MiniData Test

Mini Data Library
Brian Beckman
Version of 10 Feb 2012

Definitions

An **[association list]** is a list of **[attributes]**, that is, **[key]**-**[value]** pairs (or, perhaps later, pairs of keys and value-**[type]** pairs), where the keys are always strings (or symbols? or arbitrary hashables? UNDONE) and the values can be anything. E.g., `{{"foo", 1}, {"bar", 2}}`. No duplicate keys allowed, order is not significant. They're really enumerable sets, but realized as lists for convenience. Use *lookup[alist, key]* to get the value associated with *key* from the association list *alist*.

An **[association vector]** is a partial function from strings to values (or value-type pairs, or perhaps symbols (UNDONE)). Expressions for looking up values by key have the form of function calls, that is, if *v* is an association vector and *k* is a key, write the associated value as *v*[*k*]. Association lists and association vectors are in one-to-one correspondence because the domain, strings, is enumerable. Functions are single-valued, meaning that each input string maps to a unique output value, implying that association lists may not have duplicate keys.

There difference between an attribute and a **[property]**, which is also a key-value pair, is that property getters and setters may do significant computation, but attribute setters and getters never do. For instance, a representation for complex numbers may store the real and imaginary parts, and also have properties for getting and setting the modulus and argument, related to the real and imaginary types by coordinate transforms.

Association vectors are the primary vehicle for structured storage. Association lists are convenient for presentation and for some manipulations. Since they are identical in form and function, use the term **[dictionary]** to refer generically to these two kinds of data structure.

The implementation automatically stringulates arguments of type <*key*>.

```
IndexOfMaxs[<list>, <valueFunction>] --> <integerIndex>
IndexOfMins[<list>, <valueFunction>] --> <integerIndex>
IndexMaxs[<list>, <valueFunction>] --> {<integerIndex>,<maxValue>}
IndexMins[<list>, <valueFunction>] --> {<integerIndex>,<minValue>}
al2Av[<aList>] --> <aVec>
lookup[<aList>,<key>] --> <value>
lookupWithDefault[<aList>,<key>,<default>] --> <value> or <default>
containsKey[<aList>, <key>] --> <bool>
removeKey[<aList>, <key>] --> <aVec>
fetchAttribute[<aList>, <key>] --> {<key>, <value>}
adjoin[<aList>,           <key>,           <value>]           -->           <aList>
adjoin[<aList>,                 <aList>]                 -->                 <aList>
getSchema[<aList>]                  -->                {<key>,                  ...}
schemaAndRecord2Al[<schema>, <record>] --> <aList>
```

```
sieve[<aList>, <keyList>] --> {<aList(matchingKeys)>, <aList(notMatchingKeys)>}

// <aVec>[<key>] --> <value>
// lookup[<aVec>,<key>] --> <value>
// lookupWithDefault[<aVec>,<key>,<default>] --> <value> or <default>
// containsKey[<aVec>, <key>] --> <bool>
// removeKey[<aVec>, <key>] --> <aVec>
// fetchAttribute[<aVec>, <key>] --> {<key>, <value>}
// copyAv[<aVec>] --> <aVec>
// equalAvs[<aVec1>, <aVec2>] --> <bool>
//          adjoin[<aVec>,          <key>,          <value>]          -->          <aVec>
//              adjoin[<aVec>,              <aList>]              -->              <aVec>
//              adjoin[<aVec>,              <aVec>]              -->              <aVec>
// av2Al[<aVec>] --> {{<key>, <value>}, ...}
//              getSchema[<aVec>]              -->              {<key>,              ...}
// schemaAndRecord2Av[<schema>, <record>] --> <aVec>
// sieve[<aVec>, <keyList>] --> {<aVec(matchingKeys)>, <aVec(notMatchingKeys)>}
```

In[1]:=
```
Get["MiniData`"];
```

In[2]:=
```
$Packages
```

Out[2]=
```
{MiniData`, ResourceLocator`, DocumentationSearch`,
  GetFEKernelInit`, JLink`, PacletManager`, WebServices`, System`, Global`}
```

In[3]:=
```
?MiniData`*
```

▼ MiniData`

adjoin	equalAvs	isAv
al2Av	fetchAttribute	lookup
ArgIndexMaxs	getSchema	lookupWithDefault
ArgIndexMins	IndexMaxs	removeKey
as2Al	IndexMins	schemaAndRecord2Al
av2Al	IndexOfMaxs	schemaAndRecord2Av
containsKey	IndexOfMins	sieve
copyAv	isAl	

In[4]:=
```
?isAl
```

```
isAl[<object>] --> <bool> or non-reduce
```

In[5]:=
```
On[Assert];
```

In[6]:=
```
Assert @isAl[{{"foo", 42}}]
```

In[7]:=
```
Assert@Not@Block[{z},
    z = al2Av[{{"foo", 42}}];
    isAl[z]]
```

In[8]:=
```
Assert@Not@Block[{z},
    z["a"] = 1;
    isAl[z]]
```

In[9]:=
```
? al2Av
```

al2Av[<aList>] --> <aVec>

In[10]:=
```
al2Av[32]
```

Out[10]= 32

In[11]:=
```
isAv@(al2Av@{{"foo", 42}})
```

Out[11]= True

In[12]:=
```
Head@al2Av@{{"foo", 42}}
```

Out[12]= Symbol

In[13]:=
```
(DownValues@Evaluate@al2Av@{{"foo", 42}})[[1, 1, 1, 1]]
```

Out[13]= foo

In[14]:=
```
(DownValues@Evaluate@al2Av@{{"foo", 42}})[[1, 2]]
```

Out[14]= 42

In[15]:=
```
? equalAvs
```

equalAvs[<aVec>,<aVec>] --> bool

In[16]:=
```
? lookup
```

lookup[<aList|aVec>,<key>] --> <value>, <aVec>[<key>] --> <value>

In[17]:=
```
? lookupWithDefault
```

lookupWithDefault[<aList|aVec>,<key>,<default>] ––> <value> or <default>

In[18]:=
```
? containsKey
```

containsKey[<aList|aVec>,<key>] ––> <bool>

In[19]:=
```
? removeKey
```

removeKey[<aList|aVec>,<key>] ––> <aList> or <aVec>

Unit Testing the Package

IndexOfMaxs, IndexOfMins

In[20]:=
```
IndexOfMaxs[{2, 3, 1}]
```

Out[20]= 2

In[21]:=
```
IndexOfMaxs[{2, 3, 1}, # &]
```

Out[21]= 2

In[22]:=
```
IndexOfMaxs[{2, 3, 1}, -# &]
```

Out[22]= 3

In[23]:=
```
IndexOfMaxs[Range[1000]]
```

Out[23]= 1000

In[24]:=
```
IndexOfMins[{2, 3, 1}]
```

Out[24]= 3

In[25]:=
```
IndexOfMins[Range[1000], Sin]
```

Out[25]= 344

In[26]:=
```
IndexOfMins[Range[1000], (-#) &]
```
Out[26]=
```
1000
```

In[27]:=
```
IndexOfMins[RandomSample[Range[1000]]]
```
Out[27]=
```
832
```

In[28]:=
```
IndexMaxs[RandomSample[Range[1000]]]
```
Out[28]=
```
{31, 1000}
```

The current implementation is fatally limited.

In[29]:=
```
$IterationLimit
```
Out[29]=
```
4096
```

In[30]:=
```
IndexOfMins[{1, 2, 3}]
```
Out[30]=
```
1
```

In[31]:=
```
IndexOfMins[{3, 2, 2, 1, 1, 3}]
```
Out[31]=
```
5
```

In[32]:=
```
Block[{$IterationLimit = 4096}, IndexOfMins[{3, 2, 2, 1, 1, 3}]]
```
Out[32]=
```
5
```

In[33]:=
```
Block[{$IterationLimit = 20 000}, IndexOfMins[{3, 2, 2, 1, 1, 3}]]
```
Out[33]=
```
5
```

In[34]:=
```
Block[{$IterationLimit = 20 000}, IndexOfMins[RandomSample[Range[10 000]]]]
```
Out[34]=
```
6349
```

In[35]:=
```
Block[{$IterationLimit = 20 000}, IndexOfMins[Range[10 000]]]
```
Out[35]=
```
1
```

In[36]:=
```
$IterationLimit
```
Out[36]=
4096

In[37]:=
```
IndexOfMins[{}]
```
Out[37]=
0

In[38]:=
```
IndexOfMaxs[{}, Sin]
```
Out[38]=
0

- **(*isAl*) Is Association List, (*isAv*) Is Association Vector**
- **(*al2Av*) Association List To Association Vector**

In[39]:=
```
z = al2Av[{{"name", Unique[]}, {"lon", 100}, {"lat", 100}}];
```

In[40]:=
```
z["name"] // FullForm
```
Out[40]//FullForm=
$3

In[41]:=
```
isAl[z]
```
Out[41]=
False

In[42]:=
```
? equalAvs
```
equalAvs[<aVec>,<aVec>] --> bool

In[43]:=
```
equalAvs[z, al2Av@{{"name", Unique[]}, {"lon", 100}, {"lat", 100}}]
```
Out[43]=
False

In[44]:=
```
equalAvs[z, z]
```
Out[44]=
True

In[45]:=
```
? copyAv
```
copyAv[<aVec>] --> <aVec>

In[46]:=
```
equalAvs[z, copyAv@z]
```
Out[46]=
True

In[47]:=
```
z2 = copyAv@z
```
Out[47]=
MiniData`Private`d$983

In[48]:=
```
av2Al@z
```
Out[48]=
$$\begin{pmatrix} \text{lat} & 100 \\ \text{lon} & 100 \\ \text{name} & \$3 \end{pmatrix}$$

In[49]:=
```
lookup[z, "bar"]
```

In[50]:=
```
z["bar"]
```
Out[50]=
MiniData`Private`d$956(bar)

In[51]:=
```
lookup[av2Al@z, "bar"]
```

In[52]:=
```
z["name"] = Unique[]
```
Out[52]=
$5

In[53]:=
```
equalAvs[z, z2]
```
Out[53]=
False

In[54]:=
```
isAv[z]
```
Out[54]=
True

In[55]:=
```
z["lon"]
```
Out[55]=
100

In[56]:=
```
z
```
Out[56]=
MiniData`Private`d$956

In[57]:=
```
Evaluate@z
```
Out[57]=
```
MiniData`Private`d$956
```

In[58]:=
```
DownValues@z
```
Out[58]=
```
{}
```

In[59]:=
```
DownValues[Evaluate@z]
```
Out[59]=
```
{HoldPattern[MiniData`Private`d$956(lat)] :→ 100,
  HoldPattern[MiniData`Private`d$956(lon)] :→ 100, HoldPattern[MiniData`Private`d$956(name)] :→ $5}
```

(*av2Al*) Association Vector to Assoction List

In[60]:=
```
av2Al[Unique[]]
```
Out[60]=
```
{}
```

In[61]:=
```
av2Al[{}]
```
Out[61]=
```
{}
```

In[62]:=
```
?? av2Al
```

av2Al[<aVec>] –> <aList>

```
av2Al[MiniData`Private`d_Symbol] := With[{MiniData`Private`al =
     Sort[Union[({#1[[1, 1, 1]], #1[[2]]} &) /@ DownValues[MiniData`Private`d]]]},
  MiniData`Private`av2Alα /@ MiniData`Private`al]

av2Al[MiniData`Private`x_] := MiniData`Private`x
```

In[63]:=
```
av2Al[100]
```
Out[63]=
```
100
```

In[64]:=
```
av2Al[zzz]
```
Out[64]=
```
{}
```

In[65]:=
```
av2Al[z]
```
Out[65]=
$$\begin{pmatrix} \text{lat} & 100 \\ \text{lon} & 100 \\ \text{name} & \$5 \end{pmatrix}$$

- **(*as2Al*) Association to AList**
- **(*getSchema*) Get Schema**

In[66]:=
```
getSchema[{{"name", Zark}, {"lon", 300}, {"lat", 300}}]
```
Out[66]=
{lat, lon, name}

In[67]:=
```
getSchema[z]
```
Out[67]=
{lat, lon, name}

In[68]:=
```
getSchema[{}]
```
Out[68]=
{}

In[69]:=
```
getSchema[42]
```
Out[69]=
{}

- **(*schemaAndRecord2Al*, *schemaAndRecord2Av*)**

In[70]:=
```
av2Al@schemaAndRecord2Av[{"name", "lat", "lon"}, {Unique[], 120, 130}]
```
Out[70]=
$$\begin{pmatrix} \text{lat} & 120 \\ \text{lon} & 130 \\ \text{name} & \$7 \end{pmatrix}$$

- **(*copyAv*, *equalAvs*, *adjoin*)**

In[71]:=
```
DownValues@Evaluate@copyAv[z]
```
Out[71]=
{HoldPattern[MiniData`Private`d$1031(lat)] :↦ 100,
HoldPattern[MiniData`Private`d$1031(lon)] :↦ 100, HoldPattern[MiniData`Private`d$1031(name)] :↦ $5}

In[72]:=
```
equalAvs[z, copyAv@z]
```
Out[72]=
True

A smart adjoin can share the parts of structures that do not change. Structure sharing is not observable with immutables. This dumb adjoin just copies the structure and (invisibly) mutates the copy:

In[73]:=
```
av2Al@adjoin[z, {{"foo", 1}, {"lon", 200}}]
```

Out[73]=

foo	1
lat	100
lon	200
name	$5

In[74]:=
```
z2 = adjoin[z, "lat", 200];
```

In[75]:=
```
av2Al@z2
```

Out[75]=

lat	200
lon	100
name	$5

In[76]:=
```
av2Al@z
```

Out[76]=

lat	100
lon	100
name	$5

In[77]:=
```
av2Al[adjoin[z, z2]]
```

Out[77]=

lat	200
lon	100
name	$5

In[78]:=
```
av2Al[adjoin[z2, z]]
```

Out[78]=

lat	100
lon	100
name	$5

In[79]:=
```
adjoin[av2Al@z, "lat", 200]
```

Out[79]=

lat	200
lon	100
name	$5

In[80]:=
```
adjoin[av2Al@z, "fon", "fun"]
```

Out[80]=

fon	fun
lat	100
lon	100
name	$5

In[81]:=
```
adjoin[av2Al@z, {{"name", Zark}, {lon, 300}, {"lat", 300}}]
```

Out[81]=

lat	300
lon	300
name	Zark

(*containsKey*, *removeKey*)

In[82]:=
```
containsKey[{{"name", Zark}, {"lon", 300}, {"lat", 300}}, lon]
```

Out[82]=

True

In[83]:=
```
containsKey[{}, fon]
```

Out[83]=

False

In[84]:=
```
containsKey[{{"name", Zark}, {"lon", 300}, {"lat", 300}}, fon]
```

Out[84]=

False

In[85]:=
```
containsKey[al2Av@{{"name", Zark}, {"lon", 300}, {"lat", 300}}, fon]
```

Out[85]=

False

In[86]:=
```
containsKey[al2Av@{{"name", Zark}, {"lon", 300}, {"lat", 300}}, lon]
```

Out[86]=

True

In[87]:=
```
containsKey[{{"name", Zark}, {"lon", 300}, {"lat", 300}}, fon]
```

Out[87]=

False

In[88]:=
```
containsKey[z, "long"]
```

Out[88]=

False

In[89]:=
```
containsKey[z, "lon"]
```
Out[89]=
True

In[90]:=
```
removeKey[{{"name", Zark}, {"lon", 300}, {"lat", 300}}, lon]
```
Out[90]=
$$\begin{pmatrix} \text{name} & \text{Zark} \\ \text{lat} & 300 \end{pmatrix}$$

In[91]:=
```
av2Al@removeKey[al2Av@{{"name", Zark}, {"lon", 300}, {"lat", 300}}, lon]
```
Out[91]=
$$\begin{pmatrix} \text{lat} & 300 \\ \text{name} & \text{Zark} \end{pmatrix}$$

In[92]:=
```
removeKey[{{"name", Zark}, {"lon", 300}, {"lat", 300}}, fon]
```
Out[92]=
$$\begin{pmatrix} \text{name} & \text{Zark} \\ \text{lon} & 300 \\ \text{lat} & 300 \end{pmatrix}$$

In[93]:=
```
removeKey[{}, lon]
```
Out[93]=
{}

In[94]:=
```
av2Al[removeKey[z, "lon"]]
```
Out[94]=
$$\begin{pmatrix} \text{lat} & 100 \\ \text{name} & \$5 \end{pmatrix}$$

In[95]:=
```
av2Al[removeKey[z, "long"]]
```
Out[95]=
$$\begin{pmatrix} \text{lat} & 100 \\ \text{lon} & 100 \\ \text{name} & \$5 \end{pmatrix}$$

In[96]:=
```
av2Al[al2Av[{}]]
```
Out[96]=
{}

In[97]:=
```
av2Al[removeKey[al2Av[{}], "bar"]]
```
Out[97]=
{}

In[98]:=
```
av2Al[removeKey[al2Av[{}], bar]]
```
Out[98]=
```
{}
```

(*fetchAttribute*)

In[99]:=
```
? fetchAttribute
```
```
fetchAttribute[<aList|aVec>,<key> --> {<key>,<value>}
```

In[100]:=
```
fetchAttribute[{{"foo", 1}, {"bar", 2}}, "foo"]
```
Out[100]=
```
{foo, 1}
```

In[101]:=
```
fetchAttribute[{{"foo", 1}, {"bar", 2}}, "fob"]
```
Out[101]=
```
{}
```

(*lookup*, *lookupWithDefault*)

For ALists:

In[102]:=
```
lookup[{{"foo", 1}, {"bar", 2}}, "foo"]
```
Out[102]=
```
1
```

In[103]:=
```
lookup[{{"foo", 1}, {"bar", 2}}, "fob"] === Null
```
Out[103]=
```
True
```

In[104]:=
```
lookupWithDefault[{{"foo", 1}, {"bar", 2}}, "foo", 42]
```
Out[104]=
```
1
```

In[105]:=
```
lookupWithDefault[{{"foo", 1}, {"bar", 2}}, "fob", 42]
```
Out[105]=
```
42
```

for association vectors

In[106]:=
```
fetchAttribute[z, lon]
```
Out[106]=
```
{lon, 100}
```

In[107]:=
```
fetchAttribute[z, "lon"]
```
Out[107]= {lon, 100}

In[108]:=
```
fetchAttribute[z, "foo"]
```

In[109]:=
```
fetchAttribute[z, "foo"] === Null
```
Out[109]= True

In[110]:=
```
lookup[z, foo] === Null
```
Out[110]= True

In[111]:=
```
lookupWithDefault[z, "lon", 42]
```
Out[111]= 100

In[112]:=
```
lookupWithDefault[z, "foo", 42]
```
Out[112]= 42

(adjoin)

In[113]:=
```
? adjoin
```
adjoin[<aList|aVec>,<key>,<value>] –> <aList|aVec>; adjoin[<aList1|aVec1>,<aList|aVec>] –> <aList|aVec>

In[114]:=
```
av2Al@z
```
Out[114]=
$$\begin{pmatrix} \text{lat} & 100 \\ \text{lon} & 100 \\ \text{name} & \$5 \end{pmatrix}$$

In[115]:=
```
av2Al@adjoin[z, "foo", 42]
```
Out[115]=
$$\begin{pmatrix} \text{foo} & 42 \\ \text{lat} & 100 \\ \text{lon} & 100 \\ \text{name} & \$5 \end{pmatrix}$$

av, al

In[116]:=
```
av2Al@adjoin[z, {{"foo", 42}}]
```

Out[116]=
$$\begin{pmatrix} \text{foo} & 42 \\ \text{lat} & 100 \\ \text{lon} & 100 \\ \text{name} & \$5 \end{pmatrix}$$

al, al

In[117]:=
```
adjoin[av2Al@z, {{"foo", 42}}]
```

Out[117]=
$$\begin{pmatrix} \text{foo} & 42 \\ \text{lat} & 100 \\ \text{lon} & 100 \\ \text{name} & \$5 \end{pmatrix}$$

al, av

In[118]:=
```
adjoin[av2Al@z, al2Av@{{"foo", 42}}]
```

Out[118]=
$$\text{adjoin}\left(\begin{pmatrix} \text{lat} & 100 \\ \text{lon} & 100 \\ \text{name} & \$5 \end{pmatrix}, \text{MiniData`Private`d\$1166}\right)$$

av, av

In[119]:=
```
av2Al@adjoin[z, al2Av@{{"foo", 42}}]
```

Out[119]=
$$\begin{pmatrix} \text{foo} & 42 \\ \text{lat} & 100 \\ \text{lon} & 100 \\ \text{name} & \$5 \end{pmatrix}$$

(*sieve*)

In[120]:=
```
? sieve
```

sieve[<aList|aVec>,<keyList>] –> {<aVec(matchingKeys)>,<aVec(notMatchingKeys)>}

In[121]:=
```
sieve[{{"foo", 1}, {"bar", 2}, {"baz", 3}, {"qux", 4}}, {"boo", "far"}]
```

Out[121]=
$$\left\{\{\}, \begin{pmatrix} \text{foo} & 1 \\ \text{bar} & 2 \\ \text{baz} & 3 \\ \text{qux} & 4 \end{pmatrix}\right\}$$

In[122]:=
```
sieve[{{"foo", 1}, {"bar", 2}, {"baz", 3}, {"qux", 4}}, {"qux", "bla", "foo"}]
```

Out[122]=
$$\begin{pmatrix} \{foo,\ 1\} & \{qux,\ 4\} \\ \{bar,\ 2\} & \{baz,\ 3\} \end{pmatrix}$$

In[123]:=
```
sieve[{{"foo", 1}, {"bar", 2}, {"baz", 3}, {"qux", 4}}, {}]
```

Out[123]=
$$\left\{\{\}, \begin{pmatrix} foo & 1 \\ bar & 2 \\ baz & 3 \\ qux & 4 \end{pmatrix}\right\}$$

In[124]:=
```
sieve[{}, {"qux", "bla", "foo"}]
```

Out[124]=
```
{{}, {}}
```

In[125]:=
```
av2Al /@
 sieve[al2Av@{{"foo", 1}, {"bar", 2}, {"baz", 3}, {"qux", 4}}, {"qux", "bla", "foo"}]
```

Out[125]=
$$\begin{pmatrix} \{foo,\ 1\} & \{qux,\ 4\} \\ \{bar,\ 2\} & \{baz,\ 3\} \end{pmatrix}$$

getSchema

In[126]:=
```
? getSchema
```

getSchema[<aList|aVec>] –-> {<key>, ...}

In[127]:=
```
getSchema@z
```

Out[127]=
```
{lat, lon, name}
```

schemaAndRecord2Al

In[128]:=
```
? schemaAndRecord2Al
```

schemaAndRecord2Al[<schema>,<record>] –-> <aList>

In[129]:=
```
schemaAndRecord2Al[getSchema@z, {1, 2, 3}]
```

Out[129]=
$$\begin{pmatrix} lat & 1 \\ lon & 2 \\ name & 3 \end{pmatrix}$$

In[130]:=
```
? schemaAndRecord2Av
```

schemaAndRecord2Av[<schema>,<record>] −−> <aVec>

Examples

`d = al2Av[{{"foo", 1}, {"bar", 2}}]`
will present as the association list `{{"foo", 1}, {"bar", 2}}`

`d["foo"] --> 1`

`containsKey[d, "foo"] --> True`

`e = removeKey[d, "foo"]`
will present as the association list `{{"bar", 2}}`

`fetchAttribute[d, "foo"] --> {"foo", 1}`

`f = copyAv[d]` *(not strictly necessary, but see* `equalAvs`*)*
will present as the association list `{{"foo", 1}, {"bar", 2}}`

`equalAvs[d, f] --> True`

`g = adjoin[d, "baz", 3]`
will present as the association list `{{"foo", 1}, {"bar", 2}, {"baz", 3}}`

Also use adjoin to "change" values, meaning, to write a new association from an old one with the named attribute changed.

`av2Al[d] --> {{"foo", 1}, {"bar", 2}}`

`av2Schema[d] --> {"foo", "bar"}`

`schemaAndRecord2Av[{"foo", "bar"}, {1, 2}]`
will present as the association list `{{"foo", 1}, {"bar", 2}}`

experimental variation including types

UNFINISHED

`av2Schema[d] --> {{"foo", Integer}, {"bar", Integer}}`

`schemaAndRecord2Av[{{"foo", Integer}, {"bar", Integer}}, {{1, {1}}, {2, {2}}}]`
will present as the association list `{{"foo", 1, Integer}, {"bar", 2, Integer}}`

When combining the record with the schema, check that the data type of each value in the record is a subset of the type described in the schema. If so, take the resulting type from the schema, as it is the "most permissible" or "most permissive."

This should check as equal to `{{"foo", 1, {1, 2}}, {"bar", 2, {1, 2}}}` and a host of others. Type judgement questions will be questions about membership in sets or about subset relationships. The small-

est enclosing set for the value **1** is **{1}**. **1** is a member of any set that is a superset of **{1}**, so the quesion "is **1** of type **T**?" is equivalent to the question "is **{1}** a subset of **T**?"

By re-defining a **[value]** as a pair of a **[member]** or **[element]** or **[witness]** of a **[type]** *qua* a set of values (or a name of a set of values, or a reference to a set of values), we *perhaps* become compatible with the type system recorded in the next subsection:

d = al2Av[{{"foo", 1, Integer}, {"bar", 2, Integer}}]

d["foo"] --> {1, Integer}

containsKey[d, "foo"] --> True

e = removeKey[d, "foo"] --> *will present as the association list* **{{"bar", 2, Integer}}**

fetchAttribute[d, "foo"] --> {"foo", 1, Integer}

f = copyAv[d] --> *(not strictly necessary, but see* **equalAvs***) will present as the association list* **{{"foo", 1, Integer}, {"bar", 2, Integer}}**

equalAvs[d, f] --> True

g = adjoin[d, "baz", 3] --> *will present as the association list*
{{"foo", 1, Integer}, {"bar", 2, Integer}, {"baz", 3, Integer}}

av2Al[d] --> {{"foo", 1, Integer}, {"bar", 2, Integer}}

Data Transformers

A **[data transformer (Dt)]** takes an association list (or association vector) and transforms certain members contained in it, returning a new association list (or association vector) containing both the unchanged attributes and the changed attributes.

A **[data transformer maker (Dtm)]** takes a schema and returns a data transformer that will transform members whose keys match those given in the schema.

A **[data transformer maker maker (Dtmm)]** takes a pair of transformation functions, one for transforming the key of an attribute and one for transforming the value of an attribute, and returns a data transformer maker.

In[131]:=
```
Pair[x_, y_] := {x, y};
```

In[132]:=
```
Second[thing_] := thing[[2]];
```

In[133]:=
```
Remove[idDtm, logDtm, capDtm, dtmm, selectDtmm, dtJoin, dropDtm, keepDtm, dtUnit]
```

The id Dtm

The id Dtm takes a schema and returns a Dt (function of an association list) that does not change the association. The id Dtm ignores its input schema.

In[134]:=
```
idDtm[schema_List] := # &;
```

In[135]:=
```
testData = RandomSample@
  {{"Result", R},
   {"NVotes", 3000},
   {"Precinct", alameda},
   {"foo", 5},
   {"bar", 6},
   {"baz", 7},
   {"qux", 8},
   {"blargh", 9}}
```

Out[135]=

Precinct	alameda
baz	7
qux	8
NVotes	3000
blargh	9
foo	5
bar	6
Result	R

In[136]:=
```
idDtm[{"Result"}]@testData
```

Out[136]=

Precinct	alameda
baz	7
qux	8
NVotes	3000
blargh	9
foo	5
bar	6
Result	R

In[137]:=
```
idDtm[{"foo", "bar"}]@testData
```

Out[137]=

Precinct	alameda
baz	7
qux	8
NVotes	3000
blargh	9
foo	5
bar	6
Result	R

The Log Dtm

The log Dtm takes a schema and returns a Dt function that will take the log of any attributes whose keys match those in the given schema. The Dt returned by the log Dtm transforms both the key of the matching attribute and the value of the matching attribute. It changes the key by wrapping it in a "Log10(...)" notation,

and it changes the value by actually taking the log base 10.

In[138]:=
```
logDtm[schema_List] :=
  Function[aList,
   Map[
    Function[attribute,
     Module[{atkey, atval},
      {atkey, atval} = attribute;
      If[MemberQ[schema, atkey],
       Pair[
        StringJoin["Log10(", atkey, ")"],
        Log[10, N[atval]]],
       attribute]]],
    aList]];
```

In[139]:=
```
logDtm[{"NVotes"}]@testData
```

Out[139]=

Precinct	alameda
baz	7
qux	8
Log10(NVotes)	3.47712
blargh	9
foo	5
bar	6
Result	*R*

The Dt Composer, dtJoin

Composition of data transformers is straightforward.

In[140]:=
```
dtJoin = Composition;
```

In[141]:=
```
idDtm[{"Ringo"}]@testData
```

Out[141]=

Precinct	alameda
baz	7
qux	8
NVotes	3000
blargh	9
foo	5
bar	6
Result	*R*

In[142]:=
```
dtJoin[
  idDtm[{"Result"}],
  idDtm[{"Ringo"}]]@testData
```

Out[142]=

Precinct	alameda
baz	7
qux	8
NVotes	3000
blargh	9
foo	5
bar	6
Result	*R*

In[143]:=
```
dtJoin[
  idDtm[{"Result"}],
  logDtm[{"NVotes"}]]@testData
```

Out[143]=

Precinct	alameda
baz	7
qux	8
Log10(NVotes)	3.47712
blargh	9
foo	5
bar	6
Result	*R*

In[144]:=
```
dtJoin[
  idDtm[{"NVotes", "foo"}],
  logDtm[{"NVotes", "bar"}]]@testData
```

Out[144]=

Precinct	alameda
baz	7
qux	8
Log10(NVotes)	3.47712
blargh	9
foo	5
Log10(bar)	0.778151
Result	*R*

The Capitalization Dtm

The Capitalization Dtm takes a schema and returns a Dt that capitalizes the value of any attribute whose key matches any of the keys given in the schema.

In[145]:=
```
capitalize[s_String] :=
 StringJoin@
  (FromCharacterCode /@
    (If[# ≥ 97 && # ≤ 122, # - 32, #] & /@
      (ToCharacterCode[s])))
```

In[146]:=
```
capDtm[schema_List] :=
  Function[aList,
   Map[
    Function[attribute,
     Module[{atkey, atval},
      {atkey, atval} = attribute;
      If[MemberQ[schema, atkey],
       Pair[
        atkey,
        capitalize[ToString[atval]]],
       attribute]]],
    aList]];
```

In[147]:=
```
capDtm[{"Precinct", "Result", "foo"}]@
 testData
```

Out[147]=

Precinct	ALAMEDA
baz	7
qux	8
NVotes	3000
blargh	9
foo	5
bar	6
Result	R

In[148]:=
```
dtJoin[
  capDtm[{"Precinct", "Result", "foo"}],
  logDtm[{"NVotes"}]]@
 testData
```

Out[148]=

Precinct	ALAMEDA
baz	7
qux	8
Log10(NVotes)	3.47712
blargh	9
foo	5
bar	6
Result	R

The Drop Dtm

The Drop Dtm takes a schema and returns a Dt that drops any attributes whose keys match any of the keys in the schema

In[149]:=
```
dropDtm[schema_List] :=
  Function[aList,
   Select[aList,
    Function[attribute,
     Module[{atkey, atval},
      {atkey, atval} = attribute;
      Not@MemberQ[schema, atkey]]]]];
```

In[150]:=
```
dropDtm[RandomSample@{"foo", "bar", "baz", "qux", "barbie", "ken"}]@
 testData
```

Out[150]=

Precinct	alameda
NVotes	3000
blargh	9
Result	*R*

The Keep Dtm

The logical converse of the Drop Dtm: this creates a Dt that keeps any attributes with keys matching anything in the schema

In[151]:=
```
keepDtm[schema_List] :=
  Function[aList,
   Select[aList,
    Function[attribute,
     Module[{atkey, atval},
      {atkey, atval} = attribute;
      MemberQ[schema, atkey]]]]];
```

Data Transformer Maker Makers (Dtmms)

Many Dtms differ only in the transformation function that gets mapped or filtered over. Write Dtmms that build these Dtms

This Dtmm takes a pair of transformation functions, one for keys and one for values, both defaulting to the identity function, and returns a Dtm that will take a schema and return a Dt that applies these functions to attributes in a given association that match any of the keys in the schema:

Make DTs Work on Association Vectors, too

while we're at it...

In[152]:=
```
id = Function[x, x];
```

In[153]:=
```
Remove[dtmm];
dtmm[valueTransform_: id, keyTransform_: id] :=
  Function[schema, (* Dtm returned by this Dtmm *)
   Function[assoc, (* Dt returned by the Dtm *)
    Map[
     Function[attribute,
      Module[{atkey, atval},
       {atkey, atval} = attribute;
       If[MemberQ[schema, atkey],
        Pair[
         keyTransform[atkey],
         valueTransform[atval]],
        attribute]]],
     assoc]]];
```

In[155]:=
```
Remove[idDtm, logDtm, capDtm];
idDtm = dtmm[];
logDtm = dtmm[Log[10, N@#] &, StringJoin["Log10(", #, ")"] &];
capDtm = dtmm[Composition[capitalize, ToString]];
```

In[159]:=
```
capDtm[{"Precinct"}]@testData
```

Out[159]=

Precinct	ALAMEDA
baz	7
qux	8
NVotes	3000
blargh	9
foo	5
bar	6
Result	*R*

In[160]:=
```
capDtm[{"Precinct"}]@testData
```

Out[160]=

Precinct	ALAMEDA
baz	7
qux	8
NVotes	3000
blargh	9
foo	5
bar	6
Result	*R*

In[161]:=
```
dtJoin[
  idDtm[{"Result"}],
  logDtm[{"NVotes"}],
  capDtm[{"Precinct"}]]@testData
```

Out[161]=

Precinct	ALAMEDA
baz	7
qux	8
Log10(NVotes)	3.47712
blargh	9
foo	5
bar	6
Result	*R*

In[162]:=
```
dtJoin[
  idDtm[{"Result"}],
  logDtm[{"NVotes"}],
  capDtm[{"Precinct"}]]@testData
```

Out[162]=

Precinct	ALAMEDA
baz	7
qux	8
Log10(NVotes)	3.47712
blargh	9
foo	5
bar	6
Result	*R*

Another kind of Dtmm is one that abstracts over the dropDtm and the keepDtm: a Dtmm takes ***criterion function*** of a schema, key, and value and returns a Dtm that will take a schema and return a Dt that will apply the criterion to every member that realizes it. The default for the criterion function is one that always returns true.

In[163]:=
```
Remove[selectDtmm];
selectDtmm[criterion_: Function[{s, k, v}, True]] :=
  Function[schema,
   Function[assoc, (* Dt returned by the Dtm *)
    Select[assoc,
     Function[attribute,
      Module[{atkey, atval},
       {atkey, atval} = attribute;
       criterion[schema, atkey, atval]]]]]];
```

In[165]:=
```
Remove[dropDtm, keepDtm]
dropDtm = selectDtmm[Function[{s, k, v}, Not@MemberQ[s, k]]];
keepDtm = selectDtmm[Function[{s, k, v}, MemberQ[s, k]]];
```

In[168]:=
```
dropDtm[RandomSample@{"foo", "bar", "baz", "qux", "barbie", "ken"}]@testData
```

Out[168]=

Precinct	alameda
NVotes	3000
blargh	9
Result	*R*

In[169]:=
```
dropDtm[RandomSample@{"foo", "bar", "baz", "qux", "barbie", "ken"}]@testData
```

Out[169]=

Precinct	alameda
NVotes	3000
blargh	9
Result	*R*

In[170]:=
```
keepDtm[RandomSample@{"Precinct", "NVotes", "Result", "barbie", "ken"}]@testData
```

Out[170]=

Precinct	alameda
NVotes	3000
Result	*R*

In[171]:=
```
keepDtm[RandomSample@{"Precinct", "NVotes", "Result", "barbie", "ken"}]@testData
```

Out[171]=

Precinct	alameda
NVotes	3000
Result	*R*

In[172]:=
```
dropDtm[{}]@testData
```

Out[172]=

Precinct	alameda
baz	7
qux	8
NVotes	3000
blargh	9
foo	5
bar	6
Result	*R*

In[173]:=
```
dropDtm[{}]@testData
```

Out[173]=

Precinct	alameda
baz	7
qux	8
NVotes	3000
blargh	9
foo	5
bar	6
Result	*R*

In[174]:=
```
keepDtm[{}]@testData
```

Out[174]=
```
{}
```

In[175]:=
```
keepDtm[{}]@testData
```

Out[175]=
```
{}
```

Associativity?

In[176]:=
```
dtJoin[
  idDtm[{"Result"}],
  dtJoin[
   logDtm[{"NVotes"}],
   capDtm[{"Precinct"}]]]@testData
```

Out[176]=

Precinct	ALAMEDA
baz	7
qux	8
Log10(NVotes)	3.47712
blargh	9
foo	5
bar	6
Result	*R*

In[177]:=
```
dtJoin[
  idDtm[{"Result"}],
  dtJoin[
   logDtm[{"NVotes"}],
   capDtm[{"Precinct"}]]]@testData
```

Out[177]=

Precinct	ALAMEDA
baz	7
qux	8
Log10(NVotes)	3.47712
blargh	9
foo	5
bar	6
Result	*R*

In[178]:=
```
dtJoin[
  dtJoin[
   idDtm[{"Result"}],
   logDtm[{"NVotes"}]],
  capDtm[{"Precinct"}]]@testData
```

Out[178]=

Precinct	ALAMEDA
baz	7
qux	8
Log10(NVotes)	3.47712
blargh	9
foo	5
bar	6
Result	*R*

In[179]:=
```
dtJoin[
  dtJoin[
   idDtm[{"Result"}],
   logDtm[{"NVotes"}]],
  capDtm[{"Precinct"}]]@testData
```

Out[179]=

Precinct	ALAMEDA
baz	7
qux	8
Log10(NVotes)	3.47712
blargh	9
foo	5
bar	6
Result	*R*

Unit?, Left and Right?

In[180]:=
```
dtUnit = idDtm[{}];
```

In[181]:=
```
dtJoin[capDtm[{"Precinct"}], dtUnit]@testData
```

Out[181]=

Precinct	ALAMEDA
baz	7
qux	8
NVotes	3000
blargh	9
foo	5
bar	6
Result	*R*

In[182]:=
```
dtJoin[dtUnit, capDtm[{"Precinct"}]]@testData
```

Out[182]=

Precinct	ALAMEDA
baz	7
qux	8
NVotes	3000
blargh	9
foo	5
bar	6
Result	*R*