Rust

Memory safety is a good thing in that you can’t accidentally use a value that has been deallocated: but even better is that ownership also helps with data races

Rust

Memory safety is a good thing in that you can't accidentally use a value that has been deallocated: but even better is that ownership also helps with data races

A data race can happen when one memory location is accessed by two threads, at least one doing a write

Rust

Memory safety is a good thing in that you can't accidentally use a value that has been deallocated: but even better is that ownership also helps with data races

A data race can happen when one memory location is accessed by two threads, at least one doing a write

A read-only (const) value can be shared; a writable (mutable) value shouldn't be shared (c.f., RW locks)

Rust

Memory safety is a good thing in that you can't accidentally use a value that has been deallocated: but even better is that ownership also helps with data races

A data race can happen when one memory location is accessed by two threads, at least one doing a write

A read-only (const) value can be shared; a writable (mutable) value shouldn't be shared (c.f., RW locks)

The Rust compiler can use ownership to track a value and will spot an attempt to modify a shared value and refuse to compile

Rust

Thus making the programmer face up to the problem and fix it before the code will even compile

Rust

Thus making the programmer face up to the problem and fix it before the code will even compile

Rust developers call this “Fearless Concurrency” as the language itself prevents these kinds of data-race

Rust

Thus making the programmer face up to the problem and fix it before the code will even compile

Rust developers call this “Fearless Concurrency” as the language itself prevents these kinds of data-race

Rust provides both mechanism and analysis for concurrency

Rust

Thus making the programmer face up to the problem and fix it before the code will even compile

Rust developers call this “Fearless Concurrency” as the language itself prevents these kinds of data-race

Rust provides both mechanism and analysis for concurrency

This fixes data races: unfortunately the Rust compiler is not (yet?) able to spot non-data-race race conditions, like deadlock

Rust

Rust has a classical heavyweight thread approach, with a `thread` module that contains functions like `thread::spawn()`

Rust

Rust has a classical heavyweight thread approach, with a `thread` module that contains functions like `thread::spawn()`

This takes a closure as argument as the thing to execute

Rust

Rust has a classical heavyweight thread approach, with a `thread` module that contains functions like `thread::spawn()`

This takes a closure as argument as the thing to execute

```
let threadid = thread::spawn(|| foo(x+1,y-1));
...
let val = threadid.join().unwrap();
```

Rust

What do we do if we need to mutate a shared value in different threads?

Rust

What do we do if we need to mutate a shared value in different threads?

We can use a mutex to sequentialise the accesses

Rust

What do we do if we need to mutate a shared value in different threads?

We can use a mutex to sequentialise the accesses

Mutexes can be shared across threads

Rust

A `Mutex` can be used to wrap any data:

```
let mtx = Mutex::new(v);
```

Rust

A `Mutex` can be used to wrap any data:

```
let mtx = Mutex::new(v);
```

And now the *only* way of accessing the data that used to be in `v` is via the mutex: `let mut data = mtx.lock().unwrap();`

Rust

A `Mutex` can be used to wrap any data:

```
let mtx = Mutex::new(v);
```

And now the *only* way of accessing the data that used to be in `v` is via the mutex: `let mut data = mtx.lock().unwrap();`

An important thing here is that the ownership of the value has passed to within the mutex `mtx`, and so is no longer available from the variable `v`

Rust

A `Mutex` can be used to wrap any data:

```
let mtx = Mutex::new(v);
```

And now the *only* way of accessing the data that used to be in `v` is via the mutex: `let mut data = mtx.lock().unwrap();`

An important thing here is that the ownership of the value has passed to within the mutex `mtx`, and so is no longer available from the variable `v`

This prevents accidental direct access to the data, and this is checked and enforced by the compiler: we can't accidentally use `v`

Rust

A `Mutex` can be used to wrap any data:

```
let mtx = Mutex::new(v);
```

And now the *only* way of accessing the data that used to be in `v` is via the mutex: `let mut data = mtx.lock().unwrap();`

An important thing here is that the ownership of the value has passed to within the mutex `mtx`, and so is no longer available from the variable `v`

This prevents accidental direct access to the data, and this is checked and enforced by the compiler: we can't accidentally use `v`

And we can get mutable access to the data only when locked

Rust

There is no unlock method: the mutex automatically unlocks when the holder goes out of scope

Rust

There is no unlock method: the mutex automatically unlocks when the holder goes out of scope

Thus the programmer can’t forget to unlock a mutex, **or access the data without using the mutex**

Rust

Rust also has barriers, condition variables, channels, etc.

Rust

Rust also has barriers, condition variables, channels, etc.

As always, channels are still an excellent way for threads to communicate, but Rust's ownership model means sharing variables is no longer dangerous: the compiler simply won't let you share things unsafely

Rust

Rust is still being developed, but has already been taken up by many big companies and projects (e.g., by Google for Android, alongside Java and Kotlin; Microsoft are rewriting parts of Windows in Rust)

Rust

Rust is still being developed, but has already been taken up by many big companies and projects (e.g., by Google for Android, alongside Java and Kotlin; Microsoft are rewriting parts of Windows in Rust)

The ownership mechanism is a stumbling block to many programmers coming from other languages

Rust

Rust is still being developed, but has already been taken up by many big companies and projects (e.g., by Google for Android, alongside Java and Kotlin; Microsoft are rewriting parts of Windows in Rust)

The ownership mechanism is a stumbling block to many programmers coming from other languages

Mostly those programmers who don't like the compiler telling them their code is broken: you need to get more things correct before you can compile code

Rust

Rust is still being developed, but has already been taken up by many big companies and projects (e.g., by Google for Android, alongside Java and Kotlin; Microsoft are rewriting parts of Windows in Rust)

The ownership mechanism is a stumbling block to many programmers coming from other languages

Mostly those programmers who don't like the compiler telling them their code is broken: you need to get more things correct before you can compile code

But the learning curve is worth it for the safety achieved

Rust

Exercise For C++ geeks. The idea of tracking ownership ("move semantics") has recently been adopted by C++, though its use is optional and not the default. Read about this

Exercise The Rust compiler guarantees that a mutable (writable) memory location can never be accessed by more than one thread at a time. How might the compiler use this knowledge to optimise operations on that memory location?

Rust

Shared mutable state is the root of all evil. Most languages attempt to deal with this problem through the 'mutable' part, but Rust deals with it by solving the 'shared' part.
From the Rust website

It may be harder to write Rust code than Java code, but it's a lot harder *to write* incorrect *Rust code than* incorrect *Java code*
"Llogiq on stuff" Feb 2016

SISAL

Another single assignment, functional language, this time with *implicit* parallelism

SISAL

Another single assignment, functional language, this time with *implicit* parallelism

Streams and Iteration in a Single Assignment Language, as its name suggests has special regard for streams and iterations

SISAL

Another single assignment, functional language, this time with *implicit* parallelism

Streams and Iteration in a Single Assignment Language, as its name suggests has special regard for streams and iterations

It distinguishes carefully between loops where the computations in the loop body are independent (thus parallelisable, they call them *for-loops*) and those where they are not independent (they call these *iterations*)

SISAL

The for-loop looks like

```
for <range>
   <optional body>
returns <returns clause>
end for
```

All expressions in SISAL return one or more values

SISAL

An example:

```
for i in 1, n
   sqs := vals[i]*vals[i]
returns array of sqs
end for
```

returns an array of the squares of the values

SISAL

An example:

```
for i in 1, n
   sqs := vals[i]*vals[i]
returns array of sqs
end for
```

returns an array of the squares of the values

The effect is like a new instance of `sqs` is made for each value of `i`, then the `array of` operator collects (a reduce operation) them into an array

SISAL

Other reductions are possible

```
for i in 1, n
   sqs := vals[i]*vals[i]
returns array of sqs,
        value of sum sqs
end for
```

returns two things: the array as before, and the sum of the squares; `sum` is another reduction operation

SISAL

The point here is that each squaring is independent

SISAL

The point here is that each squaring is independent

SISAL makes us write the loop in such a way to make this simple and evident

SISAL

The point here is that each squaring is independent

SISAL makes us write the loop in such a way to make this simple and evident

So it may choose to run this in parallel: automatic parallelisation

SISAL

SISAL was briefly popular in the mid-1980s when people were looking for ways for extracting parallelism automatically

SISAL

SISAL was briefly popular in the mid-1980s when people were looking for ways for extracting parallelism automatically

It is an example of a *dataflow language*

SISAL

SISAL was briefly popular in the mid-1980s when people were looking for ways for extracting parallelism automatically

It is an example of a *dataflow language*

These work on the idea that it is the data that should direct the processing

SISAL

SISAL was briefly popular in the mid-1980s when people were looking for ways for extracting parallelism automatically

It is an example of a *dataflow language*

These work on the idea that it is the data that should direct the processing

A spreadsheet is a simple example of the dataflow concept: change the value in a cell and this triggers various (re)computations, possibly running in parallel

SISAL

SISAL was briefly popular in the mid-1980s when people were looking for ways for extracting parallelism automatically

It is an example of a *dataflow language*

These work on the idea that it is the data that should direct the processing

A spreadsheet is a simple example of the dataflow concept: change the value in a cell and this triggers various (re)computations, possibly running in parallel

SISAL is of academic interest, but is not used widely

Strand

A single assignment language reminiscent of Prolog with dataflow (again, mid to late 1980s), declarative

Strand

A single assignment language reminiscent of Prolog with dataflow (again, mid to late 1980s), declarative

There is a single, shared global namespace and threads communicate by writing and reading variables

Strand

A single assignment language reminiscent of Prolog with dataflow (again, mid to late 1980s), declarative

There is a single, shared global namespace and threads communicate by writing and reading variables

If a thread tries to read a variable before it is set, that thread will block

Strand

A single assignment language reminiscent of Prolog with dataflow (again, mid to late 1980s), declarative

There is a single, shared global namespace and threads communicate by writing and reading variables

If a thread tries to read a variable before it is set, that thread will block

Thus we get both message passing and synchronisation

Strand

A single assignment language reminiscent of Prolog with dataflow (again, mid to late 1980s), declarative

There is a single, shared global namespace and threads communicate by writing and reading variables

If a thread tries to read a variable before it is set, that thread will block

Thus we get both message passing and synchronisation

And so variables are also a bit like single-use channels

Strand

Strand only supports parallel composition: i.e., you cannot write sequentially

Strand

Strand only supports parallel composition: i.e., you cannot write sequentially

The dataflow between the variables is all the sequencing we get

Strand

Strand only supports parallel composition: i.e., you cannot write sequentially

The dataflow between the variables is all the sequencing we get

And, conversely, if one expression does not depend on another, that can be run in parallel

Strand

Strand only supports parallel composition: i.e., you cannot write sequentially

The dataflow between the variables is all the sequencing we get

And, conversely, if one expression does not depend on another, that can be run in parallel

Again allowing automatic parallelism

Strand

Code is a list of *rules* rather like Prolog:

```
clause :- guard, guard, ... | body
```

Strand

Code is a list of *rules* rather like Prolog:

```
clause :- guard, guard, ... | body
```

A program consists of many rules

Strand

All rules are eligible for execution at all times as long as all their guard conditions are satisfied

Strand

All rules are eligible for execution at all times as long as all their guard conditions are satisfied

Guards can be evaluated in parallel

Strand

All rules are eligible for execution at all times as long as all their guard conditions are satisfied

Guards can be evaluated in parallel

If a rule is selected, then a new process evaluates the body

Strand

All rules are eligible for execution at all times as long as all their guard conditions are satisfied

Guards can be evaluated in parallel

If a rule is selected, then a new process evaluates the body

If no rules match, then it's an error in your program

Strand

Rules:

```
consumer(X) :- X | eat(X).
producer(Y) :- Y := "food".
```

with program:

```
producer(Z), consumer(Z).
```

the variable `Z` acts as a shared “channel” between the producer and consumer

Strand

As always, there's much more to Strand than this: streams, foreign language interface (to call C, etc.), garbage collection, and so on

Strand

As always, there's much more to Strand than this: streams, foreign language interface (to call C, etc.), garbage collection, and so on

And, just like Prolog, not widely used

Strand

As always, there's much more to Strand than this: streams, foreign language interface (to call C, etc.), garbage collection, and so on

And, just like Prolog, not widely used

It's just not the way most programmers think!

Parallel Languages

Thus there are several ways a language design can avoid races:

Parallel Languages

Thus there are several ways a language design can avoid races:

- have no shared variables (e.g., Erlang)

Parallel Languages

Thus there are several ways a language design can avoid races:

- have no shared variables (e.g., Erlang)
- have no mutable shared variables (e.g., Rust)

Parallel Languages

Thus there are several ways a language design can avoid races:

- have no shared variables (e.g., Erlang)
- have no mutable shared variables (e.g., Rust)
- have no mutation (e.g., Haskell)

Parallel Languages

Thus there are several ways a language design can avoid races:

- have no shared variables (e.g., Erlang)
- have no mutable shared variables (e.g., Rust)
- have no mutation (e.g., Haskell)
- have no parallelism! (e.g., JavaScript, Python)

Parallel Languages

Thus there are several ways a language design can avoid races:

- have no shared variables (e.g., Erlang)
- have no mutable shared variables (e.g., Rust)
- have no mutation (e.g., Haskell)
- have no parallelism! (e.g., JavaScript, Python)

Allowing unrestricted access to shared values (as we are used to in sequential programming) is a sure route to creating races

Parallel Languages

Thus there are several ways a language design can avoid races:

- have no shared variables (e.g., Erlang)
- have no mutable shared variables (e.g., Rust)
- have no mutation (e.g., Haskell)
- have no parallelism! (e.g., JavaScript, Python)

Allowing unrestricted access to shared values (as we are used to in sequential programming) is a sure route to creating races

But having any the above restrictions in a language is guaranteed to irritate some programmers — they don't like being forced to write correct programs!

Parallel Languages

And so on. See Wikipedia!

- C*. Connection Machine, SIMD
- Cω. Cray, modified C, like data parallel Fortran
- Concurrent Euclid. Functional influenced descendant of Pascal
- Data Parallel Haskell.
- E. Secure distributed programming
- Ease. A CSP language
- Fortress. Secure Fortran, implicit parallelism
- Janus. “bag channels” pool-like communications

Parallel Languages

- Joule. Dataflow, like E
- Joyce. Pascal syntax, CSP
- Limbo. Channels
- Lucid. Dataflow
- MultiLisp. Scheme extension, arguments to function calls explicitly evaluated in parallel, lazy evaluation
- NESL. Precursor to Data Parallel Haskell
- Orc. Concurrent, non-deterministic
- Oz. Multiparadigm: dataflow and declarative
- Parlog. Parallel Prolog

Parallel Languages

- SALSA. Actor, runs on Java machine
- Sing#. Extension of C#. Message passing
- SPARK. Based on Ada
- SR. Message passing
- $*$Lisp. Connection Machine
- Turing$+$. Monitors
- XC. Explicit parallelism
- ZPL. Like C/C++, implicit parallelism.

Parallel Languages

Exercise Swift, Rust and Go are all “modern” languages, designed in the current era of parallel hardware. Compare their approaches to parallelism

Parallel Languages

Exercise Swift, Rust and Go are all “modern” languages, designed in the current era of parallel hardware. Compare their approaches to parallelism

Exercise Think about using all of OpenMP, MPI (and CUDA/OpenCl on GPUs) in a single program

Parallel Languages

Exercise Swift, Rust and Go are all “modern” languages, designed in the current era of parallel hardware. Compare their approaches to parallelism

Exercise Think about using all of OpenMP, MPI (and CUDA/OpenCl on GPUs) in a single program

Redo Assignment 1 using Swift, Rust, Go, CUDA, etc.