You might not need your garbage collector**

An introduction to ASAP

Nathan Corbyn

University of Oxford

21st June 2022

Outline

Memory management à la C

Memory management à la C

"*Trust me — I'm a programmer.*"

Memory management à la C

"*Trust me — I'm a programmer.*"

"`Process terminated with signal SIGSEGV`"

malloc() & free()

```c
int* x = (int*) malloc(sizeof(int) * 32);

// ...

free(x);
```

Double free()

```c
int* x = (int*) malloc(sizeof(int) * 32);
// ...
free(x);
// ...
free(x); // Whoops
```

Use after free()

```
int* x = (int*) malloc(sizeof(int) * 32);
// ...
free(x);
// ...
printf("%d", x[4]); // Whoops
```

No free()

```c
int* x = (int*) malloc(sizeof(int) * 32);

// ...

// Whoops?
```

Automatic garbage collection

Automatic garbage collection

- Automate the problem away

Automatic garbage collection

- Automate the problem away
- Have a system monitor heap state and free automatically

Automatic garbage collection

- Automate the problem away
- Have a system monitor heap state and free automatically
- Various approaches & techniques

Automatic garbage collection

- Automate the problem away
- Have a system monitor heap state and free automatically
- Various approaches & techniques
  - Reference counting

Automatic garbage collection

- Automate the problem away
- Have a system monitor heap state and free automatically
- Various approaches & techniques
  - Reference counting
  - Tracing

Automatic garbage collection

- Automate the problem away
- Have a system monitor heap state and free automatically
- Various approaches & techniques
  - Reference counting
  - Tracing
  - Hybrid

Automatic garbage collection

- Automate the problem away
- Have a system monitor heap state and free automatically
- Various approaches & techniques
  - Reference counting
  - Tracing
  - Hybrid
  - Generational

Automatic garbage collection

- Automate the problem away
- Have a system monitor heap state and free automatically
- Various approaches & techniques
  - Reference counting
  - Tracing
  - Hybrid
  - Generational
- Very popular in practice

Automatic garbage collection

Regions & ownership

Regions & ownership

- Ideally: every allocation is freed exactly once

Regions & ownership

- Ideally: every allocation is freed exactly once
- Similar to linear logic

Regions & ownership

- Ideally: every allocation is freed exactly once
- Similar to linear logic
  - Linear assumptions must be used exactly once

Regions & ownership

- Ideally: every allocation is freed exactly once
- Similar to linear logic
  - Linear assumptions must be used exactly once
- Use the type system to enforce this invariant

Regions & ownership

```
fn consume(x: MyRecord) { /* ... */ }

let x = MyRecord { /* ... */ };
consume(x);
consume(x); // ERROR
```

Regions & ownership

```
fn borrow(y: &MyRecord) { /* ... */ }
fn consume(x: MyRecord) { /* ... */ }

let x = MyRecord { /* ... */ };
let y = &x;
borrow(y);
consume(x);
borrow(y); // ERROR
```

Regions & ownership

```
'r: {
    let x: MyRecord + 'r = MyRecord { /* ... */ };
    let y: &'r MyRecord = &'r x;
    borrow(y);
    consume(x);
}
borrow(y); // ERROR
```

Regions & ownership

```
fn borrow<'r>(y: &'r MyRecord) { /* ... */ }
```

Regions & ownership

Regions & ownership

- Implemented successfully in Rust

Regions & ownership

- Implemented successfully in Rust
- Enforces rigid style

Regions & ownership

- Implemented successfully in Rust
- Enforces rigid style
  - Rust provides fallbacks

Regions & ownership

- Implemented successfully in Rust
- Enforces rigid style
  - Rust provides fallbacks
- Steep learning curve

As-static-as-possible (ASAP)

As-static-as-possible (ASAP)

- Introduced by Proust in 2017

As-static-as-possible (ASAP)

- Introduced by Proust in 2017
- Idea: use static analyses to approximate heap liveness and generate appropriate freeing code

Liveness analysis

Liveness analysis

- Let's look at live-variable analysis first

Liveness analysis

```
// ...
let x = foo(a, b, c);
// ...
bar(x);
return a;
```

Liveness analysis

```
// ...                      // {a, b, c}
let x = foo(a, b, c); // {x, a}
// ...
bar(x);                     // {a}
return a;                   // {}
```

Liveness analysis

```
// ...                    // {x, y, z, a, b}
if x == 3 {
    // ...                // {x, y, z}
} else {
    // ...                // {a, b}
}
```

Liveness analysis

```
                        // {}
let mut x = 0;          // {x}
while x <= 3 {          // {}
    x = 4;              // {}
}                       // {}
```

Liveness analysis

```
                        // {}
let mut x = 0;          // {x}
while x <= 3 {          // {x}
    x = 4;              // {x}
}                       // {}
```

Modelling the heap

Modelling the heap

- Now we've covered liveness, let's try and generalise this to heap structures

Modelling the heap

- Now we've covered liveness, let's try and generalise this to heap structures
- We need a way to talk about the heap statically...

Paths

Paths

$$\frac{\tau = \{\cdots ; F : \tau'; \cdots\}}{F : \mathsf{Path}(\tau, \tau')} \text{ (Field)}$$

Paths

$$\frac{\tau = \{\cdots;F:\tau';\cdots\}}{F:\mathsf{Path}(\tau,\tau')}\ \text{(Field)}$$

$$\frac{\tau = \cdots + D(\tau') + \cdots}{D:\mathsf{Path}(\tau,\tau')}\ \text{(Variant)}$$

Paths

Paths

$$\frac{}{\epsilon : \mathsf{Path}(\tau, \tau)} \text{ (Empty)}$$

Paths

$$\frac{}{\epsilon : \mathsf{Path}(\tau, \tau)} \text{ (Empty)}$$

$$\frac{p : \mathsf{Path}(\tau, \tau)}{p^* : \mathsf{Path}(\tau, \tau)} \text{ (Star)}$$

Paths

$$\frac{}{\epsilon : \mathsf{Path}(\tau, \tau)} \; (\mathsf{Empty}) \qquad \frac{p : \mathsf{Path}(\tau, \tau)}{p^* : \mathsf{Path}(\tau, \tau)} \; (\mathsf{Star})$$

$$\frac{p : \mathsf{Path}(\tau, \tau') \qquad q : \mathsf{Path}(\tau', \tau'')}{p \cdot q : \mathsf{Path}(\tau, \tau'')} \; (\mathsf{Seq.})$$

Paths

$$\frac{}{\epsilon : \mathsf{Path}(\tau, \tau)} \text{ (Empty)} \qquad \frac{p : \mathsf{Path}(\tau, \tau)}{p^* : \mathsf{Path}(\tau, \tau)} \text{ (Star)}$$

$$\frac{p : \mathsf{Path}(\tau, \tau') \qquad q : \mathsf{Path}(\tau', \tau'')}{p \cdot q : \mathsf{Path}(\tau, \tau'')} \text{ (Seq.)}$$

$$\frac{p : \mathsf{Path}(\tau, \tau') \qquad q : \mathsf{Path}(\tau, \tau')}{p + q : \mathsf{Path}(\tau, \tau')} \text{ (Alt.)}$$

Example

```
type Unit = {};
type Head = { /* ... */ };

type List = Nil(Unit) + Cons(Cell);
type Cell = { head: Head, tail: List };
```

Example

Example

- Head

$$\texttt{Cons} \cdot \texttt{head}$$

Example

- Head

$$\texttt{Cons} \cdot \texttt{head}$$

- Spine

$$(\texttt{Cons} \cdot \texttt{tail})^*$$

Example

- Head

$$\texttt{Cons} \cdot \texttt{head}$$

- Spine

$$(\texttt{Cons} \cdot \texttt{tail})^*$$

- Elements

$$(\texttt{Cons} \cdot \texttt{tail})^* \cdot \texttt{Cons} \cdot \texttt{head}$$

Example

- Head

$$\texttt{Cons} \cdot \texttt{head}$$

- Spine

$$(\texttt{Cons} \cdot \texttt{tail})^*$$

- Elements

$$(\texttt{Cons} \cdot \texttt{tail})^* \cdot \texttt{Cons} \cdot \texttt{head}$$

- Terminator

$$(\texttt{Cons} \cdot \texttt{tail})^* \cdot \texttt{Nil}$$

Zones

$$[\![l|p]\!] : \mathsf{Stack} \times \mathsf{Heap} \rightarrow \mathcal{P}(\mathsf{Loc})$$

Zones

$$
\begin{aligned}
[\![l | \epsilon]\!](\sigma, \eta) &= \{l\} \\
[\![l | \alpha]\!](\sigma, \eta) &= \begin{cases} \varnothing & \text{if } \tau' \text{ a value type} \\ \{\pi_\alpha(l)(\sigma, \eta)\} & \text{otherwise} \end{cases} \\
[\![l | p + q]\!](\sigma, \eta) &= [\![l | p]\!](\sigma, \eta) \cup [\![l | q]\!](\sigma, \eta) \\
[\![l | p \cdot q]\!](\sigma, \eta) &= \bigcup_{l' \in [\![l | p]\!](\sigma, \eta)} [\![l' | q]\!](\sigma, \eta)
\end{aligned}
$$

Zones

$$[\![l | p^*]\!](\sigma, \eta) = \bigcup_{i \in \omega} Z_i$$

where

$$Z_0 = \{l\}$$
$$Z_{i+1} = \bigcup_{l' \in Z_i} [\![l' | p]\!](\sigma, \eta)$$

Access analysis (*very* vaguely)

```
// ...                          // {(y, F.p), ...}
let x = y.F;                    // {(x, p), ...}
// ...
```

Generating cleaning code

Generating cleaning code

- At each program point we have the set of zones that *may* be accessed

Generating cleaning code

- At each program point we have the set of zones that *may* be accessed
- Looking *between* program points, we'll learn what we can hope to deallocate

Generating cleaning code

Generating cleaning code

- At every program point, compute two sets:

Generating cleaning code

- At every program point, compute two sets:
  - The *matter* set — everything we may still need

Generating cleaning code

- At every program point, compute two sets:
  - The *matter* set — everything we may still need
  - The *antimatter* set — everything we definitely don't need

Generating cleaning code

- At every program point, compute two sets:
  - The *matter* set — everything we may still need
  - The *antimatter* set — everything we definitely don't need
- Generate code to:

Generating cleaning code

- At every program point, compute two sets:
  - The *matter* set — everything we may still need
  - The *antimatter* set — everything we definitely don't need
- Generate code to:
  - 'Mark' the matter set

Generating cleaning code

- At every program point, compute two sets:
  - The *matter* set — everything we may still need
  - The *antimatter* set — everything we definitely don't need
- Generate code to:
  - 'Mark' the matter set
  - `free()` anything in the anti-matter set that isn't marked

Generating cleaning code

- At every program point, compute two sets:
  - The *matter* set — everything we may still need
  - The *antimatter* set — everything we definitely don't need
- Generate code to:
  - 'Mark' the matter set
  - `free()` anything in the anti-matter set that isn't marked
  - Clear the marks

Generating cleaning code

Generating cleaning code

- Various optimisations we can do

Generating cleaning code

- Various optimisations we can do
  - Identify redundancy to minimise work

Generating cleaning code

- Various optimisations we can do
  - Identify redundancy to minimise work
  - Aggregate work across program points to minimise context switching

Generating cleaning code

- Various optimisations we can do
  - Identify redundancy to minimise work
  - Aggregate work across program points to minimise context switching
  - Improve accuracy of analyses

Generating cleaning code

- Various optimisations we can do
  - Identify redundancy to minimise work
  - Aggregate work across program points to minimise context switching
  - Improve accuracy of analyses
  - ...

Generating cleaning code

- Various optimisations we can do
  - Identify redundancy to minimise work
  - Aggregate work across program points to minimise context switching
  - Improve accuracy of analyses
  - ...
- I like to think of the whole technique as staging your tracing collector

Generating cleaning code

- Various optimisations we can do
  - Identify redundancy to minimise work
  - Aggregate work across program points to minimise context switching
  - Improve accuracy of analyses
  - ...
- I like to think of the whole technique as staging your tracing collector
- But — there are some key issues to deal with!

Fixing fixpoints

Fixing fixpoints

- Fixpoints aren't automatically reachable

Fixing fixpoints

- Fixpoints aren't automatically reachable
- For any finite $i \in \omega$

$$\epsilon + p + p \cdot p + p \cdot p \cdot p + \cdots + p^i \neq p^*$$

Fixing fixpoints

- Fixpoints aren't automatically reachable
- For any finite $i \in \omega$

$$\epsilon + p + p \cdot p + p \cdot p \cdot p + \cdots + p^i \neq p^*$$

- People who know about this would say we've violated the *ascending chain condition*

Compact paths

Compact paths

$$\lfloor - \rfloor : \mathsf{Path}(\tau, \tau') \to \underbrace{\mathsf{CPath}(\tau, \tau')}_{\mathsf{Finite}}$$

Compact paths

$$\lfloor - \rfloor : \mathsf{Path}(\tau, \tau') \to \underbrace{\mathsf{CPath}(\tau, \tau')}_{\text{Finite}}$$

- Every path has a corresponding DFA

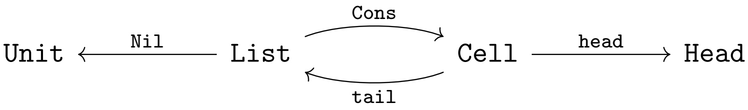
Compact paths

$$\lfloor - \rfloor : \mathsf{Path}(\tau, \tau') \to \underbrace{\mathsf{CPath}(\tau, \tau')}_{\text{Finite}}$$

- Every path has a corresponding DFA
- Idea: bound these automata by the type graph!

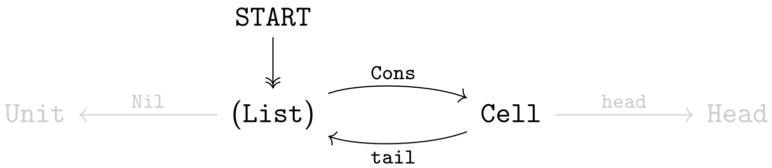
Compact paths

$(\texttt{Cons} \cdot \texttt{tail})^*$



Compact paths

$(\texttt{Cons} \cdot \texttt{tail})^*$



Compact paths

$$
\begin{aligned}
\lfloor \epsilon \rfloor &= \varnothing \\
\lfloor \alpha \rfloor &= \{\tau \xrightarrow{\alpha} \tau'\} \\
\lfloor p \cdot q \rfloor &= \lfloor p \rfloor \cup \lfloor q \rfloor \\
\lfloor p + q \rfloor &= \lfloor p \rfloor \cup \lfloor q \rfloor \\
\lfloor p^* \rfloor &= \lfloor p \rfloor
\end{aligned}
$$

Dealing with aliasing

Dealing with aliasing

- Aliasing causes problems

Dealing with aliasing

- Aliasing causes problems
- Two types:

Dealing with aliasing

- Aliasing causes problems
- Two types:
  - *External* — two distinct zones overlap

Dealing with aliasing

- Aliasing causes problems
- Two types:
  - *External* — two distinct zones overlap
  - *Internal* — multiple routes to a single block

Dealing with aliasing

- Aliasing causes problems
- Two types:
  - *External* — two distinct zones overlap
  - *Internal* — multiple routes to a single block
- Have corresponding analyses:

Dealing with aliasing

- Aliasing causes problems
- Two types:
  - *External* — two distinct zones overlap
  - *Internal* — multiple routes to a single block
- Have corresponding analyses:
  - *Shape* — identifies potential external aliasing

Dealing with aliasing

- Aliasing causes problems
- Two types:
  - *External* — two distinct zones overlap
  - *Internal* — multiple routes to a single block
- Have corresponding analyses:
  - *Shape* — identifies potential external aliasing
  - *Share* — identifies potential internal aliasing

Dealing with aliasing

- Aliasing causes problems
- Two types:
  - *External* — two distinct zones overlap
  - *Internal* — multiple routes to a single block
- Have corresponding analyses:
  - *Shape* — identifies potential external aliasing
  - *Share* — identifies potential internal aliasing
- Interdependent! I refer to them collectively as *implied access*

Accuracy

Accuracy

- Analysing functions and methods in isolation isn't accurate enough for ASAP to perform well

Accuracy

- Analysing functions and methods in isolation isn't accurate enough for ASAP to perform well
- We need to consider the interactions *between* procedures

Accuracy

- Analysing functions and methods in isolation isn't accurate enough for ASAP to perform well
- We need to consider the interactions *between* procedures
- Enter *inter-procedural analysis*

Summaries & amalgamated call-contexts

Summaries & amalgamated call-contexts

- *Summary* — information passed from callee to caller

Summaries & amalgamated call-contexts

- *Summary* — information passed from callee to caller
- *Amalgamated call-context* — information passed from callers to callee

Summaries & amalgamated call-contexts

```
fn foo(a: u64, b: u64, c: u64) -> u64 {
                               // {a, b, c}
    let w = a + b + c;         // {w}
    w                          // {}
}
```

Summaries & amalgamated call-contexts

```
fn foo(a: u64, b: u64, c: u64) -> u64 {
                                // {}
    let w = a + b + c;          // {}
    w                           // {}
}
```

Summaries & amalgamated call-contexts

```
fn foo(a: u64, b: u64, c: u64) -> u64 {
                                // {}
    let w = a + b + c;          // {}
    w                           // {}
}
```

This generalises as before!

Some thoughts

Some thoughts

- Is ASAP best understood as a data-flow analysis or an effect system?

Some thoughts

- Is ASAP best understood as a data-flow analysis or an effect system?
- Can effect polymorphism help improve accuracy?

Some thoughts

- Is ASAP best understood as a data-flow analysis or an effect system?
- Can effect polymorphism help improve accuracy?
- Do we always need compact paths?

Conclusion & future work

Conclusion & future work

- ASAP is a long way from production

Conclusion & future work

- ASAP is a long way from production
- But — early performance data is interesting!

Conclusion & future work

- ASAP is a long way from production
- But — early performance data is interesting!
  - Extremely cache friendly

Conclusion & future work

- ASAP is a long way from production
- But — early performance data is interesting!
  - Extremely cache friendly
  - Small binaries

Conclusion & future work

- ASAP is a long way from production
- But — early performance data is interesting!
  - Extremely cache friendly
  - Small binaries
  - Low memory footprint

Conclusion & future work

- ASAP is a long way from production
- But — early performance data is interesting!
  - Extremely cache friendly
  - Small binaries
  - Low memory footprint
- We still need:

Conclusion & future work

- ASAP is a long way from production
- But — early performance data is interesting!
  - Extremely cache friendly
  - Small binaries
  - Low memory footprint
- We still need:
  - Better understanding of semantics

Conclusion & future work

- ASAP is a long way from production
- But — early performance data is interesting!
  - Extremely cache friendly
  - Small binaries
  - Low memory footprint
- We still need:
  - Better understanding of semantics
  - High performance scanning code

Conclusion & future work

- ASAP is a long way from production
- But — early performance data is interesting!
  - Extremely cache friendly
  - Small binaries
  - Low memory footprint
- We still need:
  - Better understanding of semantics
  - High performance scanning code
  - Proper experimental platform

References I

Nathan Corbyn, *Practical static memory management*, Tech. report, University of Cambridge, 2020, BA Dissertation.

Raphaël L. Proust, *ASAP: as static as possible memory management*, Tech. report, University of Cambridge, 2017, PhD Thesis.