allocation names *deterministically*, and (2) to witness *dynamic independence* between subproblems, thereby improving incremental reuse.

Deterministic allocation via precise names. The first role of explicit names for incremental computing concerns *deterministic store allocation*, which permits us to give a meaningful definition to *cached* allocation. To understand this role, consider these two evaluation rules (each of the judgement form $\sigma; e \Downarrow \sigma'; v$) for reference cell allocation:

$$\frac{\ell \notin \mathsf{dom}(\sigma)}{\sigma; \mathtt{ref}_1(v) \Downarrow \sigma\{\ell \mapsto v\}; \mathsf{ref}\ \ell}\ \mathsf{alloc}_1 \qquad\qquad \frac{n \notin \mathsf{dom}(\sigma)}{\sigma; \mathtt{ref}_2(n, v) \Downarrow \sigma\{n \mapsto v\}; \mathsf{ref}\ n}\ \mathsf{alloc}_2$$

The left rule is conventional: ref_1 allocates a value v at a store location ℓ; because the program does not determine ℓ, the implementor of this rule has the freedom to choose ℓ any way that they wish. Consequently, this program is *not* deterministic, and hence, *not* a function. Because of this fact, it is not immediately obvious what it means to *cache* this kind of dynamic allocation using a technique like *function caching* [Pugh and Teitelbaum 1989], or techniques based on it. To address this question, the programmer can determine a name n for the value v, as in the right rule. The point of this version is to expose the naming choice directly to the programmer, and their incremental algorithm.

In some systems, the programmer chooses this name n as the hash value of value v. This naming style is often called "hash-consing". We refer to it as *structural naming*, since by using it, the name of each ref cell reflects the *entire structure* of that cell's content.[1] By contrast, in an incremental system with explicit names, the programmer generally chooses n to be related to the *evaluation context of using* v, and often, to be *independent* of the value v itself, providing *dynamic independence* for subcomputations may otherwise be treated as dependent. Notably, since structural names lack *any* independence with the content that they name, structural names cannot provide dynamic independence. We give one example of an explicit naming strategy below, and many more in Sec. 2.

Dynamic independence via explicit *names.* Programmers of incremental computations augment ordinary algorithms with names, permitting incremental computing techniques like memoization and change propagation to exploit dynamic independence in the cached computation.

As a simple illustrative example, consider the left version of `rev` below, the recursive program that reverses a list. After being reversed, suppose the input list to `rev` undergoes incremental insertions and removals of list elements. To respond to these input changes, we wish to update the output (the reversed list) by using the algorithm for `rev` below, along with general-purpose incremental computing techniques, including memoization of recursive calls to `rev`, marked by the `memo` keyword:

```
rev : List -> List -> List
rev l r = match l with
 | Nil      ⇒ r
 | Cons(h,t) ⇒
    let rr = ref(hash(r), r) in
    memo(rev !t (Cons(h, rr)))
```

Structural naming:
`rr`'s name is a (hash) function of `r`.

```
rev : List -> List -> List
rev l r = match l with
 | Nil         ⇒ r
 | Cons(n, h, t) ⇒
    let rr = ref(n, r) in
    memo(rev !t (Cons(n, h, rr)))
```

Explicit naming:
`rr`'s name is `n`, and is *independent* of `r`.

The function `rev` reverses a list of alternating `Cons` cells and reference cells. (Within each run of `rev`, the input and output data is immutable; however, across successive, incremental runs of this program, these reference cells generally change value.) Apart from the placement of these reference operations and the `memo` keyword, this functional program is conventional: it reverses the input using an accumulator `r` that holds the reversed prefix of the input list. In the `Cons` case,

[1]The structural hashing approach is closely related to Merkle trees, the basis for revision-control systems like `git`.

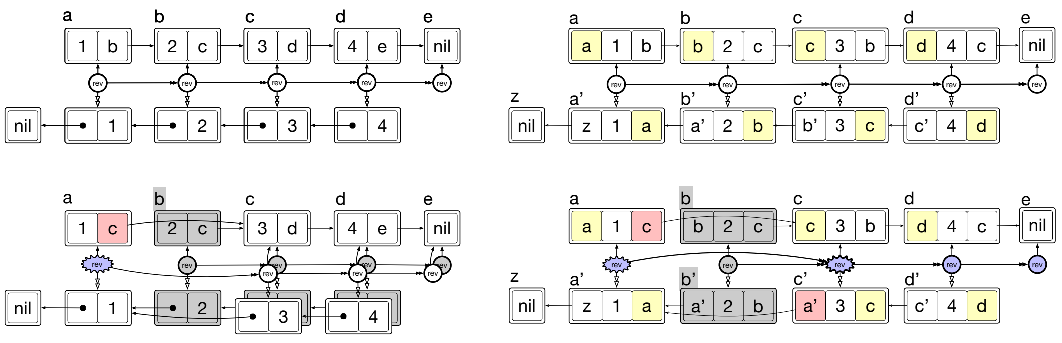

Fig. 1. Four runs of `rev`: two successive runs each of the *structural name version* (left hand side, top and bottom), and the *explicit name version* (right hand side, top and bottom) over a common input list $[1, 2, 3, 4]$, and input change (removal of element 2, at position $\mathtt{b}$). Each thunk for `rev` contains several effect edges: to the `ref` that it observes (upward), to the ref cell that it allocates (downward), and to the thunk it allocates and calls as tail-recursive subroutine (rightward).
Table 1 gives further details, listing the arguments and results of each call to `rev` shown above.

Structural-name version of `rev`					**Explicit**-name version of `rev`				
Run 1			Run 2		Run 1			Run 2	
Input: [1,2,3,4]			Input: [1,3,4]		Input: [1,2,3,4]			Input: [1,3,4]	
arg. `l`	arg. `r`	⇓	arg. `r`	⇓	arg. `l`	arg. `r`	⇓	arg. `r`	⇓
a	[]	[4,3,2,1]	[]	[4,3,1]	a	z	d′	z	d′
b	[1]	[4,3,2,1]	*unobserved*		b	a′	d′	*unobserved*	
c	[2,1]	[4,3,2,1]	[1]	[4,3,1]	c	b′	d′	a′	d′
d	[3,2,1]	[4,3,2,1]	[3,1]	[4,3,1]	d	c′	d′	c′	d′
e	[4,3,2,1]	[4,3,2,1]	[4,3,1]	[4,3,1]	e	d′	d′	d′	d′

Table 1. Four runs of `rev`: two successive runs each of the *structural name version* (left), and the *explicit name version* (right) over a common input list $[1, 2, 3, 4]$, and input change (removal of element 2, at position $\mathtt{b}$). Note that Fig. 1's graph illustration relates the input names ($\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$), allocated pointers ($\{\mathtt{a}', \mathtt{b}', \mathtt{c}', \mathtt{d}'\}$) and list contents.

the function `rev` calls itself on the (dereferenced) tail `!t`, pushing the list element `h` onto the accumulator `r`.

During incremental (re)executions, we intend for change propagation to exploit the fact that each recursive step of `rev` is, *in a sense*, independent of the other recursive steps. However, the left version's use of `memo` fails to achieve this effect on its own because, in another sense, this version's recursive steps are *dependent* on each other. In fact, encoding the desired cache eviction and change propagation strategy requires an additional name `n` in each `Cons` cell, as shown in the right hand version of `rev`.

To understand why, we consider how change propagation re-evaluates and how memoization reuses the cached calls to function `rev` from the initial run, on an initial input $[1, 2, 3, 4]$. For each recursive call to `rev`, the system cache stores a thunk; each thunk records its result, the input list, and reversed accumulator at that point in the dynamic call graph. We illustrate these initial input-output structures in Fig. 1 (top left).

Suppose an external user mutates the input at pointer a, removing element 2, and that they demand the updated output of `rev`. Behind the scenes, change propagation re-evaluates the step(s) of `rev` where the input changed; in this case, it was when the program first observed a, the changed pointer, in the first recursive call. When this thunk reevaluates in Run 2, it re-allocates a reference cell to hold [1], which (structurally) matches its allocation from the initial run. Next, the first `rev` thunk (again) recurs, attempting to match a prior recursive call with the same pair of arguments. However, these arguments have fallen out of alignment with those of the first run. Table 1 lists the arguments and results of each thunk shown in Fig. 1.

Due to this "misalignment" with the prior run's argument pairs, the second run repeatedly fails to `memo`-match the cached calls to `rev`; likewise, each attempt to hash-cons `rr` after the first fails, since each value of `r` is distinct, structurally, from those of the prior run: All those prior allocations contained lists with the element 2, now absent. Consequently, change propagation will evaluate *all the calls* that include and follow the input list change; in this case, there are *four* such places (at a, c, d, and e); moreover, for all but the first, the misaligned arguments also identify structurally-distinct thunks, leaving "stale" copies of old output and thunks in the cache, without an obvious eviction strategy (What event should "trigger" this eviction? What if the element 2 is later re-inserted?).

The failure of `rev` to exploit memoization above is a simple example of a general pattern, also found in the output accumulators of quicksort and quickhull, and more generally, in many cases when a recursive algorithm allocates and consumes dynamic data structures. To address this memoization issue, we uniquely name each input `Cons` cell, and use these names to determine the pointers used to store (and overwrite) the output [Hammer et al. 2015].

In the explicit-name version (right hand version of `rev`), each name `n` localizes any change to the accumulator argument `r`, since the dynamic dependency graph will record the fact that `rev` allocates, but never accesses, these reference cells. More conceptually, these reference cells' (independent) names and dynamic dependency structure witness that, in fact, the recursive steps of running `rev` on a list are independent from each other. Unlike the structural-name version above, where we reevaluate $O(n)$ thunks per *single* input insertion or deletion, the explicit-name version here only requires $O(1)$ re-evaluations per input insertion or deletion, which is optimal.

Fig. 1 illustrates the initial and updated input and outputs of this right hand version; in particular, it shows that change propagation avoids reconstructing the full output prefix, by instead reflecting the input mutation (at a) into a corresponding output mutation (at c'), leaving no residual cache garbage, except for the old call to `rev` at b, and its `ref` allocation. Were we to re-insert 2, change propagation would restore this (temporarily stale) thunk, and its allocation of b'. As in the first update, this next update would only require $O(1)$ thunk evaluations.

In Sec. 3.4, we show that quickhull (for computing convex hull) has a similar accumulator structure; again, we use names to separate the subproblems, yet still efficiently accumulate their results. In place of "names", earlier work variously used the terms *(allocation) keys* [Acar 2005; Hammer and Acar 2008; Acar and Ley-Wild 2009] and *indices* [Acar et al. 2006a,b], but the core idea is the same.

1.2 Fungi types and effects: Static reasoning for dynamic names

The correct use of names is critical for incremental computing. At a minimum, such names must be precise to ensure that a program is type safe and that the runtime system works correctly, viz., that caching and change propagation work correctly together. Beyond precision, dynamic cache eviction behavior also warrants static approximations and associated verification techniques. In larger systems, as we build incremental algorithms that compose, we seek a way of statically organizing and documenting different cache location name choices made throughout (e.g., see the quick hull examples in Sec. 3.4).

Meanwhile, existing literature lacks static reasoning tools for cache location names (see Sec. 9 for details). As a first step in this direction, Fungi offers a refinement type and effects system, toured in the next two sections, and defined formally in Sec. 5. As our examples demonstrate, these (dependent) refinement types provide a powerful descriptive tool for statically organizing and documenting choices about cache location names.

Contributions:

- We define the *precise name* verification problem for programs that use names to uniquely identify their dynamic data and sub-computations. To formalize this problem and implement our proposed solution, we design Fungi, a core calculus for such programs.
- Fungi programs employ a novel refinement type and effect system to describe and verify their use of explicit allocation names. The design of this type system is our main contribution.
- To refine its types, Fungi employs separate name and index term languages, for statically modeling names and name sets, respectively (Sec. 5). The name-set indices were inspired by set-indexed types in the style of DML [Xi and Pfenning 1999]; we extend this notion with polymorphism over kinds and index functions; the latter was inspired by abstract refinement types [Vazou et al. 2013].
- In Sec. 3, we tour a collections library for sequences and sets. These programs demonstrate the descriptive power of Fungi's types and effects.
- To validate our approach theoretically, we give a definition of *precise effects* for an evaluation derivation $\mathcal{D}$. We prove that for all well-typed Fungi programs, every evaluation derivation $\mathcal{D}$ consists of precise effects (Sec. 7).
- Drawing closer to an implementation, we give a bidirectional version of the type system that we show is sound and complete with respect to our main type assignment system (Sec. D).
- Based on the bidirectional formulation of the type system, we implement a closely related prototype of Fungi in Rust, as a deeply-embedded DSL (Sec. 8).

2 OVERVIEW OF OUR APPROACH

Fungi's type system statically approximates the *name set* of each first-class name, and the *write set* of each sub-computation. By reasoning about these sets statically, we rule out programs that contain dynamic type errors, enforce precise names where desired, and provide descriptive affordances to the programmer, so that she can specify her program's nominal effects to others (e.g., in the types of a module signature).

Without a suitable type system, programmer-chosen allocation names can quickly give rise to imperative state, and unfortunately, to the potential for unintended errors that undermine the type safety of a program.

Re-allocation at *different* types:

```
let r1 = ref(n, (0,1))
let r2 = ref(n, 2)
```

Re-allocation at the *same* type:

```
let r1 = ref(n, (0,1))
let r2 = ref(n, (2,2))
```

What if a common name `n` is used simultaneously for two reference cells of two different types? Should a program that re-allocates a name at different types be meaningful? (e.g., see the first program to left, above). We say, "no", such programs are not meaningful: They use a name `n` imprecisely at two *different* types, undermining the basic principle of type safety.

On the other hand, suppose a name `n` is re-allocated at the *same* type (e.g., if `r1` and `r2` had the same type, as in the second program), we may consider this behavior meaningful, though *imperative*. In this case, we want to know that the program is imperative, and that we should not cache its behavior in contexts where we expect precise names.

Name sets and write sets. Consider the following typing rules, which approximate Fungi's full type system, and illustrate its basic design principles:

$$\frac{\Gamma \vdash v_n : \mathsf{Nm}\,[X] \qquad \Gamma \vdash v : A}{\Gamma \vdash \mathsf{ref}(v_n, v) : \mathsf{Ref}(A) \rhd X} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : A \rhd X \\ \Gamma, x : A \vdash e_2 : B \rhd Y \\ \Gamma \vdash (X \perp Y) \equiv Z : \mathbf{NmSet}\end{array}}{\Gamma \vdash \mathtt{let}(e_1, x.e_2) : B \rhd Z}$$

The rules conclude with the judgement form $\Gamma \vdash e : A \rhd X$, where the set X approximates the set of written names (it may contain more names than those written, but must contain every name that is written).

The first rule types a reference cell allocation named by programmer-chosen value v_n, whose type $\mathsf{Nm}\,[X]$ is *indexed by* an approximate name set X from which name value v_n is drawn; in the rule's conclusion, this name set X serves as the allocation's *write set*.

The second rule gives `let` sequencing: a premise judges the equivalence of name set Z (at sort **NmSet**) with sets X and Y, which index the write sets of e_1 and e_2, respectively. For names to be precise, the written sets X and Y must be disjoint, which we notate as $X \perp Y$. This final premise is capturing the constraint that Fungi programs be pure: it is not derivable when $X \not\perp Y$. In particular, this premise is not derivable for either of the two program fragments shown above, where in each, the name `n` is used in distinct allocations.

Name functions, higher-order apartness, write scope. In most cases, the source of names for a program is its input data structure(s), not some fixed set of predefined name variables, as in the pair of two-line programs above. Further, names are *not* linear resources in most incremental programs. In fact, it is common for precise programs to consume a name more than once (e.g., see listings for quickhull in Sec. 3.4). To disambiguate multiple writes of the same name, implementations commonly employ *distinct memo tables*, one for each unique function. In this paper, we develop a more general notion for Fungi programs: *name functions*.

In particular, we define a higher-order account of *separation*, or *apartness* (the term we use throughout) to compose precise programs. Just as apart names $n \perp m$ do not overwrite each other (when written, the two singleton write sets are disjoint), *apart* name functions $f \perp g$ give rise to *name spaces* that do not overwrite each other, i.e., $f \perp g \iff \forall x. f\,x \perp g\,x$. That is, $f \perp g$ when the images of f and g are always disjoint name sets, for any preimage.

To streamline programs for common composition patterns, Fungi's type system employs a special, ambient name space, the *write scope*. As illustrated in Sec. 3, name functions and write scopes naturally permit name-precise sub-computations to compose into larger (name-precise) computations: Just as functions provide the unit of composition for functional programming, (apart) name functions provide the unit of composition for (precise) naming strategies.

3 EXAMPLES

In this section, we demonstrate larger example programs and their types, focusing on a collections library of sequences and sets.

3.1 Examples of datatype definitions: Sequences and sets

We present nominal datatype definitions for sequences and finite maps, as first presented in Hammer et al. [2015], but without types to enforce precision.

We represent sequences as *level trees*. Unlike lists, level trees permit efficient random editing [Hammer et al. 2015; Headley and Hammer 2016], and their balanced binary structure is useful for organizing efficient incremental algorithms [Pugh and Teitelbaum 1989]. We represent sets and finite maps as binary hash tries, which we build from these level trees.

Level tree background: A level tree is a binary tree whose internal binary nodes are each labeled with a level, and whose leaves consist of sequence elements. A level tree *well-formed* when every path from the root to a leaf passes through a sequence of levels that decrease monotonically. Fortunately, it is simple to pseudo-randomly generate random levels that lead to probabilistically balanced level trees [Pugh and Teitelbaum 1989]. Because they are balanced, level trees permit efficient persistent edits (insertions, removals) in logarithmic time, and zipper-based edits can be even be more efficient [Headley and Hammer 2016]. Further, level trees assign sequences of elements (with interposed levels) a single canonical tree that is *history independent*, making cache matches more likely.

For nominal level trees, we use names to identify allocations, permitting us to express incremental insertions or removals in sequences as $\mathsf{O}(1)$ re-allocations (store mutations). By contrast, without names, each insertion or removal from the input sequence generally requires $\mathsf{O}(\log(\mathsf{N}))$ fresh pointer allocations, which generally cascade during change propagation, often giving rise to larger changes, as in Sec. 1.1's `rev` example.

Sequences as nominal level trees. Below, type $\mathsf{Seq}\,[\mathsf{X}]$ refines the "ordinary type" for sequences Seq; it classifies sequences whose names are drawn from the set X, and has two constructors for binary nodes and leaves:

$$\begin{aligned}
\texttt{SeqBin} &: \forall \mathsf{X}\perp\mathsf{Y}\perp\mathsf{Z} : \textbf{NmSet}.\mathsf{Nm}\,[\mathsf{X}] \to \mathsf{Lev} \to \mathsf{Ref}(\mathsf{Seq}\,[\mathsf{Y}]\,) \to \mathsf{Ref}(\mathsf{Seq}\,[\mathsf{Z}]\,) \to \mathsf{Seq}\,[\mathsf{X}\perp\mathsf{Y}\perp\mathsf{Z}] \\
\texttt{SeqLf} &: \forall \mathsf{X} : \textbf{NmSet}. \qquad \mathsf{Vec} \to \mathsf{Seq}\,[\mathsf{X}]
\end{aligned}$$

For simplicity here, we focus on monomorphic sequences at base type (a vector type Vec for vectors of natural numbers); in the appendix, we show refinement types for polymorphic sequences whose type parameters may be higher-kinded (parameterized by names, name sets, and name functions).

In the type of constructor `SeqBin`, the notation $\mathsf{X}\perp\mathsf{Y}\perp\mathsf{Z}$ denotes a set of names, and it asserts *apartness* (pair-wise disjointness) among the three sets. Binary nodes store a natural number *level*, classified by type Lev, a name, and two incremental pointers to left and right sub-trees, of types $\mathsf{Ref}(\mathsf{Seq}\,[\mathsf{Y}]\,)$ and $\mathsf{Ref}(\mathsf{Seq}\,[\mathsf{Z}]\,)$, respectively. The type $\mathsf{Nm}\,[\mathsf{X}]$ classifies first-class names drawn from the set X. In this case, the apartness $\mathsf{X} \perp \mathsf{Y} \perp \mathsf{Z}$ means that the name from X at the `SeqBin` node is distinct from those in its subtrees, whose names are distinct from one another (name sets Y vs Z). Generally, computations that consume a data structure assume that names are non-repeating within a sequence (as enforced by this type); however, computations conventionally reuse input names to identify corresponding constructors in the output, as in the `filter`, `trie` and `quickhull` examples below.

In the `SeqLf` case, we store short vectors of elements to capitalize on low-level operations that exploit cache coherency. (In empirical experiments, we often choose vectors with one thousand elements). Notably, this leaf constructor permits any name set X; we find it useful when types over-approximate their names, e.g., as used in the type for `filter` below, which generally filters away some input names, but does not reflect this fact in its output type.

Sets as nominal hash tries. In addition to sequences, we use balanced trees to represent sets and finite maps. A *nominal hash trie* uses the hash of an element to determine a path in a binary tree, whose pointers are named. The type for $\mathsf{Set}\,[\mathsf{X}]$ is nearly the same as that of $\mathsf{Seq}\,[\mathsf{X}]$, above; we

```
max  :  ∀X : NmSet.
          Seq[X] → Nat
        ▷ (λx : Nm. {x·1} ⊥ {x·2}) [X]

max seq = match seq with
| SeqLf(vec) ⇒ vec_max vec
| SeqBin(n,_,l,r) ⇒
  let (_,ml) = memo[n·1](max !l)
  let (_,mr) = memo[n·2](max !r)
  if ml > mr then ml else mr

vec_max    : Vec → Nat
vec_filter : Vec → (Nat → Bool) → Vec
```

```
filter  :  ∀X : NmSet.
             Seq[X] → (Nat → Bool) → Seq[X]
           ▷ (λx : Nm. {x·1} ⊥ {x·2}) [X]

filter seq pred = match seq with
| SeqLf(vec) ⇒ SeqLf(vec_filter vec pred)
| SeqBin(n, lev, l, r) ⇒
  let (rl,sl) = memo[n·1](filter !l pred)
  let (rr,sr) = memo[n·2](filter !r pred)
  match (is_empty sl, is_empty sr) with
  | (false,false) ⇒ SeqBin(n, lev, rl, rr)
  | (_,true) ⇒ sl
  | (true,_) ⇒ sr
```

Fig. 2. Recursive functions over sequences: Finding the max element, and filtering elements by a predicate.

just omit the level at the binary nodes:

$$\begin{array}{lcll} \texttt{SetBin} & : & \forall X{\perp}Y{\perp}Z : \textbf{NmSet}. & \mathsf{Nm}[X] \rightarrow \mathsf{Ref}(\mathsf{Set}[Y]) \rightarrow \mathsf{Ref}(\mathsf{Set}[Z]) \rightarrow \mathsf{Set}[X{\perp}Y{\perp}Z] \\ \texttt{SetLf} & : & \forall X : \textbf{NmSet}. & \mathsf{Vec} \rightarrow \mathsf{Set}[X] \end{array}$$

The natural number vectors at the leaves at meant to represent “small” sets of natural numbers. (In this section, we focused on giving types for sequences and sets of natural numbers; Sec. A of the supplement describes polymorphic type definitions).

A collections library. Sec. 3.2 gives simple structurally recursive algorithms over level trees that we use as subroutines for later implementing quickhull. In Sec. 3.3, we give an incrementally efficient conversion algorithm `trie` for building binary hash tries from level trees; it uses nested structural recursion to convert a binary tree of type Seq[X] into one of type Set[X]. Sec. 3.4 gives two variants of the divide-and-conquer quickhull algorithm, which only differ in their naming strategy. In the context of these examples, we illustrate the key patterns for typing the collections of Hammer et al. [2015], and relate these patterns to the relevant typing rules of Fungi’s type system.

3.2 Examples of structural recursion: Reducing and filtering sequences

Fig. 2 gives the type and code for `max` and `filter`, which compute the maximum element of the sequence, and filter the sequence by a predicate, respectively. Both algorithms use structural recursion over the Seq[X] datatype, defined above.

The computation of `max` is somewhat simpler. In the leaf case (`SeqLf`), `max` uses the auxiliary function `vec_max`. In the binary case (`SeqBin`), `max` performs two memoized recursive calls, for its left and right sub-trees. We access these reference cells using `!`, as in SML/OCaml notation.

In Fungi’s core calculus, named thunks are the unit of function caching. To use them, we introduce syntactic sugar for memoization, where `memo[n·1](e)` expands to code that allocates a named thunk (here, named `n·1`) and demands its output value: `let x = thunk(n·1,e) in forceref(x)`. The primitive `forceref` forces a thunk, and returns a pair consisting of a reference cell representation of the forced thunk, and its cached value; the reference cell representation of the thunk is useful in `filter`, and later examples, but is ignored in `max`. The code listing for `max` uses two instances of `memo`, with two *distinct* names, each based on the name of the binary node `n`.

A formal structure for names. Fungi’s core calculus defines names as binary trees, $n ::= \mathsf{leaf} \mid \langle\langle n, n \rangle\rangle$. In practice, we often want to build names from primitive data, such as string constants, natural numbers and bit strings; without loss of generality, we assume embeddings of these types as binary trees. For instance, we let 1 be shorthand for ⟨⟨leaf, leaf⟩⟩, the name with one binary node; similarly, 2 stands for ⟨⟨leaf, ⟨⟨leaf, leaf⟩⟩⟩⟩ and 0 for leaf. In the code listing, we use n·1 as shorthand for ⟨⟨n, 1⟩⟩. More generally, we use n·m as a right-associative shorthand for name lists of binary nodes, where 1·2·3 represents the name ⟨⟨1, ⟨⟨2, 3⟩⟩⟩⟩.

The following deductive reasoning rules help formalize notions of name equivalence and apartness for name terms:

$$\frac{\Gamma \vdash M_1 \equiv N_1 : \mathbf{Nm} \quad \Gamma \vdash M_2 \equiv N_2 : \mathbf{Nm}}{\Gamma \vdash \langle\langle M_1, M_2\rangle\rangle \equiv \langle\langle N_1, N_2\rangle\rangle : \mathbf{Nm}} \qquad \frac{\Gamma \vdash M_1 \perp N_1 : \mathbf{Nm}}{\Gamma \vdash \langle\langle M_1, M_2\rangle\rangle \perp \langle\langle N_1, N_2\rangle\rangle : \mathbf{Nm}} \qquad \frac{\Gamma \vdash M_1 \equiv N : \mathbf{Nm}}{\Gamma \vdash \langle\langle M_1, M_2\rangle\rangle \perp N : \mathbf{Nm}}$$

We use the context Γ to store assumptions about equivalence and apartness (e.g., from type signatures) and judge pairs of name terms at a common sort **Nm**. The first rule says that two binary names are equivalent if their left and right sub-names are equivalent. The second rule says that two binary names are apart (distinct) if their left names are apart. The third rule says that a binary name is always distinct from a name equivalent to its left sub-name. In the supplement (Sec. F and Sec. G), we give more rules for deductive apartness and equivalence of name terms and index terms; here and below, we give a few selected examples.

We turn to the type of max given in the listing. For all sets of names X, the function finds the largest natural number in a sequence (index by X). The associated effect (after the ▷) consists of the following write set:

$$\big(\lambda x : \mathbf{Nm}.\{x{\cdot}1\} \perp \{x{\cdot}2\}\big)[X] \quad \equiv \quad \big((\lambda x : \mathbf{Nm}.\{x{\cdot}1\})[X]\big) \perp \big((\lambda x : \mathbf{Nm}.\{x{\cdot}2\})[X]\big)$$

Specifically, this is an index term that consists of mapping a function over a name set X, and taking the (disjoint) union of these results. Intuitively (but informally), this term denotes the following set comprehension: $\bigcup_{x \in X}\{x{\cdot}1, x{\cdot}2\}$. That is, the set of all names x in X, appended with either one of name constants 1 and 2. More generally, the index form $f[X]$ employs a name set mapping function f of sort $\mathbf{Nm} \overset{\text{idx}}{\to} \mathbf{NmSet}$, where index X is a name set and has sort **NmSet**. Here, $f :\equiv \lambda x.\{x{\cdot}1, x{\cdot}2\}$. In some listings, we use $X \cdot Y$ as shorthand for $\big(\lambda y.\,(\lambda x.\{x{\cdot}y\})[X]\big)[Y]$, that is, the set of all pairwise name compositions from name sets X and Y.

The following table of rules give equivalence and apartness reasoning (left vs. right columns) for index function abstraction and nameset-mapping (top vs. bottom rows); by convention, we use i and j for general indices that may not be name sets, and use X, Y and Z for name sets.

$$\frac{\Gamma, (a \equiv b : \gamma_1) \vdash i \equiv j : \gamma_2}{\Gamma \vdash \lambda a.\, i \equiv \lambda b.\, j : \gamma_1 \overset{\text{idx}}{\to} \gamma_2} \qquad \frac{\Gamma, (a \equiv b : \gamma_1) \vdash i \perp j : \gamma_2}{\Gamma \vdash \lambda a.\, i \perp \lambda b.\, j : \gamma_1 \overset{\text{idx}}{\to} \gamma_2}$$

$$\frac{\Gamma \vdash i \equiv j : \mathbf{Nm} \overset{\text{idx}}{\to} \mathbf{NmSet} \quad \Gamma \vdash X \equiv Y : \mathbf{NmSet}}{\Gamma \vdash i[X] \equiv j[Y] : \mathbf{NmSet}} \qquad \frac{\Gamma \vdash i \perp j : \mathbf{Nm} \overset{\text{idx}}{\to} \mathbf{NmSet} \quad \Gamma \vdash X \equiv Y : \mathbf{NmSet}}{\Gamma \vdash i[X] \perp j[Y] : \mathbf{NmSet}}$$

To be equivalent, function bodies must be equivalent under the assumption that their arguments are equivalent; to be apart, their bodies must be apart. To be equivalent, a set mapping must have equivalent mapping functions and set arguments; to be apart, the functions must be apart, with the same argument. These set-mapping rules are specialized to set comprehensions, using a function from names to name sets, and a starting name set. In the full set of rules, we give an analogous pair of application rules reason about ordinary function application, with arbitrary input and output sorts (not shown here).

$$\texttt{trie} : \forall X : \mathbf{NmSet}.\ \mathsf{Seq}[X] \rightarrow \mathsf{Set}[\widehat{\mathsf{join}}(X)] \ \rhd\ (\lambda x : \mathbf{Nm}. \{x{\cdot}1\} \perp \{x{\cdot}2\})[X] \perp (\lambda x{:}\mathbf{Nm}. (\lambda y{:}\mathbf{Nm}. \{x{\cdot}y\})[\widehat{\mathsf{join}}(X)])[X]$$

```
trie seq = match seq with
| SeqLf(vec) ⇒ trie_lf vec
| SeqBin(n,_,l,r) ⇒
 let (tl,_) = memo[n·1](trie !l)
 let (tr,_) = memo[n·2](trie !r)
 let trie = ws[n](join n tl tr)
 trie
```

where:

$$\widehat{\mathsf{join}}(X) :\equiv (\lambda x{:}\mathbf{Nm}. \{x{\cdot}1\} \perp \{x{\cdot}2\})^{*}[X]$$

$$\texttt{join} : \forall X \perp Y \perp Z : \mathbf{NmSet}.\ \mathbf{Nm}[X] \rightarrow \mathsf{Set}[Y] \rightarrow \mathsf{Set}[Z] \rightarrow \mathsf{Set}[\widehat{\mathsf{join}}(X \perp Y \perp Z)] \ \rhd\ \widehat{\mathsf{join}}(X \perp Y \perp Z)$$

```
join n l r = match (l,r) with
| SetLf(l), SetLf(r) ⇒ join_vecs n l r
| SetBin(_,_,_), SetLf(r) ⇒
                join n·1 l (split_vec n·2 r)
| SetLf(l), SetBin(_,_,_) ⇒
                join n·1 (split_vec n·2 l) r
| SetBin(ln,l0,l1), SetBin(rn,r0,r1) ⇒
  let (_,j0) = memo[ln·1](join ln·2 l0 r0)
  let (_,j1) = memo[rn·1](join rn·2 l1 r1)
  SetBin(n, j0, j1)
```

and where:

$$\texttt{join\_vecs} : \forall X : \mathbf{NmSet}.\ \mathbf{Nm}[X] \rightarrow \mathsf{Vec} \rightarrow \mathsf{Vec} \rightarrow \mathsf{Set}[X] \ \rhd\ (\lambda x : \mathbf{Nm}. \{x{\cdot}1\} \perp \{x{\cdot}2\})[X]$$

$$\texttt{split\_vec} : \forall X : \mathbf{NmSet}.\ \mathbf{Nm}[X] \rightarrow \mathsf{Vec} \rightarrow \mathsf{Set}[X] \ \rhd\ (\lambda x : \mathbf{Nm}. \{x{\cdot}1\} \perp \{x{\cdot}2\})[X]$$

Fig. 3. Nested structural recursion: Converting sequences into maps

Allocating structurally recursive output. The type and listing for `filter` is similar to that of `max`. The key difference is that `filter` builds an output data structure with named reference cells; it returns a filtered sequence, also of type Seq [X]. Meanwhile, its write set is the same as that of `max`.

Rather than allocate redundant cells (which we could, but do not), each reference cell of filtered output arises from the `memo` shorthand introduced above. In particular, when the left and right filtered sub-trees are non-empty, `filter` introduces a `SeqBin` code, with two named pointers, `rl` and `rr`, the left and right references introduced by memoizing function calls' results. This "trick" works because the output is built via structural recursion, via memoized calls that are themselves structurally recursive over the input. Below, we nest structural recursions.

3.3 Example of nested structural recursion: Sequences into maps

Fig. 3 defines functions `trie` and `join`. The function `trie` uses structural recursion, following a pattern similar to `max` and `filter`, above. However, compared to `max` and `filter`, the body of the `SeqBin` case for `trie` is more involved: rather than compare numbers, or create a single datatype constructor, it invokes `join` on the recursive results of calling `trie` on the left and right sub-sequences. When a `trie` represents a set, the function `join` computes the set union, following an algorithm inspired by Pugh and Teitelbaum [1989], but augmented with names. Given two tries with apart name sets X and Y, `join` constructs a trie with names $\widehat{\mathsf{join}}(X \perp Y)$, where $\widehat{\mathsf{join}}$ is a function over name sets, whose definition we discuss below.

The ambient write scope: a distinguished name space. To disambiguate the names allocated by the calls to `join`, the code for `trie` uses a *name space* defined by `n`, with the notation `let trie = ws[n](`e`)`. In particular, this notation controls the ambient write scope, the name space for allocations that write the store. Informally, it says to perform the writes of the sub-computation e by first prepending the name `n` to each allocation name.

To type programs that use the ambient write scope, we use a typing judgement form $\Gamma \vdash^{N} e : A \rhd X$ that tracks a name space N, a name term of sort $\mathbf{Nm} \xrightarrow{\mathsf{Nm}} \mathbf{Nm}$. (This arrow sort over names is more restricted than $\mathbf{Nm} \xrightarrow{\mathsf{idx}} \mathbf{NmSet}$: it only involves name construction, and lacks set structure.) Consider the following typing rules for write scopes (left rule) and name function application (right rule):

$$\frac{\Gamma \vdash v_1 : \mathsf{Nm}[X] \qquad \Gamma \vdash v_2 : A}{\Gamma \vdash^{M} \mathtt{ref}(v_1, v_2) : \mathsf{Ref}[M[X]]\, A \rhd M[X]} \qquad \frac{\Gamma \vdash v : (\mathbf{Nm} \xrightarrow{\mathsf{idx}} \mathbf{Nm})[N'] \quad \Gamma \vdash^{N \circ N'} e : A \rhd W}{\Gamma \vdash^{N} \mathsf{scope}(v, e) : A \rhd W}$$

The left rule uses the name term M to determine the final allocation name for a reference cell. As indicated by the conclusion's type and write effect, it computes a name in set $M[X]$, where index term X gives the name set of the argument. The thunk rule (not shown here) is similar: it uses the ambient write scope to determine the allocated name.

The right rule extends the ambient write scope N by name function composition with N', for the duration of evaluating a given subterm e. The notation in the code listing `let x = ws[n](e)`e_2 is shorthand for $\mathsf{let}(\mathsf{scope}(\lambda a.\, n{\cdot}a, e), x.e_2)$. That is, we prepend name n to each name written by the evaluation of subterm e; then, we evaluate e_2. Fungi's collections library uses this shorthand to disambiguate names that are reused across sequenced calls, as in the listings for `trie`, and in quickhull, below.

Types for recursive name sets. Sometimes we need to specify the names of a set recursively, as in the code listing for `join`, which is parameterized by an argument name `n`. In some recursive calls to `join`, this name argument "grows" into `n·1` or `n·2`. In general, each name may grow a variable number of times. This generality is required: The algorithm for `trie` allocates a final binary hash trie whose structure is generally *not* isomorphic to the initial level tree; it is generally *larger*.

To capture these recursive name-growth patterns, we introduce another special variant of index terms, for describing sets that we "grow" inductively by applying a function zero or more times to an initial set. We use this index form to define $\widehat{\mathsf{join}}$ from Fig. 3:

$$\frac{\Gamma \vdash i \equiv j : \mathbf{Nm} \xrightarrow{\mathsf{idx}} \mathbf{NmSet} \quad \Gamma \vdash X \equiv Y : \mathbf{NmSet}}{\Gamma \vdash i^{*}[X] \equiv j^{*}[Y] : \mathbf{NmSet}} \qquad \frac{\Gamma \vdash i \equiv j : \mathbf{Nm} \xrightarrow{\mathsf{idx}} \mathbf{NmSet} \quad \Gamma \vdash X \equiv Y : \mathbf{NmSet}}{\Gamma \vdash i^{*}[X] \equiv i^{*}[j[Y]] \cup Y : \mathbf{NmSet}} \qquad \frac{\Gamma \vdash i \perp j : \mathbf{Nm} \xrightarrow{\mathsf{idx}} \mathbf{NmSet} \quad \Gamma \vdash X \perp Y : \mathbf{NmSet}}{\Gamma \vdash i^{*}[X] \perp j^{*}[Y] : \mathbf{NmSet}}$$

The first rule matches that of the non-recursive form. The second rule is unique to recursive applications; it witnesses one unfolding of the recursion. The final rule says that two uses of recursive application are apart when the starting sets and functions are apart.

3.4 Examples of naming divide-and-conquer: Quickhull

Fig. 4 defines two quickhull functions `qh1` and `qh2`. They consist of the same geometric point-processing steps, but use distinct naming strategies, and consequently, lead to distinct incremental cache behavior. Each version computes the furthest point `p` from the current line `ln` by using `max_pt`, which is similar to `max` in Fig. 2. This point `p` defines two additional lines, `(ln.0,p)` and `(p,ln.1)`, which quickhull uses to filter the input points before recurring on these new lines.

We use the typing ingredients introduced above to give two naming strategies, each with distinct cache behavior. To summarize the naming strategies of `max_pt` and `filter` (each uses structural recursion over level trees), we introduce the shorthand, with associated the equivalences:

$$\widehat{\mathsf{bin}}[X] \;:\equiv\; X \cdot \{1,2\} \;:\equiv\; (\lambda x.\{x{\cdot}1, x{\cdot}2\})[X] \equiv \big((\lambda x.\{x{\cdot}1\}[X]) \perp (\lambda x.\{x{\cdot}2\}[X])\big)$$

```
qh1 :
```

$$\forall X \perp Y \perp Z : \mathbf{NmSet}.\ \mathsf{Line}[X] \rightarrow \mathsf{Seq}[Y] \rightarrow \mathsf{List}[Z] \rightarrow \mathsf{List}[X \perp Z] \\ \triangleright\ \widehat{\mathsf{lines}}(X,Y){\cdot}\{1,2,3\} \\ \perp \widehat{\mathsf{lines}}(X,Y){\cdot}\{1,2,3\}{\cdot}Y{\cdot}\{1,2\}$$

```
qh1 ln pts h =
 let p = ws[ln·1](max_pt ln pts)
 let l = ws[ln·2](filter (ln.0,p) pts)
 let r = ws[ln·3](filter (p,ln.1) pts)
 let h = memo[ln·1](qh1 (p,ln.1) r h)
 let h = Cons(ln,p,ref(ln·2,h))
 let h = memo[ln·3](qh1 (ln.0,p) l h)
 h
```

$\widehat{\mathsf{lines}}(X,Y) :\equiv$ "All lines formed by lines X and points Y"
(definitions provided by the geometry library)

```
qh2 :
```

$$\forall X \perp Y \perp Z : \mathbf{NmSet}.\ \mathsf{Line}[X] \rightarrow \mathsf{Seq}[Y] \rightarrow \mathsf{List}[Z] \rightarrow \mathsf{List}[X \perp Z] \\ \triangleright\ (\lambda y{:}\mathbf{Nm}.\{1{\cdot}y\} \perp \{2{\cdot}y\})^{*}[\widehat{\mathsf{qh}_2}(Y \perp Z)]$$

```
qh2 ln pts h =
 let p = ws[3·1](max_pt ln pts)
 let l = ws[3·2](filter (ln.0,p) pts)
 let r = ws[3·3](filter (p,ln.1) pts)
 let h = memo[1](ws[1](qh2 (p,ln.1) r h))
 let h = Cons(ln,p,ref(2,h))
 let h = memo[3](ws[2](qh2 (ln.0,p) l h))
 h
```

$$\widehat{\mathsf{qh}_2}(X) :\equiv \{3{\cdot}1, 3{\cdot}2, 3{\cdot}3\}{\cdot}\widehat{\mathsf{bin}}[X] \\ \perp \{1,2,3\}$$

Fig. 4. Two naming strategies for quickhull, same algorithmic structure

The left naming strategy of qh1 uses the identity of the *partition lines* to introduce three name spaces (for the calls to max_pt and filter), and three allocations, for the two calls to qh1, and the output hull point p that is computed by max_pt. In this case, the library for computational geometry provides the lines' names, where we exploit that quickhull does not process the same line twice. We elide some details in this reasoning, for brevity in the listing.

The right version (qh2) ignores the names of lines. Instead, it names everything based on a binary path in the quickhull call graph: The two recursive calls to qh2 each extend write scope, first with name 1, then with 2.

Ignoring their use of names, the two versions of quickhull are identical: When run from scratch, they always produce the same output, from the same steps, in the same order.

Incrementally, they are very different: they have distinct naming strategies, and by virtue of this fact, they have distinct cache behavior. In brief, the first strategy is better at changes that disrupt and then restore the hull (inserting and then removing a new hull point) but this line-based naming strategy also lacks a simple cache *eviction* strategy (when to "forget" about work for a line? What if we forget about a line, but it reappears later?). In a way, by using *lines* as names, the left version can suffer a cache consumption problem similar to the structurally-named variant of rev from Sec. 1.1: each new line will generally consume additional cache space.

In contrast, by overwriting earlier computations based on call graph position, the second quickhull naming strategy is cruder, but gives a natural cache eviction strategy: When the same callgraph path is reached, this program's names will overwrite its old callgraph content with new callgraph content. This strategy corresponds (roughly) with naming strategies that use a "global position" in the call graph to determine cache locations (see [Burckhardt et al. 2011; Çiçek et al. 2015, 2016], and Sec. 9 for more discussion).

4 PROGRAM SYNTAX

The examples from the prior section use an informally defined variant of ML, enriched with a (slightly simplified) variant of our proposed type system. In this section and the next, we focus on a core calculus for programs and types, and on making these definitions precise.

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

Fig. 5. Syntax of expressions

4.1 Values and 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. First, 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. Second, since we make the unit of memoization a thunk, and CBPV makes explicit the creation of thunks and closures, it exposes exactly the structure that we extend to 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, which is the name of the reference or thunk, *not* the contents of the reference or thunk.

The syntax described above 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 and other simple transformations on names. The construct $\mathtt{scope}(v, e)$ construct controls monadic state for the current name function, composing it with a name function v within the dynamic extent of its subexpression e. Name function application $M\ v$ 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.

4.2 Names

Figure 6 shows the syntax of literal names, name terms, name term values, and name term sorts. Literal names m, n are simply binary trees: either an empty leaf leaf or a branch node $\langle\langle n_1, n_2\rangle\rangle$. Name terms M, N consist of literal names n and branch nodes $\langle\langle M_1, M_2\rangle\rangle$, abstraction $\lambda a.\, M$ and application $M(N)$.

Name terms are classified by sorts γ: sort $\mathbf{Nm}$ for names n, and $\gamma \overset{\mathsf{Nm}}{\to} \gamma$ for (name term) functions.

Names (binary trees)	m, n	$::=$	$\mathtt{leaf}$	leaf name
			$\mid \langle\langle n_1, n_2 \rangle\rangle$	binary name composition
Name terms (STLC+names)	M, N	$::=$	$n \mid \langle\langle M_1, M_2 \rangle\rangle$	literal names, binary name composition
			$\mid a \mid \lambda a.\, M \mid M(N)$	variable, abstraction, application
Name term values	V	$::=$	$n \mid \lambda a.\, M$	
Name term sorts	γ	$::=$	$\mathbf{Nm}$	name; inhabitants n
			$\mid \gamma \overset{\mathbf{Nm}}{\Rightarrow} \gamma$	name term function; inhabitants $\lambda a.\, M$
Typing contexts	Γ	$::=$	$\cdot \mid \Gamma, a : \gamma \mid \cdots$	full definition in Figure 10

Fig. 6. Syntax of name terms: a λ-calculus over names, as binary trees

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

$$\frac{}{\Gamma \vdash n : \mathbf{Nm}}\text{M-const} \qquad \frac{(a : \gamma) \in \Gamma}{\Gamma \vdash a : \gamma}\text{M-var} \qquad \frac{\Gamma \vdash M_1 : \mathbf{Nm} \quad \Gamma \vdash M_2 : \mathbf{Nm}}{\Gamma \vdash \langle\langle M_1, M_2 \rangle\rangle : \mathbf{Nm}}\text{M-bin}$$

$$\frac{\Gamma, a : \gamma' \vdash M : \gamma}{\Gamma \vdash (\lambda a.\, M) : (\gamma' \overset{\mathbf{Nm}}{\Rightarrow} \gamma)}\text{M-abs} \qquad \frac{\Gamma \vdash M : (\gamma' \overset{\mathbf{Nm}}{\Rightarrow} \gamma) \qquad \Gamma \vdash N : \gamma'}{\Gamma \vdash M(N) : \gamma}\text{M-app}$$

Fig. 7. Sorting rules for name terms M

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

$$\frac{}{V \Downarrow_M V}\text{teval-value} \qquad \frac{M_1 \Downarrow_M n_1 \qquad M_2 \Downarrow_M n_2}{\langle\langle M_1, M_2 \rangle\rangle \Downarrow_M \langle\langle n_1, n_2 \rangle\rangle}\text{teval-bin} \qquad \frac{M \Downarrow_M \lambda a.\, M' \quad N \Downarrow_M V \quad [V/a]M' \Downarrow_M V'}{M(N) \Downarrow_M V'}\text{teval-app}$$

Fig. 8. Evaluation rules for name terms

The rules for name sorting $\Gamma \vdash M : \gamma$ are straightforward (Figure 7), as are the rules for name term evaluation $M \Downarrow_M V$ (Figure 8). We write $M =_\beta M'$ when name terms M and M' are convertible, that is, applying any series of β-reductions and/or β-expansions changes one term into the other.

5 TYPE SYSTEM

The structure of our type system is inspired by Dependent ML [Xi and Pfenning 1999; Xi 2007]. Unlike full dependent typing, DML 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.

Indices in DML 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 can indirectly reflect properties of run-time values. The simplest case is

Index exprs.	$i, j,$ $X, Y, Z,$ R, W	$::= \alpha$	index variable
		$\mid \{N\}$	singleton name set
		$\mid \emptyset \mid X \perp Y$	empty set, separating union
		$\mid X \cup Y$	union (not necessarily disjoint)
		$\mid () \mid (i, i) \mid \mathtt{prj}_1 i \mid \mathtt{prj}_2 i$	unit, pairing, and projection
		$\mid \lambda a.\, i \mid i(j)$	function abstraction and application
		$\mid M\,[i] \mid i[j] \mid i^*\,[j]$	name set mapping and set building
Index sorts	γ	$::= \cdots \mid \mathbf{NmSet}$	name set sort
		$\mid \mathbf{1}$	unit index sort; inhabitant $()$
		$\mid \gamma * \gamma$	product index sort; inhabitants (i, j)
		$\mid \gamma_1 \overset{\text{idx}}{\to} \gamma_2$	index functions over name sets

Fig. 9. Syntax of indices, name set sort

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 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:

$$\forall a : \mathbf{Nm}.\ \forall X : \mathbf{NmSet}.\ (a \in X) \supset \big(A\,[a] \to B\,[f(a)]\big)$$

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 removes half the quantifiers and obviates the $\in$-constraint attached via $\supset$: $\forall X : \mathbf{NmSet}.\ A\,[X] \to B\,[f(X)]$. 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: $f(X) = \{f(N) \mid N \in X\}$.

(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

$$\forall Y : \mathbf{NmSet}.\ \forall X : \mathbf{NmSet}.\ (Y \subseteq X) \supset \big(A\,[Y] \to B\,[f(Y)]\big)$$

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

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.

5.1 Index Level

Figure 9 gives the syntax of index expressions and index sorts (which classify indices). We use several meta-variables for index expressions; by convention, we use X, Y, Z, R and W only for sets of names—index expressions of sort **NmSet**.

Kinds	K	$::=$	type	kind of value types
			$\mid \mathsf{type} \Rightarrow K$	type argument (binder space)
			$\mid \gamma \Rightarrow K$	index argument (binder space)
Propositions	P	$::=$	$\mathbf{tt} \mid P \;\mathbf{and}\; P$	truth and conjunction
			$\mid i \perp j : \gamma$	index apartness
			$\mid i \equiv j : \gamma$	index equivalence
Effects	ϵ	$::=$	$\langle W; R \rangle$	
Value types	A, B	$::=$	$\alpha \mid d \mid \mathsf{unit}$	type variables, type constructors, unit
			$\mid A + B \mid A \times B$	sum, product
			$\mid \mathsf{Ref}\,[i]\; A$	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 in name set i)
			$\mid (\mathbf{Nm} \overset{\mathsf{Nm}}{\to} \mathbf{Nm})\,[M]$	name function type (singleton)
			$\mid \forall a : \gamma \mid P.\, A$	universal index quantifier
			$\mid \exists a : \gamma \mid P.\, A$	existential index quantifier
Computation types	C, D	$::=$	$\mathbf{F}\, A \mid A \to E$	li**F**t, functions
…with effects	E	$::=$	$C \rhd \epsilon$	effects
			$\mid \forall \alpha : K.\, E$	type polymorphism
			$\mid (\forall a : \gamma \mid P.\, E)$	index polymorphism
Typing contexts	Γ	$::=$	$\cdot$	
			$\mid \Gamma, a : \gamma$	index variable sorting
			$\mid \Gamma, \alpha : K$	type variable kinding
			$\mid \Gamma, d : K$	type constructor kinding
			$\mid \Gamma, N : A$	ref pointer
			$\mid \Gamma, N : E$	thunk pointer
			$\mid \Gamma, x : A$	value variable
			$\mid \Gamma, P$	proposition P holds

Fig. 10. Syntax of kinds, effects, and types

Name sets. If 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, $\emptyset$ for the empty name set, and $X \perp Y$ for a union of two sets X and Y that requires X and Y to be disjoint; this is inspired by the separating conjunction of separation logic [Reynolds 2002]. While disjoint union dominates the types that we believe programmers need, our effects discipline requires non-disjoint union, so we include it ($X \cup Y$) as well.

Variables, pairing, functions. An index i (also written $X, Y, \ldots$ when the index is a set of names) is either an index-level variable a, a name set (described above: $\{N\}$, $X \perp Y$ or $X \cup Y$), the unit index $()$, a pair of indices (i_1, i_2), pair projection $prj_b i$ for $b \in \{1, 2\}$, an abstraction $\lambda a.\, i$, application $i(j)$, or name term application $M\,[i]$.

Name terms M are *not* a syntactic subset of indices i, though name terms can appear inside indices (for example, singleton name sets $\{M\}$). Because name terms are not a syntactic subset of

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

$$\frac{(a : \gamma) \in \Gamma}{\Gamma \vdash a : \gamma}\ \text{sort-var} \qquad \frac{}{\Gamma \vdash () : \mathbf{1}}\ \text{sort-unit} \qquad \frac{\Gamma \vdash i_1 : \gamma_1 \quad \Gamma \vdash i_2 : \gamma_2}{\Gamma \vdash (i_1, i_2) : (\gamma_1 * \gamma_2)}\ \text{sort-pair}$$

$$\frac{\Gamma \vdash i : \gamma_1 * \gamma_2}{\Gamma \vdash \mathsf{prj}_b i : \gamma_b}\ \text{sort-proj} \qquad \frac{}{\Gamma \vdash \emptyset : \mathbf{NmSet}}\ \text{sort-empty} \qquad \frac{\Gamma \vdash N : \mathbf{Nm}}{\Gamma \vdash \{N\} : \mathbf{NmSet}}\ \text{sort-singleton}$$

$$\frac{\Gamma \vdash X : \mathbf{NmSet} \quad \Gamma \vdash Y : \mathbf{NmSet}}{\Gamma \vdash (X \cup Y) : \mathbf{NmSet}}\ \text{sort-union} \qquad \frac{\Gamma \vdash X : \mathbf{NmSet} \quad \Gamma \vdash Y : \mathbf{NmSet} \quad \mathit{extract}(\Gamma) \Vdash X \perp Y : \mathbf{NmSet}}{\Gamma \vdash (X \perp Y) : \mathbf{NmSet}}\ \text{sort-sep-union}$$

$$\frac{\Gamma, a : \gamma_1 \vdash i : \gamma_2}{\Gamma \vdash (\lambda a.\, i) : (\gamma_1 \overset{\text{idx}}{\Rightarrow} \gamma_2)}\ \text{sort-abs} \qquad \frac{\Gamma \vdash i : \gamma_1 \overset{\text{idx}}{\Rightarrow} \gamma_2 \quad \Gamma \vdash j : \gamma_1}{\Gamma \vdash i(j) : \gamma_2}\ \text{sort-apply}$$

$$\frac{\Gamma \vdash M : \mathbf{Nm} \overset{\text{Nm}}{\Rightarrow} \mathbf{Nm} \quad \Gamma \vdash j : \mathbf{NmSet}}{\Gamma \vdash M[j] : \mathbf{NmSet}}\ \text{sort-map} \qquad \frac{\Gamma \vdash i : \mathbf{Nm} \overset{\text{idx}}{\Rightarrow} \mathbf{NmSet} \quad \Gamma \vdash j : \mathbf{NmSet}}{\Gamma \vdash i[j] : \mathbf{NmSet}}\ \text{sort-build} \qquad \frac{\Gamma \vdash i : \mathbf{Nm} \overset{\text{idx}}{\Rightarrow} \mathbf{NmSet} \quad \Gamma \vdash j : \mathbf{NmSet}}{\Gamma \vdash i^*[j] : \mathbf{NmSet}}\ \text{sort-star}$$

Fig. 11. Sorts statically classify name terms M, and the name indices i that index types

indices (and name sets are not name terms), the application form $i(j)$ does not allow us to apply a name term function to a name set. Thus, we also need name term application $M[i]$, which applies the name function M to each element of the name set i. The index-level map form $i[j]$ collects the output sets of function i on the elements of the input set j. The Kleene star variation $i^*[j]$ applies the function i zero or more times to each input element in set j.

Sorts. We use the meta-variable γ to classify indices as well as name terms. We inherit the function space $\overset{\text{Nm}}{\Rightarrow}$ from the name term sorts (Figure 6). The sort **NmSet** (Figure 9) classifies indices that are name sets. The function space $\overset{\text{idx}}{\Rightarrow}$ classifies functions over *indices* (e.g., tuples of name sets), not merely name terms. The unit sort and product sort classify tuples of index expressions.

Most of the sorting rules in Figure 11 are straightforward, but rule 'sort-sep-union' includes a premise $\mathit{extract}(\Gamma) \Vdash X \perp Y : \mathbf{NmSet}$, which says that X and Y are *apart* (disjoint).

Propositions and extraction. Propositions P are conjunctions of atomic propositions $i \equiv j : \gamma$ and $i \perp j : \gamma$, which express equivalence and apartness of indices i and j. For example, $\{n_1\} \perp \{n_2\} : \mathbf{NmSet}$ implies that $n_1 \neq n_2$. Propositions are introduced into Γ via index polymorphism $\forall a : \gamma \mid P.\, E$, discussed below.

The function $\mathit{extract}(\Gamma)$ (Figure 28 in the appendix) looks for propositions P, which become equivalence and apartness assumptions. It also translates Γ into the relational context used in the definition of apartness. We give semantic definitions of equivalence and apartness in the appendix (Definitions G.4 and G.5).

5.2 Kinds

We use a simple system of *kinds* K (Figure 23 in the appendix). Kind type classifies value types, such as unit and $(\mathsf{Thk}[i]\ E)$.

Kind type $\Rightarrow K$ classifies type expressions that are parametrized by a type. Such types are called *type constructors* in some languages.

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

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\,\text{var} \qquad \frac{\Gamma \vdash v : A_1 \qquad \Gamma \vdash A_1 \leq_V A_2}{\Gamma \vdash v : A_2}\,\text{vtype-sub}$$

$$\frac{}{\Gamma \vdash () : \mathsf{unit}}\,\text{unit} \qquad \frac{\Gamma \vdash v_1 : A_1 \qquad \Gamma \vdash v_2 : A_2}{\Gamma \vdash (v_1, v_2) : (A_1 \times A_2)}\,\text{pair} \qquad \frac{\Gamma \vdash v_i : A_i}{\Gamma \vdash \mathbf{inj}_i\, v_i : (A_1 + A_2)}\,\text{inj}$$

$$\frac{\Gamma \vdash n \in X}{\Gamma \vdash (\mathtt{name}\, n) : \mathsf{Nm}\,[X]}\,\text{name} \qquad \frac{\Gamma \vdash M_v : \mathbf{Nm} \overset{\mathsf{Nm}}{\Rightarrow} \mathbf{Nm} \qquad M_v =_\beta M}{\Gamma \vdash (\mathtt{nmfn}\, M_v) : (\mathbf{Nm} \overset{\mathsf{Nm}}{\Rightarrow} \mathbf{Nm})\,[M]}\,\text{namefn}$$

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

$$\frac{\Gamma, a : \gamma, P \vdash v : A}{\Gamma \vdash v : (\forall a : \gamma \mid P.\, A)}\,\text{vtype-}\forall\text{IndexIntro} \qquad \frac{\begin{array}{c} \textit{extract}(\Gamma) \Vdash [i/a]P \\ \Gamma \vdash i : \gamma \qquad \Gamma \vdash v : (\forall a : \gamma \mid P.\, A) \end{array}}{\Gamma \vdash v : [i/a]A}\,\text{vtype-}\forall\text{IndexElim}$$

$$\frac{\begin{array}{c} \textit{extract}(\Gamma) \Vdash [i/a]P \\ \Gamma \vdash i : \gamma \qquad \Gamma \vdash v : [i/a]A \end{array}}{\Gamma \vdash \mathtt{pack}(a.v) : (\exists a : \gamma \mid P.\, A)}\,\text{vtype-}\exists\text{IndexIntro}$$

Fig. 12. Value typing

Kind $\gamma \Rightarrow K$ classifies type expressions parametrized by an index. For example, the Seq type constructor from Section 3 takes a name set, e.g. Seq $[X]$. Therefore, this (simplified variant of) Seq has kind **NmSet** $\Rightarrow$ type. A more general Seq type would also track its pointers (not just its names), and permit any element type, and would thus have kind **NmSet** $\Rightarrow$ $\big(\textbf{NmSet} \Rightarrow (\text{type} \Rightarrow \text{type})\big)$.

5.3 Effects

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

Effect sequencing (Figure 13) is a (meta-level) partial function over a pair of effects: the judgment $\Gamma \vdash \epsilon_1$ then $\epsilon_2 = \epsilon$, means that 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.

Effect coalescing, written "E after ϵ", combines "clusters" of effects:

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

Effect subsumption $\epsilon_1 \leq \epsilon_2$ holds when the write and read sets of ϵ_1 are subsets of the respective sets of ϵ_2.

5.4 Types

The value types (Figure 10), written A, B, 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

$\boxed{\Gamma \vdash (\epsilon_1 \text{ then } \epsilon_2) = \epsilon}$ Effect sequencing

$$\frac{\begin{array}{ll} \textit{extract}(\Gamma) \vdash W_1 \perp W_2 & \textit{extract}(\Gamma) \vdash W_1 \cup W_2 \equiv W_3 \\ \textit{extract}(\Gamma) \vdash R_1 \perp W_2 & \textit{extract}(\Gamma) \vdash R_1 \cup R_2 \equiv R_3 \end{array}}{\Gamma \vdash \langle W_1; R_1 \rangle \text{ then } \langle W_2; R_2 \rangle = \langle W_3; R_3 \rangle}$$

$\boxed{\Gamma \vdash \epsilon_1 \preceq \epsilon_2}$ Effect subsumption

$$\frac{\begin{array}{l} \textit{extract}(\Gamma) \vdash (X_1 \perp Z_1) \equiv Y_1 : \mathbf{NmSet} \\ \textit{extract}(\Gamma) \vdash (X_2 \perp Z_2) \equiv Y_2 : \mathbf{NmSet} \end{array}}{\Gamma \vdash \langle X_1; X_2 \rangle \preceq \langle Y_1; Y_2 \rangle}$$

$\boxed{\Gamma \vdash (E \text{ after } \epsilon) = E'}$ Effect coalescing

$$\frac{\Gamma \vdash (\epsilon_1 \text{ then } \epsilon_2) = \epsilon}{\Gamma \vdash \big((C \rhd \epsilon_2) \text{ after } \epsilon_1\big) = (C \rhd \epsilon)}$$

$$\frac{\Gamma \vdash (E \text{ after } \epsilon) = E'}{\begin{array}{l} \Gamma \vdash (\forall \alpha : K.\, E) \text{ after } \epsilon \;=\; (\forall \alpha : K.\, E') \\ \Gamma \vdash (\forall a : \gamma \mid P.\, E) \text{ after } \epsilon \;=\; (\forall a : \gamma \mid P.\, E') \end{array}}$$

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

$$\frac{\Gamma \vdash^{M} e : E_1 \qquad \Gamma \vdash E_1 \leq_E E_2}{\Gamma \vdash^{M} e : E_2} \text{ etype-sub}$$

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

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

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

$$\frac{\begin{array}{l} \Gamma \vdash^{M} e_1 : (\mathbf{F}\, A) \rhd \epsilon_1 \\ \Gamma, x : A \vdash^{M} e_2 : (C \rhd \epsilon_2) \qquad \Gamma \vdash (\epsilon_1 \text{ then } \epsilon_2) = \epsilon \end{array}}{\Gamma \vdash^{M} \mathsf{let}(e_1, x.e_2) : (C \rhd \epsilon)} \text{ let}$$

$$\frac{\Gamma, x : A \vdash^{M} e : E}{\Gamma \vdash^{M} (\lambda x.\, e) : \big((A \rightarrow E) \rhd \langle \emptyset; \emptyset \rangle\big)} \text{ lam}$$

$$\frac{\begin{array}{l} \Gamma \vdash (E \text{ after } \epsilon_1) = E_1 \\ \Gamma \vdash^{M} e : \big((A \rightarrow E) \rhd \epsilon_1\big) \qquad \Gamma \vdash v : A \end{array}}{\Gamma \vdash^{M} (e\, v) : E_1} \text{ app}$$

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

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

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

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

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

$$\frac{\begin{array}{l} \Gamma \vdash v : (\mathbf{Nm} \overset{\mathbf{Nm}}{\Rightarrow} \mathbf{Nm})[N'] \\ \Gamma \vdash^{N \circ N'} e : C \rhd \langle W; R \rangle \end{array}}{\Gamma \vdash^{N} \mathsf{scope}(v, e) : C \rhd \langle W; R \rangle} \text{ scope}$$

$$\frac{\Gamma, \alpha : K \vdash^{M} t : E}{\Gamma \vdash^{M} t : (\forall \alpha : K.\, E)} \text{ etype-}\forall\text{Intro}$$

$$\frac{\Gamma \vdash^{M} e : (\forall \alpha : K.\, E) \qquad \Gamma \vdash A : K}{\Gamma \vdash^{M} e : [A/\alpha]E} \text{ etype-}\forall\text{Elim}$$

$$\frac{\Gamma, a : \gamma, P \vdash^{M} t : E}{\Gamma \vdash^{M} t : (\forall a : \gamma \mid P.\, E)} \text{ etype-}\forall\text{IndexIntro}$$

$$\frac{\Gamma \vdash i : \gamma \qquad \begin{array}{l} \textit{extract}(\Gamma) \Vdash [i/a]P \\ \Gamma \vdash^{M} e : (\forall a : \gamma \mid P.\, E) \end{array}}{\Gamma \vdash^{M} e : [i/a]E} \text{ etype-}\forall\text{IndexElim}$$

$$\frac{\Gamma \vdash v : (\exists a : \gamma \mid P.\, A) \qquad \Gamma, a : \gamma, P, x : A \vdash^{M} e : E}{\Gamma \vdash^{M} \mathsf{vunpack}(v, a.x.e) : E} \text{ etype-}\exists\text{IndexElim}$$

Fig. 13. Computation typing

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 $(\mathbf{Nm} \overset{\mathsf{Nm}}{\Rightarrow} \mathbf{Nm})\,[M]$ 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 in Appendix B.1.

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 ($\forall a : \gamma \mid P.\, E$). In the latter quantifier, the proposition P lets us express quantification over disjoint sets of names.

Value typing rules. The typing rules for values (Figure 12) for unit, variables and pairs are standard. Rule ‘name’ uses index-level entailment to check that the name n is in the name set X. Rule ‘namefn’ checks that M_v is well-sorted, and that M_v is convertible to M. Rule ‘ref’ checks that n is in X, and that $\Gamma(n) = A$, that is, the typing $n : A$ appears somewhere in Γ. Rule ‘thunk’ is similar to ‘ref’.

Computation typing rules. Many of the rules that assign computation types (Figure 13) are standard—for call-by-push-value—with the addition of effects and the namespace M. The rules ‘split’ and ‘case’ have nothing to do with namespaces or effects, so they pass M up to their premises, and leave the type E unchanged. Empty effects are added by rules ‘ret’ and ‘lam’, since both $\mathtt{ret}$ and λ do not read or write anything. The rule ‘let’ uses effect sequencing to combine the effects of e_1 and the let-body e_2. The rule ‘force’ also uses effect sequencing, to combine the effect of forcing the thunk with the read effect $\langle \emptyset; X \rangle$.

The only rule that modifies the namespace is ‘scope’, which composes the given namespace N (in the conclusion) with the user’s $v = \mathtt{nmfn}\ N'$ in the second premise (typing e).

5.5 Subtyping

As discussed above, our type system can overapproximate names. The type $\mathsf{Nm}\,[X]$ means that the name is contained in the set of X; unless X is a singleton, the type system does not guarantee the specific name. Approximation induces subtyping: we want to allow a program to pass $\mathsf{Nm}\,[X_1]$ to a function expecting $\mathsf{Nm}\,[X_1 \perp X_2]$.

To design subtyping rules that are correct and easy to implement, we turn to the DML descendant Stardust [Dunfield 2007]. The subtyping rules in Stardust are generally a helpful guide, with the exception of the rule that compares atomic refinements. In Dunfield’s system, $\tau\,[i] \le \tau\,[j]$ if $i = j$ in the underlying index theory. For example, a list of length i is a subtype of a list of length j if and only if $i = j$ in the theory of integers. While approximate in the sense of considering all lists of length i to have the same type, the length itself is not approximate.

In contrast, our name set indices are approximations. Thus, our rule $\le_V$-name (Figure 14) says that $\mathsf{Nm}\,[X] \le_V \mathsf{Nm}\,[Y]$ if $X \subseteq Y$, rather than $X = Y$. Similarly, subtyping for references and thunks ($\le_V$-ref, $\le_V$-thk) checks inclusion of the associated name (pointer) set, not strict equality.

Our polymorphic types combine two fundamental typing constructs, universal quantification and guarded types (requiring that P hold for the quantified index a), so our rule $\le_V$-$\forall$L combines the Stardust rules ΠL for index-level quantification and $\supset$L for the guarded type [Dunfield 2007, p. 33]. Likewise, our $\le_V$-$\forall$R combines Stardust’s ΠR and $\supset$R.

$\boxed{\Gamma \vdash A \leq_V B}$ Value type A is a subtype of B

$$\frac{}{\Gamma \vdash A \leq_V A}\ \leq_V\text{-refl} \qquad \frac{\Gamma \Vdash X \subseteq Y}{\Gamma \vdash \mathsf{Nm}[X] \leq_V \mathsf{Nm}[Y]}\ \leq_V\text{-name}$$

$$\frac{\Gamma \vdash A_1 \leq_V B_1 \qquad \Gamma \vdash A_2 \leq_V B_2}{\Gamma \vdash A_1 \times A_2 \leq_V B_1 \times B_2}\ \leq_V\text{-}\times \qquad \frac{\Gamma \vdash A_1 \leq_V B_1 \qquad \Gamma \vdash A_2 \leq_V B_2}{\Gamma \vdash A_1 + A_2 \leq_V B_1 + B_2}\ \leq_V\text{-}+$$

$$\frac{\mathit{extract}(\Gamma) \Vdash X \subseteq Y \qquad \Gamma \vdash A \leq_V B}{\Gamma \vdash (\mathsf{Ref}[X]\ A) \leq_V (\mathsf{Ref}[Y]\ B)}\ \leq_V\text{-ref}$$

$$\frac{\mathit{extract}(\Gamma) \Vdash X \subseteq Y \qquad \Gamma \vdash E_1 \leq_C E_2}{\Gamma \vdash (\mathsf{Thk}[X]\ E_1) \leq_V (\mathsf{Thk}[Y]\ E_2)}\ \leq_V\text{-thk}$$

$$\frac{\Gamma \vdash M_1 =_\beta M_2}{\Gamma \vdash (\mathbf{Nm} \overset{\mathsf{Nm}}{\to} \mathbf{Nm})[M_1] \leq_V (\mathbf{Nm} \overset{\mathsf{Nm}}{\to} \mathbf{Nm})[M_2]}\ \leq_V\text{-namefn}$$

$$\frac{\Gamma \vdash i : \gamma \qquad \begin{array}{c}\mathit{extract}(\Gamma) \Vdash [i/a]P \\ \Gamma \vdash [i/a]A \leq_V B\end{array}}{\Gamma \vdash (\forall a : \gamma \mid P.\, A) \leq_V B}\ \leq_V\text{-}\forall\text{L} \qquad \frac{\Gamma, b : \gamma, P \vdash A \leq_V B}{\Gamma \vdash A \leq_V (\forall b : \gamma \mid P.\, B)}\ \leq_V\text{-}\forall\text{R}$$

$$\frac{\Gamma, a : \gamma, P_a \vdash A \leq_V [a/b]B \qquad \mathit{extract}(\Gamma, a : \gamma, P_a) \Vdash [a/b]P_b}{\Gamma \vdash (\exists a : \gamma \mid P_a.\, A) \leq_V (\exists b : \gamma \mid P_b.\, B)}\ \leq_V\text{-}\exists$$

Fig. 14. Subtyping on value types

$\boxed{\Gamma \vdash C \leq_C D}$ Computation type C is a subtype of D

$$\frac{\Gamma \vdash A \leq_V B}{\Gamma \vdash \mathbf{F}\,A \leq_C \mathbf{F}\,B}\ \leq_C\text{-lift} \qquad \frac{\Gamma \vdash A_2 \leq_V A_1 \qquad \Gamma \vdash E_1 \leq_E E_2}{\Gamma \vdash (A_1 \to E_1) \leq_C (A_2 \to E_2)}\ \leq_C\text{-arr}$$

$\boxed{\Gamma \vdash E_1 \leq_E E_2}$ Type-with-effects E_1 is a subtype of E_2

$$\frac{\Gamma \vdash C_1 \leq_C C_2 \qquad \Gamma \vdash \epsilon_1 \leq \epsilon_2}{\Gamma \vdash (C_1 \rhd \epsilon_1) \leq_C (C_2 \rhd \epsilon_2)}\ \leq_E\text{-eff} \qquad \frac{\Gamma, \alpha : K \vdash E_1 \leq_E E_2}{\Gamma \vdash (\forall \alpha : K.\, E_1) \leq_E (\forall \alpha : K.\, E_2)}\ \leq_E\text{-all-type}$$

$$\frac{\Gamma \vdash i : \gamma \qquad \begin{array}{c}\mathit{extract}(\Gamma) \Vdash [i/a]P \\ \Gamma \vdash [i/a]E_1 \leq_E E_2\end{array}}{\Gamma \vdash (\forall a : \gamma \mid P.\, E_1) \leq_E E_2}\ \leq_E\text{-all-index-L} \qquad \frac{\Gamma, a : \gamma, P \vdash E_1 \leq_E E_2}{\Gamma \vdash E_1 \leq_E (\forall a : \gamma \mid P.\, E_2)}\ \leq_E\text{-all-index-R}$$

Fig. 15. Subtyping on computation types

Unlike Stardust's Σ (and unlike our $\forall$), our existential types have a term-level pack construct, so an $\exists$ cannot be a sub- or supertype of a non-existential type. Thus, instead of rules analogous to Stardust's ΣL and ΣR, we have a single rule $\leq_V$-$\exists$ with $\exists$ on both sides, which specializes ΣR to the case when ΣL derives its premise. Like $\forall$, our $\exists$ incorporates a constraint P on the quantified variable, so our $\leq_V$-$\exists$ also incorporates the Stardust rules for *asserting types* ($\invamp$), checking that P_a entails P_b.

For refs and thunks, rules $\leq_V$-ref and $\leq_V$-thk are covariant in the name set describing the location. They are also covariant in the type of their contents: unlike an ordinary ML `ref` type, our Ref names a location, but the programs described by our type system cannot mutate that location. (To extend our theory to describe *editor* programs, we would need different rules; see Section 8.2.)

In our subtyping rules for computation types (Figure 15), rule $\leq_C$-arr reflects the usual contravariance of function domains, rule $\leq_E$-eff allows subsumption within effects ϵ, and the rules for computation-level $\forall$ follow our rules for value-level $\forall$.

Instead of an explicit transitivity rule, which is not trivial to implement, the transitivity of subtyping is admissible.

5.6 Bidirectional Version

The typing rules in Figures 12 and 13 are declarative: they define what typings are valid, but not how to derive those typings. The rules' use of names and effects annotations means that standard unification-based techniques, like Damas–Milner inference, are not easily applicable. For example, it is not obvious when to apply chk-AllIntro, or how to solve unification constraints over names and name sets.

We therefore formulate bidirectional typing rules that directly give rise to an algorithm. For space reasons, this system is presented in the supplementary material (Appendix D). We prove (in Appendix E) that our bidirectional rules are sound and complete with respect to the type assignment rules in this section:

Soundness (Thms. E.1, E.3): Given a bidirectional derivation for an annotated expression e, there exists a type assignment derivation for e without annotations.

Completeness (Thms. E.2, E.4): Given a type assignment derivation for e without annotations, there exist two annotated versions of e: one that synthesizes, and one that checks. (This result is sometimes called *annotatability.*)

6 DYNAMIC SEMANTICS

Name terms. Recall Fig. 8 (Sec. 4.2), which gives the dynamics for evaluating name term M to name term value V. Because name terms have no recursion, evaluating a well-sorted name term always produces a value (Theorem H.9).

Program expressions (Figure 16). Stores hold the mutable state that names dynamically identify. Big-step evaluation for expressions relates an initial and final store, and the "current scope" and "current node", to a program and value. We define this dynamic semantics, which closely mirrors prior work, to show that well-typed evaluations always allocate precisely.

To make this theorem meaningful, the dynamics permits programs to *overwrite* prior allocations with later ones: if a name is used ambiguously, the evaluation will replace the old store content with the new store content. The rules $\Downarrow$-ref and $\Downarrow$-thunk either extend or overwrite the store, depending on whether the allocated pointer name is precise or ambiguous, respectively. We prove that, in fact, well-typed programs always extend (and never overwrite) the store in any single derivation. (During change propagation, not modeled here, we begin with a store and dependency graph from a prior run, and even precise programs overwrite the store/graph, as discussed in Sec. 1.)

While motivated by incremental computation, we are interested in precise effects here, not change propagation itself. Consequently, this semantics is simpler than the dynamics of prior work. First, the store never caches values from evaluation, that is, it does not model function caching (memoization). Next, we do not build the dependency edges required for change propagation. Likewise, the "current node" is not strictly necessary here, but we include it for illustration. Were

Pointers	p, q	$::= n$	name constants
Stores	S	$::= \cdot$	empty store
		$\mid S, p{:}v$	p points to value v
		$\mid S, p{:}e @ M$	p points to thunk e, run in scope M

Notation: $S\{p \mapsto v\}$ and $S\{p \mapsto e@M\}$ *extend* S at p when $p \notin \mathrm{dom}(S)$
$S\{p \mapsto v\}$ and $S\{p \mapsto e@M\}$ *overwrite* S at p when $p \in \mathrm{dom}(S)$

$\boxed{S_1 \vdash^M_m e \Downarrow S_2; t}$ Under store S in namespace M at current node m, expression e produces new store S_2 and result t

$$\frac{S_1 \vdash^{M_1 \circ M_2}_m e \Downarrow S_2; e'}{S_1 \vdash^{M_1}_m \mathtt{scope}(M_2, e) \Downarrow S_2; e'}\ \Downarrow\text{-scope} \qquad \frac{M_1 \Downarrow_M \lambda a. M_2 \qquad [n/a]M_2 \Downarrow_M p}{S \vdash^M_m M_1\ (\mathtt{name}\ n) \Downarrow S; \mathtt{ret}(\mathtt{name}\ p)}\ \Downarrow\text{-name-app}$$

$$\frac{(M\, n) \Downarrow_M p \qquad S_1\{p \mapsto e @ M\} = S_2}{S_1 \vdash^M_m \mathtt{thunk}(\mathtt{name}\ n, e) \Downarrow S_2; \mathtt{ret}(\mathtt{thunk}\ p)}\ \Downarrow\text{-thunk} \qquad \frac{(M\, n) \Downarrow_M p \qquad S_1\{p \mapsto v\} = S_2}{S_1 \vdash^M_m \mathtt{ref}(\mathtt{name}\ n, v) \Downarrow S_2; \mathtt{ret}(\mathtt{ref}\ p)}\ \Downarrow\text{-ref}$$

$$\frac{S(p) = e @ M_0 \qquad S_1 \vdash^{M_0}_p e \Downarrow S_2; t}{S_1 \vdash^M_m \mathtt{force}(\mathtt{thunk}\ p) \Downarrow S_2; t}\ \Downarrow\text{-force} \qquad \frac{S(p) = v}{S \vdash^M_m \mathtt{get}(\mathtt{ref}\ p) \Downarrow S; \mathtt{ret}(v)}\ \Downarrow\text{-get} \qquad \frac{}{S \vdash^M_m t \Downarrow S; t}\ \Downarrow\text{-term}$$

Fig. 16. Excerpt from the dynamic semantics (see also Figure 21)

we modeling change propagation, rules ⇓-ref, ⇓-thunk, ⇓-get and ⇓-force would create dependency edge structure that we omit here. (These edges relate the current node with the node being observed.)

7 METATHEORY: TYPE SOUNDNESS AND PRECISE EFFECTS

In this section, we prove that our type system is sound with respect to evaluation and that the type system enforces precise effects: We establish that a well-typed, terminating program produces a terminal computation of the program's type, and that the actual dynamic effects are precise (Def. 7.2). Specifically, we show that the type system's static effects soundly approximate this dynamic behavior. Consequently, sequenced writes never overwrite one another.

We sometimes constrain typing contexts to be *store types*, which type store pointers but not program variables; hence, they only type *closed* values and programs:

Definition 7.1 (Store type). *We say that* Γ *is a* store typing, *written* Γ store-type, *when each assumption in* Γ *has the reference-pointer form* $p : A$ *or the thunk-pointer form* $p : E$.

Definition 7.2 (Precise effects). *Given an evaluation derivation* $\mathcal{D}$, *we write* $\mathcal{D}$ reads R writes W *for its* precise effects *(Figure 22 in the appendix).*

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

Main theorem: We write $\langle W'; R' \rangle \leq \langle W; R \rangle$ to mean that $W' \subseteq W$ and $R' \subseteq R$. For proofs, see Appendix C.

Theorem 7.1 (Subject Reduction).
If Γ_1 store-type *and* $\Gamma_1 \vdash M : \mathbf{Nm} \overset{\mathrm{Nm}}{\Rightarrow} \mathbf{Nm}$ *and* $\mathcal{S}$ *derives* $\Gamma_1 \vdash^M e : C \rhd \langle W; R \rangle$
and $\vdash S_1 : \Gamma_1$ *and* $\mathcal{D}$ *derives* $S_1 \vdash^M_m e \Downarrow S_2; t$ *then*
there exists $\Gamma_2 \supseteq \Gamma_1$ *such that* Γ_2 store-type *and* $\vdash S_2 : \Gamma_2$ *and* $\Gamma_2 \vdash t : C \rhd \langle \emptyset; \emptyset \rangle$
and $\mathcal{D}$ reads $R_{\mathcal{D}}$ writes $W_{\mathcal{D}}$ *and* $\langle W_{\mathcal{D}}; R_{\mathcal{D}} \rangle \leq \langle W; R \rangle$.

8 IMPLEMENTATION

8.1 Prototype in Rust

Using this on-paper design as a guide, we have implemented a preliminary prototype of Fungi in Rust. In particular, we implement each abstract syntax definition and typing judgement presented in this paper and appendix as a Rust datatype (a "deep" embedding of the language into Rust). We implement the bidirectional type system (Sec. D) as a family of Rust functions that produce judgement data structures (possibly with nested type or effect errors) from a Fungi syntax tree.

By using Rust macros, we implement a concrete syntax and associated parser that suffices for authoring examples similar to those in Sec. 3. In two ways, we deviate from the Fungi program syntax presented here: (1) Rust macros can only afford certain concrete syntaxes (2) Fungi programs use explicit (not implicit) index and type applications; inferring these arguments is future work.

The implementation of Fungi is documented and publicly available. At present, it consists of about 10K lines of Rust, and is complete enough to type check many basic examples, including the effects of structural recursion (`max` and `filter`, Fig. 3.2). We are actively extending the implementation to encompass the full reasoning power of the index and name term sub-languages, to type the rest of Sec. 3, and beyond.

For the latest version of Fungi, see `crates.io` and/or `docs.rs`, and search for "fungi-lang". *Note to reviewers: visiting those sites will deanonymize the authors; see supplemental material instead.*

8.2 Ongoing and Future Work

We are currently exploring a number of extensions to the formulation of Fungi presented here.

Interactive type derivations. To debug the examples' type and effect errors, we load the (possibly incomplete) typing derivations in an associated interactive, web-based tool. The tool makes the output typing derivation *interactive*: using a pointer, we can inspect the syntactic family/constructor, typing context, type and effect of each subterm in the input program, including indices, name terms, sorts, values, expressions, etc. Compared with getting parsing or type errors out of context (or else, only with an associated line number), we've found this interactive tool very helpful for teaching newcomers about Fungi's abstract syntax rules and type system, and for debugging examples (and Fungi) ourselves. This tool, the *Human-Fungi Interface* (HFI), is publicly available software.

As future work, we will extend HFI into an interactive *program editor*, based on our existing bidirectional type system, and the (typed) structure editor approach developed by Omar et al. [2017a]. We speculate that Fungi *itself* may be useful in the implementation of this tool, by providing language support for interactive, *incremental* developer features [Omar et al. 2017b]. Current approaches prescribe conversion to a distinct, "co-contextual" judgement form, whose design circumvents the memoization failure issue outlined in Sec. 1.1, but requires first *transforming* the contexts, and the modalities of the typing rules [Erdweg et al. 2015a; Kuci et al. 2017]. Fungi's explicit-name programming model may offer an alternative approach for authoring incremental type checkers, based on their "ordinary" judgments (rules, typing contexts, and modalities).

Incremental semantics for Fungi. Though not the focus on this paper, Fungi is an incremental language. We implement the incremental runtime semantics of Fungi by writing an interpreter using Adapton in Rust, as provided by an existing external library [Hammer et al. 2014, 2015; Adapton Developers 2018]. This external library implements the dynamic dependency graphs described (statically) by Fungi's type and effect system. Our near-time goal is to use Fungi as a target language for programs that act like "incremental glue" for mixtures of Fungi code and (appropriately behaved) high-performance Rust code.

Future work: Editor and Archivist. To distinguish imperative name allocation from name-precise computation, future versions of Fungi will introduce two *incremental computation roles*, which we term the *editor* and the *archivist*, respectively; specifically, we define the syntax for roles as $r ::= \mathsf{ed} \mid \mathsf{ar}$. The archivist role (ar) corresponds to computation whose dependencies we cache, and the editor role (ed) corresponds to computation that feeds the archivist with input changes, and demands any changed output that is relevant; in short, the editor represents the world outside the cached computation.

While the current type system prototype focuses only on the *archivist* role, leaving the editor role to the surrounding Rust code, future work will integrate the editor role into Fungi programs. For example, consider the following typing rules, which approximate (and extend) our full type system with a *role* r in each rule:

$$\frac{\Gamma \vdash v_n : \mathsf{Nm}[X] \qquad \Gamma \vdash v : A}{\Gamma \vdash \mathsf{ref}(v_n, v) : \mathsf{Ref}(A) \rhd r(X)} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : A \rhd \mathsf{ar}(X)\\ \Gamma, x : A \vdash e_2 : B \rhd \mathsf{ar}(Y)\\ \Gamma \vdash (X \perp Y) \equiv Z : \textbf{NmSet}\end{array}}{\Gamma \vdash \mathtt{let}(e_1, x.e_2) : B \rhd \mathsf{ar}(Z)} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : A \rhd \mathsf{ed}(X)\\ \Gamma, x : A \vdash e_2 : B \rhd \mathsf{ed}(Y)\\ \Gamma \vdash (X \cup Y) \equiv Z : \textbf{NmSet}\end{array}}{\Gamma \vdash \mathtt{let}(e_1, x.e_2) : B \rhd \mathsf{ed}(Z)}$$

These rules are similar to the simplified rules presented in Sec. 2. In contrast to those rules, these conclude with the judgement form $\Gamma \vdash e : A \rhd r(X)$, mentioning the written set with the notation $\rhd r(X)$, where the set X approximates the set of written names (as in the earlier formulation), and r is the role (absent from the earlier formulation).

The first rule types a reference cell allocation, as before; in the rule's conclusion, this name set X serves as the allocation's *write set.* The undetermined role r means that this rule is applicable to both the editor and the archivist roles.

What was one `let` sequencing rule (in Sec. 2) is now two rules here: The second rule enforces the archivist role, where names are precise. The third rule permits the editor role, where names allocated later may *overwrite* names allocated earlier. Finally, a new syntax form `archivist(e)` permits the editor's computations to delegate to archivist sub-computations; the program `archivist(e)` has role ed whenever program e types under role ar under the same typing context.

Among the future work for mixing these roles, we foresee that extending the theory of Fungi, including covariant index subtyping, to this mixture of imperative-functional execution semantics requires mixing imperative effects (for the editor) and type index subtyping (for the archivist) in a disciplined, sound manner.

9 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.

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 $0{\cdot}n = \langle\!\langle \texttt{leaf}, n \rangle\!\rangle$ and $1{\cdot}n = \langle\!\langle \langle\!\langle \texttt{leaf}, \texttt{leaf} \rangle\!\rangle, n \rangle\!\rangle$ share the right subtree n—which allows programmers to deterministically compute two distinct names from one.

Type systems for variable binding and fresh name generation, such as FreshML [Pitts and Gabbay 2000] and Pure FreshML [Pottier 2007], can express that sets of names are disjoint. But the names lack internal structure that relates specific names across disjoint name sets.

Compilers have long used alias analysis to support optimization passes. Brandauer et al. [2015] extend alias analysis with disjointness domains, which can express local (as well as global) aliasing constraints. Such local constraints are more fine-grained than classic region systems; our work differs in having a rich structure on names.

Techniques for general-purpose incremental computation. General-purpose incremental computation techniques provide a general-purpose *change propagation* algorithm. In particular, after an initial run of the program, as the input changes dynamically, change propagation provides a provably sound approach for recomputing the affected output [Acar et al. 2006a; Acar and Ley-Wild 2009; Hammer et al. 2014, 2015]. Incremental computation can deliver *asymptotic* speedups for certain algorithms [Acar et al. 2007, 2008, 2009; Sümer et al. 2011; Burckhardt et al. 2011; Chen et al. 2012], and has even addressed open problems [Acar et al. 2010]. These incremental computing abstractions exist in many languages [Shankar and Bodik 2007; Hammer et al. 2009; Acar and Ley-Wild 2009]. The type and effect system proposed here complements past work on self-adjusting computation. In particular, we expect that variations of the proposed type system can express and verify the use of names in much of the work cited above.

Çiçek et al. [2015] develop cost semantics for a limited class of incremental programs: they support only in-place input changes and fixed control flow, so that the structure of the dynamic dependency graph is fixed. For example, the length of an input list cannot change across successive incremental runs, nor can the structure of its dependency graph. Çiçek et al. [2016] relax the restriction on control flow (but not input changes) to permit replacing a dependency subgraph according to a different, from-scratch execution. Extending their cost semantics to allow general structural changes (e.g. insertion or removal of list elements), while describing the cost of change propagation for programs like `rev` from Sec. 1.1, will require integrating a general notion of names. Without such a notion, constant-sized input changes may cascade, precipitating needlessly inefficient change propagation behavior.

Imprecise (ambiguous) names. Some past systems dynamically detect ambiguous names, either forcing the system to fall back to a non-deterministic name choice [Acar et al. 2006a; Hammer and Acar 2008], or to signal an error and halt [Hammer et al. 2015]. In scenarios with a non-deterministic fall-back mechanism, a name ambiguity carries the potential to degrade incremental performance, making it less responsive and asymptotically unpredictable in general [Acar 2005]. To ensure that incremental performance gains are predictable, past work often merely assumes, without enforcement, that names are precise [Ley-Wild et al. 2009].

Fortunately, these existing approaches are complementary to Fungi, whose type and effect system is applicable to each, either *directly* (in the case of Adapton, and variants), or with some minor adaptations (as we speculate for the others).

10 CONCLUSION

We define the *precise name problem* for programs that use explicit names to identify their dynamic data and sub-computations. We define a solution in the form of Fungi, a core calculus for such programs, whose type and effect system describes and verifies their allocation names. We derive a bidirectional version of the type and effect system, and we implement a closely-related prototype of Fungi in Rust, as a deeply-embedded DSL. We apply Fungi to a library of incremental collections.

Our ongoing and future work on Fungi builds on initial prototypes reported here: We are extending Fungi to settings that *mix* imperative and functional programming models, and we are creating richer tools for developing, debugging and visualizing Fungi programs in the context of larger systems (e.g., written in Rust).

ACKNOWLEDGMENTS

We thank Ryan L. Vandersmith, who leads the development of the *Human-Fungi Interface* described in Sec. 8; this tool has been invalable for implementing and testing our Fungi prototype in Rust.

We thank Neelakantan R. Krishnaswami, Deepak Garg, Roly Perera, and David Walker for insightful discussions about this work, and for their suggestions and comments. This material is based in part upon work supported by a gift from Mozilla, a gift from Facebook, 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, Facebook or the National Science Foundation.

REFERENCES

Umut A. Acar. 2005. *Self-Adjusting Computation.* Ph.D. Dissertation. Department of Computer Science, Carnegie Mellon University.

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

Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. 2006b. A Library for Self-Adjusting Computation. *Electronic Notes in Theoretical Computer Science* 148, 2 (2006).

Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. 2009. An Experimental Analysis of Self-Adjusting Computation. *TOPLAS* 32, 1 (2009), 3:1–53.

Umut A. Acar, Guy E. Blelloch, Matthias Blume, and Kanat Tangwongsan. 2006a. An Experimental Analysis of Self-Adjusting Computation. In *Proceedings of the ACM Conference on Programming Language Design and Implementation.*

Umut A. Acar, Andrew Cotter, Benoît Hudson, and Duru Türkoğlu. 2010. Dynamic Well-Spaced Point Sets. In *Symposium on Computational Geometry.*

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

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

Adapton Developers. 2018. *Adapton.* https://github.com/adapton

Pramod Bhatotia, Pedro Fonseca, Umut A. Acar, Björn B. Brandenburg, and Rodrigo Rodrigues. 2015. iThreads: A Threading Library for Parallel Incremental Computation. In *ASPLOS.*

Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin. 2011. Incoop: MapReduce for Incremental Computations. In *ACM Symposium on Cloud Computing.*

Stephan Brandauer, Dave Clarke, and Tobias Wrigstad. 2015. Disjointness Domains for Fine-grained Aliasing. In *OOPSLA.* ACM Press, 898–916.

Sebastian Burckhardt, Daan Leijen, Caitlin Sadowski, Jaeheon Yi, and Thomas Ball. 2011. Two for the Price of One: A Model for Parallel and Incremental Computation. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications.*

Ezgi Çiçek, Deepak Garg, and Umut A. Acar. 2015. Refinement Types for Incremental Computational Complexity. In *ESOP.*

Ezgi Çiçek, Zoe Paraskevopoulou, and Deepak Garg. 2016. A Type Theory for Incremental Computational Complexity with Control Flow Changes. In *ICFP.*

Yan Chen, Joshua Dunfield, and Umut A. Acar. 2012. Type-Directed Automatic Incrementalization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* ACM Press, 299–310.

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

Evan Czaplicki and Stephen Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. In *PLDI.*

Joshua Dunfield. 2007. *A Unified System of Type Refinements.* Ph.D. Dissertation. Carnegie Mellon University. CMU-CS-07-129.

Joshua Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism. In *ICFP*. ACM Press. arXiv:`1306.6032 [cs.PL]`.

Joshua Dunfield and Frank Pfenning. 2004. Tridirectional Typechecking. In *Principles of Programming Languages*. ACM Press, 281–292.

Sebastian Erdweg, Oliver Bracevac, Edlira Kuci, Matthias Krebs, and Mira Mezini. 2015a. A co-contextual formulation of type rules and its application to incremental type checking. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. 880–897.

Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. 2015b. A sound and optimal incremental build system with dynamic dependencies. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. 89–106.

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

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

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

Matthew A. Hammer, Umut A. Acar, and Yan Chen. 2009. CEAL: a C-Based Language for Self-Adjusting Computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Matthew A. Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael Hicks, and David Van Horn. 2015. Incremental Computation with Names. In *OOPSLA*. ACM Press, 748–766.

Matthew A. Hammer, Yit Phang Khoo, Michael Hicks, and Jeffrey S. Foster. 2014. Adapton: Composable, Demand-driven Incremental Computation. In *PLDI*. ACM Press.

Kyle Headley and Matthew A. Hammer. 2016. Simple Persistent Sequences. In *Trends in Functional Programming*.

Fritz Henglein, Henning Makholm, and Henning Niss. 2005. Effect Types and Region-Based Memory Management. In *Advanced Topics in Types and Programming Languages*, B. C. Pierce (Ed.). MIT Press, Chapter 3, 87–135.

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

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

Edlira Kuci, Sebastian Erdweg, Oliver Bracevac, Andi Bejleri, and Mira Mezini. 2017. A Co-contextual Type Checker for Featherweight Java. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*. 18:1–18:26.

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

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

Ruy Ley-Wild, Umut A. Acar, and Matthew Fluet. 2009. A Cost Semantics for Self-Adjusting Computation. In *Principles of Programming Languages*.

Ruy Ley-Wild, Matthew Fluet, and Umut A. Acar. 2008. Compiling Self-Adjusting Programs with Continuations. In *ICFP*.

Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017a. Hazelnut: a bidirectionally typed structure editor calculus. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 86–99.

Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A. Hammer. 2017b. Toward Semantic Foundations for Program Editors. In *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*. 11:1–11:12.

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Prog. Lang. Syst.* 22 (2000), 1–44.

Andrew M. Pitts and Murdoch J. Gabbay. 2000. A Metalanguage for Programming with Bound Names Modulo Renaming. In *Mathematics of Program Construction*. Springer.

François Pottier. 2007. Static Name Control for FreshML. In *Logic in Computer Science*. 356–365.

William Pugh and Tim Teitelbaum. 1989. Incremental computation via function caching. In *POPL*.

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science*. 55–74. http://www.cs.cmu.edu/~jcr/seplogic.pdf

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

Ajeet Shankar and Rastislav Bodik. 2007. DITTO: Automatic Incrementalization of Data Structure Invariant Checks (in Java). In *Programming Language Design and Implementation.*

Özgür Sümer, Umut A. Acar, Alexander Ihler, and Ramgopal Mettu. 2011. Adaptive Exact Inference in Graphical Models. *Journal of Machine Learning* 8 (2011), 180–186.

Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Information and Computation* 132, 2 (1997), 109–176.

Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. 2013. Abstract Refinement Types. In *European Symp. on Programming*. Springer, Berlin Heidelberg, 209–228.

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

Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *Principles of Programming Languages.* ACM Press, 214–227.