Delta Dictionaries

Total and Extensional Finite Maps in Proof Assistants

Anonymous
Anonymous
anonymous

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”)]

Figure 1. Dictionary representations after inserting the sequence of keys 6, 3, 1, and 3: { 1 ↦ “a” , 3 ↦ “b” , 6 ↦ “c” }

Abstract

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.

We develop a novel list-based representation, called *delta dictionaries*, that simultaneously achieves benefits of the conventional representations. Delta dictionaries are *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*. We present an implementation of delta dictionaries and relevant metatheory in Agda, and we discuss when delta dictionaries may or may not be preferable to conventional solutions.

CCS Concepts: • **Software and its engineering** → *Software verification*; *Software notations and tools*; • **Theory of computation** → *Logic and verification.*

Keywords: proof assistants, data structures, dictionaries

Woodstock ’18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/XXX

1 Introduction

Conventionally, the design of data structures and algorithms is chiefly focused on performance, which is of significantly diminished concern in proof assistants. 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.

Consider dictionaries, which—among a broad range of purposes—are often used for implementing type contexts and evaluation environments. The most basic representation for a dictionary, an *association list*, comprises a list of key-value pairs [10, Lists]. Association lists are simple to create, manipulate, and reason about. But 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, first two lines of Figure 1 show two distinct association lists (among others) to represent a dictionary with three particular bindings. This lack of ***Extensionality*** can make proofs more difficult [10, Maps] or impossible, especially when it comes to proving *contraction* and *exchange* [11] as discussed further in Section 3 and Section 4.2.

To establish a one-to-one correspondence between association lists and semantic mappings, one solution [8] 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 described in Figure 1 is one such example. This approach is Extensional, that is, it permits using built-in equality for semantic equivalence. However, this approach is not ***Semantically Total***; the underlying list type allows arbitrary, possibly invalid lists, so a proof of validity is required to refine the coarser type. The downsides of having to use validity proofs are discussed further in Section 3 and Section 4.2.

[1] The Coq standard library [7], describes unordered but deduplicated association lists as “weak,” implying that canonical association lists are “strong.”

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

Figure 2. Properties of dictionary representations.

A third conventional solution is to represent dictionaries as finite partial functions [10, Maps]. The third line of Figure 1 depicts a nested λ-expression that serves as the "lookup table" (we omit the "Some" constructors for space). Assuming *functional extensionality* [10, Logic]—which can be postulated in Coq and Agda without introducing a contradiction, and is a theorem in some newer proof assistants built on a cubical logical framework—this representation is also Extensional. However, a plain function type does not rule out terms that are *non*-finite maps, possibly requiring additional proofs of validity as with canonical association lists.[2] Worse are the facts that, unlike either association list representation, partial functions lack ***Decidable Equality*** and cannot be ***Destructed***. The function type could be refined with its domain—i.e. a canonical list of keys—but would then suffer the same drawbacks of validity proofs à la canonical association lists.

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 mechanization 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 the following properties:

1. ***Semantic Totality:*** Every term in the representation type is semantically valid, i.e. the mapping from terms to their semantic meanings is total.
2. ***Extensionality:*** Built-in equality corresponds to semantic equivalence, i.e. two unequal terms 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.

[2] This is often not a problem in practice: if not iterated or destructed, an infinite dictionary will work just as well, so a finiteness proof is unnecessary.

Figure 2 summarizes the preceding discussion along these dimensions; note that the association list representations can be easily destructed, but destruction is not possible for partial functions. None of the conventional representations satisfies both Semantic Totality and Extensionality, nor more than three of the four properties.

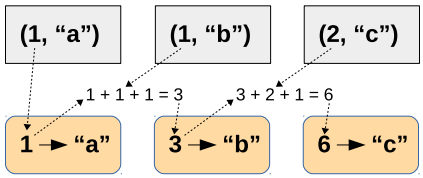
A New Representation. We offer a solution, dubbed *delta dictionaries*, which achieves the first three of the desired properties, and partially achieves the fourth. A delta dictionary can be described as a "canonical-by-construction" association list: instead of storing each literal key value, it stores the *difference* from the previous key, minus 1 (details in Section 2). For example, compare the canonical association list and delta dictionary for the Figure 1 example:

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

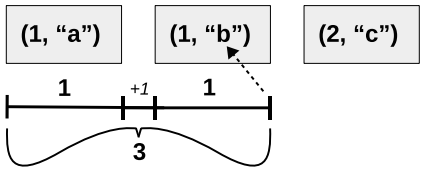
Every well-typed list-of-pairs is a valid delta dictionary (Semantic Totality), thus no proof term is needed to establish validity. Every unique delta dictionary represents a unique semantic mapping, thus built-in equality may be used for semantic equivalence (Extensionality). Furthermore, we define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

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 lone property among those we identify which the canonical association list does not. However, destruction and iteration for delta dictionaries is substantially more difficult. Furthermore, unlike all of the conventional representations, 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.

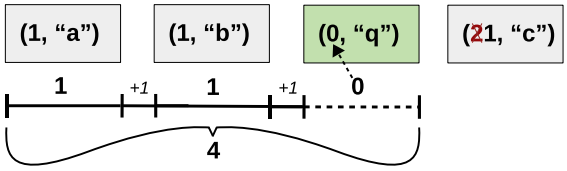
Outline. Next, we describe the core operations for delta dictionaries and the relevant metatheory in Section 2. In Section 3, we describe a small case study in proof development that demonstrates the necessity of delta dictionaries. Finally, we conclude in Section 4 with a discussion.



(a) An example delta dictionary.



(b) How do we find key 3?



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

Figure 3. Example delta dictionary operations.

2 Implementation

Our implementation of delta dictionaries and corresponding proofs are formalized in Agda and available in the anonymous supplementary materials.

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.

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. We call this number 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. 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 our implementation, each key is represented by a natural number, so that we can use addition, subtraction, and successor operations in order to define deltas. There may be an algebraic structure more general than the natural numbers which satisfies these requirements, but for practical purposes, working with naturals is easier for both the implementer and the client. In Agda, the bijection typeclass accepts a `convert` function (for clarity, the code in this paper uses the synonym `toNat`), as well as proofs that `convert` is injective and surjective, and the inverse function is defined by the library using the proof of surjectivity.[3] We demonstrate this in Agda 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 countable 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, we do not expose the ordering to the client—functions such as `destruct` and `dlt⇒list` 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. Notice that we parameterize the `Delta` module by key type `K` and bijection `bij` to the naturals; 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,

[3] 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
delta n<m =
  difference (n<m→1+n≤m n<m) -- i.e. m - n - 1

-- 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 n<hn       = None
    ... | Inr (Inl refl) = Some ha
    ... | Inr (Inr hn<n) = lkup t (delta 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 refl) =
          (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))
```

Figure 4. Delta dictionary operations.

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 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 [6, 7] describes key metatheory about dictionaries, including that for lookup and insert (it does not define `destruct`). Our Agda mechanization proves the relevant subset of this metatheory. Some of the metatheory in the Coq treatment involves concepts not relevant to our treatment, especially multiple distinct notions of equality. Also, since we do not expose mapping order to the client via our interface, we do not prove any of the theorems pertaining to ordering (i.e. those that are not "weak").

2.3 Additional Operations

Our Agda mechanization also defines key deletion, union, map, and to/from-list operations, along with appropriate metatheory (not reproduced here).

2.4 Properties

We now consider properties about delta dictionaries, starting with the four design goals identified in Section 1. Theorems below are proven in Agda. 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 refined with validity premises.

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))
```

We discuss why destruction is "not-so-easy" in Section 4.1.

Lastly, we prove analogs to the *structural properties contraction* and *exchange*.

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)
```

3 Case Study

To illustrate the practical importance of the four core properties, we 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 and exchange.

Figure 5 shows a sketch of this environment-based evaluation (`_⊢_⇒_`) of expressions (`exp`) to results (`res`). The definitions of evaluation environments (`env`), types, type contexts, 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 such 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. 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 partial functions. 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.

```
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)
```

Figure 5. Mechanization task requiring delta dictionaries.

4 Discussion

We conclude with a broader discussion of the design space, as well as related and future work.

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 `dd'`. To enable manual establishment of structural recursion, via an extra parameter that explicitly tracks the dictionary length, we provide another theorem:

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

Although possible, manually establishing termination is painful, especially given that it is not necessary for association lists or canonical association lists. Thus delta dictionaries fail at Easy Destructibility, although 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, we believe we have uncovered 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 a preferable.

4.2 Potential Practical Applications

Next, we provide additional discussion about the use of dictionaries in our case study, as well as programming language metatheory in general.

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 authors: to mechanize the formalization of AnonymousSystem [1], 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 our much larger mechanization [2] for AnonymousSystem. Although that mechanization has not been completed, the challenges faced during its development—described 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, AnonymousSystem 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.[4] 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 can be the source of less severe pain points in a broader range of circumstances. The ability to refine ordinary types

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

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 mechanizations 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 [3]. Linear type systems, on the other hand, require sensitivity to duplicate insertions, so duplication-insensitive dictionaries are inappropriate for them as well [11]. 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* [4, Library Firstorder] for an introduction.

For these reasons, many existing mechanizations make little use of order-and-duplication-insensitive dictionaries. However, as mechanization 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.

4.3 Related Work

How often are different dictionary representations used?

Conventional Representations. Due to their simplicity, 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* [10, Maps]—despite requiring a bit of extra overhead.

Canonical association lists seem to get little use, perhaps because working with refinement proofs might add more hassle than Extensionality alleviates. Canonical association lists are defined in the Coq standard library as `FMapList` [8]; 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` [9], on the other hand, turns up 4916 results, though most of these are probably unrelated to dictionaries.[5] de Amorim [5] 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, performance is likely to become an even greater concern than it is today. In cases where there is no extraction step, 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 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.

[5] 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.

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 terms and semantic meanings requires understanding of the graph isomorphism problem, which involves complex algebra.

References

[1] Anonymous. 20XX. AnonymousSystem. *Omitted for Review* (20XX).

[2] Anonymous. 20XX. AnonymousSystem. (20XX). http://omitted

[3] 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)*.

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

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

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

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

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

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

[10] 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.

[11] 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.