Making and Utilizing TranscriptDb Objects

Marc Carlson Patrick Aboyoun Hervé Pagès
Seth Falcon Martin Morgan

December 6, 2010

1 Introduction

The *GenomicFeatures* package retrieves and manages transcript-related features from the UCSC Genome Bioinformatics[1] and BioMart[2] data resources. The package is useful for ChIP-chip, ChIP-seq, and RNA-seq analyses.

```
> library("GenomicFeatures")
```

2 Transcript Metadata

2.1 *TranscriptDb* Objects

The *GenomicFeatures* package uses *TranscriptDb* objects to store transcript metadata. This class maps the 5' and 3' untranslated regions (UTRs), protein coding sequences (CDSs) and exons for a set of mRNA transcripts to their associated genome. *TranscriptDb* objects also have accessors functions to allow such features to be retrieved individually or grouped together in a way that reflects the underlying biology.

All *TranscriptDb* objects are backed by a SQLite database that manages genomic locations and the relationships between pre-processed mRNA transcripts, exons, protein coding sequences, and their related gene identifiers.

2.2 Creating New *TranscriptDb* Objects

The *GenomicFeatures* package provides functions to create *TranscriptDb* objects based on data downloaded from UCSC Genome Bioinformatics or BioMart. The following subsections demonstrate the use of these functions.

[1] `http://genome.ucsc.edu/`
[2] `http://www.biomart.org/`

There is also support for creating *TranscriptDb* objects from custom data sources using `makeTranscriptDb`; see the help page for this function for details.

2.2.1 Using `makeTranscriptDbFromUCSC`

The function `makeTranscriptDbFromUCSC` downloads UCSC Genome Bioinformatics transcript tables (e.g. `"knownGene"`, `"refGene"`, `"ensGene"`) for a genome build (e.g. `"mm9"`, `"hg19"`). Use the `supportedUCSCtables` utility function to get the list of supported tables.

```
> supportedUCSCtables()[1:4, ]
```

```
                                track        subtrack
knownGene                     UCSC Genes            <NA>
knownGeneOld3              Old UCSC Genes            <NA>
wgEncodeGencodeManualV3   Gencode Genes Genecode Manual
wgEncodeGencodeAutoV3     Gencode Genes   Genecode Auto
```

```
> mm9KG <- makeTranscriptDbFromUCSC(genome = "mm9", tablename = "knownGene")
```

The function `makeTranscriptDbFromUCSC` also takes an important argument called `circ_seqs` to label which chromosomes are circular. The argument is a character vector of strings that correspond to the circular chromosomes (as labeled by the source). To discover what the source calls their chromosomes, use the `getChromInfoFromUCSC` function to list them. By default, there is a supplied character vector that will attempt to label all the mitochondrial chromosomes as circular by matching to them. This is the `DEFAULT_CIRC_SEQS` vector. It contains strings that usually correspond to mitochondrial chromosomes. Once the database has been generated with the circular chromosomes tagged in this way, all subsequent analysis of these chromosomes will be able to consider their circularity for analysis. So it is important for the user to make sure that they pass in the correct strings to the `circ_seqs` argument to ensure that the correct sequences are tagged as circular by the database.

```
> head(getChromInfoFromUCSC("hg19"))
```

```
  chrom    length
1  chr1 249250621
2  chr2 243199373
```

```
3  chr3 198022430
4  chr4 191154276
5  chr5 180915260
6  chr6 171115067
```

2.2.2 Using `makeTranscriptDbFromBiomart`

Retrieve data from BioMart by specifying the mart and the data set to the `makeTranscriptDbFromBiomart` function (not all BioMart data sets are currently supported):

```
> mmusculusEnsembl <-
+    makeTranscriptDbFromBiomart(biomart = "ensembl",
+                                dataset = "mmusculus_gene_ensembl")
```

As with the `makeTranscriptDbFromUCSC` function, the `makeTranscriptDbFromBiomart` function also has a `circ_seqs` argument that will default to using the contents of the `DEFAULT_CIRC_SEQS` vector. And just like those UCSC sources, there is also a helper function called `getChromInfoFromBiomart` that can show what the different chromosomes are called for a given source.

Using the `makeTranscriptDbFromBiomart` `makeTranscriptDbFromUCSC` functions can take a while and may also require some bandwidth as these methods have to download and then assemble a database from their respective sources. It is not expected that most users will want to do this step every time. Instead, we suggest that you save your annotation objects and label them with an appropriate time stamp so as to facilitate reproducible research.

2.3 Saving and Loading a *TranscriptDb* Object

Once a *TranscriptDb* object has been created, it can be saved to avoid the time and bandwidth costs of recreating it and to make it possible to reproduce results with identical genomic feature data at a later date. Since *TranscriptDb* objects are backed by a SQLite database, the save format is a SQLite database file (which could be accessed from programs other than Rif desired). Note that it is not possible to serialize a *TranscriptDb* object using R's `save` function.

```
> saveFeatures(mm9KG, file="fileName.sqlite")
```

A *TranscriptDb* object can be initialized from a file using `loadFeatures`.

```
> mm9KG <- loadFeatures("fileName.sqlite")
```

3 Retrieving Transcript, Exon, and Coding Sequence Ranges

3.1 Loading some sample genomic feature data

Here was are loading a previously created *TranscriptDb* object based on UCSC known gene data. This database only contains a small subset of the possible annotations for human and is only included to demonstrate and test the functionality of the *GenomicFeatures* package.

```
> samplefile <- system.file("extdata", "UCSC_knownGene_sample.sqlite",
+                           package="GenomicFeatures")
> txdb <- loadFeatures(samplefile)
> txdb
```

```
TranscriptDb object:
| Db type: TranscriptDb
| Data source: UCSC
| Genome: hg18
| UCSC Table: knownGene
| Type of Gene ID: Entrez Gene ID
| Full dataset: no
| transcript_nrow: 135
| exon_nrow: 544
| cds_nrow: 324
| Db created by: GenomicFeatures package from Bioconductor
| Creation time: 2010-09-21 16:27:22 -0700 (Tue, 21 Sep 2010)
| GenomicFeatures version at creation time: 1.1.12
| RSQLite version at creation time: 0.9-2
| DBSCHEMAVERSION: 1.0
```

3.2 Working with Basic Features

The most basic operations on a *TranscriptDb* object retrieve the genomic coordinates or *ranges* for exons, transcripts or coding sequences. The functions `transcripts`, `exons`, and `cds` return the coordinate information as a *GRanges* object.

For example, all transcripts present in a *TranscriptDb* object can be obtained as follows:

```
> GR <- transcripts(txdb)
> GR[1:3]

GRanges with 3 ranges and 2 elementMetadata values
    seqnames       ranges strand |     tx_id     tx_name
       <Rle>    <IRanges>  <Rle> | <integer> <character>
[1]     chr1 [1116, 4121]      + |         1  uc001aaa.2
[2]     chr1 [1116, 4272]      + |         2  uc009vip.1
[3]     chr1 [4269, 6628]      - |         3  uc009vis.1

seqlengths
          chr1   chr1_random         chr10 ...   chrX_random          chrY
     247249719       1663265     135374737 ...       1719168      57772954
```

The `transcripts` function returns a *GRanges* class object. You can learn a lot more about the manipulation of these objects by reading the *GenomicRanges* introductory vignette. The `show` method for a *GRanges* object will display the ranges, seqnames (a chromosome or a contig), and strand on the left side and then present related metadata on the right side. At the bottom, the seqlengths display all the possible seqnames along with the length of each sequence.

In addition, the `transcripts` function can also be used to retrieve a subset of the transcripts available such as those on the +-strand of chromosome 1.

```
> GR <- transcripts(txdb, vals <- list(tx_chrom = "chr1", tx_strand = "+"))
> length(GR)

[1] 2

> unique(strand(GR))

[1] +
Levels: + - *
```

The `exons` and `cds` functions can also be used in a similar fashion to retrive genomic coordinates for exons and coding sequences.

3.3 Working with Grouped Features

Often one is interested in how particular genomic features relate to each other, and not just their location. For example, it might be of interest to

group transcripts by gene or to group exons by transcript. Such groupings are supported by the `transcriptsBy`, `exonsBy`, and `cdsBy` functions.

The following call can be used to group transcripts by genes:

```
> GRList <- transcriptsBy(txdb, by = "gene")
> length(GRList)

[1] 51

> names(GRList)[10:13]

[1] "23192"  "245938" "255403" "26751"

> GRList[11:12]

GRangesList of length 2
$245938
GRanges with 1 range and 2 elementMetadata values
    seqnames         ranges strand |     tx_id     tx_name
       <Rle>      <IRanges>  <Rle> | <integer> <character>
[1]    chr20 [16351, 25296]      + |        67  uc002wcw.1

$255403
GRanges with 1 range and 2 elementMetadata values
    seqnames          ranges strand |     tx_id     tx_name
       <Rle>       <IRanges>  <Rle> | <integer> <character>
[1]     chr4 [43227, 146490]      + |        10  uc003fzt.2


seqlengths
          chr1   chr1_random         chr10 ...   chrX_random          chrY
     247249719       1663265     135374737 ...       1719168      57772954
```

The `transcriptsBy` function returns a *GRangesList* class object. As with *GRanges* objects, you can learn more about these objects by reading the *GenomicRanges* introductory vignette. The `show` method for a *GRangesList* object will display as a list of *GRanges* objects. And, at the bottom the seqlengths will be displayed once for the entire list.

For each of these three functions, there is a limited set of options that can be passed into the `by` argument to allow grouping. For the `transcriptsBy`

function, you can group by gene, exon or cds, whereas for the `exonsBy` and `cdsBy` functions can only be grouped by transcript (tx) or gene.

So as a further example, to extract all the exons for each transcript you can call:

```
> GRList <- exonsBy(txdb, by = "tx")
> length(GRList)

[1] 135

> names(GRList)[10:13]

[1] "10" "11" "12" "13"

> GRList[[12]]

GRanges with 4 ranges and 3 elementMetadata values
    seqnames         ranges strand |   exon_id   exon_name exon_rank
       <Rle>      <IRanges>  <Rle> | <integer> <character> <integer>
[1]     chr4 [43227, 43385]      + |        50          NA         1
[2]     chr4 [49323, 49449]      + |        51          NA         2
[3]     chr4 [49951, 50046]      + |        52          NA         3
[4]     chr4 [57732, 58380]      + |        56          NA         4

seqlengths
          chr1   chr1_random         chr10 ...   chrX_random          chrY
     247249719       1663265     135374737 ...       1719168      57772954
```

As you can see, the *GRangesList* objects returned from each function contain locations and identifiers grouped into a list like object according to the type of feature specified in the `by` argument. The object returned can then be used by functions like `findOverlaps` to contextualize alignments from high-throughput sequencing.

The identifiers used to label the *GRanges* objects depend upon the data source used to create the *TranscriptDb* object. So the list identifiers will not always be Entrez Gene IDs, as they were in the first example. Furthermore, some data sources do not provide a unique identifier for all features. In this situation, the group label will be a synthetic ID created by *GenomicFeatures* to keep the relations between features consistent in the database this was the case in the 2nd example. Even though the results will sometimes have to come back to you as synthetic IDs, you can still always retrieve the original IDs. In the following example, we will find the original transcript names stored in the database for each ID by calling the `transcripts` function.

```
> tx_ids <- names(GRList)
> vals <- list(tx_id=tx_ids)
> txs <- transcripts(txdb, vals, columns = c("tx_id", "tx_name"))
> head(values(txs))

DataFrame with 6 rows and 2 columns
      tx_id     tx_name
  <integer> <character>
1         1  uc001aaa.2
2         2  uc009vip.1
3         3  uc009vis.1
4        39  uc001ifj.1
5        37  uc001ifi.1
6        38  uc009xhe.1
```

Finally, the order of the results in a *GRangesList* object can vary with the way in which things were grouped. In most cases the grouped elements of the *GRangesList* object will be listed in the order that they occurred along the chromosome. However, when exons or CDS are grouped by transcript, they will instead be grouped according to their position along the transcript itself. This is important because alternative splicing can mean that the order along the transcript can be different from that along the chromosome.

3.4 Prespecfied grouping functions

The `intronsByTranscript`, `fiveUTRsByTranscript` and `threeUTRsByTranscript` are convenience functions that provide behavior equivalent to the grouping functions, but in prespecified form. These functions return a *GRangesList* object grouped by transcript for introns, 5' UTR's, and 3' UTR's, respectively.

```
> length(intronsByTranscript(txdb))

[1] 135

> length(fiveUTRsByTranscript(txdb))

[1] 61

> length(threeUTRsByTranscript(txdb))

[1] 58
```

3.5 Convenience functions for computing overlap

The `transcriptsByOverlaps`, `exonsByOverlaps` and `cdsByOverlaps` functions return a *GRangesList* object containing data about transcripts, exons, or coding sequences that overlap genomic coordinates specified by a *GRanges* object. So for example, lets just mock up some fake data:

```
> gr <- GRanges(
+     seqnames = rep("chr5",4),
+     ranges = IRanges(start = c(244620, 244670, 245804, 247502),
+                      end = c(244652, 244702, 245836, 247534)),
+     strand = rep("+", 4))
```

Then we can call the convenience function to see what transcripts overlap with our ranges.

```
> transcriptsByOverlaps(txdb, gr)
```

```
GRanges with 1 range and 2 elementMetadata values
    seqnames           ranges strand |     tx_id     tx_name
       <Rle>        <IRanges>  <Rle> | <integer> <character>
[1]     chr5 [244626, 248468]      + |        14  uc003jal.1

seqlengths
          chr1   chr1_random         chr10 ...   chrX_random          chrY
     247249719       1663265     135374737 ...       1719168      57772954
```

The convenience functions can be a great shortcut, but because they have to make assumptions about how the results are compared and represented, they are ultimately not as flexible as just using the basic and grouping accessors in combination with `findOverlaps`.

4 A typical example treating gr as RNA-seq data

Let's suppose that you have run an experiment. After mapping all your reads to a genome and collapsing them into a set of ranges, you want to find out which genomic features a particular range overlaps with. What would be the usual way to proceed?

For this example, let's also assume that you are only interested in mapping the ranges that overlap with exons (not introns). From our *TranscriptDb* object, we want to recover the annotations for all of the relevant

exons, but grouped according to their transcripts. Therefore, we want to use `exonsby` and group them by transcripts.

```
> annotGr <- exonsBy(txdb, "tx")
```

Then we need to use the `findOverlaps` method to learn which of our data ranges, `gr`, will overlap with the in exons that we have grouped by transcripts.

```
> OL <- findOverlaps(query = annotGr, subject = gr)
```

Finally, once we have called `findOverlaps` we can subset out the annotations that meet our criteria. The `queryHits` method will allow us to retrieve only the parts of the query that overlapped from our original `findOverlaps` call. Once we have subsetted out annotations in this way, the length of the resulting *GRangesList* object is also the number of transcripts that overlap with our data.

```
> tdata <- annotGr[unique(queryHits(OL)),]
> tdata
```

```
GRangesList of length 1
$14
GRanges with 2 ranges and 3 elementMetadata values
    seqnames           ranges strand |   exon_id   exon_name exon_rank
       <Rle>        <IRanges>  <Rle> | <integer> <character> <integer>
[1]     chr5 [244626, 245552]      + |        75          NA         1
[2]     chr5 [247823, 248468]      + |        76          NA         2


seqlengths
          chr1   chr1_random         chr10 ...   chrX_random          chrY
     247249719       1663265     135374737 ...       1719168      57772954
```

```
> length(tdata)
```

```
[1] 1
```

By using `findOverlaps` along with the different accessors in this way, it is possible to connect any data that has been represented as a *GRanges* object with the annotations stored in a *TranscriptDb* object. Calling `findOverlaps` along with the appropriate *GRanges* object not only allows users

to quickly determine what has overlapped, but also controls what criteria are used for determining whether an overlap has occurred. This can be done by passing in an alternate `type` parameter to `findOverlaps`. In addition, because the basic accessors allow for the users to retrieve data grouped in different ways, the user has control over which parts of a transcript or gene are included in the overlap. For a more complete example of how you could approach RNA-seq analysis, including explanation on methods that will help to tally up the counts etc. please see the *GenomicRanges* Use Cases vignette.

5 Session Information

```
R version 2.12.0 Patched (2010-11-28 r53696)
Platform: x86_64-unknown-linux-gnu (64-bit)

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=C
 [5] LC_MONETARY=C              LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] GenomicFeatures_1.2.3 GenomicRanges_1.2.1   IRanges_1.8.5

loaded via a namespace (and not attached):
 [1] BSgenome_1.18.2    Biobase_2.10.0     Biostrings_2.18.2  DBI_0.2-5
 [5] RCurl_1.5-0        RSQLite_0.9-4      XML_3.2-0          biomaRt_2.6.0
 [9] rtracklayer_1.10.5 tools_2.12.0
```