CSCI.6962/4962 Software Verification— Fundamental Proof Methods in Computer Science (Arkoudas and Musser)—Section 10.4

Instructor: Carlos Varela
Rensselaer Polytechnic Institute
Spring 2018

Fundamental Discrete Structures

Goal: to reason about ordered pairs, options, sets, relations, functions, and maps as fundamental data types in computer science, e.g., for modeling and verification of digital systems.

- *ordered pairs*
- *options*
- *sets, relations, and functions*
- maps

Representation and notation

- Finite maps are another pervasive data type and an indispensable tool in our mathematical arsenal for the modeling and analysis of discrete systems.
- Sets will be widely used in this theory.
  - For instance, the domain of a map will be defined to return a *set* of keys, while the range of a map will be a set of values.
  - We will use upper-case letters from the beginning of the alphabet as variables ranging over finite sets of arbitrary sorts:

```
define [A B C] := [?A:(Set.Set 'S1) ?B:(Set.Set 'S2) ?C:(Set.Set 'S3)]
```

- All of the code in this section appears inside a `Map` module that is part of the Athena library.

Representation and notation

- Maps are intended to represent finite functions from arbitrary keys to arbitrary values.
- Conventional algorithmic treatments focus on the computational complexity of the fundamental map operations in different implementations (red-black trees vs. hash tables, etc.).
- Here we are primarily interested in maps as an abstract data type (the so-called *dictionary* data type), so our focus will be on the specification and deductive analysis of some of the essential properties and relationships of the fundamental map operations.
- As usual, we will define these operations constructively, so we will be able to evaluate map insertions, deletions, and so on.
- But, as elsewhere, the emphasis will be on formal analysis rather than computational complexity.

Representation and notation

We introduce maps as a polymorphic inductive structure, abstracted over both the sort of the keys and that of the values:

```
structure (Map Key Value) := empty-map
                           | (update (Pair Key Value) (Map Key Value))
```

- Note that the general shape of this structure is also similar to that of lists, with `empty-map` playing the role of `nil` and `update` the role of `::` (the "cons" operation).
- The intuition here is that `update` is used to incrementally add new key-value bindings to a map, with the bindings represented as ordered pairs.
- These key-value pairs are added in front of the map, so, like lists, maps grow to the left.

Representation and notation

- The construction process always starts with the empty map.
- If a new key-value binding is added to a map m that shares a key with an already existing binding in m, then the new (leftmost) binding takes precedence.
- Thus, even though the older pair is not removed from the new map, it is shadowed by the more recent binding.
- This will become more precise soon, when we come to define how a map is applied to a given key.
- Also, because this is a structure rather than a datatype, we will have to precisely define what it means for two maps to be identical.
- As you might guess, two maps will be the same iff they produce the same values when applied to the same keys.

Representation and notation

- As with other sorts that are structurally similar to lists, it will be notationally convenient to have procedures that can convert Athena lists to maps and conversely.
- We start with a procedure for converting lists to `Map` terms.
- Key-value bindings are recognized as either two-element key-value lists `[k v]`, or, for enhanced notational effect, as three-element lists of the form `[k --> v]`:

```
define (alist->map L) :=
  match L {
    [] => empty-map
  | (list-of (|| [k --> v] [k v]) rest) => (update (pair k v)
                                                   (alist->map rest))
  | _ => L
  }
```

Representation and notation

The converse procedure will also come handy:

```
define (map->alist m) :=
  match m {
    empty-map => []
  | (update (pair k v) rest) => (add [k --> v] map->alist rest)
  | _ => m
  }
```

We also introduce a procedure for converting bindings given in such list notation to ordered pairs:

```
define (alist->pair inner-1 inner-2) :=
  lambda (L)
    match L  {
      [a b] =>     ((inner-1 a) @ (inner-2 b))
    | [a --> b] => ((inner-1 a) @ (inner-2 b))
    | _ => L
    }
```

Representation and notation

We now expand the input range of `update` so that it can receive bindings as lists of the above form in the first argument position; and lists representing maps in the above notation in the second position:

```
expand-input update [(alist->pair id id) alist->map]
```

Since we will be dealing with sets and set operations a good deal, we introduce a few abbreviations for their fully qualified names:

```
define [null ++ in subset \/ /\ \ -] :=
   [Set.null Set.++ Set.in Set.subset Set.\/ Set./\ Set.\ Set.-]
```

More importantly, we overload ++ so that it can be used as an alias for the `Map` constructor `update` in appropriate contexts:

```
overload ++ update
```

Representation and notation

Thus:

```
> (['a --> 1] ++ [])

Term: (update (pair 'a 1)
              empty-map:(Map Ide Int))

> (75 ++ [])

Term: (Set.insert 75
                  Set.null:(Set.Set Int))
```

We also transform the output of `eval` so that it prints out sets and maps as Athena lists:

```
transform-output eval [Set.set->alist map->alist]
```

Representation and notation

Finally, we introduce a few convenient variable names:

```
define [key key1 key2 k k' k1 k2] := [?key ?key1 ?key2 ?k ?k' ?k1 ?k2]

define [val val1 val2 v v' v1 v2 x x1 x2] :=
       [?val ?val1 ?val2 ?v ?v' ?v1 ?v2 ?x ?x1 ?x2]

define [m m1 m2 m3 rest rest1] :=
       [?m:(Map 'S1 'S2) ?m1:(Map 'S3 'S4) ...]

define [s1 s2 s3 A B C] := [Set.s1 Set.s2 Set.s3 Set.A Set.B Set.C]
```

Map operations and theorems

- We introduce the first operation on maps, which is also the most fundamental one: that of *applying* a map to a given key. This is also known as *looking up* the value associated with a key.
- And here we must decide what value to specify as the result when the given key does not exist in the map. Barring subsorting and explicit error elements, there are three main alternatives.
  - One is to return an arbitrary (but fixed) value.
  - Another is to say nothing on the matter—to leave the value of the lookup operation unspecified when the key is not in the map.
  - And another is to return an `Option` value of the form `(SOME` v`)` when the given key *is* bound to a value v in the map, and `NONE` otherwise.

Map operations and theorems

- Here we take the `Option` approach; we examine the first alternative later.

```
declare apply: (Key, Value) [(Map Key Value) Key] -> (Option Value)
                            [110 [alist->map id]]

define at := apply
```

We have specified an input converter for the first argument position that allows us to write the maps in the list notation we described earlier:

```
> ([['a --> 1] ['b --> 2] ['c --> 3]] at 'a)

Term: (apply (update (pair 'a 1)
                     (update (pair 'b 2)
                             (update (pair 'c 3)
                                     empty-map:(Map Ide Int))))
             'a)
```

Map operations and theorems

We now define map application as follows:

```
assert* apply-def :=
   [([] at _ = NONE)
    ([key val] ++ _ at x = SOME val    <== key = x)
    ([key _] ++ rest at x = rest at x <== key =/= x)]
```

We test the definition to make sure it behaves sensibly:

```
define ide-map := [['a --> 1] ['b --> 2] ['c --> 3] ['a --> 99]]

> (eval ide-map at 'a)
Term: (SOME 1)

> (eval ide-map at 'b)
Term: (SOME 2)

> (eval ide-map at 'foo)
Term: NONE: (Option Int)
```

Map operations and theorems

```
conclude apply-lemma-1 :=
  (forall key val rest x .
     [key val] ++ rest at x = NONE ==> rest at x = NONE)
   pick-any key val rest x
    let {m := ([key val] ++ rest);
         hyp := (m at x = NONE);
         goal := (rest at x = NONE)}
      assume hyp
        (!two-cases
          (!chain [(key = x)
                   ==> (m at x = SOME val)  [apply-def]
                   ==> (m at x =/= NONE)    [option-results]
                   ==> (hyp & ~hyp)         [augment]
                   ==> goal                 [prop-taut]])
          (!chain [(key =/= x)
                   ==> (m at x = rest at x) [apply-def]
                   ==>  (NONE = rest at x)  [hyp]
                   ==> goal                 [sym]]))
```

Map operations and theorems

As usual, the conditions of the defining equations of the operation give rise to a corresponding case analysis inside the proof.

Two more simple but useful lemmas concerning application follow:

```
define apply-lemma-2 :=
  (forall k v rest x .
     [k v] ++ rest at x =/= NONE <==> k = x | rest at x =/= NONE)

define apply-lemma-3 :=
  (forall m k v1 v2 . m at k = SOME v1 & m at k = SOME v2 ==> v1 = v2)
```

Map operations and theorems

We continue with an operation that removes a key from a map:

```
declare remove: (Key, Value) [(Map Key Value) Key] -> (Map Key Value)
                             [- 120 [alist->map id]]

left-assoc -

assert* remove-def :=
  [(empty-map - _ = empty-map)
   ([key _] ++ rest - key = rest - key)
   (key =/= x ==> [key val] ++ rest - x = [key val] ++ (rest - x))]
```

- Observe the second equation: A recursive call is needed even with the key at the front of the map, to remove all occurrences of the key further inside the map.
- Also note that `-` associates to the left, so that we can write something like `(m - k1 - k2)` to mean `((m - k1) - k2)`.

Map operations and theorems

```
> (eval ide-map - 'a)

List: [['b --> 2] ['c --> 3]]

> (eval ide-map - 'a - 'c)

List: [['b --> 2]]

> (eval ide-map - 'a - 'b at 'c)

Term: (SOME 3)
```

The following states that the removal operation achieves the desired effect:

```
define remove-correctness :=
  (forall m x . m - x at x = NONE)
```

Map operations and theorems

A proof by induction follows (basis case):

```
by-induction remove-correctness {
  empty-map =>
    pick-any x
      (!chain [([] - x at x)
               = ([] at x)                          [remove-def]
               = NONE                               [apply-def]])
| (m as (update (pair key val) rest)) =>
    ...
}
```

Map operations and theorems

Inductive step follows:

```
by-induction remove-correctness {
  ...
| (m as (update (pair key val) rest)) =>
    let {IH := (forall x . rest - x at x = NONE)}
      pick-any x
        (!two-cases
           assume case1 := (key = x)
             (!chain [(m - x at x)
                      = (m - key at key)                [case1]
                      = (rest - x at x)                 [case1 remove-def]
                      = NONE                            [IH]])
           assume case2 := (key =/= x)
             (!chain [(m - x at x)
                      = ([key val] ++ (rest - x) at x) [remove-def]
                      = (rest - x at x)                 [apply-def]
                      = NONE                            [IH]]))
}
```

Map operations and theorems

This is only one half, the important one, of `remove`'s correctness. We also need to prove that removing a key from a map does not affect any of the other keys:

```
define remove-correctness-2 :=
  (forall m k k' . k =/= k' ==> m - k at k' = m at k')
```

Map operations and theorems

Next, we define the *domain* of a map m as the set of all keys in m:

```
declare dom: (Key, Value) [(Map Key Value)] -> (Set.Set Key) [[alist->map]]

assert* dom-def :=
  [(dom empty-map = null)
   (dom [k _] ++ rest = k ++ dom rest)]

> (eval dom ide-map)

List: ['a 'b 'c]
```

Observe the overloaded use of ++: On the left-hand side, ++ is used as an alias for `update`, a map operation; whereas on the right-hand side it is used as an alias for `Set.insert`, a set operation.

Map operations and theorems

We define the *size* of a map m as the cardinality of its domain, namely, the number of keys in m:

```
declare size: (S, T) [(Map S T)] -> N [[alist->map]]

assert* size-def := [(size m = Set.card dom m)]

transform-output eval [nat->int]

> (eval size ide-map)

Term: 3

> (eval size ide-map - 'a - 'b)

Term: 1
```

Map operations and theorems

A few results about domains follow. The first two are simple lemmas, but the third is a more fundamental characterization theorem; the fourth one is a useful property of the domain of a map obtained from a key removal.

```
define dom-lemma-1 := (forall k v rest . k in dom [k v] ++ rest)

define dom-lemma-2 := (forall m k v . dom m subset dom [k v] ++ m)

define dom-characterization :=
  (forall m k . k in dom m <==> m at k =/= NONE)

define dom-lemma-3 := (forall m k . dom (m - k) subset dom m)
```

Map operations and theorems

A proof of domain characterization by induction (basis case):

```
conclude dom-characterization :=
  (forall m k . k in dom m <==> m at k =/= NONE)
by-induction dom-characterization {
  (m as empty-map) =>
    pick-any k
      (!equiv
        (!chain [(k in dom m)
                 ==> (k in null)                    [dom-def]
                 ==> false                          [Set.NC]
                 ==> (m at k =/= NONE)              [prop-taut]])
       assume hyp := (m at k =/= NONE)
         (!chain-> [true
                    ==> (m at k = NONE)             [apply-def]
                    ==> (m at k = NONE & hyp)       [augment]
                    ==> false                       [prop-taut]
                    ==> (k in dom m)                [prop-taut]]))
| (m as (update (pair x y) rest)) =>  ...  }
```

Map operations and theorems

And the inductive step:

```
conclude dom-characterization :=
  (forall m k . k in dom m <==> m at k =/= NONE)
by-induction dom-characterization {
  ...
| (m as (update (pair x y) rest)) =>
   let {IH := (forall k . k in dom rest <==> rest at k =/= NONE)}
     pick-any k
       (!chain [(k in dom m)
            <==> (k in x ++ dom rest)          [dom-def]
            <==> (k = x | k in dom rest)       [Set.in-def]
            <==> (k = x | rest at k =/= NONE)  [IH]
            <==> (x = k | rest at k =/= NONE)  [sym]
            <==> (m at k =/= NONE)             [apply-lemma-2]])
}
```

Map operations and theorems

We now introduce an operation that converts a map into a set of ordered pairs. This operation will be used to define identity conditions for maps:

```
declare map->set: (K, V) [(Map K V)] -> (Set.Set (Pair K V)) [[alist->map]]

assert* map->set-def :=
  [(map->set empty-map = null)
   (map->set [k v] ++ rest = [k v] ++ map->set (rest - k))]

> (eval map->set ide-map)

List: [(pair 'a 1) (pair 'b 2) (pair 'c 3)]
```

Map operations and theorems

We now define two maps to be identical iff they are identical as sets:

```
assert* map-identity := (m1 = m2 <==> map->set m1 = map->set m2)
```

We can immediately put the identity conditions to the test:

```
> let {map1 := (alist->map [['b --> 2] ['a --> 1]]);
       map2 := (alist->map [['a --> 1] ['b --> 2] ['b --> 3]])}
   (eval map1 = map2)

Term: true

> let {map1 := (alist->map [['b --> 2] ['a --> 1]]);
       map2 := (alist->map [['b --> 2]])}
   (eval map1 = map2)

Term: false
```

Map operations and theorems

A major characterization theorem needs to be proved at this point, namely, that two maps are identical in the sense we just described iff they are identical in the intuitive sense, that is, iff they produce identical outputs when applied to identical inputs:

```
define identity-characterization :=
  (forall m1 m2 . m1 = m2 <==> forall k . m1 at k = m2 at k)
```

This is fairly straightforward to derive once a closely related characterization of `map->set` has been derived:

```
define map->set-characterization :=
  (forall m k v . k @ v in map->set m <==> m at k = SOME v)
```

Map operations and theorems

Let us now define the *range* of a map as follows:

```
declare range: (K, V) [(Map K V)] -> (Set.Set V) [[alist->map]]

assert* range-def := [(range m = Set.range map->set m)]

> (eval range ide-map)

List: [1 2 3]
```

Map operations and theorems

Most of the results we derived about domains have natural analogues for ranges, albeit with some important differences.

```
conclude range-lemma-1 :=
  (forall m v . v in range m <==> exists k . k @ v in map->set m)
pick-any m v
  (!chain [(v in range m)
       <==> (v in Set.range map->set m)       [range-def]
       <==> (exists k . k @ v in map->set m) [Set.range-characterization]])

conclude range-characterization :=
  (forall m v . v in range m <==> exists k . m at k = SOME v)
pick-any m v
  (!chain [(v in range m)
       <==> (exists k . k @ v in map->set m) [range-lemma-1]
       <==> (exists k . m at k = SOME v)    [map->set-characterization]])
```

Map operations and theorems

```
conclude range-lemma-2 :=
   (forall k v rest . v in range [k v] ++ rest)
pick-any k v rest
  (!chain<- [(v in range [k v] ++ rest)
         <== (v in Set.range map->set [k v] ++ rest)     [range-def]
         <== (v in Set.range k @ v ++ map->set rest - k) [map->set-def]
         <== (v in v ++ Set.range map->set rest - k)     [Set.range-def]
         <== (v = v | v in Set.range map->set rest - k)  [Set.in-def]
         <== (v = v)                                     [alternate]])
```

Map operations and theorems

A subtle difference is that the analogue of `dom-lemma-2` does *not* hold:

```
define range-lemma-conjecture :=
  (forall m k v . range m subset range [k v] ++ m)

> (falsify range-lemma-conjecture 10)

List: ['success
|{
?k:Int := 1
?m:(Map Int Int) := (update (pair 1 1)
                            empty-map:(Map Int Int))
?v:Int := 0
}|]
```

This should not be surprising. A key can always be rebound to a different value in a map, thereby possibly removing the previous value from the range of the map (assuming that no other key maps to that same value).

Map operations and theorems

We continue with an operation that *restricts* a map to a given set of keys, using the symbol |^ as an infix alias for it:

```
declare restricted-to: (S, T) [(Map S T) (Set.Set S)] -> (Map S T)
                              [|^ 120 [alist->map Set.alist->set]]

assert* restrict-def :=
   [(empty-map |^ _ = empty-map)
    (k in A ==> [k v] ++ rest |^ A = [k v] ++ (rest |^ A))
    (~ k in A ==> [k v] ++ rest |^ A = rest |^ A)]

> (eval ide-map |^ ['b 'c])

List: [['b --> 2] ['c --> 3]]
```

Map operations and theorems

We state that the domain of any restriction of a map m to a set of restricted keys A is always a subset of A:

```
define restriction-theorem-1 := (forall m A . dom m |^ A subset A)

define restriction-theorem-2 :=
  (forall m A . dom m subset A ==> m |^ A = m)
```

Map operations and theorems

We close this section with two more useful binary operations on maps: *overriding* and *composition*.

Overriding is like union except that preference is given to the values of the rightmost map. In other words, applying the operation of overriding to two maps m_1 and m_2 results in a map that contains all and only the bindings of both maps, except that for bindings with a shared key k, preference is given to m_2 (the rightmost operand), meaning that the value associated with k in the resulting map is the same value that m_2 associates with k. We use the symbol ** as an alias for this operation:

```
declare override: (S, T) [(Map S T) (Map S T)] -> (Map S T)
                         [** [alist->map alist->map]]

assert* override-def :=
  [(m ** [] = m)
   (m ** [k v] ++ rest = [k v] ++ (m ** rest))]
```

Map operations and theorems

```
> (eval [[1 --> 'a] [2 --> 'b]] ** [[1 --> 'foo] [3 --> 'c]])

List: [[1 --> 'foo] [3 --> 'c] [1 --> 'a] [2 --> 'b]]
```

While the recursive definition of overriding basically stipulates that the empty map is a right identity element for the operation, we can readily prove that it is also a left identity element:

```
by-induction (forall m . [] ** m = m) {
  (m as empty-map) =>
    (!chain [(empty-map ** m) = empty-map [override-def]])
| (m as (update (pair k v) rest)) =>
    (!chain [(empty-map ** m)
             = ([k v] ++ (empty-map ** rest))  [override-def]
             = ([k v] ++ rest)                 [([] ** rest = rest)]])
}
```

Map operations and theorems

The following states that domain restriction distributes over **:

```
(forall m2 m1 A . (m1 ** m2) |^ A = m1 |^ A ** m2 |^ A).
```

Map operations and theorems

One way to define the composition of a map m_2 with a map m_1 is as follows: For every key k in the domain of m_1, we retrieve the corresponding value v and then apply m_2 to it, provided that v is in the domain of m_2; if not, we ignore the binding of k to v. For instance, if m_1 and m_2 are, respectively, the following maps:

```
[['paris --> 'france] ['tokyo --> 'japan] ['cairo --> 'egypt]]
```

and

```
[['france --> 'europe] ['algeria --> 'africa] ['japan --> 'asia]
```

then composing m_2 with m_1 would yield:

```
[['paris --> 'europe] ['tokyo --> 'asia]].
```

Note that the binding `['cairo --> 'egypt]` is not included in the result, nor are any bindings from m_2 itself.

Map operations and theorems

The following is a constructive definition of the compose operation; we use o as an infix alias for it.

```
declare compose: (S1, S2, S3) [(Map S2 S3) (Map S1 S2)] -> (Map S1 S3)
                              [o [alist->map alist->map]]

assert* compose-def :=
  [(_ o empty-map = empty-map)
   (m o [k v] ++ rest = [k v'] ++ (m o rest) <== m at v = SOME v')
   (m o [k v] ++ rest = m o rest <== m at v = NONE)]
```

Map operations and theorems

It is not necessary for both maps to share the sorts of their keys and values. All that is needed is for the sort of the values of the first map to be the same as the sort of the keys of the second map. Thus:

```
> let {m1 := [[1 --> 'a] [2 --> 'b] [1 --> 'c]];
       m2 := [['a --> true] ['b --> false] ['foo --> true]]}
    (eval m2 o m1)
List: [[1 --> true] [2 --> false]]

define capitals :=
  [['paris --> 'france] ['tokyo --> 'japan] ['cairo --> 'egypt]]

define countries :=
  [['france --> 'europe] ['algeria --> 'africa] ['japan --> 'asia]]

> (eval countries o capitals)
List: [['paris --> 'europe] ['tokyo --> 'asia]]
```

Map operations and theorems

Composition is not commutative:

```
> (falsify (forall m1 m2 . m1 o m2 = m2 o m1) 10)

List: ['success
|{
?m1:(Map Int Int) := (update (pair 1 1)
                             empty-map:(Map Int Int))
?m2:(Map Int Int) := (update (pair 1 2)
                             empty-map:(Map Int Int))
}|]
```

Map operations and theorems

Composition is not associative:

```
> (falsify (forall m1 m2 m3 . m1 o (m2 o m3) = (m1 o m2) o m3) 10)

List: ['success
|{
?m1:(Map Int Int) := (update (pair 1 1)
                             empty-map:(Map Int Int))
?m2:(Map Int Int) := (update (pair 1 2)
                             (update (pair 1 1)
                                     empty-map:(Map Int Int)))
?m3:(Map Int Int) := (update (pair 1 1)
                             empty-map:(Map Int Int))
}|]
```

Default maps

- An alternative approach to formalizing maps is that all keys are initially mapped to a default value.
- One advantage of this approach is that it maps keys to values directly, rather than to value options: If a key is not in the map's domain, the lookup operation simply returns the map's default value (say `zero`, if the values are natural numbers).
- They are defined in Athena's library module `DMap`.
- Default maps are defined as the following polymorphic inductive structure, abstracted over the sort variables `K` and `V` (the sorts of keys and values, respectively):

```
structure (DMap K V) := (empty-map V) | (update (Pair K V) (DMap K V))

set-precedence empty-map 250
```

Default maps

Note that this is virtually identical to the `Map` structure, with one important difference:

- The irreflexive constructor `empty-map`, which gets a relatively high precedence, is now unary rather than nullary (a constant).
- The sole argument of `empty-map` is a value—an element of sort `V`—that represents the default value that will be produced when a key is not present in the map.

Other than that, these maps work as before:

- `update` incrementally adds new bindings to a map, a binding being simply an ordered pair of a key and a value.
- These maps also grow to the left.
- A new binding added by `update` might shadow an older binding with the same key.

Default maps

The input expander is `alist->dmap` ("Athena list to default map") and the output transformer is `dmap->alist` ("default map to Athena list").
In Athena (bracket) list notation, a default map is represented as a two-element list

$$[d \;\; pairs]$$

where d is the default value of the map and *pairs* is a list of pairs representing the map's bindings (with bindings on the left taking precedence).
Each binding is written as either $[k \;\; v]$ or as $[k \;\; \texttt{-->} \;\; v]$.

Default maps

Thus, we have:

```
define dm := (alist->dmap [0 [['a --> 1] ['b --> 2] ['a --> 99]]])

> dm

Term: (DMap.update (pair 'a 1)
                   (DMap.update (pair 'b 2)
                                (DMap.update (pair 'a 99)
                                             (DMap.empty-map 0))))

> (dmap->alist dm)

List: [0 [['a --> 1] ['b --> 2] ['a --> 99]]]
```

Default maps

We issue some directives to make update more notationally flexible and define a number of variable abbreviations:

```
expand-input update [(alist->pair id id) alist->dmap]
overload ++ update
set-precedence ++ 210

define [key k k' k1 k2 d d' val v v' v1 v2] := [?key ... ]
define [h t A B] := [Set.h Set.t Set.A Set.B]
define [m m' m1 m2 rest] := [?m:(DMap 'S1 'S2) ... ]
```

Default maps

The chief operation is of course key lookup:

```
declare apply: (K, V) [(DMap K V) K] -> V [110 [alist->dmap id]]

define at := apply

assert* apply-def :=
  [(empty-map d at _ = d)
   ([k v] ++ rest at x = v <== k = x)
   ([k v] ++ rest at x = rest at x <== k =/= x)]
```

Default maps

Let's test the definition:

```
define dm := (alist->dmap [0 [['a --> 1] ['b --> 2] ['c --> 3]]])

> (eval dm at 'a)

Term: 1

> [(eval dm at 'b) (eval dm at 'c)]

List: [2 3]

> (eval dm at 'foo)

Term: 0
```

Default maps

Let's define a function to get the default value of any map:

```
declare default: (K, V) [(DMap K V)] -> V [200 [alist->dmap]]

assert* default-def :=
  [(default empty-map d = d)
   (default _ ++ rest = default rest)]

> (eval default dm)

Term: 0
```

Default maps

Key removal is defined as follows:

```
declare remove: (S, T) [(DMap S T) S] -> (DMap S T)
                       [- 120 [alist->dmap id]]

left-assoc -

assert* remove-def :=
  [(empty-map d - _ = empty-map d)
   ([key _] ++ rest - key = rest - key)
   (key =/= x ==> [key val] ++ rest - x = [key val] ++ (rest - x))]

> (eval dm - 'b)

Term: (DMap.update (pair 'a 1)
                   (DMap.update (pair 'c 3)
                                (DMap.empty-map 0)))
```

Default maps

The domain of a default map is a bit more subtle. We count only those keys that are bound to a value other than the map's default value. We make sure that subsequent bindings of the key are removed:

```
declare dom: (K, V) [(DMap K V)] -> (Set.Set K) [[alist->dmap]]

assert* dom-def :=
  [(dom empty-map _ = null)
   (dom [k v] ++ rest = dom (rest - k) <== v = default rest)
   (dom [k v] ++ rest = k ++ dom rest <== v =/= default rest)]

define dm :=
   (alist->dmap [0 [['a --> 1] ['b --> 0] ['c --> 3] ['b --> 99]]])

> (eval dom dm)
Term: (Set.insert 'a
                  (Set.insert 'c
                              Set.null:(Set.Set Ide)))
```

Default maps

The size of a default map is defined as the cardinality of its domain:

```
declare size: (S, T) [(DMap S T)] -> N [[alist->dmap]]

assert* size-def := [(size m = Set.card dom m)]
```

Default maps

We list some pertinent results below:

```
define remove-correctness-1 := (forall m x . m - x at x = default m)

define remove-correctness-2 :=
  (forall m k x . k =/= x ==> m - k at x = m at x)

define remove-correctness-3 :=
  (forall m k . default m = default m - k)

define dom-lemma-1 :=
  (forall k v rest . v =/= default rest ==> k in dom [k v] ++ rest)

define dom-lemma-2 :=
  (forall m k v . v =/= default m ==> dom m subset dom [k v] ++ m)

define dom-lemma-2b :=
  (forall m x k v . v =/= default m & x in dom m ==> x in dom [k v] ++ m)
```

Default maps

In addition to size, a very simplistic notion of map length is also useful for strong inductive proofs:

```
define [< <=] := [N.< N.<=]

declare len: (K, V) [(DMap K V)] -> N [[alist->dmap]]

assert* len-def :=
  [(len empty-map _ = zero)
   (len [_  _] ++ rest = S len rest)]

define len-lemma-1 := (forall m k v . len m < len (k @ v) ++ m)
define len-lemma-2 := (forall m k . len m - k <= len m)
define len-lemma-3 := (forall key val k rest .
                         len rest - k < len key @ val ++ rest)

define lemma-D := (forall m k . k in dom m <==> m at k =/= default m)
```

Default maps

```
define remove-correctness-0 := (forall m x . ~ x in dom m - x)

define dom-lemma-3 := (forall m k . dom m - k subset dom m)

define dom-corollary-1 :=
  (forall key val k rest  .
    k in dom rest - key ==> k in dom [key val] ++ rest)
```

Default maps

The following will extract a set of key-value pairs from a map. Note that, consistent with the definition of dom, we only include those bindings whose values are different from the map's default value:

```
declare dmap->set: (K, V) [(DMap K V)] -> (Set.Set (Pair K V))
                          [[alist->dmap]]

assert* dmap->set-def :=
  [(dmap->set empty-map _ = null)

   (dmap->set k @ v ++ rest = dmap->set rest - k
           <== v = default rest)

   (dmap->set k @ v ++ rest = (k @ v) ++ dmap->set rest - k
           <== v =/= default rest)]
```

Default maps

The following are useful results:

```
define dms-lemma-1 := (forall k v . k @ v in dmap->set m ==> k in dom m)

define dms-lemma-1b :=
   (forall m k . ~ k in dom m ==> forall v . ~ k @ v in dmap->set m)

define dms-lemma-1b' :=
  (forall m k . ~ k in dom m ==> ~ exists v . k @ v in dmap->set m)

define dms-theorem-1 :=
  (forall m k v . k @ v in dmap->set m ==> m at k = v)

define dms-theorem-2 :=
  (forall m k . ~ k in dom m ==> m at k = default m)

define dms-lemma-2 :=
  (forall m k k' . k in dom m & k =/= k' ==> k in dom m - k')
```

Default maps

```
define dms-lemma-3 :=
  (forall m key val .
    val =/= default m ==> dom key @ val ++ m = key ++ dom m - key)

define dms-theorem-3 :=
  (forall m k . k in dom m ==> exists v . k @ v in dmap->set m)

define at-characterization :=
  (forall m k v . m at k = v <==> k @ v in dmap->set m |
                                  ~ k in dom m & v = default m)
```

Default maps

We now assert the following identity conditions for default maps:

```
assert* dmap-identity :=
 (forall m1 m2 . m1 = m2 <==> default m1 = default m2 &
                              dmap->set m1 = dmap->set m2)
```

In other words, two maps are identical iff (a) their default values are the same, and (b) they have the exact same sets of bindings, as these are computed by `dmap->set`.

Default maps

And a restriction operation:

```
declare restricted-to: (S, T) [(DMap S T) (Set.Set S)] -> (DMap S T)
                              [150 |^ [alist->dmap Set.alist->set]]

assert* restrict-def :=
   [(empty-map d |^ _ = empty-map d)
    (k in A ==> [k v] ++ rest |^ A = [k v] ++ (rest |^ A))
    (~ k in A ==> [k v] ++ rest |^ A = rest |^ A)]
```

Default maps

As a nonconstructive characterization of identity, we state that two maps are identical iff their default values are the same and they map *all* keys to the same values:

```
define identity-characterization :=
  (forall m1 m2 . m1 = m2 <==> default m1 = default m2 &
                               forall k . m1 at k = m2 at k)
```

The explicit presence of the condition

$$\texttt{default m1 = default m2} \tag{1}$$

might seem somewhat odd. After all, one might expect that condition to be redundant in view of the seemingly much stronger condition

$$\texttt{forall k . m1 at k = m2 at k.} \tag{2}$$

Default maps

But that is actually not the case when there are only finitely many possible keys and the domain of the maps includes all of them.
For example, consider:

```
define m1 :=  (alist->dmap [0 [[true --> 1] [false --> 0]]])
define m2 :=  (alist->dmap [9 [[true --> 1] [false --> 0]]])
```

For these two maps, with `Boolean` keys, condition (2) holds but (1) does not, and hence the two are not identical (according to `dmap-identity`). However, when there are infinitely many possible keys then (2) does entail (1).