Typed Adapton:
Refinement types for incremental computations with precise names

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

Over the past decade, programming language techniques for incremental computation have demonstrated that incremental programs with precise, carefully chosen dynamic names for data and sub-computations can dramatically outperform non-incremental programs, as well as those using traditional memoization (without such names). However, prior work lacks a verification mechanism to solve the ambiguous name problem, the problem of statically enforcing precise names. We say that an allocated pointer name is precise for an evaluation derivation when it identifies at most one value or sub-computation, and ambiguous otherwise.

In this work, we define a refinement type system that gives practical static approximations to enforce precise, deterministic allocation names in otherwise functional programs. We show that this type system permits expressing familiar functional programs, and generic, composable library components. We prove that our type system enforces that well-typed programs name their values and sub-computations precisely, without ambiguity. Drawing closer to an implementation, we derive a bidirectional version of the type system, and prove that it corresponds to our declarative type system. A key challenge in implementing the bidirectional system is handling constraints over names, name terms and name sets; toward this goal, we give decidable, syntactic rules to guide these checks.

ACM Reference format:
DOI: 10.1145/nnnnnn.nnnnnnn

1 INTRODUCTION

A computation is incremental if repeating it with a changed input is faster than from-scratch recomputation. Noticing their ubiquity and complexity in modern software, programming language researchers have proposed creating incremental computations systematically, by augmenting non-incremental algorithms with general-purpose abstractions. In this paper, we focus on an unsolved problem that spans this research area: the ambiguous naming problem. In brief, general-purpose incremental computing techniques require the programmer to augment their algorithms to unambiguously name dynamic allocations and callgraph nodes. In a single evaluation $D$ of an incremental algorithm, a name is unambiguous (or precise) when it identifies at most one value or sub-computation within $D$; a name is ambiguous if it identifies two or more distinct values and/or subcomputations within $D$.

In past literature on general-purpose incremental computation, precise names play a critical role in an algorithm’s performance, though these works lack a mechanism for static enforcement (Acar 2005; Acar et al. 2006a,b, 2009; Acar and Ley-Wild 2009; Hammer and Acar 2008; Hammer et al. 2009; Ley-Wild et al. 2008, 2009; Chen et al. 2012; Hammer et al. 2015). In each such system, a runtime library uses programmer-provided names to identify the nodes of a dynamic dependency graph that it maintains. The nodes of this graph consist of call
graph nodes, and (incrementally changing) reference cells that hold the input, output and intermediate data of the computation (generally, these cells hold inductive data structures, such as lists or trees).

Specifically, these names indirectly guide a change propagation algorithm’s decisions about when and how to overwrite information about previously cached computations in this graph. Depending on the specifics of these dynamic mechanisms, an ambiguous name can have various negative effects on an incremental computation, or may be explicitly forbidden. In some past systems, ambiguous names are dynamically detected, either forcing the system to fall back to a non-deterministic name choice (Acar et al. 2006a; Hammer and Acar 2008), or to signal an error and halt (Hammer et al. 2015). In scenarios with a non-deterministic fall-back mechanism, a name ambiguity carries the potential to degrade incremental performance, making it less responsive and asymptotically unpredictable in general (Acar 2005). To ensure that incremental performance gains are predictable, past work often merely assumes, without enforcement, that names are precise (Ley-Wild et al. 2009; Çiček et al. 2015, 2016). Other incremental computing techniques go further, forbidding ambiguous names and non-determinism; instead of using a non-deterministic fallback mechanism, these systems signal a dynamic error when the programmer mistakenly names values or computations ambiguously (Hammer et al. 2015).

This paper gives a general-purpose type and effect system to verify that explicitly-named allocations are precise. The types describe which first-class names appear within inductive and coinductive data structures. The effect system enforces that each name is written at most once within any well-typed computation that consumes it. In particular, we show that every well-typed computation consists of an unambiguous sequence of writes, where each written name appears at most once. Further, we show that every read name is unambiguous: it occurs after it is written, ensuring that each observation sees the latest written value.

To statically enforce these effect invariants, yet permit general programming patterns, we also develop static abstractions for generic naming strategies, in the form of stratified sub-languages for name terms and index terms, which describe restricted (decidable) computations over names and name sets, respectively. The particulars of Typed Adaptation focus on providing a typing discipline to the core calculus of (Nominal) Adaptation, a foundational semantics for demand-driven incremental computation with names (Hammer et al. 2014, 2015). However, these type and effect mechanisms are general, and also apply to each of the various implementations of self-adjusting computation (Acar 2005; Ley-Wild 2010; Hammer 2012; Chen et al. 2012).

Warm-up example: List map with names. To better appreciate the role of names, and the ambiguous naming problem, consider an often-used warm-up example for general-purpose incremental computing systems: mapping a functional list that changes over time. In particular, suppose that the incremental computation \( e \) is \( \lambda x. \text{list} \cdot \text{map} \ f \ x \), which maps a list using function \( f \). For illustration, we consider the four-element input lists \( I_1 = [1,2,3,5] \) and \( I_2 = [1,2,4,5] \) in the following diagram:

\[
\begin{align*}
(1 \overset{\alpha}{\mapsto} 2 \overset{\beta}{\mapsto} 3 \overset{\zeta}{\mapsto} 5 \overset{\delta}{\mapsto} \text{Nil}) & \quad \overset{\Delta I_1}{\mapsto} \quad e \quad \overset{\Delta O_1}{\mapsto} \quad (f \ 1 \overset{\alpha}{\mapsto} \ f \ 2 \overset{\beta}{\mapsto} \ f \ 3 \overset{\zeta}{\mapsto} \ f \ 5 \overset{\delta}{\mapsto} \text{Nil}) \\
\text{"write b with Cons(4,c)"} & \quad \overset{\text{propagate}}{\Delta O_1} \quad \overset{\text{"write b' with Cons(f 4,c')"}}{\Delta O_1} \\
(1 \overset{\alpha}{\mapsto} 2 \overset{\beta}{\mapsto} 4 \overset{\zeta}{\mapsto} 5 \overset{\delta}{\mapsto} \text{Nil}) & \quad \overset{\Delta I_2}{\mapsto} \quad e \quad \overset{\Delta O_2}{\mapsto} \quad (f \ 1 \overset{\alpha}{\mapsto} \ f \ 2 \overset{\beta}{\mapsto} \ f \ 4 \overset{\zeta}{\mapsto} \ f \ 5 \overset{\delta}{\mapsto} \text{Nil})
\end{align*}
\]

To employ incremental computing techniques, we enrich the input lists with names and named pointers, whose identities are drawn from name set \( X = \{a, b, c, d\} \). For simplicity, we initially combine names used for allocation and allocated pointer names into the single notion of a named list position. The benefit of naming list positions is that the input editor and the programmer of \( e \) can each describe spatial and computational correlations over

---

1 For expository purposes here, we temporarily employ the notion of a named list position, which combines the finer-grained notions of unallocated names (not associated with content) and allocated pointers (associated with content). In the next section, we tour a type structure for these lists that differentiates these notions, for both theoretical and practical reasons that will become clear later.
successive executions, respectively. Pictorially, the temporal correlations extend \textit{vertically}, relating \( I_1 \) to \( I_2 \) and relating \( O_1 \) to \( O_2 \). This vertical axis represents the input editor’s experience of time as they change the input, re-demand the computation, and observe the (updated) output. Meanwhile, the computational correlations of \( e \) extend \textit{horizontally}, relating \( I_1 \) to \( O_1 \) and relating \( I_2 \) to \( O_2 \).

Vertically, the input editor uses names to concisely describe \( \Delta I_1 \), which identifies a spatial position in the input to change (namely tail position \( b \)), and another position whose content is reused in this change, and remains unchanged (namely tail position \( c \), and its content, the rest of the list). Likewise, the names in \( X' = \{ a', b', c', d' \} \) play a critical role in describing the corresponding output change \( \Delta O_1 \). This output change mirrors the input change, overwriting \( f \) with \( f' \), at list position \( b' \). The relationship between the input and output change is given indirectly by the author of \( e \), who never describes changes directly. Beyond specifying the ordinary behavior of mapping a functional list, the author of \( e \) uses the input list names to name the output list positions (by mapping each name in \( X \) to a distinct name in \( X' \)). In the next section, we give a detailed listing for this program \( e \). As with Adapton programs in general, this program resembles an ordinary typed \( \lambda \)-calculus term, enriched with some abstractions for \textit{write-once, multi-read named references} (for naming inductive data structures, such as the input and output lists), and \textit{write-once, multi-force named thunks} (for naming callgraph nodes, and coinductive structures, such as lazy lists).

Horizontally, Adapton computes \( \Delta O_1 \) from \( \Delta I_1 \) by using the program \( e \), the dependence graph it created from running \( e \), and a general-purpose change propagation mechanism (\textit{propagate}). In sum, Adapton only evaluates \( f' \), and it reuses the other computations of \( f \), and the construction of the original output list cells. This result follows from the interplay of names, named allocations, the list-map algorithm, and the change propagation algorithm:

\begin{itemize}
  \item \textit{propagate} begins by re-evaluating the \textit{map} thunk that directly read the now-changed input cell \( b \).
  \item Recursively, \textit{map} on list tail \( c \) is unaffected by \( \Delta I_1 \), so \textit{propagate} reuses the result in cell \( c' \).
  \item Next, \textit{map} deterministically maps name \( b \) to name \( b' \) (as in the run on \( I_1 \)), and named reference cell allocation implicitly updates list position \( b' \) to hold list cell \( \text{Cons}(f 4, c') \) in place of list cell \( \text{Cons}(f 3, c') \).
  \item When this call involving cell \( b \) terminates, it produces the same resulting cell \( b' \) as in the prior run. Consequently, change propagation terminates, having only performed one recursive invocation of \textit{map}, to accommodate the one changed value at position \( b \).
\end{itemize}

This efficiency stems directly from using the names in sets \( X \) and \( X' \) to deterministically relate the input and output (horizontal correlations), over successive executions (vertical correlations). For instance, if we instead chose names through a global counter, or a non-deterministic choice, we would sabotage the behavior above, since either naming strategy would fail to reallocate \( b' \), as in the prior run. For further details about the role of names in change propagation, we refer the reader to Hammer et al. (2015, Sec. 2).

Supporting the archivist’s role with Typed Adapton. In Typed Adapton, we refer to role of authoring program \( e \) as the \textit{archivist} role. The archivist begins with a functional algorithm, and they employ dynamic names that identify and isolate sub-structures that change over time, as shown above for \texttt{list_map}. These dynamic name abstractions support general-purpose functional programs whose data structures contain first-class names, and where computations deterministically and precisely name their data structures (by naming references) and sub-computations (by naming thunks).

In the next section, we apply Typed Adapton to the \texttt{list_map} example above, showing how to encode the high-level notion of named list positions with these abstractions. Though \texttt{list_map} uses names correctly, programs may still use it in larger compositions that contain name ambiguities. For instance, consider these three, two-line programs over list input \( \text{inp} \):

\begin{verbatim}
let xs = map f inp | let xs = map s1 f inp | let xs = scope s1 [ map f inp ]
let ys = map g inp | let ys = map s2 g inp | let ys = scope s2 [ map g inp ]
\end{verbatim}


In each of the three compositions, the archivist consumes the names in input list `inp` twice, producing two different output lists `xs` and `ys`. However, the first composition is ambiguous, since it lacks a mechanism to distinguish the dynamic scope of these output lists' named list positions. Consequently, output list `xs` occupies the same named list positions as output list `ys`, conceptually overwriting `xs` with `ys` in the dependency graph; this ambiguity signals a fatal dynamic error in past work on Adapton (Hammer et al. 2015).

To resolve this ambiguity in Typed Adapton, the archivist distinguishes the two uses of names in `inp` with two different name functions `s1` and `s2`. In the second version (middle listing), suppose that `listmap` accepts this name function as an additional argument, applying it to each name in `inp` that it writes. For instance, suppose that `s1 := λa. ⟨⟨m_1, a⟩⟩` and `s2 := λa. ⟨⟨m_2, a⟩⟩`, for two name constants `m_1 ⊥ m_2`. In Typed Adapton, names constants consist of binary trees (`n := leaf | ⟨⟨n, n⟩⟩`), which can encode arbitrary symbolic constants (such as numbers, strings and lists of such symbols). Concretely, we could choose name constants `m_1 := leaf` and `m_2 := ⟨leaf, leaf⟩`, which we’ll use often, and write as 1 and 2, for short. Since these names are distinct (1 ≠ 2), the scopes `s1` and `s2` are distinct (written `s1 ⊥ s2`): when applied to equal name arguments, they will always produce distinct names as results. Hence, the two invocations of `listmap` compose without overwriting one another: each name from `inp` is used twice, but each time it is composed with a distinct name, either `m_1` or `m_2`.

Since this dynamic scoping pattern is extremely common, Typed Adapton provides an in-built monadic mechanism for allocation effects. This mechanism implicitly passes these name functions from calling contexts to the allocation sites in `listmap`. The third composition illustrates this pattern, using the Typed Adapton keyword `scope` to extend the current scope, composing the given name function with the current scope (another name function). Dynamic scopes, and their presence in the effect system of Typed Adapton, solves a key program composition problem, as shown above.

Polymorphic types pose another composition problem for the archivist. For instance, when a list holds another inductive structure, such as nested sub-lists, the archivist’s algorithms require further invariants and type-level accommodations. As an example, when it combines a list of sub-lists, the archivist may want `list_join` to combine all the names of the outer and inner lists. To justify this, we must reason statically that the named positions in each sub-list are distinct from the other sub-lists, and from the positions in the outer list-of-lists. To handle this case, and others, Sec. 2 demonstrates a recipe for generic polymorphic lists that can express these complex structural invariants, as well as simpler lists of primitive elements that lack name-based structure (e.g., integers). This recipe generalizes to other inductive and coinductive structures, such as balanced binary trees, and lazy streams, etc.

**Contributions.**

- We present Typed Adapton, the first type system to statically approximate precise allocation names in incremental programs.
- As evidence of its generality, we tour a series of example programs over lists, illustrating general programming recipes for program composition and type composition in the presence of names (Sec. 2).
- To refine the types, we employ separate name and index term languages, for statically modeling names and name sets, respectively (Sec. 4). The name-set indices were inspired by set-indexed types in the style of DML (Xi and Pfenning 1999); we extend this notion with polymorphism over kinds and index functions; the latter was inspired by abstract refinement types (Vazou et al. 2013).
- Drawing closer to an implementation, we give a bidirectional version of the type system that we show is sound and complete with respect to our main type assignment system (Sec. B).
- In particular, we give a syntactic proof theory of name term and index term equivalence and apartness. We define apartness, written `a ⊥ b`, as the property of two names being distinct, of two name sets being disjoint, or two name functions producing apart results given equal inputs. Apartness-based reasoning plays a critical role in Typed Adapton, but may also be of independent interest.
We begin with a type for incremental lists of integers. We use this list type to write list_map, which adheres to a linear typing discipline. We still wish to name these computation patterns and data structures.

\[ X \]

The names in \( X \) impose the constraint that each name in the remainder of the list. For instance, when quantifying over these sets, we write \( X \) and \( Y \) and the pointer names \( Y \) drawn from set \( \text{NmSet} \). The type below defines incremental lists of integers, \( \text{List}[X,Y] \text{Int} \). This type has two conventional constructors, \( \text{Nil} \) and \( \text{Cons} \), as well as two additional constructors that use the two type indices \( X \) and \( Y \). Each index has \text{name set sort} \( \text{NmSet} \), which classifies indices that are sets of names.

The \text{Name} constructor permits names from set \( X_1 \) to appear in the list sequence; it creates a \text{Cons}-cell-like pair holding a first-class name from \( X_1 \) (of type \( \text{Nm}[X_1] \)), and the rest of the sequence, whose names are from \( X_2 \). The \text{Ref} constructor permits injecting reference cells holding lists into the list type; these pointers’ names are drawn from set \( Y = [Y_1 \uplus Y_2] \), the disjoint union of the possible pointer names \( Y_1 \) for the head reference cell, and the pointer names \( Y_2 \) contained in this cell’s list.

\[
\begin{align*}
\text{Nil} & : \forall X,Y : \text{NmSet}. \quad \text{unit} \quad \rightarrow \quad \text{List}[X,Y] \text{Int} \\
\text{Cons} & : \forall X,Y : \text{NmSet}. \quad \text{Int} \rightarrow \quad \text{List}[X,Y] \text{Int} \quad \rightarrow \quad \text{List}[X,Y] \text{Int} \\
\text{Name} & : \forall X_1 \uplus X_2, Y : \text{NmSet}. \quad \text{Nm}[X_1] \rightarrow \quad \text{List}[X_2,Y] \text{Int} \quad \rightarrow \quad \text{List}[X_1 \uplus X_2,Y] \text{Int} \\
\text{Ref} & : \forall X,Y_1 \uplus Y_2 : \text{NmSet}. \quad \text{Ref}[Y_1] \text{List}[X,Y_2] \text{Int} \quad \rightarrow \quad \text{List}[X;Y_1 \uplus Y_2] \text{Int}
\end{align*}
\]

For both of these latter forms, the constructor’s types enforce that each name (or pointer name) in the list is disjoint from those in the remainder of the list. For instance, when quantifying over these sets, we write \( X_1 \uplus X_2 \) to impose the constraint that \( X_1 \) be disjoint from \( X_2 \), and similarly for \( Y_1 \uplus Y_2 \); However, for the \text{Name} constructor, the names in \( X_1 \) and pointer names in \( Y \) may overlap (or even coincide). These disjointness constraints consist of syntactic sugar that we expand in Sec. 4. The \text{Nil} constructor creates an empty sequence with any pointer
or name type sets in its resulting type, since, for practical reasons, we find it helpful to permit type indices to over-approximate name sets.

In Sec. 1, we described incrementally-changing lists using the notion of a named list position. Using the type listed above, we can decompose this high-level notion into two distinct low-level notions: unallocated names, which are unassociated with content of any type, and which need only be locally unique (e.g., unique to a list or other data structure), and allocated names, which dynamically name content of some fixed type and must be globally unique; they represent store pointers.

In the Name(n, t) form, the name n is unallocated, and can be used to allocate a reference or thunk of any type. In the Ref(r) form, by contrast, we have a reference cell r that consists of a pointer name whose type is parameterized by the type of the content it names, in this case an integer list. To expand the named list positions from Sec. 1 into these two constructors, imagine that for each such tail position named n, we introduce an unallocated Name n, followed by an allocated Ref named s(n), for some name-function s. The reference cell s(n) holds the rest of the list at position n. The scope s distinguishes dynamic scopes, as introduced in Sec. 1, and explained in further detail below; intuitively, name function s translates a locally-unique name (unique to the list) into a globally-unique name (unique to global program evaluation).

2.2 Mapping a list of integers, with names

Fig. 1 (left listing) lists the type, effect and code for list_map0, which consumes the list type defined above, but does not allocate any references or thunks. We include it here for illustrative purposes, as preparation for further examples. The type signature includes abstract name sets X and Y, for the names and named pointers of the input list l. The resulting list type is indexed by the name set X and the empty pointer set ∅ since the output contains the same names as the input (all drawn from X), but does contain any pointers. The function’s effect, written as ▷(∅; Y), indicates that the code writes no names, but reads the pointers Y from the input list l.

Turning to the program text for list_map0, the Nil and Cons cases are entirely conventional: They return Nil and apply f to the head of each Cons cell, respectively. The Name and Ref cases handle unallocated names and (allocated) pointers in the list; both are simple recursive cases. In the Name case, list_map0 maps the input name into corresponding position of the output list and recurs. In the Ref case, list_map0 uses get to observe the content of the reference cell holding the remainder of the input, and recurs on this input list.

The right listing gives list_map1, whose code and behavior is identical to list_map0 on the left, except for in the Name case, where the right version allocates a reference cell to hold the output list, and it allocates (and forces) a thunk to memoize the recursive call. (The shorthand memo(e1, e2) expands into force(thunk(e1, e2))). To name these allocations, list_map1 uses the input list’s name n for the reference cell and a distinct name for the thunk, (n-1) := (n, leaf) ≠ n. Critically, the name n-1 is distinct from both n and any name that is distinct from n, including the other names in the list. That is, for all m, we have that (n ⊥ m) ⇒ (n ⊥ n-1 ⊥ m ⊥ m-1), meaning that the four elements in the consequent are pairwise distinct. Generalizing further, we can use this pattern to systematically add memoization and reference cells to any structurally-recursive algorithm that includes an analog of this Name case. We show examples of such algorithms, below.

Comparing its type with list_map0, the returned list type of list_map1 now contains names from X (as before) and pointers from ∗(X). The effect is also affected: the written name set is now ∗(X ⊥ X-1), not ∅. These sets refer to the names written in the revised Name case, mapped by the current monadic scope, which we denote with the notation ∗(−); recall that the scope is a dynamically-specified name-transforming function that we apply to each written name, in this case drawn from X ⊥ X-1. Further, we use shorthand X-1 to informally mean “the names of set X, mapped by the function λa. ⟨⟨a, leaf⟩⟩”; in Sec. 4, we make the syntax for name sets, type indices and effects precise.
list_map0: ∀X,Y:NmSet. 
[Int → Int] → (List [X,Y] Int) → (List [X,Y] Int)

> (0,Y)

list_map0 = λf. fix rec. λl. 
m.match l with

| Nil    ⇒ Nil       | Nil    ⇒ Nil       |
| Cons(h,t) ⇒ Cons(f h, rec t) | Cons(h,t) ⇒ Cons(f h, rec t) |
| Name(n,t) ⇒ Name(n, rec t) | Name(n,t) ⇒ Name(n, Ref(Ref(n, memo(n-1, rec t)))) |
| Ref(r) ⇒ rec (get r)   | Ref(r) ⇒ rec (get r)   |

Fig. 1. list_map0 (left) and list_map1 (right) consist of the standard algorithm to map a list of integers l using a given integer function f, augmented with additional cases for Ref and Name. The left version performs no allocations; the right version uses names from the input list to name reference cells and recursive calls.

Fig. 2. **Left:** The dependence graph produced by evaluating function list_map1 on inp, a two-integer, two-name input list, shown in the two upper blue regions of the figure. Its names are drawn from set X = [a, b, ...], and its pointer names are drawn from the set Y = [p, q, ...]. The two purple sections consist of allocated dependence graph nodes that depend on the input list: cell a, thunk a-1, cell b, and thunk b-1. **Right:** The dependence graph produced by evaluating map_pair on the same input list, in two different scopes: 1 and 2. The scopes give rise to precise names for the output lists and their associated computations.

2.3 Names identify nodes in a dynamic dependency graph

Suppose that we execute list_map1 on an input list with two integers, two names a and b, and two reference cells p and q, as shown in the left of Fig. 1. Bold boxes denote reference cells in the input and output lists, and bold circles denote memoized thunks that compute with them. Each bold object (thunk or reference) is named precisely. The two thunks, shown as bold circles named by a-1 and b-1, follow a similar pattern: each has outgoing edges to the reference that they dereference (northeast, to p and q, respectively), the recursive thunk that they force, if any (due west, to b-1 for a-1, and none for b-1), and the reference that they name and allocate (southeast, to a and b, respectively).

As illustrated in the example from Sec. 1, the benefit of using names is that imperative O(1) changes to the input structure can be reflected into the output with only O(1) changes to its named content. Further, for the memoized thunks, the names also provide a primitive form of cache eviction: Memoized results are overwritten when names are re-associated with new content.
2.4 Scopes distinguish distinct sub-computations

As introduced in Sec. 1, Adapton programs commonly disambiguate two instances of the same written name by using two different scopes. Specifically, the scope monad implicitly threads a name function through the computation, applying it to each name that the computation allocates. Conceptually, the scope at the outermost layer of the computation is the identity name function. As the computation enters subtrees of the dynamic call tree, the scope monad nests scope by composing name functions.

```plaintext
let map_pair l =
  let xs = scope s1 [ list_map1 f l ]
  let ys = scope s2 [ list_map1 g l ]
  (xs, ys)
let out = memo(1, map_pair inp)
```

For instance, we adapt the code on the left for map_pair from Sec. 1. It uses disjoint scopes s1 and s2 to map an input list inp twice within a single incremental program to produce a pair of distinct output lists out that each depend on the common input list inp. To make the example concrete, as in Sec. 1, suppose that s1 and s2 preprend the name constants 1 and 2 on to their name argument, respectively. This program gives rise to the dependence graph shown in Fig. 1 (right side). This diagram illustrates how the two sub-computations of list_map1 share the input list inp: Each of its reference cells has two incoming get edges. (Recall, names are not linear resources, so this is not problematic). To distinguish these scopes, the distinct name functions 〈1, −〉 and 〈2, −〉 disambiguate the name of each reference cell and thunk for list_map1 (e.g., compare the allocated names of the left-hand and right-hand versions of the figure).

2.5 Polymorphic lists, with names

Above, we considered lists of integers. Now, suppose the archivist wishes to employ lists of other data structures, such as lists of lists. For instance, past literature commonly implements incremental mergesort using functions over lists of lists. First, consider what we call naive type polymorphism, where we replace each instance of "Int" in type of integer lists \(\text{List}[X; Y] \text{Int}\) with the universally-quantified type variable \(\alpha\):

\[
\begin{align*}
\text{Nil} & : \forall \alpha: \text{type. } \forall X, Y: \text{NmSet. } 1 & \rightarrow & \text{List}[X; Y] \alpha \\
\text{Cons} & : \forall \alpha: \text{type. } \forall X, Y: \text{NmSet. } \alpha & \rightarrow & \text{List}[X; Y] \alpha \\
\text{Name} & : \forall \alpha: \text{type. } \forall X_1 \downarrow X_2, Y: \text{NmSet. } \text{Nm}[X_1] & \rightarrow & \text{List}[X_2; Y] \alpha \\
\text{Ref} & : \forall \alpha: \text{type. } \forall X, Y_1 \downarrow Y_2: \text{NmSet. } \text{Ref}[Y_1] \left( \text{List}[X; Y_2] \alpha \right) & \rightarrow & \text{List}[X; Y_1 \downarrow Y_2] \alpha
\end{align*}
\]

As we demonstrate, naively parameterizing lists by \(\alpha\) does not afford enough invariants to verify all the programs that we wish to author. For instance, suppose the archivist wishes to express the list transformation list_join that appends the sub-lists of a list of lists; in the Cons case, it uses auxiliary function list_append to append the sublist at the head with the remainder of the sublists’ elements in the (flattened) tail.

Fig. 3 lists the code for these standard list algorithms, as well as types for each. As with list_map0, we initially consider versions that do not allocate, for simplicity; later, we can augment the algorithms with named allocations in the same fashion that we systematically transformed list_map0 into list_map1. The code for list_append0 checks against the listed type, with reasoning similar to that of list_map0; it does not involve lists of lists, only lists of integers. As we explain below, the code for list_join0 does not check against the listed type: The listed type uses polymorphism naively with respect to name sets. To overcome this problem, we need to change the refinement type for Cons. To see why, first consider checking the body against the listed type.

Consider the Cons case of list_join0, whose code is standard; in the case of a sublist \(h\), we append this sublist to the result of flattening the rest of the input list of lists. Critically, this step involves reasoning about the relationship between three structures, each involving a set of names:

- The head \(h\) of the list of lists \(l\), with names drawn from \(X_2\) (Inner list \(h\) has type \(\text{List}[X_2; Y_2] \text{Int}\)).
- The tail \(t\) of the list of lists \(l\), with outer list names drawn from \(X_1\) and inner list names from \(X_2\), and...
The flattened tail \( t' \) of of tail \( t \), with names drawn from \( X_1 \perp X_2 \).

In particular, to prove that the use of \( \text{list_append} \) \( h \ t' \) is well-typed, we wish to argue that head \( h \) (drawn from \( X_2 \)) has names that are distinct from those in the flattened list \( t' \) (drawn from \( X_1 \perp X_2 \)). However, the naive polymorphic list type places no constraints on names in different sub-lists: It merely says that their names are all drawn from \( X_2 \), not that each distinct sub-list uses a distinct, non-overlapping subset of \( X_2 \).

Recall the type given above for \( \text{Cons} : \forall \alpha : \text{type}. \forall X, Y : \text{NmSet.} \alpha \rightarrow (\text{List} [X; Y] \alpha) \rightarrow (\text{List} [X; Y] \alpha) \). In particular, we want the type for \( \text{Cons} \) to enforce a relationship of disjointness between the name sets of the new element and the existing elements in the list, which is does not. Further, as written, each occurrence of \( \alpha \) must be the same, and thus, must over-approximate name sets. Instead, the following types for \( \text{Cons} \) will permit \( \text{list_join} \) to type-check; we discuss them in turn:

Lists of lists of integers:
\[
\begin{align*}
\text{Cons} : & \forall X_1 \perp X_2 : \text{NmSet.} \\
& \forall Y \perp Y_1 \perp Y_2 : \text{NmSet.}
\end{align*}
\]

Lists of \( \alpha \) (type \( \alpha \) has kind \( \gamma \rightarrow \text{type} \)):
\[
\begin{align*}
\text{Cons} : & \forall X, Y : \text{NmSet}. \forall i, j : \gamma. \\
& \forall k_1 \equiv f i : \gamma. \\
& \forall k_2 \equiv g X Y k_1 : \text{NmSet} \times \text{NmSet.}
\end{align*}
\]

First consider the type on the left, which is specific to lists of lists of integers. In particular, this type enforces that the sublists’ names are disjoint from one another, and from the names of the outer list. Consequently, in the \( \text{Cons} \) case of \( \text{list_join} \), we can justify the recursive call, since the outer and inner lists have distinct names; and, we can justify the call to \( \text{list_append} \), since the names of each inner list are distinct from those of other inner lists.

Instead of writing a new version of \( \text{Cons} \) for every list element type, we want to capture this pattern, and others, generically. The second (rightmost) type of \( \text{Cons} \) is generic in type \( \alpha \), but unlike the naive polymorphic type, this version permits the indices of the type parameter \( \alpha \) to vary across occurrences (\( i \) versus \( j \)), while enforcing user-defined constraints over these index occurrences (index function \( f \)), and between these element indices and the indices of the list (function \( g \)). In particular, the client of this generic list chooses a type \( \alpha \),
with some index sort \( \gamma \), and they choose index functions \( f \) and \( g \):

\[
\begin{align*}
  f & : \gamma \rightarrow_{\text{idx}} \gamma \\
  g & : \text{NmSet} \rightarrow_{\text{idx}} \text{NmSet} \rightarrow_{\text{idx}} \gamma \rightarrow_{\text{idx}} (\text{NmSet} \times \text{NmSet})
\end{align*}
\]

As described by its sort, index function \( f \) forms an element index from two elements’ indices (each of sort \( \gamma \)). Likewise, index function \( g \) forms a list index (a pair of name sets) from a list index (two, curried name sets) and an element index. By choosing type \( \alpha \), sort \( \gamma \) and index functions \( f \) and \( g \), we can recover integer lists, and lists of integer lists as special cases:

*Integer lists:*

\[
\begin{align*}
  \alpha & : \equiv \text{Int} \\
  \gamma & : \equiv \text{1} \\
  f & : \equiv \lambda().\lambda().() \\
  g & : \equiv \lambda\lambda X.\lambda Y.\lambda().(X,Y)
\end{align*}
\]

*Lists of integer lists:*

\[
\begin{align*}
  \alpha & : \equiv \text{List}[\text{1};\text{1}]\text{Int} \\
  \gamma & : \equiv \text{NmSet}\times\text{NmSet} \\
  f & : \equiv \lambda(X_1,Y_1).\lambda(X_2,Y_2).\text{(}X_1 \perp X_2, Y_1 \perp Y_2\text{)} \\
  g & : \equiv \lambda\lambda X_1.\lambda Y_1.\lambda(X_2,Y_2).\text{(}X_1 \perp X_2, Y_1 \perp Y_2\text{)}
\end{align*}
\]

On the left, we express list of integers by choosing \( \alpha : \equiv \text{Int} \) and \( \gamma : \equiv \text{1} \), the sort of \text{Int}’s (trivial) type index. On the right, we express lists of integer lists by choosing \( \alpha : \equiv \text{List}[\text{1};\text{1}]\text{Int} \), the type of integer lists, and define \( f \) to accumulate name sets for the inner lists, and \( g \) to accumulate name sets for both the inner and outer lists. In this case, the resulting functions \( f \) and \( g \) happen to be isomorphic.

3 PROGRAM SYNTAX

The examples from the prior section use an informally-defined variant of ML, enriched with 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.

Values

\[
v ::= x | 0 | (v_1, v_2) | \text{inj}_1 v | \text{name} n | \text{nref} M | \text{ref} n | \text{thunk} n
\]

Terminal exprs.

\[
t ::= \text{ret}(v) | \lambda x. e
\]

Expressions

\[
e ::= t | \text{split}(v, x_1, x_2, e) | \text{case}(v, x_1, e_1, x_2, e_2) \\
| e v | \text{let}(e_1, x, e_2) | \text{thunk}(v, e) | \text{force}(v) | \text{ref}(v, v) | \text{get}(v) \\
| \text{scope}(v, e) | v_M v
\]

Fig. 4. Syntax of expressions

3.1 Values and Expressions

Fig. 4 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 \text{ret}(v)—the expression that returns the value \( v \)—or a \( \lambda \). Expressions \( e \) include terminal expressions, elimination forms for pairs, sums, and functions (\text{split}, \text{case} and \( e v \), respectively); let-binding (which evaluates \( e_1 \) to \text{ret}(v) and substitutes \( v \) for \( x \) in
Names (binary trees) \( m, n \ ::= \text{leaf} \)

leaf name

| \( \langle \langle n_1, n_2 \rangle \rangle \) |

binary name composition

Name terms \( M, N \ ::= \langle \rangle | (M_1, M_2) \)

unit and tuple of name terms

| \( n | \langle \langle M_1, M_2 \rangle \rangle \) |

literal names and binary name composition

| \( a | \lambda a. M | M(N) \) |

variable, abstraction, application

Name term values \( V ::= n | \lambda a. M | a | \langle \rangle | (V, V) \)

Name term sorts \( \gamma ::= Nm \)

name; inhabitants \( n \)

| \( \langle \rangle \) |

unit index sort; inhabitant \( () \)

| \( \gamma * \gamma \) |

product index sort; inhabitants \( (M_1, M_2) \)

| \( \gamma Nm \) |

name term function; inhabitants \( \lambda a. M \)

Typing contexts \( \Gamma ::= \cdot | \Gamma, a : \gamma | \cdots \)

full definition in Figure 9

Fig. 5. Syntax of name terms: a \( \lambda \)-calculus over names, as binary trees

\[
\begin{align*}
\Gamma \vdash M : \gamma & \quad \text{Under } \Gamma \text{, name term } M \text{ has sort } \gamma \\
\Gamma \vdash \langle \rangle : 1 & \quad M\text{-unit} \\
\Gamma \vdash n : Nm & \quad M\text{-const} \\
\Gamma \vdash a : \gamma & \quad M\text{-var} \\
\Gamma \vdash (\lambda a. M) : (\gamma Nm \gamma) & \quad M\text{-abs} \\
\Gamma \vdash (\langle \langle M_1, M_2 \rangle \rangle) : Nm & \quad M\text{-bin}
\end{align*}
\]

Fig. 6. Sorting rules for name terms \( M \)

\( e_2 \); introduction (thunk) and elimination (force) forms for thunks; and introduction (ref) and elimination (get) forms for pointers (reference cells that hold values).

The special forms of values are names \( n \), name-level functions \( nmfn \), references (pointers), and thunks. References and thunks include a name \( n \), which is the name of the reference or thunk, not the contents of the reference or thunk.

The syntax described above follows that of prior work on Adapton, including Hammer et al. (2015). We add the notion of a name function, which captures the idea of a namespace and other simple transformations on names. The construct 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 5 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 \( \text{leaf} \) or a branch node \( \langle \langle n_1, n_2 \rangle \rangle \). Name terms \( M, N \) consist of the unit name \( \langle \rangle \), a pair of names \( \langle M_1, M_2 \rangle \), 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 $\gamma$: sort $\mathsf{Nm}$ for names $n$, unit $1$ for (), product $\gamma \times \gamma$ for pairs $(M_1, M_2)$ and $\gamma \equiv \gamma$ for (name term) functions. Note that $\langle \langle n_1, n_2 \rangle \rangle$ is a name—a tree with subtrees $n_1$ and $n_2$—and has sort $\mathsf{Nm}$, while $(n_1, n_2)$ is a pair of names and has sort $\mathsf{Nm} \times \mathsf{Nm}$.

The rules for name sorting $\Gamma \vdash M : \gamma$ are straightforward (Figure 6), as are the rules for name term evaluation $M \Downarrow_M V$ (Figure 7). We write $M \equiv_\beta M'$ when name terms $M$ and $M'$ are convertible, that is, applying any series of $\beta$-reductions and/or $\beta$-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 $\text{Int}[k]$. Here, the ordinary type $\text{Int}$ and the index domain of integers are in one-to-one correspondence; the type $\text{Int}[3]$ 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 $\mathsf{B}$ whose name is in the set $f(X)$". A DML-style system can express such constraints, but the types become verbose:

$$\forall a : \mathsf{Nm}. \ \forall X : \mathsf{NmSet}. (a \in X) \supset (A[a] \rightarrow B[f(a)])$$

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

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

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

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

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

Figure 8 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 $\text{NmSet}$.

4.1.1 Name sets. If we give a name to each element of a list, then the entire list should carry the set of those names. We write $\{N\}$ for the singleton name set, $\emptyset$ for the empty name set, and $X \perp Y$ for a union of two sets $X$ and $Y$ that requires $X$ and $Y$ to be disjoint; this is inspired by the separating conjunction of separation logic (Reynolds 2002). While disjoint union dominates the types that we believe programmers need, our effects discipline requires non-disjoint union, so we include it ($X \cup Y$) as well.

4.1.2 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 $\text{prj}_b i$ 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 $\{M\}$). However, name terms and indices overlap—for example, $\lambda a. ()$ is both a name term function and an index-level function. On the other hand, $\lambda a. \text{prj}_2 a$ is an index-level function but not a name term function, because projection is not included in the syntax of name terms (Figure 5).

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 $i$.

4.1.3 Sorts. We use the meta-variable $\gamma$ to classify indices as well as name terms. The sort $\text{NmSet}$ (Figure 8) classifies indices that are sets of names; we inherit the unit sort, product sort, and function space $\Rightarrow_{\text{idx}}$ from the name term sorts (Figure 5).

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

$$\forall Y : \text{NmSet}. \quad \forall X : \text{NmSet}. \quad (Y \subseteq X) \supset (A[Y] \rightarrow B[f(Y)])$$

can be expressed by $\forall X : \text{NmSet}. A[X] \rightarrow B[f[X]]$.

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.
Most of the sorting rules in Figure 10 are straightforward, but rule ‘sort-sep-union’ includes a premise \( \text{extract}(\Gamma) \vdash X \perp Y : \text{NmSet} \), which says that \( X \) and \( Y \) are apart (disjoint).

4.1.4 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, \( \{n_1\} \perp \{n_2\} : \text{NmSet} \) implies that \( n_1 \neq n_2 \). Propositions are introduced into \( \Gamma \) via index polymorphism \( \forall \alpha : \gamma \mid P. E \), discussed below.

The function \( \text{extract}(\Gamma) \) (Figure 21 in the appendix) looks for propositions \( P \), which become equivalence and apartness assumptions. It also translates \( \Gamma \) into the relational context used in the definition of apartness. We give semantic definitions of equivalence and apartness in the appendix (Definitions E.4 and E.5).
4.2 Kinds

We use a simple system of kinds \( K \) (Figure 9). Kind type classifies value types, such as unit and \( \{ \text{Thk}[i] \ E \} \).

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 two name sets and the type of the list elements, e.g. \( \text{List}[X; Y] \text{Int} \). Therefore, List has kind \( \text{NmSet} \Rightarrow (\text{NmSet} \Rightarrow (\text{type} \Rightarrow \text{type})) \).

4.3 Effects

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

Effect sequencing (Figure 13) is a (meta-level) partial function over a pair of effects: the judgment \( \Gamma \vdash \epsilon \) then \( \epsilon_1 = \epsilon \), means that \( \epsilon \) describes the combination of having effects \( \epsilon_1 \) followed by effects \( \epsilon_2 \). Sequencing is a partial function because the effects are only valid when (1) the writes of \( \epsilon_1 \) are disjoint from the writes of \( \epsilon_2 \), and (2) the reads of \( \epsilon_1 \) are disjoint from the writes of \( \epsilon_2 \). Condition (1) holds when each cell or thunk is not written more than once (and therefore has a unique value). Condition (2) holds when each cell or thunk is written before it is read.

Effect coalescing, written "E after e", combines "clusters" of effects. For example:

\[
(C \triangleright (\langle n_2 \rangle; \emptyset)) \text{ after } (\langle n_1 \rangle; \emptyset) = C \triangleright (\langle n_1 \rangle; \emptyset) \text{ then } (\langle n_2 \rangle; \emptyset) = C \triangleright (\langle n_1, n_2 \rangle; \emptyset)
\]

Effect subsumption \( \epsilon_1 \leq \epsilon_2 \) holds when the write and read sets of \( \epsilon_1 \) are subsets of the respective sets of \( \epsilon_2 \).

4.4 Types

The value types (Figure 9), written \( A, B \), include standard sums \( + \) and products \( \times \); a unit type; the type \( \text{Ref}[i] \ A \) of references named \( i \) containing a value of type \( A \); the type \( \text{Thk}[i] \ E \) of thunks named \( i \) whose contents have type \( E \) (see below); the application \( A[i] \) of a type to an index; the application \( A . B \) of a type \( A \) (e.g. a type
\[ \Gamma \vdash A : K \] Under \( \Gamma \), value type \( A \) has kind \( K \)

\[ (\alpha : \text{type}) \in \Gamma \quad \frac{\Gamma \vdash \alpha : \text{type}}{\Gamma \vdash \alpha : \text{type}} \] \( \text{k-typevar} \)

\[ (d : K) \in \Gamma \quad \frac{\Gamma \vdash d : K}{\Gamma \vdash d : K} \] \( \text{k-tycon} \)

\[ \Gamma \vdash A_1 : \text{type} \quad \Gamma \vdash A_2 : \text{type} \quad \frac{\Gamma \vdash (A_1 \times A_2) : \text{type}}{\Gamma \vdash (A_1 + A_2) : \text{type}} \] \( \text{k-binop} \)

\[ \Gamma \vdash \text{unit} : \text{type} \quad \frac{\Gamma \vdash i : \text{NmSet}}{\Gamma \vdash \text{k-unit}} \] \( \text{k-unit} \)

\[ \Gamma \vdash i : \text{NmSet} \quad \frac{\Gamma \vdash \text{unit} : \text{type}}{\Gamma \vdash (\text{Thk}[i]) : \text{type}} \] \( \text{k-thk} \)

\[ \Gamma \vdash i : \text{NmSet} \quad \frac{\Gamma \vdash \text{unit} : \text{type}}{\Gamma \vdash (\text{Ref}[i]) : \text{type}} \] \( \text{k-ref} \)

\[ \Gamma \vdash A : \text{(type} \Rightarrow K) \quad \Gamma \vdash A : (\gamma \Rightarrow K) \quad \frac{\Gamma \vdash A[i] : K}{\Gamma \vdash A[i] : K} \] \( \text{k-app} \)

\[ \Gamma \vdash \text{F} \quad \frac{\Gamma \vdash (A \Rightarrow B) : K}{\Gamma \vdash (A \Rightarrow B) : K} \] \( \text{k-app-type} \)

\[ \Gamma \vdash \text{P} \quad \frac{\Gamma \vdash \text{P} \text{[1] and P}[2])}{\Gamma \vdash \text{P} \text{[1] and P} [2]) \} \] \( \text{k-app-index} \)

Fig. 11. Kinding and well-formedness for types and effects

\[ \Gamma \vdash \text{v} : A \] Under assumptions \( \Gamma \), value \( v \) has type \( A \)

\[ \Gamma \vdash () : \text{unit} \quad \frac{\Gamma \vdash (x : A) \in \Gamma}{\Gamma \vdash x : A} \] \( \text{var} \)

\[ \Gamma \vdash v_1 : A_1 \quad \Gamma \vdash v_2 : A_2 \quad \frac{\Gamma \vdash (v_1, v_2) : (A_1 \times A_2)}{\Gamma \vdash (\text{pair}) : (A_1 \times A_2)} \] \( \text{pair} \)

\[ \Gamma \vdash n \in X \quad \frac{\Gamma \vdash \text{name} n : \text{Nm}[X]}{\Gamma \vdash \text{name} n : \text{Nm}[X]} \] \( \text{name} \)

\[ \Gamma \vdash M_v : \text{Nm} \quad \frac{\Gamma \vdash \text{namefn} M_v : [(\text{Nm} \Rightarrow \text{Nm})][M]}{\Gamma \vdash \text{namefn} M_v : [(\text{Nm} \Rightarrow \text{Nm})][M]} \] \( \text{namefn} \)

\[ \Gamma \vdash (\text{ref} n) : (\text{Ref}[X] \ A) \quad \frac{\Gamma \vdash (\text{thunk} n) : (\text{Thk}[X] \ E)}{\Gamma \vdash (\text{thunk} n) : (\text{Thk}[X] \ E)} \] \( \text{ref} \)

Fig. 12. Value typing

constructor \( d \) to a type \( B \); the type \( \text{Nm}[i] \); and a singleton type \( (\text{Nm} \Rightarrow \text{Nm})[M] \) where \( M \) is a function on names.

As usual in call-by-push-value, computation types \( C \) and \( D \) include a connective \( \text{F} \), which "lifts" value types to computation types: \( \text{F} \ A \) is the type of computations that, when run, return a value of type \( A \). (Call-by-push-value usually has a connective dual to \( \text{F} \), written \( \text{U} \), that "thUnks" a computation type into a value type; in our system, \( \text{Thk} \) plays the role of \( \text{U} \).)
\[
\begin{align*}
\Gamma \vdash (e_1 \text{ then } e_2) &= e & \text{Effect sequencing} \\
\Gamma \vdash W_1 \perp W_2 & \quad \Gamma \vdash R_1 \perp R_2 \\
\Gamma \vdash \langle W_1; R_1 \rangle \text{ then } \langle W_2; R_2 \rangle = \langle W_1 \cup W_2; R_1 \cup R_2 \rangle \\
\Gamma \vdash (E \text{ after } e) &= E' & \text{Effect coalescing} \\
\Gamma \vdash (e_1 \text{ then } e_2) &= e & \Gamma \vdash (E \text{ after } e) = E' \quad \Gamma \vdash (\forall \alpha : K. E) \text{ after } e = (\forall \alpha : K. E') \\
\Gamma \vdash (\forall \alpha : \gamma | P. E) \text{ after } e = (\forall \alpha : \gamma | P. E')
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash M : A & \\
\Gamma \vdash e : E & \text{ Under } \Gamma, \text{ within namespace } M, \text{ computation } e \text{ has type-with-effects } E \\
\Gamma \vdash v : (A_1 \times A_2) & \\
\Gamma, x_1 : A_1, x_2 : A_2 \vdash M \vdash e : E & \\
\Gamma \vdash \text{split}(v, x_1, x_2 : e) : E & \text{ split} \\
\Gamma \vdash v : A & \\
\Gamma \vdash \text{ret}(v) : (\langle A \rangle \triangleright \langle \emptyset \rangle) & \text{ ret} \\
\Gamma, x : A \vdash M \vdash e : E & \\
\Gamma \vdash \langle \lambda x. e \rangle : (\langle A \rightarrow E \rangle \triangleright \langle \emptyset \rangle) & \text{ lam} \\
\Gamma \vdash v : (A_1 + A_2) & \\
\Gamma, x_1 : A_1 \vdash M \vdash e_1 : E & \\
\Gamma, x_2 : A_2 \vdash M \vdash e_2 : E & \text{ case} \\
\Gamma \vdash v : (A_1 + A_2) & \\
\Gamma, x_1 : A_1 \vdash M \vdash e_1 : E & \\
\Gamma, x_2 : A_2 \vdash M \vdash e_2 : E & \Gamma \vdash (e_1 \text{ then } e_2) = e & \text{ let} \\
\Gamma \vdash v : (A_1 + A_2) & \\
\Gamma, x_1 : A_1 \vdash M \vdash e_1 : E & \\
\Gamma, x_2 : A_2 \vdash M \vdash e_2 : E & \Gamma \vdash (\langle \emptyset \rangle; \langle \emptyset \rangle) \text{ then } e = e' & \text{ force} \\
\Gamma \vdash v : (F A) \triangleright \langle \emptyset \rangle & \\
\Gamma \vdash \text{thunk}(v, e) : (F \langle \text{Thk}[\langle M[X] \rangle E] \rangle \triangleright \langle M[X] \rangle; \emptyset) & \text{ thunk} \\
\Gamma \vdash v : (\langle \emptyset \rangle; \langle \emptyset \rangle) \text{ then } e = e' \\
\Gamma \vdash v : \text{Ref}[X] A & \\
\Gamma \vdash \text{ref}(v_1, v_2) : (F \langle \text{Ref}[\langle M[X] \rangle A] \rangle \triangleright \langle M[X] \rangle; \emptyset) & \text{ ref} \\
\Gamma \vdash v : \text{Nm}[X] & \\
\Gamma \vdash v_1 : \text{Nm}[X] & \\
\Gamma \vdash v_2 : A & \text{ ref} \\
\Gamma \vdash v : \text{Nm}[i] & \\
\Gamma \vdash v : \text{Nm}[\text{Nm}[i]] & \text{ name-app} \\
\Gamma, a : \gamma | P \vdash M \vdash t : E & \\
\Gamma \vdash \text{AllIndexIntro} \\
\Gamma, a : \gamma | P \vdash \langle \forall a : \gamma | P. E \rangle & \text{ AllIndexElim} \\
\Gamma \vdash \langle \forall a : \gamma | P. E \rangle & \Gamma \vdash \langle \exists \gamma | P \rangle \vdash [i/a]P & \Gamma \vdash i : \gamma & \Gamma \vdash \text{extract}(\Gamma) \vdash [i/a]P & \text{AllIndexElim} \\
\Gamma \vdash i : \gamma & \\
\Gamma \vdash \langle \forall a : \gamma | K. E \rangle & \Gamma \vdash a : K & \text{AllElim} \\
\Gamma, a : K \vdash M \vdash t : E & \\
\Gamma \vdash \text{AllIntro} \\
\Gamma, a : K \vdash M \vdash t : (\forall a : K. E) & \Gamma \vdash \langle \forall a : K. E \rangle & \Gamma \vdash a : K & \text{AllElim} \\
\Gamma \vdash \text{AllIntro} \\
\Gamma, a : K \vdash M \vdash t : (\forall a : K. E) & \\
\Gamma \vdash \langle \forall a : K. E \rangle & \Gamma \vdash \langle [a/a']E \rangle & \text{AllElim} \\
\Gamma \vdash \langle [a/a']E \rangle
\end{align*}
\]

Fig. 13. Computation typing
Computation types also include functions, written $A \rightarrow E$. In standard CBPV, this would be $A \rightarrow C$, not $A \rightarrow E$. We separate computation types alone, written $C$, from computation types with effects, written $E$; this decision is explained below.

Computation types-with-effects $E$ consist of $C \triangleright \epsilon$, which is the bare computation type $C$ with effects $\epsilon$, as well as universal quantifiers (polymorphism) over types $(\forall \alpha : K.E)$ and indices $(\forall \alpha : \gamma | P.E)$. In the latter quantifier, the proposition $P$ lets us express quantification over disjoint sets of names.

Value typing rules. The typing rules for values (Figure 12) for unit, variables and pairs are standard. Rule ‘name’ uses index-level entailment to check that the name $n$ is in the name set $V$. 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 $\Gamma$. Rule ‘thunk’ is similar to ‘ref’.

Computation typing rules. Many of the rules that assign computation types (Figure 13) are standard—for call-by-push-value—with the addition of effects and the namespace $M$. The rules ‘split’ and ‘case’ have nothing to do with namespaces or effects, so they pass $M$ up to their premises, and leave the type $E$ unchanged. Empty effects are added by rules ‘ret’ and ‘lam’, since both $\text{ret}$ and $\text{lam}$ do not read or write anything. The rule ‘let’ uses effect sequencing to combine the effects of $\epsilon_1$ 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 $\nu = \text{nmfn} N'$ in the second premise (typing $e$).

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

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

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

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 $E ::= C \triangleright \epsilon$, 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.
4.5 Bidirectional Version

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

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

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

Completeness (Theorems C.2, C.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 DYNAMICS

\[
\begin{align*}
\text{Pointers} & \quad p, q ::= n & \text{(Pointers are name constants)} \\
\text{Stores} & \quad S ::= & \text{empty store} \\
& & \mid S, p;v & \text{p points to value v} \\
& & \mid S, p: e @ M & \text{p points to thunk e, run in scope M}
\end{align*}
\]

\[S_1 \vdash_M^M e \Downarrow S_2; t\] Under store S in namespace M at current node m, expression e produces new store S_2 and result t

\[S_1 \vdash_m^M \text{term} \quad S \vdash_m^M t \Downarrow S; t\]

\[S_1 \vdash_m^M [v_1/x_1] e_1 \Downarrow S_2; e'\] split

\[S_1 \vdash_m^M \text{case}(inj_1 n, v_1, e_1, x_2, e_2) \Downarrow S_2; e'\] case

\[S_1 \vdash_m^M e_1 \Downarrow S_1'; \lambda x. e_2 \quad S_1 \vdash_m^M (v_1/x_1) e_2 \Downarrow S_2'; e_2'\] app

\[S_1 \vdash_m^M e_1 v \Downarrow S_1'; e_2'\]

\[\frac{M_1 \Downarrow_M^M \lambda M_2 [n/a] M_2 \Downarrow_M^M p}{S_1 \vdash_m^M \text{name-app}}\]

\[\frac{(M n) \Downarrow_M^M p \quad S_1 [p \rightarrow e @ M] = S_2}{S_1 \vdash_m^M \text{thunk}(name n, e) \Downarrow S_2; \text{ret}(\text{thunk p})}\]

\[\exp(S_1, p) = e \quad \text{scope}(S_1, p) = M_0 \quad S_1 \vdash_m^M e \Downarrow S_2; t\] force

\[S_1 \vdash_m^M \text{force}(\text{thunk p}) \Downarrow S_2; t\]

\[\frac{(M n) \Downarrow_M^M p \quad S_1 [p \rightarrow v] = S_2}{S_1 \vdash_m^M \text{ref}(\text{name n}, v) \Downarrow S_2; \text{ret}(\text{ref p})}\]

\[\frac{S(p) = v}{S \vdash M \text{get}(\text{ref p}) \Downarrow S; \text{ret}(v)}\]

Fig. 14. Dynamics, as big-step evaluation
\[
\begin{align*}
\Gamma \vdash v : A & \quad \Gamma(p) = A \\
\vdash (S, p : v) : \Gamma & \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e : E & \quad \Gamma(p) = E \\
\vdash (S, p : e) : \Gamma & \\
\end{align*}
\]

Fig. 15. Store typing: \( S \vdash \Gamma \), read "store \( S \) typed by \( \Gamma \)."

5.1 Dynamics of name terms

Recall Fig. 7 (Sec. 3.2), which gives the dynamics for evaluating a name term \( M \) into a name term value \( V \). Because name terms lack the ability to perform recursion and pattern-matching, one may expect that they always terminate. Indeed, we demonstrate this fact with a standard logical relation (Theorem F.5).

5.2 Dynamics of program expressions

Fig. 14 (top) defines stores, which consist of the mutable state that names dynamically identify. Fig. 14 also defines the big-step evaluation relation for expressions, relating an initial and final store, as well as the "current scope" and "current node" to a program and value. We define this dynamics, which closely mirrors prior work's dynamics, to show that well-typed evaluations always allocate precisely.

To make this theorem meaningful, the dynamics permits programs to overwrite prior allocations with later ones: if a name is used ambiguously, the evaluation will replace the old store content with the new store content. The rules ref and thunk either extend or overwrite the store, depending on whether the allocated pointer name is precise or ambiguous, respectively. We prove that, in fact, well-typed programs always extend (and never overwrite) the store in any single derivation.

While motivated by incremental computation, we are interested in precise effects here, not change propagation itself. Consequently, this semantics is simpler than the dynamics of prior work. First, the store never caches values from evaluation; i.e., it does not model function caching (aka, 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 ref, thunk, get and force would create dependency edge structure that we omit here. These edges relate the current node with the node being observed by creating edges in the graph, as illustrated in Fig. 1 (Sec. 1).

6 METATHEORY: TYPE SOUNDNESS AND PRECISE EFFECTS

In this section, we prove that our type system is sound with respect to evaluation. Further, we simultaneously show that the type system enforces precise effects, codified formally by Def. 6.1, below. In particular, our main theorem establishes that a well-typed, terminating program produces a terminal computation of the program's type, and that the actual dynamic effects are precise. Specifically, we show that the type system's static effects soundly approximate this dynamic behavior. Consequently, sequenced writes never overwrite one another.

In some places below, we constrain typing contexts to be store types, which contain store pointers but no free variables; hence, they only type closed values and programs:

**Definition 6.1 (Store type).** We say that \( \Gamma \) is a store typing, written \( \Gamma \) store-type, when each assumption in \( \Gamma \) has the reference-pointer-type form \( p : A \) or the thunk-pointer-type form \( p : E \).

6.1 Precise effects: Read and write sets

**Definition 6.1 (Precise effects).** The precise effect of an evaluation derived by \( D \), written \( D \) reads \( R \) writes \( W \), is defined in Figure 16.

\( ^{3} \) During change propagation, not modeled here, we begin with an existing store (and dependency graph) from a prior run, and even precise programs overwrite the store/graph, as illustrated in Sec. 1.

\( ^{3} \) During change propagation, not modeled here, we begin with an existing store (and dependency graph) from a prior run, and even precise programs overwrite the store/graph, as illustrated in Sec. 1.
\( \mathcal{D} \) by Eval-term() reads \( \emptyset \) writes \( \emptyset \)

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

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

\( \mathcal{D} \) by Eval-scope(\( \mathcal{D}_0 \)) reads \( R \) writes \( W \) if \( \mathcal{D}_0 \) reads \( R \) writes \( W \)

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

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

\( \mathcal{D} \) by Eval-get() reads \( \emptyset \) writes \( p \) where \( e = \text{get}(\text{ref } p) \)

\( \mathcal{D} \) by Eval-thunk() reads \( \emptyset \) writes \( p \) where \( e = \text{thunk}(\text{name } n, e_0) \) and \( p \equiv M n \)

\( \mathcal{D} \) by Eval-ref() reads \( \emptyset \) writes \( p \) where \( e = \text{ref}(\text{name } n, v) \) and \( p \equiv M n \)

\( \mathcal{D} \) by Eval-force() reads \( q, R' \) writes \( W' \) where \( e = \text{force}(\text{thunk } q) \)
and \( \mathcal{D}' \) reads \( R' \) writes \( W' \)
where \( \mathcal{D}' \) is the derivation that computed \( t \)

![Fig. 16. Read-and write-sets of a non-incremental evaluation derivation.](image)

This is a (partial) function over derivations. We call these effects “precise” since sibling sub-derivations must have disjoint write sets. We write “\( \mathcal{D} \) by \text{RuleName (Dlist)} \) reads \( R \) writes \( W \)” to mean that rule \text{RuleName} concludes \( \mathcal{D} \) and has subderivations \( \text{Dlist} \). For example, \( \mathcal{D} \) by Eval-scope(\( \mathcal{D}_0 \)) reads \( R \) writes \( W \) provided that \( \mathcal{D} \) reads \( R \) writes \( W \), where \( \mathcal{D}_0 \) derives the only premise of Eval-scope.

### 6.2 Lemmas

**Lemma 6.2 (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. \( \square \)

**Lemma 6.3 (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 E.1 (Weakening of semantic equivalence and apartness) (for example, in the case for the value typing rule ‘name’) and Lemma 6.2 (Index-level weakening) (for example, in the case for the computation typing rule ‘AllIndexElim’). \( \square \)

**Lemma 6.4 (Substitution).**

1. If \( \Gamma \vdash v : A \) and \( \Gamma, x : A \vdash e : C \) then \( \Gamma \vdash (\{v/x\}e) : C \).
2. If \( \Gamma \vdash v : A \) and \( \Gamma, x : A \vdash v' : B \) then \( \Gamma \vdash (\{v/x\}v') : B \).

**Proof.** By mutual induction on the derivation typing \( e \) (in part 1) or \( v' \) (in part 2). \( \square \)
LEMMA 6.5 (Canonical Forms). If $\Gamma$ store-type and $\Gamma \vdash v : A$, then
1. If $A = 1$ then $v = 0$.
2. If $A = (B_1 \times B_2)$ then $v = (v_1, v_2)$.
3. If $A = (B_1 + B_2)$ then $v = \text{inj}_i v$ where $i \in \{1, 2\}$.
4. If $A = (\text{Nm}[X] )$ then $v = \text{name } n$ where $\Gamma \vdash n \in X$.
5. If $A = (\text{Ref}[X] \ A)$ then $v = \text{ref } n$ where $\Gamma \vdash n \in X$.
6. If $A = (\text{Thk}[X] \ E)$ then $v = \text{thunk } n$ where $\Gamma \vdash n \in X$.
7. If $A = (\text{Nm} \equiv \text{Nm}) [M]$ then $v = \text{nmfn } M_v$ where $M =_\beta (\lambda a. M')$
and $\vdash (\lambda a. M') : (\text{Nm} \equiv \text{Nm})$ and $M_v =_\beta M$.

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

LEMMA 6.6 (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 $M(i)$.

6.3 Main Theorem

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

THEOREM 6.7 (Subject Reduction).

If $\Gamma_1$ store-type and $\Gamma_1 \vdash M : \text{Nm} \equiv \text{Nm}$ and $S : \Gamma_1 \vdash M : C$ and $\langle W; R \rangle$
and $\vdash S_1 : \Gamma_1$ and $\vdash S_2 : \Gamma_2$ and $\Gamma_2 \vdash t : C$ and $\langle \emptyset; \emptyset \rangle$
and $D$ reads $R_D$ writes $W_D$ and $\langle W_D; R_D \rangle \leq \langle W; R \rangle$.

For proofs, see Appendix A.

7 RELATED WORK

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

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

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 $2 \cdot n = \langle \langle \text{leaf}, \text{leaf} \rangle, n \rangle$ and $1 \cdot n = \langle \langle \text{leaf}, n \rangle \rangle$ share the right subtree $n$—which allows programmers to deterministically assign names.
Compilers have long used alias analysis to support optimization passes. Brandauer et al. (2015) extend alias analysis with disjointness domains, which can express local (as well as global) aliasing constraints. Such local constraints are more fine-grained than classic region systems; our work differs in having a rich structure on names.

**Techniques for general-purpose incremental computation.** Incremental algorithms, variously called *online algorithms* and *dynamic algorithms*, are expressly designed to be run repeatedly on changing inputs. Through careful language design, modern incremental computation (IC) abstractions elevate implementation questions, such as “how does this particular change pattern affect a particular incremental state of the system?” into more general questions, answered through special programming abstractions. These IC abstractions identify changing data and reusable sub-computations (Acar et al. 2006a; Hammer et al. 2009, 2014, 2015; Mitschke et al. 2014), and they allow the programmer to relate the expression of an incremental algorithm to the ordinary version of the algorithm that operates over fixed, unchanging input: The gap between these two programs is witnessed by the special abstractions offered by the incremental language.

Through careful algorithm and run-time system design, these IC abstractions admit a fast change-propagation implementation. In particular, after an initial run of the program, as the input changes dynamically, change propagation provides a general (provably sound) approach for recomputing the affected output (Acar et al. 2006a; Acar and Ley-Wild 2009; Hammer et al. 2015). Further, incremental computation can deliver asymptotic speedups (Acar et al. 2007, 2008; Sümmer et al. 2011; Chen et al. 2012). These IC abstractions exist in many languages (Shankar and Bodik 2007; Hammer et al. 2009; Acar and Ley-Wild 2009).

As noted in the Sec. 1, the type and effect we propose is complementary to the general-purpose languages and libraries for incremental computation, including past work on self-adjusting computation. In particular, we expect that variations of the proposed type system could verify the usage of names in much of the work cited above.

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

8 **CONCLUSION AND FUTURE WORK**

In the context of programming language techniques for general-purpose incremental computation, we define Typed Adapton, the first type system to enforce precise, deterministic allocation names. We show that this type system permits expressing familiar functional programs, and composable library components (generic types and generic functions over these parametric structures). We prove that our type system enforces that well-typed programs use names precisely, without ambiguity. Drawing closer to an implementation, we derive a bidirectional version of the type system, and prove that it corresponds to our declarative type system. A key challenge in implementing the bidirectional system is handling constraints over names, name terms and name sets; toward this goal, we give decidable, syntactic rules to guide these checks.

**Future work: Meta-level programs.** The entire point of incremental computation (IC) is to update input with changes, and then propagate these changes (efficiently) into a changed output. Hence, imperative updates are fundamental to IC. Consequently, future work should follow the direction of Hammer et al. (2014) and give explicit type-based annotations that permit such imperative behavior in explicit locations. We discuss incremental feedback in further detail, below.
Future work: Functional reactive programming with explicit names. Future work will enrich Typed Adaption to express incremental loops with feedback effects. In particular, consider this explicit programming annotation for marking where feedback effects occur in a reactive loop, allowing programmers to combine reactive loops and general-purpose incremental computing:

\[
\Gamma \vdash e : \text{LoopComp } A \triangleright (R \perp F; W \cup F; F)
\]

\[
\Gamma \vdash \text{loop } e \text{ with } F : F A \triangleright (R \perp F; W \cup F; \emptyset)
\]

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

\[
(\text{LoopComp } A \triangleright \epsilon) = (F (A + \text{Thk}[X]) (\text{LoopComp}_X A) \triangleright \epsilon) \triangleright \epsilon
\]

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

The dynamics of this rule would re-run expression \(e\) until it produces its value (if ever) and in so doing, the write effects of expression \(e\) would be free to overwrite the reads of \(e\), creating a feedback loop. In the rule above, the annotation \(\cdots\) over \(F\) makes the over-written set of names \(F\) explicit in the program. To statically distinguish delayed, feedback writes from ordinary (immediate) writes, we may track three (not two) name sets in each effect \(\epsilon\): the read set, the write set, and the set of feedback writes, here \(F\). Operationally, the dynamic semantics treats feedback writes specially, by delaying them until the LoopComp fully completes an iteration. This proposal corresponds closely with prior work on synchronous, discrete-time FRP—especially Krishnaswami (2013).

ACKNOWLEDGMENTS

We thank Roly Perera and Kyle Headley for their comments and suggestions on an earlier draft. This material is based in part upon work supported by a gift from Mozilla, and support from the National Science Foundation under grant number CCF-1619282. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of Mozilla or the National Science Foundation.

REFERENCES


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


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


A  OMITTED PROOFS

In each case of the proof, we write “\(\ast\)” to the left of each goal, as we prove it.

**Theorem 6.7 (Subject Reduction).**

If \(\Gamma_1\) store-type and \(\Gamma_1 \vdash M : \text{Nm} \Rightarrow \text{Nm}\)
and \(S : \Gamma_1 \vdash M : C \triangleright \langle W; R \rangle\)
and \(D : S_1 \vdash M \downarrow S_2; t\)
then there exists \(\Gamma_2 \supseteq \Gamma_1\) such that \(\Gamma_2\) store-type and \(\vdash S_2 : \Gamma_2\) and \(\vdash t : C \triangleright \langle \emptyset; \emptyset \rangle\)
and \(D\) reads \(R_D\) writes \(W_D\) and \(W_D; R_D \leq \langle W; R \rangle\).

**Proof.** By induction on the typing derivation \(S\).

- **Case** \(\Gamma_1 \vdash v : A\)

  \[
  \Gamma_1 \vdash M : \text{ret}((FA) \triangleright \langle \emptyset; \emptyset \rangle)
  \]

  - \(e = t\) and \((S_1 = S_2)\) Given
  - \((R_D = W_D = R = W = \emptyset)\) "
  - \((\Gamma_2 = \Gamma_1)\) Suppose
  - \(\vdash S_2 : \Gamma_2\) by above equalities
  - \(\vdash t : C \triangleright \langle \emptyset; \emptyset \rangle\) "
  - \((W_D; R_D) \leq \langle W; R \rangle\) All are empty

- **Case** \(\Gamma_1 \vdash v : \text{Ref}[X] A\)

  \[
  \Gamma_1 \vdash M : \text{get}(v) : (FA) \triangleright \langle \emptyset; X \rangle
  \]

  - \((W = \emptyset)\) and \((R = X)\) Given
  - \(\exists p. (v = \text{ref } p)\) Lemma 6.5 (Canonical Forms)
  - \(\Gamma_1 \vdash p : X\) "
  - \(\Gamma_1(p) = A\) 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 \((t = \text{ret } (v_p))\) Suppose
  - \((R_D = \{p\})\) and \((W_D = \emptyset = W)\) "
  - \(\vdash S_2 : \Gamma_2\) By above equalities
  - \(\vdash t : C \triangleright \langle \emptyset; \emptyset \rangle\) "
  - \((W_D; R_D) \leq \langle W; R \rangle\) By Def. 6.1
  - \((W_D; R_D) \leq \langle W; R \rangle\) By above equality \(W_D = W = \emptyset\),
  - \(\vdash \Gamma_2 \vdash t : C \triangleright \langle \emptyset; \emptyset \rangle\) All are empty

- **Case** \(\Gamma_1 \vdash v : \text{Thk}[X] \{C \triangleright \epsilon\}\)

  \[
  \Gamma_1 \vdash M : \text{force}(v) : (C \triangleright ((\emptyset; X) \text{ then } \epsilon))
  \]
\[(W = \emptyset) \text{ and } (R = X) \quad \text{Given} \]
\[\Gamma \vdash v : \text{Thk}[X] \quad \text{(C \triangleright e)} \quad \text{Given} \]
\[\exists p. \ (v = \text{thunk } p) \quad \text{Lemma 6.5 (Canonical Forms)} \]
\[\Gamma \vdash p \in X \quad \text{"} \]
\[\Gamma_1(p) = (C \triangleright e) \quad \text{By inversion of value typing} \]
\[\exists p. \ S_1(p) = e_p \quad \text{Inversion on } \vdash S_1 : \Gamma_1 \]
\[S_0 :: \quad \Gamma_1 \vdash e_p : (C \triangleright e) \quad \text{"} \]
\[D_0 :: \quad S_1 \vdash_{m}^{m} e_p \downarrow S_2 : t \quad \text{Inversion of } D \]
\[\vdash S_2 : \Gamma_2 \quad \text{By i.h. on } S_0 \text{ and } D_0 \]
\[\vdash t : C \triangleright (\emptyset, \emptyset) \quad \text{"} \]
\[D_0 \text{ reads } R_{D_0} \text{ writes } W_{D_0} \quad \text{"} \]
\[\langle W_{D_0}, R_{D_0} \rangle \leq (W; R) \quad \text{"} \]
\[D \text{ reads } R_{D_0} \text{ writes } W_{D_0} \quad \text{By Def. 6.1} \]
\[\langle W_{D_0}, R_{D_0} \rangle \leq (W; R) \quad \text{by above equality } W_{D_0} = W = \emptyset, \ldots \text{and inequality } (R_{D_0} = \{p\}) \subseteq (X = R). \]

**Case**

\[
\begin{array}{c}
\Gamma_1 \vdash v : (\text{Nm} \supseteq \text{Nm}) [M'] \\
\Gamma_1 \vdash^{M \circ M'} e_0 : C \triangleright (W; R) \\
\Gamma_1 \vdash^{M \circ M'} v : C \triangleright (W; R) \\
\end{array}
\]

\[
\begin{array}{c}
S_0 :: \quad \Gamma_1 \vdash^{M \circ M'} e_0 : C \triangleright (W; R) \quad \text{Subderivation 2 of } S \\
D :: \quad S_1 \vdash_{m}^{m} \text{scope}(v, e_0) \downarrow S_2 : t \quad \text{Given} \\
D_0 :: \quad S_1 \vdash_{m}^{M \circ M'} e_0 \downarrow S_2 : t \quad \text{By inversion (scope)} \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma_1 \vdash M : \text{Nm} \supseteq \text{Nm} \quad \text{Assumption} \\
\Gamma_1 \vdash v : (\text{Nm} \supseteq \text{Nm}) [M'] \quad \text{Subderivation 1 of } S \\
\Gamma_1 \vdash M' : \text{Nm} \supseteq \text{Nm} \quad \text{By inversion} \\
\Gamma_1, x : \text{Nm} \vdash M' x : \text{Nm} \quad \text{By rule t-app} \\
\Gamma_1, x : \text{Nm} \vdash M (M' x) : \text{Nm} \quad \text{By rule t-app} \\
\Gamma_1 \vdash (\lambda x. M (M' x)) : \text{Nm} \supseteq \text{Nm} \quad \text{By definition of } M \circ M' \\
\end{array}
\]

\[
\begin{array}{c}
\vdash S_2 : \Gamma_2 \quad \text{By i.h. on } S_0 \\
\vdash t : C \triangleright (\emptyset, \emptyset) \quad \text{"} \\
D_0 \text{ reads } R_{D_0} \text{ writes } W_{D_0} \quad \text{"} \\
\langle W_{D_0}, R_{D_0} \rangle \leq (W; R) \quad \text{"} \\
D \text{ reads } R_{D_0} \text{ writes } W_{D_0} \quad \text{By Def. 6.1} \\
\langle W_{D_0}, R_{D_0} \rangle \leq (W; R) \quad \text{By above equalities} \\
\end{array}
\]

**Case**

\[
\begin{array}{c}
\Gamma_1 \vdash v : \text{Nm}[X] \\
\Gamma_1 \vdash e : E \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma_1 \vdash^{M} \text{thunk}(v_1, v_2) : F (\text{Thk}[M(X)] E) \triangleright (M(X); \emptyset) \\
\end{array}
\]
Given from $S$
\[ C = F (\text{Thk}[M(X)] E) \quad \text{and} \quad R = \emptyset \quad \text{and} \quad W = M(X) \]
\[ \Gamma_1 \vdash v : Nm[X] \] Subderivation
\[ (v = \text{name } n) \quad \text{and} \quad (n \in X) \] By Lemma 6.5
\[ M n \downarrow p \quad \text{and} \quad R_D = \emptyset \quad \text{and} \quad W_D = \{p\} \] Given from $D$
\[ S_2 = (S_1, p : e) \] "

Suppose
\[ \Gamma_2 = (\Gamma_1, p : \text{Thk}[p] E) \]
\[ \vdash S_2 : \Gamma_2 \] By rule (Fig. 15)
\[ \Gamma_2(p) = E \] By inversion of value typing
\[ \vdash \text{ref } p : \text{Ref}[p] E \] By rule thunk
\[ \vdash \text{ref } p : \text{Ref}[p] E \] By rule ret
\[ \Downarrow \quad \text{D reads } R_D \quad \text{writes } W_D \quad \text{and} \quad W_D = \{p\} \] By Def. 6.1

\[ n \in X \quad \text{Above} \]
\[ M(n) \in M(X) \quad \text{Name term application is pointwise} \]
\[ M(n) \in W \quad \text{By above equality} \]
\[ M(n) = p \]
\[ \{p\} \subseteq W \quad \text{By set theory} \]
\[ \langle W_D, R_D \rangle \leq \langle W, R \rangle \]

\[ \text{Case} \]
\[ \Gamma_1 \vdash v_1 : Nm[X] \quad \Gamma_1 \vdash v_2 : A \]
\[ \vdash \text{ref } (v_1, v_2) : F (\text{Ref}[M(X)] A) \supset (M(X); \emptyset) \]

Given from $S$
\[ C = F (\text{Ref}[M(X)] A) \quad \text{and} \quad R = \emptyset \quad \text{and} \quad W = M(X) \]
\[ \Gamma_1 \vdash v_1 : Nm[X] \] Subderivation
\[ (v_1 = \text{name } n) \quad \text{and} \quad (n \in X) \] Lemma 6.5 (Canonical Forms)
\[ M n \downarrow p \quad \text{and} \quad R_D = \emptyset \quad \text{and} \quad W_D = \{p\} \] Given from $D$
\[ S_2 = (S_1, p : v_2) \] "

Suppose
\[ \Gamma_2 = (\Gamma_1, p : \text{Ref}[p] A) \]
\[ \vdash S_2 : \Gamma_2 \] By rule (Fig. 15)
\[ \Gamma_2(p) = A \] By inversion of value typing
\[ \vdash \text{ref } p : \text{Ref}[p] A \] By rule ref
\[ \vdash \text{ref } p : \text{Ref}[p] A \] By rule ret
\[ \Downarrow \quad \text{D reads } R_D \quad \text{writes } W_D \quad \text{and} \quad W_D = \{p\} \] By Def. 6.1

\[ n \in X \quad \text{Above} \]
\[ M(n) \in M(X) \quad \text{Name term application is pointwise} \]
\[ M(n) \in W \quad \text{By above equality} \]
\[ M(n) = p \]
\[ \{p\} \subseteq W \quad \text{By set theory} \]
\[ \langle W_D, R_D \rangle \leq \langle W, R \rangle \]
• **Case**
  \[\Gamma \vdash^{M} e_1 : (F \triangleright e_1) \quad \Gamma, x : A \vdash^{M} e_2 : (C \triangleright e_2)\]
  \[\vdash^{M} \text{let} (e_1, x.e_2) : C \triangleright (e_1 \text{ then } e_2)\]

  \[\vdash S_1 \vdash^{M} \Gamma \quad \text{Given}\]
  \[S_1 :: \Gamma_1, x : A \vdash^{M} F \triangleright e_1 \quad \text{Subderivation 1 of } S\]
  \[D_1 :: S_1 \vdash^{M} e_1 \downarrow S_1 \vdash^{M} t_1 \quad \text{Subderivation 1 of } D\]
  
  exists \(\Gamma_2 \supseteq \Gamma_1\) such that \(S_{12} : \Gamma_{12}\)
  
  By i.h. on \(S_1\)
  
  \[\vdash \Gamma_{12} \vdash t_1 : F \triangleright \langle \emptyset, \emptyset \rangle\]
  
  \[D_1 \text{ reads } R_{D_1} \text{ writes } W_{D_1}\]
  
  \[\langle W_{D_1} ; R_{D_1} \rangle \preceq e_1\]
  
  \[\langle W_{D_1} ; R_{D_1} \rangle \preceq \langle W_1, R_1 \rangle\]

  \[\Gamma_{12} \vdash v : A\]
  
  inversion of typing rule \text{ret},
  
  for terminal computation \(t_1\)

  \[S_2 :: \Gamma_1, x : A \vdash^{M} e_2 : C \triangleright e_2 \quad \text{Subderivation 2 of } S\]
  \[\Gamma_{12}, x : A \vdash^{M} e_2 : C \triangleright e_2\]
  
  Lemma 6.3 (Weakening)
  
  \[\Gamma_{12} \vdash^{M} [v/x]e_2 : C \triangleright e_2\]
  
  Lemma 6.4 (Substitution)
  
  \[D_2 :: S_{12} \vdash^{M} [v/x]e_2 \downarrow S_2 \vdash^{M} t_2\]
  
  Subderivation 2 of \(D\)
  
  exists \(\Gamma_2 \supseteq \Gamma_{12} \supseteq \Gamma_1\) such that \(\Gamma_2 \supseteq \Gamma_{12} \supseteq \Gamma_1\)
  
  By i.h. on \(S_2\)

  \[\vdash S_2 \vdash^{M} \Gamma_2\]
  
  \[\vdash \Gamma_2 \vdash^{M} t_2 : C \triangleright \langle \emptyset, \emptyset \rangle\]
  
  \[D_2 \text{ reads } R_{D_2} \text{ writes } W_{D_2}\]
  
  \[\langle W_{D_2} ; R_{D_2} \rangle \preceq e_2\]
  
  \[\langle W_{D_2} ; R_{D_2} \rangle \preceq \langle W_2, R_2 \rangle\]

  \[W_1 \perp W_2 \text{ and } R_1 \perp W_2\]

  Definition of \(e_1\) then \(e_2\)

  \[W_{D_1} \perp W_{D_2} \text{ and } R_{D_1} \perp W_{D_2}\]

  By Def. 6.1

  \[W_D = W_{D_1} \perp W_{D_2}\]

  \[R_D = R_{D_1} \cup (R_{D_2} - W_{D_1})\]

  \[\vdash D \text{ reads } R_{D} \text{ writes } W_{D}\]

  \[\langle W_{D} ; R_{D} \rangle \preceq \langle W, R \rangle\]

  Since \(W_D \subseteq W\) and \(R_D \subseteq R\)

• **Case**
  \[\Gamma \vdash^{M} e : ((A \rightarrow E) \triangleright e_1) \quad \Gamma \vdash^{M} v : A\]
  
  \[\vdash^{M} (e \ v) : (E \text{ after } e_1)\]

  Similar to the case for \text{let}.

• **Case**
  \[\Gamma \vdash^{M} v : (A_1 \times A_2) \quad \Gamma, x_1 : A_1, x_2 : A_2 \vdash^{M} e : E\]
  
  \[\vdash^{M} \text{split } (v, x_1, x_2.e) : E\]

  Similar to the case for \text{let}, using Lemma 6.5 (Canonical Forms).
Case: \[ \Gamma, x_1 : A_1 \vdash_{\Gamma} M_1 : E \]
\[ \Gamma, x_2 : A_2 \vdash_{\Gamma} M_2 : E \]
\[ \Gamma \vdash_{\Gamma} \text{case}(\nu, x_1. e_1, x_2. e_2) : E \]

Similar to the case for let, using Lemma 6.5 (Canonical Forms).

Case: Given \( \Gamma \vdash \nu M : (\text{Nm} \xhookrightarrow{\text{Nm}} \text{Nm})[M] \)
\( \Gamma \vdash \nu : \text{Nm}[i] \)
\[ \Gamma \vdash (\nu M \nu) : F(\text{Nm}[\Gamma[1,2]]) \triangleright (\emptyset, \emptyset) \]

By rule name
\[ \Gamma \vdash \text{ret}(\text{name} p) : F(\text{Nm}[\Gamma[1,2]]) \triangleright (\emptyset, \emptyset) \]

By rule ret
\[ \mathcal{D} \text{ by Eval-name-app reads } \emptyset \text{ writes } \emptyset \]

By Def. 6.1
\[ (R_{\mathcal{D}} = R = \emptyset), (W_{\mathcal{D}} = W = \emptyset) \]

By above equalities

Case: \[ \Gamma, a : \gamma, P \vdash_{\Gamma} t : E \]
\[ \Gamma \vdash_{\Gamma} t : (\forall a : \gamma | P, E) \]

By AllIndexIntro
\[ S_0 : \Gamma_1, \alpha : \gamma, \Gamma \vdash M \vdash E \quad \text{Subderivation} \]
\[ D_0 :: S_1 \vdash M \vdash E \quad \text{Subderivation} \]
\[ \exists \Gamma_2 \subseteq \Gamma_1 \quad \text{By i.h.} \]
\[ + S_2 : \Gamma_2 \quad \text{``} \]
\[ D_0 \text{ reads } R_{D_0} \text{ writes } W_{D_0} \quad \text{``} \]
\[ \Gamma_2 \vdash t : E \quad \text{``} \]
\[ \langle R_{D_0}, W_{D_0} \rangle \preceq \langle R; W \rangle \quad \text{``} \]
\[ \Gamma_2 \vdash M : (\forall \alpha : \gamma, E) \quad \text{By typing rule} \]
\[ D \text{ reads } R_D \text{ writes } W_D \quad \text{By Def. 6.1} \]
\[ \langle R_D, W_D \rangle \preceq \langle R; W \rangle \quad \text{By set theory} \]

**Case**  
\[ \Gamma_1 \vdash M \vdash E : (\forall \alpha : \gamma, \Gamma, \alpha) \quad \Gamma_1 \vdash i : \gamma \quad \text{extract}(\Gamma_1) \vdash [i/\alpha] P \]
\[ \Gamma_1 \vdash M : ([i/\alpha] E) \quad \text{AllIndexElim} \]

\[ S_0 :: \Gamma_1 \vdash M : (\forall \alpha : \gamma, E) \quad \text{Subderivation} \]
\[ D_0 :: S_1 \vdash M \vdash E \quad \text{Subderivation} \]
\[ \exists \Gamma_2 \subseteq \Gamma_1 \quad \text{By i.h.} \]
\[ + S_2 : \Gamma_2 \quad \text{``} \]
\[ D_0 \text{ reads } R_{D_0} \text{ writes } W_{D_0} \quad \text{``} \]
\[ \Gamma_2 \vdash t : (\forall \alpha : \gamma, E) \quad \text{``} \]
\[ \langle R_{D_0}, W_{D_0} \rangle \preceq \langle R; W \rangle \quad \text{``} \]
\[ \Gamma_2 \vdash t : (\forall \alpha : \gamma, E) \quad \text{By weakening} \]
\[ \Gamma_2 \vdash t : [i/\alpha] E \quad \text{By typing rule} \]
\[ D \text{ reads } R_D \text{ writes } W_D \quad \text{By Def. 6.1} \]
\[ \langle R_D, W_D \rangle \preceq \langle R; W \rangle \quad \text{By set theory} \]

**Case**  
\[ \Gamma, \alpha : K \vdash M : E \quad \text{AllIntro} \]
\[ \Gamma \vdash M : (\forall \alpha : K, E) \quad \text{AllIntro} \]

Similar to the AllIntroIntro case.

**Case**  
\[ \Gamma \vdash M : (\forall \alpha : K, E) \quad \Gamma \vdash A : K \quad \text{AllElim} \]
\[ \Gamma \vdash M : [A/\alpha] E \quad \text{AllElim} \]

Similar to the AllIndexElim case. □

### B  BIDIRECTIONAL TYPING

#### B.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.
\[ \Gamma \vdash v \Rightarrow A \]  
Under \( \Gamma \), value \( v \) synthesizes type \( A \)

\[ \Gamma \vdash v \Leftarrow A \]  
Under \( \Gamma \), value \( v \) checks against type \( A \)

\[
\begin{align*}
\Gamma \vdash \varnothing & \Leftarrow \text{unit} & \Gamma \vdash v_1 \Leftarrow A_1 & \Gamma \vdash v_2 \Leftarrow A_2 & \Gamma \vdash (v_1, v_2) \Leftarrow (A_1 \times A_2) & \Gamma \vdash (\text{name } n) \Leftarrow \text{Nm}[X] & \Gamma \vdash \varnothing \Rightarrow \text{unit} & \Gamma \vdash (\text{name } n) \Rightarrow \text{Nm}[X] \\
\Gamma \vdash M_v \Rightarrow (\text{Nm} \xrightarrow{\text{Nm}} \text{Nm}) & \text{M}_v = \beta M & \Gamma \vdash (\text{name } n) \Rightarrow \text{Nm}[X] & \Gamma \vdash (\text{name } n) \Rightarrow \text{Nm}[X] & \Gamma \vdash (\text{name } n) \Rightarrow \text{Nm}[X] \\
\Gamma \vdash (\text{thunk } n) \Leftarrow (\text{Thk}[X] E) & \Gamma \vdash v \Rightarrow A_1 & A_1 = A_2 & \Gamma \vdash v \Leftarrow A_2
\end{align*}
\]

\( \text{vchk-conv} \)

\( \text{vchk-unit} \)

\( \text{vchk-pair} \)

\( \text{vchk-name} \)

\( \text{vchk-namefn} \)

\( \text{vchk-ref} \)

\( \text{vchk-thunk} \)

\( \text{vchk-conv} \)

Fig. 17. Bidirectional value typing

B.2 Bidirectional Typing Rules

The typing rules in Figures 12 and 13 are declarative: they define what typings are valid, but not how to derive those typings. The rules’ use of names and effects annotations means that standard unification-based techniques, like Damas–Milner inference, are not clearly applicable. For example, it is not obvious when to apply \( \text{chk-AllIntro} \).

Following the tradition of DML, 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 \( \cdots \cdot e : E \) 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 \( \Gamma \)).

In formulating the bidirectional versions of value and computation typing (Figures 17 and 18), 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 \( \text{chk-lam} \), 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) \( \text{'vchk-conv'} \) for value typing, or (2) \( \text{'chk-conv'} \) or \( \text{'chk-eff-subsume'} \) for computation typing. The latter rule allows effect subsumption; together, \( \text{'chk-conv'} \) and \( \text{'chk-eff-subsume'} \) encode a subsumption rule, common in bidirectional systems with subtyping, such as Dunfield and Pfenning (2004).
\[\Gamma \vdash E\] Under \(\Gamma\), within namespace \(M\), computation \(e\) synthesizes type-with-effects \(E\)

\[
\Gamma \vdash e \Leftarrow E \quad \text{syn-anno} \\
\Gamma \vdash (e : E) \Rightarrow E \\
\Gamma \vdash (e : E) \Rightarrow (E \text{ after } \epsilon_1) \quad \text{syn-app}
\]

\[\Gamma \vdash v \Rightarrow \text{Thk}[X] (C \triangleright e) \quad \text{syn-force} \\
\Gamma \vdash v \Rightarrow \text{Ref}[X] A \\
\Gamma \vdash (C \triangleright (\emptyset; X) \text{ then } \epsilon)) \quad \text{syn-get}
\]

\[\Gamma \vdash (\text{N} m \triangleright \text{N} m)[M] \quad \Gamma \vdash v \Rightarrow \text{N} m[i] \quad \text{syn-name-app} \\
\Gamma \vdash \chi[i] \Rightarrow [i/a]E \quad \text{syn-AllIndexElim} \\
\Gamma \vdash e[A : K] \Rightarrow [A/\alpha]E \quad \text{syn-AllElim}
\]

\[\Gamma \vdash E \Rightarrow E_1 \quad \text{chk-conv} \\
\Gamma \vdash e \Leftarrow E_2 \quad \text{chk-eff-subsume} \\
\Gamma \vdash \chi[i] : A_1 \times A_2 \quad \text{chk-split} \\
\Gamma \vdash e \Leftarrow E \quad \text{chk-case(v, x_1, x_2, e)} \Leftarrow E \quad \text{chk-case}
\]

\[\Gamma \vdash v \Leftarrow A \quad \text{chk-ret} \\
\Gamma \vdash \chi : A \vdash e \Leftarrow E \quad \text{chk-let(e_1, x, e_2)} \Leftarrow (C \triangleright (\epsilon_1 \text{ then } \epsilon_2)) \quad \text{chk-let}
\]

\[\Gamma \vdash (\lambda x . e) \Leftarrow ((A \rightarrow E) \triangleright (\emptyset; X)) \quad \text{chk-lam} \\
\Gamma \vdash v_1 \Leftarrow \text{N} m[X] \quad \text{chk-thunk(v, e)} \Leftarrow (F (\text{Thk}[M[X]] E) \triangleright (M[X]; \emptyset)) \quad \text{chk-thunk}
\]

\[\Gamma \vdash v \Rightarrow (\text{N} m \triangleright \text{N} m)[N'] \quad \text{chk-scope(v, e)} \Leftarrow (C \triangleright (W[R])) \quad \text{chk-scope}
\]

\[\Gamma \vdash t \Leftarrow (\forall \alpha : \gamma | P . E) \quad \text{chk-AllIntro} \\
\Gamma \vdash t \Leftarrow (\forall \alpha : K . E) \quad \text{chk-AllIntro} \\
\]

Fig. 18. Bidirectional computation typing
C BIDIRECTIONAL TYPING PROOFS

Theorem C.1 (Soundness of Bidirectional Value 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 \( (x : A) \in \Gamma \)

\[ \frac{}{\Gamma \vdash x : A} \text{vsyn-var} \]

Given

\[ \Gamma \vdash x : A \]

By rule var

\[ \|x\| = x \]

By definition of \( \|\| \)

\[ \Gamma \vdash v' : A \text{ and } \|v\| = v' \text{ where } v' = x \text{ and } v = x \]

Case \( \Gamma \vdash v_1 \Leftarrow A \)

\[ \frac{}{\Gamma \vdash \{v_1 : A\} \Rightarrow \{A\}} \text{vsyn-anno} \]

\[ \exists v'_1 \text{ such that } \Gamma \vdash v'_1 : A \text{ and } \|v_1\| = v'_1 \]

By inductive hypothesis

\[ \|\{v_1 : A\}\| = \|v_1\| = v'_1 \]

By definition of \( \|\| \); Since \( \|v_1\| = v'_1 \)

\[ \Gamma \vdash v' : A \text{ and } \|v\| = v' \text{ where } v' = v'_1 \text{ and } v = \{v_1 : A\} \]

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

Case \( \Gamma \vdash \{\} \Leftarrow \text{unit} \)

\[ \frac{}{\Gamma \vdash \{\} : \text{unit}} \text{vchk-unit} \]

\[ \|\{\}\| = \{\} \]

By definition of \( \|\| \)

\[ \Gamma \vdash v' : \{\} \text{ and } \|v\| = v' \text{ where } v' = \{\} \text{ and } v = \{\} \]

Case \( \Gamma \vdash v_1 \Leftarrow A_1 \quad \Gamma \vdash v_2 \Leftarrow A_2 \)

\[ \frac{}{\Gamma \vdash \langle v_1, v_2 \rangle \Leftarrow (A_1 \times A_2)} \text{vchk-pair} \]

\[ \exists v'_1 \text{ such that } \Gamma \vdash v'_1 : A_1 \text{ and } \|v_1\| = v'_1 \]

By inductive hypothesis

\[ \exists v'_2 \text{ such that } \Gamma \vdash v'_2 : A_2 \text{ and } \|v_2\| = v'_2 \]

By inductive hypothesis

\[ \Gamma \vdash \langle v'_1, v'_2 \rangle : (A_1 \times A_2) \]

By rule pair

\[ \|\langle v_1, v_2 \rangle\| = \|v_1\|, \|v_2\| = \langle v'_1, v'_2 \rangle \]

By definition of \( \|\| \); Since \( \|v_1\| = v'_1, \|v_2\| = v'_2 \)

\[ \Gamma \vdash v' : (A_1 \times A_2) \text{ and } \|v\| = v' \text{ where } v' = \langle v'_1, v'_2 \rangle \text{ and } v = \langle v_1, v_2 \rangle \]

Case \( \Gamma \vdash n \in X \)

\[ \frac{}{\Gamma \vdash \text{name} n \Leftarrow Nm[X]} \text{vchk-name} \]

Given

\[ \Gamma \vdash n \in X \]

By rule name

\[ \|\text{name} n\| = \text{name} n \]

By definition of \( \|\| \)

\[ \Gamma \vdash v' : Nm[X] \text{ and } \|v\| = v' \text{ where } v' = \text{name} n \text{ and } v = \text{name} n \]
Case \( \Gamma \vdash M_v \Rightarrow (\text{Nm} \rightsquigarrow \text{Nm}) \quad M_v =_{\beta} M \) vchk-namefn
\[ \Gamma \vdash (\text{nmfn} M_v) \Leftarrow (\text{Nm} \rightsquigarrow \text{Nm}) [M] \]

\[ \exists M'_v \text{ such that } \Gamma \vdash M'_v : (\text{Nm} \rightsquigarrow \text{Nm}) \text{ and } |M_v| = M'_v \] By inductive hypothesis
\[ M_v =_{\beta} M \] Given
\[ |M_v| =_{\beta} M \] Type erasure does not affect convertibility
\[ M'_v =_{\beta} M \] Since \(|M_v| = M'_v|
\[ \Gamma \vdash (\text{nmfn} M'_v) : (\text{Nm} \Rightarrow \text{Nm}) [M] \] By rule namefn
\[ |(\text{nmfn} M_v)| = |(\text{nmfn} |M_v|)| = |(\text{nmfn} M'_v)| \] By definition of \|\|; Since \(|M_v| = M'_v|
\[ \Gamma \vdash v' : (\text{Nm} \Rightarrow \text{Nm}) [M] \text{ and } |v| = v' \] where \(v' = (\text{nmfn} M'_v)\) and \(v = (\text{nmfn} M_v)\)

Case \( \Gamma \vdash n \in X \quad \Gamma(n) = A \) vchk-ref
\[ \Gamma \vdash (\text{ref n}) \Leftarrow \text{Ref}[X] A \]
\[ \Gamma \vdash n \in X \] Given
\[ \Gamma(n) = A \] Given
\[ \Gamma \vdash (\text{ref n}) : \text{Ref}[X] A \] By rule ref
\[ |(\text{ref n})| = \text{ref} n \] By definition of \|\|
\[ \Gamma \vdash v' : \text{Ref}[X] A \text{ and } |v| = v' \] where \(v' = (\text{ref n})\) and \(v = (\text{ref n})\)

Case \( \Gamma \vdash n \in X \quad \Gamma(n) = E \) vchk-thunk
\[ \Gamma \vdash (\text{thunk n}) \Leftarrow \text{Thk}[X] E \]
\[ \Gamma \vdash n \in X \] Given
\[ \Gamma(n) = E \] Given
\[ \Gamma \vdash (\text{thunk n}) : \text{Thk}[X] E \] By rule thunk
\[ |(\text{thunk n})| = \text{thunk} n \] By definition of \|\|
\[ \Gamma \vdash v' : \text{Thk}[X] E \text{ and } |v| = v' \] where \(v' = (\text{thunk n})\) and \(v = (\text{thunk n})\)

Case \( \Gamma \vdash v_1 \Rightarrow A_1 \quad A_1 = A_2 \) vchk-conv
\[ \Gamma \vdash v_1 \Leftarrow A_2 \]
\[ \exists v'_1 \text{ such that } \Gamma \vdash v'_1 : A_1 \text{ and } |v_1| = v'_1 \] By inductive hypothesis
\[ A_1 = A_2 \] Given
\[ \Gamma \vdash v'_1 : A_2 \] By rule conv
\[ \Gamma \vdash v' : A_2 \text{ and } |v| = v' \] where \(v' = v'_1\) and \(v = v_1\)
\[ \square \]

**Theorem C.2 (Completeness of Bidirectional Value Typing).**

If \( \Gamma \vdash v : A \) then there exist values \(v'\) and \(v''\) such that

1. \( \Gamma \vdash v' \Rightarrow A \text{ and } |v'| = v \)
2. \( \Gamma \vdash v'' \Leftarrow A \text{ and } |v''| = v \)

**Proof.** By induction on the derivation of \( \Gamma \vdash v : A \).
Case

\[ \Gamma \vdash \text{unit} \]

By rule vchk-unit

\[ (\text{unit}) = (\text{unit}) \]

By definition of \([\vdash]\)

\[ \Gamma \vdash \nu'' \iff (\nu'') \text{ and } v'' = v \]

where \(v'' = (\text{unit})\) and \(v = (\text{unit})\)

\[ \Gamma \vdash (\text{unit}) \Rightarrow \text{unit} \]

By rule vsyn-anno

\[ [((\text{unit})) \Rightarrow (\text{unit})] = (\text{unit}) \]

By definition of \([\vdash]\)

\[ \Gamma \vdash \nu' \Rightarrow (\nu') \text{ and } v' = v \]

where \(v = (\text{unit})\) and \(v = (\text{unit})\)

Case

\[ \{x : A\} \in \Gamma \]

\[ \Gamma \vdash x : A \]

\[ \{x : A\} \in \Gamma \]

Given

\[ \Gamma \vdash x \Rightarrow A \]

By rule vsyn-var

\[ |x| = x \]

By definition of \([\vdash]\)

\[ \Gamma \vdash \nu' \Rightarrow A \text{ and } v' = v \]

where \(v' = x\) and \(v = x\)

\[ \Gamma \vdash x \Leftarrow A \]

By rule vchk-conv

\[ \Gamma \vdash \nu'' \Leftarrow A \text{ and } v'' = v \]

where \(v'' = x\) and \(v = x\); Since \(|x| = x\)

Case

\[ \Gamma \vdash \nu_1 : A_1 \] \hspace{1cm} \[ \Gamma \vdash \nu_2 : A_2 \]

\[ \Gamma \vdash (\nu_1, \nu_2) : (A_1 \times A_2) \]

pair

\[ \exists \nu'_1 \text{ such that } \Gamma \vdash \nu'_1 \Leftarrow A_1 \text{ and } |\nu'_1| = \nu_1 \]

By inductive hypothesis

\[ \exists \nu'_2 \text{ such that } \Gamma \vdash \nu'_2 \Leftarrow A_2 \text{ and } |\nu'_2| = \nu_2 \]

By inductive hypothesis

\[ \Gamma \vdash (\nu'_1, \nu'_2) \Leftarrow (A_1 \times A_2) \]

By rule vchk-pair

\[ |(\nu'_1, \nu'_2)| = (|\nu'_1|, |\nu'_2|) = (\nu_1, \nu_2) \]

By definition of \([\vdash] \); \(|\nu'_1| = \nu_1 \); \(|\nu'_2| = \nu_2 \)

\[ \Gamma \vdash \nu'' \Leftarrow (A_1 \times A_2) \text{ and } |\nu''| = \nu \]

where \(\nu'' = (\nu'_1, \nu'_2)\) and \(v = (\nu_1, \nu_2)\)

\[ \Gamma \vdash (\nu''_1, \nu''_2) : (A_1 \times A_2) \]

By rule vsyn-anno

\[ |(\nu''_1, \nu''_2)| = (|\nu''_1|, |\nu''_2|) = (\nu_1, \nu_2) \]

By definition of \([\vdash] \); \(|\nu''_1| = \nu_1 \); \(|\nu''_2| = \nu_2 \)

\[ \Gamma \vdash \nu' \Rightarrow (A_1 \times A_2) \text{ and } |\nu'| = \nu \]

where \(\nu' = (\nu''_1, \nu''_2)\) and \(v = (\nu_1, \nu_2)\)

Case

\[ \Gamma \vdash n \in X \]

\[ \Gamma \vdash \text{name} n \text{ : } \text{Nm}[X] \]

name

\[ \Gamma \vdash n \in X \]

Given

\[ \Gamma \vdash \text{name} n \Leftarrow \text{Nm}[X] \]

By rule vchk-name

\[ |\text{name} n| = |\text{name} n| \]

By definition of \([\vdash]\)

\[ \Gamma \vdash \nu'' \Leftarrow \text{Nm}[X] \text{ and } |\nu''| = \nu \]

where \(\nu'' = \text{name} n\) and \(v = \text{name} n\)

\[ \Gamma \vdash \text{name} n : \text{Nm}[X] \Rightarrow \text{Nm}[X] \]

By rule vsyn-anno

\[ |(\text{name} n : \text{Nm}[X])| = |(\text{name} n)| = |(\text{name} n)| \]

By definition of \([\vdash]\)

\[ \Gamma \vdash \nu' \Rightarrow \text{Nm}[X] \text{ and } |\nu'| = \nu \]

where \(\nu' = \text{name} n : \text{Nm}[X]\) and \(v = \text{name} n\)

, Vol. 1, No. 1, Article 1. Publication date: January 2016.
Case \[ \Gamma 
vdash M_v : (\text{Nm} \Rightarrow \text{Nm}) \quad M_v =_{\text{\beta}} M \]
\[ \Gamma 
vdash (\text{nmfn} M_v) : (\text{Nm} \Rightarrow \text{Nm})[M] \]

\[ \exists M'_v \text{ such that} \]
\[ \Gamma 
vdash M'_v : (\text{Nm} \Rightarrow \text{Nm}) \text{ and } |M'_v| = M_v \]
\[ \text{By inductive hypothesis} \]
\[ M_v =_{\text{\beta}} M \]
\[ |M'_v| =_{\beta} M \]
\[ M'_v =_{\beta} M \]
\[ \Gamma 
vdash (\text{nmfn} M'_v) : (\text{Nm} \Rightarrow \text{Nm})[M] \]
\[ \text{By rule vchk-namefn} \]
\[ ||(\text{nmfn} M'_v)|| = (\text{nmfn} |M'_v|) = (\text{nmfn} M_v) \]
\[ \text{By definition of } \Rightarrow; \text{ Since } |M'_v| = M_v \]
\[ \text{Type annotation does not affect convertibility} \]
\[ \text{By rule vchk-namefn} \]
\[ ||(\text{nmfn} M'_v)|| = (\text{nmfn} M_v) \]
\[ \text{By definition of } \Rightarrow \]
\[ \therefore \Gamma 
vdash M'' \leftarrow (\text{Nm} \Rightarrow \text{Nm})[M] \text{ and } |v''| = v \]
\[ \text{where } v'' = (\text{nmfn} M'_v) \text{ and } v = (\text{nmfn} M_v) \]

Case \[ \Gamma 
vdash n \in X \quad \Gamma(n) = A \]
\[ \Gamma 
vdash (\text{ref } n) : \text{Ref}[X] A \]
\[ \text{ref} \]
\[ \Gamma 
vdash n \in X \quad \text{Given} \]
\[ \Gamma(n) = A \quad \text{Given} \]
\[ \Gamma 
vdash (\text{ref } n) \leftarrow \text{Ref}[X] A \quad \text{By rule vchk-ref} \]
\[ ||(\text{ref } n)|| = (\text{ref } n) \quad \text{By definition of } \Rightarrow \]
\[ \therefore \Gamma 
vdash v'' \leftarrow \text{Ref}[X] A \text{ and } |v''| = v \quad \text{where } v'' = (\text{ref } n) \text{ and } v = (\text{ref } n) \]

Case \[ \Gamma 
vdash n \in X \quad \Gamma(n) = E \]
\[ \Gamma 
vdash (\text{thunk } n) : (\text{Thk}[X] E) \]
\[ \text{thunk} \]
\[ \Gamma 
vdash n \in X \quad \text{Given} \]
\[ \Gamma(n) = E \quad \text{Given} \]
\[ \Gamma 
vdash (\text{thunk } n) \leftarrow (\text{Thk}[X] E) \quad \text{By rule vchk-thunk} \]
\[ ||(\text{thunk } n)|| = (\text{thunk } n) \quad \text{By definition of } \Rightarrow \]
\[ \therefore \Gamma 
vdash v'' \leftarrow (\text{Thk}[X] E) \text{ and } |v''| = v \quad \text{where } v'' = (\text{thunk } n) \text{ and } v = (\text{thunk } n) \]
\[ \Gamma 
vdash (\text{thunk } n : (\text{Thk}[X] E)) \Rightarrow (\text{Thk}[X] E) \quad \text{By rule vchk-namefn} \]
\[ ||(\text{thunk } n : (\text{Thk}[X] E))|| = |\text{thunk } n| = \text{thunk } n \quad \text{By definition of } \Rightarrow \]
\[ \therefore \Gamma 
vdash v' \Rightarrow (\text{Thk}[X] E) \text{ and } |v'| = v \quad \text{where } v' = (\text{thunk } n : (\text{Thk}[X] E)) \text{ and } v = (\text{thunk } n) \]

\[ \square \]

Theorem C.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 $|e| = e'$

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

Proof. By induction on the given derivation.

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

\textbf{Case}

\[ \Gamma \vdash^M e_1 \Rightarrow ((A \rightarrow E) \triangleright e_1) \]

\[ \Gamma \vdash v \Leftarrow A \]

\[ \vdash^M (e_1 \triangleright v) \Rightarrow (E \text{ after } e_1) \]

\[ \exists e'_1 \text{ such that } \Gamma \vdash^M e'_1 : ((A \rightarrow E) \triangleright e_1) \text{ and } |e_1| = e'_1 \] By inductive hypothesis

\[ \exists v' \text{ such that } \Gamma \vdash v' : A \text{ and } |v| = v' \] By Thm. C.1

\[ \Gamma \vdash^M (e'_1 \triangleright v') : (E \text{ after } e_1) \] By rule app

\[ |(e_1 \triangleright v)| = (|e_1| \triangleright |v|) = (e'_1 \triangleright v') \] By definition of $|\cdot|$

\[ \vdash^M e' : (E \text{ after } e_1) \text{ and } |e| = e' \] where $e' = (e'_1 \triangleright v')$ and $e = (e_1 \triangleright v)$

\textbf{Case}

\[ \Gamma \vdash v \Rightarrow \text{Thk}[X] (C \triangleright e) \]

\[ \Gamma \vdash^M \text{force}(v) \Rightarrow (C \triangleright ((\emptyset; X) \text{ then } e)) \]

\[ \exists v' \text{ such that } \Gamma \vdash v' : \text{Thk}[X] (C \triangleright e) \text{ and } |v| = v' \] By Thm. C.1

\[ \Gamma \vdash^M \text{force}(v') : (C \triangleright ((\emptyset; X) \text{ then } e)) \] By rule force

\[ |\text{force}(v)| = |\text{force}(v)| = \text{force}(v') \] By definition of $|\cdot|$; Since $|v| = v'$

\[ \vdash^M e' : (C \triangleright ((\emptyset; X) \text{ then } e)) \text{ and } |e| = e' \] where $e' = \text{force}(v')$ and $e = \text{force}(v)$

\textbf{Case}

\[ \Gamma \vdash v \Rightarrow \text{Ref}[X] A \]

\[ \Gamma \vdash^M \text{get}(v) \Rightarrow (F A) \triangleright (\emptyset; X) \]

\[ \exists v' \text{ such that } \Gamma \vdash v' : \text{Ref}[X] A \text{ and } |v| = v' \] By Thm. C.1

\[ \Gamma \vdash^M \text{get}(v') : (F A) \triangleright (\emptyset; X) \] By rule get

\[ |\text{get}(v)| = |\text{get}(v)| = |\text{get}(v')| \] By the definition of $|\cdot|$; Since $|v| = v'$

\[ \vdash^M e' : (F A) \triangleright (\emptyset; X) \text{ and } |e| = e' \] where $e' = \text{get}(v')$ and $e = \text{get}(v)$

\textbf{Case}

\[ \Gamma \vdash v_M \Rightarrow (\text{Nm} \overset{\gamma}{\Rightarrow} \text{Nm}) [M] \]

\[ \Gamma \vdash v \Rightarrow \text{Nm}[i] \]

\[ \vdash^N (v_M \triangleright v) \Rightarrow F (\text{Nm}[M[i]]) \triangleright (\emptyset; 0) \]

\[ \exists v'_M \text{ such that } \Gamma \vdash v'_M : (\text{Nm} \overset{\gamma}{\Rightarrow} \text{Nm}) [M] \text{ and } |v_M| = v'_M \] By Thm. C.1

\[ \exists v' \text{ such that } \Gamma \vdash v' : \text{Nm}[i] \text{ and } |v| = v' \] By Thm. C.1

\[ \vdash^N (v'_M \triangleright v') : F (\text{Nm}[M[i]]) \triangleright (\emptyset; 0) \] By rule name-app

\[ |(v_M \triangleright v)| = |(v_M| \triangleright |v'_M| \triangleright v'_M|) \]

By definition of $|\cdot|$; $|v_M| = v'_M$; $|v| = v'$

\[ \vdash^M e' : F (\text{Nm}[M[i]]) \triangleright (\emptyset; 0) \text{ and } |e| = e' \] where $e' = (v'_M \triangleright v')$ and $e = (v_M \triangleright v)$

\textbf{Case}

\[ \Gamma \vdash \triangleright (\forall a : \gamma, E) \]

\[ \Gamma \vdash^M i : \gamma \]

\[ \vdash^M e[i] \Rightarrow [i/a]E \]

\[ \text{syn-AllIndexElim} \]
\[ \exists e' \text{ such that } \Gamma \vdash^M e' : (\forall x : \gamma, E) \text{ and } \mid e \mid = e' \quad \text{By inductive hypothesis} \]

\[ \Gamma \vdash i : \gamma \quad \text{Given} \]

\[ e[i] = \mid |e'|_1 \mid \quad \text{By } \mid |e| = e' \]

\[ \Gamma \vdash^M e' : [i/\alpha]E \quad \text{By rule AllIndexElim} \]

**Case** \[ \Gamma \vdash^M e \Rightarrow (\forall x : K, E) \quad \Gamma \vdash A : K \]

\[ \Gamma \vdash^M e[A] \Rightarrow [A/\alpha]E \quad \text{syn-AllElim} \]

Similar to the syn-AllIndexElim case.

**Case** \[ \Gamma \vdash^M e_1 \Leftarrow E \]

\[ \Gamma \vdash^M (e_1 : E) \Rightarrow E \quad \text{syn-anno} \]

\[ \exists e'_1 \text{ such that } \Gamma \vdash^M e'_1 : E \text{ and } \mid e_1 \mid = e'_1 \quad \text{By inductive hypothesis} \]

\[ \mid e_1 : E \mid = \mid e_1 \mid = e'_1 \quad \text{By the definition of } \mid - \mid ; \text{ Since } \mid e_1 \mid = e'_1 \]

\[ \Gamma \vdash^M e'_1 : E \text{ and } \mid e \mid = e' \quad \text{where } e' = e'_1 \text{ and } e = (e_1 : E) \quad \text{By inductive hypothesis} \]

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

**Case** \[ \Gamma \vdash e \Rightarrow (C \triangleright e_1) \]

\[ \Gamma \vdash e_1 \Leftarrow (C \triangleright e_2) \quad \text{chk-eff-subsume} \]

\[ \exists e' \text{ such that } \Gamma \vdash^M e' : (C \triangleright e_1) \text{ and } \mid e \mid = e' \quad \text{By inductive hypothesis} \]

\[ e_1 \leq e_2 \quad \text{Given} \]

\[ \Gamma \vdash^M e' : (C \triangleright e_2) \quad \text{By rule eff-subsume} \]

**Case** \[ \Gamma \vdash v \Rightarrow (A_1 \times A_2) \]

\[ \Gamma, x_1 : A_1, x_2 : A_2 \vdash^M e_1 \Leftarrow E \quad \text{chk-split} \]

\[ \Gamma \vdash^M \text{split}(v, x_1, x_2, e_1) \Leftarrow E \]

\[ \exists v' \text{ such that } \Gamma \vdash v' : (A_1 \times A_2) \text{ and } \mid v \mid = v' \quad \text{By Thm. C.1} \]

\[ \exists e'_1 \text{ such that } \Gamma, x_1 : A_1, x_2 : A_2 \vdash^M e'_1 : E \text{ and } \mid e_1 \mid = e'_1 \quad \text{By inductive hypothesis} \]

\[ \Gamma \vdash^M \text{split}(v, x_1, x_2, e'_1) : E \quad \text{By rule split} \]

\[ = \text{split}(v', x_1, x_2, e'_1) \quad \text{By definition of } \mid - \mid ; \text{ Since } \mid v \mid = v', \mid e_1 \mid = e'_1 \]

\[ \Gamma \vdash^M e' : E \text{ and } \mid e \mid = e' \quad \text{where } e' = \text{split}(v', x_1, x_2, e'_1) \]

\[ \text{and } e = \text{split}(v, x_1, x_2, e_1) \]

**Case** \[ \Gamma \vdash v \Rightarrow (A_1 + A_2) \]

\[ \Gamma, x_1 : A_1 \vdash^M e_1 \Leftarrow E \]

\[ \Gamma, x_2 : A_2 \vdash^M e_2 \Leftarrow E \]

\[ \Gamma \vdash^M \text{case}(v, x_1, e_1, x_2, e_2) \Leftarrow E \quad \text{chk-case} \]
\[\exists \nu' \text{ such that } \Gamma \vdash \nu' : (A_1 + A_2) \text{ and } |\nu| = \nu' \quad \text{By Thm. C.1}\]
\[\exists e_1' \text{ such that } \Gamma, x_1 : A_1 \vdash M e_1' : E \text{ and } |e_1| = e_1' \quad \text{By inductive hypothesis}\]
\[\exists e_2' \text{ such that } \Gamma, x_2 : A_2 \vdash M e_2' : E \text{ and } |e_2| = e_2' \quad \text{By inductive hypothesis}\]
\[\Gamma \vdash \text{case}(\nu', x_1.e_1', x_2.e_2') : E \quad \text{By rule case}\]
\[|\text{case}(\nu, x_1.e_1, x_2.e_2)| = \text{case}(|\nu|, x_1.|e_1|, x_2.|e_2|) \quad \text{By definition of } -|\]
\[= \text{case}(\nu', x_1.e_1', x_2.e_2') \quad \text{Since } |\nu| = \nu', |e_1| = e_1', |e_2| = e_2'\]
\[\Gamma \vdash e' : E \text{ and } |e| = e' \quad \text{where } e' = \text{case}(\nu', x_1.e_1', x_2.e_2') \text{ and } e = \text{case}(\nu, x_1.e_1, x_2.e_2)\]

**Case**

\[
\begin{align*}
\Gamma \vdash \nu & \iff A \\
\Gamma \vdash M \text{ret}(\nu) & \iff ((F A) \triangleright (\emptyset; \emptyset)) \\
\Gamma \vdash \nu' & \iff A \text{ and } |\nu| = \nu' & \text{By Thm. C.1}\end{align*}
\]

\[\exists \nu' \text{ such that } \Gamma \vdash \nu' : A \text{ and } |\nu| = \nu' \quad \text{By Thm. C.1}\]
\[\Gamma \vdash M \text{ret}(\nu') : ((F A) \triangleright (\emptyset; \emptyset)) \quad \text{By rule ret}\]
\[\text{ret}(\nu) = \text{ret}(\nu') = \text{ret}(\nu) \quad \text{By definition of } -|; \text{ Since } |\nu| = \nu'\]

\[\Gamma \vdash M e' : ((F A) \triangleright (\emptyset; \emptyset)) \text{ and } |e| = e' \quad \text{where } e' = \text{ret}(\nu') \text{ and } e = \text{ret}(\nu)\]

**Case**

\[
\begin{align*}
\Gamma \vdash M e_1 & \Rightarrow (F A) \triangleright e_1 \\
\Gamma, x : A \vdash M e_2 & \iff (C \triangleright e_2) \\
\Gamma \vdash M \text{let}(e_1, x.e_2) & \iff (C \triangleright (e_1 \text{ then } e_2)) & \text{chk-let}\end{align*}
\]

\[\exists e_1' \text{ such that } \Gamma \vdash M e_1' : (F A) \triangleright e_1 \text{ and } |e_1| = e_1' \quad \text{By inductive hypothesis}\]
\[\exists e_2' \text{ such that } \Gamma, x : A \vdash M e_2' : (C \triangleright e_2) \text{ and } |e_2| = e_2' \quad \text{By inductive hypothesis}\]
\[\Gamma \vdash M \text{let}(e_1', x.e_2') : (C \triangleright (e_1 \text{ then } e_2)) \quad \text{By rule let}\]
\[|\text{let}(e_1, x.e_2)| = \text{let}(|e_1|, x.|e_2|) \quad \text{By definition of } -|\]
\[= \text{let}(e_1', x.e_2') \quad \text{Since } |e_1| = e_1', |e_2| = e_2'\]

\[\Gamma \vdash M e' : (C \triangleright (e_1 \text{ then } e_2)) \text{ and } |e| = e' \quad \text{where } e' = \text{let}(e_1', x.e_2') \text{ and } e = \text{let}(e_1, x.e_2)\]

**Case**

\[
\begin{align*}
\Gamma, x : A \vdash M e_1 & \iff E \\
\Gamma \vdash M (\lambda x.e_1) & \iff ((A \rightarrow E) \triangleright (\emptyset; \emptyset)) & \text{chk-lam}\end{align*}
\]

\[\exists e_1' \text{ such that } \Gamma, x : A \vdash M e_1' : E \text{ and } |e_1| = e_1' \quad \text{By inductive hypothesis}\]
\[\Gamma \vdash M (\lambda x.e_1') : ((A \rightarrow E) \triangleright (\emptyset; \emptyset)) \quad \text{By rule lam}\]
\[|\lambda x.e_1| = |\lambda x.e_1'| = (\lambda x.e_1') \quad \text{By definition of } -|; \text{ Since } |e_1| = e_1'\]

\[\Gamma \vdash M e' : ((A \rightarrow E) \triangleright (\emptyset; \emptyset)) \text{ and } |e| = e' \quad \text{where } e' = (\lambda x.e_1') \text{ and } e = (\lambda x.e_1)\]

**Case**

\[
\begin{align*}
\Gamma \vdash \nu & \iff \text{Nm}[X] \\
\Gamma \vdash M e_1 & \iff E_1 & \text{chk-thunk}\end{align*}
\]

\[\Gamma \vdash M \text{thunk}(\nu, e_1) \iff (F \text{Thk}[M[X]] E_1) \triangleright (M[X]; \emptyset)\]
Let $E = (F(\text{Thk}[M(X)] E_1)) \vdash (M(X); \emptyset)$  
Assumption

$\exists v' \text{ such that } \Gamma \vdash v' : \text{Nm}[X] \text{ and } |v| = v'$  
By Thm. C.1

$\exists e_1' \text{ such that } \Gamma \vdash e_1' : E \text{ and } |e_1| = e_1'$  
By inductive hypothesis

$\Gamma \vdash \text{thunk}(v', e_1') : E$  
By rule thunk

$\text{thunk}(v, e_1) = \text{thunk}(v, e_1')$  
By definition of $|\cdot|$. Since $|v| = v'$, $|e_1| = e_1'$

$\therefore \Gamma \vdash e' : E \text{ and } |e| = e'$  
where $e' = \text{thunk}(v', e_1')$ and $e = \text{thunk}(v, e_1)$

**Case**  
$\Gamma \vdash v_1 \Leftarrow \text{Nm}[X]$  
$\Gamma \vdash v_2 \Leftarrow A$

$\Gamma \vdash \text{ref}(v_1, v_2) \Leftarrow (F(\text{Ref}[M(X)] A)) \vdash (M(X); \emptyset)$  
chk-ref

$\exists v_1' \text{ such that } \Gamma \vdash v_1' : \text{Nm}[X] \text{ and } |v_1| = v_1'$  
By Thm. C.1

$\exists v_2' \text{ such that } \Gamma \vdash v_2' : A \text{ and } |v_2| = v_2'$  
By Thm. C.1

$\Gamma \vdash \text{ref}(v_1', v_2) : (F(\text{Ref}[M(X)] A)) \vdash (M(X); \emptyset)$  
By rule ref

$|\text{ref}(v_1, v_2)| = |\text{ref}(v_1', v_2)|$  
By definition of $|\cdot|$. $|v_1| = v_1'; |v_2| = v_2'$

$\therefore \Gamma \vdash e' : (F(\text{Ref}[M(X)] A)) \vdash (M(X); \emptyset)$  
where $e' = \text{ref}(v_1', v_2)$ and $e = \text{ref}(v_1, v_2)$

**Case**  
$\Gamma \vdash \nu \Rightarrow (\text{Nm} \overset{\text{Nm}}{\vdash} \text{Nm}) [N']$  
$\Gamma \vdash \text{scope}(v, e_1) \Leftarrow C \Rightarrow \langle W; R \rangle$

$\Gamma \vdash \text{scope}(v, e_1) \Leftarrow C \Rightarrow \langle W; R \rangle$  
chk-scope

$\exists v' \text{ such that } \Gamma \vdash v' : \text{Nm} \overset{\text{Nm}}{\vdash} \text{Nm} \text{ and } |N'| = v'$  
By inductive hypothesis

$\exists e_1' \text{ such that } \Gamma \vdash e_1' : C \Rightarrow \langle W; R \rangle \text{ and } |e_1| = e_1'$  
By inductive hypothesis

$\Gamma \vdash \text{scope}(v', e_1') : C \Rightarrow \langle W; R \rangle$  
By rule scope

$|\text{scope}(v, e_1)| = |\text{scope}(v, e_1)|$  
By definition of $|\cdot|$. $|v| = v'$; $|e_1| = e_1$

$\therefore \Gamma \vdash e' : C \Rightarrow \langle W; R \rangle$  
where $e' = \text{scope}(v', e_1')$ and $e = \text{scope}(v, e_1)$

**Case**  
$\Gamma, a : \gamma \vdash M \ t \Leftarrow E$

$\Gamma \vdash M \ t \Leftarrow (\forall a : \gamma. E)$  
chk-AllIndexIntro

$\exists t' \text{ such that } \Gamma, a : \gamma \vdash M \ t' : E \text{ and } |t| = t'$  
By inductive hypothesis

$\Gamma \vdash M \ t' : (\forall a : \gamma. E)$  
By rule AllIndexIntro

$\therefore \Gamma \vdash e' : (\forall a : \gamma. E)$  
where $e' = t'$ and $e = t$

**Case**  
$\Gamma, a : \gamma \vdash M \ t \Leftarrow E$

$\Gamma \vdash M \ t' \Leftarrow (\forall a : K. E)$  
chk-AllIntro

$\exists t' \text{ such that } \Gamma, a : \gamma \vdash M \ t' : E \text{ and } |t| = t'$  
By inductive hypothesis

$\Gamma \vdash M \ t' : (\forall a : K. E)$  
By rule AllIntro

$\therefore \Gamma \vdash e' : (\forall a : K. E)$  
where $e' = t'$ and $e = t$

**Case**  
$\Gamma \vdash M \ e \Rightarrow E_1$  
$E_1 = E_2$

$\Gamma \vdash M \ e \Leftarrow E_2$  
chk-conv
\[ \exists e' \text{ such that } \Gamma \vdash^M e' : E_1 \text{ and } |e| = e' \] By inductive hypothesis

\[
\begin{align*}
E_1 &= E_2 \\
\Gamma \vdash^M e' : E_2 &\text{ Given} \\
\Gamma \vdash^M e' : E_2 &\text{ Since } E_1 = E_2
\end{align*}
\]

\[ \square \]

**Theorem C.4 (Completeness of Bidirectional Computation Typing).**

If \( \Gamma \vdash^M e : E \), then there exists computations \( e', e'' \) such that

1. \( \Gamma \vdash^M e' \Rightarrow E \) and \( |e'| = e \)
2. \( \Gamma \vdash^M e'' \Leftrightarrow E \) and \( |e''| = e \)

**Proof.** By induction on the derivation of \( \Gamma \vdash^M e : E \).

**Case** \( \Gamma \vdash^M e : (C \triangleright e_1) \)

\[ \vdash^M e : (C \triangleright e_2) \] eff-subsume

\[ \exists e' \text{ such that } \Gamma \vdash^M e' \Rightarrow (C \triangleright e_1) \text{ and } e = |e'| \] By inductive hypothesis

\[ e_1 \leq e_2 \] Given

\[ \Gamma \vdash^M e' \Leftrightarrow (C \triangleright e_2) \] By chk-eff-subsume

\[ \Gamma \vdash^M (e' : (C \triangleright e_2)) \Rightarrow (C \triangleright e_2) \] By syn-anno

\[ |(e' : (C \triangleright e_2))| = |e'| = e \] By definition of \( |-| \); Since \( |e'| = e \)

\[ \exists e'' \text{ such that } \Gamma \vdash^M e'' \Rightarrow (C \triangleright e_2) \text{ and } |e''| = e \] where \( e'' = (e' : (C \triangleright e_2)) \)

**Case** \( \Gamma \vdash v : (A_1 \times A_2) \)

\[ \Gamma, x_1 : A_1, x_2 : A_2 \vdash^M e_1 : E \] split

\[ \Gamma \vdash^M \text{split}(v, x_1, x_2, e_1) : E \] split

\[ \exists e'_1 \text{ such that } \Gamma, x_1 : A_1, x_2 : A_2 \vdash e'_1 \Leftarrow E \text{ and } e_1 = |e'_1| \] By inductive hypothesis

\[ \exists v' \text{ such that } \Gamma \vdash v' \Rightarrow (A_1 \times A_2) \text{ and } v_1 = |v'| \] By Thm. C.2

\[ \Gamma \vdash^M \text{split}(v', x_1, x_2, e'_1) \Leftarrow E \] By chk-split

\[ \Gamma \vdash^M (\text{split}(v', x_1, x_2, e'_1) : E) \Rightarrow E \] By syn-anno

\[ |(\text{split}(v', x_1, x_2, e'_1) : E)| = |\text{split}(v', x_1, x_2, e'_1)| \] By definition of \( |-| \)

\[ |\text{split}(v', x_1, x_2, e'_1)| |\text{split}(v'_1, x_1, x_2, e'_1)| = |\text{split}(v'_1, x_1, x_2, e'_1)| \] By definition of \( |-| \)

\[ \text{split}(v'_1, x_1, x_2, e'_1) = \text{split}(v', x_1, x_2, e_1) \] Since \( |v'_1| = v_1, |e'_1| = e_1 \)

\[ \Gamma \vdash e' \Rightarrow E \text{ and } |e'| = e \] where \( e' = \text{split}(v', x_1, x_2, e'_1) \)

\[ \text{and } e = \text{split}(v, x_1, x_2, e_1) \]

\[ \Gamma \vdash e'' \Leftarrow E \text{ and } |e''| = e \] where \( e'' = (\text{split}(v', x_1, x_2, e'_1) : E) \)

\[ \text{and } e = \text{split}(v, x_1, x_2, e_1) \]

**Case** \( \Gamma \vdash v : (A_1 + A_2) \)

\[ \Gamma \vdash^M \text{case}(v, x_1, e_1, x_2, e_2) : E \] case
\[ \exists v' \text{ such that } \Gamma \vdash v' \Rightarrow (A_1 + A_2) \text{ and } |v'| = v \]

- By Thm. C.2

- By inductive hypothesis

\[ \exists e'' \text{ such that } \Gamma, x_1 : A_1 \vdash^M e'' \Leftarrow E \text{ and } |e''| = e_1 \]

- By inductive hypothesis

\[ \exists e'' \text{ such that } \Gamma, x_2 : A_2 \vdash^M e'' \Leftarrow E \text{ and } |e''| = e_2 \]

- By rule chk-case

\[ \Gamma \vdash^M \text{case}(v', x_1.e''', x_2.e''') \Leftarrow E \]

- By rule chk-conv

\[ \Gamma \vdash^M (\text{case}(v', x_1.e''', x_2.e''') : E) \Rightarrow E \]

- By definition of |—|

\[ \Gamma \vdash^M \exists e' \Leftarrow E \text{ and } |e'| = e \]

- Since $|v'| = v, |e'''| = e_1, |e''| = e_2$

\[ \Gamma \vdash^M \exists e'' \Leftarrow E \text{ and } |e''| = e \]

- where $e' = (\text{case}(v', x_1.e''', x_2.e''')) : E$

\[ \Gamma \vdash^M \exists e'' \Leftarrow E \text{ and } |e''| = e \]

- and $e = \text{case}(v, x_1.e_1, x_2.e_2)$

### Case

\[ \Gamma \vdash v : A \]

- Let $E = ((F \ A) \triangleright (\emptyset; \emptyset))$

\[ \exists v'' \text{ such that } \Gamma \vdash v'' \Leftarrow A \text{ and } |v''| = v \]

- By Thm. C.2

\[ \Gamma \vdash^M \text{ret}(v'') \Leftarrow E \]

- By rule chk-ret

\[ \Gamma \vdash^M (\text{ret}(v'')) : E \Rightarrow E \]

- By syn-anno

\[ |\text{ret}(v'') : E| = |\text{ret}(v'')| \]

- By definition of |—|

\[ |\text{ret}(v'')| = |\text{ret}(v'')| = |\text{ret}(v)| \]

- By definition of |—|; Since $|v''| = v$

\[ \Gamma \vdash^M e' \Rightarrow E \text{ and } |e'| = e \]

- where $e' = (\text{ret}(v'') : E)$ and $e = \text{ret}(v)$

\[ \Gamma \vdash^M e'' \Leftarrow E \text{ and } |e''| = e \]

- where $e'' = \text{ret}(v'')$ and $e = \text{ret}(v)$

### Case

\[ \Gamma \vdash e_1 : (F \ A) \triangleright e_1 \quad \Gamma, x : A \vdash^M e_2 : (C \triangleright e_2) \]

- Let $E = (C \triangleright (e_1 \text{ then } e_2))$

\[ \exists e'_1 \text{ such that } \Gamma \vdash^M e'_1 \Rightarrow (F \ A) \triangleright e_1 \text{ and } |e'_1| = e_1 \]

- By inductive hypothesis

\[ \exists e'_2 \text{ such that } \Gamma, x : A \vdash^M e'_2 \Leftarrow (C \triangleright e_2) \text{ and } |e'_2| = e_2 \]

- By inductive hypothesis

\[ \Gamma \vdash^M \text{let}(e'_1, x.e'_2) \Leftarrow E \]

- By rule chk-let

\[ \Gamma \vdash^M (\text{let}(e'_1, x.e'_2) : E) \Rightarrow E \]

- By rule chk-conv

\[ |\text{let}(e'_1, x.e'_2) : E| = |\text{let}(e'_1, x.e'_2)| \]

- By definition of |—|

\[ |\text{let}(e'_1, x.e'_2)| = |\text{let}(e'_1, x.e'_2)| = |\text{let}(e_1, x.e_2)| \]

- By definition of |—|; Since $|e'_1| = e_1, |e'_2| = e_2$

\[ \Gamma \vdash^M e' \Rightarrow E \text{ and } |e'| = e \]

- where $e' = (\text{let}(e'_1, x.e'_2) : E)$ and $e = \text{let}(e_1, x.e_2)$

\[ \Gamma \vdash^M e'' \Leftarrow E \text{ and } |e''| = e \]

- where $e'' = \text{let}(e'_1, x.e'_2)$ and $e = \text{let}(e_1, x.e_2)$

### Case

\[ \Gamma, x : A \vdash^M e_1 : E_1 \]

- \[ \Gamma \vdash^M (\lambda x. e_1) : ((A \rightarrow E_1) \triangleright (\emptyset; \emptyset)) \]

, Vol. 1, No. 1, Article 1. Publication date: January 2016.
Let $E = ((A \rightarrow E_1) \triangleright (\emptyset; \emptyset))$ 

Assumption

$\exists e''_1$ such that $\Gamma, x : A \vdash^M e''_1 \leftarrow E$ and $|e''_1| = e_1$

By inductive hypothesis

$\Gamma \vdash^M (\lambda x. e''_1) \leftarrow E$

By rule chk-lam

$\Gamma \vdash^M ((\lambda x. e''_1) : E) \triangleright E$

By syn-anno

$(\lambda x. e''_1) = (\lambda x. e''_1) = (\lambda x. e_1)$

By definition of $\triangleright$; Since $|e''_1| = e_1$

$\Gamma \vdash^M e' \triangleright E$ and $|e'| = e$

where $e' = ((\lambda x. e''_1) : E)$ and $e = (\lambda x. e_1)$

$\Gamma \vdash^M e'' \leftarrow E$ and $|e''| = e$

where $e'' = (\lambda x. e''_1)$ and $e = (\lambda x. e_1)$

**Case**

$\Gamma \vdash^M e_1 : ((A \rightarrow E) \triangleright e_1)$

$\Gamma \vdash v : A$

$\Gamma \vdash^M (e_1 v) : (E after e_1)$

$\exists e'_1$ such that $\Gamma \vdash^M e'_1 \triangleright (A \rightarrow E) \triangleright e_1$ and $|e'_1| = e_1$

By inductive hypothesis

$\exists v''$ such that $\Gamma \vdash v'' \leftarrow A$ and $|v''| = v$

By Thm. C.2

$\Gamma \vdash^M (e'_1 v'') \triangleright (E after e_1)$

By rule syn-app

$\Gamma \vdash^M (e'_1 v'') \leftarrow E$

By rule chk-conv

$(e'_1 v'') = (e_1 v)$

By the definition of $\triangleright$; Since $|e'_1| = e_1, |v''| = v$

$\Gamma \vdash^M e' \triangleright E$ 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**

$\Gamma \vdash v : \text{Nm}[X]$

$\Gamma \vdash^M e_1 : E$

$\Gamma \vdash^M \text{thnk}(v, e_1) : (F (\text{Thk}[M(X)] E)) \triangleright (\text{M}(X); \emptyset)$

thunk

Let $E = (F (\text{Thk}[M(X)] E)) \triangleright (\text{M}(X); \emptyset)$

Assumption

$\exists v''$ such that $\Gamma \vdash v'' \leftarrow \text{Nm}[X]$ and $|v''| = v$

By Thm. C.2

$\exists e''_1$ such that $\Gamma \vdash^M e''_1 \leftarrow E$ and $|e''_1| = e_1$

By inductive hypothesis

$\Gamma \vdash^M \text{thnk}(v'', e''_1) \leftarrow E$

By rule chk-thunk

$\Gamma \vdash^M (\text{thnk}(v'', e''_1) : E) \triangleright E$

By rule syn-anno

$|(\text{thnk}(v'', e''_1) : E)| = |\text{thnk}(v'', e''_1)|$

By definition of $\triangleright$

$|\text{thnk}(v'', e''_1)| = \text{thnk}(|v''|, |e''_1|) = \text{thnk}(v, e_1)$

By definition of $\triangleright$; Since $|v''| = v, |e''_1| = e_1$

$\Gamma \vdash^M e' \triangleright E$ and $|e'| = e$

where $e' = (\text{thnk}(v'', e''_1) : E)$ and $e = \text{thnk}(v, e_1)$

$\Gamma \vdash^M e'' \leftarrow E$ and $|e''| = e$

where $e'' = \text{thnk}(v'', e''_1)$ and $e = \text{thnk}(v, e_1)$

**Case**

$\Gamma \vdash v : \text{Thk}[X]$ (C $\triangleright e$)

$\Gamma \vdash^M \text{force}(v) : (C \triangleright (\emptyset; X) then e))$

force

Let $E = (C \triangleright (\emptyset; X) then e))$

Assumption

$\exists v'$ such that $\Gamma \vdash v' \triangleright \text{Thk}[X]$ (C $\triangleright e$) and $|v'| = v$

By Thm. C.2

$\Gamma \vdash^M \text{force}(v') \leftarrow E$

By rule chk-conv

$\Gamma \vdash^M \text{force}(v') \triangleright E$

By syn-force

$|\text{force}(v')| = \text{force}(|v'|) = \text{force}(v)$

By definition of $\triangleright$; Since $|v'| = v$

$\Gamma \vdash^M e' \triangleright E$ and $|e'| = e$

where $e' = \text{force}(v')$ and $e = \text{force}(v)$

$\Gamma \vdash^M e'' \leftarrow E$ and $|e''| = e$

where $e'' = \text{force}(v')$ and $e = \text{force}(v)$

\[1:44\] Matthew A. Hammer, Joshua Dunfield, Dimitrios J. Economou, and Monal Narasimhamurthy
Case

\[ \Gamma \vdash \nu_1 : \text{Nm}[X] \quad \Gamma \vdash \nu_2 : A \]

\[ \Gamma \vdash^M \text{ref}(\nu_1, \nu_2) : (F (\text{Ref}[M[X]] A)) \to (M[X]; \emptyset) \]

Let \( E = (F (\text{Ref}[M[X]] A)) \to (M[X]; \emptyset) \)

Assumption

\[ \exists \nu''_1 \text{ such that } \Gamma \vdash \nu''_1 \mapsto \text{Nm}[X] \text{ and } \nu''_1 = \nu_1 \]

By Thm. C.2

\[ \exists \nu''_2 \text{ such that } \Gamma \vdash \nu''_2 \mapsto A \text{ and } \nu''_2 = \nu_2 \]

By Thm. C.2

\[ \Gamma \vdash^M \text{ref}(\nu''_1, \nu''_2) \mapsto E \]

By rule \text{chk-ref}

\[ \Gamma \vdash^M (\text{ref}(\nu''_1, \nu''_2)) : E \to E \]

By rule \text{syn-anno}

\[ |\text{ref}(\nu''_1, \nu''_2) : E| = |\text{ref}(\nu''_1, \nu''_2)| \]

By definition of \( |-| \)

\[ |\text{ref}(\nu''_1, \nu''_2)| = \text{ref}(\nu''_1, \nu''_2) = \text{ref}(\nu_1, \nu_2) \]

By definition of \( |-| \); Since \( \nu''_1 = \nu_1, \nu''_2 = \nu_2 \)

\[ \Gamma \vdash^M e' \Rightarrow E \text{ and } |e'| = e \]

where \( e' = (\text{ref}(\nu''_1, \nu''_2) : E) \) and \( e = \text{ref}(\nu_1, \nu_2) \)

\[ \Gamma \vdash^M e'' \Leftarrow E \text{ and } |e''| = e \]

where \( e'' = \text{ref}(\nu''_1, \nu''_2) \) and \( e = \text{ref}(\nu_1, \nu_2) \)

Case

\[ \Gamma \vdash \nu : \text{Ref}[X] A \]

\[ \Gamma \vdash^M \text{get}(\nu) : (F A) \to (\emptyset; X) \]

get

\[ \exists \nu' \text{ such that } \Gamma \vdash \nu' \Rightarrow \text{Ref}[X] A \text{ and } |\nu'| = \nu \]

By Thm. C.2

\[ \Gamma \vdash^M \text{get}(\nu') \Rightarrow (F A) \to (\emptyset; X) \]

By rule \text{syn-get}

\[ \Gamma \vdash^M \text{get}(\nu') \Leftarrow (F A) \to (\emptyset; X) \]

By rule \text{chk-conv}

\[ |\text{get}(\nu')| = |\text{get}(\nu')| = |\text{get}(\nu)| \]

By definition of \( |-| \); Since \( |\nu'| = \nu \)

\[ \Gamma \vdash^N e' \Rightarrow (F A) \to (\emptyset; X) \text{ and } |e'| = e \]

where \( e' = \text{get}(\nu') \) and \( e = \text{get}(\nu) \)

\[ \Gamma \vdash^N e'' \Leftarrow (F A) \to (\emptyset; X) \text{ and } |e''| = e \]

where \( e'' = \text{get}(\nu') \) and \( e = \text{get}(\nu) \)

Case

\[ \Gamma \vdash \nu_M : (\text{Nm} \cong \text{Nm})[M] \]

\[ \Gamma \vdash \nu : \text{Nm}[i] \]

\[ \Gamma \vdash^N (\nu_M, \nu) : F (\text{Nm}[M[i]]) \to (\emptyset; \emptyset) \]

name-app

\[ \exists \nu'_M \text{ such that } \Gamma \vdash \nu'_M \Rightarrow (\text{Nm} \cong \text{Nm})[M] \text{ and } |\nu'_M| = \nu_M \]

By Thm. C.2

\[ \exists \nu' \text{ such that } \Gamma \vdash \nu' \Rightarrow \text{Nm}[i] \text{ and } |\nu'| = \nu \]

By Thm. C.2

\[ \Gamma \vdash^N (\nu'_M, \nu') \Rightarrow F (\text{Nm}[M[i]]) \to (\emptyset; \emptyset) \]

By rule \text{syn-name-app}

\[ \Gamma \vdash^N (\nu'_M, \nu') \Leftarrow F (\text{Nm}[M[i]]) \to (\emptyset; \emptyset) \]

By rule \text{chk-conv}

\[ |(\nu'_M, \nu')| = |\nu'_M| \cdot |\nu'| = (\nu_M, \nu) \]

By definition of \( |-| \); Since \( |\nu'_M| = \nu_M, |\nu'| = \nu \)

\[ \Gamma \vdash^N e' \Rightarrow F (\text{Nm}[M[i]]) \to (\emptyset; \emptyset) \text{ and } |e'| = e \]

where \( e' = (\nu'_M, \nu') \) and \( e = (\nu_M, \nu) \)

\[ \Gamma \vdash^N e'' \Leftarrow F (\text{Nm}[M[i]]) \to (\emptyset; \emptyset) \text{ and } |e''| = e \]

where \( e'' = (\nu'_M, \nu') \) and \( e = (\nu_M, \nu) \)

Case

\[ \Gamma \vdash \nu : (\text{Nm} \cong \text{Nm})[N'] \quad \Gamma \vdash^N \nu_{e_1} : C \to (W; R) \]

\[ \Gamma \vdash^N \text{scope} (\nu, e_1) : C \to (W; R) \]

scope
Let \( E = C \triangleright (W; R) \)

Assumption

\[ \exists v'' \text{ such that }, \Gamma \vdash v'' \Rightarrow (\text{Nm} \equiv \text{Nm})[N'] \text{ and } |v''| = v \]

By Thm. 2

\[ \exists e_1' \text{ such that }, \Gamma \vdash \text{scope}(v'', e_1') \Leftarrow E \text{ and } |e_1'| = e_1 \]

By inductive hypothesis

\[ \Gamma \vdash \text{scope}(v'', e_1') : E \]

By rule chk-scope

\[ |\text{scope}(v'', e_1')| = |\text{scope}(v'', e_1')| \]

By definition of | |-

\[ \Gamma \vdash M e' \Rightarrow E \text{ and } |e'| = e \quad \text{where } e' = \text{scope}(v'', e_1') \text{ and } e = \text{scope}(v, e_1) \]

\[ \Gamma \vdash M e'' \Leftarrow E \text{ and } |e''| = e \quad \text{where } e'' = \text{scope}(v', e_1') \text{ and } e = \text{scope}(v, e_1) \]

Case

\[ \Gamma, a : \gamma \vdash M t : E \quad \text{AllIndexIntro} \]

\[ \Gamma, a : \gamma \vdash M t'' : \langle \forall a : \gamma, E \rangle \]

By inductive hypothesis

\[ \Gamma, a : \gamma \vdash M t'' : \langle \forall a : \gamma, E \rangle \Rightarrow (\forall a : \gamma, E) \]

By rule syn-anno

\[ |(t'': \langle \forall a : \gamma, E \rangle)| = |t''| = t \]

By definition of | |-

\[ \Gamma \vdash M e' \Rightarrow (\forall a : \gamma, E) \text{ and } |e'| = e \quad \text{where } e' = (t'': (\forall a : \gamma, E)) \text{ and } e = t \]

\[ \Gamma \vdash M e'' \Leftarrow (\forall a : \gamma, E) \text{ and } |e''| = e \quad \text{where } e'' = t'' \text{ and } e = t \]

Case

\[ \Gamma \vdash M e : \langle \forall a : \gamma, E \rangle \quad \Gamma, i : \gamma \quad \text{AllIndexElim} \]

\[ \exists e' \text{ such that } \Gamma, i \vdash [i/a]E \text{ and } |e'| = e \]

By inductive hypothesis

\[ \Gamma, i : \gamma \]

Given

\[ \Gamma, i \vdash M e' \Rightarrow [i/a]E \]

By rule syn-AllIndexElim

\[ \Gamma, i \vdash M e' \Leftarrow [i/a]E \]

By rule chk-conv

\[ \Gamma, i \vdash M e'[i] \Rightarrow [i/a]E \text{ and } |e'[i]| = e \]

\[ \Gamma, i \vdash M e'[i] \Leftarrow [i/a]E \text{ and } |e'[i]| = e \]

Case

\[ \Gamma, a : \gamma \vdash M t : E \quad \text{AllIntro} \]

\[ \Gamma, i \vdash M t : \langle \forall a : \gamma, K, E \rangle \]

Similar to the AllIntro case.

Case

\[ \Gamma \vdash M e : \langle \forall a : \gamma, K, E \rangle \quad \Gamma, A : K \quad \text{AllElim} \]

\[ \Gamma \vdash M e : [A/\alpha]E \]

Similar to the AllIndexElim case.

\[ \square \]

D \quad \text{NAME TERM LANGUAGE}

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

- Syntax for \textit{names}, \textit{name terms} and \textit{sorts}.
• Name term sorting: A judgment that assigns sorts to name terms.
• Big-step evaluation for name terms: A judgment that assigns name term values to name terms.
• Semantic definition of equivalent and disjoint name terms.
• 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.

D.1 Name term equivalence and apartness
For instance, the following two functions have the same sort, and are apart:

\[
\lambda a. (a, a) \quad : \quad \text{Nm} \xrightarrow{=} \text{Nm} \times \text{Nm}
\]
\[
\lambda b. (b, \langle \text{leaf}, b \rangle) \quad : \quad \text{Nm} \xrightarrow{=} \text{Nm} \times \text{Nm}
\]

These functions have a common sort \(\text{Nm} \xrightarrow{=} \text{Nm} \times \text{Nm}\), which says that they map a single name (of sort \(\text{Nm}\)) to a pair of names (of sort \(\text{Nm} \times \text{Nm}\)). These functions are apart, and this apartness ultimately follows from reasoning about binary composition of names: any name \(n\) is distinct from the binary composition of the name constant \(\text{leaf}\) with name \(n\).

These two name functions also have the same sort, but are not apart:

\[
\lambda a. (a, \langle a, \text{leaf} \rangle) \quad : \quad \text{Nm} \xrightarrow{=} \text{Nm} \times \text{Nm}
\]
\[
\lambda b. (a, \langle \text{leaf}, b \rangle) \quad : \quad \text{Nm} \xrightarrow{=} \text{Nm} \times \text{Nm}
\]

In particular, when \(b = a = \text{leaf}\), they will produce the same pair, namely \(\langle \text{leaf}, \langle \text{leaf}, \text{leaf} \rangle \rangle\).

Notice that these two functions are obviously not equivalent either: Given any name \(n\) such that \(n \neq \text{leaf}\), these two functions will produce two distinct names. This follows from another reasoning principle for name composition: binary composition with \(\text{leaf}\) in the first function is in a distinct order from that in the second function, precisely when \(n \neq \text{leaf}\).

Finally, these two functions are equivalent: Given equivalent arguments, they will always produce equivalent results:

\[
\lambda a.\lambda b. a\ b \quad : \quad (\text{Nm} \xrightarrow{=} \text{Nm}) \xrightarrow{=} \text{Nm} \xrightarrow{=} \text{Nm}
\]
\[
\lambda a_1.\lambda b_1. (\lambda a_2.\lambda b_2. a_2\ b_2)\ a_1\ b_1 \quad : \quad (\text{Nm} \xrightarrow{=} \text{Nm}) \xrightarrow{=} \text{Nm} \xrightarrow{=} \text{Nm}
\]

To see this equivalence more clearly, imagine reducing the inner \(\beta\)-reduces by substituting \(a_1\) and \(b_1\) for \(a_2\) and \(b_2\), respectively. After this reduction, the two functions are \(\alpha\)-equivalent.

Below, we give a semantic definition of both equivalence and disjointness, in terms of the sorting and operational definitions given above.

These definitions are clear, but not immediately practical: for function sorts, they universally quantify over all possible arguments for the function’s argument. Since these arguments can include names, which consist of arbitrary finite binary trees, as well as other functions, there is not an obvious finite set of arguments to test while still being sound with respect to these definitions. For a practical implementation of Typed Adapton, we seek definitions of equivalence and disjointness that admit decision procedures, and are sound (and, ideally, complete) with respect to these semantic definitions. For this purpose, we give decidable logical rules that induct over the syntax of the two terms.

D.2 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 $\Gamma$ that relate two variables, either asserting that they are equivalent, or disjoint:

\[
\begin{align*}
\text{Substitutions} & \quad \sigma ::= \cdot | \sigma, (a \mapsto N) & \quad (\cdot), 1 = \cdot \\
\text{Relational sorting contexts} & \quad \Gamma ::= \cdot & \quad (\cdot), 2 = \cdot \\
\text{(Hypothetical variable equivalence)} & \quad | \Gamma, (a \equiv b : \gamma) & \quad (\cdot), 1, a : \gamma = (\Gamma), 1, a : \gamma \\
\text{(Hypothetical variable apartness)} & \quad | \Gamma, (a \perp b : \gamma) & \quad (\cdot), 1, b : \gamma = (\Gamma), 1, a : \gamma \\
\end{align*}
\]

\textbf{Definition D.1 (Closing substitutions).}
We define closing substitution pairs related by equivalence and disjointness assumptions in a context $\Gamma$. 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 $(\sigma_1(x) = N \text{ and } \sigma_2(y) = M)$ and $\vdash N \equiv M : \gamma$
- $\vdash \sigma_1 \perp \sigma_2 : \Gamma$ means that $(x \perp y : \gamma) \in \Gamma$ implies $(\sigma_1(x) = N \text{ and } \sigma_2(y) = M)$ and $\vdash N \perp M : \gamma$
- $\vdash \sigma_1 \sim \sigma_2 : \Gamma$ means that $\vdash \sigma_1 \equiv \sigma_2 : \Gamma$ and $\vdash \sigma_1 \perp \sigma_2 : \Gamma$

\textbf{Definition D.2 (Semantic equivalence).} We define $\Gamma \vdash M_1 \equiv M_2 : \gamma$ as follows:

$$\begin{align*}
(\cdot), 1 \vdash M_1 : \gamma \quad \text{and} \quad (\cdot), 2 \vdash M_2 : \gamma \quad \text{and},
\end{align*}$$

for all $\sigma_1, \sigma_2$ 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 $V_2$:

<table>
<thead>
<tr>
<th>Sort ($\gamma$)</th>
<th>Equivalence property for values $V_1$ and $V_2$ of sort $\gamma$ (written $\vdash V_1 \equiv V_2 : \gamma$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bot$</td>
<td>Always</td>
</tr>
<tr>
<td>$\text{N}m$</td>
<td>When $V_1 = n_1$ and $V_2 = n_2$ and $n_1 = n_2$ (identical binary trees)</td>
</tr>
<tr>
<td>$\gamma_1 \ast \gamma_2$</td>
<td>When $V_1 = (V_{11}, V_{12})$ and $V_1 = (V_{21}, V_{22})$ and $\vdash V_{11} \equiv V_{21} : \gamma_1$ and $\vdash V_{21} \equiv V_{22} : \gamma_2$</td>
</tr>
<tr>
<td>$\gamma_1 \Rightarrow \gamma_2$</td>
<td>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 $\vdash W_1 \equiv W_2 : \gamma_2$</td>
</tr>
</tbody>
</table>

\textbf{Definition D.3 (Semantic apartness).} We define $\Gamma \vdash M_1 \perp M_2 : \gamma$ as follows:

$$\begin{align*}
(\cdot), 1 \vdash M_1 : \gamma \quad \text{and} \quad (\cdot), 2 \vdash M_2 : \gamma \quad \text{and},
\end{align*}$$

for all $\sigma_1, \sigma_2$ 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 $V_2$:

<table>
<thead>
<tr>
<th>Sort ($\gamma$)</th>
<th>Apartness property for values $V_1$ and $V_2$ of sort $\gamma$ (written $\vdash V_1 \perp V_2 : \gamma$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bot$</td>
<td>Always</td>
</tr>
<tr>
<td>$\text{N}m$</td>
<td>When $V_1 = n_1$ and $V_2 = n_2$ and $n_1 \neq n_2$ (distinct binary trees)</td>
</tr>
<tr>
<td>$\gamma_1 \ast \gamma_2$</td>
<td>When $V_1 = (V_{11}, V_{12})$ and $V_1 = (V_{21}, V_{22})$ and $\vdash V_{11} \perp V_{21} : \gamma_1$ and $\vdash V_{21} \perp V_{22} : \gamma_2$</td>
</tr>
<tr>
<td>$\gamma_1 \Rightarrow \gamma_2$</td>
<td>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 $\vdash W_1 \perp W_2 : \gamma_2$</td>
</tr>
</tbody>
</table>

\subsection*{D.3 Metatheory of name term language}

Some lemmas in this section are missing complete proofs and should be considered conjectures (Lemma D.1 (Projections of syntactic equivalence)—Lemma D.10 (Reflexivity of name term evaluation)).

\textbf{Lemma D.1 (Projections of syntactic equivalence).}

If $\Gamma \vdash M_1 \equiv M_2 : \gamma$, then $(\cdot), 1 \vdash M_1 : \gamma$ and $(\cdot), 2 \vdash M_2 : \gamma$. 

\textcopyright 2016 ACM. All rights reserved.
The name terms $M$ and $N$ are equivalent at sort $\gamma$

\[
\Gamma \vdash M \equiv N : \gamma
\]

The name terms $M$ and $N$ are apart at sort $\gamma$

\[
\Gamma \vdash M \nmid N : \gamma
\]

**Lemma D.2 (Compatibility of substitution with name term structure).**

1. We have $[\sigma](M_1, M_2) = ([\sigma]M_1, [\sigma]M_2)$.
2. We have $[\sigma]\langle M_1, M_2 \rangle = \langle [\sigma]M_1, [\sigma]M_2 \rangle$.
3. We have $[\sigma](MN) = ([\sigma]M)([\sigma]N)$.

**Lemma D.3 (Determinism of evaluation up to substitution).**

If $\Gamma, 1 \vdash M : \gamma$, $\Gamma, 2 \vdash M : \gamma$, $\models \sigma_1 \equiv \sigma_2 : \Gamma$, $[\sigma_1]M \Downarrow V_1$, and $[\sigma_2]M \Downarrow V_2$, then there exists a $V$ such that $V_1 = [\sigma_1]V$ and $V_2 = [\sigma_2]V$. 

,, Vol. 1, No. 1, Article 1. Publication date: January 2016.
LEMMA D.4 (Reflexivity of semantic equivalence).

(1) If the type of a closed term is in the same equivalence class as the term itself, then the term is in the same equivalence class as itself.

(2) If the type of a closed term is in the same equivalence class as the term itself, then the term is in the same equivalence class as itself.

LEMMA D.5 (Type safety).

If the type of a closed term is in the same equivalence class as the term itself, then the term is in the same equivalence class as itself.

LEMMA D.6 (Symmetry of semantic equivalence).

(1) If the type of a closed term is in the same equivalence class as the term itself, then the term is in the same equivalence class as itself.

(2) If the type of a closed term is in the same equivalence class as the term itself, then the term is in the same equivalence class as itself.

LEMMA D.7 (Evaluation respects semantic equivalence).

If the type of a closed term is in the same equivalence class as the term itself, then there exists another term such that the type of the term is in the same equivalence class as the term itself.

LEMMA D.8 (Closing substitutions respect syntactic equivalence).

If the type of a closed term is in the same equivalence class as the term itself, then the term is in the same equivalence class as itself.

LEMMA D.9 (Closing substitutions respect composition).

If the type of a closed term is in the same equivalence class as the term itself, then the term is in the same equivalence class as itself.

LEMMA D.10 (Reflexivity of name term evaluation).

If the type of a closed term is in the same equivalence class as the term itself, then the term is in the same equivalence class as itself.

LEMMA D.11 (Transitivity of value equivalence).

If the type of a closed term is in the same equivalence class as the term itself, then the term is in the same equivalence class as itself.

Proof. By induction on the given derivation.

Case \( a = b : \gamma \in \Gamma \)

\[ \frac{a = b : \gamma \in \Gamma}{\Gamma \vdash a = b : \gamma} \text{ Eq-Var} \]

By definition of closing substitutions.

Case \( \Gamma, 1 \vdash M : \gamma \)

\[ \frac{\Gamma, 1 \vdash M : \gamma}{\Gamma \vdash M \equiv M : \gamma} \text{ Eq-Refl} \]

By Lemma D.3 (Determinism of evaluation up to substitution), Lemma D.5 (Type safety), and Lemma D.4 (Reflexivity of semantic equivalence).

Case \( \Gamma \vdash N \equiv M : \gamma \)

\[ \frac{\Gamma \vdash M \equiv N : \gamma}{\Gamma \vdash M \equiv N : \gamma} \text{ Eq-Sym} \]

By Lemma D.6 (Symmetry of semantic equivalence).
Case  \( \Gamma \vdash M_1 \equiv M_2 : \gamma \)
\( \Gamma \vdash M_2 \equiv M_3 : \gamma \)
\( \Gamma \vdash M_1 \equiv M_3 : \gamma \)  Eq-Trans

By idempotency of flipping relational contexts, Lemma D.7 (Evaluation respects semantic equivalence), inductive hypotheses on the two given subderivations, Lemma D.6 (Symmetry of semantic equivalence), and Lemma D.11 (Transitivity of value equivalence).

Case  \( \Gamma \vdash M_1 \equiv N_1 : \gamma_1 \quad \Gamma \vdash M_2 \equiv N_2 : \gamma_2 \)
\( \Gamma \vdash \langle M_1, M_2 \rangle \equiv \langle N_1, N_2 \rangle : \gamma_1 \star \gamma_2 \)  Eq-Pair

By Lemma D.2 (Compatibility of substitution with name term structure) and inductive hypothesis.

Case  \( \Gamma \vdash M_1 \equiv N_1 : \text{Nm} \quad \Gamma \vdash M_2 \equiv N_2 : \text{Nm} \)
\( \Gamma \vdash \langle M_1, M_2 \rangle \equiv \langle N_1, N_2 \rangle : \text{Nm} \)  Eq-Bin

By Lemma D.2 (Compatibility of substitution with name term structure) and inductive hypothesis.

Case  \( \Gamma, (a \equiv b : \gamma_1) \vdash M \equiv N : \gamma_2 \)  Eq-Lam
\( \Gamma \vdash \lambda a. M \equiv \lambda b. N : \gamma_1 \)  \( \mapsto \)  \( \gamma_2 \)

By Lemma D.9 (Closing substitutions respect composition) and inductive hypothesis.

Case  \( \Gamma \vdash M_1 \equiv N_1 : \gamma_1 \)  \( \mapsto \)  \( \gamma_2 \)
\( \Gamma \vdash M_2 \equiv N_2 : \gamma_1 \)
\( \Gamma \vdash M_1(M_2) \equiv N_1(N_2) : \gamma_2 \)  Eq-App

By Lemma D.2 (Compatibility of substitution with name term structure) 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  \( \Gamma \vdash M_2 \equiv M'_2 : \gamma_1 \quad \Gamma, a \equiv a : \gamma_1 \vdash M_1 \equiv M'_1 : \gamma_2 \)
\( \Gamma \vdash (\lambda a. M_1)M_2 \equiv [M'_2/a]M'_1 : \gamma_2 \)  Eq-\( \beta \)

Fix \( \vdash \sigma_1 \equiv \sigma_2 : \Gamma \). Suppose \( [\sigma_1](\lambda a. M_1)M_2 \) \( \Downarrow \) \( V_1 \) and \( [\sigma_2](\lambda a. M_1)M_2 \) \( \Downarrow \) \( V_2 \). We need to show \( \vdash V_1 \equiv V_2 : \gamma_2 \). By Lemma D.2 (Compatibility of substitution with name term structure) 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 \( [\sigma_1, a \mapsto V]M_1 \Downarrow V_1 \). Rewrite \( [\sigma_2](\lambda a. M_2)M_2 \) \( \Downarrow V_2 \) as \( [\sigma_2, a \mapsto [\sigma_2]M'_2]M'_2 \Downarrow V_2 \). By Lemma D.8 (Closing substitutions respect syntactic equivalence), \( \vdash V \equiv [\sigma_2]M'_2 : \gamma_1 \). Therefore,

\( \vdash \sigma_1, a \mapsto V \equiv \sigma_2, a \mapsto [\sigma_2]M'_2 : \Gamma, a \equiv a : \gamma_1 \)

By the inductive hypothesis on \( \Gamma, a \equiv a : \gamma_1 \vdash M_1 \equiv M'_1 : \gamma_2 \), we get \( \vdash V_1 \equiv V_2 : \gamma_2 \). \( \Box \)

Conjecture D.13 (Soundness of deductive disjointness). If \( \Gamma \vdash M_1 \perp M_2 : \gamma \) then \( \Gamma \vdash M_1 \perp M_2 : \gamma \).

Conjecture D.14 (Completeness of deductive equivalence). If \( \Gamma \vdash M_1 \equiv M_2 : \gamma \) then \( \Gamma \vdash M_1 \equiv M_2 : \gamma \).

Conjecture D.15 (Completeness of deductive disjointness). If \( \Gamma \vdash M_1 \perp M_2 : \gamma \) then \( \Gamma \vdash M_1 \perp M_2 : \gamma \).
We define closing substitution pairs related by equivalence and disjointness assumptions in a context \( \Gamma \).

**Definition E.1** (Closing substitutions for index terms).

We define closing substitution pairs related by equivalence and disjointness assumptions in a context \( \Gamma \). These definitions use and are based on the definitions below for equivalence and apartness of open terms.
Name term $M$ is a member of name set $X$, assuming $X \text{val}$

\[
\begin{align*}
\models M & \in X \\
\models M & \in (X \perp Y) \quad \text{Apart}_1 \\
\models M & \in (X \perp \neg Y) \quad \text{Apart}_2 \\
\models M & \equiv N : \text{Nm} \\
\models M & \in \{N\} \quad \text{Single} \\
\end{align*}
\]

The name of name term $M$ is not a member of name set $X$, assuming $X \text{val}$

\[
\begin{align*}
\models M & \not\in X \\
\models M & \not\in Y \quad \text{Apart} \\
\models M & \not\in (X \perp Y) \\
\models M & \not\in N : \text{Nm} \\
\models M & \not\in \{N\} \quad \text{Empty} \\
\end{align*}
\]

Fig. 23. Name term membership

- $\models \sigma_1 \equiv \sigma_2 : \Gamma$ means that $(x \equiv y : \gamma) \in \Gamma$ implies $(\sigma_1(x) = i \text{ and } \sigma_2(y) = j \text{ and } \models i \equiv j : \gamma)$
- $\models \sigma_1 \perp \sigma_2 : \Gamma$ means that $(x \equiv y : \gamma) \in \Gamma$ implies $(\sigma_1(x) = i \text{ and } \sigma_2(y) = j \text{ and } \models i \perp j : \gamma)$
- $\models \sigma_1 \sim \sigma_2 : \Gamma$ means that $(\models \sigma_1 \equiv \sigma_2 : \Gamma \text{ and } \models \sigma_1 \perp \sigma_2 : \Gamma)$

Definition E.2 (Semantic equivalence of index terms). We define $\Gamma \models i_1 \equiv i_2 : \gamma$ as follows:

\[
\begin{align*}
(\Gamma).1 & \models i_1 : \gamma \text{ and } (\Gamma).2 \models i_2 : \gamma \text{ and, for all } \sigma_1, \sigma_2 \text{ such that } \models \sigma_1 \equiv \sigma_2 : \Gamma \text{ and } [\sigma_1]_1 \downarrow i_1 \text{ and } [\sigma_2]_2 \downarrow i_2, \\
\text{we have the following about } i_1 \text{ and } i_2: \\
\end{align*}
\]

\[
\begin{array}{|c|c|}
\hline
\text{Sort } \gamma & \text{Equivalence property for index values } j_1 \text{ and } j_2 \text{ of sort } \gamma \text{ (written } \models i_1 \equiv i_2 : \gamma) \\
\hline
\text{1} & \text{Always} \\
\text{NmSet} & \text{When } (\models M \in j_1 \text{ if and only if } \models M \in j_2) \\
\gamma_1 \neq \gamma_2 & \text{When } j_1 = [j_{11}, j_{12}] \text{ and } j_2 = [j_{21}, j_{22}] \text{ and } \models j_{11} \equiv j_{21} : \gamma_1 \text{ and } \models j_{12} \equiv j_{22} : \gamma_2 \\
\gamma_1 \rightarrow_{idx} \gamma_2 & (\lambda_{\alpha_1, X_1}(Y_1) \downarrow Z_1 \text{ and } \lambda_{\alpha_2, X_2}(Y_2) \downarrow Z_2 \text{ implies } \models Z_1 \equiv Z_2 : \gamma_2) \\
\hline
\end{array}
\]

Definition E.3 (Semantic apartness of index terms). We define $\Gamma \models i_1 \perp i_2 : \gamma$ as follows:

\[
\begin{align*}
(\Gamma).1 & \models i_1 : \gamma \text{ and } (\Gamma).2 \models i_2 : \gamma \text{ and, for all } \sigma_1, \sigma_2 \text{ such that } \models \sigma_1 \sim \sigma_2 : \Gamma \text{ and } [\sigma_1]_1 \downarrow i_1 \text{ and } [\sigma_2]_2 \downarrow i_2, \\
\text{we have the following about } i_1 \text{ and } i_2: \\
\end{align*}
\]

\[
\begin{array}{|c|c|}
\hline
\text{Sort } (\gamma) & \text{Apartness property for index values } j_1 \text{ and } j_2 \text{ of sort } \gamma \text{ (written } \models i_1 \perp i_2 : \gamma) \\
\hline
\text{1} & \text{Always} \\
\text{NmSet} & \text{When } (\models M \in j_1 \text{ implies } \models M \not\in j_2) \text{ and } (\models M \not\in j_2 \text{ implies } \models M \not\in j_1) \\
\gamma_1 \neq \gamma_2 & \text{When } j_1 = [j_{11}, j_{12}] \text{ and } j_2 = [j_{21}, j_{22}] \text{ and } \models j_{11} \perp j_{21} : \gamma_1 \text{ and } \models j_{12} \perp j_{22} : \gamma_2 \\
\gamma_1 \rightarrow_{idx} \gamma_2 & (\lambda_{\alpha_1, X_1}(Y_1) \downarrow Z_1 \text{ and } \lambda_{\alpha_2, X_2}(Y_2) \downarrow Z_2 \text{ implies } \models Z_1 \perp Z_2 : \gamma_2) \\
\hline
\end{array}
\]

The next two definitions bridge the gap with the type system, in which contexts $\Gamma$ also include propositions $P$. It is defined assuming that extract($\Gamma_{\gamma}$) (defined in Figure 21) has given us some propositions $P_1, \ldots, P_n$ and a relational context $\Gamma$.

Definition E.4 (Extended semantic equivalence of index terms). We define $P_1, \ldots, P_n; \Gamma \models i \equiv j : \gamma$ to hold if and only if

\[
\mathcal{J}(P_1) \text{ and } \cdots \text{ and } \mathcal{J}(P_n) \text{ implies } \Gamma \models i \equiv j : \gamma
\]

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

Definition E.5 (Extended semantic apartness of index terms). We define $P_1, \ldots, P_n; \Gamma \models i \perp j : \gamma$ to hold if and only if
\[ J(P_1) \text{ and } \cdots \text{ and } J(P_n) \text{ implies } \Gamma \vdash i \perp j : \gamma \]

where \( J(i \合成 j : \gamma) = (\Gamma \vdash i \合成 j : \gamma) \).

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

**Lemma E.1 (Weakening of semantic equivalence and apartness).**

*If \( \text{extract}(\Gamma_T) \vdash i_1 \equiv i_2 : \gamma \) (respectively \( i_1 \perp i_2 : \gamma \)) then \( \text{extract}(\Gamma_T, \Gamma'_T) \vdash i_1 \equiv i_2 : \gamma \) (respectively \( i_1 \perp i_2 : \gamma \)).

**Proof.** By induction on \( \Gamma'_T \).

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., \( \text{extract}(\Gamma_T, \Gamma') \vdash i_1 \equiv i_2 : \gamma \).
  
  That is, \( \text{extract-assns}(\Gamma_T, \Gamma'); \text{extract-ctx}(\Gamma_T, \Gamma) \vdash i_1 \equiv i_2 : \gamma \).

  By its definition, \( \text{extract-ctx}(\Gamma_T, \Gamma', P) = \text{extract-ctx}(\Gamma_T, \Gamma', P) \).

  Therefore, we have \( \text{extract-assns}(\Gamma_T, \Gamma'); \text{extract-ctx}(\Gamma_T, \Gamma, P) \vdash i_1 \equiv i_2 : \gamma \).

  Adding an assumption before the semicolon only supplements the antecedent in Def. E.4, so

  \[ \text{extract-assns}(\Gamma_T, \Gamma', P); \text{extract-ctx}(\Gamma_T, \Gamma', P) \vdash i_1 \equiv i_2 : \gamma \]

  which was to be shown.

- If \( \Gamma'_T = (\Gamma', \alpha : \gamma) \) then by i.h.,

  \[ \text{extract}(\Gamma_T, \Gamma') \vdash i_1 \equiv i_2 : \gamma \]

  By definition of \( \text{extract-ctx}, \)

  \[ \text{extract-ctx}(\Gamma_T, \Gamma', \alpha : \gamma) = \text{extract-ctx}(\Gamma_T, \Gamma'), \alpha \equiv \alpha : \gamma \]

  By the i.h. and Def. E.2, \( \text{extract-ctx}(\Gamma_T, \Gamma') \vdash i_1 : \gamma \). We need to show that \( \text{extract-ctx}(\Gamma_T, \Gamma', \alpha : \gamma) \vdash i_1 : \gamma \), which follows by weakening on sorting. The "\( \alpha \)" part is similar.

  Since \( \alpha \) does not occur in \( i_1 \) and \( i_2 \), applying longer substitutions that include \( \alpha \) to \( i_1 \) and \( i_2 \) does not change them; thus, we get the same \( j_1 \) and \( j_2 \) as for \( \Gamma_T, \Gamma' \).

- In the remaining cases of \( Z \) for \( \Gamma'_T = (\Gamma', Z) \), neither \( \text{extract-assns} \) nor \( \text{extract-ctx} \) change, and the i.h. immediately gives the result. \( \square \)
E.2 Deductive equivalence and apartness for index terms

1. The name terms i and j are equivalent at sort γ

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a ⊑ b : γ) ∈ Γ</td>
<td>Eq-Var</td>
<td>(a ⊑ b : γ) ∈ Γ</td>
</tr>
<tr>
<td>Γ ⊢ a ⊑ b : γ</td>
<td>E-Var</td>
<td>Γ ⊢ a ⊑ b : γ</td>
</tr>
<tr>
<td>Γ ⊢ i₁ ≡ j₁ : γ₁</td>
<td>Eq-Pair</td>
<td>Γ ⊢ i₁ ≡ j₁ : γ₁</td>
</tr>
<tr>
<td>Γ ⊢ (i₁, i₂) ≡ (j₁, j₂) : γ₁ * γ₂</td>
<td>Eq-App</td>
<td>Γ ⊢ (i₁, i₂) ≡ (j₁, j₂) : γ₁ * γ₂</td>
</tr>
<tr>
<td>Γ ⊢ i₁ ⊥ j₂ : γ₂</td>
<td>D-Sym</td>
<td>Γ ⊢ i₁ ⊥ j₂ : γ₂</td>
</tr>
<tr>
<td>Γ ⊢ (i₁, i₂) ⊥ (j₁, j₂) : γ₁ * γ₂</td>
<td>D-Project₂</td>
<td>Γ ⊢ (i₁, i₂) ⊥ (j₁, j₂) : γ₁ * γ₂</td>
</tr>
<tr>
<td>(Γ), 1 ⊢ i₁ ⊥ j₁ : γ₁</td>
<td>D-Project₁</td>
<td>(Γ), 1 ⊢ i₁ ⊥ j₁ : γ₁</td>
</tr>
<tr>
<td>Γ ⊢ (i₁, i₂) ⊥ (j₁, j₂) : γ₁ * γ₂</td>
<td>D-Empty</td>
<td>Γ ⊢ (i₁, i₂) ⊥ (j₁, j₂) : γ₁ * γ₂</td>
</tr>
<tr>
<td>Γ ⊢ i₁ ⊥ j₁ : γ₁</td>
<td>D-App</td>
<td>Γ ⊢ i₁ ⊥ j₁ : γ₁</td>
</tr>
<tr>
<td>(Γ), 2 ⊢ X : NmSet</td>
<td>D-Empty</td>
<td>(Γ), 2 ⊢ X : NmSet</td>
</tr>
<tr>
<td>Γ ⊢ X ⊥ Y : NmSet</td>
<td>D-Single</td>
<td>Γ ⊢ X ⊥ Y : NmSet</td>
</tr>
</tbody>
</table>

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

2. The index terms i and j are apart at sort γ

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a ⊑ b : γ) ∈ Γ</td>
<td>Eq-Var</td>
<td>(a ⊑ b : γ) ∈ Γ</td>
</tr>
<tr>
<td>Γ ⊢ a ⊑ b : γ</td>
<td>E-Var</td>
<td>Γ ⊢ a ⊑ b : γ</td>
</tr>
<tr>
<td>Γ ⊢ i₁ ⊥ j₁ : γ₁</td>
<td>Eq-Empty</td>
<td>Γ ⊢ i₁ ⊥ j₁ : γ₁</td>
</tr>
<tr>
<td>Γ ⊢ (i₁, i₂) ⊥ (j₁, j₂) : γ₁ * γ₂</td>
<td>Eq-Single</td>
<td>Γ ⊢ (i₁, i₂) ⊥ (j₁, j₂) : γ₁ * γ₂</td>
</tr>
<tr>
<td>Γ ⊢ i₁ ⊥ j₂ : γ₂</td>
<td>D-Lam</td>
<td>Γ ⊢ i₁ ⊥ j₂ : γ₂</td>
</tr>
<tr>
<td>Γ ⊢ (λa₁, i₁) ⊥ (j₁, j₂) : γ₁ * γ₂</td>
<td>D-Lam</td>
<td>Γ ⊢ (λa₁, i₁) ⊥ (j₁, j₂) : γ₁ * γ₂</td>
</tr>
<tr>
<td>(Γ), 1 ⊢ i₂ ⊥ j₂ : γ₂</td>
<td>D-β</td>
<td>(Γ), 1 ⊢ i₂ ⊥ j₂ : γ₂</td>
</tr>
<tr>
<td>Γ ⊢ (λa₁, i₁) ⊥ j₁ : γ</td>
<td>D-β</td>
<td>Γ ⊢ (λa₁, i₁) ⊥ j₁ : γ</td>
</tr>
<tr>
<td>Γ ⊢ M ⊥ N : NmSet</td>
<td>D-Empty</td>
<td>Γ ⊢ M ⊥ N : NmSet</td>
</tr>
<tr>
<td>Γ ⊢ [M] ⊥ [N] : NmSet</td>
<td>D-Empty</td>
<td>Γ ⊢ [M] ⊥ [N] : NmSet</td>
</tr>
<tr>
<td>Γ ⊢ X ⊥ Y : NmSet</td>
<td>D-Single</td>
<td>Γ ⊢ X ⊥ Y : NmSet</td>
</tr>
<tr>
<td>Γ ⊢ M ⊥ N : NmSet</td>
<td>D-Single</td>
<td>Γ ⊢ M ⊥ N : NmSet</td>
</tr>
<tr>
<td>Γ ⊢ X ⊥ Y : NmSet</td>
<td>D-Single</td>
<td>Γ ⊢ X ⊥ Y : NmSet</td>
</tr>
</tbody>
</table>

Fig. 25. Deductive rules for showing that two index terms are apart
F STRONG NORMALIZATION PROOF FOR THE NAME TERM LANGUAGE

Definition F.1 \((R_\gamma(e))\).

(1) \(R_\gamma(e)\) if and only if \(e\) halts, and \(\gamma \neq (\gamma_1 \rightarrow \gamma_2)\)
(2) \(R_{(\gamma_1 \rightarrow \gamma_2)}(e)\) if and only if (1) \(e\) halts and (2) \(R_{\gamma_1}(e')\) implies \(R_{\gamma_2}(e e')\).

Lemma F.2 (Termination). If \(R_\gamma(e)\), then \(e\) halts.
Proof. By definition of \(R_\gamma(e)\). □

Lemma F.3 (Preservation). If \(\Gamma \vdash e : \gamma\) and \(\sigma \vdash \Gamma\), then \([\sigma]e : \gamma\).
Proof. By induction on the size of \(\sigma\). When \(\text{size}(\sigma) = 1\), the Lemma reduces to the Substitution Lemma. □

Lemma F.4 (Normalization). If \(\Gamma \vdash e : \gamma\) and \(\sigma \vdash \Gamma\), then \(R_\gamma([\sigma]e)\).
Proof. By induction on the given derivation.

Case

\[ \frac{\Gamma \vdash \cdot : 1}{[\sigma]() = ()} \text{ M-unit} \]
[\[\sigma]() = ()\] By definition of \([\sigma](-)\)
\[ () \Downarrow_M () \] By Rule teval-value
\[ [\sigma]() \Downarrow_M () \] Since \([\sigma]() = (), () \Downarrow_M ()\)
\[ \Gamma \vdash () : 1 \] Given
\[ \vdash [\sigma]() : 1 \] By Lemma F.3 (Preservation)
\[ \equiv R_{\gamma}([\sigma]()) \] Since \([\sigma]() \Downarrow_M (), \vdash [\sigma]() : 1\)

Case

\[ \frac{\Gamma \vdash \cdot : \text{Nm}}{M-\text{const}} \]
Similar to the M-unit case.

Case

\[ \frac{\cdot \in \Gamma}{\vdash a : \gamma} \text{ M-var} \]
\[ \exists \nu_1. ([\sigma]a \Downarrow_M \nu_1) \] By definition of \([\sigma](-)\)
\[ \vdash [\sigma]a : \gamma \] From Lemma F.3 (Preservation)
\[ \equiv R_{\gamma}([\sigma]a) \] Since \([\sigma]a \Downarrow_M \nu_1, \vdash [\sigma]a : \gamma\)

Case

\[ \frac{\Gamma, a : \gamma_1 \vdash M : \gamma_2}{\vdash (\lambda a. M) : (\gamma_1 \rightarrow \gamma_2)} \text{ M-abs} \]
\[ [\sigma, a \rightarrow \gamma_1] \vdash \Gamma' \] Assumption, where \(\Gamma' = \Gamma, a : \gamma_1\)
\[ \vdash ([\sigma, a \rightarrow \gamma_1](\lambda a. M)) : (\gamma_1 \rightarrow \gamma_2) \] By Lemma F.3 (Preservation)
\[ [\sigma, a \rightarrow \gamma_1](\lambda a. M) = (\lambda a. M) \] By definition of \([\sigma](-)\)
\[ \equiv R_{\gamma}([\sigma, a \rightarrow \gamma_1](\lambda a. M)) \] Since \([\sigma, a \rightarrow \gamma_1](\lambda a. M) \Downarrow_M (\lambda a. M), \vdash [\sigma, a \rightarrow \gamma_1](\lambda a. M) : (\gamma_1 \rightarrow \gamma_2)\]
Case

\[ \Gamma \vdash M_1 : (\gamma' \xrightarrow{\text{y}} \gamma) \quad \Gamma \vdash M_2 : \gamma' \]
\[ \Gamma \vdash (M_1 \ M_2) : \gamma \quad \text{M-app} \]

\( R_{\gamma' \xrightarrow{\text{y}} \gamma}([\sigma]M_1) \) \quad \text{By inductive hypothesis}

\( R_{\gamma'}([\sigma]M_2) \) \quad \text{By inductive hypothesis}

\( R_\gamma([\sigma]M_1 \ [\sigma]M_2) \) \quad \text{By definition of } R_\gamma(\cdot)

\[ \vdash [\sigma](M_1 \ M_2) : \gamma \quad \text{From Lemma F.3 (Preservation)} \]

Case

\[ \Gamma \vdash M_1 : \gamma_1 \quad \Gamma \vdash M_2 : \gamma_2 \]
\[ \Gamma \vdash (M_1, M_2) : (\gamma_1 \times \gamma_2) \quad \text{M-pair} \]

\( R_{\gamma_1}([\sigma]M_1) \) \quad \text{By inductive hypothesis}

\( R_{\gamma_2}([\sigma]M_2) \) \quad \text{By inductive hypothesis}

\[ \exists v_1.([\sigma]M_1 \Downarrow_M v_1) \]
\[ \exists v_2.([\sigma]M_2 \Downarrow_M v_2) \]
\[ ([\sigma]M_1, [\sigma]M_2) \Downarrow_M (v_1, v_2) \]
\[ [\sigma](M_1, M_2) \Downarrow_M (v_1, v_2) \]

\[ \vdash [\sigma](M_1, M_2) : (\gamma_1 \times \gamma_2) \quad \text{By definition of } [\sigma](\cdot) \]

\[ \vdash [\sigma](M_1, M_2) : (\gamma_1 \times \gamma_2) \quad \text{From Lemma F.3 (Preservation)} \]

Case

\[ \Gamma \vdash M_1 : \text{Nm} \]
\[ \Gamma \vdash M_2 : \text{Nm} \]
\[ \Gamma \vdash \langle M_1, \ M_2 \rangle : \text{Nm} \quad \text{M-bin} \]

\( R_{\text{Nm}}([\sigma]M_1) \) \quad \text{By inductive hypothesis}

\( R_{\text{Nm}}([\sigma]M_2) \) \quad \text{By inductive hypothesis}

\[ \exists v_1.([\sigma]M_1 \Downarrow_M v_1) \]
\[ \exists v_2.([\sigma]M_2 \Downarrow_M v_2) \]
\[ \langle [\sigma]M_1, [\sigma]M_2 \rangle \Downarrow_M (v_1, v_2) \]
\[ [\sigma]\langle M_1, M_2 \rangle \Downarrow_M (v_1, v_2) \]

\[ \vdash [\sigma]\langle M_1, M_2 \rangle : \text{Nm} \quad \text{From Lemma F.3 (Preservation)} \]

\( R_{\text{Nm}}([\sigma]\langle M_1, M_2 \rangle) \) \quad \text{By definition of } [\sigma](\cdot) \]

\[ \vdash [\sigma]\langle M_1, M_2 \rangle : \text{Nm} \quad \text{From Lemma F.3 (Preservation)} \]

\( \square \)

**Theorem F.5 (Strong Normalization).** If \( \Gamma \vdash e : \gamma \), then \( e \) is normalizing.

**Proof.**

- \( R_\gamma(e) \) \quad \text{By Lemma F.4 (Normalization)}

- \( e \text{ halts} \) \quad \text{By Lemma F.2 (Termination)}
G STRONG NORMALIZATION PROOF FOR THE INDEX TERM LANGUAGE

Definition G.1 ($R_Y(e)$).

(1) $R_{\text{NmSet}}(e)$ iff $e$ halts.
(2) $R_{\gamma_1, \gamma_2}(e)$ if and only if (1) $e$ halts and (2) if $R_{\gamma_1}(e')$ then $R_{\gamma_2}(e')$.
(3) $R_{\gamma_1, \gamma_2}(e)$ if and only if $e$ halts and $R_{\gamma_1}(e)$ and $R_{\gamma_2}(e)$.

Lemma G.2 (Termination). If $R_Y(e)$, then $e$ halts.

Proof. By definition of $R_Y(e)$.

Lemma G.3 (Preservation). If $\Gamma \vdash e : \gamma$ and $\sigma \vdash \Gamma$, then $\vdash [\sigma]e : \gamma$.

Proof. By induction on the size of $\sigma$. When $\text{size}(\sigma) = 1$, the Lemma reduces to the Substitution Lemma.

Lemma G.4 (Normalization). If $\Gamma \vdash e : \gamma$ and $\sigma \vdash \Gamma$, then $R_Y([\sigma]e)$.

Proof. By induction on the given derivation.

Case $\alpha : \gamma \in \Gamma$

\[ \frac{}{\Gamma \vdash \alpha : \gamma} \]

By definition of $[\sigma][-]$

\[ \exists \nu_1, ([\sigma]a \downarrow \nu_1) \]

By Lemma G.3 (Preservation)

\[ \vdash [\sigma]a : \gamma \]

Since $[\sigma]a \downarrow \nu_1, \vdash [\sigma]a : \gamma$

Case

\[ \frac{}{\Gamma \vdash \sigma() : 1} \]

By definition of $[\sigma][-]$

\[ \sigma() = () \]

By Rule value; Since $()$ val

\[ () \downarrow () \]

Since $[\sigma() = O, O \downarrow O$

\[ \Gamma \vdash () : 1 \]

Given

\[ \vdash [\sigma() : 1 \]

By Lemma G.3 (Preservation)

\[ R_1([\sigma()]) \]

Since $[\sigma()] \downarrow O, \vdash [\sigma()] : 1$

Case

\[ \frac{}{\Gamma \vdash i_1 : \gamma_1 \ \ \Gamma \vdash i_2 : \gamma_2} \]

\[ \Gamma \vdash (i_1, i_2) : (\gamma_1 \times \gamma_2) \]

By inductive hypothesis

\[ R_{\gamma_1}([\sigma]i_1) \]

By inductive hypothesis

\[ R_{\gamma_2}([\sigma]i_2) \]

Since $R_{\gamma_1}([\sigma]i_1)$

\[ \exists \nu_1, ([\sigma]i_1 \downarrow \nu_1) \]

Since $R_{\gamma_2}([\sigma]i_2)$

\[ \exists \nu_2, ([\sigma]i_2 \downarrow \nu_2) \]

By Rule pair

\[ [\sigma][i_1, i_2] \downarrow (\nu_1, \nu_2) \]

By definition of $[\sigma][-]$

\[ \vdash [\sigma][i_1, i_2] : (\gamma_1 \times \gamma_2) \]

By Lemma G.3 (Preservation)

\[ R_{\gamma_1, \gamma_2}([\sigma][i_1, i_2]) \]

Since $[\sigma][i_1, i_2] \downarrow (\nu_1, \nu_2), \vdash [\sigma][i_1, i_2] : (\gamma_1 \times \gamma_2)$
Case $\Gamma \vdash i : \gamma_1 \ast \gamma_2$

$\Gamma \vdash \text{prj}_b i : \gamma_b$

$R_{(\gamma_1 \ast \gamma_2)}([\sigma] i)$ By inductive hypothesis

$R_{\gamma_b}([\sigma] \text{prj}_b i)$ By definition of $R_{\ast}(-)$

Case $\Gamma \vdash \emptyset : \text{NmSet}$

sort-empty

Similar to the sort-unit case.

Case $\Gamma \vdash N : \text{Nm}$

$\Gamma \vdash \{N\} : \text{NmSet}$

sort-singleton

$[\sigma][N] = \{N\}$ By definition of $[\sigma](\cdot)$

$\{N\}$ val By Rule val-singleton

$\Gamma \vdash \{N\} : \text{NmSet}$ Given

$\vdash [\sigma][N] : \text{NmSet}$ By Lemma G.3 (Preservation)

$R_{\text{NmSet}}([\sigma][N])$ Since $[\sigma][N]$ val, $\vdash [\sigma][N] : \text{NmSet}$

Case $\Gamma \vdash X : \text{NmSet}$

$\Gamma \vdash Y : \text{NmSet}$

$\Gamma \vdash (X \cup Y) : \text{NmSet}$

sort-union

$R_{\text{NmSet}}([\sigma]X)$ By inductive hypothesis

$R_{\text{NmSet}}([\sigma]Y)$ By inductive hypothesis

$\exists v_1.([\sigma]X \downarrow v_1)$ Since $R_{\text{NmSet}}([\sigma]X)$

$\exists v_2.([\sigma]Y \downarrow v_2)$ Since $R_{\text{NmSet}}([\sigma]Y)$

$([\sigma]X \cup [\sigma]Y) \downarrow (v_1 \cup v_2)$ By Rule union

$[\sigma]([X \cup Y]) \downarrow (v_1 \cup v_2)$ By definition of $[\sigma](\cdot)$

$\vdash [\sigma](X \cup Y) : \text{NmSet}$ By Lemma G.3 (Preservation)

$R_{\text{NmSet}}([\sigma](X \cup Y))$ Since $[\sigma](X \cup Y) \downarrow (v_1 \cup v_2)$, $\vdash [\sigma](X \cup Y) : \text{NmSet}$

Case $\Gamma \vdash X : \text{NmSet}$

$\Gamma \vdash Y : \text{NmSet}$

extract($\Gamma)$ $\parallel X \perp Y : \text{NmSet}$

sort-sep-union

$\Gamma \vdash (X \perp Y) : \text{NmSet}$

$R_{\text{NmSet}}([\sigma]X)$ By inductive hypothesis

$R_{\text{NmSet}}([\sigma]Y)$ By inductive hypothesis

$\exists v_1.([\sigma]X \downarrow v_1)$ Since $R_{\text{NmSet}}([\sigma]X)$

$\exists v_2.([\sigma]Y \downarrow v_2)$ Since $R_{\text{NmSet}}([\sigma]Y)$

$([\sigma]X \perp [\sigma]Y) \downarrow (v_1 \perp v_2)$ By Rule union

$[\sigma]([X \perp Y]) \downarrow (v_1 \perp v_2)$ By definition of $[\sigma](\cdot)$

$\vdash [\sigma](X \perp Y) : \text{NmSet}$ By Lemma G.3 (Preservation)

$R_{\text{NmSet}}([\sigma](X \perp Y))$ Since $[\sigma](X \perp Y) \downarrow (v_1 \perp v_2)$, $\vdash [\sigma](X \perp Y) : \text{NmSet}$
Case \( \Gamma, a : \gamma_1 \vdash i : \gamma_2 \)

\[
\Gamma \vdash (\lambda a. i) : (\gamma_1 \rightarrow_{\text{idx}} \gamma_2)
\]

sort-abs

Assumption, where \( \Gamma' = \Gamma, a : \gamma_1 \)

\[
\begin{align*}
[\sigma, a \rightarrow \gamma_1] \vdash [\Gamma'] \\
+ (\Gamma' \vdash (\lambda a. i)) : (\gamma_1 \rightarrow_{\text{idx}} \gamma_2)
\end{align*}
\]

By Lemma G.3 (Preservation)

\[
[\sigma, a \rightarrow \gamma_1](\lambda a. i) = (\lambda a. i)
\]

By definition of \([\sigma](-)\)

\[
R_\gamma ([\sigma, a \rightarrow \gamma_1](\lambda a. i))
\]

Since \([\sigma, a \rightarrow \gamma_1](\lambda a. i); (\lambda a. i), \vdash [\sigma, a \rightarrow \gamma_1](\lambda a. i) : (\gamma_1 \rightarrow_{\text{idx}} \gamma_2)\)

Case \( \Gamma \vdash i : \gamma_1 \rightarrow_{\text{idx}} \gamma_2 \)

\[
\Gamma \vdash i : \gamma_2
\]

sort-apply

\[
R_{\gamma_1 \rightarrow_{\text{idx}} \gamma_2}([\sigma]i)
\]

By inductive hypothesis

\[
R_{\gamma_1}([\sigma])
\]

By inductive hypothesis

\[
R_{\gamma_2}([\sigma][1([\sigma]])])
\]

By definition of \(R_{-}(-)\)

\[
R_{\gamma_2}([\sigma][i])
\]

By definition of \([\sigma](-)\)

Case \( \Gamma \vdash M : \text{Nm} \xrightarrow{\text{Nm}} \text{Nm} \)

\[
\Gamma \vdash M[j] : \text{NmSet}
\]

sort-map-set

\[
R_{\text{NmSet}}(j)
\]

By inductive hypothesis

\[
\exists \nu, (M[j] \downarrow \nu)
\]

By Thm. F.5

\[
\exists \nu', (M[j] \leftarrow \nu)
\]

By Rule Single

\[
M[j] \downarrow \nu'
\]

By Rule map-set

\[
\vdash ([\sigma](M[j]) : \text{NmSet}
\]

By Lemma G.3 (Preservation)

\[
R_{\text{NmSet}}([\sigma](M[j]))
\]

Since \(M[j] = \nu'\) and \(\vdash ([\sigma](M[j]) : \text{NmSet}\)

\[\square\]

**Theorem G.5 (Strong Normalization).** If \( \Gamma \vdash e : \gamma \), then \( e \) is normalizing.

**Proof.**

\[
R_\gamma(e)
\]

By Lemma G.4 (Normalization)

\[
e \text{ halts}
\]

By Lemma G.2 (Termination)

\[\square\]