Safe Low-Overhead Memory Management for Concurrency and Parallelism

Denys Shabalin

21 July 2015

Overview

1. Region-based memory in Cyclone
2. Uniqueness and reference immutability for safe parallelism
3. An efficient on-the-fly cycle collection
4. Research proposal

Region-based memory in Cyclone

Dan Grossman, Michael Hicks, Greg Morrisett,
Yanling Wang, Trevor Jim, James Cheney

Safe region-based memory management

Pioneered by M. Tofte and J.P. Talpin who explored the concept in functional programming languages of the ML family.

Cyclone overview

- Close to C syntactically to ease porting of existing programs.
- Adds tagged unions, polymorphism, *regions* etc.
- Guarantees memory safety via static type checking.

Regions in Cyclone

```
void main() {
    region r {
        struct Point*r p = rnew(r) { 10.0, 10.0 };
        printf("Point at (%d, %d)", p->x, p->y);
    }
}
```

Regions in Cyclone

```
void main() {
    region r {
        struct Point*r p = rnew(r) { 10.0, 10.0 };
        printPoint<r>(p)
    }
}

void printPoint<r>(Point*r p) {
    printf("Point at (%d, %d)", p->x, p->y);
}
```

Regions in Cyclone

```
void main() {
    region r {
        struct Point*r p = rnew(r) { 10.0, 10.0 };
        printPoint<r>(p)
    }
}

void printPoint<r>(Point*r p; {r}) {
    printf("Point at (%d, %d)", p->x, p->y);
}
```

Cyclone type system: abstract syntax

$$\tau ::= \tau_1 \xrightarrow{\epsilon} \tau_2 \mid \tau * \rho \mid \mathit{handle}(\rho) \mid ...$$

$$e ::= x_\rho \mid \ * e \mid e_1(e_2) \mid \mathit{rnew}(e_1)\ e_2 \mid ...$$

$$s ::= e \mid \mathit{return}\ e \mid s_1; s_2 \mid \mathit{region}\langle\rho\rangle\ x_\rho\ s \mid ...$$

$$\Gamma ::= \bullet \mid \Gamma, x_\rho : \tau$$

$$\Delta ::= \bullet \mid \Delta, \alpha : \kappa$$

$$\gamma ::= \emptyset \mid \gamma, \epsilon <: \rho$$

$$\epsilon ::= \alpha_1\ \cup\ \alpha_2\ \cup\ ...\ \cup\ \alpha_n\ \cup\ \{\rho_1,\ ..., \rho_m\}$$

$$C ::= \Gamma; \Delta; \gamma; \epsilon$$

Cyclone type system: judgments

Expression typing	$\Delta; \Gamma; \gamma; \epsilon \vdash e : \tau$
Statement typing	$\Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{stmt} s$
Region liveness	$\gamma \vdash \epsilon \Rightarrow \rho$
	$\gamma \vdash \epsilon_1 \Rightarrow \epsilon_2$

Cyclone type system: typing pointer dereference

$$\frac{\Delta; \Gamma; \gamma; \epsilon \vdash e : \tau * \rho \qquad \gamma \vdash \epsilon \Rightarrow \rho}{\Delta; \Gamma; \gamma; \epsilon \vdash *e : \tau}$$

Cyclone type system: typing function application

$$\frac{\Delta; \Gamma; \gamma; \epsilon \vdash e_1 : \tau_2 \xrightarrow{\epsilon_1} \tau \quad \Delta; \Gamma; \gamma; \epsilon \vdash e_2 : \tau_2 \quad \gamma \vdash \epsilon \Rightarrow \epsilon_1}{\Delta; \Gamma; \gamma; \epsilon \vdash e_1(e_2) : \tau}$$

Cyclone: summary

- Region based memory management ties memory management to explicitly delimited blocks of code
- Static safety is ensured through region-annotated reference types and simple effect checking

Uniqueness and reference immutability for safe parallelism

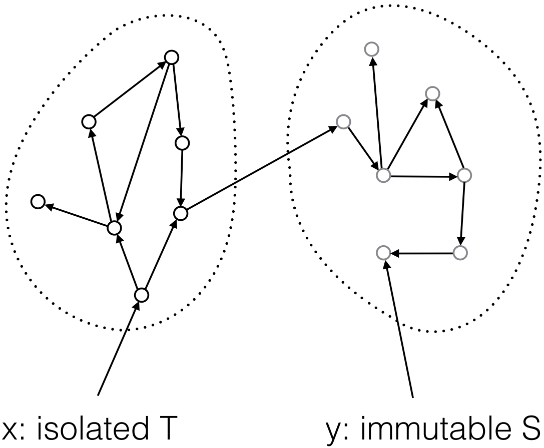
Colin S. Gordon, Matthew J. Parkinson,
Jared Parsons, Aleks Bromfield, Joe Duffy

Uniqueness and reference immutability

Introduces reference qualifiers to the C# language:

- **writable** T
- **readable** T
- **immutable** T
- **isolated** T

Uniqueness and reference immutability



Uniqueness and reference immutability: abstract syntax

$a ::= x = y \mid x.f = y \mid x = y.f \mid x = consume(y.f) \mid ...$

$C ::= a \mid C; C \mid ...$

$T ::= cn$

$TD ::= class\ cn[<: T]\{\ fld *\ \ meth*\ \}$

$p ::= readable \mid writable \mid immutable \mid isolated$

$t ::= int \mid bool \mid p\ \ T$

$\Gamma ::= \epsilon \mid \Gamma, x : t$

Uniqueness and reference immutability: judgements

Command typing	$\Gamma \vdash C \dashv \Gamma$
Program typing	$\vdash P$
	$P \vdash TD$
	$P; TD \vdash fld$
	$P; TD \vdash meth$
Subtyping	$\vdash p \prec p'$
	$\vdash T \prec T'$
	$\vdash t_1 \prec t_2$
Permission combination	$p_1 \rhd p_2 = p_3$
	$p_1 \rhd p_2 T = (p_1 \rhd p_2) T$

Uniqueness and reference immutability: typing

$$\frac{t \neq \mathit{isolated}\ \_}{x : \_,\ y : t \vdash x = y \dashv y : t,\ x : t}$$

$$\frac{\begin{array}{c} t'\ f \in T \\ p \neq \mathit{isolated} \vee t' = \mathit{immutable}\ \_ \\ t' \neq \mathit{isolated}\ \_ \vee p = \mathit{immutable} \end{array}}{x : \_,\ y : p\ T \vdash x = y.f \dashv y : p\ T,\ x : p \triangleright t'}$$

$$\frac{t\ f \in T}{y : \mathit{writable}\ T,\ x : t \vdash y.f = x \dashv y : \mathit{writable}\ T,\ \mathit{RemIso}(x : t)}$$

$$\frac{\mathit{isolated}\ T_f\ f \in T}{y : \mathit{writable}\ T \vdash x = \mathit{consume}(y.f) \dashv y : \mathit{writable}\ T,\ x : \mathit{isolated}\ T}$$

Uniqueness and reference immutability: typing

$$\frac{\begin{array}{c} t'\ m(\overline{u'\ z'})\ p' \in T \qquad \vdash p \prec p' \qquad \overline{\vdash u \prec u'} \\ p = \mathit{isolated} \Rightarrow \\ t \neq \mathit{readable}\ \_ \ \wedge\ t \neq \mathit{writable}\ \_ \\ \wedge\ \mathit{IsoOrImm}(\overline{z : t})\ \wedge\ p' \neq \mathit{immutable} \end{array}}{y : p\ T,\ \overline{z : u} \vdash x = y.m(\overline{z})\ \dashv\ y : p\ T,\ \mathit{RemIso}(\overline{z : t}),\ x : t'}$$

Uniqueness and reference immutability: applications

- Statically enforced data-race freedom
- Optimizations of the GC due to known invariants
- Ownership-based memory management

An efficient on-the-fly cycle collection

Harel Paz, David F. Bacon, Elliot K. Kolodner,
Erez Petrank, V. T. Rajan

Efficient on-the-fly collection: previous work

1. Yossi Levanoni and Erez Petrank. *An on-the-fly reference counting garbage collection for Java*.
2. David F. Bacon and V. T. Rajan. *Concurrent cycle collection in reference counted systems*.
3. Harel Paz, Erez Petrank, and Stephen M. Blackburn. *Age-oriented concurrent garbage collection*.

1. On-the-fly ref. counting garbage collection for Java

Introduces two algorithms:

1. Stop-the-world snapshot algorithm
2. On-the-fly sliding views algorithm

Stop-the-world snapshot collector

- Reference counts are only heap-to-heap.
- No need to maintain reference counts between cycle collections. Out of multiple assignments `obj.slot =` v_1, ..., v_n only RC(v_1) `-= 1` and RC(v_n) `+= 1` are relevant.
- Instead of constantly maintaining reference counts lets just record old value for all of the fields that changed.
- To reclaim an object it must both have 0 reference count and not be referenced from stack or registers.

Stop-the-world snapshot algorithm

Synchronization-free write barrier:

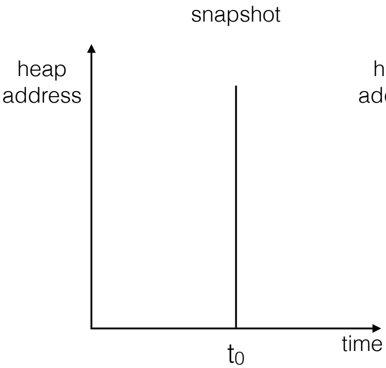
```
Procedure Update(s: Slot, new: Object)
begin
    local old := read(s)
    if not Dirty(s) then
        Buffer_i[CurrPos_i] := <s, old>
        CurrPos_i := CurrPos_i + 1
        Dirty(s) := true
    write(s, new)
end
```

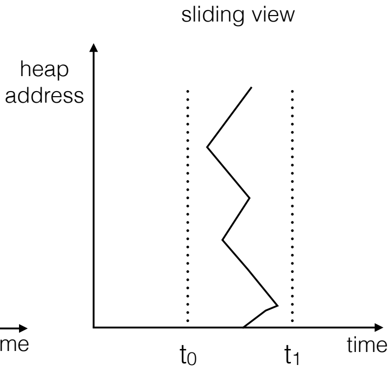
Stop-the-world snapshot algorithm

Collector logic:

```
Procedure Collection-Cycle
begin
    // Stop-The-World
    Read-Current-State
    Update-Reference-Counters
    Read-Buffers
    Fix-Undetermined-Slots
    Reclaim-Garbage
end
```

On-the-fly sliding views collector





On-the-fly sliding views collector

Extra precautions needed:

- Can't collect objects to which new reference has been created during cycle collection.
- To fix this problem a *snooping* mechanism is introduced. It effectively pins all such objects.

On-the-fly sliding views collector

Modified write barrier with snooping:

```
Procedure Update(s: Slot, new: Object)
begin
    local old := read(s)
    if not Dirty(s) then
        Buffer_i[CurrPos_i] := <s, old>
        CurrPos_i := CurrPos_i + 1
        Dirty(s) := true
    write(s, new)
    if Snoop_i then
        Locals_i := Locals_i + new
end
```

2. Concurrent cycle collection in ref. counted systems

- Introduces two collectors: extremely efficient synchronization-free stop-the-world one and less efficient concurrent one.
- Synchronous collector cleans up the cycles starting from suspected cycle roots in $O(N + E)$.
- Linear performance obtained through single graph traversal that colors graph as it goes through it.
- Sufficient constant benefits by special treatment of acyclic data structures.

3. Age-oriented concurrent garbage collection

- Develops idea of generational collection into on-the-fly setting.
- Age-oriented is defined as:
  1. *Always collects the entire heap*.
  2. During collection treats each generation differently.
- Mark-and-sweep for young generation and on-the-fly reference counting for the old generation.

Efficient on-the-fly collection: crème de la crème

- Low pause times thanks to sliding views.
- Efficient cycle collection through a single traversal.
- Generational with dynamically sized young generation that is handled by mark-and-sweep tracing collector.

State of memory management

	Manual	Regions	Ownership	GC
Performance	Good	Good	Good	Bad
Predictability	Good	Good	Good	Bad
Notational convenience	Good	Bad	Bad	Good
Safety	Bad	Good	Good	Good

Research proposal

Research proposal

Develop memory management system where:

- Application developers don't need to worry about memory management.
- Expert library developers are able to tune their projects by providing fine-grain memory management hints that exploit domain knowledge for their projects.

Research proposal

- Low-pause on-the-fly GC as a baseline
- Optional region annotations to hint at expected object lifetimes for performance critical sections of code.
- Effectively "programmable" garbage collection.

Questions?