Delta Dictionaries

Total and Extensional Finite Maps in Proof Assistants

NICK COLLINS, University of Chicago

Dictionaries, or finite maps, are a core data structure in any programming language. General-purpose languages implement dictionaries with hash tables or balanced trees, but proof assistants — which favor simplicity and provability over performance — generally employ association lists or finite partial functions. Though suitable for many uses, each solution has drawbacks: association lists allow arbitrary order and may contain duplicates (unless refined with an explicit proof about validity), and partial functions cannot be destructed and their equality is not decidable.

This paper illustrates a novel list-based representation, called *delta dictionaries*, that simultaneously achieves benefits of the conventional representations. Delta dictionaries are *semantically total* (every type-correct list is semantically valid), *extensional* (there is a one-to-one correspondence between lists and semantic mappings), *destructible* (sub-dictionaries can be inspected as needed), and have *decidable equality*. Included is an implementation of delta dictionaries in Agda, and a discussion of when delta dictionaries may or may not be preferable to conventional solutions.

Additional Key Words and Phrases: proof assistants, data structures, dictionaries

1 INTRODUCTION

In most programming contexts, the design of data structures and algorithms is chiefly focused on performance, but in *dependently-typed proof assistants*, evaluation performance is often unimportant since checking a proof requires type-checking but not actual evaluation. As such, proof assistants often use entirely different data structures than those of conventional settings, with a focus on simplicity or useful proof-theoretic properties.

This paper considers the implementation of *dictionaries*, i.e., finite mappings from keys to values. The dictionary is a fundamental utility in most programming contexts and serves a broad range of purposes, so the choice of implementation is very important. In conventional languages, dictionaries are implemented with auto-balanced trees or hashtables, which are highly performant, but also very complicated. This paper discusses four alternatives — three conventional and one novel — that are often better suited to proof development. A fifth alternative is discussed briefly toward the end as a potential avenue for future work.

1.1 Background

This paper assumes familiarity with statically typed functional languages (e.g., Haskell, OCaml), and relevant concepts such as Hindley-Milner type systems, pattern-matching, data types, and product types. A cursory background on *proof assistants* and *dependent types* is developed during the course of this introduction, but unfamiliar readers are encouraged to review a more thorough treatment such as [13] or [12].

A *proof assistant* is a programming language with first-class support for defining and proving theorems, together with editor tools that ease the development of proofs. Many popular proof languages, including Coq and Agda, are founded on *dependent types*, in which a type can be parameterized by concrete values such as numbers, lists, or functions, and in which theorems are types, proofs are functions (or other concrete values), and type checking subsumes proof checking.

Association List	[(3, “b”), (1, “a”), (3, “q”), (6, “c”)]
Canonical Association List	[(1, “a”), (3, “b”), (6, “c”)]
Partial Function	$\lambda x.\ x = 3\ ?\ \text{“b”} : (\lambda x.\ x = 1\ ?\ \text{“a”} : (\lambda x.\ x = 3\ ?\ \text{“q”} : (\lambda x.\ x = 6\ ?\ \text{“c”} : \text{None})\ x)\ x)\ x$
Delta Dictionary	[(1, “a”), (1, “b”), (2, “c”)]

Fig. 1. Dictionary representations after insertions for keys 6, 3, 1, and 3: { 1 ↦ “a” , 3 ↦ “b” , 6 ↦ “c” }

1.2 Conventional Solutions and their Limitations

1.2.1 Association Lists. The simplest representation for a dictionary, an *association list*, is merely a list of key-value pairs [12, Lists]. Insertion and destruction are trivially achieved by the cons operator (`::`) and pattern matching, respectively. Lookup is only slightly more involved, per the Agda code below:

```
-- assumes that the key type K is fixed
-- ∀{V} means that V is an implicit type parameter - the curly braces indicate implicitness
assoc-list-lookup : ∀{V} → List (K × V) → K → Maybe V
assoc-list-lookup [] _ = None
assoc-list-lookup ((k1 , v1) :: l) k =
  with eq-dec-K k k1                -- 'eq-dec-K k k1' checks whether k == k1 ...
... | Inl _ = Some v1                -- if k == k1, then 'eq-dec-K k k1' evals to 'Inl _'
... | Inr _ = assoc-list-lookup l k  -- otherwise, 'eq-dec-K k k1' evals to 'Inr _'
```

Limitation: Lack of Extensionality. Though easy to work with and reason about, association lists have a drawback that can cause difficulty during proof development: because they allow duplicate bindings for the same key and they are sensitive to the order of insertions, many distinct association lists represent the same semantic mapping. For example, the first two lines of Figure 1 show two distinct association lists (among others) that represent a dictionary with three particular bindings. In other words, association lists lack ***Extensionality***: the property that any two non-identical valid lists must have distinct semantic meanings (i.e., if $\forall k(d1[k] == d2[k])$, then $d1 == d2$). Extensionality can make proofs easier [12, Maps] and permits the use of the built-in equality proposition to establish semantic equivalence.

Proof languages such as Agda have built-in support for *reflexive equality*, which judges two values (of the same type) as being equal if and only if they are precisely identical:

```
-- In Agda, 'Set' is (roughly) the type of all types (incl. propositions).
-- _==_ means that == is infix and binary.
-- The first explicit arg is named x and is of type T,
-- the second explicit arg is also of type T but is unnamed.
-- Because == is a proposition, the "return type" is Set.
data _==_ {T : Set} (x : T) : T → Set where
  -- A constructor of a proposition is a proof of that proposition.
  -- refl is the only constructor - for any x, refl is a proof that x == x.
  refl : x == x
  -- There's no way to establish equality of two arbitrary values x and y.
```

The built-in equality proposition is a common and intuitive way to judge the equality of two values. Furthermore, proof assistants have special features for working with it, which can be harnessed to simplify and expedite proof development. Equivalence can be defined in other ways, and the proof assistant may provide *setoid* support for using custom definitions as though they were primitive, but setting this up may require extra overhead or more advanced techniques than simply using built-in reflexive equality. Section 3 and Section 4.2 further demonstrate the importance of Extensionality; its absence can render some theorems, such as *structural properties contraction* and *exchange* [14], false.

1.2.2 Canonical Association Lists. To establish a one-to-one correspondence between association lists and semantic mappings, and thus Extensionality, one solution [9] is to maintain a *canonical* form that is semantically valid and unique, namely, a list in which there are no duplicate keys and, furthermore, where keys are in sorted order.[1] The second association list shown in Figure 1 is one such example. This approach requires a more sophisticated `insert` function than that of the naive association list approach:

```
-- assumes that the key type K is fixed
cal-insert : ∀{V} → List (K × V) → (K × V) → List (K × V)
cal-insert [] (k , v) = (k , v) :: []
cal-insert ((k1 , v1) :: l) (k , v) =
  with ord-dec-K k k1
... | Inl _       = (k , v) :: (k1 , v1) :: l          -- k < k1
... | Inr (Inl _) = (k , v) :: l                       -- k == k1
... | Inr (Inr _) = (k1 , v1) :: cal-insert l (k , v)  -- k > k1
```

Limitation: Lack of Semantic Totality. Although the canonical association list approach is Extensional, it has another drawback; the list-of-pairs type allows arbitrary, possibly invalid lists, so each canonical association list must be packaged with a proof of validity:

```
-- assumes that the key type K is fixed
data _valid-cal ∀{V} (List (K × V)) : Set
  EmptyVal  : [] valid-cal
  SingleVal : ∀{k v} → ((k , v) :: []) valid-cal
  -- For any k1, k2, v1, v2, and l, if k1 < k2, and (k2 , v2) :: l is valid,
  -- then ManyVal stands as a proof that (k1 , v1) :: (k2 , v2) :: l is valid.
  ManyVal   : ∀{k1 k2 v1 v2 l} →
                k1 < k2 →
                ((k2 , v2) :: l) valid-cal →
                ((k1 , v1) :: (k2 , v2) :: l) valid-cal

cal : ∀{V} → Set
-- '∑[ x ∈ T ] (p x)' packages x (of type T) with a proof that it satisfies p
cal {V} = ∑[ l ∈ List (K × V) ] (l valid-cal)
```

[1] The Coq standard library [8], describes unordered but deduplicated association lists as "weak," implying that canonical association lists are "strong."

This paper coins the term ***Semantically Total*** to describe any data structure whose type is free of proof terms and yet contains no invalid members — i.e., the mapping from values in the underlying type to their semantic meanings is total. To illustrate, a pair of integers is not a Semantically Total data structure for the rational numbers, since if the second integer is 0 then the pair is invalid, but a pair of an integer and a natural, where the natural represents the denominator minus 1, *is* Semantically Total (note that neither of these data structures is Extensional). The downsides of having to use validity proofs are discussed in Section 3 and Section 4.2.

1.2.3 Partial Functions. A third conventional solution is to represent dictionaries as finite partial functions [12, Maps] (i.e., functions that return `None` for all but finitely many inputs). The third line of Figure 1 depicts a nested λ-expression that serves as the "lookup table" (the `Some` constructors are omitted for space). This approach can be made Extensional by postulating *functional extensionality* [12, Logic], which axiomatically asserts that $(\forall x, f(x) == g(x)) \Rightarrow f == g$. On the other hand, this approach technically fails to satisfy Semantic Totality, since the function type permits *non*-finite maps. Even so, this is often not a problem in practice — as long as the dictionary is never iterated, destructed, or counted, an infinite dictionary is indistinguishable from a finite dictionary. Furthermore, a dictionary built from a finite program can only have finitely many mappings. The more serious problems are that, unlike either association list approach, functions lack ***Decidable Equality*** and cannot be ***Destructed***.

Limitation: Lack of Decidable Equality. While a proof that $x == y$ establishes the truth that x and y are equal, the `==` proposition is not capable of *deciding* whether or not two arbitrary values are equal. To *decide* this question, we need to define a function that accepts two arguments (of the same type) and returns either a proof that they are equal or a proof that they are not equal:

```
-- 'P → ⊥' means that P is false.
eq-dec-K : (x y : K) → ((x == y) ∨ (x == y → ⊥))
eq-dec-K x y = ?  --  the implementation will depend on the type K
```

Note that proof languages like Coq and Agda are *constructive*; they elide the *law of the excluded middle*, so it might not be possible to prove that a certain proposition is either true or false. As such, the ability to decide whether some proposition (such as the equality of two values) is true or false cannot be taken for granted — if this ability is needed, it must be explicitly established by defining a function such as the one above. In the case of functions, it's not possible to establish whether two arbitrary functions are extensionally equal, so there is no way to establish decidability. Naturally, deciding whether or not two dictionaries are equal is important and useful for many of the same reasons it would be in a more conventional language, so the lack of Decidable Equality is a substantial weakness.

Limitation: Lack of Destructibility. The inability to destruct a function is more intuitive — destruction is essentially the same as in other functional languages, where pattern-matching is typically used to separate one element of a collection (often the *head*) from the rest of the collection. A lambda value can't be destructed or picked apart in any way, so its values also can't be iterated or counted. The function could be packaged with a set indicating its domain — i.e., a canonical list of keys — but would then have to include a proof that those keys are correct, violating Semantic Totality in the same troublesome way that canonical association lists do.

	Client Usage					Implementation
	Total	Extensional	Decid. Eq.	Destructible	Key Type K	Simple
Association List	✓	✗	✓	✓	(=)	✓$^{+}$
Canonical Association List	✗	✓	✓	✓	(=), (<)	✓$^{-}$
Partial Function	✗[2]	✓	✗	✗	(=)	✓
Delta Dictionary	✓	✓	✓	✓$^{-}$	$f : K \leftrightarrow Nat$	✗

Fig. 2. Properties of dictionary representations.

1.3 Core Properties

In some cases, the aforementioned drawbacks are minor or can be worked around. But developing large proofs is challenging, so any stumbling block can cause exorbitant increases in verbosity, time, effort, and accidental complexity. Furthermore, as shown in Section 3, there are cases where these drawbacks make a proof task not merely difficult, but outright impossible.

Ideally, when working in a proof assistant, an implementation of a data structure — dictionaries in particular for this paper — would satisfy these properties:

(1) ***Semantic Totality:*** Every value in the representation type is semantically valid, without requiring proof terms — i.e., the mapping from values to their semantic meanings is total.

(2) ***Extensionality:*** Built-in equality corresponds to semantic equivalence, i.e., two unequal values have different semantic meanings.

(3) ***Decidable Equality:*** Built-in equality is decidable for the representation type.

Furthermore, in addition to properties about the external interface of the data structure, it is often useful to retain the ability to inspect, iterate, and manipulate sub-dictionaries.

(4) ***Easy Destructibility:*** The ability to decompose a data structure into atomic subparts in a convenient manner.

Figure 2 summarizes the preceding discussion along these dimensions. The "Key Type" column indicates which propositions must be decidable for the key type. Note that the association list representations can be easily destructed, but destruction is not possible for partial functions, and that none of the three conventional solutions are total, extensional, and have decidable equality.

1.4 Novel Solution: Delta Dictionaries

The four core properties can be (mostly) satisfied by way of *delta dictionaries*. A delta dictionary can be described as a "canonical-by-construction" association list: instead of storing each literal key value, it stores the *difference* from the previous key, minus 1 (details in Section 2). For example, compare the canonical association list and delta dictionary in Figure 1:

Canonical Association List	[(1, "a"), (3, "b"), (6, "c")]
Delta Dictionary	[(1, "a"), (1, "b"), (2, "c")]

Every well-typed list-of-pairs is a valid delta dictionary, thus no proof term is needed to establish validity (Semantic Totality). Every unique delta dictionary represents a unique semantic mapping, thus built-in equality may be used for

[2] As aformentioned, this is often not a problem in practice, so for many practical intents and purposes, this may as well be a ✓.

semantic equivalence (Extensionality). Furthermore, it's trivial to define a function which determines whether or not two delta dictionaries are equal (Decidable Equality), and a `destruct` function, which permits destruction albeit in a more awkward manner than standard pattern matching (Not-So-Easy Destructibility).

As summarized in Figure 2, delta dictionaries strike a new balance in this design space. Compared to canonical association lists, delta dictionaries enjoy Semantic Totality — the only one of these properties that the canonical association list does not. However, destruction and iteration for delta dictionaries is substantially more difficult. Furthermore, unlike all of the conventional approaches, delta dictionaries require a bijection to the naturals, not merely decidable equality or ordering, for their key types. Lastly, though not a concern from a client's perspective, the implementation of delta dictionaries is considerably more involved than the conventional approaches.

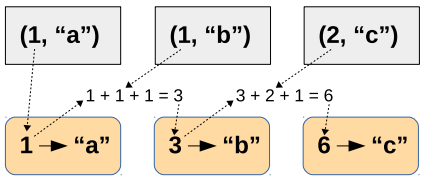
Outline. Section 2 describes the core operations for delta dictionaries and the relevant theorems. Section 3 describes a small case study in proof development that demonstrates the necessity of delta dictionaries. Finally, Section 4 concludes with a discussion.

2 IMPLEMENTATION

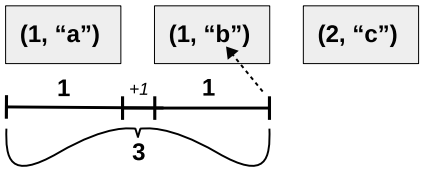
This section covers the Agda implementation of the core functionality of delta dictionaries, and also defines the key theorems. Additional theorems and functionality, as well as the Agda proofs of the theorems, are not included but are available on GitHub at [4].

2.1 Deltas and Key Types

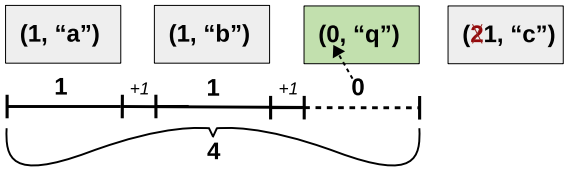
Borrowing from association lists, delta dictionaries use a list-of-pairs, where the first item in each list represents a key and the second is the literal value that is mapped to the represented key. Unlike association lists, the natural number that represents the key is not its literal value.



(a) An example delta dictionary.



(b) How do we find key 3?



(c) How do we add a mapping from 4 to "q"?

Fig. 3. Example delta dictionary operations.

In order to allow multiple key types, while relying on useful properties of natural numbers under the hood, delta dictionaries accept a typeclass argument representing a bijection between the key type and the naturals, and use this bijection to convert keys to and from naturals. This is discussed further in ***Bijections to Natural Numbers***.

In order to maintain canonical order, while ensuring that every natural number is valid as the first item in a pair, that number must be the non-negative difference between the key it represents and the previous key. This number is called a *delta*. As described so far, a 0 offset would correspond to a duplicate key. To prevent duplicate keys from being represented, a delta is actually the offset minus 1: a delta of 0 indicates an offset (from the previous key) of 1, a delta of 2 indicates an offset of 3, and so on.[3] The head of the list, however, does not follow the "minus 1" rule: so the first "delta" is interpreted literally, i.e., it represents the key value exactly.

Figure 3a depicts the delta dictionary corresponding to the example from Section 1.

Bijections to Natural Numbers. In this implementation, each key is represented by a natural number, permitting the use of addition, subtraction, and successor[4] operations for defining deltas. There may be an algebraic structure more general than the natural numbers that satisfies these requirements, but for practical purposes, working with naturals is easier for both the implementer and the client. The bijection typeclass accepts a `toNat` function, as well as proofs that `toNat` is injective and surjective, and the inverse function is defined by the library using the proof of surjectivity.[5] The GitHub repo [4] demonstrates this by providing an instance of the bijection typeclass for integers (25 lines of code).

Most types that are suitable for use as keys in the first place, especially strings and numeric types, can be bijected to the naturals, though doing so may be onerous. Finite types, such as characters, are suitable as keys, but cannot be bijected to the naturals — simultaneously achieving Semantic Totality and Extensionality for a dictionary over finite types may require a custom-made dictionary data type.

Delta dictionaries are canonically ordered, but by the natural ordering of the naturals, which, depending on the choice of bijection, may correspond to an unnatural ordering of the key type. As such, the ordering is not exposed to the client — functions such as `destruct` produce results that are in arbitrary order from the client's perspective.

2.2 Lookup, Insertion, and Destruction

Figure 3b and Figure 3c illustrate example lookup and insert operations. These proceed largely as they would for association lists, but working indirectly with keys encoded as natural numbers and, furthermore, differences between these numbers.

Figure 4 presents concrete implementations for lookup (i.e., `_⟨_⟩`), insert (i.e., `_,,_`), and `destruct` in Agda. Note that the lookup function `_⟨_⟩` takes two args, dictionary `d` and key `k`, with the syntax `d ⟨ k ⟩`. Also note that the `Delta` module is parameterized by key type `K` and bijection `bij` to the naturals, while the operations are parameterized by value type `V`. The type `DL` describes the raw "delta lists" used internally, which are wrapped by the `DD` datatype exposed to clients.

The helper function `delta` computes the delta, i.e., the offset minus 1, between two distinct numbers. Given that, lookup is straightforward. The insert function is also fairly straightforward. If the delta to insert is less than the delta of the first pair, then we place the delta to insert as the first pair of the new list, and the original first pair will be the second pair of the new list, but using the delta between its original value and the inserted delta. If the first pair is an

[3] An alternative implementation might use positive integers instead of natural numbers — this may be simpler in languages with good support for positive integers as a separate type from 0-based naturals.
[4] The successor of n is $n + 1$.
[5] Without surjectivity, a binding for an unmapped natural either invalidates the delta dictionary, violating Semantic Totality, or is ignored, in which case its presence or absence does not affect the semantic meaning, violating Extensionality.

```
module Delta (K : Set) {{bij : bij K Nat}} where

  DL : (V : Set) → Set
  DL V = List (Nat × V)

  data DD (V : Set) : Set where
    DD : DL V → DD V

  delta : ∀{n m} → n < m → Nat
  -- In Agda, almost any character can appear in an identifier.
  -- So 'n<m' is the variable name for a proof that 'n < m', whereas 'n<m→1+n≤m' is a theorem.
  -- 'difference (n<m→1+n≤m n<m)' evaluates to 'm - n - 1'
  delta n<m = difference (n<m→1+n≤m n<m)

  -- lookup
  _⟨_⟩ : ∀{V} → DD V → K → Maybe V
  (DD dd) ⟨ k ⟩ =
    lkup dd (toNat k)
    where
      lkup : ∀{V} → DL V → Nat → Maybe V
      lkup [] n = None
      lkup ((hn , ha) :: t) n
        with <dec n hn
      ... | Inl _          = None                 -- n < hn
      ... | Inr (Inl _)    = Some ha              -- n == hn
      ... | Inr (Inr hn<n) = lkup t (delta hn<n)  -- hn < n

  -- insert/extend
  _,,_ : ∀{V} → DD V → (K × V) → DD V
  (DD dd) ,, (k , v) =
    DD (ins dd (toNat k , v))
    where
      ins : ∀{V} → DL V → (Nat × V) → DL V
      ins [] (n , v) = (n , v) :: []
      ins ((hn , hv) :: t) (n , v)
        with <dec n hn
      ... | Inl n<hn       = (n , v) :: (delta n<hn , hv) :: t
      ... | Inr (Inl _)    = (n , v) :: t
      ... | Inr (Inr hn<n) = (hn , hv) :: ins t (delta hn<n , v)

  destruct : ∀{V} → DD V → Maybe ((K × V) × DD V)
  destruct (DD [])                          = None
  destruct (DD ((n , v) :: []))             = Some ((fromNat n , v) , DD [])
  destruct (DD ((n , v) :: (m , v') :: t)) = Some ((fromNat n , v) , DD ((n + m + 1 , v') :: t))
```

Fig. 4. Delta dictionary operations.

exact match, then we simply replace the old value with the new one. `destruct` simply pops the head off the list, and then augments the next key by the offset.

The Coq Standard Library [7, 8] describes key theorems about dictionaries, including those for lookup and insert (it does not define `destruct`). The GitHub repo [4] proves the relevant subset of these theorems. Some of the theorems in the Coq treatment involves concepts not relevant to this treatment, especially multiple distinct notions of equality. Also, since this interface does not expose mapping order to the client, there are no proofs of any of the theorems pertaining to ordering (i.e., those that are not "weak").

2.3 Additional Operations

The GitHub repo [4] also defines key deletion, union, map, and to/from-list operations, along with appropriate theorems (not reproduced here).

2.4 Properties

Figure 5 shows definitions of the four properties identified in Section 1.3, as well as two important consequences of Extensionality that will be discussed further in Section 3. The proofs of these theorems can be found in the GitHub repo [4]. Recall that key type `K` is a module parameter and is implicitly bound in the type `DD V`.

Remark 1 (Semantic Totality). This property cannot be formally defined — rather its truth is apparent from the fact that the other theorems do not require their delta dictionary arguments to be packaged with validity proofs.

Theorem 2 (Extensionality).

```
∀{V} {dd1 dd2 : DD V} →
  ((k : K) → dd1 ⟨ k ⟩ == dd2 ⟨ k ⟩) →
  dd1 == dd2
```

Theorem 3 (Decidable Equality).

```
∀{V} (dd1 dd2 : DD V) →
  ((v1 v2 : V) → v1 == v2 ∨ v1 ≠ v2) →
  dd1 == dd2 ∨ dd1 ≠ dd2
```

Theorem 4 (Not-So-Easy Destructibility).

```
∀{V} {dd dd' : DD V} {k : K} {v : V} →
  (destruct dd == None → dd == ∅)
    ∧
  (destruct dd == Some ((k , v) , dd') →
    (k ∉ dd' ∧ dd == dd' ,, (k , v))
```

Theorem 5 (Dictionary Contraction).

```
∀{V} {dd : DD V} {k : K} {v v' : V} →
 dd ,, (k , v') ,, (k , v) == dd ,, (k , v)
```

Theorem 6 (Dictionary Exchange).

```
∀{V} {dd : DD V} {k1 k2 : K} {v1 v2 : V} →
  k1 ≠ k2 →
  dd ,, (k1 , v1) ,, (k2 , v2) ==
  dd ,, (k2 , v2) ,, (k1 , v1)
```

Fig. 5. Core properties.

3 CASE STUDY

To illustrate the practical importance of the four core properties, consider a simply-typed λ-calculus augmented with an `assert` operator. Suppose our goal is to formalize an environment-based (as opposed to substitution-based) evaluation semantics, and prove that evaluation is strongly normalizing and that it satisfies the structural properties *contraction*

```
data exp : Set where
  Var_         : string → exp
  ·λ_·_        : string → exp → exp
  _∘_          : exp → exp → exp
  _::_         : exp → typ → exp
  nat_         : Nat → exp
  assert[_==_] : exp → exp → exp

data res : Set where
  [_]λ_·_ : res env → string → exp → res
  nat_    : Nat → res
  Err     : res

data _⊢_⇒_ : res env → exp → res → Set where
  {- Standard constructs elided... -}
  EvalAsrtEq : ∀{E e1 r1 e2 r2} →
      E ⊢ e1 ⇒ r1 →
      E ⊢ e2 ⇒ r2 →
      r1 == r2 →
      E ⊢ assert[ e1 == e2 ] ⇒ r1
  EvalAsrtNE : ∀{E e1 r1 e2 r2} →
      E ⊢ e1 ⇒ r1 →
      E ⊢ e2 ⇒ r2 →
      r1 ≠ r2 →
      E ⊢ assert[ e1 == e2 ] ⇒ Err
  EvalApErr1 : ∀{E a b} →
      E ⊢ a ⇒ Err →
      E ⊢ a ∘ b ⇒ Err
  EvalApErr2 : ∀{E a b} →
      E ⊢ b ⇒ Err →
      E ⊢ a ∘ b ⇒ Err

contraction : {E : res env} {x : string} {e : exp} {v v' r : res} →
   (E ,, (x , v') ,, (x , v)) ⊢ e ⇒ r →
   (E ,, (x , v)) ⊢ e ⇒ r

exchange : {E : res env} {x1 x2 : string} {e : exp} {v1 v2 r : res} →
   x1 ≠ x2 →
   (E ,, (x1 , v1) ,, (x2 , v2)) ⊢ e ⇒ r →
   (E ,, (x2 , v2) ,, (x1 , v1)) ⊢ e ⇒ r

strong-norm : ∀{Γ e t} → Γ ⊢ e :: t → ∑[ r ∈ res ] (E ⊢ e ⇒ r)
```

Fig. 6. Scenario that requires delta dictionaries over other approaches.

and *exchange*. Contraction and exchange hold that evaluation is insensitive to duplicated and reordered bindings in the environment, respectively.

Figure 6 shows a sketch of this environment-based evaluation (`_⊢_⇒_`) of expressions (`exp`) to results (`res`). The definitions of evaluation environments (`env`, `E`), types (`typ`, `t`), type contexts (Γ), and typechecking (`_⊢_::_`) are not shown. `assert` takes two arguments; if their evaluation results are exactly equal, `assert` evaluates to the first, otherwise it evaluates to an `Err` result.

Evaluation environments are used both as a parameter to the evaluation judgment as well as a data component of closure results. What dictionary implementation should we choose to represent environments?

Using basic association lists, contraction is false: for example, given environments $E_1 =$ `E „ (n , v') „ (n , v)` and $E_2 =$ `E „ (n , v)`, evaluation produces the closures $E_1 \vdash \lambda x.e \Rightarrow [E_1]\ \lambda x.e$ and $E_2 \vdash \lambda x.e \Rightarrow [E_2]\ \lambda x.e$ but $[E_1]\ \lambda x.e \neq [E_2]\ \lambda x.e$. A similar problem occurs for exchange, so it is also false. In order for contraction and exchange to be true for any system with closure results, the dictionary implementation used for closures must be Extensional.

Canonical association lists must be packaged with validity proofs wherever they go, including in the closure result. In a validity proposition `valid : dict -> Set`, the dictionary object is in *negative position*, and in our case the dictionary type is dependent on the result datatype, so the result is also in negative position, violating strict positivity.[6] As such, results cannot contain validity proofs, so for any system that uses dictionaries as data, the dictionaries must be Semantically Total.

Because partial functions do not have Decidable Equality, it is not possible to decide which `EvalAsrt` constructor would apply to an `assert` of two closures. As such, using partial functions would make it impossible to constructively prove strong normalization.

In contrast, delta dictionaries — uniquely amongst the surveyed solutions — possess Semantic Totality, Extensionality, and Decidable Equality. Thus, they are a suitable choice for implementing environments, enabling proofs of contraction, exchange, and strong normalization. We discuss the generality of this case study further in Section 4.2.

4 DISCUSSION

4.1 Design Tradeoffs of Difficult Destruction

Is it possible to have a data structure that possesses all four properties? Probably not.

Extensionality requires canonical ordering and deduplication, and Semantic Totality means that the canonical order and deduplication have to come from how the data is interpreted rather than how it is organized. The non-literal way in which key data is interpreted for delta dictionaries means that it is not safe for client code to work with the raw data directly — rather, all interaction with the data must be encapsulated in library functions.

Unfortunately, this includes *destruction*; an interaction which normally goes through the very natural, elegant, and well-supported mechanism of pattern matching is now only available through the library function `destruct`. Theorem 4 proves that this function destructs the delta dictionary in essentially the same way that a case expression destructs a list of pairs (although the order in which bindings are plucked away is arbitrary from the client's perspective).

`destruct` achieves the same purpose as typical pattern matching (albeit more awkwardly), but because it does not harness the primitive notions of pattern matching and structure, it does not establish structural decrease on the dictionary object, which may break out-of-the-box *structural recursion* in the likely case of recursion on the subdictionary

[6] *Strict positivity* is a requirement of datatype constructors in proof languages like Agda and Coq, so as to avoid logical contradictions. It disallows an argument of a constructor from being a function whose argument types (which are in "negative positions") refer to the datatype. For more information, see [15].

`dd'`. Structural recursion essentially means that at least one argument to any recursive call must be the direct child of the same argument to the outer call. For example, let's say a function `f` takes a tree `t` and a natural `n` as arguments — a recursive call inside `f` would have to either be on the left or right subtree of `t`, or on `n - 1`. Proof languages like Agda and Coq generally require that all functions be structurally recursive in order to ensure that they terminate, as non-terminating functions can introduce logical contradictions.

Structural recursion is often easy to establish when some argument to a function is destructed, and then recursive calls are made using the destructed subcomponents. In the case where destruction is done by way of a function, instead of pattern matching, the system has no way to assert that the output of the function is a subcomponent of one of its inputs, preventing out-of-the-box structural recursion. It may yet be possible, though, to establish structural recursion by adding an extra parameter which tracks some property of the original parameters and which decreases with every recursive call. In the case of dictionaries, the natural choice is the size of the dictionary — `dd'` returned by `destruct` has size exactly one less than the input `dd`. Having to add another parameter, as well as a proof that it is equal to the dictionary's size, is painful, requiring a lot more effort and boilerplate than out-of-the-box structural recursion. However, it does mean that proving theorems about destruction and iteration of delta dictionaries is still *possible*.

The following additional theorem facilitates this awkward alternative means of establishing structural recursion:

```
extend-size :
  ∀{V} {dd : DD V} {k : K} {v : V} →
    k ∉ dd →
    || dd ,, (k , v) || = 1 + || dd ||
```

Because destruction of basic association lists and canonical association lists can generally rely on out-of-the-box structural recursion, delta dictionaries fall short of Easy Destructibility. Nonetheless, they are still better than partial functions in this regard, seeing as it is not only hard but impossible to destruct partial functions.

Because satisfying all four properties seems unattainable, there seems to be an inherent trade-off between desirable properties. In cases where Semantic Totality and Decidable Equality are critical, and there is little to no need to destruct dictionaries, delta dictionaries seem to be a clear winner over the conventional solutions. But in cases where Semantic Totality is not so important, but inspection or destruction are, canonical association lists may be preferable.

4.2 Potential Practical Applications

Generality of Case Study. The scenario in Section 3 may seem contrived, but it is actually a simplification of a real task faced by the author: to complete an Agda formalization of Smyth [11], a program synthesis technique which uses natural semantics (i.e., big-step, environment-based evaluation) and which requires assertions. The development in Section 3 captures the essence of the much larger development [5] for Smyth. Although that development has not been completed, some of the challenges it presented — summed up in Section 3 — necessitated the invention of delta dictionaries.

Perhaps, instead of using delta dictionaries, these issues could be worked around, by changing or dropping some of our criteria? In many cases, substitution can be used to avoid environments. However, environments are often preferred — usually in combination with big-step operational semantics — because they allow the formalization to more closely resemble the implementation of a simple recursive interpreter. Furthermore, Smyth has hole closures in addition to lambda closures, so closing over environments would be necessary regardless.

Contraction and exchange are not always necessary properties for program foundation judgments — rather, substructural type systems, by definition, deliberately violate one or both of these properties.[7] That said, a judgment should generally uphold the structural properties unless there is a strong and explicit reason not to. Proving the right properties is both a matter of good housekeeping and necessity: they are standard considerations because they arise very naturally in any other interesting metatheory, and the inability to prove them is often a large red flag indicating a bug in the judgment's definition. Because contraction and exchange are properties of delta dictionaries themselves, the proofs of these properties for one or many judgments come "for free." Additionally, the useful properties of delta dictionaries may greatly reduce the difficulty of the usually more judgement-specific proof of *weakening*.

It may not seem useful to assert that two functions are intensionally identical, but if assertion is relaxed, so as to test consistency or partially extensional equality instead of purely intensional equality, it would require environments that are destructible. Scenarios which require destructibility but not Decidable Equality lead to the same conclusions, seeing as these columns are identical in Figure 2.

In addition to breaking strict positivity, refinement proofs (e.g., validity proofs that must be packaged with canonical association lists) can be the source of less severe pain points in a broader range of circumstances. The ability to refine ordinary types with proofs of validity is one of the most interesting and useful benefits of dependently-typed languages, and this power should be appreciated. However, refining with proof terms can come at a practical cost, so even in dependently-typed languages, there is high value in avoiding refinements whenever possible (or at least whenever profitable). The practical cost of refinements is that proof terms do not possess the properties Extensionality or Decidable Equality: due to *proof relevance*, two proofs of the same property may be unequal, and due to the fact that proof terms may contain functions, proof terms do not possess Decidable Equality in the general case. Thus, Semantic Totality — which obviates the need for refinement — has a lot of practical value in many general cases beyond those where it is absolutely necessary to appease the positivity checker.

Dictionaries vs. Custom Datatypes. Dictionaries are not always used to represent environments in formalizations of programming language theory.

When the order of bindings does not matter, dictionaries are a natural choice for type contexts. But if types defined "later" in inner scopes, can refer to types defined "earlier" in outer scopes, then order-insensitive dictionaries are inherently inappropriate. This is the case for languages that support subtyping — notably, the subject of the POPLmark Challenge [1]. Linear type systems, on the other hand, require sensitivity to duplicate insertions, so duplication-insensitive dictionaries are inappropriate for them as well [14]. Though association lists remain the most natural choice in these cases, future work could explore data structures that are sensitive to ordering but not duplication, or vice versa.

The use of dictionaries is furthermore avoided in many systems that use substitution rather than environments in defining the dynamic semantics of a language, as well as the many systems that avoid named variables altogether by using De Bruijn indices or comparable techniques — see *Certified Programming with Dependent Types* [2, Library Firstorder] for an introduction.

For these reasons, many existing formalisms make little use of order-and-duplication-insensitive dictionaries. However, as mechanized formalisms becomes increasingly popular, for an ever-broadening scope of applications, it seems inevitable that a programming utility as fundamental as dictionaries will eventually become ubiquitous, at which point it will be important to have the best implementations at our disposal.

[7] Technically, some substructural type systems (e.g., *relevant* type systems) uphold contraction and exchange but violate *weakening*.

4.3 Related Work

How often are different dictionary representations used?

Conventional Representations. Due to their simplicity, basic association lists are perhaps the most typical implementation for dictionaries. But because Extensionality is so important in simplifying proofs, partial functions also see significant use — in key works such as *Software Foundations* [12, Maps] and *Formal Reasoning About Programs* [3] — despite requiring a bit of extra overhead.

Canonical association lists seem to get little use, perhaps because working with validity proofs might add more hassle than Extensionality alleviates. Canonical association lists are defined in the Coq standard library as `FMapList` [9]; a GitHub search for `FMapList` shows 304 results, whereas a search for `FMapAVL` shows 474 results, suggesting that canonical association lists have been found to be less useful than high-performance implementations. A search for `FunctionalExtensionality` [10], on the other hand, turns up 4916 results, though most of these are probably unrelated to dictionaries.[8] de Amorim [6] provides a more comprehensive treatment of canonical association lists, augmenting them with a functional interface so that the client can use them as though they were partial functions (note that this is different from having a partial function augmented with keys).

Performance Concerns. It was noted above that AVL implementations are apparently more popular than canonical association lists, and as more software is mechanized (i.e., programmatically formalized), performance is likely to become an even greater concern than it is today. In cases where there is no extraction step (which transpiles non-proof parts of a mechanization into a conventional, non-proof language), and the proof language is also the language that will be executed, there may be no choice but to use implementations that are high-performance but theoretically unwieldy.

However, it seems more appropriate, especially with fundamental utilities such as dictionaries, to either extract or translate the mechanization into an implementation language that defines these utilities natively by way of highly efficient implementations such as hashtables or red-black trees. This extraction may not be completely fidelitous, since it will be using a different data structure in the implementation than was used in the proofs, but presumably the implementation's version of dictionaries is well-tested and bug-free, and its definition of equality is, at least for fundamental utilities such as dictionaries, extensional and decidable.

Regardless, even if unperformant implementations cannot be used for code that will be run, they may be useful for parts of the code which are only used for proofs and will not be executed.

4.4 When Semantic Totality Doesn't Matter

The spirit of Semantic Totality — that all values of a type are valid by construction, without requiring additional overhead — has broad utility. However, the particular criterion that the type involve no proof terms in order to uphold validity can be arbitrary in cases where the proof type is "*propositional*". To be propositional, all values of the type must be equal, i.e., for any $x, y : T$, $x = y$. As such, a propositional proof type is Extensional and has Decidable Equality, so it does not suffer from the problems that can arise when using non-propositional proofs. With the use of proof terms being no more cumbersome than the use of any other type, the requirement that a semantically total approach not involve any proof terms becomes useless.

Owing to this observation, there is yet another approach, not yet discussed, which fulfills all the core properties, besides Semantic Totality, which it technically violates but fulfills in spirit. This approach is a variant of canonical

[8] These searches and can be reproduced by URLs of the form: https://github.com/search?l=&p=3&q=FMapList+language%3ACoq&ref=advsearch&type=Code, replacing `FMapList` with `FMapAVL` and `FunctionalExtensionality`. Accessed August 23, 2020.

association lists, but instead of having a proof of the whole structure's correctness on the outside, an inequality proof is baked into the "cons" constructor. Since it is generally easy to establish the propositionality of less-than relations, this approach is Extensional and has Decidable Equality, and since the data is stored literally, it's almost as trivial to destruct as canonical association lists. Because the proofs only refer to the key type, and not the value type nor the dictionary type, there are no issues with the positivity checker, and since the proofs are buried deep in the data structure, the client never has to worry about managing them. Essentially, the purposes of Semantic Totality are satisfied even if its technical definition is not. Furthermore, this approach does not require bijection to the naturals, only that the less-than relation is proven to be propositional, and is substantially simpler to implement than delta dictionaries.

There are perhaps many ways to implement this proofs-on-the-inside (POTI) canonical association list approach — the one detailed below adds a helper parameter which indexes a list with its least key, permitting a comparison between that list and some lesser key without inspecting the list:

```
data potical' (V : Set) : Maybe Nat → Set where
  PINone : potical' V None
  PISngl : (n : Nat) → V → potical' V (Some n)
  PIMore : ∀{n2} → (n1 : Nat) → V →
           n1 < n2 →
           potical' V (Some n2) →
           potical' V (Some n1)

data potical (V : Set) : Set where
  PI : ∀{mn} → potical' V mn → potical V
```

In some sense, the POTI canonical association list approach is isomorphic to delta dictionaries. A proof that $a < b$ might constitute a natural number n along with a proof that $b = a + 1 + n$. This n corresponds to a delta in a delta dictionary. Effectively, a delta in a delta dictionary is interpreted in a similar manner to how a less-than proof term is interpreted. Consequently, a conversion function between this approach and delta dictionaries does a simple pass over the mappings, transforming deltas into less-than proofs:

```
convert' : ∀{V} → (n : Nat) → V → dl V → (potical' V (Some n))
convert' n v [] = PISngl n v
convert' n1 v1 ((n2 , v2) :: dl)
  = PIMore n1 v1 (n<m→n<s+m n<1+n) <| convert' (n2 + 1+ n1) v2 dl

convert : ∀{V} → dl V → potical V
convert []              = PI PINone
convert ((n , v) :: dl) = PI <| convert' n v dl
```

The important difference, then, is that POTI canonical association lists only work in proof-savvy type-systems, so delta dictionaries, or similar techniques, may still be useful in conventional languages. While delta dictionaries themselves are unperformant and thus unlikely to be a good choice in conventional settings, the strategy that they illustrate may be applicable to more efficient data types. This strategy may also be useful for invariants that cannot

be made propositional, e.g., implication invariants, which are represented as arrow types. Note that a key element of this strategy involves squeezing out any information redundancies, because redundancies necessitate invariants. Each mapping for the POTI canonical association list approach records three items ($n1$, $n2$, and the proof that $n1 < n2$) whereas a delta dictionary mapping records only one degree of freedom (the delta).

Since the POTI canonical association list approach is easy to implement and effortlessly checks off all the criteria, it's surprising that it seems to see little to no practical use or textbook discussion. As aforementioned, many tasks don't require the use of dictionaries per se, whereas those that do may not require that the implementation satisfy all four core properties. Nonetheless, the general utility of dictionaries and the limitations suffered by implementations that fail to satisfy one of the properties suggest that this approach, or some other suitable approach, ought to be implemented and widely used in the near future. Further development and study of this and related approaches may be worthwhile areas for future work.

4.5 Conclusion and Future Work

This paper discusses the important properties Semantic Totality, Extensionality, Decidable Equality, and Easy Destructibility, and offers an implementation for dictionaries that (mostly) fulfills these properties.

Future work could consider implementations for other key utilities, such as trees or graphs, that satisfy these properties. Doing so could be far more challenging. It is relatively easy to simultaneously satisfy Semantic Totality and Extensionality for list-like structures such as dictionaries, but much more awkward to do so for highly structural data such as graphs. For unlabeled graphs in particular, establishing a one-to-one correspondence between values and semantic meanings requires understanding of the graph isomorphism problem, which involves complex algebra.

4.6 Acknowledgements

Much thanks to my committee, Ravi Chugh, Robert Rand, and John Reppy, for all the help they've provided. Thanks also to Ian Voysey for all his help and suggestions, and to Andrew McNutt for suggesting the term "delta". Finally, thanks to the anonymous reviewers who reviewed a submission of this paper to CPP and pointed out some issues.

REFERENCES

[1] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized Metatheory for the Masses: The POPLmark Challenge. In *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*.

[2] Adam Chlipala. 2019. *Certified Programming with Dependent Types*. Electronic textbook.

[3] Adam Chlipala. 2021. Formal Reasoning About Programs. (June 2021). http://adam.chlipala.net/frap

[4] Nick Collins. 2020. Delta dictionaries in Agda. (Nov. 2020). https://github.com/nickcollins/dependent-dicts-agda/tree/CPP-2021-Submission

[5] Nick Collins. 2020. hazelnat-myth-agda. (Nov. 2020). https://github.com/hazelgrove/hazelnat-myth-agda

[6] Arthur Azevedo de Amorim. 2020. Extensional Structures: fmap. (Aug. 2020). https://github.com/arthuraa/extructures/blob/master/theories/fmap.v

[7] Coq Standard Library. 2020. FMapFacts. (Aug. 2020). https://coq.inria.fr/library/Coq.FSets.FMapFacts.html

[8] Coq Standard Library. 2020. FMapInterface. (Aug. 2020). https://coq.inria.fr/library/Coq.FSets.FMapInterface.html

[9] Coq Standard Library. 2020. FMapList. (Aug. 2020). https://coq.inria.fr/library/Coq.FSets.FMapList.html

[10] Coq Standard Library. 2020. FunctionalExtensionality. (Aug. 2020). https://coq.inria.fr/library/Coq.Logic.FunctionalExtensionality.html

[11] Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program Sketching with Live Bidirectional Evaluation. *Proceedings of the ACM on Programming Languages (PACMPL), Issue ICFP* (2020).

[12] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2018. *Logical Foundations*. Electronic textbook.

[13] Philip Wadler, Wen Kokke, and Jeremy G. Siek. 2020. *Programming Language Foundations in Agda*. http://plfa.inf.ed.ac.uk/20.07/

[14] David Walker. 2005. Substructural Type Systems. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). The MIT Press, Chapter 1, 3–44.

[15] The Agda Wiki. 2020. SimpleInductiveTypes. (Nov. 2020). https://wiki.portal.chalmers.se/agda/ReferenceManual/SimpleInductiveTypes?from=ReferenceManual.Datatypes#Strictpositivity