Process-wide type and bounds checking
(via an alliance of many language implementations)

Stephen Kell

stephen.kell@cl.cam.ac.uk

Computer Laboratory
University of Cambridge
“Join me, and together we can rule all the languages”

(illustration: sirustalcelion)
Problems

- retains boundary between “native” and “managed”
- requires buy-in
- ... whereas diversity is inevitable
An alternative to the Empire

(photo: brionv)
Rebels’ manifesto

- accommodate diversity of language
- accommodate diversity of implementations
- support interoperability across languages
- no boundary between “native” and “managed”
- compatibility
- support from below
Founders of the alliance
Introducing liballocs

- extending Unix processes with in(tro)spection
- via a whole-process meta-level protocol
- protocol is implemented by each allocator
  - VMs’ heap allocators
  - native allocators (malloc(), custom allocators…)
  - stack allocators
  - “static” allocators, mmap() etc.
- → abstraction ≈ “typed allocations”
- … covering entire process

Advertisement: see my paper at Onward! later this year.
What is “managed”? [“native”]?

1. [lack of] garbage collector(s)
2. [un]checked errors
3. [lack of] reflection
What is “managed”? [“native”]?

1. [lack of] garbage collector(s)
2. [un]checked errors (clean [vs corrupting] failure)
3. [lack of] reflection

Most of this talk:

- how to do 2 and 3 embracing native code
- focus on C as the “hard + important” case
- so far, the most developed use-case of liballocs
if (obj->type == OBJ_COMMIT) {
    if (process_commit(walker, (struct commit *)obj))
        return -1;
    return 0;
}
How to implement “unsafe” languages safely

```c
if (obj->type == OBJ_COMMIT) {
    if (process_commit(walker, (struct commit *)obj))
        return -1;
    return 0;
}
```

CHECK this (at run time)
if (obj->type == OBJ_COMMIT) {
    if (process_commit(walker, (struct commit *)obj))
        return −1;
    return 0;
}

... while being

- binary-compatible
- source-compatible
- reasonably fast
- using a mostly-generic (not C-specific) infrastructure
$ crunchcc -o myprog ... # calls host cc
libcrunch: the user's-eye view

- $ crunchcc -o myprog ...
  # calls host cc
- $ ./myprog
  # runs normally
libcrunch: the user’s-eye view

- $ crunchcc -o myprog ...  # calls host cc
- $ ./myprog            # runs normally
- $ LD_PRELOAD=libcrunch.so ./myprog # does checks
libcrunch: the user’s-eye view

$ crunchcc -o myprog ...  # calls host cc
$ ./myprog             # runs normally
$ LD_PRELOAD=libcrunch.so ./myprog # does checks

myprog: Failed __is_a_internal(0x5a1220, 0x413560 a.k.a. "uint$32") at 0x40dade, allocation was a heap block of int$32 originating at 0x40daa1
libcrunch: the user’s-eye view

$ crunchcc -o myprog ... # calls host cc

$ ./myprog # runs normally

$ LD_PRELOAD=libcrunch.so ./myprog # does checks

myprog: Failed __is_a_internal(0x5a1220, 0x413560 a.k.a. "uint$32") at 0x40dade, allocation was a heap block of int$32 originating at 0x40daa1

```c
struct {int x; float y;} z;
int *x1 = &z.x; // ok
int *x2 = (int*) &z; // check passes
int *y1 = (int*) &z.y; // check fails!
int *y2 = &((&z.x)[1]); // need bounds check
return &z; // need GC–alike
```
if (obj->type == OBJ_COMMIT) {
    if (process_commit(walker,
        (struct commit *)obj))
        return -1;
    return 0;
}
if (obj->type == OBJ_COMMIT) {
    if (process_commit(walker,
                 (assert(_is_a(obj, "struct_commit")),
                 (struct commit *)obj)))
        return -1;
    return 0;
}
How it works for C code, in a nutshell

```c
if (obj->type == OBJ_COMMIT) {
    if (process_commit(walker,  
                     (assert(__is_a(obj, "struct_commit")),  
                     (struct commit *)obj)))
        return -1;
    return 0;
}
```

Want a runtime with the power to

- tracking allocations
- with type info
- efficiently → fast __is_a() function

... i.e. what liballocs does!
What is an allocation?

- static memory
- stack memory
- heap memory
  - returned by `malloc()` – “level 1” allocation
  - returned by `mmap()` – “level 0” allocation
  - (maybe) memory issued by user allocators...

Runtime keeps *indexes* for each kind of memory...
Hierarchical model of allocations

- mmap(), sbrk()
- libc malloc()
- custom malloc()
- custom heap (e.g. Hotspot GC)
- obstack (+ malloc)
- gslice
- client code
- client code
- client code
- client code
- client code
- client code
Representation of data types

```c
struct ellipse {
    double maj, min;
    struct {
        double x, y;
    } ctr;
};
```

- use the linker to keep them unique
- → “exact type” test is a pointer comparison
- `__is_a()` is a short search
A language-agnostic model of data types: DWARF debugging info

```
$ cc -g -o hello hello.c && readelf -wi hello | column

<b>:TAG_compile_unit
   AT_language : 1 (ANSI C)
   AT_name     : hello.c
   AT_low_pc   : 0x4004f4
   AT_high_pc  : 0x400514

<7ae>:TAG_pointer_type
   AT_byte_size: 8
   AT_type     : <0x2af>

<76c>:TAG_subprogram
   AT_name     : main

<c5>: TAG_base_type
   AT_byte_size : 4
   AT_encoding  : 5 (signed)
   AT_low_pc    : 0x4004f4
   AT_high_pc   : 0x400514
   AT_name      : int

<2af>:TAG_pointer_type
   AT_byte_size: 8
   AT_type     : <0x2b5>

<2b5>:TAG_base_type
   AT_byte_size: 1
   AT_encoding : 6 (char)
   AT_name     : char
```

```
What data type is being malloc()’d?

- ... infer from use of sizeof
- dump typed allocation sites from compiler

Inference: intraprocedural “sizeofness” analysis

- e.g. size_t sz = sizeof (struct Foo); /* ... */; malloc(sz);
- some subtleties: e.g. malloc(sizeof (Blah) + n * sizeof (Foo))
Solved problems

- typed stack storage
- typed heap storage
- support \{custom, nested\} heap allocators
- fast run-time metadata
- polymorphic allocation sites (e.g. \texttt{sizeof (void*)})
- subtler C features (function pointers, \texttt{vaargs}, unions)
- non-standard C idiom (too sloppy for \_\_is\_a())
- understanding the invariant ("no bad pointers, if…")
- relating to C standard
Metadata queries are difficult

Native objects are trees; no descriptive headers!

VM-style objects: “no interior pointers”
To query heap pointers...

- use `malloc()` hooks...
- which keep an *index* of the heap
- in a *memtable*
  - efficient *address-keyed* associative map
  - must support (some) range queries
- storing object’s metadata

Memtables make aggressive use of virtual memory

- `libcrunch` contains many memtables
- not all populated by hooking allocator
Big picture of our heap memtable

- Index by high-order bits of virtual address
- Entries are one byte, each covering 512B of heap
- Interior pointer lookups may require backward search

- Pointers encoded compactly as local offsets (6 bits)
- Instrumentation adds a trailer to each heap chunk

Entries are one byte, each covering 512B of heap.

Interior pointer lookups may require backward search.
<table>
<thead>
<tr>
<th>bench</th>
<th>normal/s</th>
<th>crunch %</th>
<th>nopreload</th>
<th>onlymeta</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>4.95</td>
<td>+6.8%</td>
<td>+1.4%</td>
<td>+2.6%</td>
</tr>
<tr>
<td>gcc</td>
<td>0.983</td>
<td>+160%</td>
<td>– %</td>
<td>+14.9%</td>
</tr>
<tr>
<td>gobmk</td>
<td>14.6</td>
<td>+11%</td>
<td>+2.0%</td>
<td>+4.1%</td>
</tr>
<tr>
<td>h264ref</td>
<td>10.1</td>
<td>+3.9%</td>
<td>+2.9%</td>
<td>+0.9%</td>
</tr>
<tr>
<td>hmmer</td>
<td>2.16</td>
<td>+8.3%</td>
<td>+3.7%</td>
<td>+3.7%</td>
</tr>
<tr>
<td>lbm</td>
<td>3.42</td>
<td>+9.6%</td>
<td>+1.7%</td>
<td>+2.0%</td>
</tr>
<tr>
<td>mcf</td>
<td>2.48</td>
<td>+12%</td>
<td>(−0.5%)</td>
<td>+3.6%</td>
</tr>
<tr>
<td>milc</td>
<td>8.78</td>
<td>+38%</td>
<td>+5.4%</td>
<td>+0.5%</td>
</tr>
<tr>
<td>sjeng</td>
<td>3.33</td>
<td>+1.5%</td>
<td>(−1.3%)</td>
<td>+2.4%</td>
</tr>
<tr>
<td>sphinx3</td>
<td>1.60</td>
<td>+13%</td>
<td>+0.0%</td>
<td>+8.7%</td>
</tr>
<tr>
<td>perlbench</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Not only types, but also bounds

- **libcrunch** is now pretty good at run-time type checking
- supports idiomatic C, source- and binary-compatibly
- what about bounds checks? (+ temporal checks?)
Not only types, but also bounds

- **libcrunch** is now pretty good at run-time type checking
- supports idiomatic C, source- and binary-compatibly
- what about bounds checks? (+ temporal checks?)

```c
struct {int x; float y;} z;
int *x1 = &z.x; // ok
int *x2 = (int*) &z; // check passes
int *y1 = (int*) &z.y; // check fails!
int *y2 = &((&z.x)[1]); // need bounds check
return &z; // need GC–alike
```
Not only types, but also bounds

- **libcrunch** is now pretty good at run-time type checking
- supports idiomatic C, source- and binary-compatibly
- what about bounds checks? (+ temporal checks?)

```c
struct {int x; float y;} z;
int *x1 = &z.x; // ok
int *x2 = (int*) &z; // check passes
int *y1 = (int*) &z.y; // check fails!
int *y2 = &((&z.x)[1]); // ***
return &z; // need GC–alike
```
Existing bounds checkers use per-pointer metadata

Memcheck (coarse), ASan (fine-ish), SoftBound (fine) …

- overhead at best 50–100% (ASan & SoftBound)
- problems mixing uninstrumented code (libraries)
- false positives for some idiomatic code!

Insight: \((Ptr, T_{Ptr}, T_{Alloc})\) implies bounds for \(Ptr\)!
Why per-pointer metadata is not enough

```
Why per-pointer metadata is not enough

struct ellipse {
    struct point {
        double x, y;
    } ctr;
    double maj;
    double min;
} my_ellipses[3];

struct point {
    double x, y;
} ctr;
double maj;
double min;
} my_ellipses[3];
```
Why per-pointer metadata is not enough

```c
struct ellipse {
    struct point {
        double x, y;
    } ctr;
    double maj; double min;
} my_ellipses[3];
```

```
<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctr</td>
<td>3.5</td>
<td>8.0</td>
</tr>
<tr>
<td>maj</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>min</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ctr</td>
<td>1.0</td>
<td>1.5</td>
</tr>
<tr>
<td>maj</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>min</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>ctr</td>
<td>6.5</td>
<td>4</td>
</tr>
<tr>
<td>maj</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
```
Without type information, pointer bounds lose precision

```c
struct ellipse {
    struct point {
        double x, y;
    } ctr;
    double maj; double min;
} my_ellipses[3];
```

<table>
<thead>
<tr>
<th>ctr</th>
<th>x</th>
<th>3.5</th>
<th>8.0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>y</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>maj</td>
<td>1.0</td>
<td>1.5</td>
<td></td>
</tr>
<tr>
<td>min</td>
<td>5</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

```c
p_f = (ellipse*) p_d
```

```
p_base
```

```
p_limit
```

```
ellipse
```
Given allocation type and pointer type, bounds are implicit

```c
struct ellipse {
    struct point {
        double x, y;
    } ctr;
    double maj; double min;
} my_ellipses[3];

p_e = &my_ellipses[1]
```

<table>
<thead>
<tr>
<th>ellipse[3]</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ctr x 3.5</td>
<td>y 8.0</td>
</tr>
<tr>
<td></td>
<td>maj 2</td>
<td>min 7</td>
</tr>
<tr>
<td></td>
<td>ctr x 1.0</td>
<td>y 1.5</td>
</tr>
<tr>
<td></td>
<td>maj 5</td>
<td>min 8</td>
</tr>
<tr>
<td></td>
<td>ctr x 6.5</td>
<td>y -2.0</td>
</tr>
<tr>
<td></td>
<td>maj 4</td>
<td>min 4</td>
</tr>
</tbody>
</table>
Given allocation type and pointer type, bounds are implicit

<table>
<thead>
<tr>
<th></th>
<th>ctr x</th>
<th></th>
<th>ctr x</th>
<th></th>
<th>ctr x</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>maj</td>
<td>1.0</td>
<td></td>
<td>maj</td>
<td>3.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>min</td>
<td>1.5</td>
<td></td>
<td>min</td>
<td>8.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

double p_d = &p_e->ctr.x

```c
struct ellipse {
    struct point {
        double x, y;
    } ctr;
    double maj;
    double min;
} my_ellipses[3];
```
Given allocation type and pointer type, bounds are implicit

If only we knew the *type* of the storage!
Casts affect bounds: a real example

```c
struct driver { /* ... */ } *d = /* ... */;
struct i2c_driver { /* ... */ struct driver driver; /* ... */ };

#define container_of(ptr, type, member) \  
    ((type *)( (char *)(ptr) − offsetof(type,member) ))

i2c_drv = container_of(d, struct i2c_driver , driver);
```
Casts affect bounds: a real example

```c
struct driver {
    /* ... */
} *d = /* ... */;
struct i2c_driver {
    /* ... */
    struct driver driver;
    /* ... */
};

#define container_of(ptr, type, member) \
    ((type*)( (char*)(ptr) − offsetof(type,member) ))

i2c_drv = container_of(d, struct i2c_driver, driver);
```

- bounds of `d`: just the smaller struct
- bounds of the `char*`: the whole allocation
- bounds of `i2c_drv`: the bigger struct
In progress: libcrunch bounds checker

Using per-allocation metadata, not per pointer:

- avoid these false positives
- avoid libc wrappers, ...
- robust to uninstrumented callers/callees
- performance?

Making it fast:

- cache bounds: make pointers “locally fat, globally thin”
On x86-64, use noncanonical addresses as trap reps

(ask me!)
Bounds checking status

Does it work?

■ yes!

Is it fast?

■ not yet – basic optimisations still to-do

How fast will it be?

■ no idea yet; hopefully competitive or better
■ fewer checks: per-derive, not per-deref
■ less metadata being moved around (heap pointers)
Extra ingredients for a safe implementation of C−ε

- check union access
- check variadic calls
- always initialize pointers
- protect \{code, pointers\} from writes through char*
- check memcpy(), realloc(), etc..
- allocate address-taken locals on heap not stack
- add a GC (improve on Boehm)

Code remaining unsafe:

- reflection (e.g. stack walkers)

Surprisingly perhaps, allocators are not inherently unsafe
Conclusions

- liballocs sits under language impls
- … providing process-wide reflection-like services
- libcunch extends it to check types
- per-allocation metadata better than per-pointer

Hypothesis: unsafety is a property of C implementations

- most code can do without inherently unsafe features
- “fast enough, safe enough” should be doable

Ask me about

- native ↔ JavaScript interop using liballocs + V8

Thanks for your attention. Questions?
The invariant for C

To enforce “all memory accesses respect allocated type”:

- every live pointer respects its *contract* (pointee type)
- must also check unsafe loads/stores *not* via pointers
  - unions, varargs

Most contracts are just “points to declared pointee”

- `void**` and family are subtler (not `void*`)
The effective type of an object for an access to its stored value is the declared type of the object, if any.\textsuperscript{87}) If a value is stored into an object having no declared type through an lvalue having a type that is not a character type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify the stored value. If a value is copied into an object having no declared type using \texttt{memcpy} or \texttt{memmove}, or is copied as an array of character type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one. For all other accesses to an object having no declared type, the effective type of the object is simply the type of the lvalue used for the access.
The effective type of an object for an access to its stored value is the declared type of the object, if any.\textsuperscript{87) If a value is stored into an object having no declared type through an lvalue having a type that is not a character type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify the stored value. If a value is copied into an object having no declared type using \texttt{memcpy} or \texttt{memmove}, or is copied as an array of character type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one. For all other accesses to an object having no declared type, the effective type of the object is simply the type of the lvalue used for the access.

Instead:

- all allocations have $\leq 1$ effective type
- stack, locals / actuals: use declared types
- heap, \texttt{alloca()}: use \textit{allocation site} (+ finesse)
- trap \texttt{memcpy()} and reassign type
Memory-correctness vs type-correctness

Related properties checked by existing tools

- spatial m-c – bounds (SoftBound, Asan)
- temporal$_1$ m-c – use-after-free (CETS, Asan)
- temporal$_2$ m-c – initializedness (Memcheck, Msan)
- oblivious to data types!

Slow!

- metadata per \{value, pointer\}
- check on use
Memory-correctness vs type-correctness

Related properties checked by existing tools

- spatial m-c – bounds (SoftBound, Asan)
- temporal \(_1\) m-c – use-after-free (CETS, Asan)
- temporal \(_2\) m-c – initializedness (Memcheck, Msan)
- oblivious to data types!

**Slow! Faster:**

- metadata per \(\{\text{value, pointer}\}\) allocation
- check on use create

```
// a check over object metadata... guards creation of the pointer
(assert(_is_a(obj, "struct_commit")), (struct commit *)obj)
```
#define LIBCRUNCH_TRAP_TAG_SHIFT 48

inline void * __libcrunch_trap (const void *ptr, unsigned short tag)
{
    return (void *) (((uintptr_t) ptr) ^ (((uintptr_t) tag) << LIBCRUNCH_TRAP_TAG_SHIFT));
}

Tag allows distinguishing different kinds of trap rep:

- LIBCRUNCH_TRAP_ONE_PAST
- LIBCRUNCH_TRAP_ONE_BEFORE
What is “type-correctness”?

“Type” means “data type”

- instantiate = allocate
- concerns storage
- “correct”: reads and writes respect allocated data type
- cf. memory-correct (spatial, temporal)

Languages can be “safe”; programs can be “correct”
Telling libcrunch about allocation functions

LIBALLOCS_ALLOC_FNS="xalloc(zZ)p xmalloc(Z)p xrealloc(pZ)p"
LIBALLOCS_SUBALLOCS_FNS="ggc_alloc(Z)p ggc_alloc_cleared(Z)p"
export LIBALLOCS_ALLOC_FNS
export LIBALLOCS_SUBALLOCS_FNS
## Non-difficulties

<table>
<thead>
<tr>
<th>my_ellipse</th>
</tr>
</thead>
<tbody>
<tr>
<td>maj</td>
</tr>
<tr>
<td>min</td>
</tr>
<tr>
<td>ctr</td>
</tr>
<tr>
<td>x</td>
</tr>
<tr>
<td>y</td>
</tr>
</tbody>
</table>

```c
struct ellipse {
    double maj;
    double min;
    struct point {
        double x, y;
    } ctr;
}
```

- function pointers (most of the time)
- **void** pointers, **char** pointers
- integer ↔ pointer casts
- custom allocators, memory pools etc.

### Give up on:

- escapingly address-taken union members
- avoidance of `sizeof`
__is_a, containment...

Pointer $p$ might satisfy __is_a($p$, $T$) for $T_0$, $T_1$, ...

```c
struct ellipse {
    double maj;
    double min;
    struct point {
        double x, y;
    } ctr;
}
```

- &my_ellipse “is” ellipse and double
- &my_ellipse.ctr “is” point and double
- a.k.a. containment-based “subtyping”

→ libcrunch implements __is_a() appropriately...
Other solved problems

Structure “subtyping” via prefixing

- relax to _like_a() check

Opaque types

- relax to _named_a() check

“Open unions” like sockaddr

- _like_a() works for these too
Remaining awkwards

- `alloca`
- `unions`
- `varargs`
- generic use of non-generic pointers (`void**`, …)
- casts of function pointers *to non-supertypes* (of func’s t)
Remaining awkward

- `alloca`
- `unions`
- `varargs`
- generic use of non-generic pointers (`void**`, …)
- casts of function pointers *to non-supertypes* (of func’s `t`)

All solved/solvable with some extra instrumentation

- supply our own `alloca`
- instrument writes to unions
- instrument calls via `varargs` lvalues; use own `va_arg`
- instrument writes through `void**` (check invariant!)
- optionally instr. *all* indirect calls
deployed binaries (with data-type assertions)

/lib/foo
/lib/libxyz.so

deployment information (with allocation site information)

/bin/.debug/foo
/lib/.debug/libxyz.so

precompute unique data types

libcrunch.so
/bin/.uniqtyp/foo.so

load, link and run (ld.so)

program image

heap_index

0xdeadbeef, "Widget"?

true

_is_a

uniqtypes
What happens at run time?

Program image

__is_a(0xdeadbee8, __uniqtype_double)?

true

lookup(0xdeadbee8)
- allocsite: 0x8901234,
- offset: 0x8

lookup(0x8901234)
- &__uniqtype_ellipse

find(
- &__uniqtype_double,
- &__uniqtype_ellipse,
- 0x8)

found

heap_index

allocsites

uniqtypes
Getting from objects to their metadata

Recall: binary & source compatibility requirements

- can’t embed metadata into objects
- can’t change pointer representation
- → need out-of-band ("disjoint") metadata

Pointers can point anywhere inside an object

- which may be stack-, static- or heap-allocated
Indexing heap chunks

Inspired by free chunk binning in Doug Lea’s `malloc`...
Indexing heap chunks

Inspired by free chunk binning in Doug Lea’s `malloc`...

... but index *allocated* chunks binned by *address*
How many bins?

Each bin is a linked list of heap chunks

- thread next/prev pointers through allocated chunks...
- also store metadata (allocation site address)
- overhead per chunk: one word + two bytes

Finding chunk is $O(n)$ given bin of size $n$

- → want bins to be as small as possible
- Q: how many bins can we have?
- A: lots... really, *lots*!
Really, how big?

Bin index resembles a linear page table. Exploit

- sparseness of address space usage
- lazy memory commit on “modern OSes” (Linux)

Reasonable tuning for `malloc` heaps on Intel architectures:

- one bin covers 512 bytes of VAS
- each bin’s head pointer takes one byte in the index
- covering $n$-bit AS requires $2^{n-9}$-byte bin index
Indexing the heap with a memtable is...

- more VAS-efficient than shadow space (SoftBound)
- supports > 1 index, unlike placement-based approaches

Memtables are versatile

- buckets don’t have to be linked lists
- tunable size / coverage (limit case: bitmap)

We also use memtables to

- index every mapped page in the process (“level 0”)
- index “deep” (level 2+) allocations
- index static allocations
- index the stack (map PC to frame uniqtype)
Other flavours of check

__is_a is a nominal check, but we can also write

- __like_a – “structural” (unwrap one level)
- __refines – padded open unions (à la sockaddr)
- __named_a – opaque workaround

... or invent your own!
Link-time interventions

We also interfere with linking:

- link in uniqtypes referred to by each .o’s checks
- hook allocation functions
- ... distinguishing wrappers from “deep” allocators

Currently provide options in environment variables...

```
LIBCRUNCH_ALLOC_FNS="xcalloc(zZ) xmalloc(Z) xrealloc(pZ) xmallocz(Z)"
LIBCRUNCH_LAZY_HEAP_TYPES="__PTR_void"
```