`proftools` Implementation

November 4, 2015

For now, this file is used to generate the `prof.R` source file. Eventually it is intended that this file document the entire implementation, but this is very much work in progress.

1 Reading Profile Data

The function `readPD` reads the data file produced by the `Rprof` profiling mechanism.

⟨*[[readPD]] function*⟩=
```
 readPD <- function(file) {
     con <- file(file, "r")
     on.exit(close(con))

     hdr <- readPDheader(con)
     data <- readPDlines(con, hdr)
     sdata <- splitStacks(data)
     counts <- countStacks(sdata$trace, sdata$inGC)
     structure(c(hdr, sdata, counts), class =  "proftools_profData")
 }
```

The result is given a class to allow it to be distinguished from the data structure returned in earlier versions of the package and to allow a custom print method.

The data file consists of a header line followed by stack trace lines. The header is read fro an open connection by `readPDheader`. For now, memory profiling is not supported, so an error is signaled if the header indicates that memory profiling was enabled. The result returned is a structure containing the sampling interval and flags indicating whether GC recording or source line recording were active.

⟨*[[readPDheader]] function*⟩=
```
 readPDheader <- function(con) {
     header <- readLines(con, 1)

     pat <- ".*sample.interval=([[:digit:]]+).*"
     if (grepl(pat, header))
```

```
        interval <- as.integer(sub(pat, "\\1", header))
    else
        stop("not a valid Rprof file")

    haveMem <- grepl("memory profiling:", header)
    haveGC <- grepl("GC profiling:", header)
    haveRefs <- grepl("line profiling:", header)

    if (haveMem)
        stop("memory profiling is currently not supported")

    list(interval = interval, haveGC = haveGC, haveRefs = haveRefs)
}
```

The stack trace lines are read by `readPDlines`. The implementation reads all lines at once and processes them in memory. It would be possible to read the data in chunks, but this doesn't seem worth while given memory typically available. The returned result contains the files referenced in the source references, a vector of the unique stack traces, an index vector of length equal to the number of lines specifying which trace each line corresponds to, and a logical vector of length equal to the number of lines indicating whether that line was written during a GC. Lines containing only a GC entry are very unlikely (if they are possible at all) and are deleted.

⟨*[[readPDlines]] function*⟩=

```
readPDlines <- function(con, hdr) {
    stacks <-  readLines(con)
    if (hdr$haveRefs) {
        fstacks <- grepl("#File ", stacks)
        files <- sub("#File [[:digit:]]: (.+)", "\\1", stacks[fstacks])
        stacks <- stacks[! fstacks]
    }
    else files <- NULL

    ## remove any lines with only a <GC> entry
    onlyGC <- grepl("^\"<GC>\" $", stacks)
    if (any(onlyGC))
        stacks <- stacks[! onlyGC]

    ## record and strip out GC info
    inGC <- grepl("<GC>", stacks)
    stacks <- sub("\"<GC>\" ", "", stacks)

    ustacks <- unique(stacks)
    trace <- match(stacks, ustacks)
```

```
    list(files = files, stacks = ustacks, trace = trace, inGC = inGC)
}
```

The unique stacks are split into the function calls and their source references. Need to explain assumptions/conventions used here about references.

⟨*[[splitStacks]] function*⟩=

```
splitStacks <- function(data) {
    stacks <- data$stacks
    rsstacks <- lapply(strsplit(stacks, " +"),
                function(x) sub("\"(.+)\"", "\\1", rev(x)))
    refpat <- "[[:digit:]]+#[[:digit:]]+$"
    stacks <- lapply(rsstacks, function(x) x[! grepl(refpat, x)])
    stackrefs <- lapply(rsstacks, function(x) {
        isref <- grepl(refpat, x)
        stack <- x[! isref]
        refs <- rep(NA_character_, length(stack) + 1)
        k <- 1
        i <- 1
        n <- length(x)
        while (i <= n) {
            if (isref[i]) {
                refs[k] <- x[i]
                i <- i + 2
            }
            else
                i <- i + 1
            k <- k + 1
        }
        refs
    })
    data$stacks <- stacks
    data$refs <- stackrefs
    data
}
```

The function `countStacks` counts the number of times each unique stack trace occurs and the number of times a GC occurs in the trace. The result returned contains the counts and the GC counts as well as the total count.

⟨*[[countStacks]] function*⟩=

```
countStacks <- function(trace, inGC) {
    counts <- as.numeric(table(trace))
    gccounts <- as.numeric(tapply(inGC, trace, sum))
```

```
    list(counts = counts, gccounts = gccounts, total = sum(counts))
}
```

The custom print method prints the sample count, run time, and the head of the function summary, followed by an ellipsis if rows have been dropped.

⟨*[[print.proftools_profData]] method*⟩=
```
print.proftools_profData <- function(x, n = 6, ...) {
    cat(sprintf("Samples: %d\n", x$total))
    cat(sprintf("Time:    %.2f\n", x$total * (x$interval / 1000000)))
    fs <- funSummary(x)
    print(head(fs, n), ...)
    if (n < nrow(fs)) cat("...\n")
}
```

To integrate with the original `proftools` package `readProfileData` is available as an alias for `readPD`

⟨*[[readProfileData]] function*⟩=
```
readProfileData <- function(filename = "Rprof.out")
    readPD(filename)
```

2 Operations on Profile Data

Raw profile data may contain references to function calls that take little time or for other reasons are not of primary interest. Filtering these out can make it easier to detect true points of concern. Filtering can be done at the data level or the summary level. Most summary functions produce data frames as their results; these can be filtered using standard data frame operations such as `head` and `subset`. This section presents filtering functions that operate on the profile data itself.

2.1 `focusPD`

A useful high level function is `focusPD`. This function takes a profile data object and a character vector of function names and returns a profile data object of stack frames containing only at least one of the specified functions and with all functions leading up to the specified ones removed. As an option, if `drop` is not true then the stack frames not containing any of the specified functions are replaced by placeholder frames.

⟨*old [[focusPD]] function*⟩=
```
focusPD <- function(pd, which, drop = TRUE, regex = TRUE) {
    ## **** check that 'which' is character?
    if (drop)
        skipPD(subsetPD(pd, which, regex = regex), which, regex = regex)
    else
        skipPD(pd, which, TRUE, regex)
}
```

A revised version removes the `drop` argument and adds the option of trimming functions with low `self` percentage from the bottom of the stack or low `total` percentages from the top of the stack:

⟨*[[focusPD]] function*⟩=
```
focusPD <- function(pd, which, self.pct = 0, total.pct = 0, regex = TRUE) {
    if (self.pct > 0 || total.pct > 0) {
        fs <- funSummary(pd, srclines = FALSE)
        funs <- rownames(fs)
        if (self.pct > 0)
            pd <- focusPD(pd, funs[fs$self.pct >= self.pct], regex = FALSE)
        if (total.pct > 0) {
            high <- funs[fs$total.pct >= total.pct]
            pd <- focusPD(pd, high, regex = FALSE)
            pd <- trimTop(pd, setdiff(funs, high))
        }
    }
    if (missing(which))
        pd
    else
        skipPD(subsetPD(pd, which, regex = regex), which, regex = regex)
}
```

The support function `trimTop` handles trimming back the top of the stack. It may be possible to do this just with `prunePD`.

⟨*[[focusPD]] function*⟩=
```
trimTop <- function(pd, funs) {
    to <- sapply(pd$stacks, function(s) {
        notLow <- ! s %in% funs
        if (any(notLow)) max(which(notLow)) else 0
    })
    pd <- prunePD(pd, to = to)
    drop <- sapply(pd$stacks, function(s) all(s %in% funs))
    subsetPD(pd, omit = drop)
}
```

2.2 subsetPD

The `subsetPD` function takes a profile data object and a specification for the stack traces to keep and returns a new profile data object containing only the specified stack frames. The frames to keep can be specified as a vector of indices, as a logical vector, or as a character vector. A character vector is taken to contain regular expressions to match names of functions, and only stack frames containing calls to at least one matching function are retained. Frames to omit can be specified analogously.

The first step is to convert logical or character specifications into indices of frames to keep or omit. This is done by `subsetIDX`:

⟨*[[subsetIDX]] function*⟩=
```
subsetIDX <- function(idx, pd, regex) {
    isIn <- if (regex) patMatchAny else function(x, s) x %in% s
    if (is.character(idx))
        which(sapply(pd$stacks, function(s) any(isIn(idx, s))))
    else if (is.logical(idx))
        which(idx)
    else
        idx
}
```

The function `patMatchAny` uses `grepl` to do its work.

⟨*[[patMatchAny]] function*⟩=
```
patMatchAny <- function(p, x) {
    v <- grepl(p[1], x)
    for (q in p[-1])
        v <- v | grepl(q, x)
    v
}
```

The first step in `subsetPD` is to create a vector of indices into the original stacks to be used in remapping the reduces trace into the new stacks and to call `subsetIDX` to convert the subset specifications into indices.

⟨*Convert specification to indices and save index vector*⟩=
```
stackIDX <- seq_along(pd$stacks)
if (missing(select))
    select <- stackIDX
keep <- subsetIDX(select, pd, regex)
if (! missing(omit))
    keep <- setdiff(keep, subsetIDX(omit, pd, regex))
```

Next, the stacks, references, and counts are reduced to the ones specified in `keep`.

⟨*reduce stacks and counts*⟩=
```
pd$stacks <- pd$stacks[keep]
pd$refs <- pd$refs[keep]
pd$counts <- pd$counts[keep]
pd$gccounts <- pd$gccounts[keep]
```

Then omitted stack traces are dropped from `trace` and the `inGC` indicators.

```
⟨drop omittes stacks from [[trace]] and [[inGC]]⟩=
 traceKeep <- which(pd$trace %in% keep)
 pd$inGC <- pd$inGC[traceKeep]
 pd$trace <- pd$trace[traceKeep]
```

At this point the indices in `trace` refer to the original set of stack traces; these need to be remapped to the new set.

```
⟨remap [[trace]] to new stack traces⟩=
 map <- match(stackIDX, keep)
 pd$trace <- map[pd$trace]
```

The complete definition of `subsetPD` is then

```
⟨[[subsetPD]] function⟩=
 subsetPD <- function(pd, select, omit, regex = TRUE) {
     ⟨Convert specification to indices and save index vector⟩

     ⟨reduce stacks and counts⟩

     ⟨drop omittes stacks from [[trace]] and [[inGC]]⟩

     ⟨remap [[trace]] to new stack traces⟩

     pd
 }
```

2.3 Transformation Utilities

The remaining transformation functions are all based on rewriting the stacks and references and possibly recomputing the counts after eliminating duplicate stacks introduced by the rewriting. The common pattern is abstracted into the function `transformPD`. This function takes a profile data object and a transformation function as arguments. The transformation function in turn is called with each stack and reference pair, along with the index of the pair in the stack sequence. Why is the index needed?

```
⟨[[transformPD]] function⟩=
 transformPD <- function(pd, fun) {
     stacks <- pd$stacks
     refs <- pd$refs
     nf <- length(pd$files)
     for (i in seq_along(pd$stacks)) {
         val <- checkStackRefs(fun(stacks[[i]], refs[[i]], i), nf)
         stacks[[i]] <- val$stack
         refs[[i]] <- val$refs
```

```
    }
    pd$stacks <- stacks
    pd$refs <- refs

    compactPD(pd)
}
```

The results returned by the stack transformation function are checked for some inconsistencies; in particular they must contain at least one entry.

⟨*[[checkStackRefs]] function*⟩=

```
checkStackRefs <- function(val, nf) {
    s <- val$stack
    r <- val$refs
    if (length(s) == 0)
        stop("stacks must have at least one entry")
    if (length(r) != length(s) + 1)
        stop("stack and source references do not match")
    fn <- refFN(r)
    fn <- fn[! is.na(fn)]
    if (length(fn) > 0 && (min(fn) < 1 || max(fn) > nf))
        stop("invalid source references produced.")
    val
}
```

Since the transformation might introduce duplicate stacks the result needs to be processed to identify unique stacks and recompute counts. Make a `mapply0` function that does not simplify?

⟨*[[compactPD]] function*⟩=

```
compactPD <- function(pd) {
    ## **** need SIMPLIFY=FALSE here since mapply creates a matrix if all
    ## **** elements of the result happen to be the same length. Might
    ## **** be more robust to use paste() rather than c() (probably
    ## **** what I intended originally).
    key <- mapply(c, pd$stacks, pd$refs, SIMPLIFY = FALSE)
    map <- match(key, unique(key))
    ct <- aggregateCounts(data.frame(key = map),
                          cbind(counts = pd$counts, gccounts = pd$gccounts))
    ct <- ct[order(ct$key),] ## may not be needed
    invmap <- match(unique(key), key)
    pd$stacks <- pd$stacks[invmap]
    pd$refs <- pd$refs[invmap]
    pd$counts <- ct$counts
    pd$gccounts <- ct$gccounts
    pd$trace <- map[pd$trace]
```

```
    pd
}
```

2.4 skipPD

`skipPD` removes a specified number of initial calls from each stack frame, or all calls preceding the first of a specified set of functions to occur in the frame. If a stack frams is shorter or does not contains any of the specified functions, then it is left unchanged if `merge` is false; otherwise, it is replaced by a shortened place-holder stack.

⟨*[[skipPD]] function*⟩=

```
skipPD <- function(pd, what, merge = FALSE, regex = TRUE) {
    idx <- skipIDX(pd, what, regex)

    skip <- function(stack, refs, i) {
        n <- idx[i]
        if (n > 0) {
            if (n < length(stack)) {
                skip <- 1 : n
                stack <- stack[-skip]
                refs <- refs[-skip]
            }
            else {
                stack <- OtherFunsToken
                refs <- c(NA_character_, NA_character_)
            }
        }
        else if (merge) {
            stack <- OtherFunsToken
            refs <- c(NA_character_, NA_character_)
        }
        list(stack = stack, refs = refs)
    }

    transformPD(pd, skip)
}
```

The place-holder token is

⟨*[[OtherFunsToken]] constant*⟩=

```
OtherFunsToken <- "<Other>"
```

The function `skipIDX` converts a function name specification to the corresponding indices to skip.

```
⟨[[skipIDX]] function⟩=
 skipIDX <- function(pd, what, regex) {
     if (regex)
         isIn <- function(s, x) sapply(s, function(s) any(patMatchAny(x, s)))
     else
         isIn <- function(s, x) s %in% x
     if (is.character(what)) {
         findFirst <- function(s) {
             idx <- isIn(s, what)
             if (any(idx))
                 min(which(idx)) - 1
             else
                 0
         }
         idx <- sapply(pd$stacks, findFirst)
     }
     else
         idx <- ifelse(is.na(what), 0, what)
     if (length(idx) != length(pd$stacks))
         idx <- rep(idx, length = length(pd$stacks))
     idx
 }
```

2.5 prunePD

`prunePD` Trims entries off the top of the stacks. The `to` argument can be an integer or integer vector specifying the height to prune to, or a character vector specifying functions whose callees should be pruned. Alternatively, the `by` argument can be used to specify that a specified number of calls should be removed from the top. Again stack frames that do not match the criteria or have to few elements are either left unchanged or replaced by a place-holder, depending on the `merge` argument. `merge` is actually ignored at the moment – fix that?

```
⟨[[prunePD]] function⟩=
 prunePD <- function(pd, to, by, merge = FALSE) {
     if (missing(by))
         idx <- pruneIDX(pd, if (is.character(to)) to else -to)
     else
         idx <- pruneIDX(pd, by)

     ⟨[[prune]] function⟩

     transformPD(pd, prune)
 }
```

The `prune` transformation function interprets negative prune amounts as a height to prune to. Any negative indices are first converted to amounts to remove.

⟨*[[prune]] function*⟩=
```
prune <- function(stack, refs, i) {
    n <- idx[i]
    slen <- length(stack)

    if (n < 0)
        if (-n < slen)
            n <- slen + n
        else
            n <- 0

    if (n > 0) {
        if (n < slen) {
            drop <- (slen - n + 1) : slen
            stack <- stack[-drop]
            refs <- refs[-(drop + 1)]
        }
        else {
            stack <- OtherFunsToken
            refs <- c(NA_character_, NA_character_)
        }
    }
    list(stack = stack, refs = refs)
}
```

`pruneIDX` converts a character specification to the corresponding `by` argument. `NA` values are converted to zero, and the index vector is recycled if necessary

⟨*[[pruneIDX]] function*⟩=
```
pruneIDX <- function(pd, what) {
    if (is.character(what)) {
        findPrune <- function(s) {
            idx <- match(what, s)
            if (any(! is.na(idx)))
                length(s) - min(idx, na.rm = TRUE)
            else
                0
        }
        idx <- sapply(pd$stacks, findPrune)
    }
    else
```

```
        idx <- ifelse(is.na(what), 0, what)
    if (length(idx) != length(pd$stacks))
        idx <- rep(idx, length = length(pd$stacks))
    idx
}
```

2.6 Merging GC information

At times it may be more convenient to have calls to GC treated like calls to other functions. `mergeGC` does the conversion.

⟨*[[mergeGC]] function*⟩=
```
mergeGC <- function(pd) {
    ⟨[[mergeGC]] body⟩
}
```

The first step saves the old stacks and records the indices of the stacks that call GC.

⟨*[[mergeGC]] body*⟩=
```
ostacks <- pd$stacks
hasGC <- which(pd$gccounts > 0)
```

Handle the trailing space better in woven output.

Stacks with GC calls appended are added to the stack list; corresponding references are added as as well.

⟨*[[mergeGC]] body*⟩=
```
pd$stacks <- c(pd$stacks, lapply(pd$stacks[hasGC], c, "<GC>"))
pd$refs <- c(pd$refs, lapply(pd$refs[hasGC], c, NA_character_))
```

The `counts` for the original stacks are reduced by their `gccounts`, and the `gccounts` for stacks with GC calls become the `counts` for the new stacks.

⟨*[[mergeGC]] body*⟩=
```
pd$counts <- c(pd$counts - pd$gccounts, pd$gccounts[hasGC])
pd$gccounts <- rep(0, length(pd$counts))
```

The `trace` indices for entries during GC calls are updated to point to the new stacks with GC calls appended.

⟨*[[mergeGC]] body*⟩=
```
map <- match(seq_along(ostacks), hasGC) + length(ostacks)
pd$trace[pd$inGC] <- map[pd$trace[pd$inGC]]
```

Finally, the `inGC` and `haveGC` fields are updated and the new `pd` is returned.

⟨*[[mergeGC]] body*⟩=
```
 pd$inGC[] <- FALSE
 pd$haveGC <- FALSE

 pd
```

2.7 Selecting Part of a Trace

⟨*[[timeSubsetPD]] function*⟩=
```
 timeSubsetPD <- function(pd, interval) {
     lo <- interval[1]
     hi <- interval[2]
     idx <- lo : hi
     pd$trace <- pd$trace[idx]
     pd$inGC <- pd$inGC[idx]
     sidx <- unique(pd$trace)
     pd$trace <- match(seq_along(pd$stacks), sidx)[pd$trace]
     pd$stacks <- pd$stacks[sidx]
     pd$refs <- pd$refs[sidx]
     cs <- countStacks(pd$trace,pd$inGC)
     pd$counts <- cs$counts
     pd$gccounts <- cs$gccounts
     pd
 }
```

2.8 A Higher Level Function

**** skip argument to skip first n? Don't create `<Other>` entry? **** are focus/regex default values right?

⟨*[[filterProfileData]] function*⟩=
```
 filterProfileData <- function(pd, select, omit,
                               skip = NA,
                               maxdepth = NA,
                               self.pct = 0, total.pct = 0,
                               focus = FALSE,
                               interval,
                               normalize = FALSE,
                               regex = FALSE) {
     if (!is.na(skip))
         pd <- skipPD(pd, skip)
     if (! is.na(maxdepth))
```

```
        pd <- prunePD(pd, to = maxdepth)
    pd <- focusPD(pd, self.pct = self.pct, total.pct = total.pct)
    if (! missing(select))
        if (focus)
            pd <- focusPD(pd, select, regex = regex)
        else
            pd <- subsetPD(pd, select, regex = regex)
    if (! missing(omit))
        pd <- subsetPD(pd, omit = omit, regex = regex)
    if (! missing(interval))
        pd <- timeSubsetPD(pd, interval)
    if (normalize)
        pd$total <- sum(pd$counts)
    pd
}
```

An alternate version that allows a sequence of filters to be applied, including repetitions:

⟨*[[filterProfileData]] function*⟩=

```
fpd <- function(pd, ..., normalize = FALSE, regex = FALSE)
{
    fargs <- list(...)
    fnames <- names(fargs)
    if (is.null(fnames) || any(nchar(fnames) == 0))
        stop("all filters must be named")
    for (i in seq_along(fargs)) {
        pd <- switch(fnames[i],
                     skip = skipPD(pd, fargs[[i]]),
                     maxdepth = prunePD(pd, to = fargs[[i]]),
                     self.pct = focusPD(pd, self.pct = fargs[[i]]),
                     total.pct = focusPD(pd, total.pct = fargs[[i]]),
                     focus = focusPD(pd, fargs[[i]], regex = regex),
                     select = subsetPD(pd, fargs[[i]], regex = regex),
                     omit = subsetPD(pd, omit = fargs[[i]], regex = regex),
                     interval = timeSubsetPD(pd, fargs[[i]]),
                     stop("unknown filter: ", fnames[i]))
    }
    if (normalize)
        pd$total <- sum(pd$counts)
    pd
}
```

Need to make sure that the total.pct and self.pct filters are doing the right thing. Also need to document this alternate version.

3 Summary Functions

3.1 Hot Paths

The hot path summary is based loosely on an idea described in an MSDN Article. The stack traces, or call paths, are sorted lexicographically based on the number of hits in the first function on the stack, then the first two functions, and so on. The display abbreviates the paths in an obvious way to make the display more compact and readable. The cumulative cost for each level is shown; self costs can also be shown as an option. A sample based on profiling an `lm` fit looks like this:

```
> head(hotPaths(focusPD(d, "lm")), 10)
 path                                    total.pct gc.pct
 lm                                      100.0     75.3
 . model.response                         51.1     41.4
 . . as.character                         50.9     41.4
 . . attr                                  0.0      0.0
 . lm.fit                                 34.1     30.6
 . . .Call                                11.6      9.9
 . . c                                     9.0      8.3
 . . list                                  4.6      4.2
 . . structure                             4.5      4.1
 . . rep.int                               2.4      2.3
```

The maximal stack depth is set to 10 by default, again for readability.

The `hotPaths` function uses `prunePD` to limit the stack depth to the level `maxdepth` before ordering the data and computing summary counts. The counts are then processed into a data frame showing either percent, time, or hits. The defaults used here seem to work reasonably well but could probably use more tuning.

⟨*[[hotPaths]] function*⟩=

```
 hotPaths <- function(pd, value = c("pct", "time", "hits"),
                      self = TRUE, srclines = FALSE, GC = FALSE,
                      maxdepth = 10, self.pct = 0, total.pct = 0,
                      short = ". ", nlines = NA) {
     value <- match.arg(value)
     if (! is.na(maxdepth))
         pd <- prunePD(pd, maxdepth)

     if (! srclines && pd$haveRefs) pd <- stripRefs(pd)

     data <- hotPathData(pd)

     if (! is.na(nlines) && length(data$count) > nlines) {
         top <- head(order(data$count, decreasing = TRUE), nlines)
         data <- data[1:nrow(data) %in% top,]
```

```
    }

    if (self.pct > 0)
        data <- data[data$self >= pd$total * (self.pct / 100),]
    if (total.pct > 0)
        data <- data[data$count >= pd$total * (total.pct / 100),]

    data$pa <- pathAbbrev(data$key, short)

    if (value == "pct")
        hotPathsPct(data, self, GC && pd$haveGC, pd$total)
    else if (value == "time")
        hotPathsTime(data, self, GC && pd$haveGC, pd$interval / 1.0e6)
    else
        hotPathsHits(data, self, GC && pd$haveGC)
}
```

If source references are not needed they are first removed using `stripRefs`.

⟨*[[stripRefs]] function*⟩=

```
stripRefs <- function(pd) {
    pd$refs <- lapply(pd$refs, function(r) rep(NA_character_, length(r)))
    pd <- compactPD(pd)
    pd$haveRefs <- FALSE
    pd
}
```

The order is computed by `hotPathOrd`. This function orders each call level according to the number of hits within the call chain up to that point.

⟨*[[hotPathOrd]] function*⟩=

```
hotPathOrd <- function(stacks, counts) {
    mx <- max(sapply(stacks, length))
    ord <- seq_along(stacks)
    for (i in (mx : 1)) {
        key <- sapply(stacks[ord], `[`, i)
        tbl <- aggregate(list(val = -counts[ord]), list(key = key), sum)
        val <- tbl$val[match(key, tbl$key)]
        ord <- ord[order(val, na.last = TRUE)]
    }
    ord
}
```

The `hotPathData` function computes the stack order, the successive call paths, and the cumulative counts and returns these in a data frame.

⟨*[[hotPathData]] function*⟩=
```
 hotPathData <- function(pd) {
     files <- pd$files
     pathLabels <- function(s, t) {
         n <- length(s)
         if (n == 0)
             funLabels(UnknownFunToken, t[1], files)
         else if (is.na(t[n + 1]))
             funLabels(s, t[1:n], files)
         else
             funLabels(c(s, UnknownFunToken), t, files)
     }

     pl <- mapply(pathLabels, pd$stacks, pd$refs, SIMPLIFY = FALSE)
     ord <- hotPathOrd(pl, pd$counts)
     stacks <- pl[ord]
     counts <- pd$counts[ord]
     gccounts <- pd$gccounts[ord]

     pathData <- function(k) {
         s <- stacks[[k]]
         keys <- sapply(1 : length(s),
                        function(i) paste(s[1 : i], collapse = " -> "))
         data.frame(key = keys,
                    count = counts[k],
                    gccount = gccounts[k])
     }
     tbl <- do.call(rbind, lapply(1:length(stacks), pathData))

     ## **** turn off stringsAsFactore in aggregate?
     data <- aggregate(list(count = tbl$count, gccount = tbl$gccount),
                       list(key = tbl$key),
                       sum)
     data$key <- as.character(data$key)

     selfidx <- match(data$key, sapply(stacks, paste, collapse = " -> "))
     data$self <- ifelse(is.na(selfidx), 0, counts[selfidx])
     data$gcself <- ifelse(is.na(selfidx), 0, gccounts[selfidx])
     data
 }
```

The token used for an unknown function is

⟨*[[UnknownFunToken]] constant*⟩=
```
 UnknownFunToken <- "??"
```

Call labels are created by merging the function name and the source reference, if source references are used.

⟨*[[funLabels]] function*⟩=
```
 funLabels <- function(fun, site, files) {
     if (all(is.na(site)))
         fun
     else {
         file <- basename(files[refFN(site)])
         line <- refLN(site)
         funsite <- sprintf("%s (%s:%d)", fun, file, line)
         ifelse(is.na(site), fun, funsite)
     }
 }
```

The functions `refFN` and `refLN` extract the file indices and line numbers from source references of the form `FN#LN`.

⟨*[[refFN]] and [[refLN]] functions*⟩=
```
 refFN <- function(refs)
     as.integer(sub("([[:digit:]]+)#[[:digit:]]+", "\\1", refs))

 refLN <- function(refs)
     as.integer(sub("[[:digit:]]+#([[:digit:]]+)", "\\1", refs))
```

Path abbreviations are constructed using `pathAbbrev`. The abbreviation used for functions farther down the call stack is specified in the `short` argument. The strings are padded to be the same length so hey appear reasonably when printed with either left or right adjustment.

⟨*[[pathAbbrev]] function*⟩=
```
 pathAbbrev <- function(paths, short) {
     pad <- function(n) paste(rep(short, n), collapse = "")
     sapply(strsplit(paths, " -> "),
            function(path) {
                n <- length(path)
                label <- if (n > 1) paste0(pad(n - 1), path[n]) else path
            })
 }
```

Final results are constructed by `hotPathsPct`, `hotPathsHits`, or `hotPathsTime`. The result is a data frame, but with a subclass to allow a custom print method to be used.

⟨*[[hotPathsPct]], [[hotPathsHits]], and [[hotPathsTime]] functions*⟩=

```
hotPathsPct <- function(data, self, gc, grandTotal) {
    pa <- data$pa
    val <- data.frame(path = sprintf(sprintf("%%-%ds", max(nchar(pa))), pa),
                      total.pct = percent(data$count, grandTotal),
                      stringsAsFactors = FALSE)
    if (gc) val$gc.pct = percent(data$gccount, grandTotal)
    if (self) {
        val$self.pct = percent(data$self, grandTotal)
        if (gc) val$gcself.pct = percent(data$gcself, grandTotal)
    }
    class(val) <- c("proftools_hotPaths", "data.frame")
    val
}

hotPathsHits <- function(data, self, gc) {
    pa <- data$pa
    val <- data.frame(path = sprintf(sprintf("%%-%ds", max(nchar(pa))), pa),
                      total.hits = data$count,
                      stringsAsFactors = FALSE)
    if (gc) val$gc.hits = data$gccount
    if (self) {
        val$self.hits = data$self
        if (gc) val$gcself.hits = data$gcself
    }
    class(val) <- c("proftools_hotPaths", "data.frame")
    val
}

hotPathsTime <- function(data, self, gc, delta) {
    pa <- data$pa
    val <- data.frame(path = sprintf(sprintf("%%-%ds", max(nchar(pa))), pa),
                      total.time = data$count * delta,
                      stringsAsFactors = FALSE)
    if (gc) val$gc.time = data$gccount * delta
    if (self) {
        val$self.time = data$self * delta
        if (gc) val$gcself.time = data$gcself * delta
    }
    class(val) <- c("proftools_hotPaths", "data.frame")
    val
}
```

The custom print method suppresses the row labels and specifies left adjust ment to make the path label appear in a more natural place.

⟨*[[print.proftools_hotPaths]] method*⟩=
```
print.proftools_hotPaths <- function(x, ..., right = FALSE, row.names = FALSE)
    print.data.frame(x, ..., right = right, row.names = row.names)
```

To ensure consistency rounded percentages are computes by a common function `percent`:

⟨*[[percent]] function*⟩=
```
percent <- function(x, y) round(100 * x / y, 2)
```

3.2 Function and Call Summaries

The functions `funSummary` and `callSummary` compute percent, time, or hit summaries for individual functions and calls. If source references are enabled, then the function summaries are broken down by source location where the call occurs. For calls, summaries are broken down according to where the caller and callee are called. In addition, entries with self or total values below a specified threshold can be dropped. The function `funSummary` is defined as

⟨*[[funSummary]] function*⟩=
```
funSummary <- function(pd, byTotal = TRUE,
                       value = c("pct", "time", "hits"),
                       srclines = TRUE,
                       GC = TRUE, self.pct = 0, total.pct = 0) {
    value <- match.arg(value)

    fc <- funCounts(pd, srclines)
    if (byTotal)
        fc <- fc[rev(order(fc$total)), ]
    else
        fc <- fc[rev(order(fc$self)), ]

    if (self.pct > 0)
        fc <- fc[fc$self >= pd$total * (self.pct / 100),]
    if (total.pct > 0)
        fc <- fc[fc$total >= pd$total * (total.pct / 100),]

    label <- funLabels(fc$fun, fc$site, pd$files)

    if (value == "pct")
        funSummaryPct(fc, label, GC && pd$haveGC, pd$total)
    else if (value == "time")
        funSummaryTime(fc, label, GC && pd$haveGC, pd$interval / 1.0e6)
    else
```

```
        funSummaryHits(fc, label, GC && pd$haveGC)
}
```

The definition of `callSummary` is similar.

⟨*[[callSummary]] function*⟩=

```
callSummary <- function(pd, byTotal = TRUE,
                        value = c("pct", "time", "hits"),
                        srclines = TRUE,
                        GC = TRUE, self.pct = 0, total.pct = 0) {
    value <- match.arg(value)

    cc <- callCounts(pd, srclines, srclines)
    if (byTotal)
        cc <- cc[rev(order(cc$total)), ]
    else
        cc <- cc[rev(order(cc$self)), ]

    if (self.pct > 0)
        cc <- cc[cc$self >= pd$total * (self.pct / 100),]
    if (total.pct > 0)
        cc <- cc[cc$total >= pd$total * (total.pct / 100),]

    caller.label <- funLabels(cc$caller, cc$caller.site, pd$files)
    callee.label <- funLabels(cc$callee, cc$callee.site, pd$files)
    label <- paste(caller.label, callee.label, sep = " -> ")

    if (value == "pct")
        funSummaryPct(cc, label, GC && pd$haveGC, pd$total)
    else if (value == "time")
        funSummaryTime(cc, label, GC && pd$haveGC, pd$interval / 1.0e6)
    else
        funSummaryHits(cc, label, GC && pd$haveGC)
}
```

The summaries for percent, time, and hits are computed by `funSummaryPct`, `funSummaryTime`, and `funSummaryHits`. Rename these since they are used by both `funSummary` and `callSummary` (and maybe could be by hot paths)?

⟨*[[funSummaryPct]], [[funSummaryTime]], and [[funSummaryHits]] functions*⟩=

```
funSummaryPct <- function(fc, label, gc, grandTotal) {
    pct <- percent(fc$total, grandTotal)
    spct <- percent(fc$self, grandTotal)
    if (gc) {
        gcpct <- percent(fc$gctotal, grandTotal)
```

```
        sgcpct <- percent(fc$gcself, grandTotal)
        data.frame(total.pct = pct, gc.pct = gcpct,
                   self.pct = spct, gcself.pct = sgcpct,
                   row.names = label)
    }
    else
        data.frame(total.pct = pct, self.pct = spct, row.names = label)
}

funSummaryTime <- function(fc, label, gc, delta) {
    tm <- fc$total * delta
    stm <- fc$self * delta
    if (gc) {
        gctm <- fc$gctotal * delta
        sgctm <- fc$gcself * delta
        data.frame(total.time = tm, gc.time = gctm,
                   self.time = stm, gcself.time = sgctm,
                   row.names = label)
    }
    else
        data.frame(total.time = tm, self.time = stm, row.names = label)
}

funSummaryHits <- function(fc, label, gc) {
    if (gc)
        data.frame(total.hits = fc$total, gc.hits = fc$gctotal,
                   self.hits = fc$self, gcself.hits = fc$gcself,
                       row.names = label)
    else
        data.frame(total.hits = fc$total, self.hits = fc$self,
                   row.names = label)
}
```

4 Implementation File

⟨*prof.R*⟩=

```
###
### Read profile data
###
```

⟨*[[readPDheader]] function*⟩

⟨*[[readProfileData]] function*⟩

⟨*[[readPDlines]] function*⟩

⟨*[[splitStacks]] function*⟩

⟨*[[countStacks]] function*⟩

⟨*[[readPD]] function*⟩

⟨*[[print.proftools_profData]] method*⟩

```
###
### Operations on profile data
###
```

⟨*[[filterProfileData]] function*⟩

⟨*[[subsetIDX]] function*⟩

⟨*[[patMatchAny]] function*⟩

⟨*[[subsetPD]] function*⟩

⟨*[[focusPD]] function*⟩

⟨*[[compactPD]] function*⟩

⟨*[[checkStackRefs]] function*⟩

⟨*[[transformPD]] function*⟩

⟨*[[skipIDX]] function*⟩

⟨*[[OtherFunsToken]] constant*⟩

⟨*[[skipPD]] function*⟩

⟨*[[pruneIDX]] function*⟩

⟨*[[prunePD]] function*⟩

⟨*[[mergeGC]] function*⟩

⟨*[[timeSubsetPD]] function*⟩

```
###
### Hot path summaries
###
```

⟨*[[pathAbbrev]] function*⟩

⟨*[[stripRefs]] function*⟩

⟨*[[percent]] function*⟩

```
## The Hot Path order orders each level according to the number of
## hits within the call chain upt to that point.
```
⟨*[[hotPathOrd]] function*⟩

⟨*[[UnknownFunToken]] constant*⟩

⟨*[[hotPathData]] function*⟩

⟨*[[hotPathsPct]], [[hotPathsHits]], and [[hotPathsTime]] functions*⟩

⟨*[[hotPaths]] function*⟩

⟨*[[print.proftools_hotPaths]] method*⟩

```
###
### Function and call summaries
###

fact2char <- function(d) {
    for (i in seq_along(d))
        if (is.factor(d[[i]]))
            d[[i]] <- as.character(d[[i]])
    d
}

aggregateCounts <- function(entries, counts) {
    dcounts <- as.data.frame(counts)
```

```
    clean <- function(x)
        if (any(is.na(x)))
            factor(as.character(x), exclude = "")
        else
            x
    fact2char(aggregate(dcounts, lapply(entries, clean), sum))
}

rbindEntries <- function(entries)
    as.data.frame(do.call(rbind, entries), stringsAsFactors = FALSE)

mergeCounts <- function(data, leafdata) {
    val <- merge(data, leafdata, all = TRUE)
    val$self[is.na(val$self)] <- 0
    val$gcself[is.na(val$gcself)] <- 0
    val
}

entryCounts0 <- function(pd, fun, control, names) {
    stacks <- pd$stacks
    refs <- pd$refs
    counts <- pd$counts
    gccounts <- pd$gccounts

    which <- seq_along(stacks)

    doLine <- function(i) fun(stacks[[i]], refs[[i]], control)
    entries <- lapply(which, doLine)
    edf <- rbindEntries(entries)

    reps <- unlist(lapply(entries, nrow))
    tot <- rep(counts, reps)
    gctot <- rep(gccounts, reps)
    ct <- cbind(tot, gctot, deparse.level = 0)
    colnames(ct) <- names
    aggregateCounts(edf, ct)
}

entryCounts <- function(pd, lineFun, leafFun, control) {
    aedf <- entryCounts0(pd, lineFun, control, c("total", "gctotal"))
    aledf <- entryCounts0(pd, leafFun, control, c("self", "gcself"))
    mergeCounts(aedf, aledf)
}
```

```
lineFuns <- function(line, refs, useSite) {
    if (useSite) {
        n <- length(line)
        site <- refs[-(n + 1)]
    }
    else site <- NA_character_
    unique(cbind(fun = line, site))
}

leafFun <- function(line, refs, useSite) {
    n <- length(line)
    fun <- line[n]
    site <- if (useSite) refs[n] else NA_character_
    cbind(fun, site)
}

funCounts <- function(pd, useSite = TRUE)
    entryCounts(pd, lineFuns, leafFun, useSite)

lineCalls <- function(line, refs, cntrl) {
    n <- length(line)
    if (n > 1) {
        caller <- line[-n]
        callee <- line[-1]
        if (cntrl$useCalleeSite)
            callee.site <- refs[-c(1, n + 1)]
        else
            callee.site <- NA_character_
        if (cntrl$useCallerSite)
            caller.site <- refs[-c(n, n + 1)]
        else
            caller.site <- NA_character_
    }
    else
        caller <- callee <- callee.site <- caller.site <- character()
    unique(cbind(caller, callee, caller.site, callee.site))
}

leafCall <- function(line, refs, cntrl) {
    n <- length(line)
    if (n > 1) {
        caller <- line[n - 1]
```

```
        callee <- line[n]
        if (cntrl$useCalleeSite)
            callee.site <- refs[n]
        else
            callee.site <- NA_character_
        if (cntrl$useCallerSite)
            caller.site <- refs[n - 1]
        else
            caller.site <- NA_character_
    }
    else
        caller <- callee <- callee.site <- caller.site <- character()
    cbind(caller, callee, caller.site, callee.site)
}

callCounts <- function(pd, useCalleeSite = TRUE, useCallerSite = FALSE) {
    cntrl <- list(useCalleeSite = useCalleeSite, useCallerSite = useCallerSite)
    entryCounts(pd, lineCalls, leafCall, cntrl)
}

lineRefs <- function(line, refs, useSite)
    unique(cbind(fun = "", refs = refs))

## leafRef <- function(line, refs, useSite) {
##     n <- length(line)
##     cbind(fun = "", refs = refs[n + 1])
## }
leafRef <- function(line, refs, useSite)
    cbind(fun = "", refs = NA_character_)

refCounts <- function(pd) {
    val <- entryCounts(pd, lineRefs, leafRef, TRUE)
    val$fun <- val$self <- val$gcself <- NULL
    val[! is.na(val$refs), ]
}
```

⟨*[[funSummaryPct]], [[funSummaryTime]], and [[funSummaryHits]] functions*⟩

```
## Extract the file indices and line numbers from source references of
## the form FN#LN.
```
⟨*[[refFN]] and [[refLN]] functions*⟩

⟨*[[funLabels]] function*⟩

⟨*[[funSummary]] function*⟩

⟨*[[callSummary]] function*⟩

```
###
### Flame graph and time graph
###

## For 'standard' flame graph order the stacks so they are
## alphabetical within lines within calls, with missing entires
## first. This does a lexicographic sort by sorting on the top entry
## first, then the next, and do on; since the sorts are stable this
## keeps the top levels sorted within the lower ones.
alphaPathOrd <- function(stacks, counts) {
    mx <- max(sapply(stacks, length))
    ord <- seq_along(stacks)
    for (i in (mx : 1))
        ord <- ord[order(sapply(stacks[ord], `[`, i), na.last = FALSE)]
    ord
}

## The next two functions compute the data to be used for drawing as a
## data frame with columns left, bottom, right, top, col, and label.
fgDataLine <- function(k, stacks, counts) {
    runs <- rle(sapply(stacks, `[`, k))
    lens <- runs$lengths
    n <- length(lens)
    csums <- cumsum(tapply(counts, rep(1 : n, lens), sum))

    top <- k
    bottom <- k - 1
    left <- c(0, csums[-n])
    right <- csums
    cols <- rgb(runif(n), runif(n), runif(n))
    label <- runs$values

    show <- ! is.na(label)
    nshow <- sum(show)
    data.frame(left = left[show], bottom = rep(bottom, nshow),
               right = right[show], top = rep(top, nshow),
               col = cols[show], label = label[show],
```

```
                stringsAsFactors = FALSE)
}

default.cmap <- function(x, ...) {
    y <- unique(x)
    clr <- rainbow(length(y))
    clr[match(x, y)]
}

fgData <- function(data, reorder = c("alpha", "hot", "time"),
                   colormap) {
    stacks <- data$stacks
    mx <- max(sapply(stacks, length))

    reorder <- match.arg(reorder)
    if (reorder == "time") {
        runs <- rle(data$trace)
        stacks <- stacks[runs$values]
        counts <- runs$lengths
    }
    else {
        ## ord <- seq_along(stacks)
        ## for (i in (mx : 1))
        ##     ord <- ord[order(sapply(stacks[ord], `[`, i), na.last = FALSE)]
        counts <- data$counts
        if (reorder == "hot")
            ord <- hotPathOrd(stacks, counts)
        else
            ord <- alphaPathOrd(stacks, counts)
        stacks <- stacks[ord]
        counts <- counts[ord]
    }

    val <- do.call(rbind, lapply(1 : mx, fgDataLine, stacks, counts))
    cmap <- if (! is.null(colormap)) colormap else default.cmap
    fun <- sub(" .*", "", val$label)
    val$col <- cmap(fun, val$top, val$right - val$left)
    val
}

## This is computes the data with fdData and draws the graph with base
## graphics. This is the bit we would need to change for grid, maybe
## ggplot2, and for svg output.
```

```
stdFlameGraph <- function(data, reorder, colormap, cex, main) {
    fdg <- fgData(data, reorder, colormap)
    left <- fdg$left
    bottom <- fdg$bottom
    right <- fdg$right
    top <- fdg$top
    col <- fdg$col
    label <- fdg$label

    plot(c(min(left), max(right)), c(min(bottom), max(top)),
         type = "n", axes = FALSE, xlab = "", ylab = "", main = main)

    rect(left, bottom, right, top, col = col)

    ## half-em for half-character offset used by text() when pos
    ## argument is used.
    hm <- 0.5 * strwidth("m", cex = cex)

    show <- (strheight(label, cex = cex) <= 0.9 &
             strwidth(label, cex = cex) + 2 * hm <= 0.8 * (right - left))
    if (any(show))
        text(left[show], bottom[show] + 0.4, label[show], pos = 4, cex = cex)

    invisible(structure(list(left = left, right = right,
                             bottom = bottom, top = top,
                             label = label),
                        class = "proftools_flameGraph"))
}

htmlencode <- function(x)
    sub(">", "&gt;", sub("<", "&lt;", x))

svgFlameGraph <- function(file, data, reorder, colormap, main,
                          tooltip) {
    fdg <- fgData(data, reorder, colormap)
    fdg$col <- substr(fdg$col, 1, 7) ## remove 'alpha' component from colors
    mx <- max(fdg$top)
    totalCount <- max(fdg$right)
    counts <- fdg$right-fdg$left
    percents <- round(counts*100/totalCount, 2)
    widths <- round(percents*1180/100, 2)
    y <- 33 + (mx-fdg$top)*16
    x <- 10+round(fdg$left*1180/totalCount, 2)
```

```
    col <- fdg$col
    labels <- htmlencode(fdg$label)

    svgCode = paste("<rect x=\"", x, "\" y=\"", y,
    "\" width=\"", widths, "\" height=\"15.0\" fill=\"", col,
    "\" rx=\"2\" ry=\"2\" onmouseover=\"s('", labels, " (",
    counts, " samples, ", percents, "%)')\" onmouseout=\"c()\" />",
    sep="")

    show <- (! is.na(labels) & 10*nchar(labels)<widths)
    if (any(show))
        svgCode = append(svgCode, paste("<text text-anchor=\"\" x=\"",
        x[show]+3, "\" y=\"", y[show]+10.5, "\" font-size=\"12\" font-family=\"Verdana\" fill=
        labels[show], " (", counts[show], " samples, ", percents[show],
        "%)')\" onmouseout=\"c()\" >", labels[show],"</text>", sep=""))

    writeFile(file, svgCode, mx, main, tooltip)
}
## This writes the header of the svg file
writeFile <- function(file, svgCode, mx, main, tooltip){
    write(c(paste("<?xml version=\"1.0\" standalone=\"no\"?>
    <!DOCTYPE svg PUBLIC \"-//W3C//DTD SVG 1.1//EN\" \"http://www.w3.org/Graphics/SVG/1.1/DTD/
    <svg version=\"1.1\" width=\"1200\" height=\"", 16*mx+66, "\" onload=\"init(evt)\"", if (t
    <defs >
    <linearGradient id=\"background\" y1=\"0\" y2=\"1\" x1=\"0\" x2=\"0\" >
        <stop stop-color=\"#eeeeee\" offset=\"5%\" />
        <stop stop-color=\"#eeeeb0\" offset=\"95%\" />
    </linearGradient>
    </defs>
    <style type=\"text/css\">
    rect[rx]:hover { stroke:black; stroke-width:1; }
    text:hover { stroke:black; stroke-width:1; stroke-opacity:0.35; }
    </style>
    <script type=\"text/ecmascript\">
    <![CDATA[",
                  if (tooltip) "
    var bMouseDragging = false;
    var nMouseOffsetX = 0;
    var nMouseOffsetY = 0;
    var detailss;
    bMouseDragging = true;

    var details = document.getElementById(\"details\");
```

```
var detailsBG = document.getElementById(\"detailsBG\");
if(details) {
    var p = document.documentElement.createSVGPoint();
    p.x = evt.clientX;
    p.y = evt.clientY;

    var m = details.getScreenCTM();
    p = p.matrixTransform(m.inverse());
    nMouseOffsetX = p.x - parseInt(details.getAttribute(\"x\"));
    nMouseOffsetY = p.y - parseInt(details.getAttribute(\"y\"));
}

function mouseMove(evt) {
    var p = document.documentElement.createSVGPoint();
    p.x = evt.clientX;
    p.y = evt.clientY;
    var bClient = true;

    if(bMouseDragging) {
        var details = document.getElementById(\"details\");
        var detailsBG = document.getElementById(\"detailsBG\");
        if(details) {

            var m = details.getScreenCTM();
            p = p.matrixTransform(m.inverse());
            var totalOffset = evt.pageY;
            var totalXOffset = evt.pageX;

            details.setAttribute(\"y\", totalOffset - nMouseOffsetY + 18);
            detailsBG.setAttribute(\"y\", totalOffset - nMouseOffsetY);

            var bbox = details.getBBox();
            var width = bbox.width;
            var height = bbox.height;
            if(totalXOffset+width >= 1150){
                details.setAttribute(\"x\", totalXOffset - nMouseOffsetX - 15 - width);
                detailsBG.setAttribute(\"x\", totalXOffset - nMouseOffsetX - 20 - width);
            }
            else{
                details.setAttribute(\"x\", totalXOffset - nMouseOffsetX + 20);
                detailsBG.setAttribute(\"x\", totalXOffset - nMouseOffsetX + 15);
            }
```

```
                detailsBG.setAttribute(\"width\", width+10);
                detailsBG.setAttribute(\"height\", height+10);
                bClient = false;
            }
        }

    }

    function init() {
        detailss = document.getElementById(\"details\").firstChild;
        var details = document.getElementById(\"details\");
        var detailsBG = document.getElementById(\"detailsBG\");
        if(details) {
            details.addEventListener(\"mousemove\", mouseMove, false);
        }
    }

    function s(info) { detailss.nodeValue = info; }
        function c() { detailss.nodeValue = ' '; }
    ]]>" else "
    var details;
    function init(evt) { details = document.getElementById(\"details\").firstChild; }
    function s(info) { details.nodeValue = info; }
    function c() { details.nodeValue = ' '; }
    ]]>", "
    </script>
    <rect x=\"0.0\" y=\"0\" width=\"1200.0\" height=\"", 16*mx+66, "\" fill=\"url(#background)
    <text text-anchor=\"middle\" x=\"600\" y=\"24\" font-size=\"17\" font-family=\"Verdana\" f
    <text text-anchor=\"left\" x=\"10\" y=\"", 16*mx+50, "\" font-size=\"12\" font-family=\"Ve
    <text text-anchor=\"\" x=\"70\" y=\"", 16*mx+50, "\" font-size=\"12\" font-family=\"Verdan
    <rect x=\"10\" y=\"129\" width=\"1\" height=\"1.0\" fill=\"rgb(250,250,250)\" rx=\"2\" ry=
    <text x=\"70\" y=\"162\" font-size=\"12\" font-family=\"Verdana\" fill=\"rgb(0,0,0)\" id=\
    </svg>"), file = file)
}

## Merge non-NA refs into stacks. Add UnknownFunToken for leaf refs.
refStacks <- function(d) {
    rs <-function(s, r, d) {
        if (is.na(r[length(r)]))
            r <- r[-length(r)]
        else
            s <- c(s, UnknownFunToken)
        fl <- basename(d$files[refFN(r)])
```

```
        ln <- refLN(r)
        paste0(s, ifelse(is.na(r), "", sprintf(" (%s:%d)", fl, ln)))
    }
    lapply(1:length(d$stacks), function(i) rs(d$stacks[[i]], d$refs[[i]], d))
}

## produce a flame graph from an Rprof file.
## order = "time" produces a graph like profr.
flameGraph <- function(pd, svgfile, order = c("hot", "alpha", "time"),
                       colormap = NULL, srclines = FALSE, cex = 0.75,
                       main = "Call Graph", tooltip = FALSE) {
    order <- match.arg(order)
    if (is.character(pd))
        pd <- readPD(pd)
    data <- list(counts = pd$counts,
                 stacks = if (srclines) refStacks(pd) else pd$stacks,
                 trace = pd$trace)
    if (! missing(svgfile))
        svgFlameGraph(svgfile, data, order, colormap, main, tooltip)
    else
        stdFlameGraph(data, order, colormap, cex, main)
}

fgIdentify <- function(p) {
    loc <- locator(1)
    if (! is.null(loc)) {
        idx <- which(loc$x >= p$left & loc$x <= p$right &
                     loc$y >= p$bottom & loc$y <= p$top)
        if (length(idx) > 0)
            p$label[idx]
    }
}

fgIdentify <- function(p, n = 1, print = FALSE) {
    val <- NULL
    while (n > 0) {
        n <- n - 1
        loc <- locator(1)
        if (! is.null(loc)) {
            idx <- which(loc$x >= p$left & loc$x <= p$right &
                         loc$y >= p$bottom & loc$y <= p$top)
            if (length(idx) > 0) {
                if (print)
```

```
                    cat(p$label[idx], "\n")
                val <- c(val, p$label[idx])
            }
            else break
        }
        else break
    }
    val
}

## Outline clicked things (could redraw before outlining -- would need colors):
fgIdentify <- function(x, n = 1, print = FALSE, outline = FALSE, ...) {
    p <- x
    val <- NULL
    while (n > 0) {
        n <- n - 1
        loc <- locator(1)
        if (! is.null(loc)) {
            idx <- which(loc$x >= p$left & loc$x <= p$right &
                         loc$y >= p$bottom & loc$y <= p$top)
            if (length(idx) > 0) {
                if (outline) {
                    allIDX <- p$label %in% p$label[idx]
                    pp <- lapply(p, `[`, allIDX)
                    rect(pp$left, pp$bottom, pp$right, pp$top, lwd = 3)
                }
                if (print)
                    cat(p$label[idx], "\n")
                val <- c(val, p$label[idx])
            }
            else break
        }
        else break
    }
    val
}

identify.proftools_flameGraph <- fgIdentify


###
### Writing callgrind file
###
```

```
## For a set of source references determine if they have a common file
## index and return that index. If they do not have a common index
## then return NA.
commonFile <- function(refs) {
    fn <- unique(refFN(refs))
    if (length(fn) == 1 && ! is.na(2))
        fn
    else
        NA
}

## For each caller check whether all calls have a common file index.
## If they do, then assume this is the file in which the caller is
## defined.  Otherwise treat the caller's home file as unkown. The
## result returned by this function is a named vector with one element
## per function for which the home file is assumed know.  The names
## are the names of the callers, and the values are the indices of the
## files in whicn the callers are defined.  For leaf calls NA sites
## are ignored. Possibly a disagreement of the leaf call site with
## other calls should be ignored as well.
homeFileMap <- function(pd, cc) {
    lsites <- leafCallRefs(pd)
    site <- c(cc$callee.site, lsites$site)
    caller <- c(cc$caller, lsites$fun)
    map <- tapply(site, caller, commonFile)
    map[! is.na(map)]
}

leafCallRefs <- function(pd) {
    ln <- sapply(pd$stacks, length)
    stacks <- pd$stacks[ln > 0]
    refs <- pd$refs[ln > 0]
    lfuns <- sapply(stacks, function(x) x[length(x)])
    lrefs <- sapply(refs, function(x) x[length(x)])
    goodrefs <- ! is.na(lrefs)
    list(fun = lfuns[goodrefs], site = lrefs[goodrefs])
}

## Collect the data for the callgrind output. The basic data is
##
##     fc = function counts and leaf call references
##     cc = call counts and call site references
```

```
##
## To fc we add the indes of the home file for each cunction (NA if
## not known) in fl.
##
## To cc we add in cfl the index of the home file of the function
## called (the callee), and in cln the line number of the call (in the
## caller's file). If the caller's file is considered unknown, then
## the line number is NA.
##
## If we do not want GC information in the output then we set the
## gcself entries in fc to zero, since the output functions only
## generate GC output for positive gcself counts.
getCGdata <- function(pd, GC) {
    fc <- getCGselfData(pd)
    cc <- callCounts(pd, TRUE, FALSE)

    hfm <- homeFileMap(pd, cc)

    fc$fl <- hfm[match(fc$fun, names(hfm))]
    cc$cfl <- hfm[match(cc$callee, names(hfm))]
    cc$cln <- ifelse(is.na(match(cc$caller, names(hfm))),
                     NA, refLN(cc$callee.site))

    if (! GC)
        fc$gcself <- 0

    list(fc = fc, cc = cc, gcself = sum(fc$gcself),
         funs = sort(unique(fc$fun)),
         files = pd$files)
}

getCGselfData <- function(pd) {
    fc <- funCounts(pd, FALSE)
    fc$total <- fc$gctotal <- NULL
    f <- sapply(pd$stacks, function(x) x[length(x)])
    r <- sapply(pd$refs, function(x) x[length(x)])
    fs <- aggregateCounts(data.frame(fun = f, site = r),
                          data.frame(self = pd$counts, gcself = pd$gccounts))
    rbind(fs, fc[fc$self == 0, ])
}

writeSelfEntry <- function(con, fun, fc, files) {
    fn <- fc$fl[fc$fun == fun][1]
```

```
    file <- if (is.na(fn)) "??" else files[fn]
    self <- fc$self[fc$fun == fun]
    gcself <- fc$gcself[fc$fun == fun]
    site <- fc$site[fc$fun == fun]
    line <- ifelse(is.na(site), 0, refLN(site))

    cat(sprintf("\nfl=%s\nfn=%s\n", file, fun), file = con)
    cat(sprintf("%d %d\n", line, self - gcself), sep = "", file = con)

    gcself <- sum(gcself)
    if (gcself > 0)
        cat(sprintf("cfl=??\ncfn=<GC>\ncalls=%d 0\n0 %d\n", gcself, gcself),
            sep = "", file = con)
}

writeCallEntries <- function(con, fun, cc, files) {
    fcc <- cc[cc$caller == fun, ]
    cfun <- fcc$callee
    tot <- fcc$total
    file <- ifelse(is.na(fcc$cfl), "??", files[fcc$cfl])
    line <- ifelse(is.na(fcc$cln), 0, fcc$cln)

    cat(sprintf("cfl=%s\ncfn=%s\ncalls=%d 0\n%d %d\n",
                file, cfun, tot, line, tot),
        sep = "", file = con)
}

writeFunEntries <- function(con, fun, data) {
    fc <- data$fc
    cc <- data$cc
    files <- data$files
    writeSelfEntry(con, fun, fc, files)
    writeCallEntries(con, fun, cc, files)
}

writeGCEntry <- function(con, data)  {
    gcself <- data$gcself
    if (gcself > 0)
        cat(sprintf("\nfl=??\nfn=<GC>\n0 %d\n", gcself), file = con)
}

writeCG <- function(con, pd, GC = TRUE) {
    if (is.character(con)) {
```

```
        con <- file(con, "w")
        on.exit(close(con))
    }

    data <- getCGdata(pd, GC)

    cat("events: Hits\n", file = con)

    for (fun in data$funs)
        writeFunEntries(con, fun, data)

    writeGCEntry(con, data)
}

writeCallgrindFile <- function(pd, file = "Rprof.cg", GC = TRUE)
    writeCG(file, pd, GC)


###
### Source reference summaries
###

lineSites <- function(line, refs, useSite)
    unique(cbind(site = refs))

leafSite <- function(line, refs, useSite) {
    n <- length(refs)
    cbind(site = refs[n])
}

siteCounts <- function(pd)
    entryCounts(pd, lineSites, leafSite, TRUE)


###
### Collecting profile data
###

profileExpr <- function(expr, GC = TRUE, srclines = TRUE) {
    tmp <- tempfile()
    on.exit(unlink(tmp))
    Rprof(tmp, gc.profiling = GC, line.profiling = TRUE)
    expr
```

```
    Rprof(NULL)
    pd <- readProfileData(tmp)
    mydepth <- length(sys.calls())
    skipPD(pd, mydepth)
}

###
### Experimental stuff
###

countHits <- function(stacks, counts) {
    stacks <- lapply(stacks, function(x) x[! is.na(x)])
    uitems <- unique(unlist(stacks))
    ln <- sapply(uitems,
                 function(y) sapply(stacks,
                                    function(x) as.integer(y %in% x)))
    t(ln) %*% do.call(cbind, counts)
}

countSelfHits <- function(stacks, counts) {
    uitems <- unique(unlist(lapply(stacks, function(x) unique(x[! is.na(x)]))))
    ln <- sapply(uitems,
                 function(y) sapply(stacks,
                                    function(x) {
                                        n <- length(x)
                                        if (n > 0 && identical(y, x[n]))
                                            1
                                        else
                                            0
                                    }))
    t(ln) %*% do.call(cbind, counts)
}

recodeRefs <- function(refs, files, na.value = NA) {
    fn <- refFN(refs)
    ln <-refLN(refs)
    ifelse(is.na(refs), na.value, paste(basename(files)[fn], ln, sep = ":"))
}

recodeRefsList <- function(refs, files)
    sapply(refs, recodeRefs, files)

formatTrace <- function(trace, maxlen = 50, skip = 0, trimtop = FALSE) {
```

```
    if (skip > 0)
        trace <- trace[-(1 : skip)]
    out <- paste(trace, collapse = " -> ")
    if (trimtop)
        while (nchar(out) > maxlen && length(trace) > 1) {
            trace <- trace[-length(trace)]
            out <- paste(paste(trace, collapse = " -> "), "... ")
        }
    else
        while (nchar(out) > maxlen && length(trace) > 1) {
            trace <- trace[-1]
            out <- paste("... ", paste(trace, collapse = " -> "))
        }
    out
}

printPaths <- function(pd, n, ...) {
    ord = rev(order(pd$counts))
    if (! missing(n) && length(ord) > n)
        ord <- ord[1 : n]
    tot <- pd$total
    pct <- percent(pd$counts[ord], tot)
    gcpct <- percent(pd$gccounts[ord], tot)
    paths <- sapply(pd$stacks[ord], formatTrace, ...)
    mapply(function(x, y, z) cat(sprintf("%5.1f %5.1f   %s\n", x, y, z)),
           pct, gcpct, paths, SIMPLIFY = FALSE)
    invisible(NULL)
}

pathSummaryPct <- function(apd, gc, tot) {
    counts <- apd$counts
    gccounts <- apd$gccounts
    paths <- apd$paths

    pct <- percent(counts, tot)
    if (gc) {
        gcpct <- percent(gccounts, tot)
        data.frame(total.pct = pct, gc.pct = gcpct, row.names = paths)
    }
    else
        data.frame(total.pct = pct, row.names = paths)
}
```

```
pathSummaryTime <- function(apd, gc, delta) {
    counts <- apd$counts
    gccounts <- apd$gccounts
    paths <- apd$paths
    tm <- counts * delta
    if (gc) {
        gctm <- gccounts * delta
        data.frame(total.time = tm, gc.time = gctm, row.names = paths)
    }
    else
        data.frame(total.time = tm, row.names = paths)
}

pathSummaryHits <- function(apd, gc) {
    hits <- apd$counts
    gchits <- apd$gccounts
    paths <- apd$paths
    if (gc)
        data.frame(total.hits = hits, gc.hits = gchits, row.names = paths)
    else
        data.frame(total.hits = hits, row.names = paths)
}

pathSummary <- function(pd, value = c("pct", "time", "hits"),
                        srclines = FALSE, GC = TRUE, total.pct = 0, ...) {
    value <- match.arg(value)

    if (srclines && pd$haveRefs) {
        files <- pd$files ## shorter: as.character(seq_along(pd$files)
        rstacks <- mapply(function(a, b) funLabels(a, b, files),
                          pd$stacks,
                          sapply(pd$refs, function(x) x[-length(x)]),
                          SIMPLIFY = FALSE)
        paths <- sapply(rstacks, formatTrace, ...)
    }
    else
        paths <- sapply(pd$stacks, formatTrace, ...)

    ## need to aggregate in case some collapsed paths are identical
    ## or some paths differ only in source references.
    apd <- aggregateCounts(list(paths = paths),
                           cbind(counts = pd$counts, gccounts = pd$gccounts))
    apd <- apd[rev(order(apd$counts)),]
```

```
    if (total.pct > 0)
        apd <- apd[apd$counts >= pd$total * (total.pct / 100),]

    if (value == "pct")
        pathSummaryPct(apd, GC && pd$haveGC, pd$total)
    else if (value == "time")
        pathSummaryTime(apd, GC && pd$haveGC, pd$interval / 1.0e6)
    else
        pathSummaryHits(apd, GC && pd$haveGC)
}

trimOrPad <- function(str, width) {
    ifelse(nchar(str) > width,
           paste(substr(str, 1, width - 4), "..."),
           sprintf("%-*s", width, str))
}

srcSummary <- function(pd, byTotal = TRUE,
                       value = c("pct", "time", "hits"),
                       GC = TRUE, total.pct = 0,
                       source = TRUE, width = getOption("width")) {
    if (! pd$haveRefs)
        stop("profile data does not contain source information")

    value <- match.arg(value)

    rc <- refCounts(pd)

    file <- basename(pd$files[refFN(rc$refs)])
    line <- refLN(rc$refs)
    label <- paste(file, line, sep = ":")

    if (GC && pd$haveGC)
        val <- cbind(total = rc$total, gctotal = rc$gctotal)
    else
        val <- cbind(total = rc$total)
    rownames(val) <- label

    ## order rows of val and rc by row number within file and alphabetically
    ## by file.
    ord <- order(line)
    val <- val[ord, , drop = FALSE]
```

```
    rc <- rc[ord, , drop = FALSE]
    file <- file[ord]
    val <- val[order(file), , drop = FALSE]
    rc <- rc[order(file), , drop = FALSE]

    if (total.pct > 0)
        val <- val[val[,1] >= pd$total * (total.pct / 100), , drop = FALSE]

    if (value == "pct") {
        tot <- pd$total
        colnames(val) <- paste(colnames(val), "pct", sep = ".")
        val <- as.data.frame(percent(val, tot))
    }
    else if (value == "time") {
        delta <- pd$interval / 1.0e6
        colnames(val) <- paste(colnames(val), "time", sep = ".")
        val <- as.data.frame(val * delta)
    }
    else {
        colnames(val) <- paste(colnames(val), "hits", sep = ".")
        val <- as.data.frame(val)
    }
    if (source) {
        pad <- 5 ## allow for padding print.data.frame might insert
        width <- width - max(nchar(capture.output(print(val)))) - pad
        src <- unlist(
            lapply(seq_along(pd$files),
                   function(fn) {
                       fname <- pd$files[fn]
                       which <- refFN(rc$refs) == fn
                       if (file.exists(fname))
                           readLines(fname)[refLN(rc$refs[which])]
                       else
                           rep("<source not available>", sum(which))
                   }))
        cbind(val, data.frame(source = trimOrPad(src, width),
                              stringsAsFactors = FALSE))
    }
    else val
}

annotateSource <- function(pd, value = c("pct", "time", "hits"),
                           GC = TRUE, sep = ":  ", show = TRUE, ...) {
```

```
    if (! is.null(pd$files)) {
        r <- refCounts(pd)
        file <- refFN(r$ref)
        line <- refLN(r$ref)
        value <- match.arg(value)
        delta <- pd$interval / 1.0e6
        if (GC && pd$haveGC)
            if(value == "pct")
                s <- sprintf(" %5.2f%% %5.2f%%  ",
                             round(100 * (r$total / pd$total), 2),
                             round(100 * (r$gctotal / pd$total), 2))
            else if(value == "time")
                s <- sprintf(" %5.2f %5.2f  ", round(r$total*delta, 2),
                             round(r$gctotal*delta, 2))
            else
                s <- sprintf(" %5d %5d  ", r$total, r$gctotal)
        else
            if(value == "pct")
                s <- sprintf(" %5.2f%%  ", round(100 * (r$total / pd$total), 2))
            else if(value == "time")
                s <- sprintf(" %5.2f  ", round(r$total*delta, 2))
            else
                s <- sprintf(" %5d  ", r$total)
        ann <- lapply(seq_along(pd$files),
                      function(fn) {
                          flines <- readLines(pd$files[fn])
                          b <- rep(paste0(rep(" ", nchar(s[1])), collapse = ""),
                                   length(flines))
                          b[line[file == fn]] <- s[file == fn]
                          paste(b, flines, sep = sep)
                      })
        names(ann) <- pd$files
        if (show) {
            showAnnotation(ann, ...)
            invisible(ann)
        }
        else ann
    }
}

## **** option to only show lines with annotation, plus or minus a few?
## **** show line numbers?
showAnnotation <- function(ann, width = getOption("width"), ...) {
```

```
    tmp <- tempfile()
    on.exit(unlink(tmp))
    for (i in seq_along(ann)) {
        a <- ann[[i]]
        if (is.na(width))
            ta <- a
        else
            ta <- ifelse(nchar(a) > width,
                         paste(substr(a, 1, width - 4), "..."),
                         a)
        writeLines(ta, tmp)
        file.show(tmp, ...)
    }
}
```