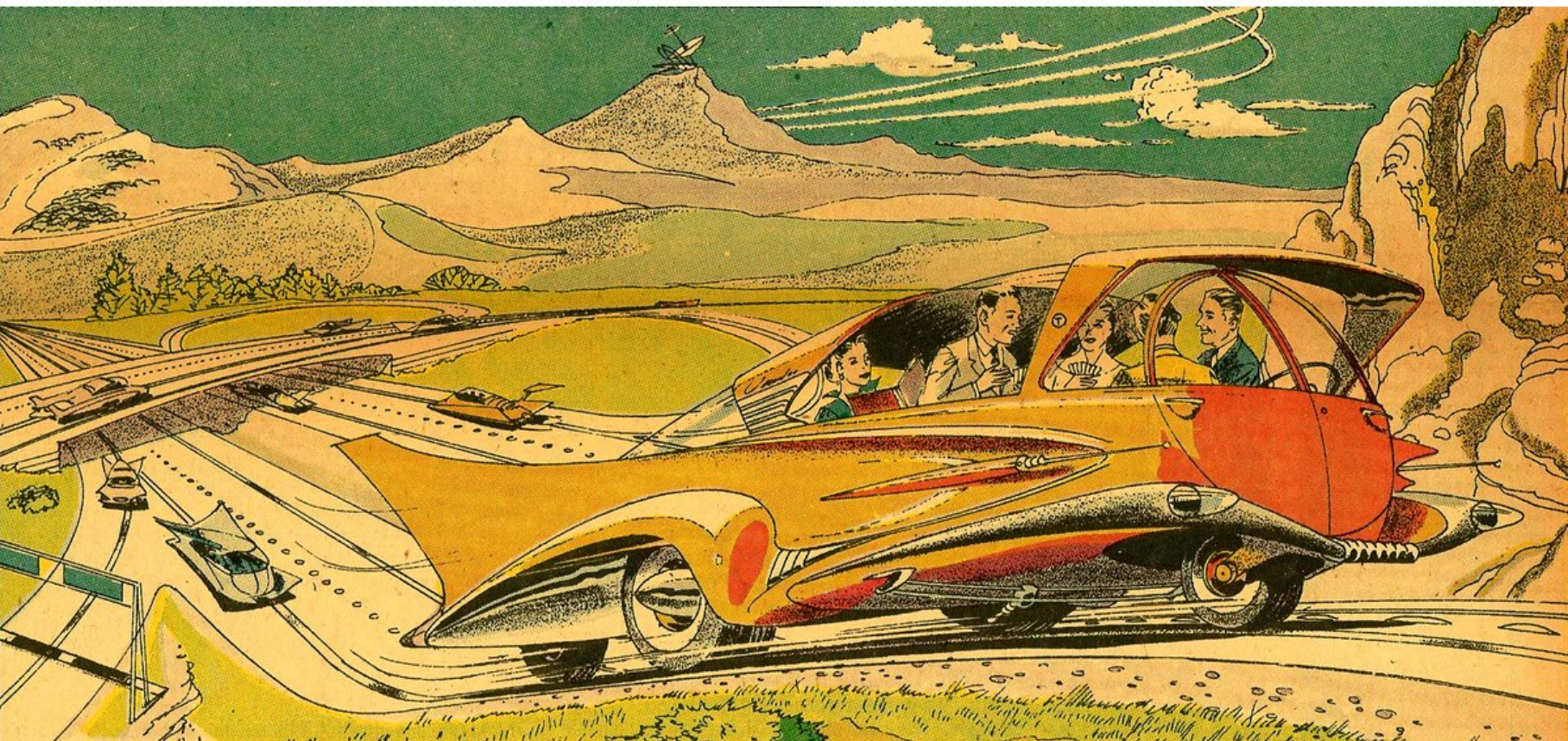


A Future for R

Henrik Bengtsson, UC San Francisco



My first R assignment

Calculating the sum $1 + 2 + \dots + 100$:

```
> y <- sum(1:100)
> y
[1] 5050
```

My first R assignment

Calculating the sum $1 + 2 + \dots + 100$:

```
> y <- slow_sum(1:100)    ## 2 min  
> y  
[1] 5050
```

My first R assignment

Calculating the sum $1 + 2 + \dots + 100$:

```
> y <- slow_sum(1:100)    ## 2 min  
> y  
[1] 5050
```

Divide-and-conquer alternative: Calculate $a = 1 + 2 + \dots + 50$, and then $b = 51 + 52 + \dots + 100$, and then add up a and b :

```
> a <- slow_sum(1:50)    ## 1 min
```

My first R assignment

Calculating the sum $1 + 2 + \dots + 100$:

```
> y <- slow_sum(1:100)    ## 2 min  
> y  
[1] 5050
```

Divide-and-conquer alternative: Calculate $a = 1 + 2 + \dots + 50$, and then $b = 51 + 52 + \dots + 100$, and then add up a and b :

```
> a <- slow_sum(1:50)     ## 1 min
```

```
> b <- slow_sum(51:100)  ## 1 min
```

My first R assignment

Calculating the sum $1 + 2 + \dots + 100$:

```
> y <- slow_sum(1:100)    ## 2 min  
> y  
[1] 5050
```

Divide-and-conquer alternative: Calculate $a = 1 + 2 + \dots + 50$, and then $b = 51 + 52 + \dots + 100$, and then add up a and b :

```
> a <- slow_sum(1:50)    ## 1 min
```

```
> b <- slow_sum(51:100)  ## 1 min
```

```
> y <- a + b  
> y  
[1] 5050
```

My first future assignment

Parallel divide-and-conquer: Calculate $a = 1 + 2 + \dots + 50$ and $b = 51 + 52 + \dots + 100$ at the same time, and then add up a and b :

My first future assignment

Parallel divide-and-conquer: Calculate $a = 1 + 2 + \dots + 50$ and $b = 51 + 52 + \dots + 100$ at the same time, and then add up a and b :

```
> library("future")  
> plan(multiprocess)      ## Parallel processing
```

Regular assignment: $y \leftarrow x$

Future assignment: $y \%<-\% x$

My first future assignment

Parallel divide-and-conquer: Calculate $a = 1 + 2 + \dots + 50$ and $b = 51 + 52 + \dots + 100$ at the same time, and then add up a and b :

```
> library("future")  
> plan(multiprocess)      ## Parallel processing
```

```
> a %<-% slow_sum(1:50)   ## These two assignments are  
> b %<-% slow_sum(51:100) ## non-blocking and in parallel  
>
```

Regular assignment: $y \leftarrow x$

Future assignment: $y \%<-\% x$

My first future assignment

Parallel divide-and-conquer: Calculate $a = 1 + 2 + \dots + 50$ and $b = 51 + 52 + \dots + 100$ at the same time, and then add up a and b :

```
> library("future")  
> plan(multiprocess)      ## Parallel processing
```

```
> a %<-% slow_sum(1:50)   ## These two assignments are  
> b %<-% slow_sum(51:100) ## non-blocking and in parallel  
>
```

```
> y <- a + b             ## Waits for a and b
```

Regular assignment: $y <- x$

Future assignment: $y \%<-\% x$

My first future assignment

Parallel divide-and-conquer: Calculate $a = 1 + 2 + \dots + 50$ and $b = 51 + 52 + \dots + 100$ at the same time, and then add up a and b :

```
> library("future")  
> plan(multiprocess)      ## Parallel processing
```

```
> a %<-% slow_sum(1:50)    ## These two assignments are  
> b %<-% slow_sum(51:100) ## non-blocking and in parallel  
>
```

```
> y <- a + b              ## Waits for a and b
```

```
> y  
[1] 5050
```

Regular assignment: $y \leftarrow x$

Future assignment: $y \%<-\% x$

Definition: Future

$$y \text{ \%<- \% } \{ \text{expr} \}$$

- A **future** is an abstraction for a **value** that will be available later.

Definition: Future

$$y \text{ \%<- \% } \{ \text{expr} \}$$

- A **future** is an abstraction for a **value** that will be available later.
- The value is the **result of an evaluated expression**.

Friedman & Wise (1976, 1977), Hibbard (1976), Baker & Hewitt (1977)

Definition: Future

$$y \text{ \%<- \% } \{ \text{expr} \}$$

- A **future** is an abstraction for a **value** that will be available later.
- The value is the **result of an evaluated expression**.
- The **state of a future** is either **unresolved** or **resolved**.

Definition: Future

$$y \text{ \%<- \% } \{ \text{expr} \}$$

- A **future** is an abstraction for a **value** that will be available later.
- The value is the **result of an evaluated expression**.
- The **state of a future** is either **unresolved** or **resolved**.
- The **value is blocking** until the future is resolved.

R package: future



- A simple **unified API**
- Works the same on **all platforms**
- **Easy to install**
- **Lightweight** (~300 kB incl. dependencies)
- **Vignettes**
- **Extendable** by anyone

CRAN

1.0.0

build

passing

Codecov

97%

Many ways to resolve futures

Strategy:	
eager	sequentially
lazy	only if needed
multiprocess	in parallel
cluster	on a set of machines

Many ways to resolve futures

Strategy:

eager	sequentially
lazy	only if needed
multiprocess	in parallel
cluster	on a set of machines

```
> plan(lazy)

> a %<-% slow_sum(1:50)
> b %<-% slow_sum(51:100)
>
```

Many ways to resolve futures

Strategy:

eager	sequentially
lazy	only if needed
multiprocess	in parallel
cluster	on a set of machines

```
> plan(lazy)

> a %<-% slow_sum(1:50)
> b %<-% slow_sum(51:100)
>
```

```
> b
[1] 3775
```

Future a will never be evaluated!

Consistent futures everywhere



- Unix
- macOS
- Windows

```
> library("future")  
> plan(multiprocess)  
> demo("mandelbrot")
```

Calculating and plotting
Mandelbrot regions ...

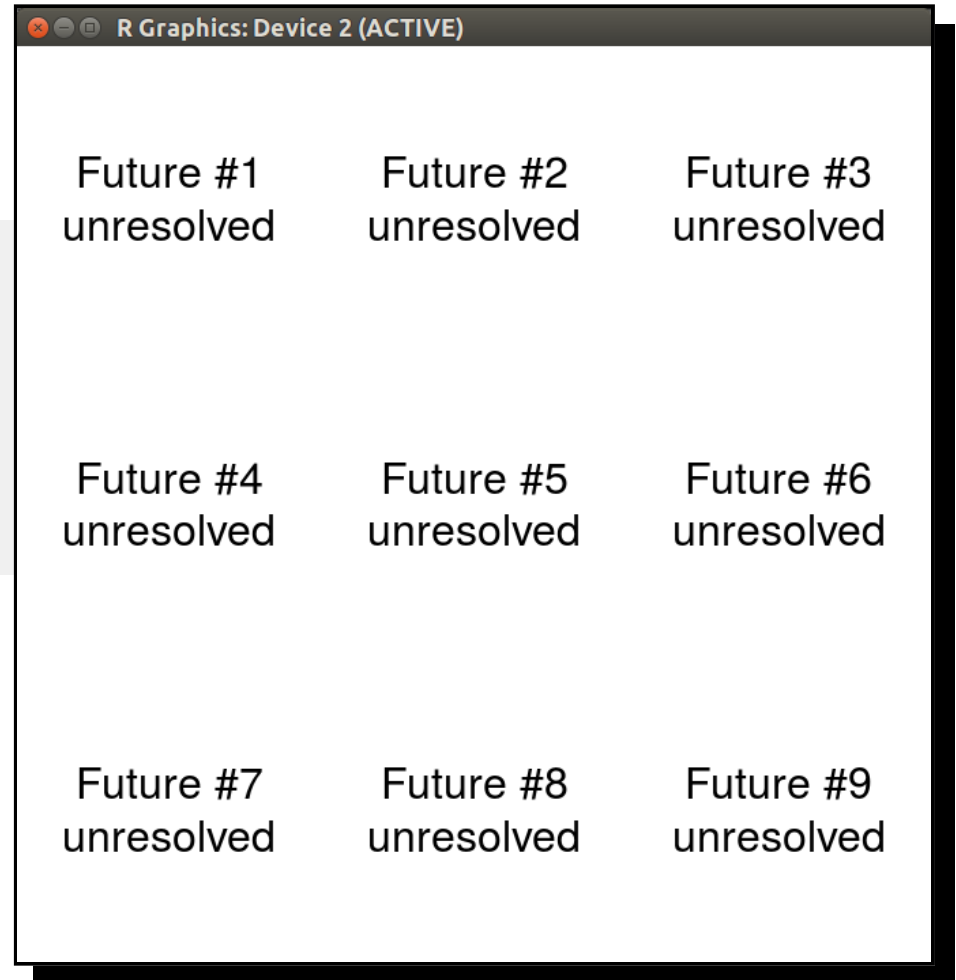
Consistent futures everywhere



- Unix
- macOS
- Windows

```
> library("future")  
> plan(multiprocess)  
> demo("mandelbrot")
```

Calculating and plotting
Mandelbrot regions ...



Consistent futures everywhere



- Unix
- macOS
- Windows

```
> library("future")  
> plan(multiprocess)  
> demo("mandelbrot")
```

Calculating and plotting
Mandelbrot regions ...

Future #2 unresolved	Future #3 unresolved	
Future #4 unresolved	Future #5 unresolved	Future #6 unresolved
Future #7 unresolved	Future #8 unresolved	Future #9 unresolved

Consistent futures everywhere



- Unix
- macOS
- Windows

```
> library("future")  
> plan(multiprocess)  
> demo("mandelbrot")
```

Calculating and plotting
Mandelbrot regions ...

R Graphics: Device 2 (ACTIVE)

Future #3
unresolved

Future #4
unresolved

Future #5
unresolved

Future #6
unresolved

Future #7
unresolved

Future #8
unresolved

Future #9
unresolved

Consistent futures everywhere

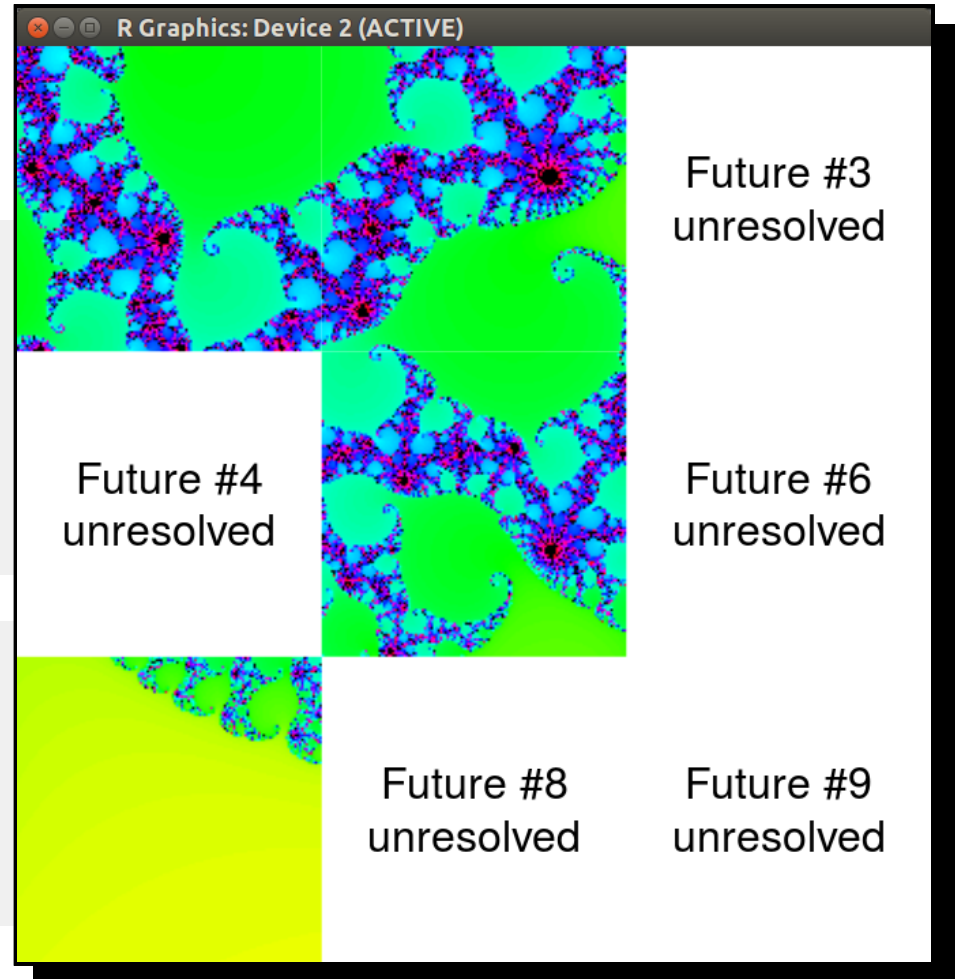


- Unix
- macOS
- Windows

```
> library("future")  
> plan(multiprocess)  
> demo("mandelbrot")
```

Calculating and plotting
Mandelbrot regions ...

- Region 1 done
- Region 2 done
- Region 7 done
- Region 5 done
- ...



Consistent futures everywhere

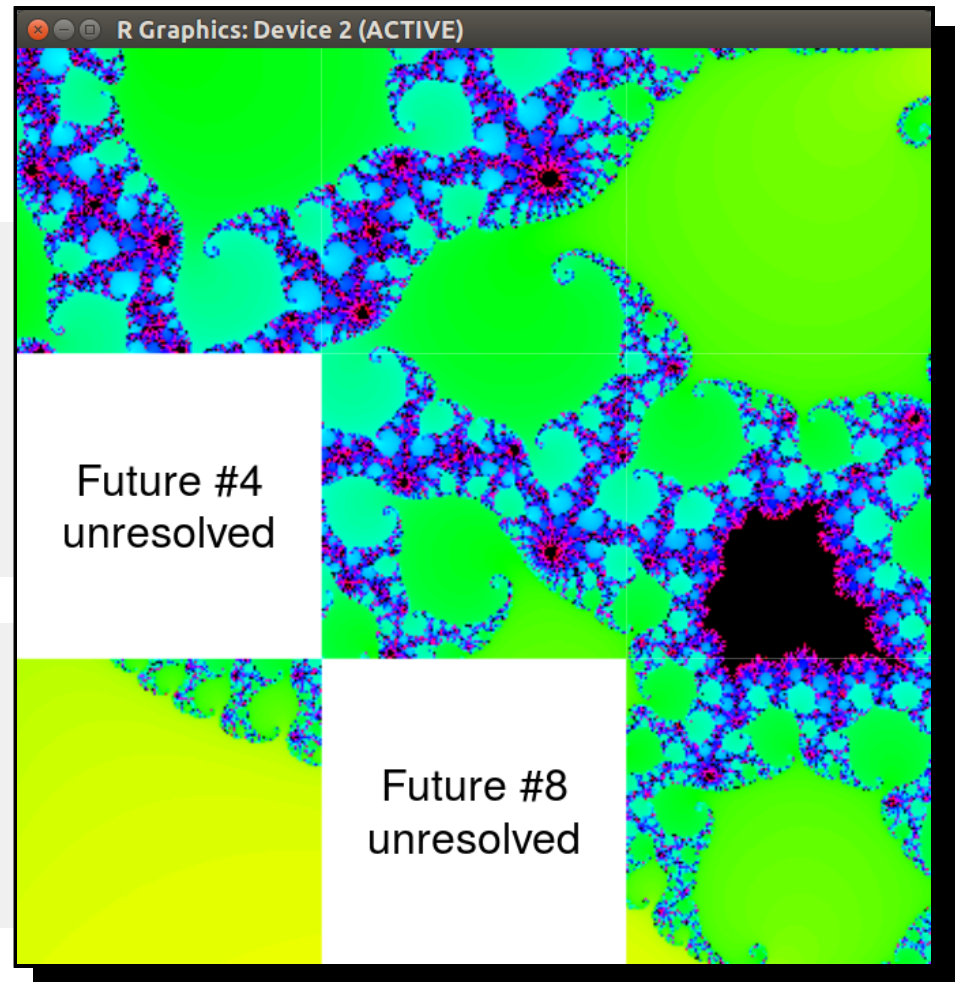


- Unix
- macOS
- Windows

```
> library("future")  
> plan(multiprocess)  
> demo("mandelbrot")
```

Calculating and plotting
Mandelbrot regions ...

- Region 1 done
- Region 2 done
- Region 7 done
- Region 5 done
- ...



Consistent futures everywhere

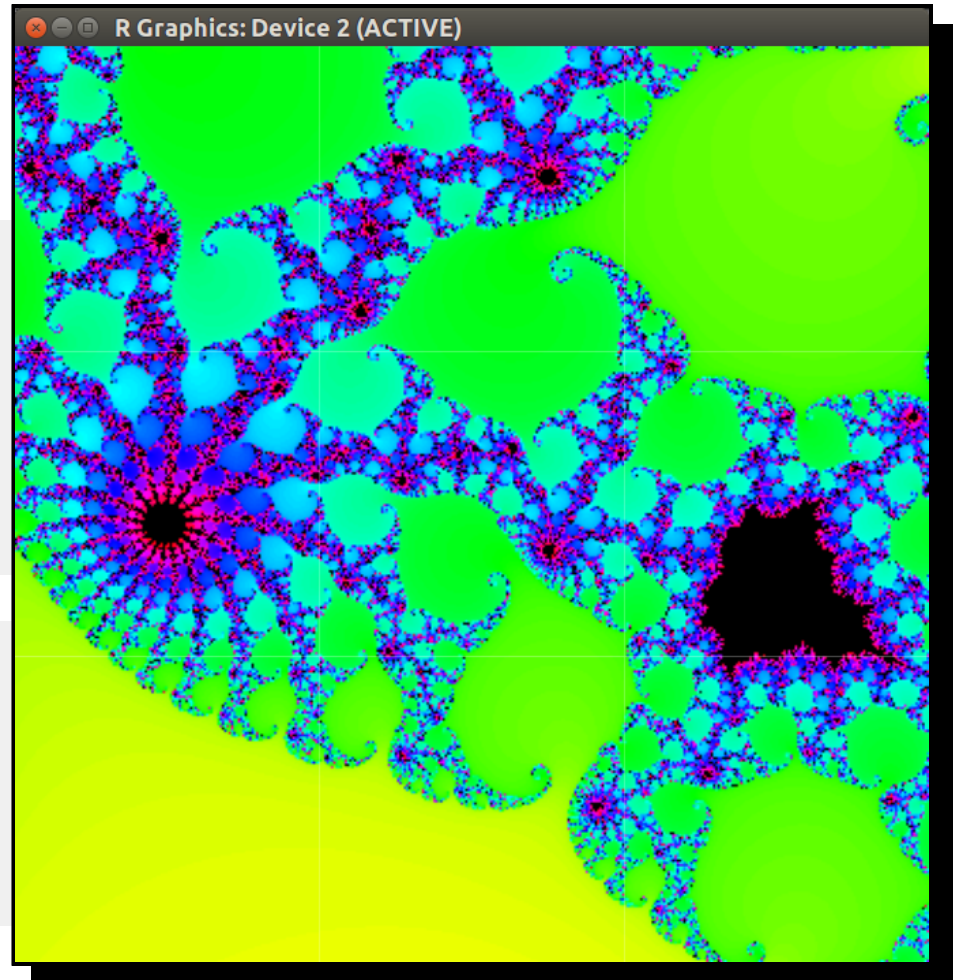


- Unix
- macOS
- Windows

```
> library("future")  
> plan(multiprocess)  
> demo("mandelbrot")
```

Calculating and plotting
Mandelbrot regions ...

- Region 1 done
- Region 2 done
- Region 7 done
- Region 5 done
- ...



Future takes care of globals



Global variables and functions that are needed for the future expression to be resolved are **identified automatically** and frozen / **exported**.

Packages are automatically loaded.

```
x <- rnorm(n=100)
y %<-% { slow_sum(x) }
```

Future takes care of globals



Global variables and functions that are needed for the future expression to be resolved are **identified automatically** and frozen / exported.

Packages are automatically loaded.

```
x <- rnorm(n=100)
y %<-% { slow_sum(x) }
```

Globals identified and frozen / exported:

1. `slow_sum()` - a function.

Future takes care of globals



Global variables and functions that are needed for the future expression to be resolved are **identified automatically** and frozen / exported.

Packages are automatically loaded.

```
x <- rnorm(n=100)
y %<-% { slow_sum(x) }
```

Globals identified and frozen / exported:

1. `slow_sum()` - a function.
2. `x` - a numeric vector of length 100.

Nested futures



```
x <- rnorm(n=100)

a %<-% {
  c %<-% slow_sum(x[1:25])
  d %<-% slow_sum(x[26:50])
  c + d
}

b %<-% {
  c %<-% slow_sum(x[51:75])
  d %<-% slow_sum(x[76:100])
  c + d
}

y <- a + b
```

Nested futures



```
x <- rnorm(n=100)

a %<-% {
  c %<-% slow_sum(x[1:25])
  d %<-% slow_sum(x[26:50])
  c + d
}

b %<-% {
  c %<-% slow_sum(x[51:75])
  d %<-% slow_sum(x[76:100])
  c + d
}

y <- a + b
```

Different strategies for resolving, e.g.

- `plan(list(cluster, multiprocess))`

High Performance Compute (HPC) clusters



Map-Reduce for HPC

```
## Find our 40 FASTQ files
fastq <- dir(pattern = "[.]fq$")          ## 200 GB each!

## Align them
bam <- lapply(fastq, FUN = DNaseq::align) ## 6 hours each!
```

Map-Reduce for HPC

```
## Find our 40 FASTQ files
fastq <- dir(pattern = "[.]fq$")          ## 200 GB each!

## Align them
bam <- lapply(fastq, FUN = DNaseq::align) ## 6 hours each!
```

```
library("BatchJobs")
reg <- makeRegistry(id="DNASEQseq")

fastq <- dir(pattern = "[.]fq$")
batchMap(reg, fastq, fun = DNaseq::align)
submitJobs(reg)
bam <- loadResults(reg)
```

future.BatchJobs: Futures for HPC

future.BatchJobs:

batchjobs_slurm

batchjobs_sge

batchjobs_torque

batchjobs_lsf

batchjobs_openlava

Job scheduler:

Slurm

Sun Grid Engine

TORQUE / PBS

Load Sharing Facility

OpenLava

```
library("future.BatchJobs")  
plan(batchjobs_slurm)
```

CRAN 0.12.1

build passing

Codecov 90%

future.BatchJobs: Futures for HPC

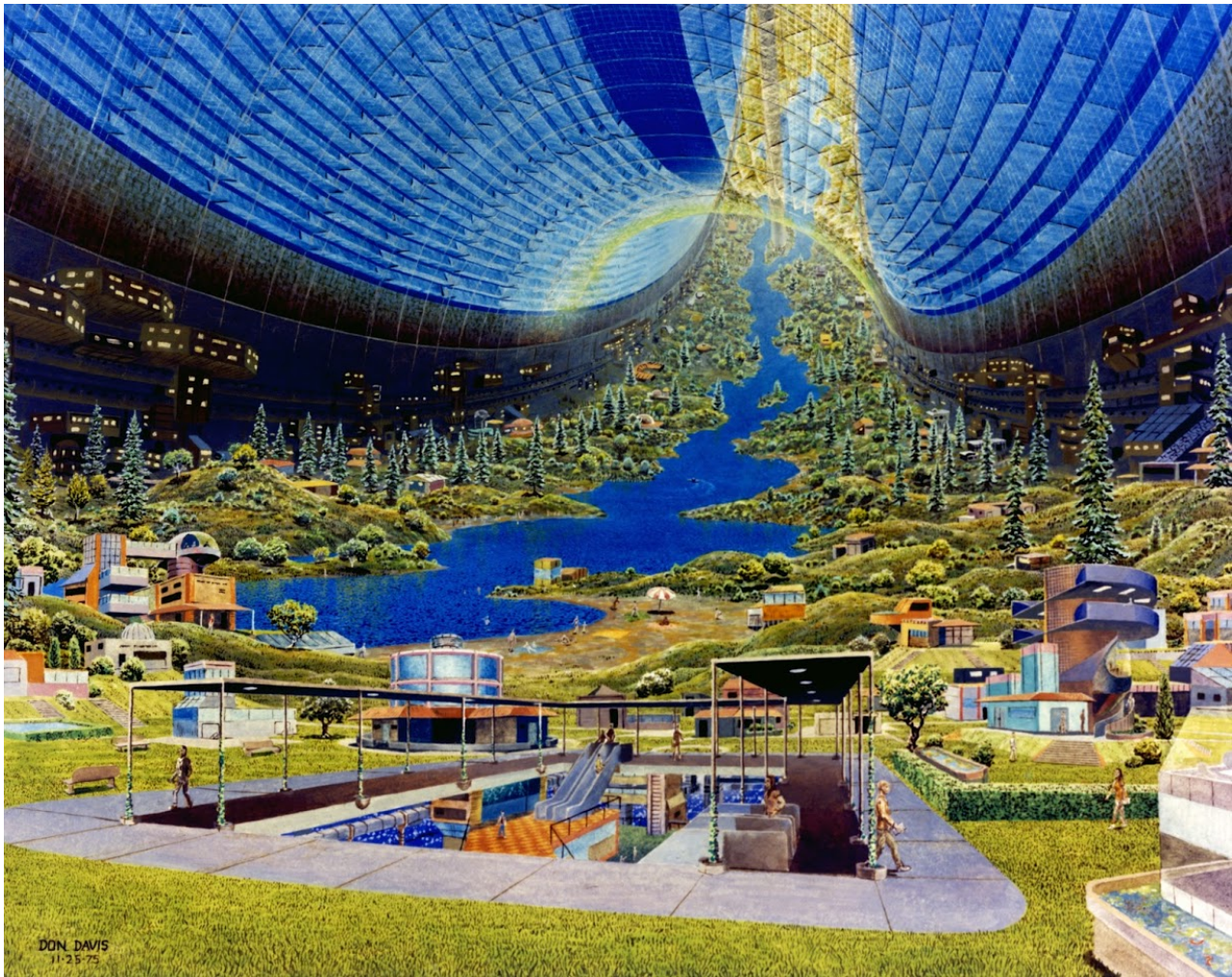
future.BatchJobs:	Job scheduler:
batchjobs_slurm	Slurm
batchjobs_sge	Sun Grid Engine
batchjobs_torque	TORQUE / PBS
batchjobs_lsf	Load Sharing Facility
batchjobs_openlava	OpenLava

```
library("future.BatchJobs")  
plan(batchjobs_slurm)
```

```
bam <- listenv()  
for (i in seq_along(fastq)) {  
  bam[[i]] %<-% DNAseq::align(fastq[i])  
}
```

CRAN 0.12.1 build passing Codecov 90%

Parallel alternatives



foreach: Futures with foreach()

```
## Align them
foreach(i = seq_along(fastq), .export = "fastq") %dopar% {
  DNaseq::align(fastq[i])
}
```

foreach: Futures with foreach()

```
## Align them
foreach(i = seq_along(fastq), .export = "fastq") %dopar% {
  DNAseq::align(fastq[i])
}
```

The **doFuture** package provides a foreach %dopar% adapter such that *any* type of futures can be used wherever the foreach package is used.

```
library("doFuture")
registerDoFuture()
plan(multiprocess)
```

CRAN 0.2.0 build passing Codecov 100% (48 lines!)

1100+ packages can now parallelize on HPC

Package that depends on foreach: ~300 directly + ~800 indirectly

These can now also take advantage of compute clusters:

```
library("doFuture")  
registerDoFuture()      ## (a) Tell foreach to use futures  
  
library("future.BatchJobs")  
plan(batchjobs_slurm)  ## (b) Resolve via Slurm scheduler
```

1100+ packages can now parallelize on HPC

Package that depends on foreach: ~300 directly + ~800 indirectly

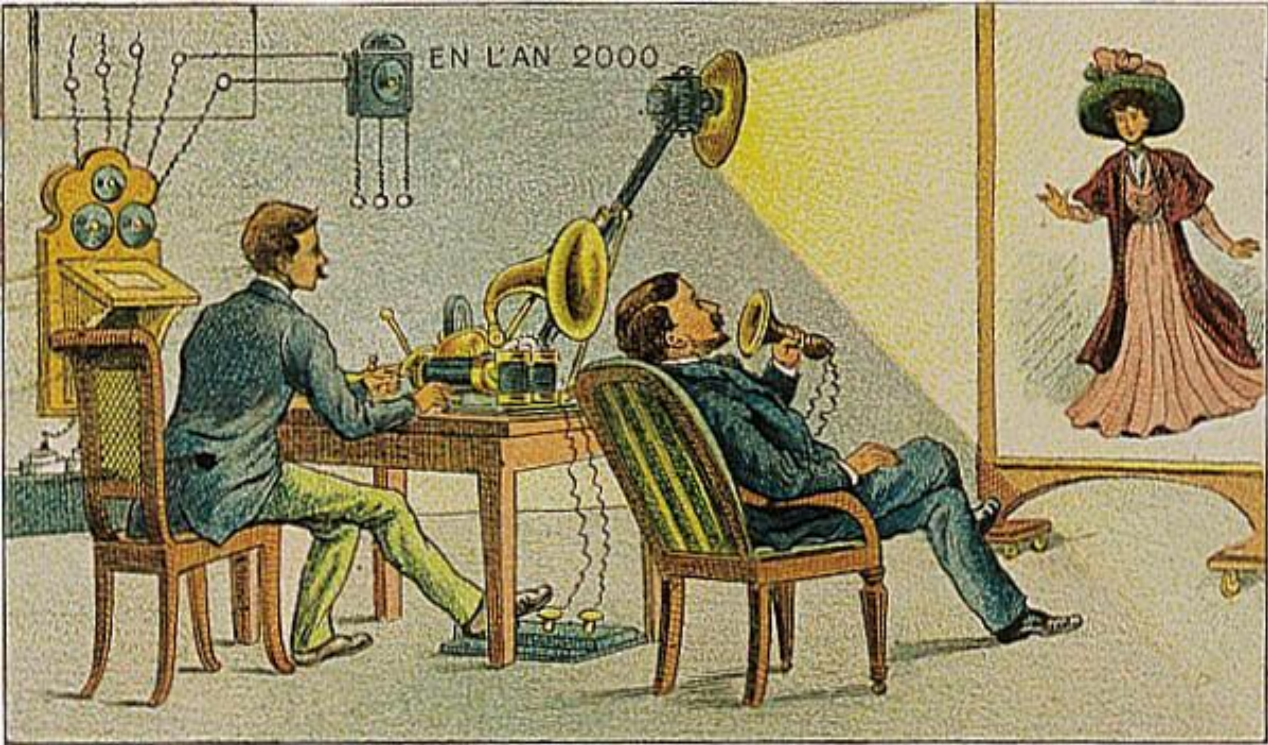
These can now also take advantage of compute clusters:

```
library("doFuture")
registerDoFuture()      ## (a) Tell foreach to use futures

library("future.BatchJobs")
plan(batchjobs_slurm)  ## (b) Resolve via Slurm scheduler
```

```
library("plyr")
fastq <- dir(pattern = "[.]fq$")
bam <- llply(fastq, DNaseq::align, .parallel = TRUE)
```

Future bonuses



Plot remotely - display locally

```
> library("future")  
> plan(remote, workers="remote.server.org")
```

Plot remotely - display locally

```
> library("future")  
> plan(remote, workers="remote.server.org")
```

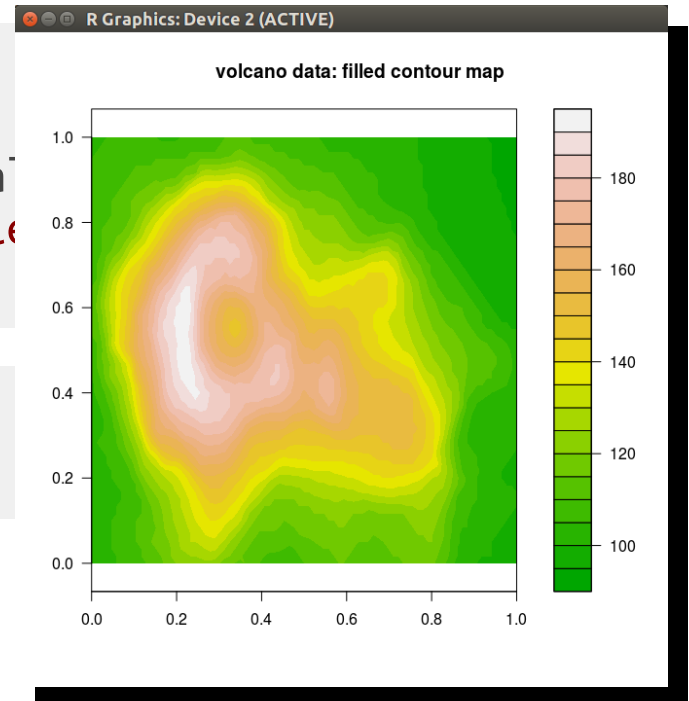
```
## Plot remotely  
> g %<-% R.devices::capturePlot({  
+   filled.contour(volcano, color.palette = terrain.colors)  
+   title(main = "volcano data: filled contour map")  
+ })
```

Plot remotely - display locally

```
> library("future")  
> plan(remote, workers="remote.server.org")
```

```
## Plot remotely  
> g %<-% R.devices::capturePlot({  
+   filled.contour(volcano, color.palette("terrain"))  
+   title(main = "volcano data: filled contour map")  
+ })
```

```
## Display locally  
> g
```



Building a better future

I  bug reports,
feedback and suggestions!

<https://github.com/HenrikBengtsson/future>
@HenrikBengtsson

Thank you!

Appendix

- A1. Summary
- A2. Future has a minimalistic API
- A3. Nested futures with plyr
- A4. BiocParallel: Futures with Bioconductor
- A5. Profile code remotely - display locally
- A6. Implement Future API for new backend
- A7. Futures I'd like to see
- A8. Future improvements

A1. Summary

	future	parallel	foreach	BatchJobs	BiocParallel
Synchronous	✓	✓	✓	✓	✓
Asynchronous	✓	✓	✓	✓	✓
Uniform API	✓		✓	✓	✓
Extendable API	✓		✓	✓	✓
Globals	✓		(✓)		
Packages	✓		(✓)		
For loops	✓		foreach()		
While loops	✓				
Nested config	✓				
Recursive protection	✓	mc		mc	
Early stopping	✓				✓
Traceback	✓				✓
RNG stream	manual	mc & SNOW	doRNG	manual	mc & SNOW

A2. Future has a minimalistic API



Future assignment:

Functional API:

```
> x <- 101:200

## Create implicit futures
> a %<-% slow_sum(x[1:50])
> b %<-% slow_sum(x[51:100])

## Get their values
> y <- a + b

> y
[1] 15050
```

```
> x <- 101:200

## Create explicit futures
> f <- future( slow_sum(x[1:50]) )
> g <- future( slow_sum(x[51:100]) )

## Get their values
> y <- value(f) + value(g)

> y
[1] 15050
```

A3. Nested futures with plyr

```
library("future.BatchJobs")
plan(list(batchjobs_slurm, multiprocess))

library(plyr)

bam <- llply(fastq, function(fq) {

  chrs <- llply(1:24, function(chr) {
    DNaseq::align(fq, chromosome = chr)
  }, .parallel = TRUE)

  merge_chromosomes(chrs)

}, .parallel = TRUE)
```

A4. BiocParallel: Futures with Bioconductor

You can use futures with BiocParallel, e.g.

```
library("BiocParallel")
register(DoparParam(), default = TRUE)

library("doFuture")
registerDoFuture()

plan(multiprocess)

bplapply(fastq, function(fq) {
  DNaseq::align(fq)
})
```

A5. Profile code remotely - display locally

```
> library("future")
> plan(remote, workers="remote.server.org")

> library("profvis")

> dat <- data.frame(
+   x = rnorm(50e3),
+   y = rnorm(50e3)
+ )

## Profile remotely
> p %<-% profvis({
+   plot(x ~ y, data = dat)
+   m <- lm(x ~ y, data = dat)
+   abline(m, col = "red")
+ })
```

A5. Profile code remotely - display locally

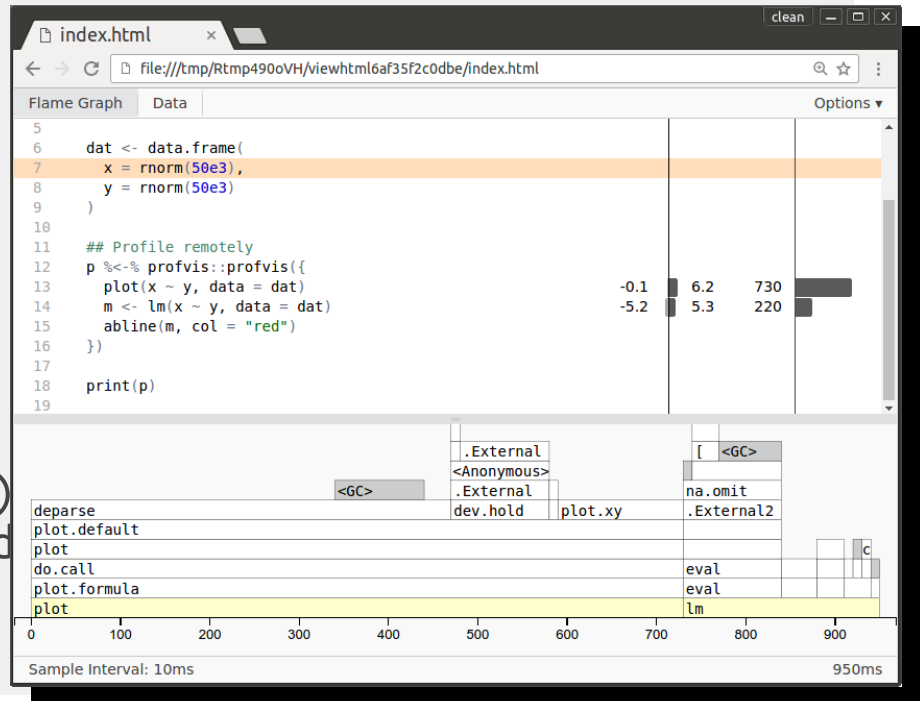
```
> library("future")  
> plan(remote, workers="remote.server.org")
```

```
> library("profvis")
```

```
> dat <- data.frame(  
+   x = rnorm(50e3),  
+   y = rnorm(50e3)  
+ )
```

```
## Profile remotely
```

```
> p %<-% profvis({  
+   plot(x ~ y, data = dat)  
+   m <- lm(x ~ y, data = dat)  
+   abline(m, col = "red")  
+ })
```



```
## Browse locally
```

```
> p
```

A6. Implement Future API for new backend

To implement a new type of future, create the following methods:

- `f <- myfuture({ expr })`
 - creates a future of class `MyFuture` extending `Future`
- `value(f)` for `MyFuture`
 - gets value of future (blocking)
- `resolved(f)` for `MyFuture`
 - checks if future is resolved or not (non-blocking)

A7. Futures I'd like to see

- `plan(r32)`
 - e.g. `RODBC::odbcConnectAccess()` works only on 32-bit R.
- `plan(p2p)`
 - Private and / or community-based peer-to-peer computer cluster
- `plan(rhelp)`
 - Post R scripts to R-help and ask for the results :P

A8. Future improvements

Standardization

- Capturing stdout and stderr uniformly
- Random number generation (Pierre L'Ecuyer's RNG streams)
 - easy to do manually right now

Optional features

- Logging
- Progress bars
- Memoization (caching of results)
- On the-fly time and memory benchmark statistics