Run-time type checking of whole programs

and other stories ■ .

Stephen Kell

stephen.kell@cl.cam.ac.uk

Computer Laboratory
University of Cambridge

Wanted (naive version): check this!

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

Wanted (naive version): check this!

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

CHECK this

(at run time)

Wanted (naive version): check this!

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

CHECK this

(at run time)

But also wanted:

- binary-compatible
- source-compatible
- reasonable performance
- avoid being C-specific!*

* mostly...

Wanted (naive version): check this!

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

CHECK this

(at run time)

But also wanted:

- binary-compatible
- source-compatible
- reasonable performance
- avoid being C-specific!*

* mostly...

... in fact, a general-purpose "dynamic" run-time (ask me)

The main part of this talk in one slide

I describe libcrunch, which is

- an infrastructure for run-time type checking
- encodes type checks as assertions over reified data types
- per-language front-ends (C; C++, Fortran, . . .)
- support idiomatic unsafe code, unmodified*
- target: safe *assuming memory safety*
- no binary interface changes

(* but sometimes out-of-band guidance helps)

Why care about unsafe languages?

- fine control of resource utilisation
- talk directly to operating system
- talk directly to hardware
- freedom to {simulate, violate} abstractions
- re-use existing code (a *huge* investment)
- unsafe is the “hard / general” case

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”

The user's eye view

- `$ crunchcc -o myprog ...   # + other front-ends`

The user's eye view

- `$ crunchcc -o myprog ...   # + other front-ends`
- `$ ./myprog                 # runs normally`

The user's eye view

- `$ crunchcc -o myprog ...  # + other front-ends`
- `$ ./myprog  # runs normally`
- `$ LD_PRELOAD=libcrunch.so ./myprog # does checks`

The user's eye view

- `$ crunchcc -o myprog ...  # + other front-ends`
- `$ ./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`

How it works for C code, in a nutshell

```
if (obj->type == OBJ_COMMIT) {
    if (process_commit(walker,

                       (struct commit *)obj))
        return -1;
    return 0;
}
```

How it works for C code, in a nutshell

```
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

```
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 magical powers

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

What does a C compiler *not* check?

```
int a = 1;
char *b = ...;
void f(double);

f(a);              // okay -- compiler adds conversion
b = a;             // not okay -- compiler tells us
f(b);              // not okay -- compiler tells us
f(*(double*)b);    // depends...
```

Want to check what the compiler punts on

- use of pointers ("distant" accesses)
- also (rarer): unions, varargs functions

Memory-correctness vs type-correctness (1)

Pointer-y things 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)
- nothing to do with types!

Slow!

- metadata per {value, pointer}
- check on use

Memory-correctness vs type-correctness (1)

Pointer-y things 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)
- nothing to do with types!

~~Slow!~~ Faster:

- metadata per ~~{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)
```

Memory-correctness vs type-correctness (2)

For now, *assume* memory-correct execution

- "also use one of those other tools"

Then do only the additional checks s.t.

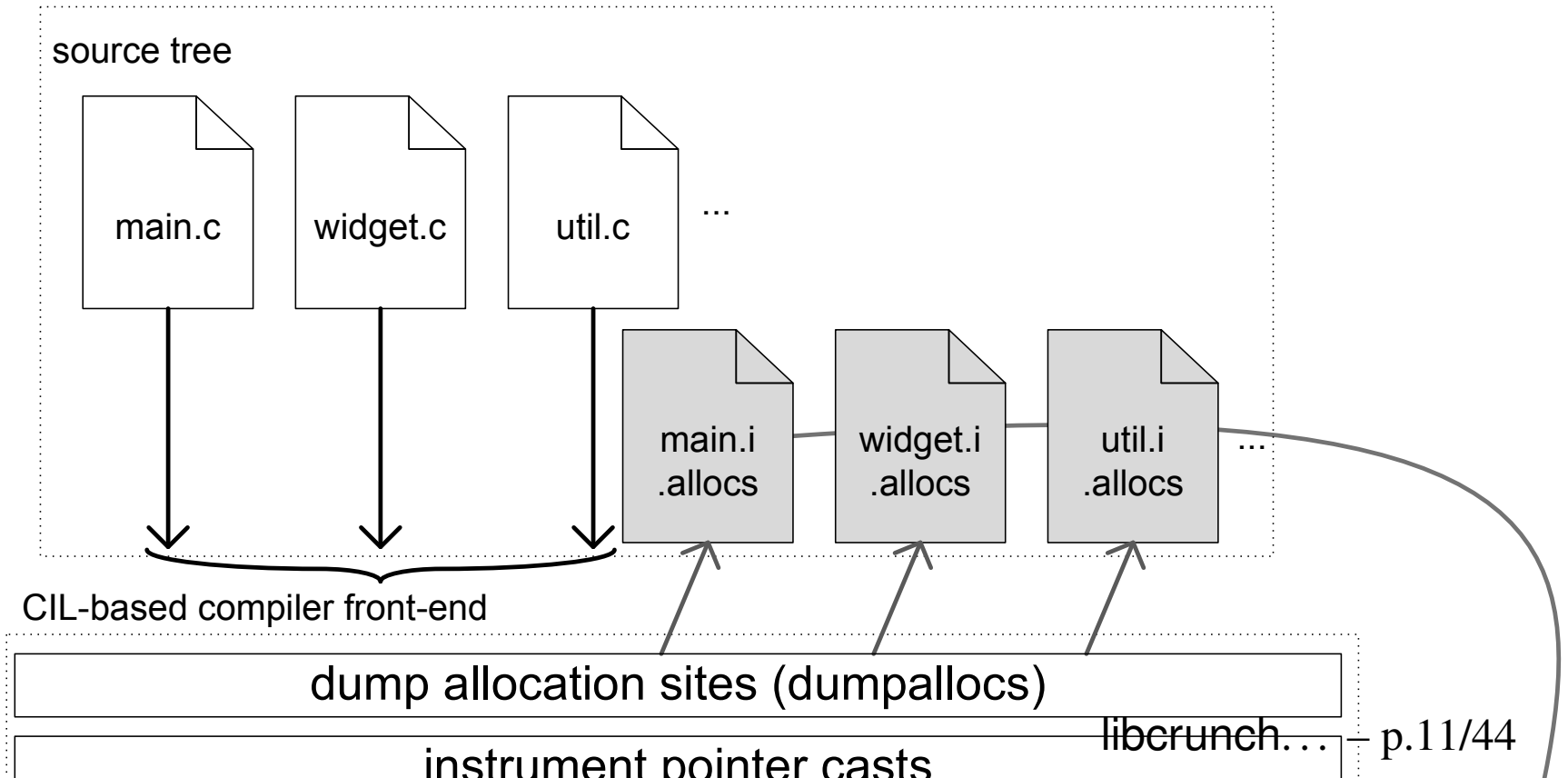
- all memory accesses respect memory's *allocated type*

. . . which, for C, can be done by maintaining an invariant:

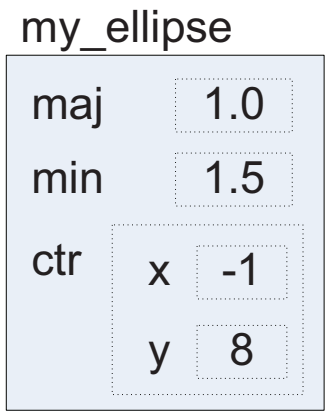
- every live pointer respects its *contract* (pointee type)
- must also check unsafe loads/stores *not* via pointers
  - unions, varargs

What data type is being malloc()'d?

- . . . guess from use of sizeof
- dump *typed allocation sites* from compiler
- guessing is moderately clever
  - e.g. malloc(sizeof (Blah) + n * sizeof (Foo))



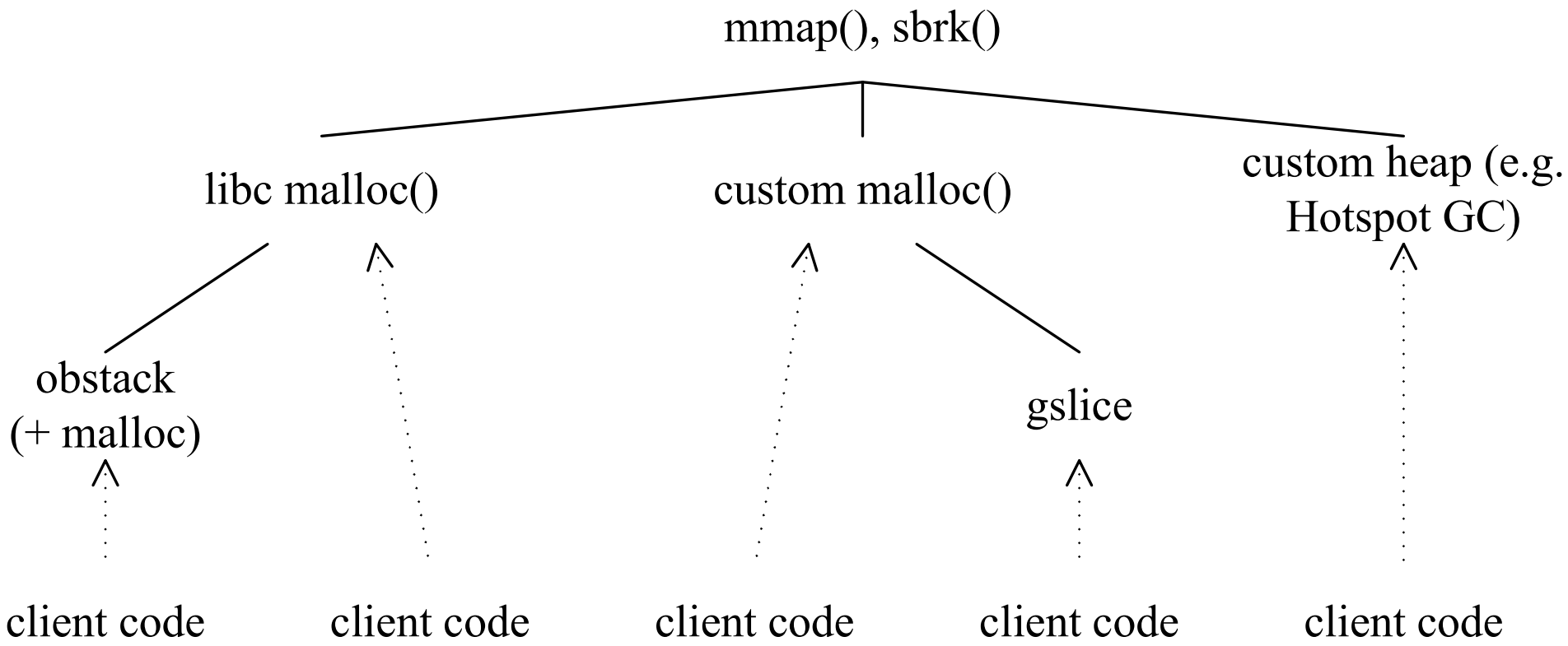
Non-difficulties




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

- structure “subtyping” via containment
- function pointers (most of the time)
- void pointers
- char pointers
- integer $\leftrightarrow$ pointer casts
- type-differing aliases
- custom allocators, memory pools etc.

Hierarchical model of allocations



Somewhat difficult cases

Solved:

- opaque types
- complex use of sizeof
- structure “subtyping” via prefixing

Give up:

- avoidance of sizeof
- address-taken union members
- non-procedurally abstracted object allocation/re-use

The remaining awkwards

- alloca
- unions
- varargs
- generic use of non-generic pointers (void**, . . .)
- casts of function pointers *to non-supertypes*

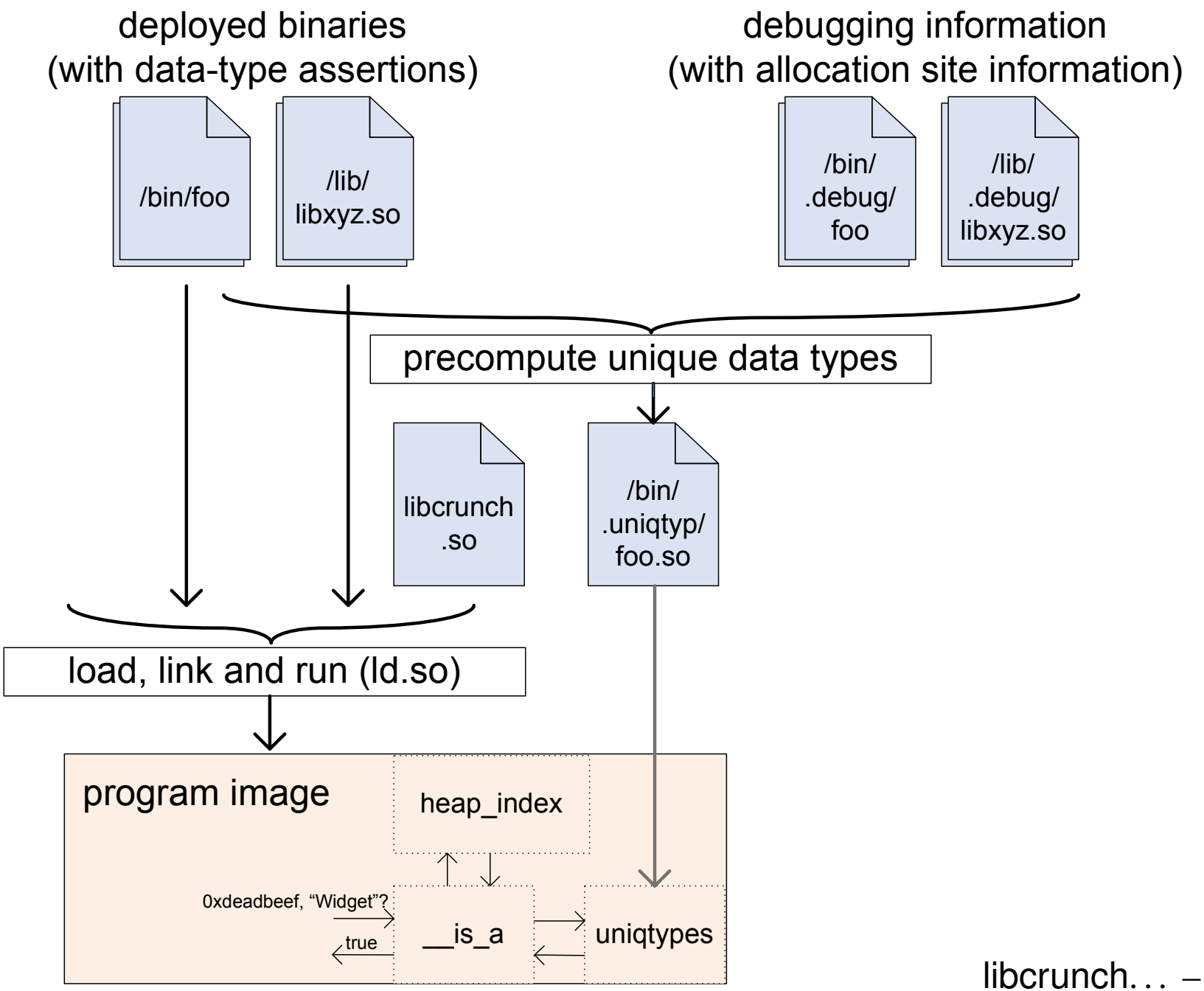
The remaining awkwards

- alloca
- unions
- varargs
- generic use of non-generic pointers (void**, . . .)
- casts of function pointers *to non-supertypes*

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

Idealised view of libcrunch toolchain



A model of data types: DWARF debugging info

```
$ cc -g -o hello hello.c && readelf -wi hello | column
 <b>:TAG_compile_unit                  <7ae>:TAG_pointer_type
      AT_language  : 1 (ANSI C)             AT_byte_size: 8
      AT_name      : hello.c                AT_type     : <0x2af>
      AT_low_pc    : 0x4004f4          <76c>:TAG_subprogram
      AT_high_pc   : 0x400514               AT_name     : main
 <c5>: TAG_base_type                        AT_type     : <0xc5>
       AT_byte_size : 4                     AT_low_pc   : 0x4004f4
       AT_encoding  : 5 (signed)            AT_high_pc  : 0x400514
       AT_name      : int               <791>: TAG_formal_parameter
 <2af>:TAG_pointer_type                      AT_name     : argc
       AT_byte_size: 8                       AT_type     : <0xc5>
       AT_type     : <0x2b5>                 AT_location : fbreg - 20
 <2b5>:TAG_base_type                   <79f>: TAG_formal_parameter
       AT_byte_size: 1                       AT_name     : argv
       AT_encoding : 6 (char)                AT_type     : <0x7ae>
       AT_name     : char                    AT_location : fbreg - 32
```

Type info for each allocation

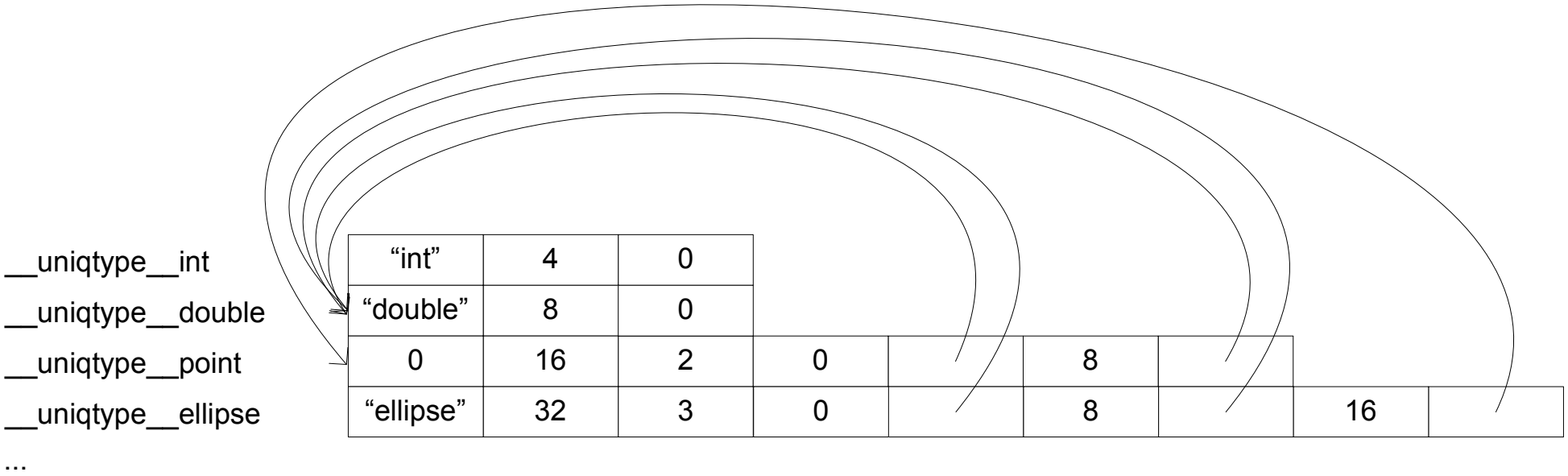
What is an allocation?

- static memory: mmap'd program binaries
- heap memory: "anonymous" mmappings
  - returned by malloc() – "level 1" allocation
  - returned by mmap() – "level 0" allocation
  - (maybe) memory issued by user allocators...
- stack memory

We keep specialised *indexes* for each kind of memory...

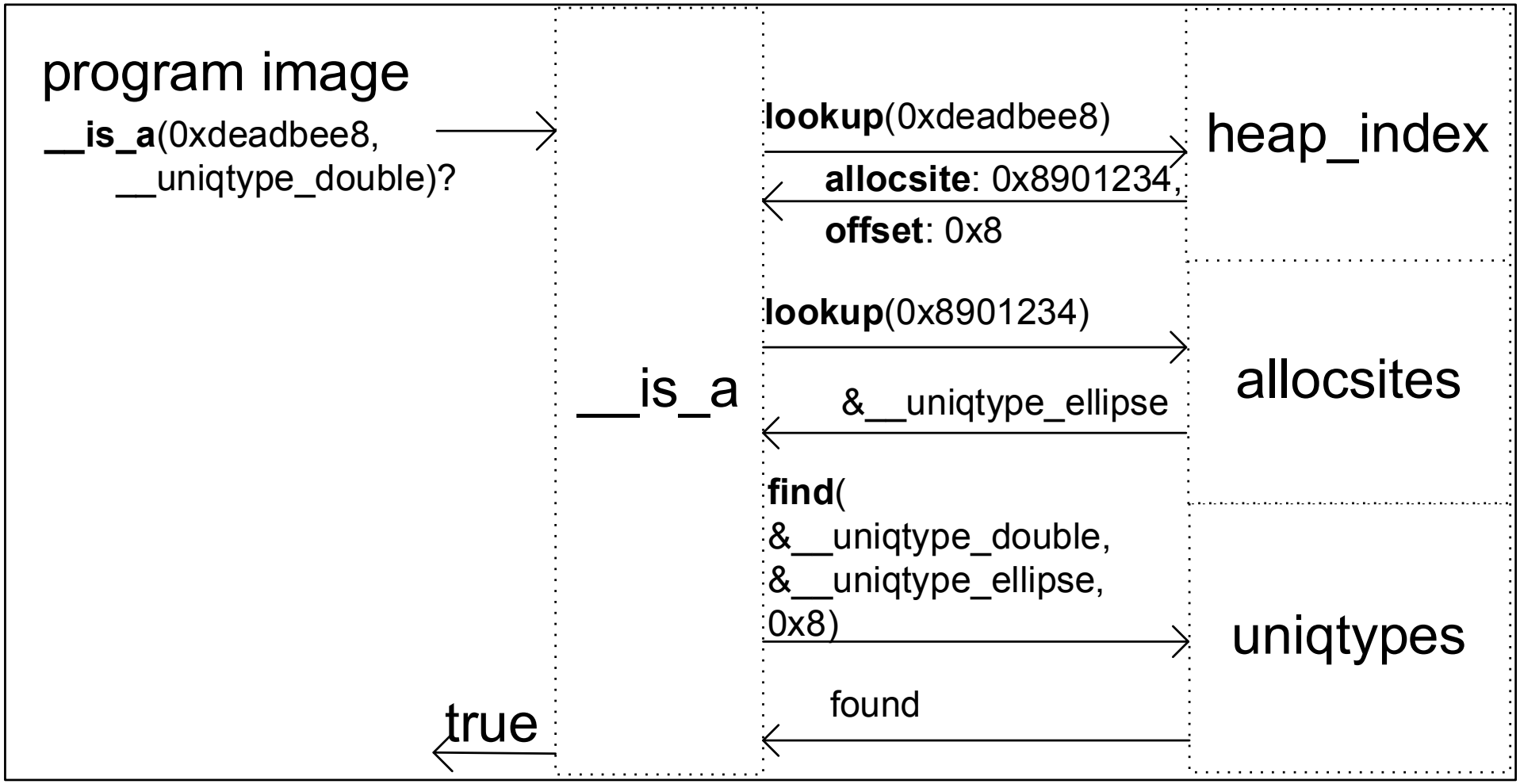
Representation of data types

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



- use the linker to keep them unique
- uniqueness → "exact type" test is a pointer comparison
- __is_a() is a short search

What happens at run time?



Getting from objects to their metadata

Recall: binary & source compatibility requirements

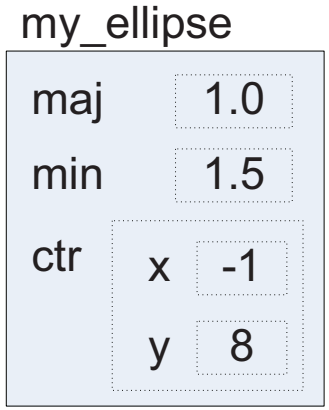
- can't embed metadata into objects
- can't change object layouts at all!
- $\rightarrow$ need out-of-band ("disjoint") metadata

Pointers can point anywhere inside an object

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

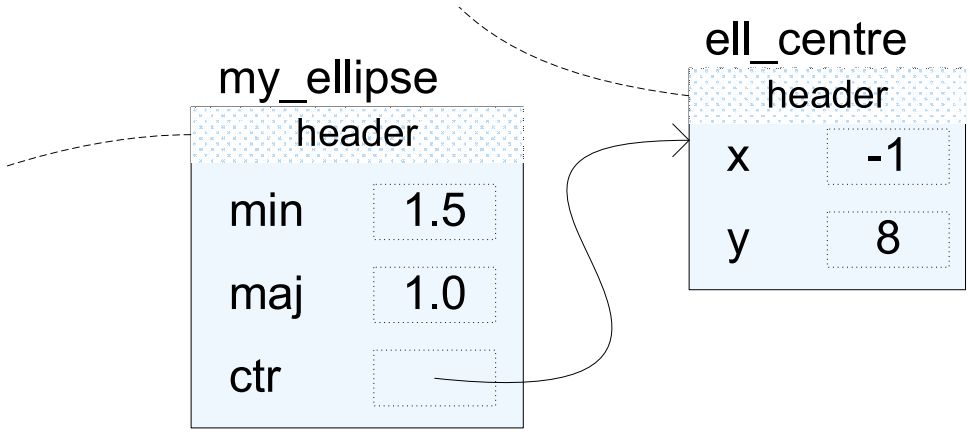
Why the heap case is difficult, cf. virtual machine heaps

Native objects are trees; no descriptive headers!




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

VM-style objects: “no interior pointers”



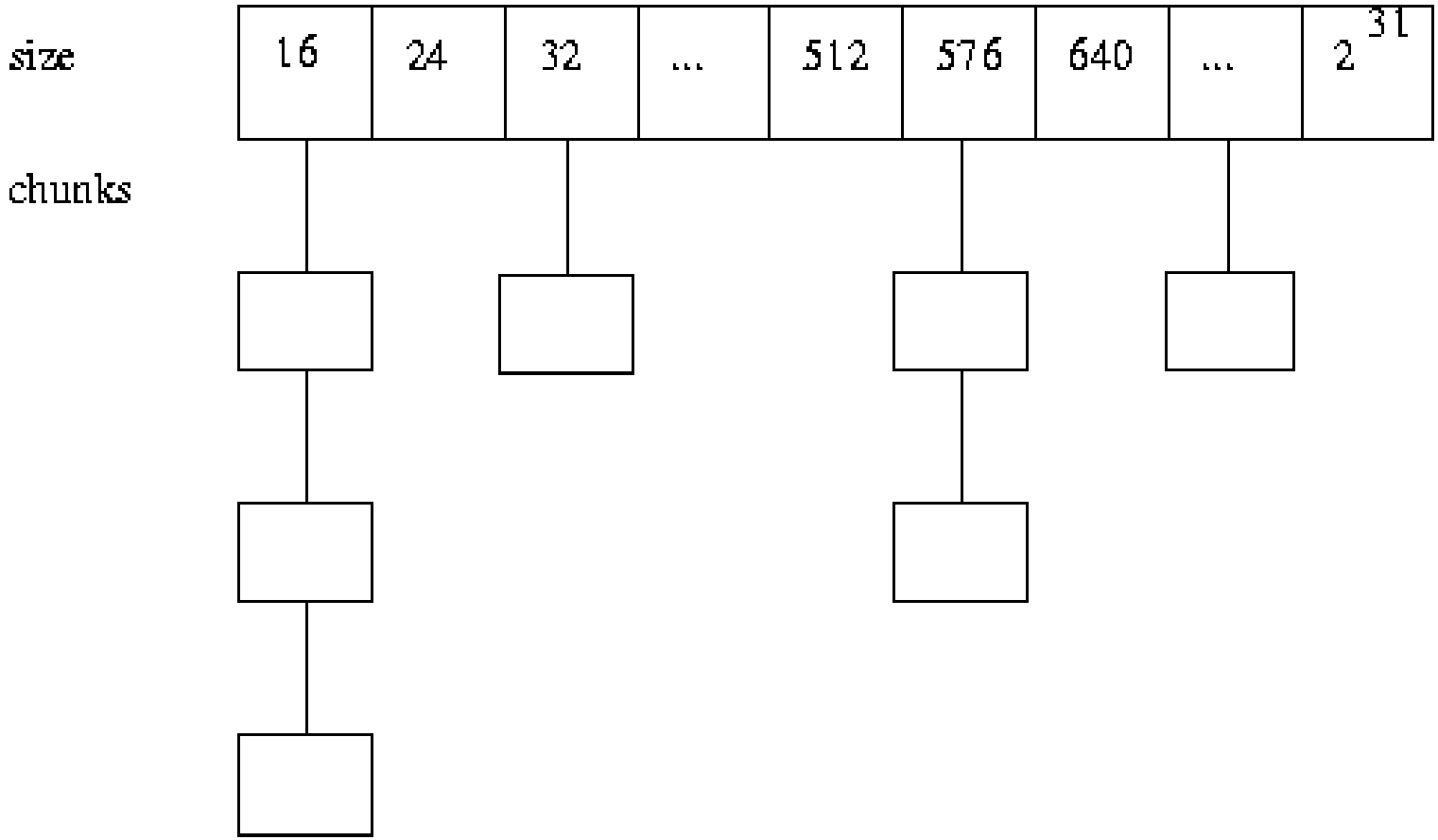
To solve the heap case...

- we'll need some 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

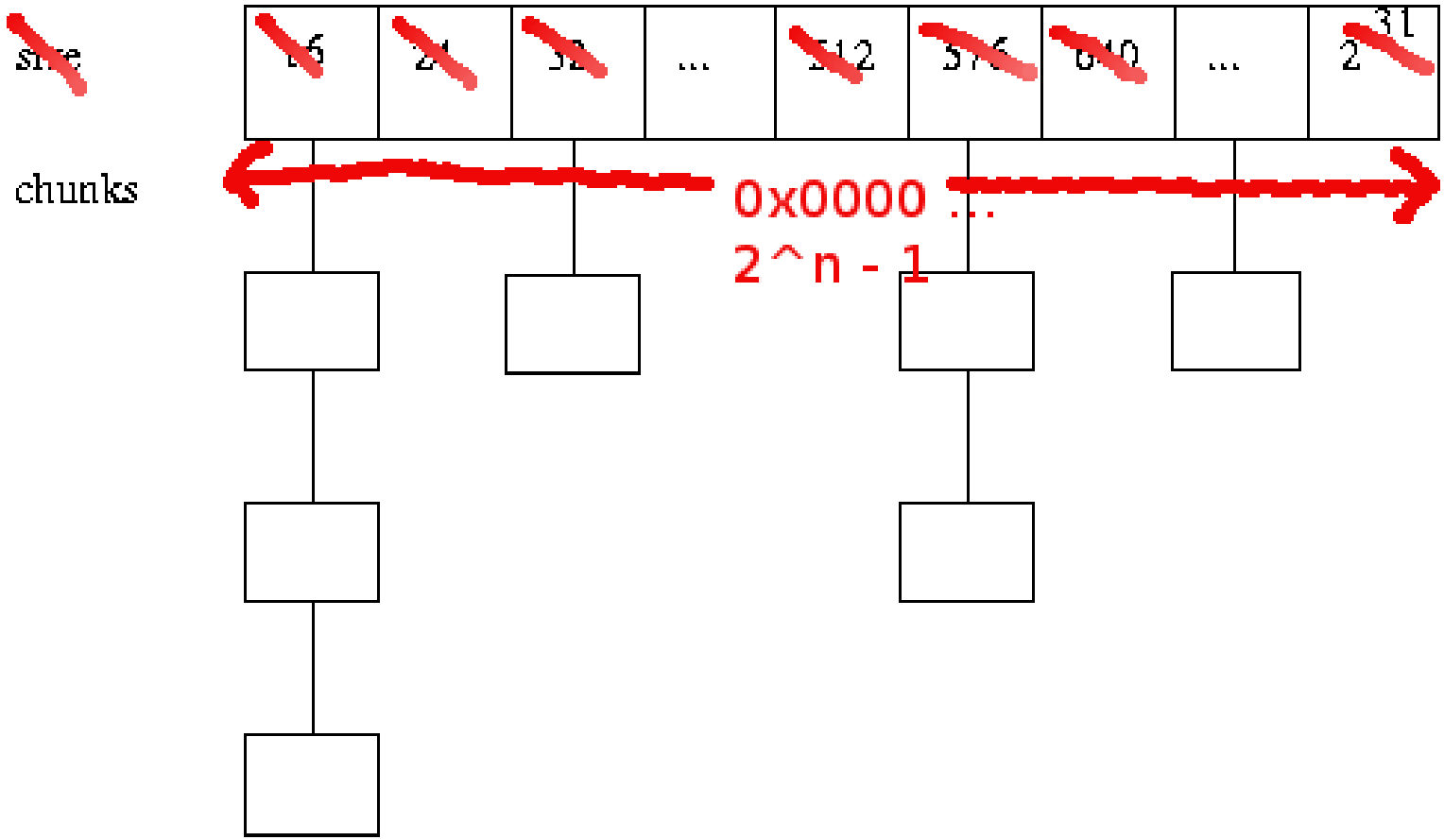
Indexing heap chunks

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

- $\rightarrow$ want bins to be as small as possible
- Q: how many bins can we have?
- A: lots... really, ***lots***!

Really, how big?

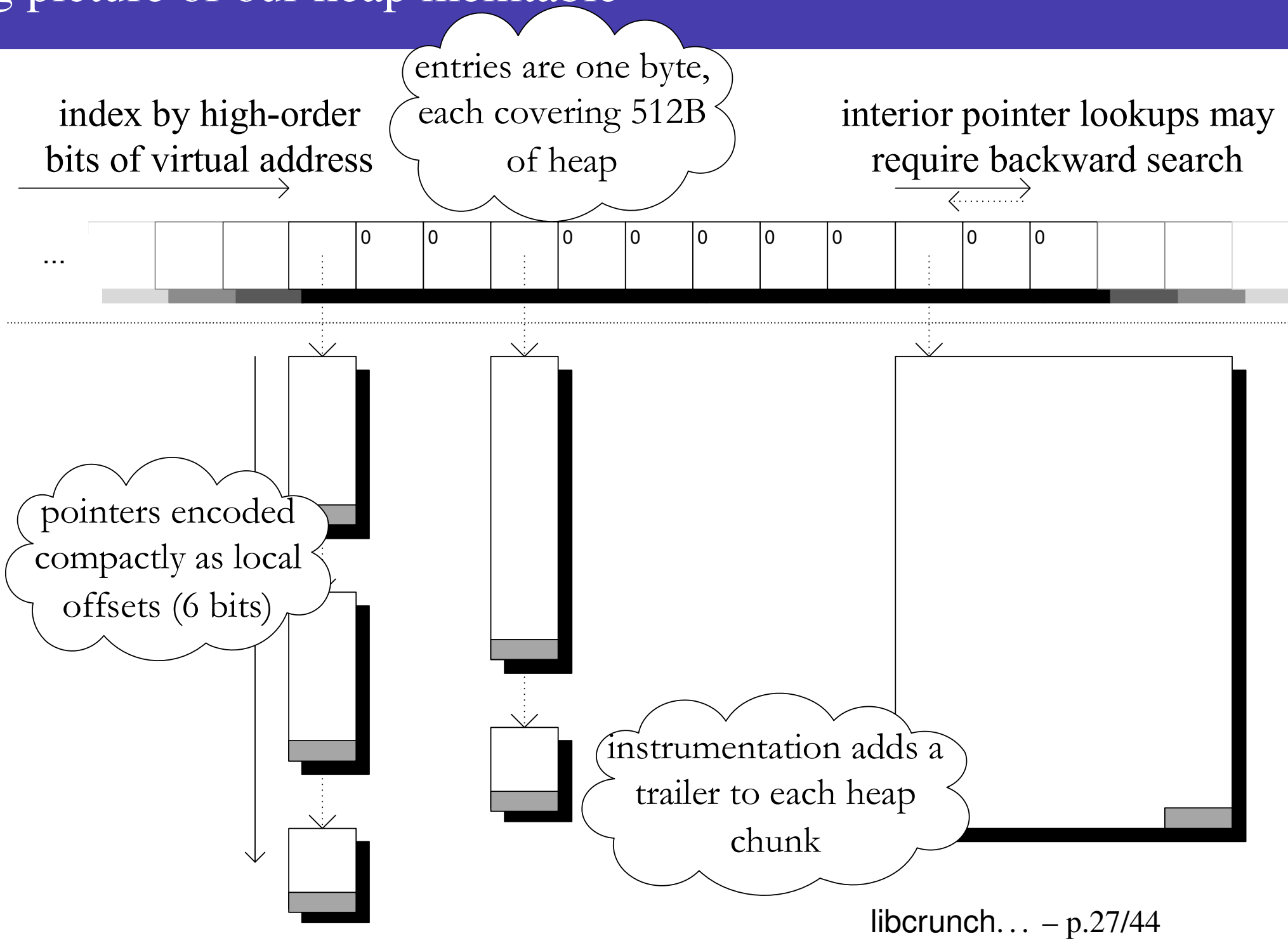
Bin index resembles a linear page table. Exploit

- 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

Big picture of our heap memtable



Indexing the heap with a memtable is. . .

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

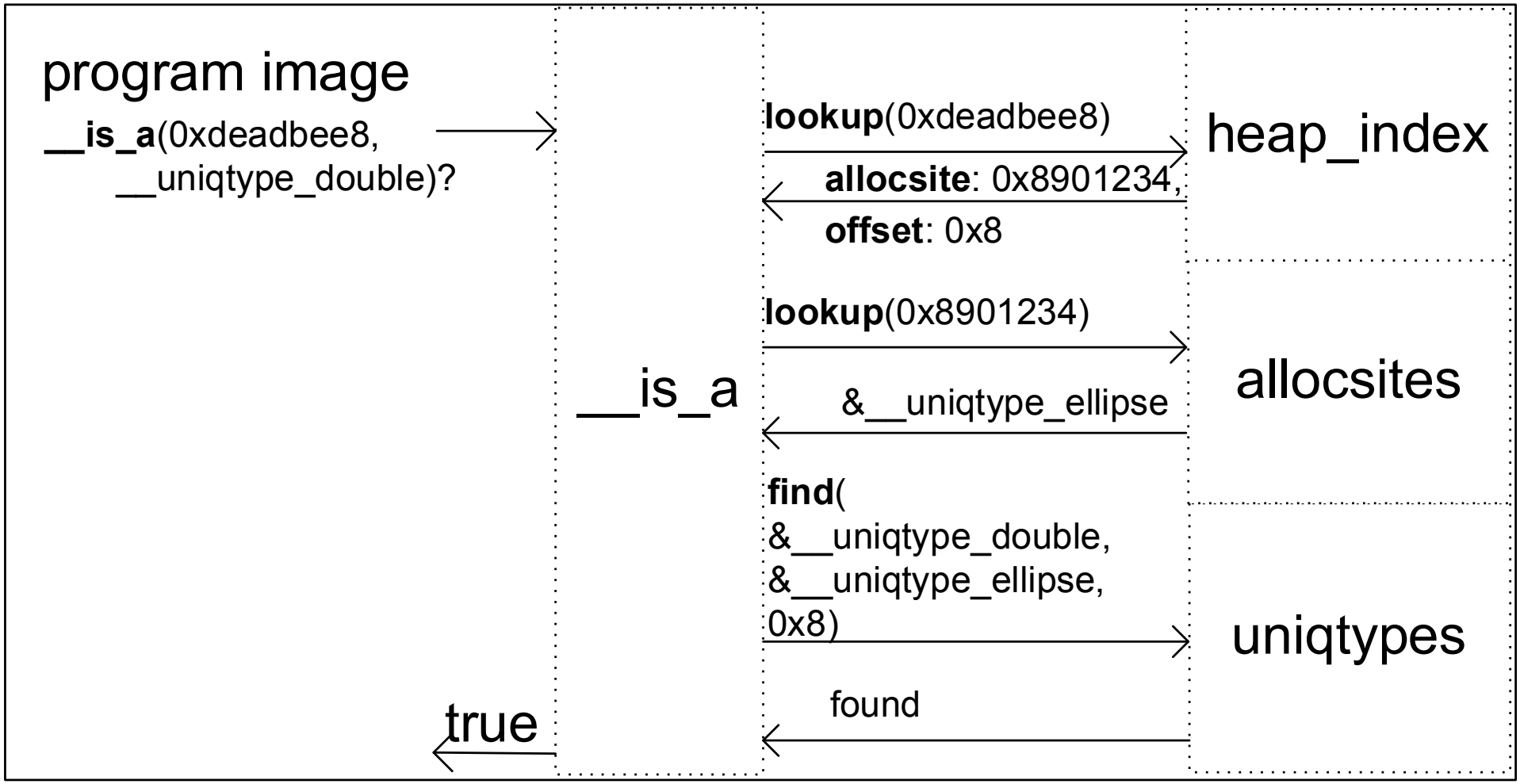
Memtables are versatile

- buckets don't have to be linked lists
- can tune size / coverage. . .

We also use memtables to

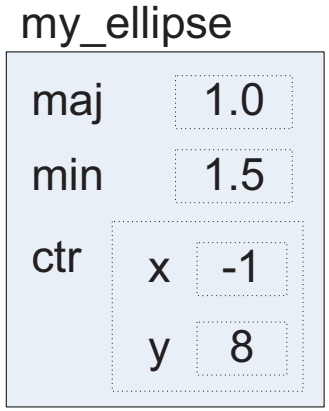
- index every mapped page in the process ("level 0")
- index "deep" (level 2+) allocations
- index static allocations
- index the stack (map PC to frame uniqtype)

Remind me: what happens at run time?



__is_a, containment...

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




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

Consider "what is"

- &my_ellipse
- &my_ellipse.ctr
- ...

(Subclassing is usually implemented this way.)

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!

Recap

What we've just seen is

- a runtime system for evaluating type checks
- fast
- flexible
- a "whole program" design
- language-neutral
- binary compatible

What about *source* compatibility?

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) x
LIBCRUNCH_LAZY_HEAP_TYPES="__PTR_void"
```

How fast is it? SPEC CPU2006 results

benchmark	normal/s	crunch	nopreload	just allocs
perlbench	1.48	+31 %	–	+3%
bzip2	5.05	+0 %	+0%	+0%
mcf	2.49	+6.8%	−1%	+0%
milc	8.75	+38 %	+2%	−1%
gobmk	14.5	+13 %	+1%	+1%
hmmer	2.13	+8.5%	+8%	+0%
sjeng	3.25	−2.2%	−2%	+0%
h264ref	10.0	+5 %	+5%	+1%
lbm	3.43	+24 %	+0%	+0%
sphinx3	1.58	+15 %	+2%	+4%
gcc	0.989	+289 %	–	+4%

Popular errors

- sloppiness about signed vs unsigned
- some user-level allocation behaviour
- some cases of multiple indirection of void

```
void get_obj(struct Foo **out);
void *opaque_obj;
get_obj(&opaque_obj);
```

False negatives:

- memory-incorrect programs
- unions
- over-coarse sloppification (e.g. __like_a)

More case studies needed...

Generic pointers to non-generic pointers

```
neighbor = (int **)calloc(NDIRS, sizeof(int *));
//  ...
sort_eight_special ((void **) neighbor );
// where
void sort_eight_special (void **pt){
void *tt [8];
register int i;
    for(i=0;i<8;i++)tt[i]=pt[i];
    for(i=XUP;i<=TUP;i++){pt[i]=tt[2*i]; pt[OPP_DIR(i)]=tt[2*i+1];}
}
```

Generic pointers to pointers to non-generic pointers

```
PUB_FUNC void dynarray_add(void ***ptab, int *nb_ptr, void *data)
{   /*  ...  */
    /*  every power of two we double array size */
    if ((nb & (nb - 1)) == 0) {
        if (!nb) nb_alloc = 1; else nb_alloc = nb * 2;
        pp = tcc_realloc (pp, nb_alloc * sizeof(void *));
        *ptab = pp;
    }
    /*  ...  */
}
char **libs = NULL;
/*  ...  */
dynarray_add((void ***) &libs, &nblibs, tcc_strdup(filename));
```

Zoo: data stuffing (1)

```
typedef double LBM_Grid[SIZE_Z*SIZE_Y*SIZE_X*N_CELL_ENTRIES];
typedef LBM_Grid* LBM_GridPtr;

#define MAGIC_CAST(v) ((unsigned int*) ((void*) (&(v))))
#define FLAG_VAR(v) unsigned int* const _aux_ = MAGIC_CAST(v)
//   ...
#define TEST_FLAG(g,x,y,z,f)    \
    ((*MAGIC_CAST(GRID_ENTRY(g, x, y, z, FLAGS))) & (f))
#define SET_FLAG(g,x,y,z,f)       \
{FLAG_VAR(GRID_ENTRY(g, x, y, z, FLAGS)); (*_aux_) |= (f);}
```

Zoo: data stuffing (2)

```
#define FUNC_CALL(r) (((AttributeDef*)&(r))->func_call)

typedef struct Sym {
    int v;      /* symbol token */
    long r;     /* associated register */
    long c;     /* associated number */
    CType type;     /* associated type */
    struct Sym *next; /* next related symbol */
    struct Sym *prev; /* prev symbol in stack */
    struct Sym *prev_tok; /* previous symbol for this token */
} Sym;

 func_attr_t *func_call = FUNC_CALL(sym->r);
```

Zoo: pointer stuffing (1)

```
typedef int parse_opt_cb(const struct option *,
        const char *arg, int unset);

static int stdin_cacheinfo_callback(struct parse_opt_ctx_t *ctx,
                                    const struct option *opt, int unset)
{ /* ... */ }

struct option options[] = {
        /* ... */,
        {OPTION_LOWLEVEL_CALLBACK, 0, /* ...*/,
                (parse_opt_cb *) stdin_cacheinfo_callback},
        /* ... */
};
```

Zoo: pointer stuffing (2)

```
if (value->kind > RTX_DOUBLE && value->un.addr.base != 0)
  switch (GET_CODE (value->un.addr.base))
    {
    case SYMBOL_REF:
      /* Use the string's address, not the SYMBOL_REF's address,
          for the sake of addresses of library routines. */
      value->un.addr.base = (rtx) XSTR (value->un.addr.base, 0);
      break;
    /* ... */
    }
```

Zoo: non-observable allocation bugs (1)

```
item->util = xcalloc(sizeof(struct branch_info), 1);
```

Zoo: non-observable allocation bugs (2)

```
if (((* array4D) = (short****)calloc(idx, sizeof(short**))) == NULL)
  no_mem_exit("get_mem4Dshort: array4D");
```

End

Code is here:

- https://github.com/stephenrkell/libcrunch

and also

- https://github.com/stephenrkell/libdwarfpp
- https://github.com/stephenrkell/dwarfidl
- https://github.com/stephenrkell/liballocs
- https://github.com/stephenrkell/libsrk31c++
- . . .
- will make a friendly download-and-build script soon

Questions?