Precise Garbage Collection in D

D-Conference 2013

Rainer Schütze

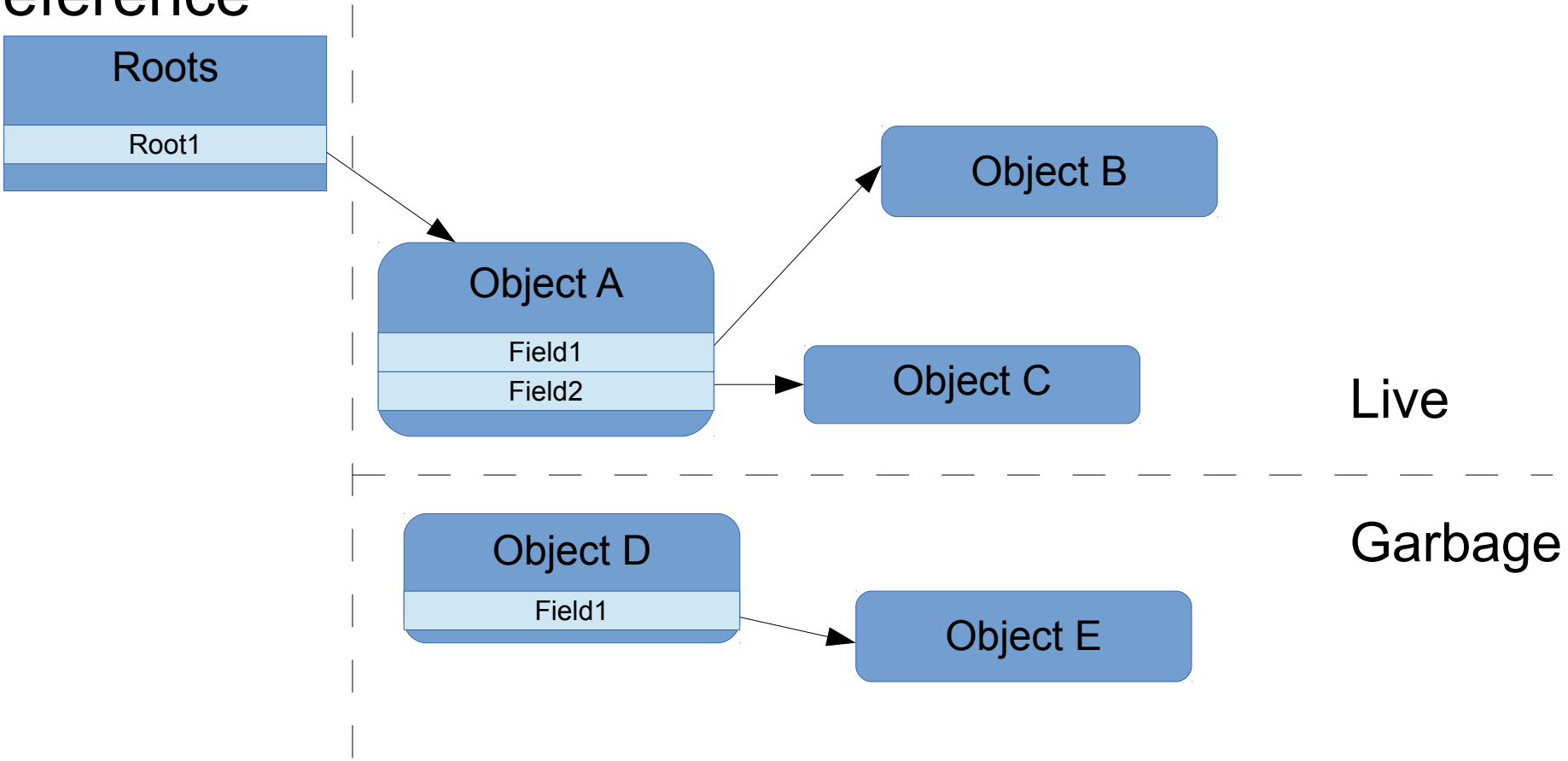
Steinberg Media Technologies GmbH
Visual D

Overview

- Introduction to garbage collection
- D's current GC
- Precise GC implementation
- Further improvements

Basic Garbage Collection

- Automatic memory management releasing unreferenced memory:
- Free objects not reachable from roots through any reference



Garbage Collection

- Benefits:
  - Ownership management unnecessary
  - Memory safety with multi-threading
  - easy use of array slices, delegates with stack closures
- Drawbacks:
  - Delayed memory reclamation
  - Pause times
  - Pointer read/write-barrier
  - Vulnerability against false pointers
- Program efficiency depends on application

Garbage Collectors

- Mark-sweep
  - Tracing, non-moving, fragmentation
- Mark-compact
  - Tracing, moving
- Copying
  - Scanning, moving
- Reference counting
  - Read/Write-Barrier, non-moving, cycles
- Combinations of GCs, generational, concurrent

Conservative GC

- No type information for memory
- Assumptions by most garbage collectors:
  - References aligned to machine word
  - Interior pointers, but no derived pointers
- False pointers: non-pointer values interpreted as memory references
- Cannot move objects because it must not modify false pointers

D's Garbage Collector

- Conservative Mark-Sweep
- Stop-the-world: all threads paused during tracing
- Segregated fits allocator with free-lists
  - Bin size 2^N for $4 \leq N < 12$
  - Large Object Space for allocations > 2kB

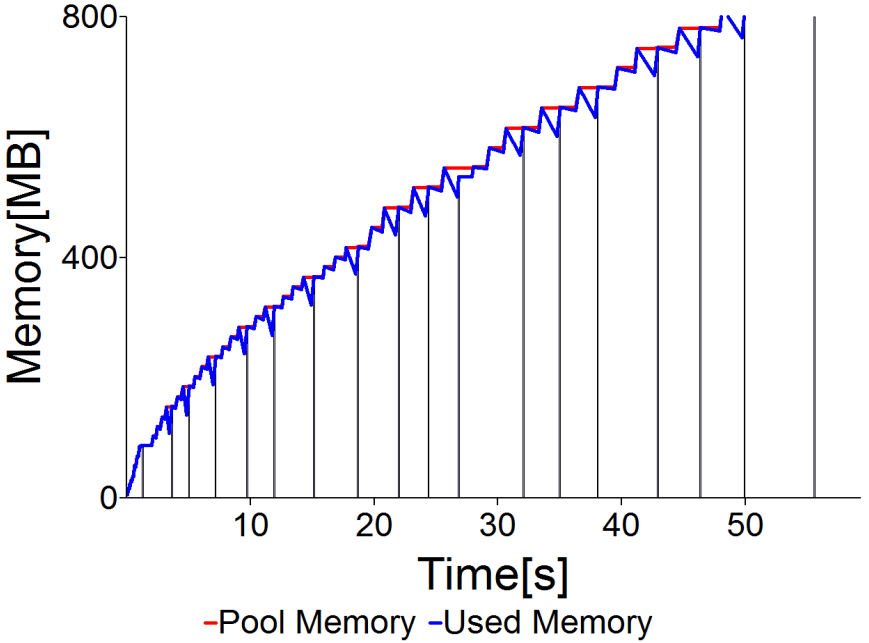
  Assuming similar distribution of used object sizes over time, fragmentation only an issue for large objects
- Optimization: D-Compiler generates „no pointer fields in class/struct“ info to mark allocation

Example: Visual D parser

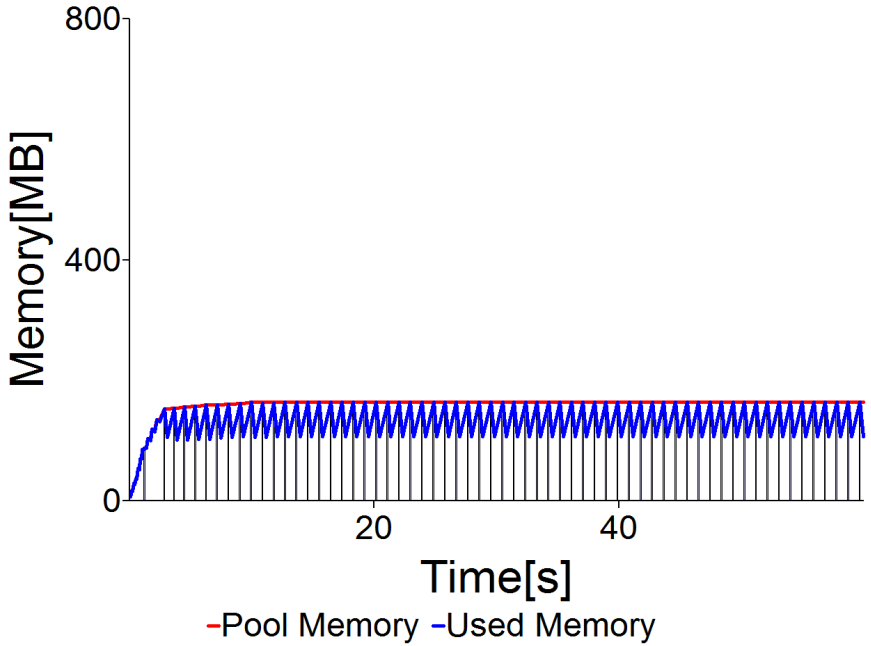
```
// simulating user edits
Module mod;
foreach(i; 0..100)
    mod = parseFile("std/datetime.d");
```

- >34000 LOC, >1.5 MB
- Creates > 350000 AST nodes
- AST needs 50 MB of memory

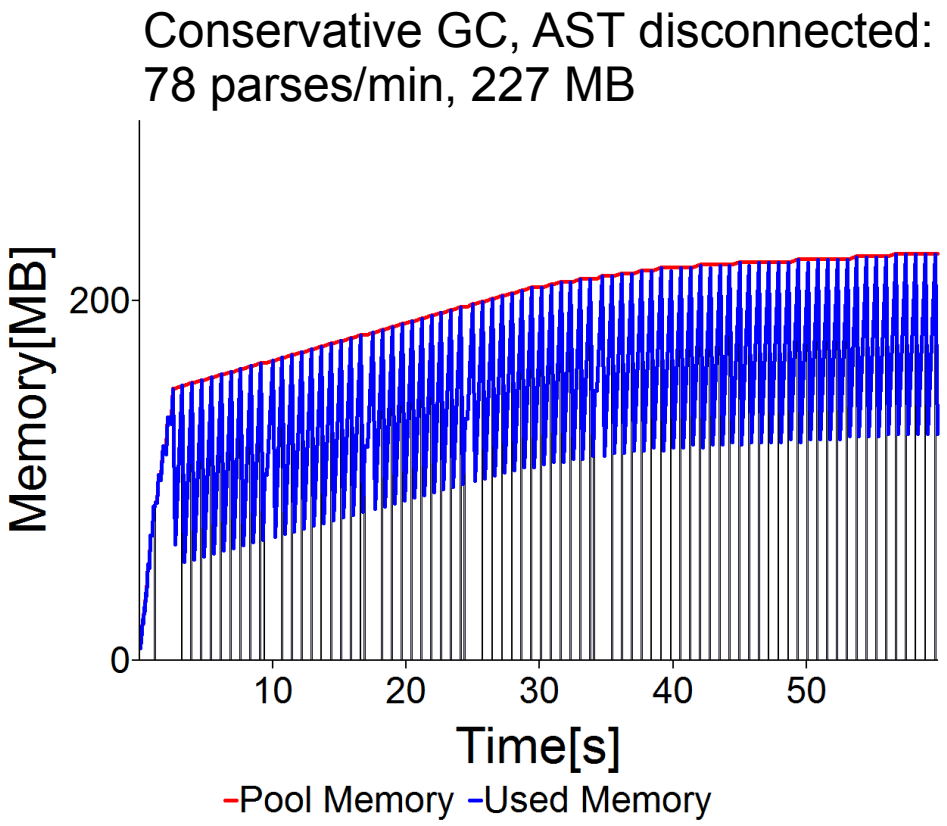
Conservative GC:
18 parses/min, 915 MB, out of memory

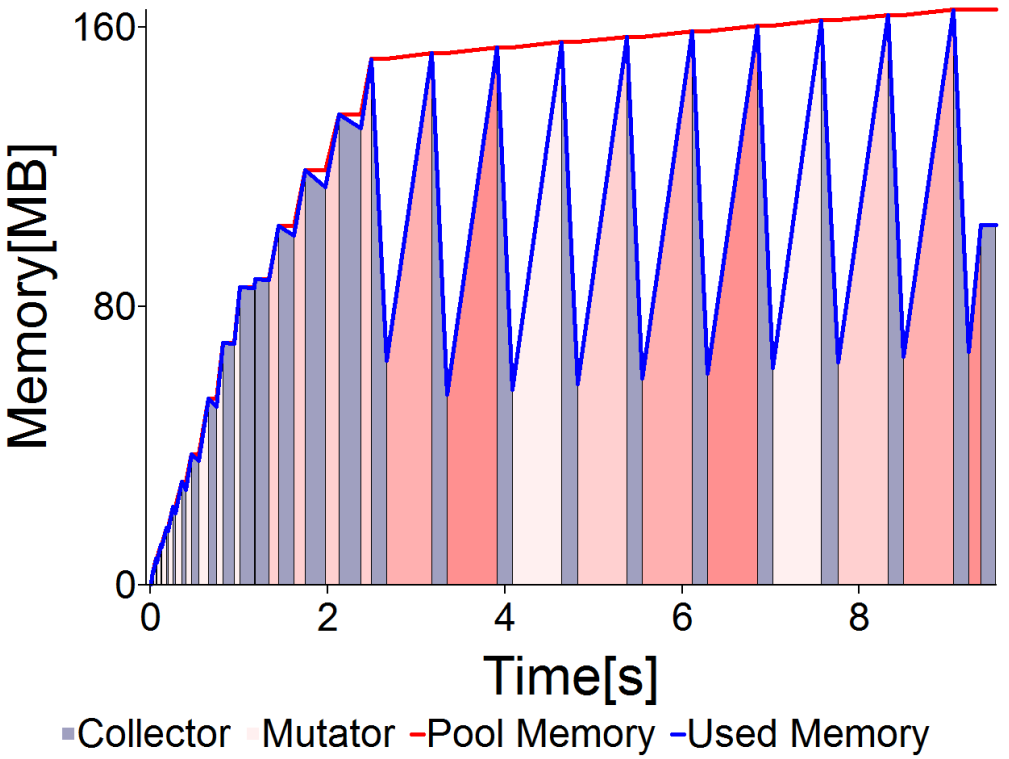


Manual unsafe GC.free:
79 parses/min, 154 MB



Visual D parser: Memory usage





False pointers

- 32-bit: A 4 MB allocation is hit by a uint with random value with a probability of 0.1%, 1000 values pin it to memory with a probability of 63%
- „Good“ false pointers: GUID, hash values, encoding tables, strings
- 64-bit: false pointers less likely, but can still exist: user process space limited to
  - Linux: 47 bits, heap starts at high addresses
  - OS X: 47 bits, heap starts at 0x1_0000_0000
  - Windows: 43 bits, heap starts at low addresses
  - Field alignment can increase probability of false pointers:

    struct Slot { uint hash; Object value }

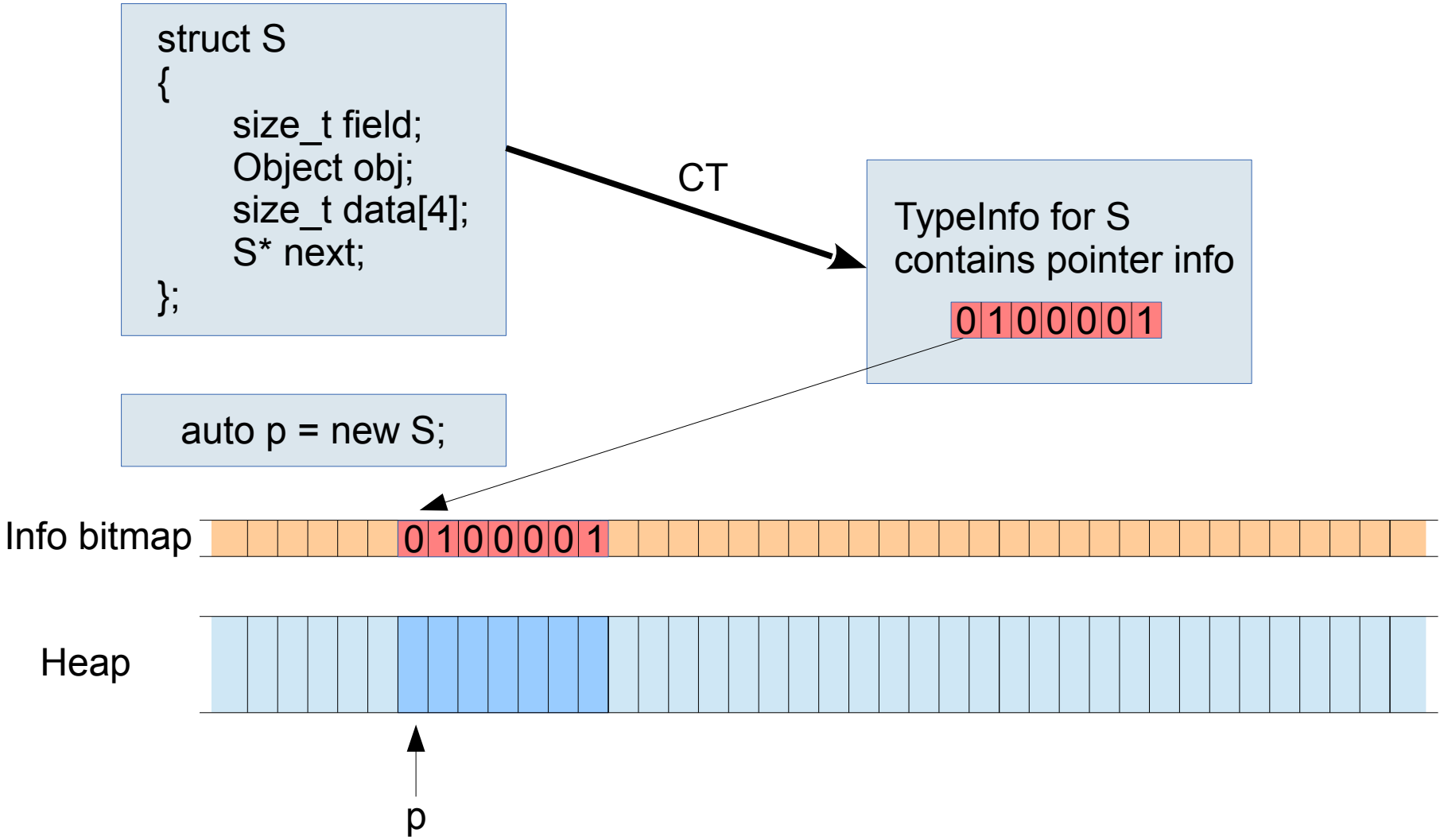
Precise GC

- Uses type info to distinguish pointer fields from non-pointer fields.
- Memory areas
  - Heap
  - Global data in binary image
  - TLS data
  - Stack, Registers
  - Manual managed memory

Precise GC for D

- Based on GSoC 2012 project by Antti-Ville Tuuainen, mentored by David Simcha
- Compile time generated bitmap with pointer-info for each word in a struct/class
- GC.malloc copies bitmap into memory allocated together with pool memory
- Marking only traces pointers with corresponding bit set in bitmap
- Allows “emplacing” new types into partial memory areas.

Precise GC Design



Standard Approach

- Store type info pointer per allocation block
- Needs more memory for allocations < 128/512 bytes (32/64 bit), less for larger allocations
- Why not?
  - Before object: either object no longer aligned to 16 byte or 16 byte overhead added
  - At the end of allocation block: more work to find it, interferes with array handling
  - In preallocated side-memory: needs pointer/granule
  - Cannot “emplace” type info

RTInfo generation

```
class TypeInfo
{
      // […] other members
    @property immutable(void)* rtInfo() nothrow pure const @safe;
}
```

```
class TypeInfo_Class : TypeInfo // same for TypeInfo_Struct
{
      // […] other members
      immutable(void)*    m_rtInfo;
}
```

- Compiler fills m_rtInfo with result of RTInfo!T

```
template RTInfo(T)
{
    enum RTInfo = gc.gctemplates.RTInfoImpl!T;
}
```

- Future: library defined TypeInfo!T

RTInfo!T Implementation 1/2

```
template RTInfoImpl(T)
{
    immutable bmp = bitmap!T();
    enum RTInfoImpl = bmp.ptr;
}

size_t[bitmapSize!T + 1] bitmap(T)()
{
    size_t[bitmapSize!T + 1] arr;
    bitmapImpl!(Unqual!T)(arr.ptr + 1);
    arr[0] = allocatedSize!T;
    return arr;
}

void bitmapImpl(T)(size_t* p)
{
    static if(is(T == class))
        mkBitmapComposite!(T)(p, 0);
    else
        mkBitmap!(Unqual!T)(p, 0);
}
```

```
void mkBitmapComposite(T)(size_t* p,
                          size_t off)
{
    static if (is(T P == super))
        static if(P.length > 0)
            mkBitmapComposite!(P[0])(p, off);

    alias typeof(T.tupleof) TTypes;
    foreach(i, _; TTypes)
    {
        enum cur_off = T.tupleof[i].offsetof;
        alias Unqual!(TTypes[i]) U;
        mkBitmap!U(p, off + cur_off);
    }
}
```

RTInfo!T Implementation 2/2

```
void mkBitmap(T)(size_t* arr, size_t offset)
{
        static if (is(T == struct) || is(T == union))     { mkBitmapComposite!T(arr, offset); }
    else static if (is(T == class) || is(T == interface)) { setbit(arr, offset); }
    else static if (is(T == void))                        { setbit(arr, offset); }
    else static if (isBasicType!(T)())                    {}
    else static if (is(T F == F*) && is(F == function)) {}
    else static if (is(T P == U*, U))                     { setbit(arr, offset); }
    else static if (is(T == delegate))                    { setbit(arr, offset); }
    else static if (is(T D == U[], U))                    { setbit(arr, offset + size_t.sizeof); }
    else static if (is(T A == U[K], U, K))                { setbit(arr, offset); }
    else static if (is(T E == enum))                      { mkBitmap!E(arr, offset); }
    else static if (is(T E == typedef))                   { mkBitmap!E(arr, offset); }
    else static if (is(T S : U[N], U, size_t N)) {
        for (size_t i = 0; i < N; i++)
            mkBitmap!(Unqual!U)(arr, offset + i * U.sizeof); }
    else static assert(false);
}
void setbit()(size_t* arr, size_t offset)
{
    size_t ptroff = offset/bytesPerPtr;
    arr[ptroff/ptrPerBmpWord] |= 1 << (ptroff % ptrPerBmpWord);
}
```

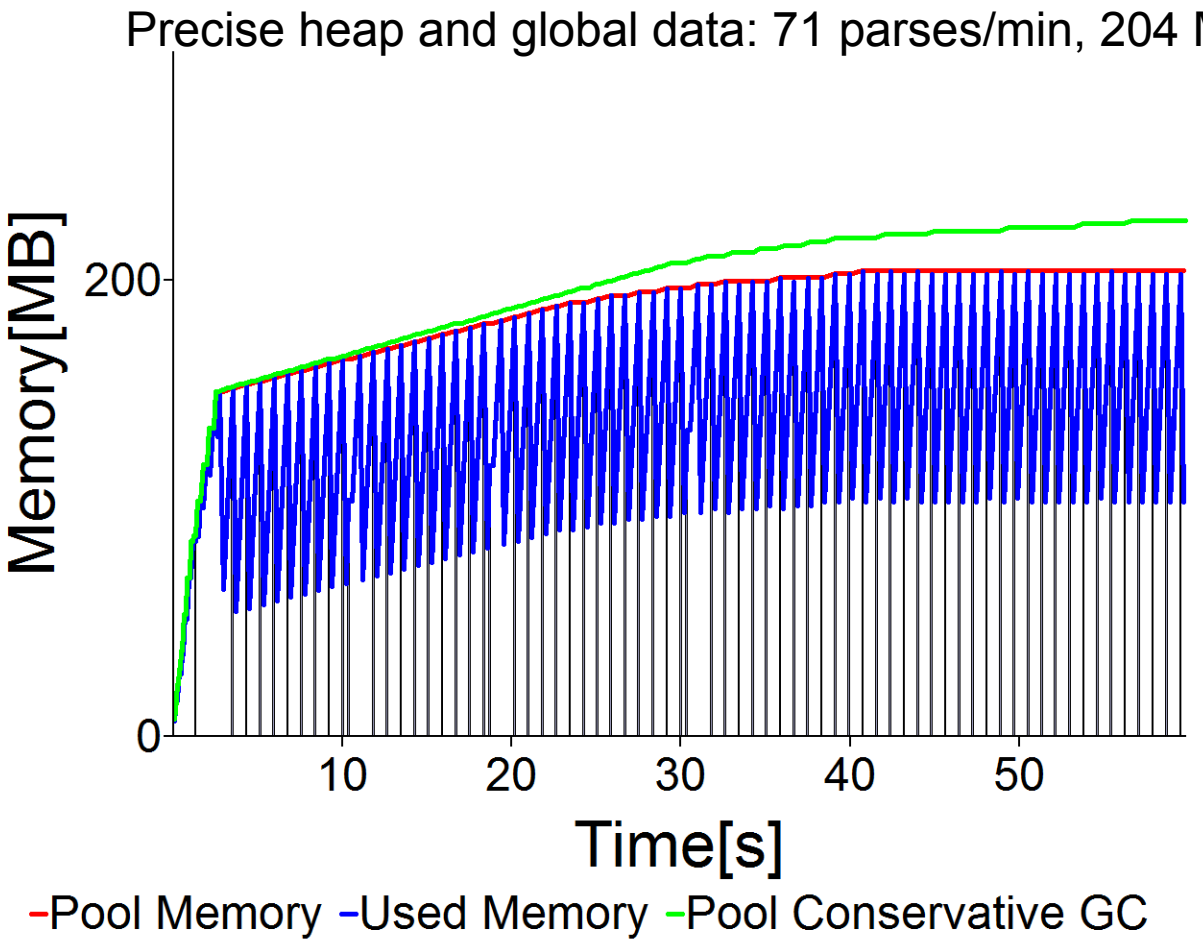
Runtime interface

```
void*      gc_malloc( size_t sz, uint ba = 0, const TypeInfo = null );
void*      gc_calloc( size_t sz, uint ba = 0, const TypeInfo = null );
BlkInfo_ gc_qalloc( size_t sz, uint ba = 0, const TypeInfo = null );
void*      gc_realloc( void* p, size_t sz, uint ba = 0, const TypeInfo = null );
size_t    gc_extend( void* p, size_t mx, size_t sz );
size_t   gc_emplace( void* p, size_t sz, const TypeInfo = null );
```

```
class GC {
    void *malloc(size_t size, uint ba, const TypeInfo ti)
    {
        size_t alloc_size;
        void* p = doAlloc(size,ba,&alloc_size);
        if (!(ba & BlkAttr.NO_SCAN))
            setPointerBitmap(p, pool, size, alloc_size, ti);
        return p;
    }
}
```

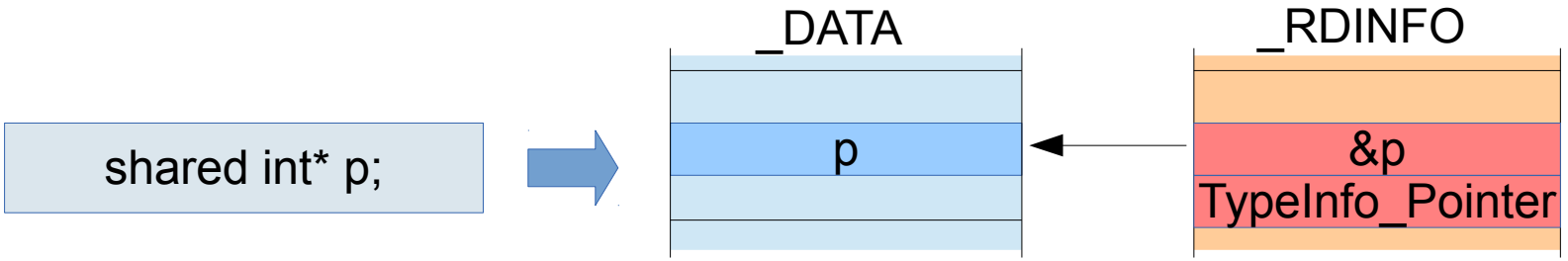
- Move copying T.init into malloc?

Improvements for the parser?



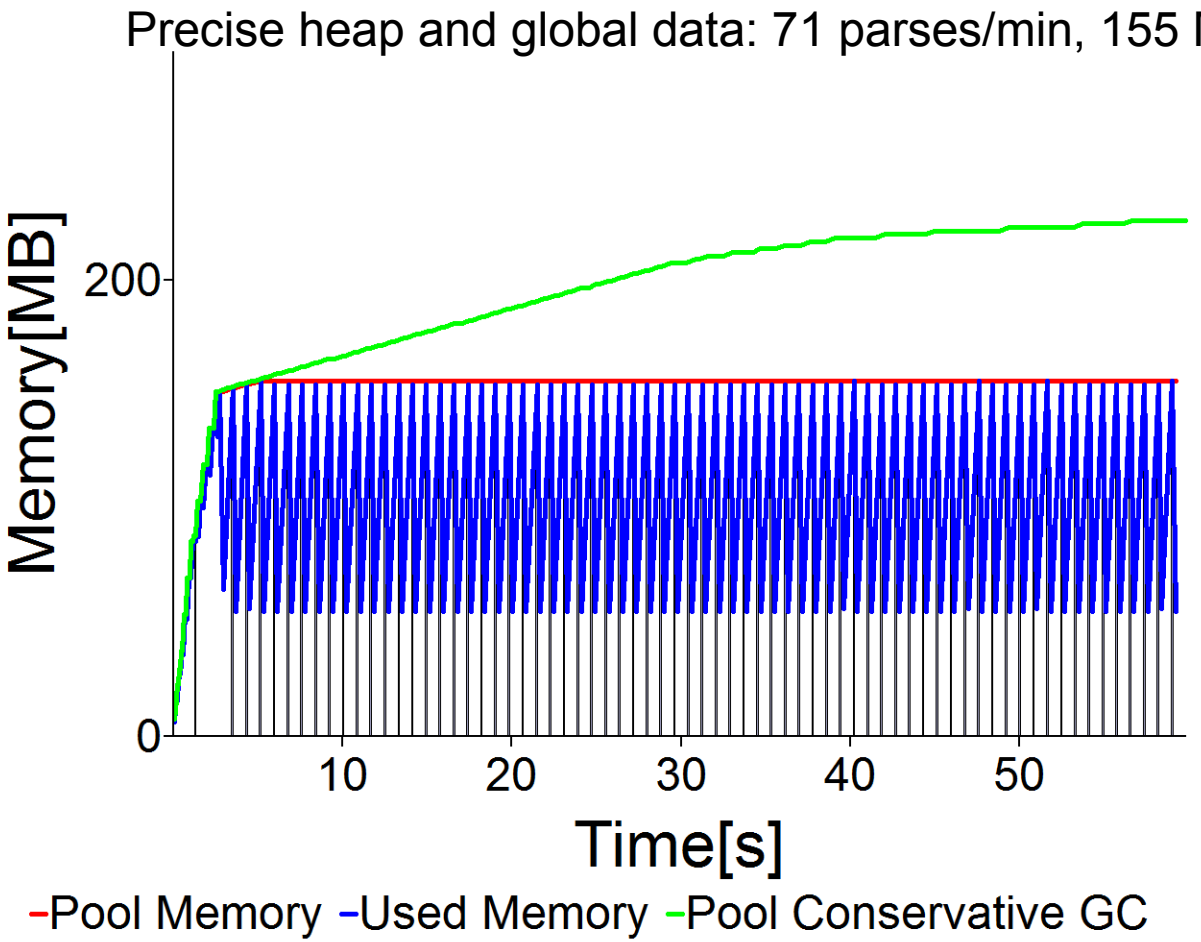
Precise GC for Global Data

- For all global/TLS variables containing references, the compiler generates (address,TypeInfo) pairs and places them into a data section to be combined by the linker



- Better: define data by an RDInfo(alias var) template, but
  - Currently unable to specify data section
  - Data relocations not possible for TLS variables
- GC uses TypeInfo.rtInfo to scan the _DATA memory
- Ambiguity: cannot distinguish class reference and instance

Results for the parser example



Almost Precise GC

- Memory areas
  - Heap ✓
  - Global data in binary image ✓
  - TLS data ✓
  - Stack (stomp stack with -gx)
  - Registers
  - Manual managed memory: treat as global data with roots ✓

Restrictions

- No type info for delegate closures
- Not possible to move objects:
  - False pointers must not be modified
  - Cannot distinguish between actual references and ambiguous data (e.g. unions, void[]), another bit per word in RTInfo could do that

    => Objects only referred to by unambiguous references can be moved
- Extra care needed when using “emplace”
- RDInfo only generated for Win32

Outlook

- Provide tools to improve GC
  - pragma(data_seg,”RDINFO”)
  - Library defined TypeInfo!T
  - Type description of closures
  - TypeInfo for class reference
- Allow implementing/attaching to existing modern GC (generational, concurrent)
  - Library defined pointer type to implement write barrier
  - Allow mostly-moving collector by extending RTInfo!T to detect ambiguous pointers
  - Reduce the number of possible false pointers on the stack

Thanks for listening

- Available at https://github.com/rainers/druntime/tree/gcx_precise
- Questions?