OrganismDbi: A meta framework for Annotation Packages

Marc Carlson

October 17, 2016

OrganismDbi is a software package that helps tie together different annotation resources. It is expected that users may have previously made seen packages like *org.Hs.eg.db* and *TxDb.Hsapiens.UCSC.hg19.knownGene*. Packages like these two are very different and contain very different kinds of information, but are still about the same organism: Homo sapiens. The *OrganismDbi* package allows us to combine resources like these together into a single package resource, which can represent ALL of these resources at the same time. An example of this is the *homo.sapiens* package, which combines access to the two resources above along with others.

This is made possible because the packages that are represented by *homo.sapiens* are related to each other via foreign keys.

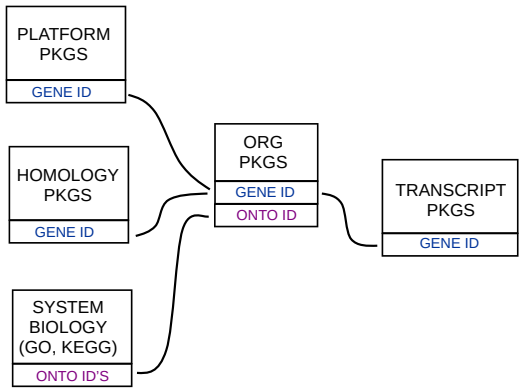

Figure 1: Relationships between Annotation packages

1 Getting started with OrganismDbi

Usage of a package like this has been deliberately kept very simple. The methods supported are the same ones that work for all the packages based on *AnnotationDb* objects. The methods that can be applied to these new packages are `columns`, `keys`, `keytypes` and `select`.

So to learn which kinds of data can be retrieved from a package like this we would simply load the package and then call the `columns` method.

```
> library(Homo.sapiens)
> columns(Homo.sapiens)
```

```
 [1] "ACCNUM"       "ALIAS"        "CDSCHROM"     "CDSEND"       "CDSID"
 [6] "CDSNAME"      "CDSSTART"     "CDSSTRAND"    "DEFINITION"   "ENSEMBL"
[11] "ENSEMBLPROT"  "ENSEMBLTRANS" "ENTREZID"     "ENZYME"       "EVIDENCE"
[16] "EVIDENCEALL"  "EXONCHROM"    "EXONEND"      "EXONID"       "EXONNAME"
[21] "EXONRANK"     "EXONSTART"    "EXONSTRAND"   "GENEID"       "GENENAME"
[26] "GO"           "GOALL"        "GOID"         "IPI"          "MAP"
[31] "OMIM"         "ONTOLOGY"     "ONTOLOGYALL"  "PATH"         "PFAM"
[36] "PMID"         "PROSITE"      "REFSEQ"       "SYMBOL"       "TERM"
[41] "TXCHROM"      "TXEND"        "TXID"         "TXNAME"       "TXSTART"
[46] "TXSTRAND"     "TXTYPE"       "UCSCKG"       "UNIGENE"      "UNIPROT"
```

To learn which of those kinds of data can be used as keys to extract data, we use the `keytypes` method.

```
> keytypes(Homo.sapiens)
```

```
 [1] "ACCNUM"       "ALIAS"        "CDSID"        "CDSNAME"      "DEFINITION"
 [6] "ENSEMBL"      "ENSEMBLPROT"  "ENSEMBLTRANS" "ENTREZID"     "ENZYME"
[11] "EVIDENCE"     "EVIDENCEALL"  "EXONID"       "EXONNAME"     "GENEID"
[16] "GENENAME"     "GO"           "GOALL"        "GOID"         "IPI"
[21] "MAP"          "OMIM"         "ONTOLOGY"     "ONTOLOGYALL"  "PATH"
[26] "PFAM"         "PMID"         "PROSITE"      "REFSEQ"       "SYMBOL"
[31] "TERM"         "TXID"         "TXNAME"       "UCSCKG"       "UNIGENE"
[36] "UNIPROT"
```

To extract specific keys, we need to use the `keys` method, and also provide it a legitimate keytype:

```
> head(keys(Homo.sapiens, keytype="ENTREZID"))
```

```
[1] "1"  "2"  "3"  "9"  "10" "11"
```

And to extract data, we can use the `select` method. The select method depends on the values from the previous three methods to specify what it will extract. Here is an example that will extract, UCSC transcript names, and gene symbols using Entrez Gene IDs as keys.

```
> k <- head(keys(Homo.sapiens, keytype="ENTREZID"),n=3)
> select(Homo.sapiens, keys=k, columns=c("TXNAME","SYMBOL"), keytype="ENTREZID")

  ENTREZID SYMBOL     TXNAME
1        1   A1BG uc002qsd.4
2        1   A1BG uc002qsf.2
3        2    A2M uc001qvk.1
4        2    A2M uc009zgk.1
5        3  A2MP1 uc021qum.1
```

In Addition to `select`, some of the more popular range based methods have also been updated to work with an *AnnotationDb* object. So for example you could extract transcript information like this:

```
> transcripts(Homo.sapiens, columns=c("TXNAME","SYMBOL"))

GRanges object with 82960 ranges and 2 metadata columns:
                seqnames            ranges strand |          TXNAME
                   <Rle>         <IRanges>  <Rle> | <CharacterList>
      [1]           chr1 [ 11874,  14409]      + |      uc001aaa.3
      [2]           chr1 [ 11874,  14409]      + |      uc010nxq.1
      [3]           chr1 [ 11874,  14409]      + |      uc010nxr.1
      [4]           chr1 [ 69091,  70008]      + |      uc001aal.1
      [5]           chr1 [321084, 321115]      + |      uc001aaq.2
      ...            ...               ...    ... .           ...
  [82956] chrUn_gl000237   [    1,  2686]      - |      uc011mgu.1
  [82957] chrUn_gl000241   [20433, 36875]      - |      uc011mgv.2
  [82958] chrUn_gl000243   [11501, 11530]      + |      uc011mgw.1
  [82959] chrUn_gl000243   [13608, 13637]      + |      uc022brq.1
  [82960] chrUn_gl000247   [ 5787,  5816]      - |      uc022brr.1
                  SYMBOL
         <CharacterList>
      [1]         DDX11L1
      [2]         DDX11L1
```

```
      [3]         DDX11L1
      [4]           OR4F5
      [5]              NA
      ...             ...
  [82956]              NA
  [82957]              NA
  [82958]              NA
  [82959]              NA
  [82960]              NA
  -------
  seqinfo: 93 sequences (1 circular) from hg19 genome
```

And the *GRanges* object that would be returned would have the information that you specified in the columns argument. You could also have used the `exons` or `cds` methods in this way.

The `transcriptsBy`,`exonsBy` and `cdsBy` methods are also supported. For example:

```
> transcriptsBy(Homo.sapiens, by="gene", columns=c("TXNAME","SYMBOL"))

GRangesList object of length 23459:
$1
GRanges object with 2 ranges and 3 metadata columns:
      seqnames               ranges strand |     tx_name          TXNAME
         <Rle>            <IRanges>  <Rle> | <character> <CharacterList>
  [1]    chr19 [58858172, 58864865]      - |  uc002qsd.4      uc002qsd.4
  [2]    chr19 [58859832, 58874214]      - |  uc002qsf.2      uc002qsf.2
               SYMBOL
      <CharacterList>
  [1]            A1BG
  [2]            A1BG

$10
GRanges object with 1 range and 3 metadata columns:
      seqnames               ranges strand |    tx_name     TXNAME SYMBOL
  [1]     chr8 [18248755, 18258723]      + | uc003wyw.1 uc003wyw.1   NAT2

$100
GRanges object with 1 range and 3 metadata columns:
      seqnames               ranges strand |    tx_name     TXNAME SYMBOL
```

```
  [1]    chr20 [43248163, 43280376]      - | uc002xmj.3 uc002xmj.3    ADA

...
<23456 more elements>
-------
seqinfo: 93 sequences (1 circular) from hg19 genome
```

2 Making your own OrganismDbi packages

So in the preceding section you can see that using an *OrganismDbi* package behaves very similarly to how you might use a `TxDb` or an `OrgDb` package. The same methods are defined, and the behave similarly except that they now have access to much more data than before. But before you make your own OrganismDbi package you need to understand that there are few logical limitations for what can be included in this kind of package.

- The 1st limitation is that all the annotation resources in question must have implemented the four methods described in the preceding section (`columns`, `keys`, `keytypes` and `select`).
- The 2nd limitation is that you cannot have more than one example of each field that can be retrieved from each type of package that is included. So basically, all values returned by `columns` must be unique across ALL of the supporting packages.
- The 3rd limitation is that you cannot have more than one example of each object type represented. So you cannot have two org packages since that would introduce two `OrgDb` objects.
- And the 4th limitation is that you cannot have cycles in the graph. What this means is that there will be a graph that represents the relationships between the different object types in your package, and this graph must not present more than one pathway between any two nodes/objects. This limitation means that you can choose one foreign key relationship to connect any two packages in your graph.

With these limitations in mind, lets set up an example. Lets show how we could make Homo.sapiens, such that it allowed access to *org.Hs.eg.db*, *TxDb.Hsapiens.UCSC.hg19.knownGene* and *GO.db*.

The 1st thing that we need to do is set up a list that expresses the way that these different packages relate to each other. To do this, we make a

`list` that contains short two element long character vectors. Each character vector represents one relationship between a pair of packages. The names of the vectors are the package names and the values are the foreign keys. Please note that the foreign key values in these vectors are the same strings that are returned by the `columns` method for the individual packages. Here is an example that shows how *GO.db*, *org.Hs.eg.db* and *TxDb.Hsapiens.UCSC.hg19.knownGene* all relate to each other.

```
> gd <- list(join1 = c(GO.db="GOID", org.Hs.eg.db="GO"),
+            join2 = c(org.Hs.eg.db="ENTREZID",
+                      TxDb.Hsapiens.UCSC.hg19.knownGene="GENEID"))
```

So this `data.frame` indicates both which packages are connected to each other, and also what these connections are using for foreign keys.

Once this is finished, we just have to call the `makeOrganismPackage` function to finish the task.

```
> destination <- tempfile()
> dir.create(destination)
> makeOrganismPackage(pkgname = "Homo.sapiens",
+   graphData = gd,
+   organism = "Homo sapiens",
+   version = "1.0.0",
+   maintainer = "Package Maintainer<maintainer@somewhere.org>",
+   author = "Some Body",
+   destDir = destination,
+   license = "Artistic-2.0")
```

`makeOrganismPackage` will then generate a lightweight package that you can install. This package will not contain all the data that it refers to, but will instead depend on the packages that were referred to in the `data.frame`. Because the end result will be a package that treats all the data mapped together as a single source, the user is encouraged to take extra care to ensure that the different packages used are from the same build etc.