Fungi: Typed incremental computation with names

MATTHEW A. HAMMER, University of Colorado Boulder
JANA DUNFIELD, Queen's University
KYLE HEADLEY, University of Colorado Boulder
MONAL NARASIMHAMURTHY, University of Colorado Boulder
DIMITRIOS J. ECONOMOU, University of Colorado Boulder

Incremental computations attempt to exploit input similarities over time, reusing work that is unaffected by input changes. To maximize this reuse in a general-purpose programming setting, programmers need a mechanism to identify dynamic allocations (of data and subcomputations) that correspond over time.

We present Fungi, a typed functional language for incremental computation with *names*. Unlike prior general-purpose languages for incremental computing, Fungi's notion of names is formal, general, and statically verifiable. Fungi's type-and-effect system permits the programmer to encode (program-specific) local invariants about names, and to use these invariants to establish *global uniqueness* for their composed programs, the property of using names correctly. We prove that well-typed Fungi programs respect global uniqueness.

We derive a bidirectional version of the type and effect system, and we have implemented a prototype of Fungi in Rust. We apply Fungi to a library of incremental collections, showing that it is expressive in practice.

1 INTRODUCTION

In many software systems, a fixed algorithm runs repeatedly over a series of *incrementally changing* inputs ($\mathsf{Inp}_1, \mathsf{Inp}_2, \ldots$), producing a series of *incrementally changing* outputs ($\mathsf{Out}_1, \mathsf{Out}_2, \ldots$). For example, programmers often change only a single line of source code and recompile, so Inp_t is often similar to Inp_{t-1}.

The goal of incremental *computation* is to exploit input similarity by reusing work from previous runs. If the source code Inp_t is almost the same as Inp_{t-1}, much of the work done to compile Inp_t and produce the target Out_t can be reused. In many settings, this reuse leads to asymptotic improvements in running time.

Such improvements are possible when the recomputation is *stable*: when the work done by run $t-1$, producing output Out_{t-1} from input Inp_{t-1}, is similar to the work needed for run t to produce output Out_t from Inp_t. In some cases, such as total replacement of the source program being compiled, stability is impossible. Thus, a central design question is how to maximize stability.

Consider a simple program that applies a binary operation g to two parts (x, y) of the input, and then applies another binary operation f to the result of g and a third part (z) of the input. This program has three inputs, one output, and one *intermediate result* (the result of g on x and y).

Authors' addresses: Matthew A. Hammer, University of Colorado Boulder, Department of Computer Science; Jana Dunfield, Queen's University, School of Computing; 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.

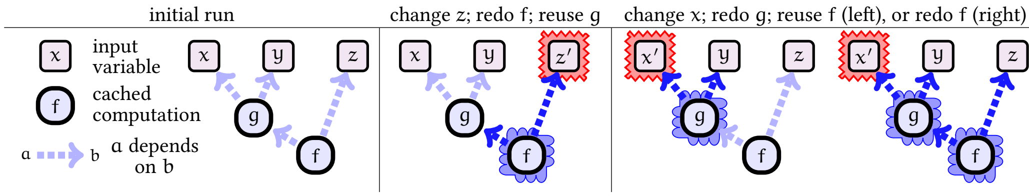

Fig. 1. Reuse across several program runs

Assuming efficient equality tests for x, y and the result of g, we can save this intermediate result and, potentially, reuse it across runs.

Fig. 1 shows some example runs. In the first run, we have stored $g(x, y)$. In the second run, the user has changed the input z to z'—but since the inputs x and y have not changed, we can reuse the result $g(x, y)$ and perform only the operation f. In the third run, the user has changed x to x', which requires doing the operation g again.

Thus, between the first and second runs we had to recompute only f; between the second and third runs, we had to recompute g. Depending on whether g's result changes, we might recompute f as well.

At this low level of complexity, it may seem straightforward to ensure that the incremental program is both *consistent* and *efficient*:

- An incremental program is *from-scratch consistent* if its output matches the output that would be produced by running the program from scratch (that is, without using saved intermediate results).
  As long as we reuse the result of g only when x and y have not changed, and reuse f only when g and z have not changed, this simple program is from-scratch consistent.
- An incremental program is *incrementally efficient* (or *achieves incremental efficiency*) if it does only the *necessary* new work.
  As long as we *always* reuse the result of g when x and y have not changed, and *always* reuse f when g and z have not changed, this simple program is incrementally efficient.

For nontrivial programs, however, achieving both incremental consistency and incremental efficiency can be extremely difficult. Consider GNU `make`, a relatively simple build system: it achieves consistency (at least in principle) only by working at a very coarse level of granularity—entire programs (`cc`, `ld`, etc.) and entire files. Opportunities to reuse work *within* a 5,000-line input to `cc` are missed, and understandably so: compilers are large systems that use complex data structures and clever algorithms. Merely comparing file modification times (or even file contents) cannot utilize, say, the fact that the result of a liveness analysis has not changed. (Or, that the analysis has changed *slightly*, which creates many subtle dependencies.)

The gold standard for incremental programs is to painstakingly design an incremental algorithm that explicitly saves results and reuses work, perhaps in very clever ways. In many development settings, it is not feasible to expend that kind of effort. Rather than giving up on incremental software (by not attempting to reuse work at all) or using simplistic approaches (along the lines of `make`) that miss many opportunities for reuse, we should offer incremental programming *languages* that allow programmers to easily build incremental programs that are correct and efficient, *at scale*. Thus, an incremental programming language should enable programmers (1) to store and reuse intermediate results, without drastically changing their source program; (2) to exploit similarities (between inputs, and between stored results), including for highly structured input data and nontrivial data structures; (3) to easily combine smaller incremental programs into incremental systems. Moreover, the language should make it as easy as possible to obtain both correctness and efficiency.

Incremental languages can be categorized by their breadth of applicability, with domain-specific languages at one end of the spectrum and general-purpose languages at the other; the language in this paper is general-purpose. The central advance we make is in *statically* verifying an important aspect of incremental programs: that subcomputations are named *uniquely* within each run.

The tiny program shown above is not adequate to illustrate the need for unique naming: the program's input has no interesting structure, and there is only one intermediate result. We argue the need for names themselves here; we will discuss a concrete example, illustrating the need for unique names, in Sec. 2.

To reuse a unit of work, we must observe that the newer result *corresponds to* the older result. The program $f(g(x, y), z)$ uses no control structures and performs the operations f and g exactly once, so it is immediate that $g(x, y)$ in the second run corresponds to $g(x, y)$ in the first run. Moreover, we say that $g(x', y)$ in the *third* run corresponds to $g(x, y)$ in the second run, even though x' is (probably) not equal to x and hence $g(x', y)$ is (probably) not equal to $g(x, y)$: Correspondence is not equality; instead, correspondence is the idea that two uses of g happen "in the same place".

The correspondence of x' to x, and z' to z, is even more immediate. But what if, instead of giving three discrete inputs (x, y, z), we gave a list of integers as input? If the change in input across runs is confined to specific list elements, say replacing the second element 22 with 23, we could say that the kth element of the previous input corresponds to the kth element of the current input. However, if the change is to *insert* an element in the input list, identifying the kth element at time $t - 1$ with the kth element at time t won't work: the small change of inserting a single element will look like the complete replacement. We need some notion of *identity* to realize that, if we insert an element at (say) the head of the list, the 1st element at time $t - 1$ corresponds to the 2nd element at time t, the 2nd element at time $t - 1$ corresponds to the 3rd element at time t, and so forth.

In our setting of a general-purpose language, there is no one-size-fits-all notion of identity. Instead, we need to enable programmers to choose a notion of identity that is appropriate for each program—a notion that exposes appropriate correspondences, and hence enables reuse. We call this notion of identity a *naming strategy*. Choosing a naming strategy that actually enables reuse is often difficult; the study of incremental cost semantics, which describe the potential for reuse, is a research area in itself. Our contribution is to make it easier for programmers to experiment with different naming strategies: the Fungi type system rules out a large class of *naming errors* that, in earlier languages such as Nominal Adapton [Hammer et al. 2015], could only be caught at run time.

In Table 1, we compare Fungi to some related approaches. The first two rows list work on incremental languages for substantially different programming models; those systems' answers to the question of how to identify corresponding subcomputations do not apply in our setting (nor would our answer apply in theirs). We briefly discuss these two systems, and other work in substantially different settings, in Section 8. The remaining rows in the table—starting with AFL [Acar et al. 2002]—list general-purpose incremental programming languages that endeavor to provide a standard programming model with (relatively) *lightweight* incrementality; as we noted above, we want to support incrementality without requiring programmers to drastically change their source programs. Within this broad setting, we can observe an evolution from no mechanism to identify corresponding subcomputations (AFL in 2002) to informal or specialized mechanisms (several papers through 2012 and Adapton in 2014), and then to formal mechanisms.

Contributions. We make the following contributions:

- We develop a type-and-effect system for a general-purpose incremental programming language (Sections 3 and 4). Using refinement types, the system statically relates names to

approach	programming model	mechanism to identify corresponding subcomputations	detection of naming errors
Demetrescu et al. [2011]	reactive/imperative	memory address	n/a
Concurrent revisions [Burckhardt et al. 2011]	revision-based imperative programming	call graphs **(1)**	n/a
AFL [Acar et al. 2002]	functional language	none	n/a
Carlsson [2002]	functional language	none	n/a
Acar et al. [2006b,a]	functional language	keys (informal)	**(2)**
DeltaML [Acar and Ley-Wild 2008]	functional language	keys (informal)	**(2)**
CEAL [Hammer et al. 2009]	imperative language	keys (informal)	**(2)**
implicit SAC [Chen et al. 2011, 2012]	functional language	keys (informal)	**(2)**
Adapton [Hammer et al. 2014]	functional language	structural (hash-consing)	n/a
Nominal Adapton [Hammer et al. 2015]	functional language	names (formal)	dynamic
Fungi (this paper)	functional language	names (formal)	static

(1) position in global call graph; small changes in call graph structure prevent reuse
(2) fall back to a global counter—preventing reuse now, in the future, or both

Table 1. Some approaches to incremental computation

allocated data (references) and computations (thunks); it supports a set of type-level operations on names that is large enough to describe sophisticated uses of names, but small enough for decidable type checking.

- In Section 6, we prove that the effects tracked by our system are sound with respect to our dynamic semantics (Section 5). As a consequence, our type system ensures, *statically*, that names are unique within each run of the program—a property that, previously, could only be checked dynamically [Hammer et al. 2015]. In nontrivial programs, this *global uniqueness* property is a consequence of *local uniqueness* properties that are specific to particular algorithms and data structures; see Section 2.
- We implement the type system, and demonstrate its applicability to a variety of examples (Section 7).

2 OVERVIEW

In this section, we use an example program to give an overview of Fungi as a typed language for incremental computation with names. Specifically, we consider the from-scratch semantics, typing, and incremental semantics of dedup, a list-processing function that removes duplicates: the output list retains only the first occurrence of each input list element.

The implementation of dedup uses names to create correspondences between similar inputs, leading to incremental reuse via an efficient application of a (general-purpose) change propagation algorithm. The correctness of change propagation relies on the global uniqueness of allocation names, explained below.

The Fungi type system ensures that dedup satisfies global uniqueness; to do so, the Fungi programmer uses types to express several local uniqueness invariants. Before discussing this example, we briefly discuss these naming properties, which are each fundamental to the novel design of Fungi as a language for typed incremental computation with names.

2.1 Naming properties

Our Fungi type system enforces the *global uniqueness* of names. For nontrivial programs, global uniqueness requires *local uniqueness* of names; our type system also checks local uniqueness properties as stated by the programmer.

Global uniqueness of allocation names: For every allocated reference cell or thunk, the name used to identify the allocated reference (or thunk) is unique.

Local uniqueness properties: The data structures in an incremental program may contain names. For example, if we map over a list, we may need to associate the third element of the input list with the third element of the output list. The name used to represent "being the third element" may then occur within related *pointer* names, such as the pointer names of the third element of the input and the third element of the output. The name that represents "being the third element" may be stored in several different lists, but it should not occur more than once within each list: the input list cannot have two third elements. Since the appropriate local uniqueness properties depend on the details of each program, they cannot be given *a priori.* Instead, the programmer or library designer expresses the appropriate properties, using the Fungi type system.

In general, local uniqueness—in the form appropriate to each program—is needed to ensure global uniqueness. Our type system rules out, statically, violations of global uniqueness *and* violations of local uniqueness. While previous systems such as Nominal Adapton included constant-time dynamic checks to catch violations of global uniqueness, most local uniqueness properties cannot be checked in constant time.

Since local uniqueness violations can lead to *subsequent* global uniqueness violations, being able to statically ensure local uniqueness rules out a large class of subtle errors—much like the advanced type system of the Rust language rules out dangling pointers. As we show below (Sec. 2.6), some violations in these principles are only triggered by *certain* inputs, which may be unlikely, and thus unlikely to show up in randomized dynamic tests.

By enforcing these principles of unique names statically, Fungi programs enjoy the guarantees they afford, e.g., that change propagation will work correctly.

These principles about names, which are fundamental to general-purpose incremental computation, have been applied in some incremental computing systems of the past, but until now, have not been codified formally, or statically verified (see Table 1 for details).

In some past systems (based on self-adjusting computation), the runtime dynamically detects and tolerates violations of these uniqueness properties—the names are called "keys", and are viewed as hints that can be wrong, or non-unique. In the cases that they are not unique, the caching/allocation mechanism falls back to using a global counter. In turn, this cache location choice is not based on the current input, is not functional, and consequently, it will generally not be reusable as a "replayed" allocation in subsequent invocations of change propagation on similar inputs.

In other systems (Nominal Adapton), the runtime system simply triggers a dynamic error for violations of global uniqueness.

No prior system of which we are aware permits programmers to systematically encode or check local uniqueness, either statically, or dynamically (which would be expensive).

Next, to make these ideas concrete, we consider an example.

2.2 The program listing and dynamic semantics of `dedup`

Fig. 2 gives the program listing for `dedup`, including type declarations. The right-hand column of the figure shows additional type declarations, explained further below (Sec. 2.4).

```
idxtm Dedup : NmSet → NmSet
  = λX.{@t}•(Insert X) ⊥ {@dd}•X ⊥ {@r}•X

dedup : ∀X1⊥X2:NmSet.
  List[X1] → Trie[X2] → List[X1]
  ▷[Dedup X1]

dedup l t =
  match (get l) with
  Nil ⇒ l
  Cons x y ys ⇒
    let (tx,b) = scope⟨@t⟩ insert x y t
    let ddys = thunk⟨@dd•x⟩{dedup ys tx}
    if b then force ddys else
      ref⟨@r•x⟩(Cons x y (force ddys))
```

```
type List[X] = Ref(ListNode[X])

type ListNode : NmSet ⇒ Type
  Nil  : ∀X:NmSet. ListNode[X]
  Cons : ∀X1⊥X2:NmSet.
    Nm[X1] → Nat → List[X2] →
    ListNode[X1⊥X2]

type Trie[X] = Ref(TrieNode[X])

idxtm Insert : NmSet → NmSet
insert : ∀X1⊥X2:NmSet.
  Nm[X1] → Nat → Trie[X2] →
   (Trie[X1⊥X2], Bool)
   ▷[Insert X1]
```

Fig. 2. The effect, type and code listing for dedup (left), and definitions for linked lists and hash tries (right).

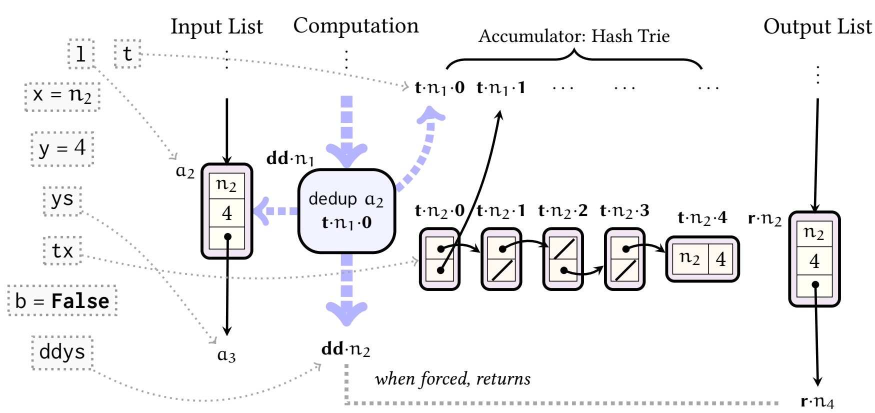

Fig. 3. The static variables of dedup (left column), and their dynamic values and structure (to the right), for the second iteration of dedup on the input list $[3, 4, 3, 9]$, where the list element y is 4.

First, let's consider an approximation of the declared type and code for dedup, ignoring the *index term* declaration (**idxtm** Dedup) and other type indices and effects. The type declaration of dedup says that it accepts two arguments, a list of type `List[X1]` and a hash trie of type `Trie[X2]`, and returns a list of type `List[X1]`. Before examining the type structure of dedup in more detail, we consider the code, and its dynamic semantics.

Consider the initial run of dedup on the input list $[3, 4, 3, 9]$, stored at the sequence of pointer addresses $\langle a_1, a_2, a_3, a_4, a_5\rangle$, which store Cons cells and a terminal `Nil` value. In addition to the elements $[3, 4, 3, 9]$, the Cons cells also contain a sequence of names (as values) $\langle n_1, n_2, n_3, n_4\rangle$, with one name per Cons cell.

The dedup function uses these names to determine its allocation names—the identities of allocated data and thunks. Moreover, it stores these names (as values) within the allocated data. Intuitively, these names identify the logical places of the Cons cells in the input list, and by copying these names into these other allocated values, they permit the dedup program to create correspondences with other logical places in its data. Further below, we will look at a full picture of this entire execution.

Fig. 3 shows a single "tile" of this execution, for the iteration on input cell a_2 containing name n_2 and input element 4. The left side of the figure gives the rest of dedup's static variables, and relates them to their dynamic values and structure, including the input list, accumulator and output list. We consider the remainder of Fig. 3 in the context of the dedup algorithm (Fig. 2). The dedup algorithm processes the input list `l` using structural recursion, retaining the first occurrence of each element and filtering out subsequent occurrences. We use a hash trie argument `t` to efficiently represent the set of input elements already processed. The final **`if`-`else`** expression branches on this case: if `b` is true, the element has already been seen so we recur (**`force`** `ddys`) without building an output `Cons` cell; if `b` is false (**`else`** branch), the element has not been seen and we construct an output `Cons` cell.

In this tile (Fig. 3), the `Cons` case of dedup consists of handling the given named list position ($\mathtt{x} = n_2$), holding a list element ($\mathtt{y} = 4$), and recursively processing the elements in the tail of the `Cons` cell ($\mathtt{ys} = a_3$). To do so efficiently, dedup inserts element $\mathtt{y} = 4$ into the current trie (`t`) using a helper function `insert`.

The call to `insert` uses a *write scope* (written **`scope`**), which permits the call site to specialize the names written by this function: the names written by function `insert` are each prefixed by the given name constant `@t`. (In listings, we include the symbol `@` to distinguish the constant name **t** from the static program variable `t`; to save space, in figures we use the face **t** for this same name constant `@t`, and omit the `@` symbol.)

As Fig. 3 shows, the value of variable `t` is (pointer) name $\mathbf{t}\cdot n_1\cdot\mathbf{0}$, which contains the prefix (sub-name) **t**, and suffix (sub-name) **0**. Similarly, the value of variable `tx` is (pointer) name $\mathbf{t}\cdot n_2\cdot\mathbf{0}$, which contains the same prefix **t** and suffix **0**, but uses name n_2 in place of name n_1. For reasons we explain below (Sec. 2.6), omitting this write scope name **t** would constitute a static typing error for this program, as it would allow *some* input lists to violate global uniqueness (Sec. 2.1).

Next, dedup allocates an *explicitly named* thunk to identify its recursive call, giving the subexpression `@dd•x`, which prepends the name constant **dd** to the name value of variable `x`, in this case n_2, resulting in the dynamic name $\mathbf{dd}\cdot n_2$. As Fig. 3 shows, our "current thunk" for this iteration is $\mathbf{dd}\cdot n_1$, and the next iteration (statically named `ddys`) is dynamically named $\mathbf{dd}\cdot n_2$, using the "current name" n_2. As we explain in Sec. 2.6, omitting this prefix **dd** constitutes a static naming error, since it violates global uniqueness (unique pointer names).

Finally, the dedup code executes the **`else`** case, since input element 4 is not present in the input trie $\mathtt{t} = \mathbf{t}\cdot n_1\cdot\mathbf{0}$, and is making its first occurrence with this `Cons` cell. Similarly with **`thunk`** above, the dedup code allocates an *explicitly named* reference cell to identify its resulting list cell, giving the subexpression `@r•x`, which prepends the name constant **r** to $\mathtt{x} = n_2$, resulting in the dynamic name $\mathbf{r}\cdot n_2$. As Fig. 3 shows, the "next thunk" that this iteration forces, $\mathbf{dd}\cdot n_2$, returns the list pointer $\mathbf{r}\cdot n_4$, which we store as the tail of the output `Cons` cell $\mathbf{r}\cdot n_2$. As above, omitting this prefix **r** violates global uniqueness. Other naming mistakes are possible as well; see Sec. 2.6.

In the remainder of the overview, we define the internal structure of names and discuss reasoning about uniqueness (Sec. 2.3), discuss the details of the `insert` function used by dedup, and consider static reasoning for dedup's use of names (Sec. 2.4).

2.3 Apartness for names, name sets and name functions

We briefly describe the structure of names, and discuss a notion that underpins Fungi's design: *apartness* of names, name sets and name functions.

Our core calculus defines names as binary trees, $n ::= \mathtt{leaf} \mid \langle\langle n, n\rangle\rangle$. In practice, we augment this definition in two small ways. First, we extend the `leaf` production with other terminal productions for numbers and symbolic constants, written **0** and **t** (respectively) in the example above. For the purposes of reasoning formally, we assume (unspecified) encodings of these terminal productions into the simple formal grammar above. Second, we use a more lightweight notation for binary name composition: $n_1 \cdot n_2 \cdot n_3$ denotes $\langle\!\langle n_1, \langle\!\langle n_2, n_3 \rangle\!\rangle \rangle\!\rangle$. This is only a convenient notation; names are still trees, so (unlike string concatenation) binary name composition is not associative: $n_1 \cdot n_2 \cdot n_3 \neq (n_1 \cdot n_2) \cdot n_3$, since $\langle\!\langle n_1, \langle\!\langle n_2, n_3 \rangle\!\rangle \rangle\!\rangle \neq \langle\!\langle \langle\!\langle n_1, n_2 \rangle\!\rangle, n_3 \rangle\!\rangle$.

To respect the principles of unique names, Fungi encodes "uniqueness" through *apartness*. Apartness plays a central role in our type-and-effects system and metatheory. In Fungi code, we read the connective $\perp$ as "apart", a notion that (1) generalizes the operation of (*disjoint*) set union and (2) asserts that the left- and right-hand operands are indeed disjoint, with no common names.

Unlike disjoint set union, which is only defined for sets, our type system defines apartness over (pairs of) *name terms* and *index terms*. Name terms include functions over names, as well as literal names; index terms include name sets. Informally, we say that (1) two *names* n_1 and n_2 are apart if they are not equal (if n_1 and n_2 are distinct binary trees), (2) two *sets of names* are apart if they are disjoint, and (3) two *functions* are apart if the functions' images are apart. For example, two functions from names to names are apart if their images (name sets) are apart.

2.4 Static effects and types for dedup

Having seen part of a dynamic execution, we consider a static view of dedup, how Fungi enforces global uniqueness for it, and how Fungi permits the programmer to express and enforce the local uniqueness invariants that support global uniqueness.

Global uniqueness: Static effects for dedup. Returning to Fig. 2, the index term declaration **idxtm** Dedup defines a function from name sets to name sets, of sort **NmSet** $\longrightarrow$ **NmSet**. Given the names in dedup's input list, the name set function Dedup gives an overapproximation of dedup's *write set*—the set of names written by executing dedup on an input list associated with the given name set. This name set function Dedup appears in the type of dedup, defining the write set in terms of X1 as $\triangleright$[Dedup X1].[1] This annotation says that Dedup is a static abstraction of the dynamic allocation effects in the body of dedup.

As explained in detail above, dedup uses each input name x (drawn from name set X1) three times. However, in each of these uses x is composed with other name constants, resulting in unique global names. The three uses are as follows.

(1) *Allocate a new path in the trie.* In aggregate, these allocations write names in the set Insert X1, but with the name constant @t prepended.
(2) *Allocate a recursive thunk.* In aggregate, these allocations write names in X1, but with the name constant @dd prepended.
(3) *Allocate an output list cell.* In aggregate, these allocations write *some* names in X1 (for names of non-duplicated input list elements), but with the name constant @r prepended.

These three terms appear in the body of Dedup. To describe pointwise binary name combination over pairs of name *sets*, Fungi uses the notation $\bullet$. (Above, we write "$\cdot$" and "$\bullet$" for binary combination of name constants and name values, respectively.) Using the apart name set operator $\perp$, the body of Dedup combines these three (disjoint) subsets, simultaneously asserting that they stand apart.

[1] Our full type system also tracks *read sets*, and checks that the read and write sets are in harmony: it is not possible to read a location before it has been allocated.

To see why these terms indeed stand apart, consider the following expansion, where we expand the definition of Dedup over $\{\mathfrak{n}_2\}$, to account for the write set of the $\mathfrak{n}_2$ tile only (Fig. 3):

$$
\begin{array}{lcccc}
\mathtt{Dedup}\,\{\mathfrak{n}_2\} & = & \{\mathbf{t}\}\bullet(\mathtt{Insert}\,\{\mathfrak{n}_2\}) & \perp\ \{\mathbf{dd}\}\bullet\{\mathfrak{n}_2\}\ \perp & \{\mathbf{r}\}\bullet\{\mathfrak{n}_2\} \\
 & = & \{\mathbf{t}\}\bullet\{\mathfrak{n}_2\}\bullet\mathtt{Nat} & \perp\ \{\mathbf{dd}\cdot\mathfrak{n}_2\}\ \perp & \{\mathbf{r}\cdot\mathfrak{n}_2\} \\
 & = & \{\mathbf{t}\}\bullet\{\mathfrak{n}_2\}\bullet\mathtt{Succ}^*\{\mathtt{Zero}\} & \perp & \{\mathbf{dd}\cdot\mathfrak{n}_2, \mathbf{r}\cdot\mathfrak{n}_2\} \\
 & = & \{\mathbf{t}\}\bullet\{\mathfrak{n}_2\}\bullet\,(\mathtt{Succ}^*\{\mathtt{Succ(Zero)}\}\perp\{\mathtt{Zero}\}) & \perp & \{\mathbf{dd}\cdot\mathfrak{n}_2, \mathbf{r}\cdot\mathfrak{n}_2\} \\
 & = & \{\mathbf{t}\}\bullet\{\mathfrak{n}_2\}\bullet\mathtt{Succ}^*\{\mathtt{Succ(Zero)}\} & \perp & \{\mathbf{t}\cdot\mathfrak{n}_2\cdot\mathtt{Zero}, \mathbf{dd}\cdot\mathfrak{n}_2, \mathbf{r}\cdot\mathfrak{n}_2\} \\
 & = & \multicolumn{3}{c}{\{\mathbf{t}\}\bullet\{\mathfrak{n}_2\}\bullet\mathtt{Succ}^*\{\mathtt{Succ}(\mathbf{4})\}\perp\{\mathbf{t}\cdot\mathfrak{n}_2\cdot\mathbf{4}, \mathbf{t}\cdot\mathfrak{n}_2\cdot\mathbf{3}, \mathbf{t}\cdot\mathfrak{n}_2\cdot\mathbf{2}, \mathbf{t}\cdot\mathfrak{n}_2\cdot\mathbf{1}, \mathbf{t}\cdot\mathfrak{n}_2\cdot\mathbf{0}, \mathbf{dd}\cdot\mathfrak{n}_2, \mathbf{r}\cdot\mathfrak{n}_2\}}
\end{array}
$$

The definition of `Insert` uses `Nat`, an infinite set defined by Kleene closure: $\mathtt{Succ}^*\{\mathtt{Zero}\}$. Sec. 2.5 explains this definition and the corresponding implementation of `insert`, but note that the "unrolled" set includes the five names that appear in Fig. 3 that are each based on $\mathfrak{n}_2$, with **t** prepended and **0**–**4** appended. We use decimal notation in place of the actual unary `Zero` and `Succ`.

As this expansion shows, the names in the image of Dedup $\{\mathfrak{n}_2\}$ are pairwise distinct: we can distinguish them by their prefixes (**t**, **dd** and **r**), or—for those with the common prefix **t**—by their distinct suffix **0**–**4**.

What about the other tiles, for input positions $\mathfrak{n}_1$, $\mathfrak{n}_3$ and $\mathfrak{n}_4$? Global uniqueness for the entire execution of dedup rests on the assumptions of local uniqueness for the input list and input trie, e.g., that $\mathfrak{n}_2$ is distinct from all other names, which are also pairwise distinct. Next, we explain how the Fungi programmer establishes and maintains the local uniqueness invariants.

Local uniqueness: Type indices for dedup. The Fungi programmer encodes local uniqueness invariants by attaching *apartness constraints* to the type indices used in the definitions of data structures and functions. Consider the type indices for the two (user-defined) data structures used by dedup, linked lists and hash tries. The invariants expressed in the types are also useful for many other functional algorithms.

The programmer defines `List` and `ListNode` recursively, giving a reference cell at the head of each list and recursive sub-list (Fig. 2, right). Though not shown, `TrieNode` is defined similarly. The type indices enforce that, in each structure, each name appears at most once; but names may be shared across different structures.

In the type for Cons, the quantifier for name sets X1 and X2 includes the constraint X1⊥X2, which says that X1 and X2 are apart (disjoint). These indices appear in the types of the Cons cell's name (X1), and its list tail (X2). Consequently, to form lists inductively, the constraint X1⊥X2 must hold, showing that each additional Cons cell name is distinct from the others already in its tail.

The type indices for `insert` are similar to those of Cons. They express a similar function in terms of name sets: stating that the resulting structure (a trie) contains an additional name (in X1) not present in the input structure (name set X2). The type for `Nil` allows any name set (a safe overapproximation), since `Nil` contains no concrete names at runtime. Similarly, the type for an empty trie (not shown) allows any name set.

Turning to the type signature of dedup, it includes the apartness constraint X1⊥X2, encoding the invariant that the type indices for dedup's input structures (name sets X1 and X2) are apart. The codomain of the type, `List[X1]`, says that the resulting list contains the same names as the input list. The type system uses the apartness constraints within the types of Cons and `insert` to show that dedup's apartness constraint holds for the recursive invocation of dedup.

Intuitively, that invocation moves name x ($\mathtt{x} = \mathfrak{n}_2$ in Fig. 3) from the head of input list `l` to the accumulated trie `tx`, maintaining the pairwise apartness of names in each of the two structures. In terms of Fig. 3, the inductive reasoning about dedup's invariants goes as follows. By assumption, the name sets of `l` and `t` are apart. (In Fig. 3, the name set of `l` is $\{\mathfrak{n}_2, \mathfrak{n}_3, \mathfrak{n}_4\}$, and the name set of

```
type TrieNode : NmSet ⇒ Type
  Emp :∀X:NmSet. TrieNode[X]
  Leaf:∀X:NmSet.
    Nm[X] → Nat → TrieNode[X]
  Bin :∀X1⊥X2:NmSet.
    Trie[X1] → Trie[X2] →
    TrieNode[X1⊥X2]

nmtm Zero : Nm = leaf
nmtm Succ : Nm → Nm = λx.⟪leaf,x⟫
idxtm Gte : Nm → NmSet = λx.Succ*{x}
idxtm Nat : NmSet = Gte Zero
idxtm Insert X = X•Nat

insert x y t = insrec x y t 0 Zero
```

```
insrec : ∀X1⊥X2:NmSet. ∀m:Nm.
  Nm[X] → Nat → Trie[X2] →
  Nat → Nm[{m}] → Trie[X1⊥X2]
  ▷[X•Gte m]

insrec x y t i ni = if i = 4 then
  ref ⟨x•ni⟩ (Leaf x y)
else
 let (j, nj) = (i+1, Succ ni)
 let (txl, txr) =
  if hash_bit y i then
   (insrec x y (left t) j nj, right t)
  else
   (left t, insrec x y (right t) j nj)
 ref ⟨x•ni⟩ (Bin txl txr)
```

Fig. 4. Types and implementation of hash tries; insrec illustrates general recursion with named effects.

t is $\{n_1\}$.) In the Cons branch, the apartness constraint in the type signature for Cons provides that the name x at the Cons cell is apart from the names of the list tail ys, if any. (In Fig. 3, recall that x = n_2, and the names in ys consist of n_3 and n_4.) The type signature for insert provides that the names of the output trie consist of the existing names from the existing trie, along with the new name for the inserted element. In Fig. 3, the inserted trie tx contains the names n_1 and n_2.

Putting these facts together, in the recursive invocation of thunk ddys, we have that the names of the list tail ys (n_3 and n_4) and those of the updated trie tx (n_1 and n_2) are apart.

Static reasoning: To statically enforce both global and local uniqueness, Fungi uses decision procedures to determine whether (static approximations of) name sets are apart. When it needs to prove such an assertion, but decides otherwise, it tells the programmer that the name sets in question—describing either global effects or local type indices—cannot be proven to be apart. For instance, if the programmer mistakenly passed l instead of ys in the recursive call, the inductive invariant would not hold: the names of l and tx overlap at name x. As a result, Fungi would report that it cannot show the invariant for the recursive call.

Below, we consider insert in more depth (Sec. 2.5) before exploring other possible uniqueness errors within dedup (Sec. 2.6),

2.5 Helper function insert

Fig. 4 shows the Fungi programmer's implementation of insert, in terms of a recursive function insrec (right column). The left column gives the type definition of TrieNode, whose use of indices is similar to ListNode from Fig. 2. Below this definition, the programmer defines various name and index terms, culminating in the definition of Insert, which gives the write set for insert, just as Dedup did for dedup in Fig. 2.

Recall that dedup used structural recursion over a list with a name at each recursive position (Cons cell). Here, insrec illustrates a pattern of naming allocations within *general recursion*. The insert function takes a name, an element (natural number) and a trie; it returns the hash trie obtained by hashing the given element and inserting it into the given trie. In addition to the updated trie, insert returns a boolean indicating whether the element was already present in the trie (but with a distinct name). For clarity, we discuss a simpler variant that only returns the updated trie; the other variant is similar, with similar allocation effects.

class of mistake	the dedup programmer…	apartness failure
missing tag	forgets `@dd` and/or `@r`	$\nvdash \lambda x.(\mathbf{dd}\cdot x) \perp \mathtt{id}$ and $\nvdash \lambda x.(\mathbf{r}\cdot x) \perp \mathtt{id}$
prefix/suffix mismatch	uses `@r` as suffix, or forgets `@t`	$\nvdash \lambda x.(\mathbf{dd}\cdot x) \perp \lambda x.(x\cdot \mathbf{r})$
		$\nvdash \lambda x.(\mathbf{dd}\cdot x) \perp \lambda x.(x\cdot \mathbf{0})$
"overlapping primes"	defines `Insert X = Succ*X`	$X : \mathbf{NmSet} \nvdash \mathsf{Succ}(\mathsf{Succ}(X))\perp\mathsf{Succ}(X)$

Table 2. Naming mistakes, with examples from dedup and their associated apartness failure

To allocate a new trie path, the Fungi programmer uses names to identify each allocation. Rather than use names from an input structure, as with the structural recursion of dedup, `insert` generates name sets (statically) with Kleene closure: repeated application of a name function $\mathsf{Succ}^*\{x\}$ to an initial set (Fig. 4). By defining write sets in this way, we can name any allocations within general recursion based on each allocation's (complete) path in the recursive call graph. Since `insert` recurs only once, there is a single chain of calls and a natural number suffices to name each call.

The implementation of `insrec` (dynamically) computes the sequence of names, starting from `Zero`. (As Fig. 3 illustrates, the last computed name corresponds to 4.) In the final iteration, `insert` creates a leaf node holding the inserted element `y`, and its associated name `x`. In a more complex structure, we would handle hash collisions by creating linked lists at these leaf positions; for simplicity, we assume here that hash collisions are impossible.

The index function `Gte` gives the inductive invariant for `insrec`: Every numeral suffix written by the recursive call to `insrec` is greater than the one written by the current iteration of the loop, `ni`. Recursive iterations will use `nj`, or some larger numeral. While natural numbers are not built-in to Fungi's type index system, the programmer encodes the "greater than" constraint using Fungi's notion of apartness.

2.6 Apartness failures violate global uniqueness

As explained above, Fungi enforces global uniqueness and local uniqueness by statically reasoning about apartness. In writing dedup without the Fungi type system, it is easy to make naming mistakes that violate an apartness constraint—breaking local uniqueness, global uniqueness, or both.

In Table 2, we list three classes of naming mistakes, showing concrete examples in the context of dedup and the apartness constraint that does not hold ($\nvdash$). We explain each mistake in more detail, to see why it violates apartness, sometimes on very specific (and unlikely) inputs. It is easy to overlook these mistakes; the authors have made all of them.

To see the problem with a missing tag, consider mapping the name set $\{\mathbf{0}, \mathbf{0}\cdot\mathbf{1}\}$ by `id` (the identity function) and by $\lambda x.x\cdot\mathbf{1}$: The images overlap on name $\mathbf{0}\cdot\mathbf{1}$, since the two names in the input set are already related by the second function $\lambda x.x\cdot\mathbf{1}$, which `id` fails to distinguish by adding any tag of its own. By contrast, consider mapping the same name set by two *apart* name functions, $\lambda x.x\cdot\mathbf{0}$ and $\lambda x.x\cdot\mathbf{1}$; the two images are disjoint ($\{\mathbf{0}\cdot\mathbf{0}, (\mathbf{0}\cdot\mathbf{1})\cdot\mathbf{0}\} \perp \{\mathbf{0}\cdot\mathbf{1}, (\mathbf{0}\cdot\mathbf{1})\cdot\mathbf{1}\}$).

To see the problem with prefix/suffix mismatches, consider mapping the name set $\{\mathbf{0}, \mathbf{1}\}$ by $\lambda x.\mathbf{0}\cdot x$ and $\lambda x.x\cdot\mathbf{1}$; the two images overlap on name $\mathbf{0}\cdot\mathbf{1}$, since the two functions disagree about how they distinguish names in the input set.

Finally, to see the problem with what we call "overlapping primes", consider mapping the name set $\{\mathbf{0}, \mathbf{1}\}$ by Succ and $\mathsf{Succ} \circ \mathsf{Succ}$: the images overlap at name $\mathbf{2}$. The problem is similar to the missing tag problem, but at the level of name sets. Overlapping primes may involve Kleene closure, e.g., $X : \mathbf{NmSet} \nvdash \mathsf{Succ}^*(\mathsf{Succ}(X))\perp X$. There is a critical difference between this apartness violation and the apartness *property* used by `insert` and `insrec` in Sec. 2.5, $x : \mathbf{Nm} \vdash \mathsf{Succ}^*(\{\mathsf{Succ}(x)\})\perp\{x\}$. Like the violation above, this valid apartness property also involves Kleene closure, but the "seed"

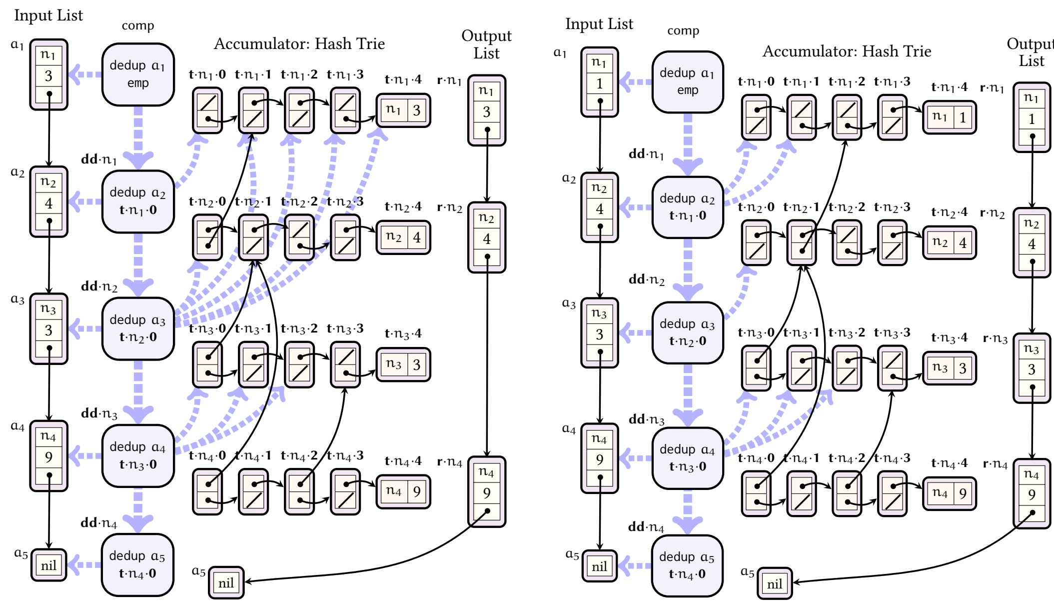

Fig. 5. Distinct runs of dedup on (similar) input lists $[3, 4, 3, 9]$ (left side) and $[1, 4, 3, 9]$ (right side).

set used by `insrec` is a *single* unknown name. We call this kind of mistake “overlapping primes” because an analogous issue arises when manually “freshening” meta-variables in a proof: adding a prime to each existing variable does not work when an existing variable is the prime of another. For example, priming the variable A in the set $\{A, B, A'\}$ clashes with the existing A'.

2.7 Global uniqueness implies correct change propagation

The metatheory of the Fungi type system considers one run at a time. Within each run, it ensures global uniqueness, the prerequisite for change propagation to ensure from-scratch consistency: When global uniqueness holds, the outcome of change propagation for any iteration is always consistent with the outcome of a from-scratch run on the current input.

Using names to compare similar from-scratch runs. Because change propagation is from-scratch consistent, we can predict its time complexity (and other dynamic behavior) by comparing two from-scratch runs and seeing where they differ. Due to their use of names, the two runs similarities and differences can be identified precisely, by name. Change propagation, described in detail below, attempts to exploit these similarities to reuse past work wherever possible.

Fig. 5 shows two full runs of dedup on (similar) input lists $[3, 4, 3, 9]$ and $[1, 4, 3, 9]$, stored in identical list cells ($a_1 - a_5$) and initial recursive thunk (**comp**). The tile in Fig. 3 is consistent with the left run. The left run consists of two occurrences of element 3, at logical positions n_1 and n_3; in the right run, logical position n_1 instead contains the (unique) element 1.

The later recursive calls depend on the trie paths allocated in earlier calls. For instance, by carefully comparing the left and right runs' allocated trie paths rooted at $\mathbf{t}{\cdot}n_1{\cdot}\mathbf{0}$, we see that the hashes of 3 and 1 consist of inverted bits: all of the pointers have “flipped” between left and right. Moving downward in the figure, the allocated trie paths rooted at $\mathbf{t}{\cdot}n_2{\cdot}\mathbf{0}$ differ at that name, and at $\mathbf{t}{\cdot}n_2{\cdot}\mathbf{1}$, but then “sync up” at $\mathbf{t}{\cdot}n_2{\cdot}\mathbf{2}$—$\mathbf{t}{\cdot}n_2{\cdot}\mathbf{4}$. The allocated trie paths rooted at $\mathbf{t}{\cdot}n_3{\cdot}\mathbf{0}$ and $\mathbf{t}{\cdot}n_4{\cdot}\mathbf{0}$ are the same in the two runs.

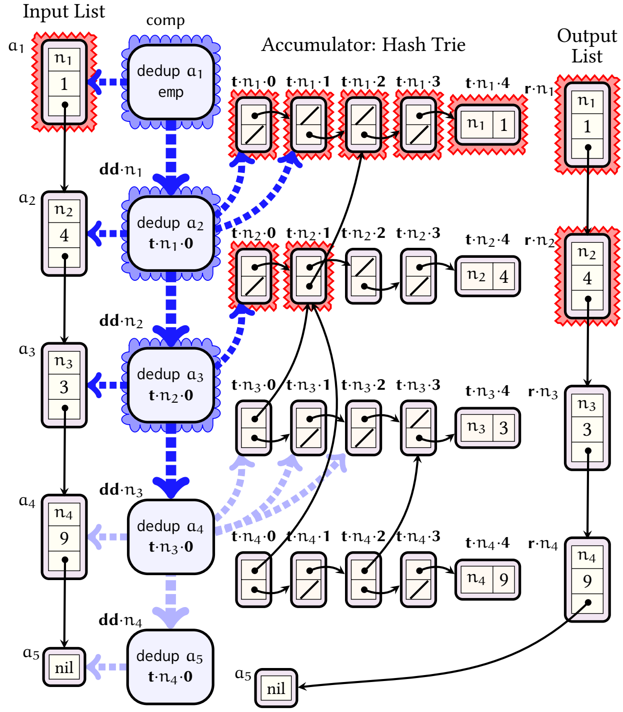

Fig. 6. Change propagation uses names to identify the correspondences between subsequent runs of dedup and efficiently exploit the similarities between these inputs, outputs and intermediate structures (hash tries).

Because of the input list's logical position names (n_1—n_4), the output list uses identical addresses in the left- and right-hand runs, where they overlap. The right-hand run's list contents are similar, with three (necessary) exceptions: (a) the element at logical position n_1 is changed to 1; (b) 3 appears at logical position n_3 (and pointer name $\mathbf{r}\cdot n_3$), whereas the left-hand run had a duplicate 3 at position n_3; (c) the tail pointer in the output `Cons` cell $\mathbf{r}\cdot n_2$ differs, since position n_3 was absent in the left-hand run.

Change propagation. Fig. 6 considers the behavior of using change propagation where the left run of Fig. 5 happens first, followed by an input change (at a_1), that precipitates change propagation updating this dependence graph to be from-scratch consistent with the right run. As explained above, the two runs in Fig. 5 differ at certain allocated names; change propagation selectively re-executes thunks in the dependence graph in an order that is consistent with a from-scratch run on the *current* input (in this case, the right run of Fig. 5). We indicate the re-executed thunks with an additional (blue) border pattern.

Change propagation re-executes thunk **comp** first, since it observes the changed input list cell a_1 that replaces the first 3 with 1. As described above, the new element 1 hashes differently, resulting in a different pattern of pointers in this trie path rooted at $\mathbf{t}\cdot n_1\cdot\mathbf{0}$.

We indicate overwritten (and changed) reference cells with an additional (red) border pattern. Fungi dynamically records thunks that depend on changed reference cells, and avoids reusing their results without first applying the change propagation algorithm to their dependence graphs. For this reason, change propagation next re-executes $\mathbf{dd}\cdot n_1$.

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

Fig. 7. Syntax of expressions

When re-executed, $\mathbf{dd}{\cdot}n_1$ changes *some* trie names with its overwrites, but not all of them (as described above). Next, it re-forces $\mathbf{dd}{\cdot}n_2$, which re-executes (due to the trie overwrites).

When a re-execution results in behavior that is the same as the prior run, the frontier of change propagation may end, as with thunk $\mathbf{dd}{\cdot}n_2$. Its allocations overwrite the prior reference cells with *identical* values. It (re-)forces $\mathbf{dd}{\cdot}n_3$, whose local effects are unaffected by the input change (either directly or indirectly). In this case, Fungi reuses the cached result of this (unaffected) thunk.

Next, control returns to $\mathbf{t}{\cdot}n_1{\cdot}\mathbf{0}$, which overwrites its output cell's tail pointer with the inserted (and new) cell at $\mathbf{r}{\cdot}n_3$. Finally, control returns to **comp**, which overwrites $\mathbf{r}{\cdot}n_1$ with the changed input value 1, but returns the same result, $\mathbf{r}{\cdot}n_1$. If **comp** were occuring in the context of more recursive calls in a longer input list, these earlier calls would be unaffected, and not re-executed.

In summary, the change propagation behavior described above critically relies on (unique) names to bring the initial and updated runs into a correspondence that it can efficiently exploit. Unique names are generally *necessary* for efficient (stable) change propagation, but not alone sufficient. In particular, the Fungi type-and-effect system enforces global uniqueness by reasoning about a single from-scratch execution, not the relationship between two (or more) similar executions on similar inputs. In Sec. 8, we discuss connections to (relational) cost semantics for incremental computation.

3 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.

3.1 Values and Expressions

Fig. 7 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 ($\mathtt{split}$, $\mathtt{case}$ and $e\,v$, respectively); let-binding (which evaluates e_1 to $\mathtt{ret}(v)$ and substitutes v for x in e_2); introduction ($\mathtt{thunk}$) and elimination

Names	m, n	$::=$	$\mathtt{leaf}$	leaf name
(binary trees)			$\mid \langle\!\langle n_1, n_2 \rangle\!\rangle$	binary name composition
Name terms	M, N	$::=$	$n \mid \langle\!\langle M_1, M_2 \rangle\!\rangle$	literal names, binary name composition
(STLC+names)			$\mid a \mid \lambda a.\, M \mid M(N)$	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 11

Fig. 8. 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

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

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

$\boxed{M \Downarrow_M V}$ 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. 9. Sorting and evaluation rules for name terms M

(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.

This syntax is similar to Adapton [Hammer et al. 2015]; we add the notion of a *name function*, which captures the idea of a namespace and other transformations on names. The $\mathtt{scope}(v, e)$ 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.

3.2 Names

Figure 8 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 $\mathtt{leaf}$ 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 $\gamma \overset{\mathbf{Nm}}{\Rightarrow} \gamma$ for (name term) functions.

The rules for name sorting $\Gamma \vdash M : \gamma$ are straightforward (Figure 9), as are the rules for name term evaluation $M \Downarrow_M V$ (Figure 9). 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.

4 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 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: $\forall a : \textbf{Nm}.\ \forall X : \textbf{NmSet}.\ (a \in X) \supset \big(A[a] \to B[f(a)]\big)$. 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 $\supset$: $\forall X : \textbf{NmSet}.\ A[X] \to B[f(X)]$. 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 are 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 *overapproximation* of names: the proper DML type $\forall Y : \textbf{NmSet}.\ \forall X : \textbf{NmSet}.\ (Y \subseteq X) \supset \big(A[Y] \to B[f(Y)]\big)$ can be expressed by $\forall X : \textbf{NmSet}.\ A[X] \to B[f(X)]$.

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.

4.1 Index Level

Figure 10 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**.

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 is common in the types

Index exprs.	$i, j,$	$::=$	a	index variable
	$X, Y, Z,$		$\mid \{N\}$	singleton name set
	R, W		$\mid \emptyset \mid X \perp Y$	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. 10. Syntax of indices, name set sort

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. 11. Syntax of kinds, effects, and types

that we believe programmers need, our effects discipline requires non-disjoint union $X \cup Y$, so we include it as well.

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

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

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

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

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

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

Fig. 12. Sorts statically classify name terms M, and the name indices i that index types

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 $\mathsf{prj}_b i$ 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 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 $\overset{\textbf{Nm}}{\to}$ from the name term sorts (Figure 8). The sort **NmSet** (Figure 10) classifies indices that are name sets. The function space $\overset{\text{idx}}{\to}$ 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 12 are straightforward, but rule 'sort-sep-union' includes a premise $\mathit{extract}(\Gamma) \Vdash X \perp Y : \textbf{NmSet}$, 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, $\{n_1\} \perp \{n_2\} : \textbf{NmSet}$ 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 F.6 and F.7).

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

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

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

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

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

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

$$\frac{\Gamma \vdash i : \gamma \qquad \mathit{extract}(\Gamma) \Vdash [i/a]P \qquad \Gamma \vdash v : [i/a]A}{\Gamma \vdash \mathsf{pack}(a.v) : (\exists a : \gamma \mid P.\, A)}\,\text{vtype-}\exists\text{IndexIntro}$$

Fig. 13. Value typing

4.2 Kinds

We use a simple system of *kinds* K (Figure 21 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.

Kind $\gamma \Rightarrow K$ classifies type expressions parametrized by an index. For example, the List type constructor from Section 2 takes a name set: $\mathsf{List}[X]$, so List has kind $\mathbf{NmSet} \Rightarrow \mathsf{type}$. A more general Seq type would also track its pointers (not just its names), and permit any element type, and would thus have kind $\mathbf{NmSet} \Rightarrow (\mathbf{NmSet} \Rightarrow (\mathsf{type} \Rightarrow \mathsf{type}))$.

4.3 Effects

Effects are described by $\langle W; R\rangle$, meaning that the associated code may write names in W, and read names in R. (To simplify the example in the overview, we omitted the read set.)

Effect sequencing (Figure 14) 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, "E after ϵ", combines "clusters" of effects: $(C \rhd \langle\{n_2\}; \emptyset\rangle)$ after $\langle\{n_1\}; \emptyset\rangle = C \rhd (\langle\{n_1\}; \emptyset\rangle$ then $\langle\{n_2\}; \emptyset\rangle) = C \rhd \langle\{n_1, n_2\}; \emptyset\rangle$. 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.

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

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

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

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

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

$$\frac{\Gamma \vdash (\epsilon_1 \,\mathsf{then}\, \epsilon_2) = \epsilon}{\Gamma \vdash \big((C \rhd \epsilon_2) \,\mathsf{after}\, \epsilon_1\big) = (C \rhd \epsilon)} \qquad \frac{\Gamma \vdash (E \,\mathsf{after}\, \epsilon) = E'}{\begin{array}{c} \Gamma \vdash (\forall \alpha : K.\, E) \,\mathsf{after}\, \epsilon \;=\; (\forall \alpha : K.\, E') \\ \Gamma \vdash (\forall a : \gamma \mid P.\, E) \,\mathsf{after}\, \epsilon \;=\; (\forall a : \gamma \mid P.\, E') \end{array}}$$

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

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

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

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

$$\frac{\Gamma, x : A \vdash^{M} e : E}{\Gamma \vdash^{M} (\lambda x.\, e) : \big((A \to E) \rhd \langle \emptyset; \emptyset \rangle\big)}\ \text{lam} \qquad \frac{\begin{array}{l} \Gamma \vdash (E \,\mathsf{after}\, \epsilon_1) = E_1 \\ \Gamma \vdash^{M} e : \big((A \to E) \rhd \epsilon_1\big) \qquad \Gamma \vdash v : A \end{array}}{\Gamma \vdash^{M} (e\ v) : E_1}\ \text{app}$$

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

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

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

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

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

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

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

Fig. 14. Computation typing

4.4 Types

The value types (Figure 11), 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 $A\mathsf{[i]}$ of a type to an index; the application $A\ B$ of a type A (e.g. a type constructor d) to a type B; the type $\mathsf{Nm[i]}$; and a singleton type $(\mathbf{Nm} \overset{\mathbf{Nm}}{\Rightarrow} \mathbf{Nm})\mathsf{[}M\mathsf{]}$ 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 "li**F**ts" 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 A.3.

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 13) 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.

Computation typing rules. Many of the rules that assign computation types (Figure 14) 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).

4.5 Subtyping

As discussed above, our type system can overapproximate names. The type $\mathsf{Nm[}X\mathsf{]}$ 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 $\mathsf{Nm[}X_1\mathsf{]}$ to a function expecting $\mathsf{Nm[}X_1 \perp X_2\mathsf{]}$.

For space reasons, the subtyping rules are given and explained in the appendix (Sec. A.1).

4.6 Bidirectional Version

The typing rules in Figures 13 and 14 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 readily applicable. For example, it is not obvious when to apply etype-∀Intro, or how to solve unification constraints over names and name sets.

Bidirectional typing [Pierce and Turner 1998] alternates between checking an expression against a known type (e.g. from a type annotation) and synthesizing a type from an expression. Since

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

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

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

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

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

Fig. 15. Excerpt from the dynamic semantics (see also Figure 19)

checking rules utilize the given type, bidirectional typing is decidable for a wide range of rich type systems; see the citations in Dunfield and Krishnaswami [2013]. Therefore, we formulate bidirectional typing rules that are decidable and directly give rise to an algorithm.

For space reasons, this system is presented in the supplementary material (Appendix C). We prove in Appendix D that our bidirectional rules are sound and complete with respect to the type assignment rules in this section:

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

Completeness (Thms. D.2, D.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*.)

5 DYNAMIC SEMANTICS

Name terms. Recall Fig. 9 (Sec. 3.2), 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 G.9).

Program expressions (Figure 15). 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 with unique names.

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 ⇓-ref and ⇓-thunk either extend or overwrite the store, depending on whether the allocated pointer name is unique 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 programs without naming errors overwrite the (old) store/graph, as discussed in Sec. 1.)

While motivated by incremental computation, we are interested in the allocation effects of a single run, not change propagation between runs. 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 we modeling change propagation, rules $\Downarrow$-ref, $\Downarrow$-thunk, $\Downarrow$-get and $\Downarrow$-force would create dependency edge structure that we omit here. (These edges relate the current node with the node being observed.)

6 METATHEORY: TYPE SOUNDNESS AND UNIQUE NAMES

In this section, we prove that our type system is sound with respect to evaluation: Every well-typed, terminating program produces a terminal computation of the program's type, the set of dynamic allocations match the program's static approximation, and each allocation is globally unique. Def. 6.2 defines which evaluation derivations have *precise effects* matching the requirements above.

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 6.1 (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 6.2 (Precise effects). *Given an evaluation derivation* $\mathcal{D}$, *we write* $\mathcal{D}$ reads R writes W *for its* precise effects *(Figure 20 in the appendix).*

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

We write $\langle W'; R' \rangle \preceq \langle W; R \rangle$ to mean that $W' \subseteq W$ and $R' \subseteq R$. For proofs, see Appendix B.

Theorem 6.1 (Subject Reduction). *If* Γ_1 store-type *and* $\Gamma_1 \vdash M : \mathbf{Nm} \overset{\mathsf{Nm}}{\Rightarrow} \mathbf{Nm}$ *and* $\Gamma_1 \vdash^{M} e : C \rhd \langle W; R \rangle$ *and* $\vdash S_1 : \Gamma_1$ *and* $\mathcal{D}$ *derives* $S_1 \vdash^{M}_{m} e \Downarrow S_2; t$ *then there exists* $\Gamma_2 \supseteq \Gamma_1$ *s.t.* 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* $\langle W_{\mathcal{D}}; R_{\mathcal{D}} \rangle \preceq \langle W; R \rangle$.

Our implementation (Sec. 7) follows the change propagation algorithm of Hammer et al. [2015], which has been formalized and proven correct (from-scratch consistent) when Def. 6.2 (precise effects) holds for every program run under consideration—a guarantee of Fungi's type-and-effect system, as stated above.

7 IMPLEMENTATION

7.1 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. C) 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. 2. 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.

We implement an *incremental* semantics for Fungi based on Adapton in Rust, as provided by an existing external library [Adapton Developers 2018]. This library uses the change propagation algorithm(s) of Hammer et al. [2014, 2015]. The implementation of Fungi is documented and publicly available. At present, it consists of about 10K lines of Rust. 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.*

7.2 Ongoing and Future Work

Sec. A.3 discusses a proposal for *imperative* (name) effects in the context of incremental subcomputations that (still) require unique names. Conceivably, future Fungi-based systems could track *reactive* names and their effects, potentially encoding reactive aspects of FRP language semantics [Elliott and Hudak 1997; Wan and Hudak 2000; Cooper and Krishnamurthi 2006; Krishnaswami and Benton 2011; Krishnaswami 2013; Czaplicki and Chong 2013]. In the long term, we intend Fungi as a *target* language for higher-level incremental programming languages.

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 that requires transforming the desired typing rules and their modes [Erdweg et al. 2015; 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 (rule and typing context structure).

8 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 $0 \cdot n = \langle\!\langle \mathtt{leaf},\, n \rangle\!\rangle$ and $1 \cdot n = \langle\!\langle \langle\!\langle \mathtt{leaf},\, \mathtt{leaf} \rangle\!\rangle,\, n \rangle\!\rangle$

share the right subtree $\mathfrak{n}$—which allows programmers to deterministically compute two distinct names from one.

Substructural type systems [O'Hearn 2003; Walker 2005] might seem suitable for statically verifying the correct usage of names. We initially believed that an affine type system would be good for checking global uniqueness, but we abandoned that route. First, sharing between data structures can be essential for efficiency (e.g. a `suffixes` function over a list). Second, while global uniqueness itself seems within the scope of affine typing, the justification for global uniqueness rests on local uniqueness properties that fall outside the scope of affine typing. It is conceivable that some not-yet-invented substructural type system could accomplish our goals, but "off-the-shelf" affine typing is not viable.

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.

Approaches to incremental computation. General-purpose incremental computation techniques use *change propagation* algorithms. Change propagation is a provably sound approach for recomputing the affected output, as the input changes dynamically after an initial run of the program [Acar et al. 2006b; Acar and Ley-Wild 2008; Hammer et al. 2014, 2015].

Our type and effect system 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 some of the work cited above.

Incremental computation can deliver *asymptotic* speedups for certain algorithms [Acar et al. 2007, 2008, 2009; Sümer et al. 2011; Chen et al. 2012], and has even addressed open problems [Acar et al. 2010]. Incremental computing abstractions exist in many settings [Shankar and Bodik 2007; Hammer et al. 2009; Acar and Ley-Wild 2008]. Cai et al. [2014] use *derivatives* in an incremental λ-calculus, which is more restricted than our setting (for example, their calculus lacks rich datatypes). Approaches such as concurrent revisions [Burckhardt et al. 2011], hybrid reactive/imperative programming [Demetrescu et al. 2011], and embedded incremental query languages [Mitschke et al. 2014] constitute alternate approaches to incremental computation, but diverge more markedly from conventional programming languages.

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 dedup from Sec. 2, would require integrating a general notion of names. Without such a notion, constant-sized input changes may cascade, preventing reuse.

Detection of naming errors. Some past systems dynamically detect ambiguous names, either forcing the system to fall back to a non-deterministic name choice [Acar et al. 2006b; 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]. 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).

9 CONCLUSION

We present Fungi, a typed functional language for incremental computation with names. Unlike prior general-purpose languages for incremental computing (Table 1), Fungi's notion of names is formal, general, and statically verified. In particular, Fungi's type-and-effect system permits the programmer to encode (program-specific) local uniqueness invariants about names, and to use these invariants to establish *global uniqueness* for their composed programs, the property of using names correctly. We derive a bidirectional version of the type and effect system, and we have implemented a prototype of Fungi in Rust. 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. 7; 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, Guy E. Blelloch, and Robert Harper. 2002. Adaptive Functional Programming. In *Principles of Programming Languages.* 247–259.

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. 2008. Self-adjusting Computation with Delta ML. In *Advanced Functional Programming.* Springer.

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

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.*

Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. 2014. A theory of changes for higher-order languages: incrementalizing λ-calculi by static differentiation. In *Programming Language Design and Implementation.* ACM Press, 145–155.

Magnus Carlsson. 2002. Monads for Incremental Computing. In *International Conference on Functional Programming.* 26–35.

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, Jana 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.

Yan Chen, Jana Dunfield, Matthew A. Hammer, and Umut A. Acar. 2011. Implicit Self-Adjusting Computation for Purely Functional Programs. In *Int'l Conference on Functional Programming (ICFP '11).* 129–141.

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.*

Camil Demetrescu, Irene Finocchi, and Andrea Ribichini. 2011. Reactive Imperative Programming with Dataflow Constraints. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).*

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

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

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

Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *ICFP.* ACM Press, 263–273.

Sebastian Erdweg, Oliver Bracevac, Edlira Kuci, Matthias Krebs, and Mira Mezini. 2015. 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.

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, Jana 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.

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.*

Ralf Mitschke, Sebastian Erdweg, Mirko Köhler, Mira Mezini, and Guido Salvaneschi. 2014. i3QL: Language-integrated Live Data Views. In *OOPSLA*. ACM Press.

Peter W. O'Hearn. 2003. On bunched typing. *J. Funct. Program.* 13, 4 (2003), 747–796. https://doi.org/10.1017/S0956796802004495

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. 1998. Local Type Inference. In *Principles of Programming Languages*. 252–265. Full version in *ACM Trans. Prog. Lang. Sys.*, 22(1):1–44, 2000.

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.

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.

David Walker. 2005. Substructural Type Systems. In *Advanced Topics in Types and Programming Languages*, B. C. Pierce (Ed.). MIT Press, Chapter 1, 3–43.

Zhanyong Wan and Paul Hudak. 2000. Functional reactive programming from first principles. In *Programming Language Design and Implementation*. ACM Press, 242–252.

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.

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

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

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

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

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

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

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

Fig. 16. 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. 17. Subtyping on computation types

A OMITTED DEFINITIONS, FIGURES, AND REMARKS

A.1 Subtyping

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, $\tau[i] \leq \tau[j]$ 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 $\leq_V$-name (Figure 16) says that $\mathsf{Nm}[X] \leq_V \mathsf{Nm}[Y]$ if $X \subseteq Y$, rather than $X = Y$. Similarly, subtyping for references and thunks ($\leq_V$-ref, $\leq_V$-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 $\leq_V$-$\forall$L combines the Stardust rules ΠL for index-level quantification and $\supset$L for the guarded type [Dunfield 2007, p. 33]. Likewise, our $\leq_V$-$\forall$R combines Stardust's ΠR and $\supset$R.

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* ($\otimes$), 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 *editor* programs, we would need different rules (Section A.3 in the appendix).

In our subtyping rules for computation types (Figure 17), 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.

A.2 Dynamic semantics, read and write sets, sorting and kinding

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

Fig. 18. Store typing: $S \vdash \Gamma$, read "store S typed by Γ".

In Figure 20, we write

$$\mathcal{D} \text{ by } \textit{Rulename (Dlist)} \text{ reads } R \text{ writes } W$$

to mean that rule *Rulename* concludes $\mathcal{D}$ and has subderivations *Dlist*. For example,

$$\mathcal{D} \text{ by } \Downarrow\text{-scope}(\mathcal{D}_0) \text{ reads } R \text{ writes } W$$

provided that $\mathcal{D}$ reads R writes W, where $\mathcal{D}_0$ derives the only premise of $\Downarrow$-scope.

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

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

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

$$
\frac{S_1 \vdash^M_m [v_2/x_2][v_1/x_1]e \Downarrow S_2; e'}{S_1 \vdash^M_m \mathtt{split}((v_1, v_2), x_1.x_2.e) \Downarrow S_2; e'}\ \Downarrow\text{-split}
\qquad
\frac{S_1 \vdash^M_m [v_i/x_i]e_i \Downarrow S_2; e'}{S_1 \vdash^M_m \mathtt{case}(\mathtt{inj}_i\, v, x_1.e_1, x_2.e_2) \Downarrow S_2; e'}\ \Downarrow\text{-case}
$$

$$
\frac{S_1 \vdash^M_m [v/x]e \Downarrow S_2; e'}{S_1 \vdash^M_m \mathtt{vunpack}(\mathtt{pack}(a.v), b.x.e) \Downarrow S_2; e'}\ \Downarrow\text{-unpack}
$$

$$
\frac{S_1 \vdash^M_m e_1 \Downarrow S_1'; \mathtt{ret}(v) \qquad S_1' \vdash^M_m [v/x]e_2 \Downarrow S_2'; e_2'}{S_1' \vdash^M_m \mathtt{let}(e_1, x.e_2) \Downarrow S_2'; e_2'}\ \Downarrow\text{-let}
$$

$$
\frac{S_1 \vdash^M_m e_1 \Downarrow S_1'; \lambda x.\, e_2 \qquad S_1 \vdash^M_m [v/x]e_2 \Downarrow S_2'; e_2'}{S_1' \vdash^M_m e_1\ v \Downarrow S_2'; e_2'}\ \Downarrow\text{-app}
$$

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

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

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

Fig. 19. Dynamic semantics, complete

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

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

Fig. 20. Read- and write-sets of a non-incremental evaluation derivation

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

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

$$\frac{}{\Gamma \vdash \mathsf{unit} : \mathsf{type}}\ \text{k-unit} \qquad \frac{\Gamma \vdash i : \mathbf{NmSet}}{\Gamma \vdash \mathsf{Nm}[i] : \mathsf{type}}\ \text{k-name} \qquad \frac{\Gamma \vdash i : \mathbf{NmSet} \qquad \Gamma \vdash A : \mathsf{type}}{\Gamma \vdash (\mathsf{Ref}[i]\, A) : \mathsf{type}}\ \text{k-ref}$$

$$\frac{\begin{array}{c}\Gamma \vdash i : \mathbf{NmSet}\\ \Gamma \vdash E\ \mathsf{efftype}\end{array}}{\Gamma \vdash (\mathsf{Thk}[i]\, E) : \mathsf{type}}\ \text{k-thk} \qquad \frac{\begin{array}{l}\Gamma \vdash A : (\mathsf{type} \Rightarrow K)\\ \Gamma \vdash B : \mathsf{type}\end{array}}{\Gamma \vdash (A\ B) : K}\ \text{k-app-type} \qquad \frac{\begin{array}{l}\Gamma \vdash A : (\gamma \Rightarrow K)\\ \Gamma \vdash i : \gamma\end{array}}{\Gamma \vdash A[i] : K}\ \text{k-app-index}$$

$$\frac{\Gamma, a : \gamma \vdash P\ \mathsf{prop} \qquad \Gamma, a : \gamma \vdash A : \mathsf{type}}{\Gamma \vdash (\forall a : \gamma \mid P.\, A) : \mathsf{type}}\ \text{k-all} \qquad \frac{\Gamma, a : \gamma \vdash P\ \mathsf{prop} \qquad \Gamma, a : \gamma \vdash A : \mathsf{type}}{\Gamma \vdash (\exists a : \gamma \mid P.\, A) : \mathsf{type}}\ \text{k-exists}$$

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

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

$\boxed{\Gamma \vdash \epsilon\ \mathsf{wf\text{-}effects}}$ Under Γ, effects ϵ are well-formed

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

$\boxed{\Gamma \vdash P\ \mathsf{prop}}$ Under Γ, proposition P is well-formed

$$\frac{}{\Gamma \vdash \mathsf{tt}\ \mathsf{prop}} \qquad \frac{\Gamma \vdash P_1\ \mathsf{prop} \qquad \Gamma \vdash P_2\ \mathsf{prop}}{\Gamma \vdash (P_1\ \mathbf{and}\ P_2)\ \mathsf{prop}} \qquad \frac{\Gamma \vdash i : \gamma \qquad \Gamma \vdash j : \gamma}{\begin{array}{c}\Gamma \vdash (i \perp j : \gamma)\ \mathsf{prop}\\ \Gamma \vdash (i \equiv j : \gamma)\ \mathsf{prop}\end{array}}$$

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

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

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

Fig. 21. Kinding and well-formedness for types and effects

A.3 Remarks

Why distinguish computation types from types-with-effects? Can we unify computation types C and types-with-effects E? Not easily. We have two computation types, $\mathbf{F}$ and $\to$. For $\mathbf{F}$, the expression being typed could create a thunk, so we must put that effect somewhere in the syntax. For $\to$, applying a function is (per call-by-push-value) just a "push": the function carries no effects of its own (though its codomain may need to have some). However, suppose we force a thunked function of type $A_1 \to (A_2 \to \cdots)$ and apply the function (the contents of the thunk) to one argument. In the absence of effects, the result would be a computation of type $A_2 \to \cdots$, meaning that the computation is waiting for a second argument to be pushed. But, since forcing the thunk has the effect of reading the thunk, we need to track this effect in the result type. So we cannot return $A_2 \to \cdots$, and must instead put effects around $(A_2 \to \cdots)$. Thus, we need to associate effects to both $\mathbf{F}$ and $\to$, that is, to both computation types.

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

Where should these wrappers live? We could add $C \rhd \epsilon$ to the grammar of computation types C. But it seems useful to have a clear notion of *the* effect associated with a type. When the effect on the outside of a type is the only effect in the type, as in $(A_1 \to \mathbf{F}\,A_2) \rhd \epsilon$, "the" effect has to be ϵ. Alas, types like $(C \rhd \epsilon_1) \rhd \epsilon_2$ raise awkward questions: does this type mean the computation does ϵ_2 and then ϵ_1, or ϵ_1 and then ϵ_2?

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

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:

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

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.

B OMITTED LEMMAS AND PROOFS

Lemma B.1 (Index-level weakening).

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

Proof. By induction on the given derivation. □

Lemma B.2 (Weakening).

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

Proof. By induction on the given derivation, using Lemma F.1 (Weakening of semantic equivalence and apartness) (for example, in the case for the value typing rule 'name') and Lemma B.1 (Index-level weakening) (for example, in the case for the computation typing rule 'AllIndexElim'). □

Lemma B.3 (Substitution).

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

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

In the presence of subtyping, canonical forms (value inversion) is not entirely straightforward.

Lemma B.4 (Subtyping Weakening). *If* $\Gamma \vdash A \leq_V B$ *then* $\Gamma, \Gamma' \vdash A \leq_V B$ *where* Γ' *consists of* $a : \gamma$ *and* P *assumptions.*

Proof. By induction on the derivation of $\Gamma \vdash A \leq_V B$. In the $\leq_V$-name, $\leq_V$-namefn, $\leq_V$-ref, $\leq_V$-thk, $\leq_V$-$\forall$L and $\leq_V$-$\exists$ cases, use weakening for the relations $\vdash$ and $\Vdash$. □

Lemma B.5 (Subtyping Substitution).
If $\Gamma, a : \gamma, P \vdash A \leq_V B$ *and* $\Gamma \vdash i : \gamma$ *and extract*$(\Gamma) \Vdash P$ *then* $\Gamma \vdash [i/a]A \leq_V [i/a]B$.

Proof. By induction on the derivation of $\Gamma \vdash A \leq_V B$. In the $\leq_V$-ref, $\leq_V$-thk, $\leq_V$-$\forall$L and $\leq_V$-$\exists$ cases, use substitution for the relation $\Vdash$. □

Lemma B.6 (Reflexivity of Subtyping). *For all* Γ *and* A, *it is the case that* $\Gamma \vdash A \leq_V A$.

Proof. Immediate by rule $\leq_V$-refl. □

Lemma B.7 (Transitivity of Subtyping).
If $\Gamma \vdash A_L \leq_V B$ *and* $\Gamma \vdash B \leq_V A_R$ *then* $\Gamma \vdash A_L \leq_V A_R$.

Proof. By simultaneous induction on the two given derivations.
If either derivation is by $\leq_V$-refl, we already have our result.
Consider cases of the rule concluding $\Gamma \vdash A_L \leq_V B$.

- $\leq_V$-$\times$:
  The derivation of $\Gamma \vdash B \leq_V A_R$ must be by $\leq_V$-refl (already handled), $\leq_V$-$\times$ or $\leq_V$-$\forall$R.
  If by $\leq_V$-$\times$, the result follows by using the i.h. twice on the respective subderivations, then applying $\leq_V$-$\times$.
  If by $\leq_V$-$\forall$R, then:

	$A_R = (\forall b : \gamma \mid P.\, A_{R0})$	By inversion ($\leq_V$-$\forall$R)
	$\Gamma, b : \gamma, P \vdash B \leq_V A_{R0}$	$''$
	$\Gamma \vdash A_L \leq_V B$	Given
	$\Gamma, b : \gamma, P \vdash A_L \leq_V B$	By Lemma B.4 (Subtyping Weakening)
	$\Gamma, b : \gamma, P \vdash A_L \leq_V A_{R0}$	By i.h.
	$\Gamma \vdash A_L \leq_V (\forall b : \gamma \mid P.\, A_{R0})$	By $\leq_V$-$\forall$R
☞	$\Gamma \vdash A_L \leq_V A_R$	By above equation

- $\leq_V$-$+$: Similar to the $\leq_V$-$\times$ case.

- $\leq_V$-name, $\leq_V$-ref, $\leq_V$-thk: Similar to the $\leq_V$-$\times$ case, using transitivity of $\subseteq$ at the index level.

- $\leq_V$-namefn: Use transitivity of conv.

- $\leq_V$-$\forall$L:
  By i.h., $\Gamma \vdash [i/a]A_{L0} \leq_V A_R$.
  By $\leq_V$-$\forall$L, $\Gamma \vdash \forall a : \gamma \mid P.\, A_{L0} \leq_V A_R$, which was to be shown.

- $\leq_V$-$\forall$R:
  The other derivation is by either $\leq_V$-refl (already handled) or $\leq_V$-$\forall$L.

	$B = (\forall b : \gamma \mid P.\, B_0)$	By inversion ($\leq_V$-$\forall$R)
	$\Gamma, b : \gamma, P \vdash A_L \leq_V B_0$	$''$
	$\Gamma \vdash i : \gamma$	By inversion ($\leq_V$-$\forall$L)
	$\mathit{extract}(\Gamma) \Vdash [i/b]P$	$''$
	$\Gamma \vdash [i/b]B_0 \leq_V A_R$	$''$
	$\Gamma \vdash [i/b]A_L \leq_V [i/b]B_0$	By Lemma B.5 (Subtyping Substitution)
	$\Gamma \vdash [i/b]A_L \leq_V A_R$	By i.h.
	$[i/b]A_L = A_L$	b not free in A_L
☞	$\Gamma \vdash A_L \leq_V A_R$	By above equation

□

Lemma B.8 (Canonical Forms). *Suppose* Γ store-type *and* $\Gamma \vdash v : A$.

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

Proof. By induction on the derivation of $\Gamma \vdash A \leq_V B$.

(1) Consider cases of the rule concluding $\Gamma \vdash v : A$.

- **Case** unit: By inversion.
- **Case** pair: Impossible because $\Gamma \vdash A_1 + A_2 \leq_V \mathsf{unit}$ is not derivable.
- **Case** name: Impossible because $\Gamma \vdash \mathsf{Nm}[X] \leq_V \mathsf{unit}$ is not derivable.
- **Case** namefn: Impossible because $\Gamma \vdash (\mathbf{Nm} \overset{\mathsf{Nm}}{\to} \mathbf{Nm})[M] \leq_V \mathsf{unit}$ is not derivable.
- **Case** ref: Impossible because $\Gamma \vdash (\mathsf{Ref}[X]\, A_0) \leq_V \mathsf{unit}$ is not derivable.
- **Case** thunk: Impossible because $\Gamma \vdash (\mathsf{Thk}[X]\, E) \leq_V \mathsf{unit}$ is not derivable.
- **Case** vtype-∀IndexIntro:

	$\Gamma \vdash (\forall a : \gamma \mid P.\, A_0) \leq_V \mathsf{unit}$	Given
	$\Gamma, a : \gamma, P \vdash A_0 \leq_V \mathsf{unit}$	By inversion ($\leq_V$-∀L)
	$\Gamma, a : \gamma, P \vdash v : A_0$	Subderivation
☞	$v = ()$	By i.h. (part 1)

- **Case**

$$\frac{\Gamma \vdash i : \gamma \qquad \begin{array}{c} \textit{extract}(\Gamma) \Vdash [i/a]P \\ \Gamma \vdash v : (\forall a : \gamma \mid P.\, A_0) \end{array}}{\Gamma \vdash v : [i/a]A_0}\ \text{vtype-∀IndexElim}$$

	$\Gamma \vdash i : \gamma$	Subderivation
	$\textit{extract}(\Gamma) \Vdash [i/a]P$	Subderivation
	$\Gamma \vdash (\forall a : \gamma \mid P.\, A_0) \leq_V [i/a]A_0$	By $\leq_V$-∀L
	$\Gamma \vdash [i/a]A_0 \leq_V A$	Given
	$\Gamma \vdash (\forall a : \gamma \mid P.\, A_0) \leq_V \mathsf{unit}$	By Lemma B.7 (Transitivity of Subtyping)
	$\Gamma \vdash v : (\forall a : \gamma \mid P.\, A_0)$	Subderivation
☞	$v = ()$	By i.h.

- **Case** vtype-∃IndexIntro:
  Impossible because $\Gamma \vdash (\exists a : \gamma \mid P.\, A_0) \leq_V \mathsf{unit}$ is not derivable.

(2) $\times$:
Consider cases of the rule concluding $\Gamma \vdash v : A$.

- **Case** unit: Impossible because $\Gamma \vdash \mathsf{unit} \leq_V (B_1 \times B_2)$ is not derivable.
- **Case** pair:

$\Gamma \vdash A \leq_V (B_1 \times B_2)$ Given
$A = (A_1 \times A_2)$ By inversion (pair)
☞ $v = (v_1, v_2)$ ″
$\Gamma \vdash v_1 : B_1$ ″
$\Gamma \vdash v_2 : B_2$ ″

$\Gamma \vdash A_1 \leq_V B_1$ By inversion ($\leq_V$-$\times$)
$\Gamma \vdash A_2 \leq_V B_2$ ″

☞ $\Gamma \vdash v_1 : B_1$ By vtype-sub
☞ $\Gamma \vdash v_2 : B_2$ By vtype-sub

- **Cases** name, namefn, ref, thunk:
  Impossible because the assumed subtyping is not derivable.
- **Case** vtype-∀IndexIntro:

$\Gamma \vdash (\forall a : \gamma \mid P.\, A_0) \leq_V (B_1 \times B_2)$ Given
$\Gamma, a : \gamma, P \vdash A_0 \leq_V (B_1 \times B_2)$ By inversion ($\leq_V$-∀L)
$\Gamma, a : \gamma, P \vdash v : A_0$ Subderivation
☞ $v = (v_1, v_2)$ By i.h. (part 2)
☞ $\Gamma \vdash v_1 : B_1$ ″
☞ $\Gamma \vdash v_2 : B_2$ ″

- **Case**

$$\frac{\Gamma \vdash i : \gamma \qquad \begin{array}{c} \mathit{extract}(\Gamma) \Vdash [i/a]P \\ \Gamma \vdash v : (\forall a : \gamma \mid P.\, A_0) \end{array}}{\Gamma \vdash v : [i/a]A_0}\ \text{vtype-}\forall\text{IndexElim}$$

$\Gamma \vdash i : \gamma$ Subderivation
$\mathit{extract}(\Gamma) \Vdash [i/a]P$ Subderivation
$\Gamma \vdash (\forall a : \gamma \mid P.\, A_0) \leq_V [i/a]A_0$ By $\leq_V$-∀L

$\Gamma \vdash [i/a]A_0 \leq_V A$ Given
$\Gamma \vdash (\forall a : \gamma \mid P.\, A_0) \leq_V (B_1 \times B_2)$ By Lemma B.7 (Transitivity of Subtyping)
$\Gamma \vdash v : (\forall a : \gamma \mid P.\, A_0)$ Subderivation
☞ $v = (v_1, v_2)$ By i.h. (part 2)
☞ $\Gamma \vdash v_1 : B_1$ ″
☞ $\Gamma \vdash v_2 : B_2$ ″

- **Case** vtype-∃IndexIntro:
  Impossible because $\Gamma \vdash (\exists a : \gamma \mid P.\, A_0) \leq_V (B_1 \times B_2)$ is not derivable.

(3) $+$: Similar to Part 2.
(4) $\mathsf{Nm}[X]$:
In the $\leq_V$-name case, use the fact that $\Gamma \Vdash X' \subseteq X$ and $\Gamma \vdash n \in X'$ implies $\Gamma \vdash n \in X$. Otherwise similar to Part 1.
(5) $\mathsf{Ref}[X]\ A_0$: Similar to Parts 1 and 4.
(6) $\mathsf{Thk}[X]\ E$: Similar to Part 5.
(7) $(\mathbf{Nm} \overset{\mathbf{Nm}}{\Rightarrow} \mathbf{Nm})[M]$: Similar to Part 5. □

Lemma B.9 (Application and membership commute). *If* $\Gamma \vdash n \in i$ *and* $p =_\beta M(n)$ *then* $\Gamma \vdash p \in M(i)$.

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

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

Theorem 6.1 (Subject Reduction). *If* Γ_1 store-type *and* $\Gamma_1 \vdash M : \mathbf{Nm} \overset{\mathbf{Nm}}{\Rightarrow} \mathbf{Nm}$ *and* $\Gamma_1 \vdash^M e : C \rhd \langle W; R\rangle$ *and* $\vdash S_1 : \Gamma_1$ *and* $\mathcal{D}$ *derives* $S_1 \vdash^M_m e \Downarrow S_2; t$ *then there exists* $\Gamma_2 \supseteq \Gamma_1$ *s.t.* 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* $\langle W_\mathcal{D}; R_\mathcal{D}\rangle \preceq \langle W; R\rangle$.

Proof. By induction on the derivation $\mathcal{S}$ of $\Gamma_1 \vdash^M e : C \rhd \langle W; R\rangle$.

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

	$(e = t)$ and $(S_1 = S_2)$	Given
	$(R_\mathcal{D} = W_\mathcal{D} = R = W = \emptyset)$	″
	$(\Gamma_2 = \Gamma_1)$	Suppose
☞	$\vdash S_2 : \Gamma_2$	by above equalities
☞	$\Gamma_2 \vdash t : C \rhd \langle \emptyset, \emptyset\rangle$	″
☞	$\mathcal{D}$ reads $R_\mathcal{D}$ writes $W_\mathcal{D}$	By Def. 6.2
☞	$\langle W_\mathcal{D}; R_\mathcal{D}\rangle \preceq \langle W; R\rangle$	All are empty

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

	$(W = \emptyset)$ and $(R = X)$	Given
	$\Gamma_1 \vdash v : \mathsf{Ref}[X]\,A$	Given
	$\exists p.\ (v = \mathsf{ref}\ p)$	Lemma B.8 (Canonical Forms)
	$\Gamma_1 \vdash p \in X$	″
	$\Gamma_1(p) = A$	By inversion of value typing
	$\exists v_p.\ S_1(p) = v_p$	Inversion on $\vdash S_1 : \Gamma_1$
	$\Gamma_1 \vdash v_p : A$	″
	$(\Gamma_2 = \Gamma_1)$ and $(t = \mathsf{ret}(v_p))$	Suppose
	$(R_\mathcal{D} = \{p\})$ and $(W_\mathcal{D} = \emptyset = W)$	″
☞	$\vdash S_2 : \Gamma_2$	By above equalities
☞	$\Gamma_2 \vdash t : C \rhd \langle \emptyset, \emptyset\rangle$	″
☞	$\mathcal{D}$ reads $R_\mathcal{D}$ writes $W_\mathcal{D}$	By Def. 6.2
☞	$\langle W_\mathcal{D}; R_\mathcal{D}\rangle \preceq \langle W; R\rangle$	By above equality $W_\mathcal{D} = W = \emptyset$,
		... and inequality for $(R_\mathcal{D} = \{p\}) \subseteq (X = R)$.

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

	$\Gamma \vdash e : \tau$	Given
	$\Gamma \vdash e : \tau$	Given
	$(W = \emptyset)$ and $(R = X)$	Given
	$\Gamma_1 \vdash v : \mathsf{Thk}[X]\,(C \rhd \epsilon)$	Given
	$\exists p.\ (v = \mathtt{thunk}\ p)$	Lemma B.8 (Canonical Forms)
	$\Gamma_1 \vdash p \in X$	″
	$\Gamma_1(p) = (C \rhd \epsilon)$	By inversion of value typing
	$\exists e_p.\ S_1(p) = e_p$	Inversion on $\vdash S_1 : \Gamma_1$
$\mathcal{S}_0$ ::	$\Gamma_1 \vdash e_p : (C \rhd \epsilon)$	″
$\mathcal{D}_0$ ::	$S_1 \vdash^M_m e_p \Downarrow S_2; t$	Inversion of $\mathcal{D}$
☞	$\vdash S_2 : \Gamma_2$	By i.h. on $\mathcal{S}_0$ and $\mathcal{D}_0$
☞	$\Gamma_2 \vdash t : C \rhd \langle\emptyset, \emptyset\rangle$	″
	$\mathcal{D}_0$ reads $R_{\mathcal{D}}$ writes $W_{\mathcal{D}}$	″
	$\langle W_{\mathcal{D}_0}; R_{\mathcal{D}_0}\rangle \preceq \langle W; R\rangle$	″
☞	$\mathcal{D}$ reads $R_{\mathcal{D}_0}$ writes $W_{\mathcal{D}_0}$	By Def. 6.2
☞	$\langle W_{\mathcal{D}_0}; R_{\mathcal{D}_0}\rangle \preceq \langle W; R\rangle$	by above equality $W_{\mathcal{D}} = W = \emptyset$,
		. . . and inequality $(R_{\mathcal{D}} = \{p\}) \subseteq (X = R)$.

- **Case** $$\dfrac{\Gamma_1 \vdash v : (\mathbf{Nm} \overset{\mathbf{Nm}}{\Rightarrow} \mathbf{Nm})[M'] \qquad \Gamma_1 \vdash^{M \circ M'} e_0 : C \rhd \langle W; R\rangle}{\Gamma_1 \vdash^{M} \mathtt{scope}(v, e_0) : C \rhd \langle W; R\rangle}\ \text{scope}$$

$\mathcal{S}_0$ ::	$\Gamma_1 \vdash^{M \circ M'} e_0 : C \rhd \langle W; R\rangle$	Subderivation 2 of $\mathcal{S}$
$\mathcal{D}$ ::	$S_1 \vdash^M_m \mathtt{scope}(v, e_0) \Downarrow S_2; t$	Given
$\mathcal{D}_0$ ::	$S_1 \vdash^{M \circ M'}_m e_0 \Downarrow S_2; t$	By inversion (scope)

$\Gamma_1 \vdash M : \mathbf{Nm} \overset{\mathbf{Nm}}{\Rightarrow} \mathbf{Nm}$	Assumption
$\Gamma_1 \vdash v : (\mathbf{Nm} \overset{\mathbf{Nm}}{\Rightarrow} \mathbf{Nm})[M']$	Subderivation 1 of $\mathcal{S}$
$\Gamma_1 \vdash M' : \mathbf{Nm} \overset{\mathbf{Nm}}{\Rightarrow} \mathbf{Nm}$	By inversion
$\Gamma_1, x : \mathbf{Nm} \vdash M'\,x : \mathbf{Nm}$	By rule t-app
$\Gamma_1, x : \mathbf{Nm} \vdash M\,(M'\,x) : \mathbf{Nm}$	By rule t-app
$\Gamma_1 \vdash \lambda x.\,M\,(M'\,x) : \mathbf{Nm} \overset{\mathbf{Nm}}{\Rightarrow} \mathbf{Nm}$	By rule t-abs
$\Gamma_1 \vdash (M \circ M') : \mathbf{Nm} \overset{\mathbf{Nm}}{\Rightarrow} \mathbf{Nm}$	By definition of $M \circ M'$

☞	$\vdash S_2 : \Gamma_2$	By i.h. on $\mathcal{S}_0$
☞	$\Gamma_2 \vdash t : C \rhd \langle\emptyset; \emptyset\rangle$	″
	$\mathcal{D}_0$ reads $R_{\mathcal{D}_0}$ writes $W_{\mathcal{D}_0}$	″
	$\langle W_{\mathcal{D}_0}; R_{\mathcal{D}_0}\rangle \preceq \langle W; R\rangle$	″
☞	$\mathcal{D}$ reads $R_{\mathcal{D}}$ writes $W_{\mathcal{D}}$	By Def. 6.2
	$\langle W_{\mathcal{D}}; R_{\mathcal{D}}\rangle = \langle W_{\mathcal{D}_0}; R_{\mathcal{D}_0}\rangle$	″
☞	$\langle W_{\mathcal{D}}; R_{\mathcal{D}}\rangle \preceq \langle W; R\rangle$	By above equalities

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

$C = \mathbf{F}\,(\mathsf{Thk}[M(X)]\,E)$ and $R = \emptyset$ and $W = M(X)$	Given from $\mathcal{S}$
$\Gamma_1 \vdash v : \mathsf{Nm}[X]$	Subderivation
$(v = \mathtt{name}\ n)$ and $(n \in X)$	By Lemma B.8
$M\ n \Downarrow p$ and $R_{\mathcal{D}} = \emptyset$ and $W_{\mathcal{D}} = \{p\}$	Given from $\mathcal{D}$
$S_2 = (S_1, p : e)$	$''$

	$\Gamma_2 = (\Gamma_1, p : \mathsf{Thk}[p]\,E)$	Suppose
☞	$\vdash S_2 : \Gamma_2$	By rule (Fig. 18)
	$\Gamma_2(p) = E$	By inversion of value typing

	$\Gamma_2 \vdash \mathtt{ref}\ p : \mathsf{Ref}[p]\,E$	By rule thunk
☞	$\Gamma_2 \vdash^M \mathtt{ret}(\mathtt{thunk}\ p) : \mathbf{F}\,(\mathsf{Thk}[p]\,A) \rhd \langle\emptyset;\emptyset\rangle$	By rule ret
☞	$\mathcal{D}$ reads $R_{\mathcal{D}}$ writes $W_{\mathcal{D}}$ and $W_{\mathcal{D}} = \{p\}$	By Def. 6.2

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

- **Case**

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

$C = \mathbf{F}\,(\mathsf{Ref}[M(X)]\,A)$ and $R = \emptyset$ and $W = M(X)$	Given from $\mathcal{S}$
$\Gamma_1 \vdash v_1 : \mathsf{Nm}[X]$	Subderivation
$(v_1 = \mathtt{name}\ n)$ and $(n \in X)$	Lemma B.8 (Canonical Forms)
$M\ n \Downarrow p$ and $R_{\mathcal{D}} = \emptyset$ and $W_{\mathcal{D}} = \{p\}$	Given from $\mathcal{D}$
$S_2 = (S_1, p : v_2)$	$''$

	$\Gamma_2 = (\Gamma_1, p : \mathsf{Ref}[p]\,A)$	Suppose
☞	$\vdash S_2 : \Gamma_2$	By rule (Fig. 18)
	$\Gamma_2(p) = A$	By inversion of value typing
	$\Gamma_2 \vdash \mathtt{ref}\ p : \mathsf{Ref}[p]\,A$	By rule ref
☞	$\Gamma_2 \vdash^M \mathtt{ret}(\mathtt{ref}\ p) : \mathtt{ret}(\mathsf{Ref}[p]\,A) \rhd \langle\emptyset;\emptyset\rangle$	By rule ret
☞	$\mathcal{D}$ reads $R_{\mathcal{D}}$ writes $W_{\mathcal{D}}$ and $W_{\mathcal{D}} = \{p\}$	By Def. 6.2

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

- **Case**

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

	$\vdash S_1 : \Gamma_1$	Given
$\mathcal{S}_1 ::$	$\Gamma_1 \vdash^M e_1 : \mathbf{F}\, A \rhd \epsilon_1$	Subderivation 1 of $\mathcal{S}$
$\mathcal{D}_1 ::$	$S_1 \vdash^M_m e_1 \Downarrow S_{12}; t_1$	Subderivation 1 of $\mathcal{D}$
	exists $\Gamma_{12} \supseteq \Gamma_1$ such that $S_{12} : \Gamma_{12}$	By i.h. on $\mathcal{S}_1$
	$\Gamma_{12} \vdash t_1 : \mathbf{F}\, A \rhd \langle\emptyset; \emptyset\rangle$	″
	$\mathcal{D}_1$ reads $R_{\mathcal{D}_1}$ writes $W_{\mathcal{D}_1}$	″
	$\langle W_{\mathcal{D}_1}; R_{\mathcal{D}_1}\rangle \leq \epsilon_1$	″
	$\langle W_{\mathcal{D}_1}; R_{\mathcal{D}_1}\rangle \leq \langle W_1, R_1\rangle$	″
	$\Gamma_{12} \vdash v : A$	inversion of typing rule ret, for terminal computation t_1
$\mathcal{S}_2 ::$	$\Gamma_1, x : A \vdash^M e_2 : C \rhd \epsilon_2$	Subderivation 2 of $\mathcal{S}$
	$\Gamma_{12}, x : A \vdash^M e_2 : C \rhd \epsilon_2$	Lemma B.2 (Weakening)
	$\Gamma_{12} \vdash^M [v/x]e_2 : C \rhd \epsilon_2$	Lemma B.3 (Substitution)
$\mathcal{D}_2 ::$	$S_{12} \vdash^M_m [v/x]e_2 \Downarrow S_2; t_2$	Subderivation 2 of $\mathcal{D}$
	exists $\Gamma_2 \supseteq \Gamma_{12} \supseteq \Gamma_1$ such that	By i.h. on $\mathcal{S}_2$
☞	$\vdash S_2 : \Gamma_2$	″
☞	$\Gamma_2 \vdash^M t_2 : C \rhd \langle\emptyset; \emptyset\rangle$	″
	$\mathcal{D}_2$ reads $R_{\mathcal{D}_2}$ writes $W_{\mathcal{D}_2}$	″
	$\langle W_{\mathcal{D}_2}; R_{\mathcal{D}_2}\rangle \leq \epsilon_2$	″
	$\langle W_{\mathcal{D}_2}; R_{\mathcal{D}_2}\rangle \leq \langle W_2, R_2\rangle$	″
	$W_1 \perp W_2$ and $R_1 \perp W_2$	Definition of ϵ_1 then ϵ_2
	$W_{\mathcal{D}_1} \perp W_{\mathcal{D}_2}$ and $R_{\mathcal{D}_1} \perp W_{\mathcal{D}_2}$	$W_{\mathcal{D}_1} \subseteq W_1$; $W_{\mathcal{D}_2} \subseteq W_2$; $R_{\mathcal{D}_1} \subseteq R_1$
	$W_{\mathcal{D}} = W_{\mathcal{D}_1} \perp W_{\mathcal{D}_2}$	By Def. 6.2
	$R_{\mathcal{D}} = R_{\mathcal{D}_1} \cup (R_{\mathcal{D}_2} - W_{\mathcal{D}_1})$	″
☞	$\mathcal{D}$ reads $R_{\mathcal{D}}$ writes $W_{\mathcal{D}}$	″
☞	$\langle W_{\mathcal{D}}, R_{\mathcal{D}}\rangle \leq \langle W, R\rangle$	Since $W_{\mathcal{D}} \subseteq W$ and $R_{\mathcal{D}} \subseteq R$

- **Case** $$\dfrac{\Gamma \vdash^M e : \big((A \to E) \rhd \epsilon_1\big) \qquad \Gamma \vdash v : A}{\Gamma \vdash^M (e\ v) : (E \text{ after } \epsilon_1)}\ \text{app}$$

  Similar to the case for let.

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

  Similar to the case for let, using Lemma B.8 (Canonical Forms).

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

  Similar to the case for let, using Lemma B.8 (Canonical Forms).

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

$$
\begin{array}{ll}
\Gamma_1 \vdash v_M : (\mathbf{Nm} \overset{\mathbf{Nm}}{\rightarrow} \mathbf{Nm})[M] & \text{Given} \\
\quad v_M = \mathtt{nmfn}\ M_v & \text{Lemma B.8 (Canonical Forms)} \\
\quad M =_\beta (\lambda a.\, M') & '' \\
\quad \cdot \vdash \lambda a.\, M' : (\mathbf{Nm} \overset{\mathbf{Nm}}{\rightarrow} \mathbf{Nm}) & '' \\
\quad M_v =_\beta M & '' \\
\quad \Gamma_1 \vdash v : \mathsf{Nm}[i] & \text{Given} \\
\qquad v = \mathtt{name}\ n & \text{Lemma B.8 (Canonical Forms)} \\
\qquad \Gamma \vdash n \in i & '' \\
\\
\qquad M \Downarrow_M (\lambda a.\, M') & \text{By inversion on } \mathcal{D} \text{ (name-app)} \\
\qquad [n/a]M' \Downarrow_M p & '' \\
\quad p =_\beta [n/a]M' & \text{By a property of } \Downarrow_M \\
\quad\ =_\beta (\lambda a.\, M')(n) & \text{By a property of } =_\beta \\
\quad\ =_\beta M(n) & \text{By a property of } =_\beta \\
\\
\qquad (\Gamma_2 = \Gamma_1), (S_2 = S_1) & \text{Suppose} \\
☞ \qquad \vdash S_2 : \Gamma_2 & \text{By above equalities and } S_1 \vdash \Gamma_1 \\
\\
\quad \Gamma_1 \vdash n \in i & \text{Above} \\
\quad p =_\beta M(n) & \text{Above} \\
\quad \Gamma_1 \vdash p \in M(i) & \text{By Lemma B.9}
\end{array}
$$

$$
\begin{array}{ll}
\quad \Gamma_1 \vdash \mathtt{name}\ p : \mathsf{Nm}[M(i)]) & \text{By rule name} \\
☞\ \Gamma_1 \vdash \mathtt{ret}(\mathtt{name}\ p) : \mathbf{F}\,(\mathsf{Nm}[M(i)]) \rhd \langle \emptyset; \emptyset \rangle & \text{By rule ret} \\
☞ \quad \mathcal{D} \text{ by } \Downarrow\text{-name-app reads } \emptyset \text{ writes } \emptyset & \text{By Def. 6.2} \\
☞ \quad (R_{\mathcal{D}} = R = \emptyset), (W_{\mathcal{D}} = W = \emptyset) & \text{By above equalities}
\end{array}
$$

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

$$
\begin{array}{ll}
\mathcal{S}_0 :: \Gamma_1, a : \gamma, P \vdash^M t : E & \text{Subderivation} \\
\mathcal{D}_0 :: \qquad S_1 \vdash^M_m e \Downarrow S_2; t & \text{Subderivation} \\
\qquad \exists \Gamma_2 \subseteq \Gamma_1 & \text{By i.h.} \\
☞ \qquad\qquad \vdash S_2 : \Gamma_2 & '' \\
\qquad\qquad \mathcal{D}_0 \text{ reads } R_{\mathcal{D}_0} \text{ writes } W_{\mathcal{D}_0} & '' \\
\qquad\qquad \Gamma_2 \vdash t : E & '' \\
\qquad\qquad \langle R_{\mathcal{D}_0}; W_{\mathcal{D}_0} \rangle \preceq \langle R; W \rangle & '' \\
☞ \qquad \Gamma_2 \vdash^M t : (\forall a : \gamma.\, E) & \text{By typing rule} \\
☞ \qquad\qquad \mathcal{D} \text{ reads } R_{\mathcal{D}} \text{ writes } W_{\mathcal{D}} & \text{By Def. 6.2} \\
☞ \qquad\qquad \langle R_{\mathcal{D}}; W_{\mathcal{D}} \rangle \preceq \langle R; W \rangle & \text{By set theory}
\end{array}
$$

- **Case** $\dfrac{\Gamma_1 \vdash^M e : (\forall a : \gamma \mid P.\, E) \qquad \begin{array}{l}\Gamma_1 \vdash i : \gamma \\ \mathit{extract}(\Gamma_1) \Vdash [i/a]P\end{array}}{\Gamma_1 \vdash^M e : [i/a]E}$ AllIndexElim

$$
\begin{array}{lll}
\mathcal{S}_0 :: & \Gamma_1 \vdash^M e : (\forall a : \gamma.\, E) & \text{Subderivation} \\
\mathcal{D}_0 :: & S_1 \vdash^M_m e \Downarrow S_2; t & \text{Subderivation} \\
& \exists \Gamma_2 \subseteq \Gamma_1 & \text{By i.h.} \\
☞ & \quad \vdash S_2 : \Gamma_2 & '' \\
& \quad \mathcal{D}_0 \text{ reads } R_{\mathcal{D}_0} \text{ writes } W_{\mathcal{D}_0} & '' \\
& \quad \Gamma_2 \vdash t : (\forall a : \gamma.\, E) & '' \\
& \quad \langle R_{\mathcal{D}_0}; W_{\mathcal{D}_0} \rangle \preceq \langle R; W \rangle & '' \\
& \Gamma_1 \vdash i : \gamma & \text{Subderivation} \\
& \Gamma_2 \vdash i : \gamma & \text{By weakening} \\
☞ & \Gamma_2 \vdash^M t : [i/a]E & \text{By typing rule} \\
☞ & \quad \mathcal{D} \text{ reads } R_{\mathcal{D}} \text{ writes } W_{\mathcal{D}} & \text{By Def. 6.2} \\
☞ & \quad \langle R_{\mathcal{D}}; W_{\mathcal{D}} \rangle \preceq \langle R; W \rangle & \text{By set theory}
\end{array}
$$

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

  Similar to the AllIndexIntro case.

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

  Similar to the AllIndexElim case. □

C BIDIRECTIONAL TYPING

C.1 Syntax

As discussed below, bidirectional typing requires some annotations, so we assume that values v and expressions e have been extended with annotations $(v : A)$ and $(e : A)$. We also assume that we have explicit syntactic forms $e[i]$ and $e[A]$, which avoid guessing quantifier instantiations.

C.2 Bidirectional Typing Rules

The typing rules in Figures 13 and 14 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 readily applicable.

Following the DML tradition, we obtain an algorithmic version of our typing rules by defining a bidirectional system [Pierce and Turner 2000]: we split judgments with a colon into judgments with an arrow. Thus, the computation typing judgment $\cdots e : E$ becomes two judgments. The first is the *checking* judgment $\Gamma \vdash^M e \Leftarrow E$, in which the type E is already known—it is an *input* to the algorithm. The second is the *synthesis* judgment $\Gamma \vdash^M e \Rightarrow E$, in which E is not known—it is an output—and the rules construct E by examining e (and Γ).

In formulating the bidirectional versions of value and computation typing (Figures 22 and 23), we mostly follow the "recipe" of Dunfield and Pfenning [2004]: introduction rules check, and elimination rules synthesize. More precisely, the *principal judgment*—the judgment, either a premise or conclusion, that has the connective being introduced or eliminated—is checking ($\Leftarrow$) for introduction rules, and synthesizing ($\Rightarrow$) for elimination rules. In many cases, once the direction of that premise (or conclusion) is determined, the direction of the other judgments follows by considering what information is known (as input, or as the output type of the principal judgment, if that judgment is synthesizing). For example, if we commit to checking the conclusion of echk-lam,

$\boxed{\Gamma \vdash v \Rightarrow A}$ Under Γ, value v synthesizes type A

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A}\ \text{vsyn-var} \qquad \frac{\Gamma \vdash v \Leftarrow A}{\Gamma \vdash (v : A) \Rightarrow A}\ \text{vsyn-anno}$$

$\boxed{\Gamma \vdash v \Leftarrow A}$ Under Γ, value v checks against type A

$$\frac{}{\Gamma \vdash () \Leftarrow \mathsf{unit}}\ \text{vchk-unit} \qquad \frac{\Gamma \vdash v_1 \Leftarrow A_1 \qquad \Gamma \vdash v_2 \Leftarrow A_2}{\Gamma \vdash (v_1, v_2) \Leftarrow (A_1 \times A_2)}\ \text{vchk-pair}$$

$$\frac{\Gamma \vdash v \Rightarrow A \qquad \Gamma \vdash A \leq_V B}{\Gamma \vdash v \Leftarrow B}\ \text{vchk-sub} \qquad \frac{\Gamma \vdash n \in X}{\Gamma \vdash (\mathtt{name}\ n) \Leftarrow \mathsf{Nm}[X]}\ \text{vchk-name}$$

$$\frac{\Gamma \vdash M_v \Rightarrow (\mathbf{Nm} \overset{\mathsf{Nm}}{\to} \mathbf{Nm}) \qquad M_v =_\beta M}{\Gamma \vdash (\mathtt{nmfn}\ M_v) \Leftarrow (\mathbf{Nm} \overset{\mathsf{Nm}}{\to} \mathbf{Nm})[M]}\ \text{vchk-namefn}$$

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

$$\frac{\Gamma \vdash v \Leftarrow A_1}{\Gamma \vdash \mathsf{inj}_1\, v \Leftarrow A_1 + A_2}\ \text{vchk-inj1} \qquad \frac{\Gamma \vdash v \Leftarrow A_2}{\Gamma \vdash \mathsf{inj}_2\, v \Leftarrow A_1 + A_2}\ \text{vchk-inj2}$$

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

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

Fig. 22. Bidirectional value typing

we should check the premise because its type is a subexpression of the type in the conclusion. (Checking is more powerful than synthesis: every expression that synthesizes also checks, but not all expressions that check can synthesize.)

When a synthesis (elimination) premise attempts to type an expression that is a checking (introduction) form, the programmer must write a type annotation $(e : E)$. Thus, following the recipe means that we have a straightforward *annotation discipline*: annotations are needed only on redexes. While we could reduce the number of annotations by adding synthesis rules—for example, allowing the unit value () to synthesize unit—this makes the system larger without changing its essential properties; for a discussion of the implications of such extensions in a different context, see Dunfield and Krishnaswami [2013].

Dually, when an expression synthesizes but we are trying to derive a checking judgment, we use (1) vchk-sub for value typing, or (2) echk-sub for computation typing. The latter rule includes effect subsumption.

D BIDIRECTIONAL TYPING PROOFS

Theorem D.1 (Soundness of Bidirectional Value Typing).

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

$$\frac{\Gamma \vdash^M e \Leftarrow E}{\Gamma \vdash^M (e : E) \Rightarrow E}\ \text{esyn-anno} \qquad \frac{\Gamma \vdash^M e \Rightarrow ((A \to E) \rhd \epsilon_1) \quad \Gamma \vdash v \Leftarrow A \quad \Gamma \vdash E \text{ after } \epsilon_1 \equiv E'}{\Gamma \vdash^M (e\ v) \Rightarrow E'}\ \text{esyn-app}$$

$$\frac{\Gamma \vdash v \Rightarrow \mathsf{Thk}[X]\ (C \rhd \epsilon) \quad \Gamma \vdash \langle \emptyset; X\rangle \text{ then } \epsilon \equiv \epsilon'}{\Gamma \vdash^M \mathsf{force}(v) \Rightarrow C \rhd \epsilon'}\ \text{esyn-force} \qquad \frac{\Gamma \vdash v \Rightarrow \mathsf{Ref}[X]\ A}{\Gamma \vdash^M \mathsf{get}(v) \Rightarrow (\mathbf{F}\, A) \rhd \langle \emptyset; X\rangle}\ \text{esyn-get}$$

$$\frac{\Gamma \vdash v_M \Rightarrow (\mathbf{Nm} \stackrel{\mathsf{Nm}}{\to} \mathbf{Nm})[M] \qquad \Gamma \vdash v \Rightarrow \mathsf{Nm}[i]}{\Gamma \vdash^N (v_M\ v) \Rightarrow \mathbf{F}\,(\mathsf{Nm}[M[i]]) \rhd \langle \emptyset; \emptyset\rangle}\ \text{esyn-name-app}$$

$$\frac{\Gamma \vdash^M e \Rightarrow (\forall a : \gamma \mid P.\, E) \qquad \Gamma \vdash i : \gamma \qquad \mathit{extract}(\Gamma) \Vdash [i/a]P}{\Gamma \vdash^M e[i] \Rightarrow [i/a]E}\ \text{esyn-}\forall\text{IndexElim}$$

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

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

$$\frac{\Gamma \vdash^M e \Rightarrow E_1 \qquad \Gamma \vdash E_1 \leq_E E_2}{\Gamma \vdash^M e \Leftarrow E_2}\ \text{echk-sub}$$

$$\frac{\Gamma \vdash v \Rightarrow (A_1 \times A_2) \quad \Gamma, x_1 : A_1, x_2 : A_2 \vdash^M e \Leftarrow E}{\Gamma \vdash^M \mathsf{split}(v, x_1.x_2.e) \Leftarrow E}\ \text{echk-split} \qquad \frac{\Gamma \vdash v \Rightarrow (A_1 + A_2) \quad \Gamma, x_1 : A_1 \vdash^M e_1 \Leftarrow E \quad \Gamma, x_2 : A_2 \vdash^M e_2 \Leftarrow E}{\Gamma \vdash^M \mathsf{case}(v, x_1.e_1, x_2.e_2) \Leftarrow E}\ \text{echk-case}$$

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

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

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

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

$$\frac{\Gamma \vdash v \Rightarrow (\mathbf{Nm} \stackrel{\mathsf{Nm}}{\to} \mathbf{Nm})[N'] \qquad \Gamma \vdash^{N \circ N'} e \Leftarrow (C \rhd \langle W; R\rangle)}{\Gamma \vdash^N \mathsf{scope}(v, e) \Leftarrow (C \rhd \langle W; R\rangle)}\ \text{echk-scope}$$

$$\frac{\Gamma, a : \gamma, P \vdash^M t \Leftarrow E}{\Gamma \vdash^M t \Leftarrow (\forall a : \gamma \mid P.\, E)}\ \text{echk-}\forall\text{IndexIntro} \qquad \frac{\Gamma, \alpha : K \vdash^M t \Leftarrow E}{\Gamma \vdash^M t \Leftarrow (\forall \alpha : K.\, E)}\ \text{echk-}\forall\text{Intro}$$

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

Fig. 23. Bidirectional computation typing

(1) *If* $\Gamma \vdash v \Rightarrow A$*, then there exists a value* v' *such that* $\Gamma \vdash v' : A$ *and* $|v| = v'$*.*
(2) *If* $\Gamma \vdash v \Leftarrow A$*, then there exists a value* v' *such that* $\Gamma \vdash v' : A$ *and* $|v| = v'$*.*

Proof. By induction on the given derivation.
Part (1): Proceed by cases on the rule concluding $\Gamma \vdash v \Rightarrow A$.

- **Case** $\dfrac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A}$ vsyn-var

$$\begin{array}{rl} (x : A) \in \Gamma & \text{Given} \\ \Gamma \vdash x : A & \text{By rule var} \\ |x| = x & \text{By definition of } |-| \\ ☞\ \Gamma \vdash v' : A \text{ and } |v| = v' & \text{where } v' = x \text{ and } v = x \end{array}$$

- **Case** $\dfrac{\Gamma \vdash v_1 \Leftarrow A}{\Gamma \vdash (v_1 : A) \Rightarrow A}$ vsyn-anno

$$\begin{array}{rl} \exists\, v_1' \text{ such that } \Gamma \vdash v_1' : A \text{ and } |v_1| = v_1' & \text{By inductive hypothesis} \\ |(v_1 : A)| = |v_1| = v_1' & \text{By definition of } |-| \\ & \quad\text{and } |v_1| = v_1' \\ ☞\ \Gamma \vdash v' : A \text{ and } |v| = v' & \text{where } v' = v_1' \text{ and } v = (v_1 : A) \end{array}$$

Part (2): Proceed by cases on the rule concluding $\Gamma \vdash v \Leftarrow A$.

- **Case** $\dfrac{}{\Gamma \vdash () \Leftarrow \mathsf{unit}}$ vchk-unit

$$\begin{array}{rl} \Gamma \vdash () : \mathsf{unit} & \text{By rule unit} \\ |()| = () & \text{By definition of } |-| \\ ☞\ \Gamma \vdash v' : () \text{ and } |v| = v' & \text{where } v' = () \text{ and } v = () \end{array}$$

- **Case** $\dfrac{\Gamma \vdash v_1 \Leftarrow A_1 \qquad \Gamma \vdash v_2 \Leftarrow A_2}{\Gamma \vdash (v_1, v_2) \Leftarrow (A_1 \times A_2)}$ vchk-pair

$$\begin{array}{rl} \exists\, v_1' \text{ such that } \Gamma \vdash v_1' : A_1 \text{ and } |v_1| = v_1' & \text{By inductive hypothesis} \\ \exists\, v_2' \text{ such that } \Gamma \vdash v_2' : A_2 \text{ and } |v_2| = v_2' & \text{By inductive hypothesis} \\ \Gamma \vdash (v_1', v_2') : (A_1 \times A_2) & \text{By rule pair} \\ |(v_1, v_2)| = (|v_1|, |v_2|) = (v_1', v_2') & \text{By definition of } |-| \\ & \quad |v_1| = v_1' \text{ and } |v_2| = v_2' \\ ☞\ \Gamma \vdash v' : (A_1 \times A_2) \text{ and } |v| = v' & \text{where } v' = (v_1', v_2') \\ & \quad\text{and } v = (v_1, v_2) \end{array}$$

- **Case** $\dfrac{\Gamma \vdash n \in X}{\Gamma \vdash (\mathtt{name}\ n) \Leftarrow \mathsf{Nm}[X]}$ vchk-name

$$\begin{array}{rl} \Gamma \vdash n \in X & \text{Given} \\ \Gamma \vdash (\mathtt{name}\ n) : \mathsf{Nm}[X] & \text{By rule name} \\ |(\mathtt{name}\ n)| = (\mathtt{name}\ n) & \text{By definition of } |-| \\ ☞\ \Gamma \vdash v' : \mathsf{Nm}[X] \text{ and } |v| = v' & \text{where } v' = (\mathtt{name}\ n) \\ & \quad\text{and } v = (\mathtt{name}\ n) \end{array}$$

- **Case** $\dfrac{\Gamma \vdash M_v \Rightarrow (\mathbf{Nm} \stackrel{\mathsf{Nm}}{\to} \mathbf{Nm}) \qquad M_v =_\beta M}{\Gamma \vdash (\mathtt{nmfn}\ M_v) \Leftarrow (\mathbf{Nm} \stackrel{\mathsf{Nm}}{\to} \mathbf{Nm})[M]}$ vchk-namefn

$\exists\, M'_v$ such that $\Gamma \vdash M'_v : (\mathbf{Nm} \overset{\mathsf{Nm}}{\to} \mathbf{Nm})$ and $|M_v| = M'_v$ — By i.h.

$M_v =_\beta M$ — Given

$|M_v| =_\beta M$ — Type erasure does not affect convertibility

$M'_v =_\beta M$ — Since $|M_v| = M'_v$

$\Gamma \vdash (\mathtt{nmfn}\ M'_v) : (\mathbf{Nm} \overset{\mathsf{Nm}}{\to} \mathbf{Nm})[M]$ — By rule namefn

$|(\mathtt{nmfn}\ M_v)| = (\mathtt{nmfn}\ |M_v|) = (\mathtt{nmfn}\ M'_v)$ — By definition of $|{-}|$ and $|M_v| = M'_v$

☞ $\Gamma \vdash v' : (\mathbf{Nm} \overset{\mathsf{Nm}}{\to} \mathbf{Nm})[M]$ and $|v| = v'$ — where $v' = (\mathtt{nmfn}\ M'_v)$ and $v = (\mathtt{nmfn}\ M_v)$

- **Case** $$\dfrac{\Gamma \vdash n \in X \qquad \Gamma(n) = A}{\Gamma \vdash (\mathsf{ref}\ n) \Leftarrow \mathsf{Ref}[X]\ A}\ \text{vchk-ref}$$

  $\Gamma \vdash n \in X$ — Given

  $\Gamma(n) = A$ — Given

  $\Gamma \vdash (\mathsf{ref}\ n) : \mathsf{Ref}[X]\ A$ — By rule ref

  $|(\mathsf{ref}\ n)| = \mathsf{ref}\ n$ — By definition of $|{-}|$

  ☞ $\Gamma \vdash v' : \mathsf{Ref}[X]\ A$ and $|v| = v'$ — where $v' = (\mathsf{ref}\ n)$ and $v = (\mathsf{ref}\ n)$

- **Case** $$\dfrac{\Gamma \vdash n \in X \qquad \Gamma(n) = E}{\Gamma \vdash (\mathsf{thunk}\ n) \Leftarrow (\mathsf{Thk}[X]\ E)}\ \text{vchk-thunk}$$

  $\Gamma \vdash n \in X$ — Given

  $\Gamma(n) = E$ — Given

  $\Gamma \vdash (\mathsf{thunk}\ n) : \mathsf{Thk}[X]\ E$ — By rule thunk

  $|(\mathsf{thunk}\ n)| = \mathsf{thunk}\ n$ — By definition of $|{-}|$

  ☞ $\Gamma \vdash v' : \mathsf{Thk}[X]\ E$ and $|v| = v'$ — where $v' = (\mathsf{thunk}\ n)$ and $v = (\mathsf{thunk}\ n)$

- **Case** $$\dfrac{\Gamma \vdash v_1 \Rightarrow A_1 \qquad \Gamma \vdash A_1 \leq_V A_2}{\Gamma \vdash v_1 \Leftarrow A_2}\ \text{vchk-sub}$$

  By i.h. and vtype-sub. □

Theorem D.2 (Completeness of Bidirectional Value Typing). *If* $\Gamma \vdash v : A$ *then there exist values* v' *and* v'' *such that*

(1) $\Gamma \vdash v' \Rightarrow A$ *and* $|v'| = v$
(2) $\Gamma \vdash v'' \Leftarrow A$ *and* $|v''| = v$

Proof. By induction on the derivation of $\Gamma \vdash v : A$.

Case

$$\dfrac{}{\Gamma \vdash () : \mathsf{unit}}\ \text{unit}$$

$$
\begin{array}{rll}
& \Gamma \vdash () \Leftarrow \mathsf{unit} & \text{By rule vchk-unit} \\
& |()| = () & \text{By definition of } |-| \\
☞ & \Gamma \vdash v'' \Leftarrow () \text{ and } |v''| = v & \text{where } v'' = () \text{ and } v = () \\
& \Gamma \vdash (() : \mathsf{unit}) \Rightarrow \mathsf{unit} & \text{By rule vsyn-anno} \\
& |(() : \mathsf{unit})| = () & \text{By definition of } |-| \\
☞ & \Gamma \vdash v' \Rightarrow () \text{ and } |v'| = v & \text{where } v' = (() : \mathsf{unit}) \text{ and } v = ()
\end{array}
$$

Case $\dfrac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\ \text{var}$

$$
\begin{array}{rll}
& (x : A) \in \Gamma & \text{Given} \\
& \Gamma \vdash x \Rightarrow A & \text{By rule vsyn-var} \\
& |x| = x & \text{By definition of } |-| \\
☞ & \Gamma \vdash v' \Rightarrow A \text{ and } |v'| = v & \text{where } v' = x \text{ and } v = x \\
& \Gamma \vdash x \Leftarrow A & \text{By rule vchk-conv} \\
☞ & \Gamma \vdash v'' \Leftarrow A \text{ and } |v''| = v & \text{where } v'' = x \text{ and } v = x \\
& & \quad \text{and } |x| = x
\end{array}
$$

Case $\dfrac{\Gamma \vdash v_1 : A_1 \qquad \Gamma \vdash v_2 : A_2}{\Gamma \vdash (v_1, v_2) : (A_1 \times A_2)}\ \text{pair}$

$$
\begin{array}{rll}
& \exists\, v_1'' \text{ such that } \Gamma \vdash v_1'' \Leftarrow A_1 \text{ and } |v_1''| = v_1 & \text{By inductive hypothesis} \\
& \exists\, v_2'' \text{ such that } \Gamma \vdash v_2'' \Leftarrow A_2 \text{ and } |v_2''| = v_2 & \text{By inductive hypothesis} \\
& \Gamma \vdash (v_1'', v_2'') \Leftarrow (A_1 \times A_2) & \text{By rule vchk-pair} \\
& |(v_1'', v_2'')| = (|v_1''|, |v_2''|) = (v_1, v_2) & \text{By definition of } |-|;\ |v_1''| = v_1;\ |v_2''| = v_2 \\
☞ & \Gamma \vdash v'' \Leftarrow (A_1 \times A_2) \text{ and } |v''| = v & \text{where } v'' = (v_1'', v_2'') \text{ and } v = (v_1, v_2) \\
& \Gamma \vdash ((v_1'', v_2'') : (A_1 \times A_2)) \Rightarrow (A_1 \times A_2) & \text{By rule vsyn-anno} \\
& |(v_1'', v_2'') : (A_1 \times A_2)| = |(v_1'', v_2'')| = (v_1, v_2) & \text{By definition of } |-|;\ |(v_1'', v_2'')| = (v_1, v_2) \\
☞ & \Gamma \vdash v' \Rightarrow (A_1 \times A_2) \text{ and } |v'| = v & \text{where } v' = (v_1'', v_2'') \text{ and } v = (v_1, v_2)
\end{array}
$$

Case $\dfrac{\Gamma \vdash n \in X}{\Gamma \vdash (\mathsf{name}\ n) : \mathsf{Nm}[X]}\ \text{name}$

$$
\begin{array}{rll}
& \Gamma \vdash n \in X & \text{Given} \\
& \Gamma \vdash (\mathsf{name}\ n) \Leftarrow \mathsf{Nm}[X] & \text{By rule vchk-name} \\
& |(\mathsf{name}\ n)| = (\mathsf{name}\ n) & \text{By definition of } |-| \\
☞ & \Gamma \vdash v'' \Leftarrow \mathsf{Nm}[X] \text{ and } |v''| = v & \text{where } v'' = (\mathsf{name}\ n) \text{ and } v = (\mathsf{name}\ n) \\
& \Gamma \vdash (\mathsf{name}\ n : \mathsf{Nm}[X]) \Rightarrow \mathsf{Nm}[X] & \text{By rule vsyn-anno} \\
& |(\mathsf{name}\ n : \mathsf{Nm}[X])| = |(\mathsf{name}\ n)| = (\mathsf{name}\ n) & \text{By definition of } |-| \\
☞ & \Gamma \vdash v' \Rightarrow \mathsf{Nm}[X] \text{ and } |v'| = v & \text{where } v' = (\mathsf{name}\ n : \mathsf{Nm}[X]) \text{ and } v = (\mathsf{name}\ n)
\end{array}
$$

Case $\dfrac{\Gamma \vdash M_v : (\mathbf{Nm} \overset{\mathsf{Nm}}{\to} \mathbf{Nm}) \qquad M_v =_\beta M}{\Gamma \vdash (\mathsf{nmfn}\ M_v) : (\mathbf{Nm} \overset{\mathsf{Nm}}{\to} \mathbf{Nm})[M]}\ \text{namefn}$

$\exists\, M_v'$ such that

$\Gamma \vdash M_v' \Rightarrow (\mathbf{Nm} \overset{\mathbf{Nm}}{\to} \mathbf{Nm})$ and $\lvert M_v' \rvert = M_v$	By inductive hypothesis
$M_v =_\beta M$	Given
$\lvert M_v' \rvert =_\beta M$	Since $\lvert M_v' \rvert = M_v$
$M_v' =_\beta M$	Type annotation does not affect convertibility
$\Gamma \vdash (\mathsf{nmfn}\ M_v') \Leftarrow (\mathbf{Nm} \overset{\mathbf{Nm}}{\to} \mathbf{Nm})[M]$	By rule vchk-namefn
$\lvert(\mathsf{nmfn}\ M_v')\rvert = (\mathsf{nmfn}\ \lvert M_v' \rvert) = (\mathsf{nmfn}\ M_v)$	By definition of $\lvert - \rvert$ and $\lvert M_v' \rvert = M_v$
☞ $\Gamma \vdash v'' \Leftarrow (\mathbf{Nm} \overset{\mathbf{Nm}}{\to} \mathbf{Nm})[M]$ and $\lvert v'' \rvert = v$	where $v'' = (\mathsf{nmfn}\ M_v')$ and $v = (\mathsf{nmfn}\ M_v)$
$\Gamma \vdash (\mathsf{nmfn}\ M_v' : (\mathbf{Nm} \overset{\mathbf{Nm}}{\to} \mathbf{Nm})[M]) \Rightarrow (\mathbf{Nm} \overset{\mathbf{Nm}}{\to} \mathbf{Nm})[M]$	By rule vsyn-anno
$\lvert(\mathsf{nmfn}\ M_v' : (\mathbf{Nm} \overset{\mathbf{Nm}}{\to} \mathbf{Nm})[M])\rvert = (\mathsf{nmfn}\ M_v)$	By definition of $\lvert - \rvert$
☞ $\Gamma \vdash v' \Rightarrow (\mathbf{Nm} \overset{\mathbf{Nm}}{\to} \mathbf{Nm})[M]$ and $\lvert v' \rvert = v$	where $v' = (\mathsf{nmfn}\ M_v' : (\mathbf{Nm} \overset{\mathbf{Nm}}{\to} \mathbf{Nm})[M])$

Case $\dfrac{\Gamma \vdash n \in X \qquad \Gamma(n) = A}{\Gamma \vdash (\mathsf{ref}\ n) : \mathsf{Ref}[X]\ A}$ ref

$\Gamma \vdash n \in X$	Given
$\Gamma(n) = A$	Given
$\Gamma \vdash (\mathsf{ref}\ n) \Leftarrow \mathsf{Ref}[X]\ A$	By rule vchk-ref
$\lvert(\mathsf{ref}\ n)\rvert = (\mathsf{ref}\ n)$	By definition of $\lvert - \rvert$
☞ $\Gamma \vdash v'' \Leftarrow \mathsf{Ref}[X]\ A$ and $\lvert v'' \rvert = v$	where $v'' = (\mathsf{ref}\ n)$ and $v = (\mathsf{ref}\ n)$
$\Gamma \vdash ((\mathsf{ref}\ n) : \mathsf{Ref}[X]\ A) \Rightarrow \mathsf{Ref}[X]\ A$	By rule vsyn-anno
$\lvert(\mathsf{ref}\ n : \mathsf{Ref}[X]\ A)\rvert = \lvert(\mathsf{ref}\ n)\rvert = (\mathsf{ref}\ n)$	By definition of $\lvert - \rvert$
☞ $\Gamma \vdash v' \Rightarrow \mathsf{Ref}[X]\ A$ and $\lvert v' \rvert = v$	where $v' = (\mathsf{ref}\ n : \mathsf{Ref}[X]\ A)$ and $v = (\mathsf{ref}\ n)$

Case $\dfrac{\Gamma \vdash n \in X \qquad \Gamma(n) = E}{\Gamma \vdash (\mathsf{thunk}\ n) : (\mathsf{Thk}[X]\ E)}$ thunk

$\Gamma \vdash n \in X$	Given
$\Gamma(n) = E$	Given
$\Gamma \vdash (\mathsf{thunk}\ n) \Leftarrow (\mathsf{Thk}[X]\ E)$	By rule vchk-thunk
$\lvert(\mathsf{thunk}\ n)\rvert = (\mathsf{thunk}\ n)$	By definition of $\lvert - \rvert$
☞ $\Gamma \vdash v'' \Leftarrow (\mathsf{Thk}[X]\ E)$ and $\lvert v'' \rvert = v$	where $v'' = (\mathsf{thunk}\ n)$ and $v = (\mathsf{thunk}\ n)$
$\Gamma \vdash (\mathsf{thunk}\ n : (\mathsf{Thk}[X]\ E)) \Rightarrow (\mathsf{Thk}[X]\ E)$	By rule vsyn-anno
$\lvert(\mathsf{thunk}\ n : (\mathsf{Thk}[X]\ E))\rvert = \mathsf{thunk}\ n$	By definition of $\lvert - \rvert$
☞ $\Gamma \vdash v' \Rightarrow (\mathsf{Thk}[X]\ E)$ and $\lvert v' \rvert = v$	where $v' = (\mathsf{thunk}\ n : (\mathsf{Thk}[X]\ E))$ and $v = (\mathsf{thunk}\ n)$

□

Theorem D.3 (Soundness of Bidirectional Computation Typing).

(1) If $\Gamma \vdash^M e \Rightarrow E$*, then there exists a value* e' *such that* $\Gamma \vdash^M e' : E$ *and* $\lvert e \rvert = e'$

(2) If $\Gamma \vdash^M e \Leftarrow E$*, then there exists a value* e' *such that* $\Gamma \vdash^M e' : E$ *and* $\lvert e \rvert = e'$

Proof. By induction on the given derivation.
Part (1): Proceed by case analysis on the rule concluding $\Gamma \vdash^M e \Rightarrow E$.

- **Case** $$\dfrac{\Gamma \vdash^M e_1 \Rightarrow \big((A \to E) \rhd \epsilon_1\big) \qquad \Gamma \vdash v \Leftarrow A}{\Gamma \vdash^M (e_1\ v) \Rightarrow (E \text{ after } \epsilon_1)}\ \text{esyn-app}$$

$$\begin{array}{rl}
\Gamma \vdash^M e_1' : \big((A \to E) \rhd \epsilon_1\big) \text{ and } |e_1| = e_1' & \text{By inductive hypothesis} \\
\Gamma \vdash v' : A \text{ and } |v| = v' & \text{By Thm. D.1} \\
\Gamma \vdash^M (e_1'\ v') : (E \text{ after } \epsilon_1) & \text{By rule app} \\
|(e_1\ v)| = (|e_1|\ |v|) = (e_1'\ v') & \text{By definition of } |{-}| \\
☞\ \Gamma \vdash^M e' : (E \text{ after } \epsilon_1) \text{ and } |e| = e' & \text{where } e' = (e_1'\ v') \text{ and } e = (e_1\ v)
\end{array}$$

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

$$\begin{array}{rl}
\Gamma \vdash v' : \mathsf{Thk}[X]\,(C \rhd \epsilon) \text{ and } |v| = v' & \text{By Thm. D.1} \\
\Gamma \vdash^M \mathsf{force}(v') : \big(C \rhd (\langle\emptyset; X\rangle \text{ then } \epsilon)\big) & \text{By rule force} \\
|\mathsf{force}(v)| = \mathsf{force}(|v|) = \mathsf{force}(v') & \text{By definition of } |{-}| \\
& \text{and } |v| = v' \\
☞\ \Gamma \vdash^M e' : \big(C \rhd (\langle\emptyset; X\rangle \text{ then } \epsilon)\big) \text{ and } |e| = e' & \text{where } e' = \mathsf{force}(v') \text{ and } e = \mathsf{force}(v)
\end{array}$$

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

$$\begin{array}{rl}
\Gamma \vdash v' : \mathsf{Ref}[X]\,A \text{ and } |v| = v' & \text{By Thm. D.1} \\
\Gamma \vdash^M \mathsf{get}(v') : (\mathbf{F}\,A) \rhd \langle\emptyset; X\rangle & \text{By rule get} \\
|\mathsf{get}(v)| = \mathsf{get}(|v|) = \mathsf{get}(v') & \text{By the definition of } |{-}| \\
& \text{and } |v| = v' \\
☞\ \Gamma \vdash^M e' : (\mathbf{F}\,A) \rhd \langle\emptyset; X\rangle \text{ and } |e| = e' & \text{where } e' = \mathsf{get}(v') \text{ and } e = \mathsf{get}(v)
\end{array}$$

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

$$\begin{array}{rl}
\Gamma \vdash v_M' : (\mathbf{Nm} \overset{\mathbf{Nm}}{\to} \mathbf{Nm})[M] \text{ and } |v_M| = v_M' & \text{By Thm. D.1} \\
\Gamma \vdash v' : \mathsf{Nm}[i] \text{ and } |v| = v' & \text{By Thm. D.1} \\
\Gamma \vdash^N (v_M'\ v') : \mathbf{F}\,(\mathsf{Nm}[M(i)]) \rhd \langle\emptyset; \emptyset\rangle & \text{By rule name-app} \\
|(v_M\ v)| = (|v_M|\ |v|) = (v_M'\ v') & \text{By definition of } |{-}|;\ |v_M| = v_M';\ |v| = v' \\
☞\ \Gamma \vdash^M e' : \mathbf{F}\,(\mathsf{Nm}[M(i)]) \rhd \langle\emptyset; \emptyset\rangle \text{ and } |e| = e' & \text{where } e' = (v_M'\ v') \text{ and } e = (v_M\ v)
\end{array}$$

- **Case** $$\dfrac{\Gamma \vdash^M e \Rightarrow (\forall a : \gamma.\, E) \qquad \Gamma \vdash i : \gamma}{\Gamma \vdash^M e[i] \Rightarrow [i/a]E}\ \text{esyn-}\forall\text{IndexElim}$$

$\Gamma \vdash^M e' : (\forall a : \gamma.\, E)$ and $\lvert e\rvert = e'$	By inductive hypothesis
$\Gamma \vdash i : \gamma$	Given
☞ $e[i] = \lvert e'\rvert$	By $\lvert e\rvert = e'$
☞ $\Gamma \vdash^M e' : [i/a]E$	By rule AllIndexElim

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

  Similar to the syn-AllIndexElim case.

- **Case** $\dfrac{\Gamma \vdash^M e_1 \Leftarrow E}{\Gamma \vdash^M (e_1 : E) \Rightarrow E}$ esyn-anno

$\Gamma \vdash^M e_1' : E$ and $\lvert e_1\rvert = e_1'$	By inductive hypothesis
$\lvert e_1 : E\rvert = \lvert e_1\rvert = e_1'$	By the definition of $\lvert-\rvert$ and $\lvert e_1\rvert = e_1'$
☞ $\Gamma \vdash^M e' : E$ and $\lvert e\rvert = e'$	where $e' = e_1'$ and $e = (e_1 : E)$

Part (2): Proceed by case analysis on the rule concluding $\Gamma \vdash^M e \Leftarrow E$.

- **Case** $\dfrac{\Gamma \vdash^M e \Rightarrow (C \rhd \epsilon_1) \qquad \epsilon_1 \preceq \epsilon_2}{\Gamma \vdash^M e \Leftarrow (C \rhd \epsilon_2)}$ echk-sub

  By i.h. and rule etype-sub.

- **Case** $\dfrac{\begin{array}{l}\Gamma \vdash v \Rightarrow (A_1 \times A_2)\\ \Gamma, x_1 : A_1, x_2 : A_2 \vdash^M e_1 \Leftarrow E\end{array}}{\Gamma \vdash^M \mathsf{split}(v, x_1.x_2.e_1) \Leftarrow E}$ echk-split

$\Gamma \vdash v' : (A_1 \times A_2)$ and $\lvert v\rvert = v'$	By Thm. D.1
$\Gamma, x_1 : A_1, x_2 : A_2 \vdash^M e_1' : E$ and $\lvert e_1\rvert = e_1'$	By inductive hypothesis
$\Gamma \vdash^M \mathsf{split}(v, x_1.x_2.e_1') : E$	By rule split
$\lvert \mathsf{split}(v, x_1.x_2.e_1)\rvert = \mathsf{split}(\lvert v\rvert, x_1.x_2.\lvert e_1\rvert)$	
$= \mathsf{split}(v', x_1.x_2.e_1')$	By definition of $\lvert-\rvert$ and $\lvert v\rvert = v', \lvert e_1\rvert = e_1'$
☞ $\Gamma \vdash^M e' : E$ and $\lvert e\rvert = e'$	where $e' = \mathsf{split}(v', x_1.x_2.e_1')$ and $e = \mathsf{split}(v, x_1.x_2.e_1)$

- **Case** $\dfrac{\Gamma \vdash v \Rightarrow (A_1 + A_2) \qquad \begin{array}{l}\Gamma, x_1 : A_1 \vdash^M e_1 \Leftarrow E\\ \Gamma, x_2 : A_2 \vdash^M e_2 \Leftarrow E\end{array}}{\Gamma \vdash^M \mathsf{case}(v, x_1.e_1, x_2.e_2) \Leftarrow E}$ echk-case

$$\begin{array}{rll}
\Gamma \vdash v' : (A_1 + A_2) \text{ and } |v| = v' & \text{By Thm. D.1} \\
\Gamma, x_1 : A_1 \vdash^M e_1' : E \text{ and } |e_1| = e_1' & \text{By inductive hypothesis} \\
\Gamma, x_2 : A_2 \vdash^M e_2' : E \text{ and } |e_2| = e_2' & \text{By inductive hypothesis} \\
\Gamma \vdash^M \mathsf{case}(v', x_1.e_1', x_2.e_2') : E & \text{By rule case} \\
|\mathsf{case}(v, x_1.e_1, x_2.e_2)| = \mathsf{case}(|v|, x_1.|e_1|, x_2.|e_2|) & \text{By definition of } |-| \\
= \mathsf{case}(v', x_1.e_1', x_2.e_2') & \text{Since } |v| = v', |e_1| = e_1', |e_2| = e_2'
\end{array}$$

☞ $\Gamma \vdash^M e' : E$ and $|e| = e'$ where $e' = \mathsf{case}(v', x_1.e_1', x_2.e_2')$ and $e = \mathsf{case}(v, x_1.e_1, x_2.e_2)$

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

$$\begin{array}{rl}
\Gamma \vdash v' : A \text{ and } |v| = v' & \text{By Thm. D.1} \\
\Gamma \vdash^M \mathtt{ret}(v') : \big((\mathbf{F}\,A) \rhd \langle\emptyset;\emptyset\rangle\big) & \text{By rule ret} \\
|\mathtt{ret}(v)| = \mathtt{ret}(|v|) = \mathtt{ret}(v') & \text{By definition of } |-| \\
& \text{ and } |v| = v'
\end{array}$$

☞ $\Gamma \vdash^M e' : \big((\mathbf{F}\,A) \rhd \langle\emptyset;\emptyset\rangle\big)$ and $|e| = e'$ where $e' = \mathtt{ret}(v')$ and $e = \mathtt{ret}(v)$

- **Case** $\dfrac{\Gamma \vdash^M e_1 \Rightarrow (\mathbf{F}\,A) \rhd \epsilon_1 \qquad \Gamma, x : A \vdash^M e_2 \Leftarrow (C \rhd \epsilon_2)}{\Gamma \vdash^M \mathtt{let}(e_1, x.e_2) \Leftarrow \big(C \rhd (\epsilon_1 \text{ then } \epsilon_2)\big)}$ echk-let

$$\begin{array}{rl}
\Gamma \vdash^M e_1' : (\mathbf{F}\,A) \rhd \epsilon_1 \text{ and } |e_1| = e_1' & \text{By inductive hypothesis} \\
\Gamma, x : A \vdash^M e_2' : (C \rhd \epsilon_2) \text{ and } |e_2| = e_2' & \text{By inductive hypothesis} \\
\Gamma \vdash^M \mathtt{let}(e_1', x.e_2') : \big(C \rhd (\epsilon_1 \text{ then } \epsilon_2)\big) & \text{By rule let} \\
|\mathtt{let}(e_1, x.e_2)| = \mathtt{let}(|e_1|, x.|e_2|) & \text{By definition of } |-| \\
= \mathtt{let}(e_1', x.e_2') & \text{Since } |e_1| = e_1', |e_2| = e_2'
\end{array}$$

☞ $\Gamma \vdash^M e' : \big(C \rhd (\epsilon_1 \text{ then } \epsilon_2)\big)$ and $|e| = e'$ where $e' = \mathtt{let}(e_1', x.e_2')$ and $e = \mathtt{let}(e_1, x.e_2)$

- **Case** $\dfrac{\Gamma, x : A \vdash^M e_1 \Leftarrow E}{\Gamma \vdash^M (\lambda x.\, e_1) \Leftarrow \big((A \to E) \rhd \langle\emptyset;\emptyset\rangle\big)}$ echk-lam

$$\begin{array}{rl}
\Gamma, x : A \vdash^M e_1' : E \text{ and } |e_1| = e_1' & \text{By inductive hypothesis} \\
\Gamma \vdash^M (\lambda x.\, e_1') : \big((A \to E) \rhd \langle\emptyset;\emptyset\rangle\big) & \text{By rule lam} \\
|(\lambda x.\, e_1)| = (\lambda x.\, |e_1|) = (\lambda x.\, e_1') & \text{By definition of } |-| \\
& \text{ and } |e_1| = e_1'
\end{array}$$

☞ $\Gamma \vdash^M e' : \big((A \to E) \rhd \langle\emptyset;\emptyset\rangle\big)$ and $|e| = e'$ where $e' = (\lambda x.\, e_1')$ and $e = (\lambda x.\, e_1)$

- **Case** $\dfrac{\Gamma \vdash v \Leftarrow \mathsf{Nm}[X] \qquad \Gamma \vdash^M e_1 \Leftarrow E_1}{\Gamma \vdash^M \mathtt{thunk}(v, e_1) \Leftarrow \big(\mathbf{F}\,(\mathsf{Thk}[M(X)]\,E_1)\big) \rhd \langle M(X);\emptyset\rangle}$ echk-thunk

Let $E = \big(\mathbf{F}\,(\mathsf{Thk}[M(X)]\,E_1)\big) \rhd \langle M(X); \emptyset\rangle$	Assumption
$\exists\, v'$ such that $\Gamma \vdash v' : \mathsf{Nm}[X]$ and $\lvert v\rvert = v'$	By Thm. D.1
$\exists\, e_1'$ such that $\Gamma \vdash^M e_1' : E$ and $\lvert e_1\rvert = e_1'$	By inductive hypothesis
$\Gamma \vdash^M \mathtt{thunk}(v', e_1') : E$	By rule thunk
$\lvert\mathtt{thunk}(v, e_1)\rvert = \mathtt{thunk}(\lvert v\rvert, \lvert e_1\rvert) = \mathtt{thunk}(v', e_1')$	By definition of $\lvert-\rvert$ and $\lvert v\rvert = v'$, $\lvert e_1\rvert = e_1'$

☞ $\Gamma \vdash^M e' : E$ and $\lvert e\rvert = e'$ where $e' = \mathtt{thunk}(v', e_1')$ and $e = \mathtt{thunk}(v, e_1)$

- **Case**
$$\frac{\Gamma \vdash v_1 \Leftarrow \mathsf{Nm}[X] \qquad \Gamma \vdash v_2 \Leftarrow A}{\Gamma \vdash^M \mathtt{ref}(v_1, v_2) \Leftarrow \big(\mathbf{F}\,(\mathsf{Ref}[M(X)]\,A)\big) \rhd \langle M(X); \emptyset\rangle}\ \text{echk-ref}$$

$\exists\, v_1'$ such that $\Gamma \vdash v_1' : \mathsf{Nm}[X]$ and $\lvert v_1\rvert = v_1'$	By Thm. D.1
$\exists\, v_2'$ such that $\Gamma \vdash v_2' : A$ and $\lvert v_2\rvert = v_2'$	By Thm. D.1
$\Gamma \vdash^M \mathtt{ref}(v_1', v_2') : \big(\mathbf{F}\,(\mathsf{Ref}[M(X)]\,A)\big) \rhd \langle M(X); \emptyset\rangle$	By rule ref
$\lvert\mathtt{ref}(v_1, v_2)\rvert = \mathtt{ref}(\lvert v_1\rvert, \lvert v_2\rvert) = \mathtt{ref}(v_1', v_2')$	By definition of $\lvert-\rvert$

☞ $\Gamma \vdash^M e' : \big(\mathbf{F}\,(\mathsf{Ref}[M(X)]\,A)\big) \rhd \langle M(X); \emptyset\rangle$
and $\lvert e\rvert = e'$ where $e' = \mathtt{ref}(v_1', v_2')$ and $e = \mathtt{ref}(v_1, v_2)$

- **Case**
$$\frac{\Gamma \vdash v \Rightarrow (\mathbf{Nm} \overset{\mathsf{Nm}}{\Rightarrow} \mathbf{Nm})[N'] \qquad \Gamma \vdash^{N \circ N'} e_1 \Leftarrow C \rhd \langle W; R\rangle}{\Gamma \vdash^N \mathsf{scope}(v, e_1) \Leftarrow C \rhd \langle W; R\rangle}\ \text{echk-scope}$$

$\Gamma \vdash v' : \mathbf{Nm} \overset{\mathsf{Nm}}{\Rightarrow} \mathbf{Nm}$ and $\lvert N'\rvert = v'$	By inductive hypothesis
$\Gamma \vdash^{N \circ N'} e_1' : C \rhd \langle W; R\rangle$ and $\lvert e_1\rvert = e_1'$	By inductive hypothesis
$\Gamma \vdash^N \mathsf{scope}(v', e_1') : C \rhd \langle W; R\rangle$	By rule scope
$\lvert\mathsf{scope}(v, e_1)\rvert = \mathsf{scope}(\lvert v\rvert, \lvert e_1\rvert) = \mathsf{scope}(v', e_1')$	By definition of $\lvert-\rvert$; $\lvert v\rvert = v'$; $\lvert e_1'\rvert = e_1$

☞ $\Gamma \vdash^M e' : C \rhd \langle W; R\rangle$ and $\lvert e\rvert = e'$ where $e' = \mathsf{scope}(v', e_1')$ and $e = \mathsf{scope}(v, e_1)$

- **Case**
$$\frac{\Gamma, a : \gamma \vdash^M t \Leftarrow E}{\Gamma \vdash^M t \Leftarrow (\forall a : \gamma.\, E)}\ \text{echk-}\forall\text{IndexIntro}$$

$\Gamma, a : \gamma \vdash^M t' : E$ and $\lvert t\rvert = t'$	By inductive hypothesis
$\Gamma \vdash^M t' : (\forall a : \gamma.\, E)$	By rule AllIndexIntro
☞ $\Gamma \vdash^M e' : (\forall a : \gamma.\, E)$ and $\lvert e\rvert = e'$	where $e' = t'$ and $e = t$

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

$\exists\, t'$ such that $\Gamma, a : \gamma \vdash^M t' : E$ and $\lvert t\rvert = t'$	By inductive hypothesis
$\Gamma \vdash^M t' : (\forall \alpha : K.\, E)$	By rule AllIntro
☞ $\Gamma \vdash^M e' : (\forall \alpha : K.\, E)$ and $\lvert e\rvert = e'$	where $e' = t'$ and $e = t$

- **Case**
$$\frac{\Gamma \vdash^M e \Rightarrow E_1 \qquad E_1 = E_2}{\Gamma \vdash^M e \Leftarrow E_2}\ \text{echk-sub}$$

By i.h. and etype-sub. □

Theorem D.4 (Completeness of Bidirectional Computation Typing). *If* $\Gamma \vdash^{M} e : E$*, then there exist computations* e', e'' *such that*

(1) $\Gamma \vdash^{M} e' \Rightarrow E$ *and* $|e'| = e$
(2) $\Gamma \vdash^{M} e'' \Leftarrow E$ *and* $|e''| = e$

Proof. By induction on the derivation of $\Gamma \vdash^{M} e : E$.

- **Case** etype-sub: By i.h. and echk-sub.
- **Case**

$$\frac{\Gamma \vdash v : (A_1 \times A_2) \qquad \Gamma, x_1 : A_1, x_2 : A_2 \vdash^{M} e_1 : E}{\Gamma \vdash^{M} \mathsf{split}(v, x_1.x_2.e_1) : E}\ \text{split}$$

$$\begin{array}{rll}
\Gamma, x_1 : A_1, x_2 : A_2 \vdash e_1' \Leftarrow E \text{ and } e_1 = |e_1'| & & \text{By inductive hypothesis} \\
\Gamma \vdash v' \Rightarrow (A_1 \times A_2) \text{ and } v_1 = |v_1'| & & \text{By Thm. D.2} \\
\Gamma \vdash^{M} \mathsf{split}(v', x_1.x_2.e_1') \Leftarrow E & & \text{By chk-split} \\
\Gamma \vdash^{M} (\mathsf{split}(v', x_1.x_2.e_1') : E) \Rightarrow E & & \text{By syn-anno} \\
|(\mathsf{split}(v', x_1.x_2.e_1') : E)| & = |\mathsf{split}(v', x_1.x_2.e_1')| & \text{By definition of } |-| \\
|\mathsf{split}(v', x_1.x_2.e_1')| & = \mathsf{split}(|v'|, x_1.x_2.|e_1'|) & \text{By definition of } |-| \\
\mathsf{split}(|v'|, x_1.x_2.|e_1'|) & = \mathsf{split}(v, x_1.x_2.e_1) & \text{Since } |v'| = v,\ |e_1'| = e_1
\end{array}$$

☞ $\Gamma \vdash e' \Rightarrow E$ and $|e'| = e$ where $e' = \mathsf{split}(v', x_1.x_2.e_1')$ and $e = \mathsf{split}(v, x_1.x_2.e_1)$

☞ $\Gamma \vdash e'' \Leftarrow E$ and $|e''| = e$ where $e'' = (\mathsf{split}(v', x_1.x_2.e_1') : E)$ and $e = \mathsf{split}(v, x_1.x_2.e_1)$

- **Case**

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

$$\begin{array}{rll}
\Gamma \vdash v' \Rightarrow (A_1 + A_2) \text{ and } |v'| = v & & \text{By Thm. D.2} \\
\Gamma, x_1 : A_1 \vdash^{M} e_1'' \Leftarrow E \text{ and } |e_1''| = e_1 & & \text{By inductive hypothesis} \\
\Gamma, x_2 : A_2 \vdash^{M} e_2'' \Leftarrow E \text{ and } |e_2''| = |e_2| & & \text{By inductive hypothesis} \\
\Gamma \vdash^{M} \mathsf{case}(v', x_1.e_1'', x_2.e_2'') \Leftarrow E & & \text{By rule chk-case} \\
\Gamma \vdash^{M} (\mathsf{case}(v', x_1.e_1'', x_2.e_2'') : E) \Rightarrow E & & \text{By rule chk-conv} \\
|(\mathsf{case}(v', x_1.e_1'', x_2.e_2'') : E)| & = |\mathsf{case}(v', x_1.e_1'', x_2.e_2'')| & \text{By definition of } |-| \\
|\mathsf{case}(v', x_1.e_1'', x_2.e_2'')| & = \mathsf{case}(|v'|, x_1.|e_1''|, x_2.|e_2''|) & \text{By definition of } |-| \\
\mathsf{case}(|v'|, x_1.|e_1''|, x_2.|e_2''|) & = \mathsf{case}(v, x_1.e_1, x_2.e_2) &
\end{array}$$

☞ $\Gamma \vdash^{M} e' \Rightarrow E$ and $|e'| = e$ where $e' = (\mathsf{case}(v', x_1.e_1'', x_2.e_2'') : E)$ and $e = \mathsf{case}(v, x_1.e_1, x_2.e_2)$

☞ $\Gamma \vdash^{M} e'' \Leftarrow E$ and $|e''| = e$ where $e'' = \mathsf{case}(v', x_1.e_1'', x_2.e_2'')$ and $e = \mathsf{case}(v, x_1.e_1, x_2.e_2)$

- **Case**

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

$$
\begin{array}{lll}
 & \text{Let } E = \big((\mathbf{F}\,A) \rhd \langle\emptyset;\emptyset\rangle\big) & \text{Assumption} \\
 & \exists\, v'' \text{ such that } \Gamma \vdash v'' \Leftarrow A \text{ and } |v''| = v & \text{By Thm. D.2} \\
 & \Gamma \vdash^{M} \mathtt{ret}(v'') \Leftarrow E & \text{By rule chk-ret} \\
 & \Gamma \vdash^{M} (\mathtt{ret}(v'') : E) \Rightarrow E & \text{By syn-anno} \\
 & |(\mathtt{ret}(v'') : E)| = |\mathtt{ret}(v'')| & \text{By definition of } |-| \\
 & |\mathtt{ret}(v'')| = \mathtt{ret}(|v''|) = \mathtt{ret}(v) & \text{By definition of } |-| \\
 & & \quad\text{and } |v''| = v \\
☞ & \Gamma \vdash^{M} e' \Rightarrow E \text{ and } |e'| = e & \text{where } e' = (\mathtt{ret}(v'') : E) \text{ and } e = \mathtt{ret}(v) \\
☞ & \Gamma \vdash^{M} e'' \Leftarrow E \text{ and } |e''| = e & \text{where } e'' = \mathtt{ret}(v'') \text{ and } e = \mathtt{ret}(v)
\end{array}
$$

- **Case** $\dfrac{\Gamma \vdash^{M} e_1 : (\mathbf{F}\,A) \rhd \epsilon_1 \qquad \Gamma, x : A \vdash^{M} e_2 : (C \rhd \epsilon_2)}{\Gamma \vdash^{M} \mathtt{let}(e_1, x.e_2) : \big(C \rhd (\epsilon_1 \text{ then } \epsilon_2)\big)}\ \text{let}$

$$
\begin{array}{lll}
 & \text{Let } E = \big(C \rhd (\epsilon_1 \text{ then } \epsilon_2)\big) & \text{Assumption} \\
 & \Gamma \vdash^{M} e_1' \Rightarrow (\mathbf{F}\,A) \rhd \epsilon_1 \text{ and } |e_1'| = e_1 & \text{By inductive hypothesis} \\
 & \Gamma, x : A \vdash^{M} e_2'' \Leftarrow (C \rhd \epsilon_2) \text{ and } |e_2''| = e_2 & \text{By inductive hypothesis} \\
 & \Gamma \vdash^{M} \mathtt{let}(e_1', x.e_2'') \Leftarrow E & \text{By rule chk-let} \\
 & \Gamma \vdash^{M} (\mathtt{let}(e_1', x.e_2'') : E) \Rightarrow E & \text{By rule chk-conv} \\
 & |(\mathtt{let}(e_1', x.e_2'') : E)| = |\mathtt{let}(e_1', x.e_2'')| & \text{By definition of } |-| \\
 & |\mathtt{let}(e_1', x.e_2'')| = \mathtt{let}(|e_1'|, x.|e_2''|) = \mathtt{let}(e_1, x.e_2) & \text{By definition of } |-| \\
 & & \quad\text{and } |e_1'| = e_1,\ |e_2''| = e_2 \\
☞ & \Gamma \vdash^{M} e' \Rightarrow E \text{ and } |e'| = e & \text{where } e' = (\mathtt{let}(e_1', x.e_2'') : E) \text{ and } e = \mathtt{let}(e_1, x.e_2) \\
☞ & \Gamma \vdash^{M} e'' \Leftarrow E \text{ and } |e''| = e & \text{where } e'' = \mathtt{let}(e_1', x.e_2'') \text{ and } e = \mathtt{let}(e_1, x.e_2)
\end{array}
$$

- **Case** $\dfrac{\Gamma, x : A \vdash^{M} e_1 : E_1}{\Gamma \vdash^{M} (\lambda x.\, e_1) : \big((A \to E_1) \rhd \langle\emptyset;\emptyset\rangle\big)}\ \text{lam}$

$$
\begin{array}{lll}
 & \text{Let } E = \big((A \to E_1) \rhd \langle\emptyset;\emptyset\rangle\big) & \text{Assumption} \\
 & \exists\, e_1'' \text{ such that } \Gamma, x : A \vdash^{M} e_1'' \Leftarrow E_1 \text{ and } |e_1''| = e_1 & \text{By inductive hypothesis} \\
 & \Gamma \vdash^{M} (\lambda x.\, e_1'') \Leftarrow E & \text{By rule chk-lam} \\
 & \Gamma \vdash^{M} ((\lambda x.\, e_1'') : E) \Rightarrow E & \text{By syn-anno} \\
 & |(\lambda x.\, e_1'')| = (\lambda x.\, |e_1''|) = (\lambda x.\, e_1) & \text{By definition of } |-| \\
 & & \quad\text{and } |e_1''| = e_1 \\
☞ & \Gamma \vdash^{M} e' \Rightarrow E \text{ and } |e'| = e & \text{where } e' = ((\lambda x.\, e_1'') : E) \text{ and } e = (\lambda x.\, e_1) \\
☞ & \Gamma \vdash^{M} e'' \Leftarrow E \text{ and } |e''| = e & \text{where } e'' = (\lambda x.\, e_1'') \text{ and } e = (\lambda x.\, e_1)
\end{array}
$$

- **Case** $\dfrac{\Gamma \vdash^{M} e_1 : \big((A \to E) \rhd \epsilon_1\big) \qquad \Gamma \vdash v : A}{\Gamma \vdash^{M} (e_1\ v) : (E \text{ after } \epsilon_1)}\ \text{app}$

$$
\begin{array}{ll}
\exists\, e_1' \text{ such that } \Gamma \vdash^{M} e_1' \Rightarrow \big((A \to E) \rhd \epsilon_1\big) \text{ and } |e_1'| = e_1 & \text{By inductive hypothesis} \\
\exists\, v'' \text{ such that } \Gamma \vdash v'' \Leftarrow A \text{ and } |v''| = v & \text{By Thm. D.2} \\
\Gamma \vdash^{M} (e_1'\ v'') \Rightarrow (E \text{ after } \epsilon_1) & \text{By rule syn-app} \\
\Gamma \vdash^{M} (e_1'\ v'') \Leftarrow (E \text{ after } \epsilon_1) & \text{By rule chk-conv} \\
|(e_1'\ v'')| = (|e_1'|\ |v''|) = (e_1\ v) & \text{By the definition of } |-| \\
 & \quad\text{and } |e_1'| = e_1,\ |v''| = v
\end{array}
$$

☞ $\Gamma \vdash^M e' \Rightarrow E$ and $|e'| = e$ where $e' = (e_1' \; v'')$ and $e = (e_1 \; v)$
☞ $\Gamma \vdash^M e'' \Leftarrow E$ and $|e''| = e$ where $e'' = (e_1' \; v'')$ and $e = (e_1 \; v)$

- **Case**

$$\frac{\Gamma \vdash v : \mathsf{Nm}[X] \qquad \Gamma \vdash^M e_1 : E}{\Gamma \vdash^M \mathtt{thunk}(v, e_1) : \big(\mathbf{F}\,(\mathsf{Thk}[M(X)]\;E)\big) \rhd \langle M(X); \emptyset\rangle}\;\text{thunk}$$

$$\begin{array}{rl}
\text{Let } E = \big(\mathbf{F}\,(\mathsf{Thk}[M(X)]\;E)\big) \rhd \langle M(X); \emptyset\rangle & \text{Assumption} \\
\exists\, v'' \text{ such that } \Gamma \vdash v'' \Leftarrow \mathsf{Nm}[X] \text{ and } |v''| = v & \text{By Thm. D.2} \\
\exists\, e_1'' \text{ such that } \Gamma \vdash^M e_1'' \Leftarrow E \text{ and } |e_1''| = e_1 & \text{By inductive hypothesis} \\
\Gamma \vdash^M \mathtt{thunk}(v'', e_1'') \Leftarrow E & \text{By rule chk-thunk} \\
\Gamma \vdash^M (\mathtt{thunk}(v'', e_1'') : E) \Rightarrow E & \text{By rule syn-anno} \\
|(\mathtt{thunk}(v'', e_1'') : E)| = |\mathtt{thunk}(v'', e_1'')| & \text{By definition of } |{-}| \\
|\mathtt{thunk}(v'', e_1'')| = \mathtt{thunk}(|v''|, |e_1''|) = \mathtt{thunk}(v, e_1) & \text{By definition of } |{-}| \\
& \text{and } |v''| = v,\ |e_1''| = e_1
\end{array}$$

☞ $\Gamma \vdash^M e' \Rightarrow E$ and $|e'| = e$ where $e' = (\mathtt{thunk}(v'', e_1'') : E)$ and $e = \mathtt{thunk}(v, e_1)$
☞ $\Gamma \vdash^M e'' \Leftarrow E$ and $|e''| = e$ where $e'' = \mathtt{thunk}(v'', e_1'')$ and $e = \mathtt{thunk}(v, e_1)$

- **Case**

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

$$\begin{array}{rl}
\text{Let } E = \big(C \rhd (\langle \emptyset; X\rangle \text{ then } \epsilon)\big) & \text{Assumption} \\
\Gamma \vdash v' \Rightarrow \mathsf{Thk}[X]\;(C \rhd \epsilon) \text{ and } |v'| = v & \text{By Thm. D.2} \\
\Gamma \vdash^M \mathtt{force}(v') \Rightarrow E & \text{By rule syn-force} \\
\Gamma \vdash^M \mathtt{force}(v') \Leftarrow E & \text{By chk-conv} \\
|\mathtt{force}(v')| = \mathtt{force}(|v'|) = \mathtt{force}(v) & \text{By definition of } |{-}| \\
& \text{and } |v'| = v
\end{array}$$

☞ $\Gamma \vdash^M e' \Rightarrow E$ and $|e'| = e$ where $e' = \mathtt{force}(v')$ and $e = \mathtt{force}(v)$
☞ $\Gamma \vdash^M e'' \Leftarrow E$ and $|e''| = e$ where $e'' = \mathtt{force}(v')$ and $e = \mathtt{force}(v)$

- **Case**

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

$$\begin{array}{rl}
\text{Let } E = \big(\mathbf{F}\,(\mathsf{Ref}[M(X)]\;A)\big) \rhd \langle M(X); \emptyset\rangle & \text{Assumption} \\
\Gamma \vdash v_1'' \Leftarrow \mathsf{Nm}[X] \text{ and } |v_1''| = v_1 & \text{By Thm. D.2} \\
\Gamma \vdash v_2'' \Leftarrow A \text{ and } |v_2''| = v_2 & \text{By Thm. D.2} \\
\Gamma \vdash^M \mathtt{ref}(v_1'', v_2'') \Leftarrow E & \text{By rule chk-ref} \\
\Gamma \vdash^M (\mathtt{ref}(v_1'', v_2'') : E) \Rightarrow E & \text{By rule syn-anno} \\
|(\mathtt{ref}(v_1'', v_2'') : E)| = |\mathtt{ref}(v_1'', v_2'')| & \text{By definition of } |{-}| \\
|\mathtt{ref}(v_1'', v_2'')| = \mathtt{ref}(|v_1''|, |v_2''|) = \mathtt{ref}(v_1, v_2) & \text{By definition of } |{-}| \\
& \text{and } |v_1''| = v_1,\ |v_2''| = v_2
\end{array}$$

☞ $\Gamma \vdash^M e' \Rightarrow E$ and $|e'| = e$ where $e' = (\mathtt{ref}(v_1'', v_2'') : E)$ and $e = \mathtt{ref}(v_1, v_2)$
☞ $\Gamma \vdash^M e'' \Leftarrow E$ and $|e''| = e$ where $e'' = \mathtt{ref}(v_1'', v_2'')$ and $e = \mathtt{ref}(v_1, v_2)$

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

$$\begin{array}{rl}
\Gamma \vdash v' \Rightarrow \mathsf{Ref}[X]\,A \text{ and } |v'| = v & \text{By Thm. D.2} \\
\Gamma \vdash^{M} \mathsf{get}(v') \Rightarrow (\mathbf{F}\,A) \rhd \langle \emptyset; X\rangle & \text{By rule syn-get} \\
\Gamma \vdash^{M} \mathsf{get}(v') \Leftarrow (\mathbf{F}\,A) \rhd \langle \emptyset; X\rangle & \text{By rule chk-conv} \\
|\mathsf{get}(v')| = \mathsf{get}(|v'|) = \mathsf{get}(v) & \text{By definition of } |-| \\
& \text{and } |v'| = v
\end{array}$$

☞ $\Gamma \vdash^{M} e' \Rightarrow (\mathbf{F}\,A) \rhd \langle \emptyset; X\rangle$ and $|e'| = e$ where $e' = \mathsf{get}(v')$ and $e = \mathsf{get}(v)$
☞ $\Gamma \vdash^{M} e'' \Leftarrow (\mathbf{F}\,A) \rhd \langle \emptyset; X\rangle$ and $|e''| = e$ where $e'' = \mathsf{get}(v')$ and $e = \mathsf{get}(v)$

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

$$\begin{array}{rl}
\Gamma \vdash v'_M \Rightarrow (\mathbf{Nm} \overset{\mathbf{Nm}}{\to} \mathbf{Nm})[M] \text{ and } |v'_M| = v_M & \text{By Thm. D.2} \\
\Gamma \vdash v' \Rightarrow \mathsf{Nm}[i] \text{ and } |v'| = v & \text{By Thm. D.2} \\
\Gamma \vdash^{N} (v'_M\ v') \Rightarrow \mathbf{F}\,(\mathsf{Nm}[M(i)]) \rhd \langle \emptyset; \emptyset\rangle & \text{By rule syn-name-app} \\
\Gamma \vdash^{N} (v'_M\ v') \Leftarrow \mathbf{F}\,(\mathsf{Nm}[M(i)]) \rhd \langle \emptyset; \emptyset\rangle & \text{By rule chk-conv} \\
|(v'_M\ v')| = (|v'_M|\ |v'|) = (v_M\ v) & \text{By definition of } |-| \\
& \text{and } |v'_M| = v_M,\ |v'| = v
\end{array}$$

☞ $\Gamma \vdash^{N} e' \Rightarrow \mathbf{F}\,(\mathsf{Nm}[M(i)]) \rhd \langle \emptyset; \emptyset\rangle$ and $|e'| = e$ where $e' = (v'_M\ v')$ and $e = (v_M\ v)$
☞ $\Gamma \vdash^{N} e'' \Leftarrow \mathbf{F}\,(\mathsf{Nm}[M(i)]) \rhd \langle \emptyset; \emptyset\rangle$ and $|e''| = e$ where $e'' = (v'_M\ v')$ and $e = (v_M\ v)$

- **Case** $$\dfrac{\Gamma \vdash v : (\mathbf{Nm} \overset{\mathbf{Nm}}{\to} \mathbf{Nm})[N'] \qquad \Gamma \vdash^{N \circ N'} e_1 : C \rhd \langle W; R\rangle}{\Gamma \vdash^{N} \mathsf{scope}(v, e_1) : C \rhd \langle W; R\rangle}\ \text{scope}$$

$$\begin{array}{rl}
\text{Let } E = C \rhd \langle W; R\rangle & \text{Assumption} \\
\exists v'' \text{ such that }, \Gamma \vdash v'' \Rightarrow (\mathbf{Nm} \overset{\mathbf{Nm}}{\to} \mathbf{Nm})[N'] \text{ and } |v''| = v & \text{By Thm. D.2} \\
\exists e'_1 \text{ such that }, \Gamma \vdash^{N \circ N'} e'_1 \Leftarrow E \text{ and } |e'_1| = e_1 & \text{By inductive hypothesis} \\
\Gamma \vdash^{N} \mathsf{scope}(v'', e'_1) \Leftarrow E & \text{By rule chk-scope} \\
\Gamma \vdash^{N} (\mathsf{scope}(v'', e'_1) : E) \Rightarrow E & \text{By rule syn-anno}
\end{array}$$

$$\begin{array}{rl}
|(\mathsf{scope}(v'', e'_1) : E)| = |(\mathsf{scope}(v'', e'_1)| & \text{By definition of } |-| \\
|(\mathsf{scope}(v'', e'_1)| = \mathsf{scope}(|v''|, |e'_1|) = \mathsf{scope}(v, e_1) & \text{By definition of } |-|
\end{array}$$

☞ $\Gamma \vdash^{M} e' \Rightarrow E$ and $|e'| = e$ where $e' = (\mathsf{scope}(v'', e'_1) : E)$ and $e = \mathsf{scope}(v, e_1)$
☞ $\Gamma \vdash^{M} e'' \Leftarrow E$ and $|e''| = e$ where $e'' = \mathsf{scope}(v'', e'_1)$ and $e = \mathsf{scope}(v, e_1)$

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

$$\begin{array}{rll}
\exists\ t'' \text{ such that } \Gamma, a:\gamma \vdash^{M} t'' \Leftarrow E \text{ and } |t''| = t & & \text{By inductive hypothesis} \\
\Gamma \vdash^{M} t'' \Leftarrow (\forall a:\gamma.\, E) & & \text{By rule chk-AllIndexIntro} \\
\Gamma \vdash^{M} (t'' : (\forall a:\gamma.\, E)) \Rightarrow (\forall a:\gamma.\, E) & & \text{By rule syn-anno} \\
|(t'' : (\forall a:\gamma.\, E))| = |t''| = t & & \text{By definition of } |{-}| \\
& & \quad \text{and } |t''| = t \\
☞\ \Gamma \vdash^{M} e' \Rightarrow (\forall a:\gamma.\, E) \text{ and } |e'| = e & \text{where } e' = (t'' : (\forall a:\gamma.\, E)) \text{ and } e = t & \\
☞\ \Gamma \vdash^{M} e'' \Leftarrow (\forall a:\gamma.\, E) \text{ and } |e''| = e & \text{where } e'' = t'' \text{ and } e = t &
\end{array}$$

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

$$\begin{array}{rll}
& \exists\ e' \text{ such that } \Gamma \vdash^{M} e' \Rightarrow (\forall a:\gamma.\, E) \text{ and } |e'| = e & \text{By inductive hypothesis} \\
& \Gamma \vdash i : \gamma & \text{Given} \\
& \Gamma \vdash^{M} e' \Rightarrow [i/a]E & \text{By rule syn-AllIndexElim} \\
& \Gamma \vdash^{M} e' \Leftarrow [i/a]E & \text{By rule chk-conv} \\
☞ & \Gamma \vdash^{M} e'[i] \Rightarrow [i/a]E \text{ and } |e'[i]| = e & \\
☞ & \Gamma \vdash^{M} e'[i] \Leftarrow [i/a]E \text{ and } |e'[i]| = e &
\end{array}$$

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

  Similar to the AllIndexIntro case.

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

  Similar to the AllIndexElim case.
- **Case** etype-∃IndexElim: By i.h. and echk-∃IndexElim. □

E NAME TERM LANGUAGE

We define a restricted *name term* language for computing larger names from smaller names. This language consists of the following:

- Syntax for *names*, *name terms* and *sorts* (Fig. 8 in Sec. 3.2).
- Name term sorting: A judgment that assigns sorts to name terms (Fig. 9 in Sec. 3.2).
- Big-step evaluation for name terms: A judgment that assigns *name term values* to *name terms* (Fig. 9 in Sec. 3.2).
- Semantic definition of equivalent and disjoint name terms (Sec. E.1).
- Logical proof rules for equivalent and disjoint name terms: Two judgements that should be sound with respect to the semantic definitions of equivalence and disjointness (Fig. 24 and Fig. 25).

The first three items are each described in Sec. 3.2. We define the last two items in this section.

E.1 Semantic equivalence and disjointness

Below, we define semantic equivalence and disjointness of (sorted) name terms. We define these semantic properties inductively, based on the common sort of the name terms. In this sense, these definitions can be viewed as instances of logical relations.

We define contexts Γ that relate two variables; each declaration either asserts that a and b are equivalent, or disjoint. We write $\Gamma.1$ and $\Gamma.2$ for the projection of a relational Γ into an ordinary Γ suitable for the left-hand ($\Gamma.1$) or right-hand ($\Gamma.2$) sides. Also, we write *flip*(Γ) for the operation of exchanging a and b in each declaration: *flip*$((a \perp b : \gamma)) = (b \perp a : \gamma)$, so that *flip*$(\Gamma).1 = \Gamma.2$ and *flip*$(\Gamma).2 = \Gamma.1$.

Substitutions	σ	$::=$	$\cdot \mid \sigma, N/a$
Relational sorting contexts	Γ	$::=$	$\cdot$
(Hypothetical variable equivalence)		$\mid$	$\Gamma, (a \equiv b : \gamma)$
(Hypothetical variable apartness)		$\mid$	$\Gamma, (a \perp b : \gamma)$

$$
\begin{array}{rcl}
(\cdot).1 &=& \cdot \\
(\Gamma, a \equiv b : \gamma).1 &=& (\Gamma).1, a : \gamma \\
(\Gamma, a \perp b : \gamma).1 &=& (\Gamma).1, a : \gamma
\end{array}
\qquad
\begin{array}{rcl}
(\cdot).2 &=& \cdot \\
(\Gamma, a \equiv b : \gamma).2 &=& (\Gamma).2, b : \gamma \\
(\Gamma, a \perp b : \gamma).2 &=& (\Gamma).2, b : \gamma
\end{array}
$$

Definition E.1 (Closing substitutions).
We define closing substitution pairs related by equivalence and disjointness assumptions in a context Γ. These definitions use and are used by the definitions below for equivalence and apartness of open terms.

- $\Vdash \sigma_1 \equiv \sigma_2 : \Gamma$ *means that* $(x \equiv y : \gamma) \in \Gamma$ *implies* $\big(\sigma_1(x) = N$ *and* $\sigma_2(y) = M$ *and* $\cdot \Vdash N \equiv M : \gamma\big)$
- $\Vdash \sigma_1 \perp \sigma_2 : \Gamma$ *means that* $(x \perp y : \gamma) \in \Gamma$ *implies* $\big(\sigma_1(x) = N$ *and* $\sigma_2(y) = M$ *and* $\cdot \Vdash N \perp M : \gamma\big)$
- $\Vdash \sigma_1 \sim \sigma_2 : \Gamma$ *means that* $\Vdash \sigma_1 \equiv \sigma_2 : \Gamma$ *and* $\Vdash \sigma_1 \perp \sigma_2 : \Gamma$

Definition E.2 (Semantic equivalence). *We define* $\Gamma \Vdash M_1 \equiv M_2 : \gamma$ *as follows:*
$(\Gamma).1 \vdash M_1 : \gamma$ *and* $(\Gamma).2 \vdash M_2 : \gamma$ *and,*
for all σ_1, σ_2 *such that* $\Vdash \sigma_1 \equiv \sigma_2 : \Gamma$ *and* $[\sigma_1]M_1 \Downarrow V_1$ *and* $[\sigma_2]M_2 \Downarrow V_2$,
we have the following about V_1 *and* V_2*:*

Sort (γ)	*Values* V_1 *and* V_2 *of sort* γ *are equivalent, written* $\Vdash V_1 \equiv V_2 : \gamma$
1	*Always*
Nm	*When* $V_1 = n_1$ *and* $V_2 = n_2$ *and* $n_1 = n_2$ *(identical binary trees)*
$\gamma_1 * \gamma_2$	*When* $V_1 = (V_{11}, V_{12})$ *and* $V_1 = (V_{21}, V_{22})$ *and* $\Vdash V_{11} \equiv V_{21} : \gamma_1$ *and* $\Vdash V_{21} \equiv V_{22} : \gamma_2$
$\gamma_1 \overset{\mathbf{Nm}}{\to} \gamma_2$	*When* $V_1 = \lambda a_1. M_1$ *and* $V_2 = \lambda a_2. M_2$, *and for all name terms* $\Vdash N_1 \equiv N_2 : \gamma_1$, $[N_1/a_1]M_1 \Downarrow W_1$ *and* $[N_2/a_2]M_2 \Downarrow W_2$ *implies* $\Vdash W_1 \equiv W_2 : \gamma_2$

Definition E.3 (Semantic apartness). *We define* $\Gamma \Vdash M_1 \perp M_2 : \gamma$ *as follows:*
$(\Gamma).1 \vdash M_1 : \gamma$ *and* $(\Gamma).2 \vdash M_2 : \gamma$ *and,*
for all σ_1, σ_2 *such that* $\Vdash \sigma_1 \sim \sigma_2 : \Gamma$ *and* $[\sigma_1]M_1 \Downarrow V_1$ *and* $[\sigma_2]M_2 \Downarrow V_2$,
we have the following about V_1 *and* V_2*:*

$\boxed{\Gamma \vdash M \equiv N : \gamma}$ The name terms M and N are *equivalent* at sort γ

$$\frac{(M \equiv N : \gamma) \in \Gamma}{\Gamma \vdash M \equiv N : \gamma}\ \text{Eq-Var} \qquad \frac{(\Gamma).1 \vdash M : \gamma \quad (\Gamma).2 \vdash M : \gamma}{\Gamma \vdash M \equiv M : \gamma}\ \text{E-Refl} \qquad \frac{\mathit{flip}(\Gamma) \vdash N \equiv M : \gamma}{\Gamma \vdash M \equiv N : \gamma}\ \text{E-Sym}$$

$$\frac{\Gamma \vdash M_1 \equiv M_2 : \gamma \qquad \Gamma \vdash M_2 \equiv M_3 : \gamma}{\Gamma \vdash M_1 \equiv M_3 : \gamma}\ \text{Eq-Trans}$$

$$\frac{\Gamma \vdash M_1 \equiv N_1 : \gamma_1 \qquad \Gamma \vdash M_2 \equiv N_2 : \gamma_2}{\Gamma \vdash (M_1, M_2) \equiv (N_1, N_2) : \gamma_1 * \gamma_2}\ \text{Eq-Pair}$$

$$\frac{\Gamma \vdash M_1 \equiv N_1 : \mathbf{Nm} \qquad \Gamma \vdash M_2 \equiv N_2 : \mathbf{Nm}}{\Gamma \vdash \langle\langle M_1, M_2\rangle\rangle \equiv \langle\langle N_1, N_2\rangle\rangle : \mathbf{Nm}}\ \text{Eq-Bin}$$

$$\frac{\Gamma, (a \equiv b : \gamma_1) \vdash M \equiv N : \gamma_2}{\Gamma \vdash \lambda a.\, M \equiv \lambda b.\, N : \gamma_1 \xrightarrow{\mathbf{Nm}} \gamma_2}\ \text{Eq-Lam} \qquad \frac{\Gamma \vdash M_1 \equiv N_1 : \gamma_1 \xrightarrow{\mathbf{Nm}} \gamma_2 \quad \Gamma \vdash M_2 \equiv N_2 : \gamma_1}{\Gamma \vdash M_1(M_2) \equiv N_1(N_2) : \gamma_2}\ \text{Eq-App}$$

$$\frac{\Gamma \vdash M_2 \equiv M_2' : \gamma_1 \qquad \Gamma, a \equiv a : \gamma_1 \vdash M_1 \equiv M_1' : \gamma_2}{\Gamma \vdash (\lambda a.\, M_1)M_2 \equiv [M_2'/a]M_1' : \gamma_2}\ \text{Eq-}\beta$$

Fig. 24. Deductive rules for showing that two name terms are equivalent

Sort (γ)	*Values* V_1 *and* V_2 *of sort* γ *are apart, written* $\Vdash V_1 \perp V_2 : \gamma$
1	*Always*
Nm	*When* $V_1 = n_1$ *and* $V_2 = n_2$ *and* $n_1 \neq n_2$ *(distinct binary trees)*
$\gamma_1 * \gamma_2$	*When* $V_1 = (V_{11}, V_{12})$ *and* $V_1 = (V_{21}, V_{22})$ *and* $\Vdash V_{11} \perp V_{21} : \gamma_1$ *and* $\Vdash V_{21} \perp V_{22} : \gamma_2$
$\gamma_1 \xrightarrow{\mathbf{Nm}} \gamma_2$	*When* $V_1 = \lambda a_1.\, M_1$ *and* $V_2 = \lambda a_2.\, M_2$, *and for all name terms* $\vdash N_1 : \gamma_1$ *and* $\vdash N_2 : \gamma_1$, $[N_1/a_1]M_1 \Downarrow W_1$ *and* $[N_2/a_2]M_2 \Downarrow W_2$ *implies* $\Vdash W_1 \perp W_2 : \gamma_2$

E.2 Metatheory of name term language

Some lemmas in this section are missing complete proofs and should be considered conjectures (Lemma E.1 (Projections of syntactic equivalence)–Lemma E.8 (Reflexivity of name term evaluation)).

Lemma E.1 (Projections of syntactic equivalence).
If $\Gamma \vdash M_1 \equiv M_2 : \gamma$, *then* $\Gamma.1 \vdash M_1 : \gamma$ *and* $\Gamma.2 \vdash M_2 : \gamma$.

Lemma E.2 (Determinism of evaluation up to substitution).
If $\Gamma.1 \vdash M : \gamma$ *and* $\Gamma.2 \vdash M : \gamma$ *and* $\Vdash \sigma_1 \equiv \sigma_2 : \Gamma$ *and* $[\sigma_1]M \Downarrow V_1$ *and* $[\sigma_2]M \Downarrow V_2$ *then there exists* V *such that* $V_1 = [\sigma_1]V$ *and* $V_2 = [\sigma_2]V$.

Lemma E.3 (Reflexivity of semantic equivalence).

(1) If $\vdash M : \gamma$ *then* $\Vdash M \equiv M : \gamma$.
(2) If $\Gamma.1 \vdash V : \gamma$ *and* $\Gamma.2 \vdash V : \gamma$ *and* $\Vdash \sigma_1 \equiv \sigma_2 : \Gamma$ *and* $V_1 = [\sigma_1]V$ *and* $V_2 = [\sigma_2]V$ *then* $\Vdash V_1 \equiv V_2 : \gamma$.

$\boxed{\Gamma \vdash M \perp N : \gamma}$ The name terms M and N are *apart* at sort γ

$$\frac{(a \perp b : \gamma) \in \Gamma}{\Gamma \vdash a \perp b : \gamma}\ \text{Var} \qquad \frac{\mathit{flip}(\Gamma) \vdash N \perp M : \gamma}{\Gamma \vdash M \perp N : \gamma}\ \text{D-Sym}$$

$$\frac{\Gamma \vdash M_1 \equiv M_2 : \gamma \qquad \Gamma \vdash M_2 \perp M_3 : \gamma}{\Gamma \vdash M_1 \perp M_3 : \gamma}\ \text{D-trans}$$

$$\frac{\Gamma \vdash M_1 \perp N_1 : \mathbf{Nm}}{\Gamma \vdash \langle\langle M_1, M_2 \rangle\rangle \perp \langle\langle N_1, N_2 \rangle\rangle : \mathbf{Nm}}\ \text{D-Bin}_1 \qquad \frac{\Gamma \vdash M_2 \perp N_2 : \mathbf{Nm}}{\Gamma \vdash \langle\langle M_1, M_2 \rangle\rangle \perp \langle\langle N_1, N_2 \rangle\rangle : \mathbf{Nm}}\ \text{D-Bin}_2$$

$$\frac{\Gamma \vdash M_1 \equiv M_2 : \mathbf{Nm}}{\Gamma \vdash \langle\langle M_2, N \rangle\rangle \perp M_1 : \mathbf{Nm}}\ \text{D-EqTag}_1 \qquad \frac{\Gamma \vdash N_1 \equiv N_2 : \mathbf{Nm}}{\Gamma \vdash \langle\langle M, N_1 \rangle\rangle \perp N_2 : \mathbf{Nm}}\ \text{D-EqTag}_2$$

$$\frac{\Gamma \vdash M \perp N : \gamma_2}{\Gamma \vdash \lambda a.\, M \perp \lambda b.\, N : \gamma_1 \overset{\mathsf{Nm}}{\rightarrow} \gamma_2}\ \text{D-Lam} \qquad \frac{\Gamma \vdash M_1 \perp N_1 : \gamma_1 \overset{\mathsf{Nm}}{\rightarrow} \gamma_2}{\Gamma \vdash M_1(M_2) \perp N_1(N_2) : \gamma_2}\ \text{D-App}$$

$$\frac{\Gamma.1 \vdash M_2 : \gamma_2 \qquad \Gamma.1, a : \gamma_2 \vdash M_1 : \gamma \qquad \Gamma \vdash [M_2/a]M_1 \perp N : \gamma}{\Gamma \vdash (\lambda a.\, M_1)\ M_2 \perp N : \gamma}\ \text{D-}\beta$$

Fig. 25. Deductive rules for showing that two name terms are apart

Lemma E.4 (Type safety). *If* $\Gamma \vdash M : \gamma$ *and* $[\sigma]M \Downarrow [\sigma]V$ *then* $\Gamma \vdash V : \gamma$.

Lemma E.5 (Symmetry of semantic equivalence).

(1) *If* $\Vdash \sigma_1 \equiv \sigma_2 : \Gamma$ *then* $\Vdash \sigma_2 \equiv \sigma_1 : \mathit{flip}(\Gamma)$.
(2) *If* $\Vdash V_1 \equiv V_2 : \gamma$ *then* $\Vdash V_2 \equiv V_1 : \gamma$.
(3) *If* $\cdot \Vdash M_1 \equiv M_2 : \gamma$ *then* $\cdot \Vdash M_2 \equiv M_1 : \gamma$.
(4) *If* $\Gamma \Vdash M_1 \equiv M_2 : \gamma$ *then* $\mathit{flip}(\Gamma) \Vdash M_2 \equiv M_1 : \gamma$.

Lemma E.6 (Evaluation respects semantic equivalence).
If $\Gamma \Vdash M \equiv N : \gamma$ *and* $\Vdash \sigma_1 \equiv \sigma_2 : \Gamma$ *and* $[\sigma_1]M \Downarrow V_1$ *then there exists* V_2 *such that* $[\sigma_2]N \Downarrow V_2$ *and* $\Vdash V_1 \equiv V_2 : \gamma$.

Lemma E.7 (Closing substitutions respect syntactic equivalence).
If $\Vdash \sigma_1 \equiv \sigma_2 : \Gamma$ *and* $\Gamma \vdash M_1 \equiv M_2 : \gamma$ *then* $\cdot \Vdash [\sigma_1]M_1 \equiv [\sigma_2]M_2 : \gamma$.

Lemma E.8 (Reflexivity of name term evaluation).
If $\Gamma.1 \vdash M : \gamma$ *and* $\Gamma.2 \vdash M : \gamma$ *and* $\Vdash \sigma_1 \equiv \sigma_2 : \Gamma$ *and* $[\sigma_1]M \Downarrow V_1$ *and* $[\sigma_2]M \Downarrow V_2$ *then* $\Vdash V_1 \equiv V_2 : \gamma$.

Lemma E.9 (Transitivity of value equivalence).
If $\vdash V_1 : \gamma$ *and* $\vdash V_2 : \gamma$ *and* $\Vdash V_1 \equiv V_2 : \gamma$ *and* $\Vdash V_2 \equiv V_3 : \gamma$ *then* $\Vdash V_1 \equiv V_3 : \gamma$.

Proof. Uses strong normalization. □

Conjecture E.10 (Soundness of deductive equivalence).
If $\Gamma \vdash M_1 \equiv M_2 : \gamma$ *then* $\Gamma \Vdash M_1 \equiv M_2 : \gamma$.

Proof. By induction on the given derivation.

Case $$\frac{(a \equiv b : \gamma) \in \Gamma}{\Gamma \vdash a \equiv b : \gamma}\ \text{Eq-Var}$$

By definition of closing substitutions.

Case $$\frac{(\Gamma).1 \vdash M : \gamma \qquad (\Gamma).2 \vdash M : \gamma}{\Gamma \vdash M \equiv M : \gamma}\ \text{Eq-Refl}$$

By Lemma E.2 (Determinism of evaluation up to substitution), Lemma E.4 (Type safety), and Lemma E.3 (Reflexivity of semantic equivalence).

Case $$\frac{\Gamma \vdash N \equiv M : \gamma}{\Gamma \vdash M \equiv N : \gamma}\ \text{Eq-Sym}$$

By Lemma E.5 (Symmetry of semantic equivalence).

Case $$\frac{\Gamma \vdash M_1 \equiv M_2 : \gamma \qquad \Gamma \vdash M_2 \equiv M_3 : \gamma}{\Gamma \vdash M_1 \equiv M_3 : \gamma}\ \text{Eq-Trans}$$

By idempotency of flipping relational contexts, Lemma E.6 (Evaluation respects semantic equivalence), inductive hypotheses on the two given subderivations, Lemma E.5 (Symmetry of semantic equivalence), and Lemma E.9 (Transitivity of value equivalence).

Case $$\frac{\Gamma \vdash M_1 \equiv N_1 : \gamma_1 \qquad \Gamma \vdash M_2 \equiv N_2 : \gamma_2}{\Gamma \vdash (M_1, M_2) \equiv (N_1, N_2) : \gamma_1 * \gamma_2}\ \text{Eq-Pair}$$

By the definition of substitution and the i.h.

Case $$\frac{\Gamma \vdash M_1 \equiv N_1 : \mathbf{Nm} \qquad \Gamma \vdash M_2 \equiv N_2 : \mathbf{Nm}}{\Gamma \vdash \langle\!\langle M_1, M_2 \rangle\!\rangle \equiv \langle\!\langle N_1, N_2 \rangle\!\rangle : \mathbf{Nm}}\ \text{Eq-Bin}$$

By the definition of substitution and the i.h.

Case $$\frac{\Gamma, (a \equiv b : \gamma_1) \vdash M \equiv N : \gamma_2}{\Gamma \vdash \lambda a.\, M \equiv \lambda b.\, N : \gamma_1 \overset{\mathbf{Nm}}{\to} \gamma_2}\ \text{Eq-Lam}$$

By transposition of substitutions and the i.h.

Case $$\frac{\begin{array}{l}\Gamma \vdash M_1 \equiv N_1 : \gamma_1 \overset{\mathbf{Nm}}{\to} \gamma_2 \\ \Gamma \vdash M_2 \equiv N_2 : \gamma_1\end{array}}{\Gamma \vdash M_1(M_2) \equiv N_1(N_2) : \gamma_2}\ \text{Eq-App}$$

By definition of substitution and inversion (teval-app) of resulting derivations, the inductive hypothesis on the two given syntactic equivalence subderivations (of Eq-App), and definition of semantic equivalence of arrow-sorted values, we get the result.

Case $$\frac{\Gamma \vdash M_2 \equiv M_2' : \gamma_1 \qquad \Gamma, a \equiv a : \gamma_1 \vdash M_1 \equiv M_1' : \gamma_2}{\Gamma \vdash (\lambda a.\, M_1)M_2 \equiv [M_2'/a]M_1' : \gamma_2}\ \text{Eq-}\beta$$

Fix $\Vdash \sigma_1 \equiv \sigma_2 : \Gamma$. Suppose $[\sigma_1]((\lambda a.\, M_1)M_2) \Downarrow V_1$ and $[\sigma_2]([M_2'/a]M_1') \Downarrow V_2$. We need to show $\Vdash V_1 \equiv V_2 : \gamma_2$. By the definition of substitution and inversion of teval-app, $[\sigma_1]M_2 \Downarrow V$ and $[V/a]([\sigma_1]M_1) \Downarrow V_1$ for some V. Hence, because $\Gamma, a \equiv a : \gamma_1$, we have $[\sigma_1, V/a]M_1 \Downarrow V_1$.

Rewrite $[\sigma_2]([M_2'/a]M_1') \Downarrow V_2$ as $[\sigma_2, [\sigma_2]M_2'/a]M_1' \Downarrow V_2$. By Lemma E.7 (Closing substitutions respect syntactic equivalence), $\cdot \vdash V \equiv [\sigma_2]M_2' : \gamma_1$. Therefore,

$$\Vdash (\sigma_1, V/a) \equiv (\sigma_2, [\sigma_2]M_2'/a) : (\Gamma, a \equiv a : \gamma_1)$$

By the inductive hypothesis on $\Gamma, a \equiv a : \gamma_1 \vdash M_1 \equiv M_1' : \gamma_2$, we get $\Vdash V_1 \equiv V_2 : \gamma_2$. □

Conjecture E.11 (Soundness of deductive disjointness). *If* $\Gamma \vdash M_1 \perp M_2 : \gamma$ *then* $\Gamma \Vdash M_1 \perp M_2 : \gamma$.

Conjecture E.12 (Completeness of deductive equivalence). *If* $\Gamma \Vdash M_1 \equiv M_2 : \gamma$ *then* $\Gamma \vdash M_1 \equiv M_2 : \gamma$.

Conjecture E.13 (Completeness of deductive disjointness). *If* $\Gamma \Vdash M_1 \perp M_2 : \gamma$ *then* $\Gamma \vdash M_1 \perp M_2 : \gamma$.

$\boxed{i \to_\beta j}$ Index i reduces to index j

$$\frac{}{\mathsf{prj}_b(j_1, j_2) \to_\beta j_b}\ \text{reduce-proj} \qquad \frac{}{(\lambda a.\, i)(j) \to_\beta [j/a]i}\ \text{reduce-app}$$

$$\frac{}{M[\emptyset] \rightsquigarrow \emptyset}\ \text{reduce-map-empty} \qquad \frac{M(N) \Downarrow_M V}{M[\{N\}] \to_\beta \{V\}}\ \text{reduce-map-single}$$

$$\frac{M \text{ injective}}{M[X_1 \perp X_2] \to_\beta M[X_1] \perp M[X_2]}\ \text{reduce-map-}\perp$$

$$\frac{}{M[X_1 \cup X_2] \to_\beta M[X_1] \cup M[X_2]}\ \text{reduce-map-}\cup$$

$$\frac{}{M^*[j] \to_\beta M[M^*[j]]}\ \text{reduce-kleene-outer} \qquad \frac{}{M^*[j] \to_\beta M^*[M[j]]}\ \text{reduce-kleene-inner}$$

Fig. 26. Reduction rules for indices

F INDEX TERM LANGUAGE

We define a restricted *index term* language for describing name sets and functions that relate them. This language consists of the following:

- Syntax for *index terms*, and (additional) *sorts* (Fig. 10 in Sec. 4.1).
- Index term sorting, which assigns sorts to index terms (Fig. 12 in Sec. 4.1).
- Reduction rules for index terms, deriving the judgment $i \to_\beta j$ (Fig. 26).
- Semantic definitions of equivalent and disjoint index terms (Sec. F.2).
- Deductive rules for equivalent and disjoint index terms, which should be sound with respect to the semantic definitions of equivalence and disjointness (Fig. 29 and Fig. 30).

The first two items are defined in Sec. 4.1. The remaining items are defined here.

F.1 Index reduction

We write $i \to_\beta j$ for the one-step head reduction of index i to j, defined in Fig. 26. This is essentially β-reduction for functions, products, and mapping name sets (with map $M[X]$ distributing over union operators).

Based on one-step head reduction, we define one-step reduction $i \to_{CC} j$ (reducing anywhere within an index, not only at the head) and multi-step reduction $i \twoheadrightarrow j$:

Definition F.1 (Reduction for indices)**.**

(1) Let $\to_{CC}$ be the congruence closure of $\to_\beta$.
(2) Let $\twoheadrightarrow$ be the reflexive-transitive closure of $\to_{CC}$.

F.2 Semantic equivalence and apartness of index terms

Definition F.2 (Closing substitutions for index terms)**.**
We define closing substitution pairs related by equivalence and disjointness assumptions in a context Γ. These definitions use and are used by the definitions below for equivalence and apartness of open terms.

- $\Vdash \sigma_1 \equiv \sigma_2 : \Gamma$ *iff* $(x \equiv y : \gamma) \in \Gamma$ *implies* $\big(\sigma_1(x) = i$ *and* $\sigma_2(y) = j$ *and* $\cdot \Vdash i \equiv j : \gamma\,\big)$
- $\Vdash \sigma_1 \perp \sigma_2 : \Gamma$ *iff* $(x \perp y : \gamma) \in \Gamma$ *implies* $\big(\sigma_1(x) = i$ *and* $\sigma_2(y) = j$ *and* $\cdot \Vdash i \perp j : \gamma\,\big)$
- $\Vdash \sigma_1 \sim \sigma_2 : \Gamma$ *iff* $\big(\Vdash \sigma_1 \equiv \sigma_2 : \Gamma$ *and* $\Vdash \sigma_1 \perp \sigma_2 : \Gamma\,\big)$

$\boxed{\Gamma \vdash M \in X}$ Name term M is a member of name set X, assuming X val

$$\frac{\Gamma \vdash (\{M\} \perp Y) \equiv X : \mathbf{NmSet}}{\Gamma \vdash M \in X}\text{Membership}_1 \qquad \frac{\Gamma \vdash (\{M\} \cup Y) \equiv X : \mathbf{NmSet}}{\Gamma \vdash M \in X}\text{Membership}_2$$

$\boxed{\Gamma \Vdash M \in X}$ Name term M is a member of name set X, assuming X val

$$\frac{\Gamma \Vdash (\{M\} \perp Y) \equiv X : \mathbf{NmSet}}{\Gamma \Vdash M \in X}\text{Membership}_1 \qquad \frac{\Gamma \Vdash (\{M\} \cup Y) \equiv X : \mathbf{NmSet}}{\Gamma \Vdash M \in X}\text{Membership}_2$$

$\boxed{\Gamma \vdash M \notin X}$ The name of name term M is *not* a member of name set X, assuming X val

$$\frac{\Gamma \vdash (\{M\} \perp X) \equiv Y : \mathbf{NmSet}}{\Gamma \vdash M \notin X}\text{NonMembership}$$

$\boxed{\Gamma \Vdash M \notin X}$ The name of name term M is *not* a member of name set X, assuming X val

$$\frac{\Gamma \Vdash (\{M\} \perp X) \equiv Y : \mathbf{NmSet}}{\Gamma \Vdash M \notin X}\text{NonMembership}$$

Fig. 27. Name term membership

Definition F.3 (Index normal form). *An index i is in* index normal form *iff none of the rules*

reduce-proj, reduce-app, reduce-map{-empty, -single, -$\perp$, -$\cup$}

can be applied (anywhere within i).

Under this definition, normal forms are not unique: the rules reduce-kleene-inner and reduce-kleene-outer can reduce normal forms. In effect, these are normal forms "modulo Kleene closure".

Definition F.4 (Semantic equivalence of index terms). *We define $\Gamma \Vdash i_1 \equiv i_2 : \gamma$ as follows:*
$(\Gamma).1 \vdash i_1 : \gamma$ and $(\Gamma).2 \vdash i_2 : \gamma$ and,
for all σ_1, σ_2, and j_1 such that $\Vdash \sigma_1 \equiv \sigma_2 : \Gamma$ and $[\sigma_1]i_1 \twoheadrightarrow j_1$ where j_1 is in index normal form,
there exists j_2 such that j_2 is in index normal form and $[\sigma_2]i_2 \twoheadrightarrow j_2$,
we have the following about j_1 and j_2:

Sort γ	*Indices j_1 and j_2 of sort γ are equivalent, written $\Vdash j_1 \equiv j_2 : \gamma$*
$\mathbf{1}$	*Always*
$\mathbf{NmSet}$	*When $(\Vdash M \in j_1$ if and only if $\Vdash M \in j_2)$*
$\gamma_1 * \gamma_2$	*When $j_1 = (j_{11}, j_{12})$ and $j_2 = (j_{21}, j_{22})$ and $\Vdash j_{11} \equiv j_{21} : \gamma_1$ and $\Vdash j_{12} \equiv j_{22} : \gamma_2$*
$\gamma_1 \overset{idx}{\to} \gamma_2$	*When $j_1 = \lambda a_1. X_1$ and $j_2 = \lambda a_2. X_2$, and for all name terms $\Vdash Y_1 \equiv Y_2 : \gamma_1$, $(\lambda a_1. X_1)(Y_1) \twoheadrightarrow Z_1$ and $(\lambda a_2. X_2)(Y_2) \twoheadrightarrow Z_2$ implies $\Vdash Z_1 \equiv Z_2 : \gamma_2$*

Definition F.5 (Semantic apartness of index terms). *We define $\Gamma \Vdash i_1 \perp i_2 : \gamma$ as follows:*
$(\Gamma).1 \vdash i_1 : \gamma$ and $(\Gamma).2 \vdash i_2 : \gamma$ and,
for all σ_1, σ_2, and j_1 such that $\Vdash \sigma_1 \equiv \sigma_2 : \Gamma$ and $[\sigma_1]i_1 \twoheadrightarrow j_1$ where j_1 is in index normal form,

there exists j_2 *such that* j_2 *is in index normal form and* $[\sigma_2]i_2 \twoheadrightarrow j_2$,
we have the following about j_1 *and* j_2*:*

Sort (γ)	*Index values* j_1 *and* j_2 *of sort* γ *are apart, written* $\Vdash j_1 \perp j_2 : \gamma$
$\mathbf{1}$	*Always*
NmSet	*When* $(\Vdash M \in j_1$ *implies* $\Vdash M \notin j_2)$ *and* $(\Vdash M \in j_2$ *implies* $\Vdash M \notin j_1)$
$\gamma_1 * \gamma_2$	*When* $j_1 = (j_{11}, j_{12})$ *and* $j_2 = (j_{21}, j_{22})$ *and* $\Vdash j_{11} \perp j_{21} : \gamma_1$ *and* $\Vdash j_{12} \perp j_{22} : \gamma_2$
$\gamma_1 \xrightarrow{idx} \gamma_2$	*When* $j_1 = \lambda a_1. X_1$ *and* $V_2 = \lambda a_2. X_2$, *and for all name terms* $\vdash Y_1 : \gamma_1$ *and* $\vdash Y_2 : \gamma_1$, $(\lambda a_1. X_1)(Y_1) \twoheadrightarrow Z_1$ *and* $(\lambda a_2. X_2)(Y_2) \twoheadrightarrow Z_2$ *implies* $\Vdash Z_1 \perp Z_2 : \gamma_2$

$$
\begin{aligned}
\textit{extract-assns}(\cdot) &= \cdot \\
\textit{extract-assns}(\Gamma, P) &= \textit{extract-assns}(\Gamma), P \\
\textit{extract-assns}(\Gamma, \mathbf{tt}) &= \textit{extract-assns}(\Gamma) \\
\textit{extract-assns}(\Gamma, (P_1 \textbf{ and } \cdots \textbf{ and } P_n)) &= \textit{extract-assns}(\Gamma), P_1, \ldots, P_n \\
&\quad \text{for } n \geq 1 \text{, where each } P_k \\
&\quad\ \text{has the form } i \perp j : \gamma \text{ or } i \equiv j : \gamma \\
\textit{extract-assns}(\Gamma, \mathcal{Z}) &= \textit{extract-assns}(\Gamma) \text{ where } \mathcal{Z} \text{ is not a proposition}
\end{aligned}
$$

$$
\begin{aligned}
\textit{extract-ctx}(\cdot) &= \cdot \\
\textit{extract-ctx}(\Gamma, a : \gamma) &= \textit{extract-ctx}(\Gamma), (a \equiv a : \gamma) \\
\textit{extract-ctx}(\Gamma, \alpha : \text{type}) &= \textit{extract-ctx}(\Gamma) \\
\textit{extract-ctx}(\Gamma, d : K) &= \textit{extract-ctx}(\Gamma) \\
\textit{extract-ctx}(\Gamma, p : \cdots) &= \textit{extract-ctx}(\Gamma) \\
\textit{extract-ctx}(\Gamma, x : A) &= \textit{extract-ctx}(\Gamma) \\
\textit{extract-ctx}(\Gamma, P) &= \textit{extract-ctx}(\Gamma)
\end{aligned}
$$

$$\textit{extract}(\Gamma) \ = \ (\textit{extract-assns}(\Gamma); \textit{extract-ctx}(\Gamma))$$

Fig. 28. Extraction function on typing contexts

The next two definitions bridge the gap with the type system, in which contexts Γ_T also include propositions P. It is defined assuming that *extract*(Γ_T) (defined in Figure 28) has given us some propositions $P_1, \ldots, P_n$ and a relational context Γ.

Definition F.6 (Extended semantic equivalence of index terms)**.**
We define $P_1, \ldots, P_n; \Gamma \Vdash i \equiv j : \gamma$ *to hold if and only if*

$$\mathcal{J}(P_1) \textit{ and} \cdots \textit{ and } \mathcal{J}(P_n) \textit{ implies } \Gamma \Vdash i \equiv j : \gamma$$

where $\mathcal{J}(i \mathrel{\Theta} j : \gamma) = (\Gamma \Vdash i \mathrel{\Theta} j : \gamma)$.

Definition F.7 (Extended semantic apartness of index terms)**.**
We define $P_1, \ldots, P_n; \Gamma \Vdash i \perp j : \gamma$ *to hold if and only if*

$$\mathcal{J}(P_1) \textit{ and} \cdots \textit{ and } \mathcal{J}(P_n) \textit{ implies } \Gamma \Vdash i \perp j : \gamma$$

where $\mathcal{J}(i \,\Theta\, j : \gamma) = (\Gamma \Vdash i \,\Theta\, j : \gamma)$.

When a typing context is weakened, semantic equivalence and apartness under the extracted context continue to hold:

Lemma F.1 (Weakening of semantic equivalence and apartness). *If* $\mathit{extract}(\Gamma_T) \Vdash i_1 \equiv i_2 : \gamma$ *(respectively* $i_1 \perp i_2 : \gamma$*) then* $\mathit{extract}(\Gamma_T, \Gamma_T') \Vdash i_1 \equiv i_2 : \gamma$ *(respectively* $i_1 \perp i_2 : \gamma$.

Proof. By induction on Γ_T'.
We prove the $\equiv$ part; the $\perp$ part is similar.

- If $\Gamma_T' = \cdot$, we already have the result.
- If $\Gamma_T' = (\Gamma', P)$ then:
  By i.h., $\mathit{extract}(\Gamma_T, \Gamma') \Vdash i_1 \equiv i_2 : \gamma$.
  That is, $\mathit{extract\text{-}assns}(\Gamma_T, \Gamma'); \mathit{extract\text{-}ctx}(\Gamma_T, \Gamma') \Vdash i_1 \equiv i_2 : \gamma$.
  By its definition, $\mathit{extract\text{-}ctx}(\Gamma_T, \Gamma', P) = \mathit{extract\text{-}ctx}(\Gamma_T, \Gamma', P)$.
  Therefore, we have $\mathit{extract\text{-}assns}(\Gamma_T, \Gamma'); \mathit{extract\text{-}ctx}(\Gamma_T, \Gamma', P) \Vdash i_1 \equiv i_2 : \gamma$.
  Adding an assumption before the semicolon only supplements the antecedent in Def. F.6, so
  $$\mathit{extract\text{-}assns}(\Gamma_T, \Gamma', P); \mathit{extract\text{-}ctx}(\Gamma_T, \Gamma', P) \Vdash i_1 \equiv i_2 : \gamma$$
  which was to be shown.
- If $\Gamma_T' = (\Gamma', a : \gamma)$ then by i.h.,
  $$\mathit{extract}(\Gamma_T, \Gamma') \Vdash i_1 \equiv i_2 : \gamma$$
  By definition of *extract-ctx*,
  $$\mathit{extract\text{-}ctx}(\Gamma_T, \Gamma', a : \gamma) = \mathit{extract\text{-}ctx}(\Gamma_T, \Gamma'), a \equiv a : \gamma$$
  By the i.h. and Def. F.4,
  $$(\mathit{extract\text{-}ctx}(\Gamma_T, \Gamma')).1 \vdash i_1 : \gamma$$
  We need to show that $(\mathit{extract\text{-}ctx}(\Gamma_T, \Gamma', a : \gamma)).1 \vdash i_1 : \gamma$, which follows by weakening on sorting. The ".2" part is similar.
  Since a does not occur in i_1 and i_2, applying longer substitutions that include a to i_1 and i_2 does not change them; thus, we get the same j_1 and j_2 as for Γ_T, Γ'.
- In the remaining cases of $\mathcal{Z}$ for $\Gamma_T' = (\Gamma', \mathcal{Z})$, neither *extract-assns* nor *extract-ctx* change, and the i.h. immediately gives the result. □

F.3 Deductive equivalence and apartness for index terms

$\boxed{\Gamma \vdash i \equiv j : \gamma}$ The index terms i and j are *equivalent* at sort γ

$$\frac{(i \equiv j : \gamma) \in \Gamma}{\Gamma \vdash i \equiv j : \gamma}\ \text{Eq-Var} \qquad \frac{(\Gamma).1 \vdash i : \gamma \qquad (\Gamma).2 \vdash i : \gamma}{\Gamma \vdash i \equiv i : \gamma}\ \text{E-Refl} \qquad \frac{\mathit{flip}(\Gamma) \vdash j \equiv i : \gamma}{\Gamma \vdash i \equiv j : \gamma}\ \text{E-Sym}$$

$$\frac{\Gamma \vdash i_1 \equiv j_1 : \gamma_1 \qquad \Gamma \vdash i_2 \equiv j_2 : \gamma_2}{\Gamma \vdash (i_1, i_2) \equiv (j_1, j_2) : \gamma_1 * \gamma_2}\ \text{Eq-Pair} \qquad \frac{\Gamma, (a \equiv b : \gamma_1) \vdash i \equiv j : \gamma_2}{\Gamma \vdash \lambda a.\, i \equiv \lambda b.\, j : \gamma_1 \xrightarrow{\text{idx}} \gamma_2}\ \text{Eq-Lam}$$

$$\frac{\begin{array}{l}\Gamma \vdash i_1 \equiv j_1 : \gamma_1 \xrightarrow{\text{idx}} \gamma_2 \\ \Gamma \vdash i_2 \equiv j_2 : \gamma_1\end{array}}{\Gamma \vdash i_1(i_2) \equiv j_1(j_2) : \gamma_2}\ \text{Eq-App} \qquad \frac{\begin{array}{l}(\Gamma).1, a : \gamma_2 \vdash i_1 : \gamma \\ (\Gamma).1 \vdash i_2 : \gamma_2 \\ \Gamma \vdash [i_2/a]i_1 \equiv j : \gamma\end{array}}{\Gamma \vdash (\lambda a.\, i_1)i_2 \equiv j : \gamma}\ \text{Eq-}\beta$$

$$\frac{}{\Gamma \vdash \emptyset \equiv \emptyset : \mathbf{NmSet}}\ \text{Eq-Empty} \qquad \frac{\Gamma \vdash M \equiv N : \mathbf{Nm}}{\Gamma \vdash \{M\} \equiv \{N\} : \mathbf{NmSet}}\ \text{Eq-Single}$$

$$\frac{\Gamma \vdash X_1 \equiv X_2 : \mathbf{NmSet} \qquad \Gamma \vdash Y_1 \equiv Y_2 : \mathbf{NmSet}}{\Gamma \vdash (X_1 \perp Y_1) \equiv (X_2 \perp Y_2) : \mathbf{NmSet}}\ \text{Eq-Apart}$$

$$\frac{\Gamma \vdash (X_2 \perp X_1) \equiv Y : \mathbf{NmSet}}{\Gamma \vdash (X_1 \perp X_2) \equiv Y : \mathbf{NmSet}}\ \text{Eq-Perm}$$

$$\frac{\Gamma \vdash M \equiv N : \mathbf{Nm} \xrightarrow{\mathbf{Nm}} \mathbf{Nm} \qquad \Gamma \vdash X \equiv Y : \mathbf{NmSet}}{\Gamma \vdash M[X] \equiv N[Y] : \mathbf{NmSet}}\ \text{Eq-Map}$$

$$\frac{\Gamma \vdash i \equiv j : \mathbf{Nm} \xrightarrow{\text{idx}} \mathbf{NmSet} \qquad \Gamma \vdash X \equiv Y : \mathbf{NmSet}}{\Gamma \vdash i[X] \equiv j[Y] : \mathbf{NmSet}}\ \text{Eq-FlatMap}$$

$$\frac{\Gamma \vdash i \equiv j : \mathbf{Nm} \xrightarrow{\text{idx}} \mathbf{NmSet} \qquad \Gamma \vdash X \equiv Y : \mathbf{NmSet}}{\Gamma \vdash i^*[X] \equiv j^*[Y] : \mathbf{NmSet}}\ \text{Eq-Star}$$

Fig. 29. Deductive rules for showing that two index terms are equivalent

$\boxed{\Gamma \vdash i \perp j : \gamma}$ The index terms i and j are *apart* at sort γ

$$\frac{(a \perp b : \gamma) \in \Gamma}{\Gamma \vdash a \perp b : \gamma}\ \text{Var} \qquad \frac{\mathit{flip}(\Gamma) \vdash j \perp i : \gamma}{\Gamma \vdash i \perp j : \gamma}\ \text{D-Sym}$$

$$\frac{\Gamma \vdash i_1 \perp j_1 : \gamma_1}{\Gamma \vdash (i_1, i_2) \perp (j_1, j_2) : \gamma_1 * \gamma_2}\ \text{D-Proj}_1 \qquad \frac{\Gamma \vdash i_2 \perp j_2 : \gamma_2}{\Gamma \vdash (i_1, i_2) \perp (j_1, j_2) : \gamma_1 * \gamma_2}\ \text{D-Proj}_2$$

$$\frac{\Gamma \vdash i \perp j : \gamma_2}{\Gamma \vdash \lambda a.\, i \perp \lambda b.\, j : \gamma_1 \xrightarrow{\text{idx}} \gamma_2}\ \text{D-Lam} \qquad \frac{\Gamma \vdash i_1 \perp j_1 : \gamma_1 \xrightarrow{\text{idx}} \gamma_2}{\Gamma \vdash i_1(i_2) \perp j_1(j_2) : \gamma_2}\ \text{D-App}$$

$$\frac{\Gamma \vdash [i_2/a]i_1 \perp j : \gamma \qquad \begin{array}{l}(\Gamma).1 \vdash i_2 : \gamma_2 \\ (\Gamma).1, a : \gamma_2 \vdash i_1 : \gamma\end{array}}{\Gamma \vdash (\lambda a.\, i_1)i_2 \perp j : \gamma}\ \text{D-}\beta$$

$$\frac{(\Gamma).2 \vdash X : \textbf{NmSet}}{\Gamma \vdash \emptyset \perp X : \textbf{NmSet}}\ \text{D-Empty} \qquad \frac{\Gamma \vdash M \perp N : \textbf{Nm}}{\Gamma \vdash \{M\} \perp \{N\} : \textbf{NmSet}}\ \text{D-Single}$$

$$\frac{\Gamma \vdash X_1 \perp Y : \textbf{NmSet} \qquad \Gamma \vdash X_2 \perp Y : \textbf{NmSet}}{\Gamma \vdash (X_1 \perp X_2) \perp Y : \textbf{NmSet}}\ \text{D-Apart} \qquad \frac{\Gamma \vdash M \perp N : \textbf{Nm} \xrightarrow{\textbf{Nm}} \textbf{Nm}}{\Gamma \vdash M[X] \perp N[Y] : \textbf{NmSet}}\ \text{D-Map}$$

$$\frac{\Gamma \vdash i \perp j : \textbf{Nm} \xrightarrow{\text{idx}} \textbf{NmSet}}{\Gamma \vdash i[X] \perp j[Y] : \textbf{NmSet}}\ \text{D-FlatMap1}$$

$$\frac{\Gamma \vdash i \equiv j : \textbf{Nm} \xrightarrow{\text{idx}} \textbf{NmSet} \qquad \Gamma \vdash X \perp Y : \textbf{NmSet}}{\Gamma \vdash i[X] \perp j[Y] : \textbf{NmSet}}\ \text{D-FlatMap2}$$

$$\frac{\Gamma \vdash i \perp j : \textbf{Nm} \xrightarrow{\text{idx}} \textbf{NmSet} \qquad \Gamma \vdash X \perp Y : \textbf{NmSet}}{\Gamma \vdash i^*[X] \perp j^*[Y] : \textbf{NmSet}}\ \text{D-Star}$$

Fig. 30. Deductive rules for showing that two index terms are apart

G NORMALIZATION FOR NAME TERMS

We write "M halts" when there exists V such that $M \Downarrow_M V$.

We write $\sigma : \Gamma$ when, for all $a \in \mathsf{dom}(\Gamma)$, we have $\cdot \vdash \sigma(a) : \Gamma(a)$. It follows that $\sigma(a)$ is closed.

Definition G.1 ($R_\gamma(M)$).

(1) $R_\gamma(M)$ if and only if $\gamma \neq (\gamma_1 \overset{\mathsf{Nm}}{\to} \gamma_2)$ and M halts.
(2) $R_{(\gamma_1 \overset{\mathsf{Nm}}{\to} \gamma_2)}(M)$ if and only if (i) M halts and
(ii) for all closed M', if $R_{\gamma_1}(M')$ then $R_{\gamma_2}(M\ M')$.

Lemma G.2 (Substitution). *If $\Gamma, a : \gamma_a \vdash M : \gamma$ and $\Gamma \vdash M_a : \gamma_a$ then $\Gamma \vdash [M_a/a]M : \gamma$.*

Proof. By induction on the derivation of $\Gamma, a : \gamma_a \vdash M : \gamma$. □

Lemma G.3 (Closedness). *If M is closed and $M \Downarrow_M V$ then V is closed.*

Proof. By induction on the derivation of $M \Downarrow_M V$. □

Lemma G.4 (Canonical Forms). *Suppose $\vdash V : \gamma$.*
(1) If $\gamma = \mathbf{Nm}$ then $V = n$.
(2) If $\gamma = (\gamma_1 \overset{\mathsf{Nm}}{\to} \gamma_2)$ then $V = (\lambda a.\, M_0)$ and $a : \gamma_1 \vdash M_0 : \gamma_2$.

Proof. By inspection of the given derivation. □

Lemma G.5 (Multiple Substitution). *If $\Gamma \vdash M : \gamma$ and $\sigma : \Gamma$ then $\vdash [\sigma]M : \gamma$.*

Proof. By induction on the length of σ, using Lemma G.2 (Substitution). □

Lemma G.6 (Type Preservation). *If $\vdash M : \gamma$ and $M \Downarrow_M V$ then $\vdash V : \gamma$.*

Lemma G.7 (Preservation). *If $R_\gamma(M)$ and $M \Downarrow_M V$ then $R_\gamma(V)$.*

Proof. By induction on γ.

If γ does not have the form $(\gamma_1 \overset{\mathsf{Nm}}{\to} \gamma_2)$, then the only requirement is to show there exists V' such that $V \Downarrow_M V'$. Let $V' = V$. Then $V \Downarrow_M V'$ by teval-value.

Otherwise, $\gamma = (\gamma_1 \overset{\mathsf{Nm}}{\to} \gamma_2)$, and we also have to show that for all closed M_1 such that $R_{\gamma_1}(M_1)$, it is the case that $R_{\gamma_2}(M\ M_1)$.

By definition of R, there exists V_1 such that $M_1 \Downarrow_M V_1$. By i.h., $R_{\gamma_1}(V_1)$.

$M \Downarrow_M V$	Above
$V = (\lambda a.\, M_0)$	By Lemma G.4 (Canonical Forms)
$M_1 \Downarrow_M V_1$	Above
$[V_1/a]M_0 \Downarrow_M V_2$	By i.h.
$M\ M_1 \Downarrow_M V_2$	By teval-app

□

Lemma G.8 (Normalization).
If $\Gamma \vdash M : \gamma$ and $\sigma : \Gamma$ and, for all $a \in \mathsf{dom}(\Gamma)$, we have $R_{\Gamma(a)}(\sigma(a))$, then $R_\gamma([\sigma]M)$.

Proof. By induction on the derivation of $\Gamma \vdash M : \gamma$.

- **Case**

$$\frac{}{\Gamma \vdash n : \mathbf{Nm}}\ \text{M-const}$$

$[\sigma]n = n$	By definition of $[\sigma](-)$
$n \Downarrow_M n$	By rule teval-value
$[\sigma]n \Downarrow_M n$	By above equation
n is a value	
☞ $R_{\mathbf{Nm}}([\sigma]M)$	By definition of R

- **Case** $\dfrac{(a : \gamma) \in \Gamma}{\Gamma \vdash a : \gamma}$ M-var

  We have $\Gamma(a) = \gamma$. It is given that $R_{\Gamma(a)}(\sigma(a))$. Since $\sigma(a) = [\sigma]a$, we have $R_\gamma[\sigma]a$, which was to be shown.

- **Case** $\dfrac{\Gamma, a : \gamma_1 \vdash M_0 : \gamma_2}{\Gamma \vdash (\lambda a.\, M_0) : (\gamma_1 \overset{\mathbf{Nm}}{\rightrightarrows} \gamma_2)}$ M-abs

  Suppose that, for some closed M', we have $R_{\gamma_1}(M')$. By the definition of R, that means there exists V' such that $M' \Downarrow_M V'$.
  We need to show $R_{\gamma_2}\big([\sigma]((\lambda a.\, M_0)\, M')\big)$.
  Let $\sigma_a = (\sigma, V'/a)$.

$R_{\gamma_1}(V')$	By Lemma G.7 (Preservation)
$\Gamma, a : \gamma_1 \vdash M_0 : \gamma_2$	Subderivation
$R_{\gamma_2}([\sigma_a]M_0)$	By i.h. with σ_a as σ
$(\lambda a.\, [\sigma]M_0) \Downarrow_M \lambda a.\, [\sigma]M_0$	By teval-value
$M' \Downarrow_M V'$	Above
$[\sigma_a]M_0 \Downarrow_M V$	By definition of R
$[\sigma_a]M_0 = [V'/a][\sigma]M_0$	By def. of subst.
$[V'/a][\sigma]M_0 \Downarrow_M V$	By above equation
$(\lambda a.\, [\sigma]M_0)\, M' \Downarrow_M V$	By teval-app
$M' = [\sigma]M'$	M' closed
$(\lambda a.\, [\sigma]M_0)\, [\sigma]M' \Downarrow_M V$	By above equation
$(\lambda a.\, [\sigma]M_0)\, [\sigma]M' = ([\sigma]\lambda a.\, M_0)\, [\sigma]M'$	By def. of subst.
$= [\sigma]\big((\lambda a.\, M_0)\, M'\big)$	By def. of subst.
$[\sigma]\big((\lambda a.\, M_0)\, M'\big) \Downarrow_M V$	By above equations
☞ $R_{\gamma_2}\big([\sigma]((\lambda a.\, M_0)\, M')\big)$	By definition of R

- **Case** $\dfrac{\Gamma \vdash M_1 : (\gamma' \overset{\mathbf{Nm}}{\rightrightarrows} \gamma) \qquad \Gamma \vdash M_2 : \gamma'}{\Gamma \vdash (M_1\ M_2) : \gamma}$ M-app

$R_{(\gamma' \overset{\mathbf{Nm}}{\rightrightarrows} \gamma)}([\sigma]M_1)$	By i.h.
$R_{\gamma'}([\sigma]M_2)$	By i.h.
$R_\gamma(([\sigma]M_1)\ [\sigma]M_2)$	By definition of R
☞ $R_\gamma([\sigma](M_1\ M_2))$	By def. of subst.

- **Case** $\dfrac{\Gamma \vdash M_1 : \mathbf{Nm} \qquad \Gamma \vdash M_2 : \mathbf{Nm}}{\Gamma \vdash \langle\langle M_1, M_2 \rangle\rangle : \mathbf{Nm}}$ M-bin

$R_{\mathbf{Nm}}([\sigma]M_1)$	By i.h.
$R_{\mathbf{Nm}}([\sigma]M_2)$	By i.h.
$[\sigma]M_1 \Downarrow_M V_1$	By $R_{\mathbf{Nm}}([\sigma]M_1)$
$\Gamma \vdash M_1 : \mathbf{Nm}$	Subderivation
$\vdash [\sigma]M_1 : \mathbf{Nm}$	By Lemma G.5 (Multiple Substitution)
$V_1 = n_1$	By Lemma G.6 (Type Preservation) and Lemma G.4 (Canonical Forms)
$[\sigma]M_2 \Downarrow_M n_2$	Similar
$\langle\langle [\sigma]M_1, [\sigma]M_2 \rangle\rangle \Downarrow_M \langle\langle n_1, n_2 \rangle\rangle$	By rule teval-bin
$[\sigma]\langle\langle M_1, M_2 \rangle\rangle \Downarrow_M \langle\langle n_1, n_2 \rangle\rangle$	By definition of $[\sigma](-)$
☞ $R_{\mathbf{Nm}}([\sigma]\langle\langle M_1, M_2 \rangle\rangle)$	By definition of R ◻

Theorem G.9 (Normalization). *If* $\vdash M : \gamma$ *then there exists* V *such that* $M \Downarrow_M V$.

Proof. By Lemma G.8 (Normalization), $R_\gamma(M)$.
By definition of R, there exists V such that $M \Downarrow_M V$. ◻