Towards a dynamic object model within Unix processes

(or something like one)

Stephen Kell

stephen.kell@cl.cam.ac.uk

Computer Laboratory
University of Cambridge

Imagine if we could. . .

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

Imagine if we could. . .

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

CHECK this (at run time)

Imagine if we could. . .

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

CHECK this (at run time)

Also wanted:

- binary-compatible
- source-compatible
- reasonable performance
- design around a generic runtime (not C-specific)

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 to instrument the C code, in a nutshell

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

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

How to instrument the 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 to instrument the 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 the power to

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

What's missing from Unix process model

- query function: "what's on the end of this pointer?"
- type metadata

Recurring theme: *evolve* Unix to plug these gaps

- repurpose debugging information as type info
- extend metadata to allocation granularity
- provide fast whole-process query interface
- tweak dynamic loading, . . .

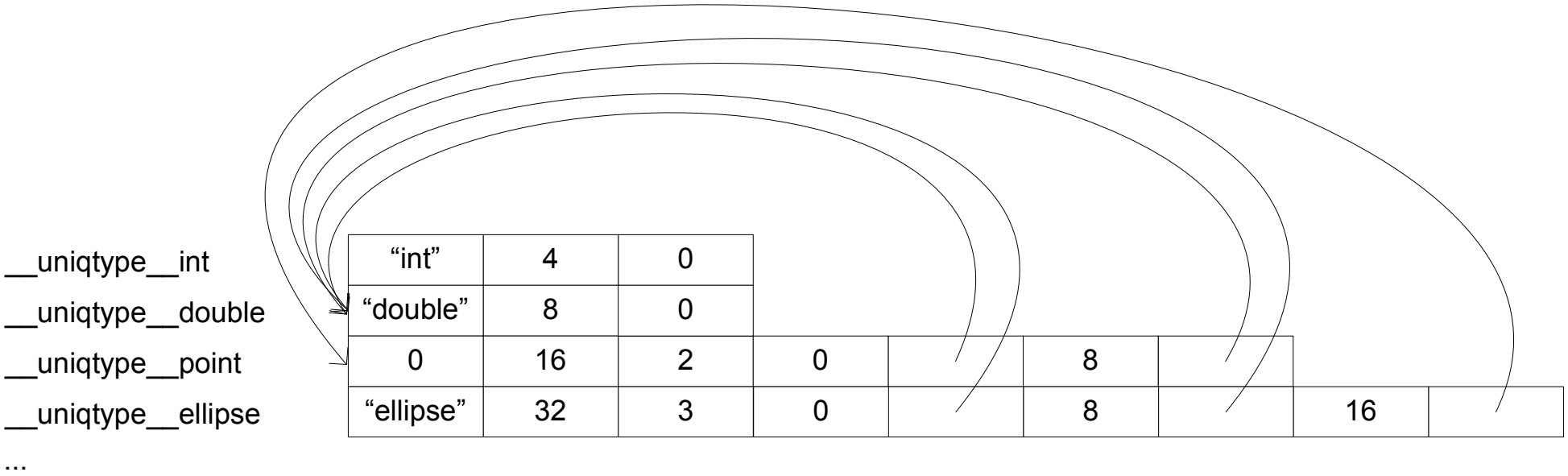
Don't implement a VM *on* Unix; make Unix *become* a VM!

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
```

Reifing data types at run time

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



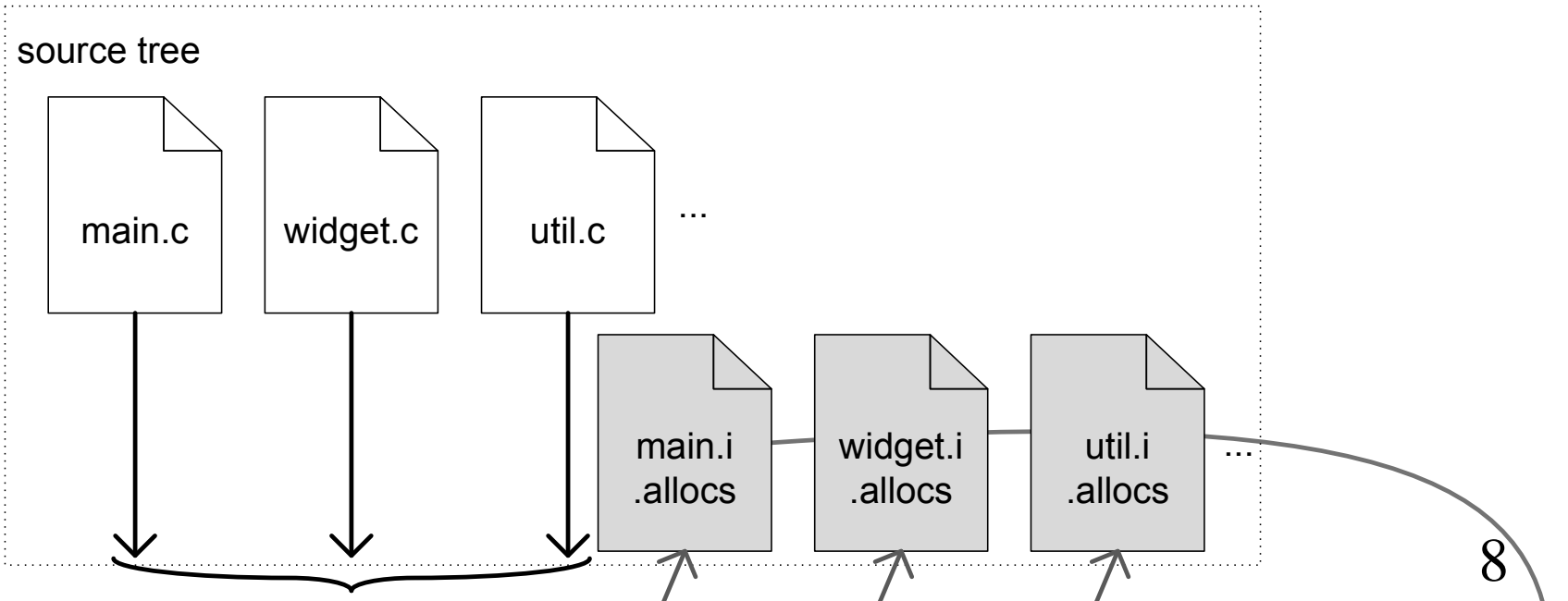
- use the linker to keep them unique
- → "exact type" test is a pointer comparison
- __is_a() is a short search

Dealing with C: 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 subties: e.g. malloc(sizeof (Blah) + n * sizeof (Foo))



Dealing with C: other challenges

- robustness to basic C idiom e.g. integer $\leftrightarrow$ pointer
- sloppy C idiom (`struct` prefixing, `sockaddr`-alikes…)
- polymorphic allocation sites (e.g. `sizeof (void*)`)
- subtler C features (function pointers, varargs, unions)
- understanding the invariant ("no bad pointers, *if*…")
- relating to C standard
- **support stack allocation** including `alloca()`
- **support custom heap allocators**
- **support nested heap allocators**

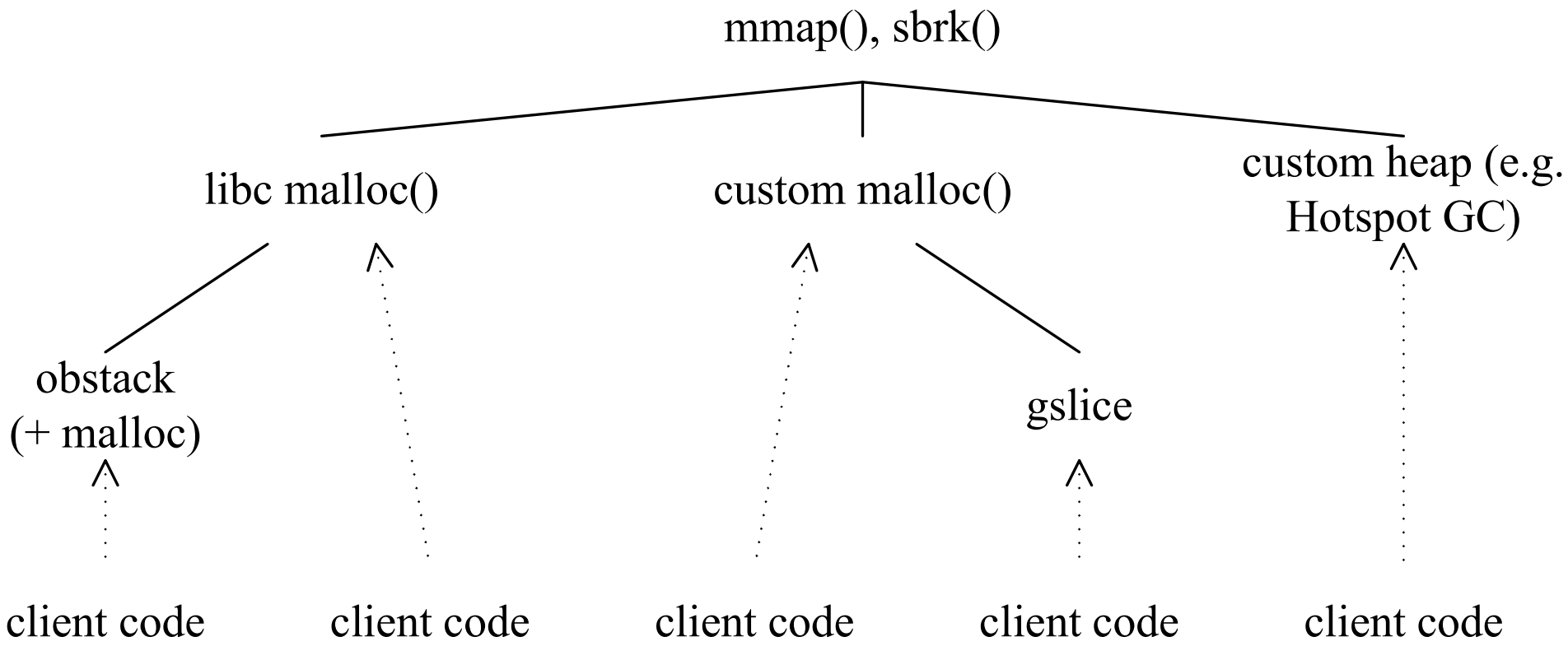
Type info for each allocation

All program memory is *allocated* somehow:

- "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



Telling libcrunch about allocation functions

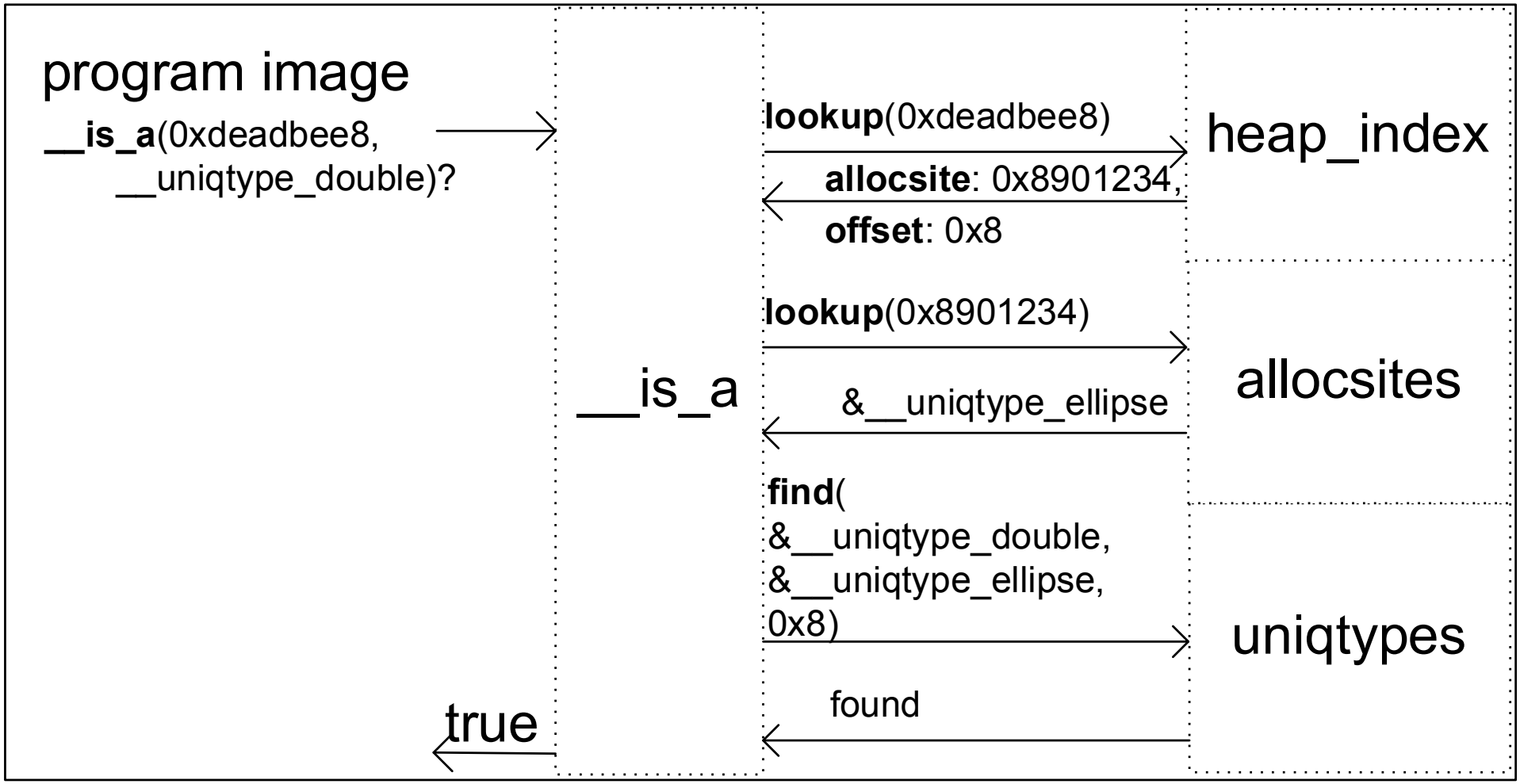
```
LIBALLOCS_ALLOC_FNS="xcalloc(zZ)p xmalloc(Z)p xrealloc(pZ)p"
LIBALLOCS_SUBALLOC_FNS="ggc_alloc(Z)p ggc_alloc_cleared(Z)p"
export LIBALLOCS_ALLOC_FNS
export LIBALLOCS_SUBALLOC_FNS
crunchcc -o myprog ... # instrument casts, postprocess metadata
```

or

```
allocscc -o myprog ... # just postprocess metadata
```

(liballocs $\subset$ libcrunch)

What happens at run time?



Getting from objects to their metadata

Recall: require binary & source compatibility

- can't embed metadata into objects
- can't change pointer representation
- $\rightarrow$ need out-of-band ("disjoint") metadata

Pointers can point anywhere inside an object

- which may be stack-, static- or heap-allocated
- ... or sub-allocated out of a heap chunk, etc..

To index the malloc() heap...

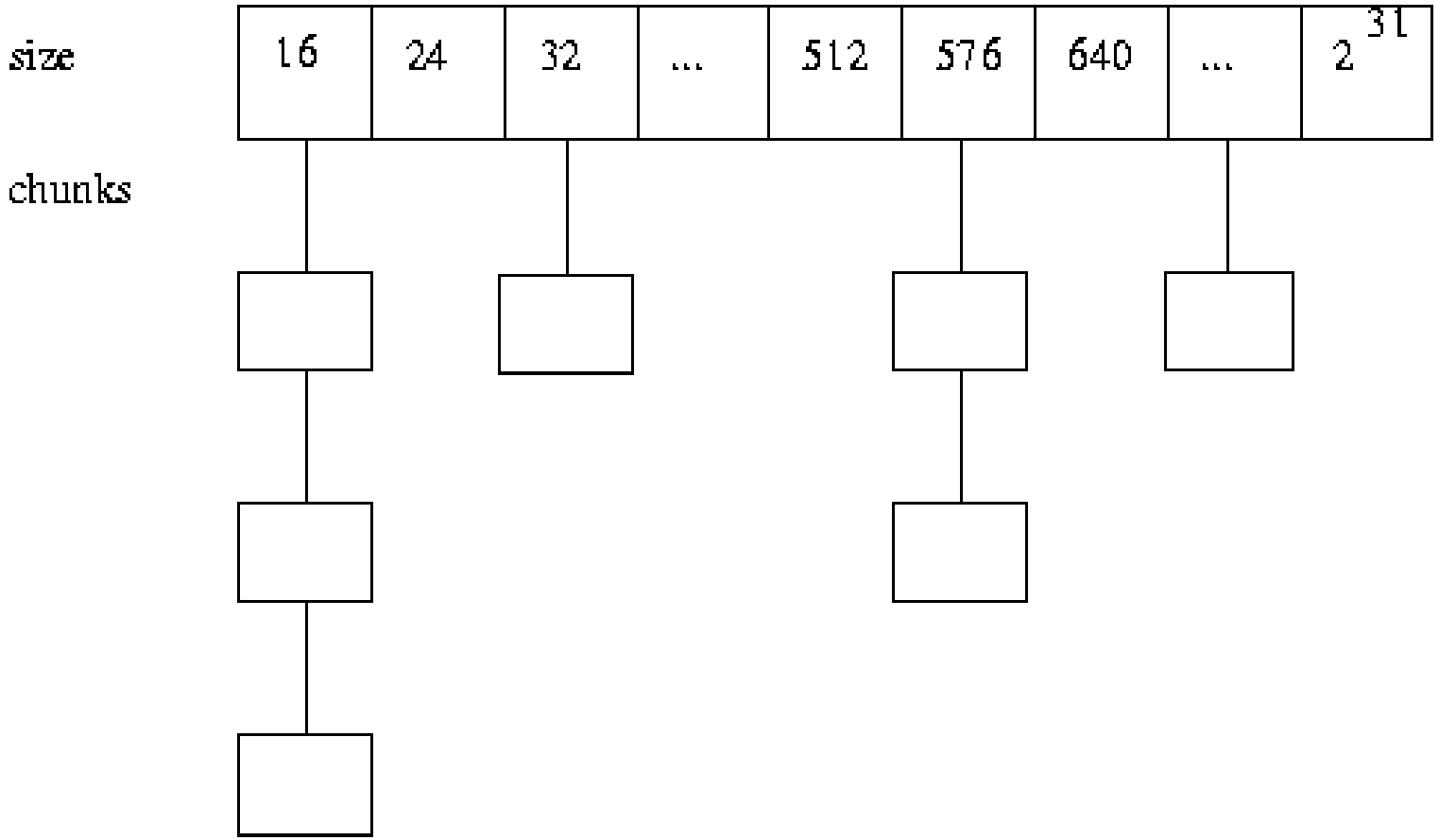
- we'll need some malloc() hooks...
- which keep an *index* of the heap
- in an efficient *address-keyed* associative map
  - must support (some) range queries
- map arbitrary pointer to allocation metadata

I call my structure "memtables"

- make aggressive use of virtual memory

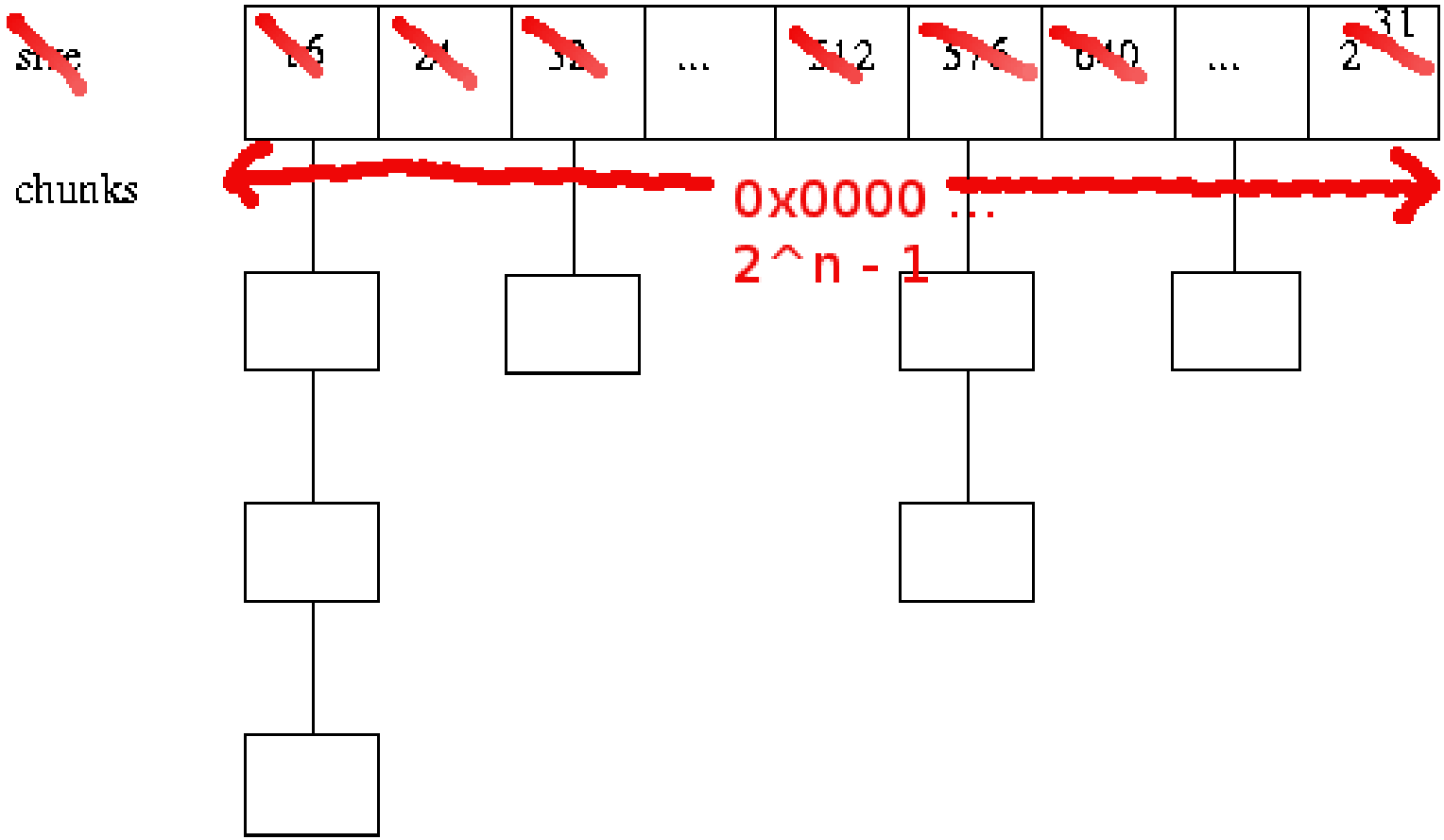
Indexing heap chunks

Inspired by free chunk binning in Doug Lea's malloc…



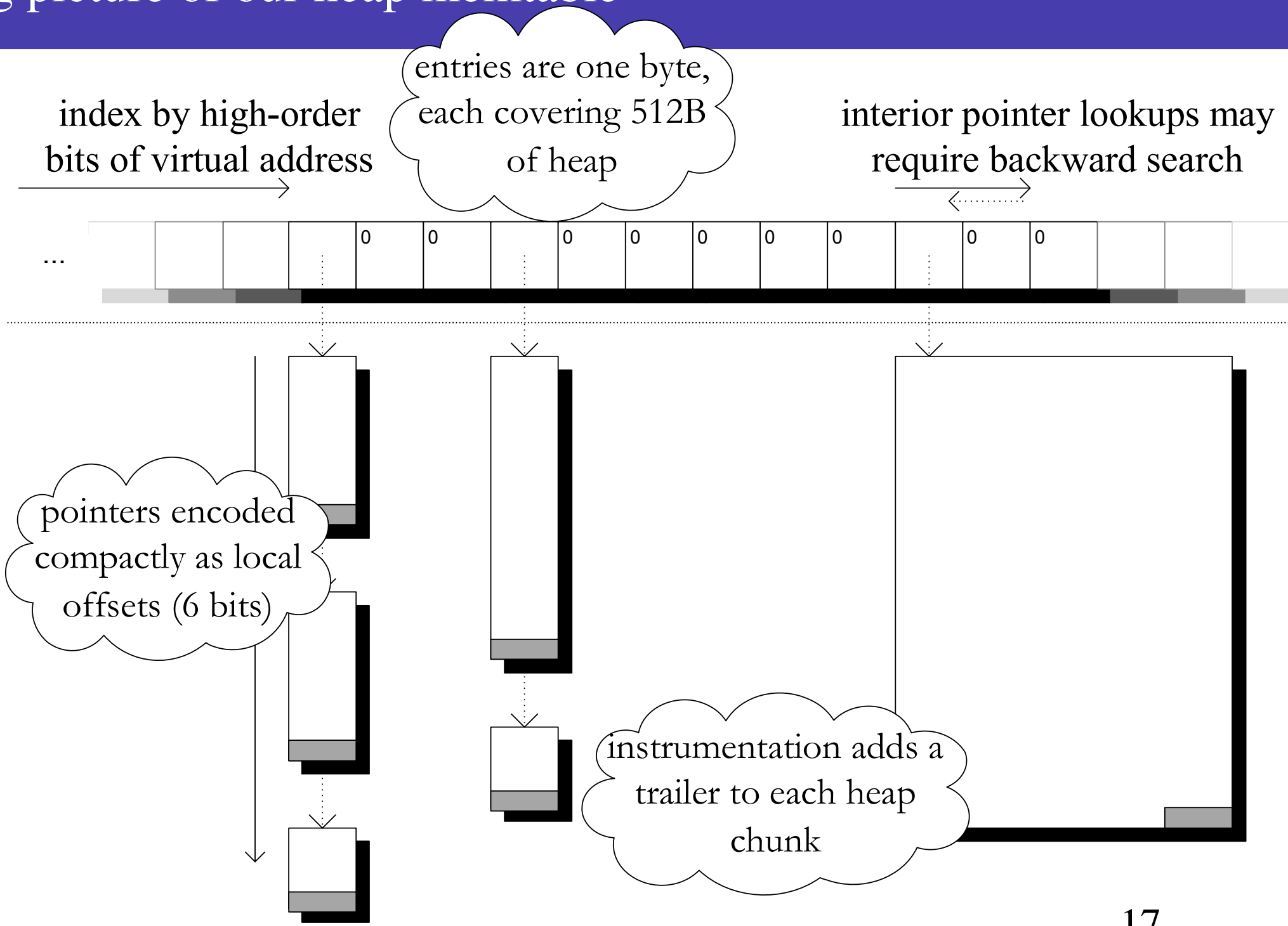
Indexing heap chunks

Inspired by free chunk binning in Doug Lea's malloc…



… but index *allocated* chunks binned by *address*

Big picture of our heap memtable



How many bins?

Lots! 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
- covering n-bit AS requires 2^{n-9} bins index

Many address-keyed maps in libcrunch use memtables

libcrunch vs C-language SPEC CPU2006 benchmarks

bench	normal/s	crunch %	nopreload	onlymeta
bzip2	4.95	+6.8%	+1.4%	+2.6%
gcc	0.983	+160 %	– %	+14.9%
gobmk	14.6	+11 %	+2.0%	+4.1%
h264ref	10.1	+3.9%	+2.9%	+0.9%
hmmer	2.16	+8.3%	+3.7%	+3.7%
lbm	3.42	+9.6%	+1.7%	+2.0%
mcf	2.48	+12 %	(−0.5%)	+3.6%
milc	8.78	+38 %	+5.4%	+0.5%
sjeng	3.33	+1.5%	(−1.3%)	+2.4%
sphinx3	1.60	+13 %	+0.0%	+8.7%
perlbench				

What else can we build with this runtime? (1)

Bounds checking

- more precise, *hopefully* faster (than ASan, SoftBound)
- in progress (but works! ask me)

"Typed" file I/O

- `mmap()`'d binary file $\approx$ heap
- (ask me)

Better native debugging

- dynamically query object on the end of `void*`

. . . i.e. adding introspection to native code. What else?

What else can we build with this runtime? (2)

Add *transparency* to VM-hosted code

- **“no FFI” interop**
- whole-process debugging & profiling
- whole-process precise garbage collection

Requires some in-VM cooperation…

A change in perspective

Current VMs:

- embedded within Unix(-like) process
- “owns” objects, services
- language (e.g. JS) is a self-contained “world”
- no metamodel shared with wider process

With liballocs:

- *federated* as *part of* Unix-like process
- deal with remote *or* external objects/services
- languages are *views* onto a universe
- shared metamodel

Software engineering wisdom

Target an interface, not an implementation.

Interfaces should hide change-prone details.

Don’t repeat yourself.

“Target an interface, not an implementation”

```
Local<Value> GetPointX(Local<String> property,
                       const AccessorInfo &info) {
  Local<Object> self = info.Holder();
  Local<External> wrap = Local<External>::Cast(self->GetInt
  void* ptr = wrap->Value();
  int value = static_cast<Point*>(ptr)->x_;
  return Integer::New(value);
}
void SetPointX(Local<String> property, Local<Value> value,
               const AccessorInfo& info) {
  Local<Object> self = info.Holder();
  Local<External> wrap = Local<External>::Cast(self->GetInt
  void* ptr = wrap->Value();
  static_cast<Point*>(ptr)->x_ = value->Int32Value();
}
```

Re: [v8-users] Re: Making v8::Persistent safe to use

Jun 21, 2013 1:34 AM

Posted in group: **v8-users**

On Fri , Jun 21, 2013 at 9:19 AM, Dan Carney <dca...@chromium.org> wrote:

> The transition from Local to Handle won't happen for a while. It's more of a cleanup step after everything else is done, and there's no urgency since there shouldn't be any performance impact.

The callback signature changes alone break almost every single line of v8-using code i've written (tens of thousands of them), and i am still

“Don’t repeat yourself”

```
var ffi = require("node-ffi");

var libm = new ffi.Library("libm", { "ceil": [ "double", [ "double" ] ] });
libm.ceil(1.5); // 2

// You can also access just functions in the current process
var current = new ffi.Library(null, { "atoi": [ "int32", [ "string" ] ] });
current.atoi("1234"); // 1234
```

Why not just. . .

```
libm.ceil(1.5)          // 2
libc.atoi("1234");      // 1234
```

. . .?

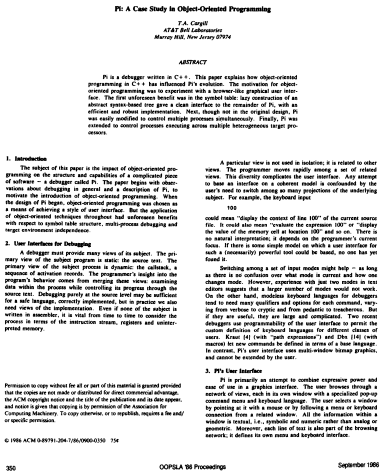
That Unix dynamism

```
// "forward" lookup, by name
double(*p_ceil)(double)
 = dlsym(RTLD_DEFAULT, "ceil");

// "reverse" lookup, by address
Dl_info i;
dladdr(p_ceil, &i);
printf("%s\n", i.dli_sname); // "ceil"

// ... but only for "static" objects!
```

The real difference between Unix and VMs



Pi: A Case Study in Object-Oriented Programming

T.A. Cargill
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Pi is a debugger written in C++. This paper explains how object-oriented programming in C++ has influenced Pi's evolution. The motivation for object-oriented programming was to cooperate with a browser-like graphical user interface. The first enthusiasm benefited users with top-level tasks: easy construction of an abstract system-based upon a clean interface to the remainder of Pi, with an efficient and robust implementation. Next, though not in the original design, Pi was easily modified to control multiple processes simultaneously. Finally, Pi was extended to control processes executing across multiple heterogeneous target processors.

1. Introduction

The subject of this paper is the impact of object-oriented programming on the structure and capabilities of a complicated piece of software: a debugger called Pi. The paper begins with observations about debugging in general and a description of Pi, to motivate the introduction of object-oriented programming. When the design of Pi began, object-oriented programming was chosen as a means of achieving a style of user interface. But the application of object-oriented techniques throughout had unforeseen benefits with respect to symbol table structure, multi-process debugging and target environment independence.

2. User Interfaces for Debugging

A debugger must provide many views of its subject. The primary view of the subject program is static: the source text. The primary view of the subject process is dynamic: the callstack, a sequence of activation records. The programmer's insight into the program's behavior comes from merging these views: examining data within the process while controlling its progress through the source text. Debugging purely at the source level may be sufficient for a safe language, correctly implemented, but in practice we also need views of the implementation. Even if none of the subject is written in assembler, it is vital from time to time to consider the process in terms of the instruction stream, registers and uninterpreted memory.

A particular view is not used in isolation; it is related to other views. The programmer moves rapidly among a set of related views. This diversity complicates the user interface. Any attempt to base an interface on a coherent model is confounded by the user's need to switch among as many projections of the underlying subject. For example, the keyboard input

100

could mean "display the context of line 100" of the current source file. It could also mean "evaluate the expression 100" or "display the value of the memory cell at hexagon 100" and so on. There is no natural interpretation; it depends on the programmer's current focus. If there is some simple model on which a user interface for such a (necessarily) powerful tool could be based, no one has yet found it.

Switching among a set of input modes might help – as long as there is no confusion over what mode is current and how one changes mode. However, experience with just two modes in text editors suggests that a larger number of modes would not work. On the other hand, modeless keyboard languages for debuggers tend to need many qualifiers and options for each command, varying from verbose to cryptic and from pedantic to treacherous. But if they are useful, they are large and complicated. Two recent debuggers use programmability of the user interface to permit the custom definition of keyboard languages for different classes of users. Kraut [4] (with "path expressions") and Dbx [14] (with macros) let new commands be defined in terms of a base language. In contrast, Pi's user interface uses multi-window bitmap graphics, and cannot be extended by the user.

3. Pi's User Interface

Pi is primarily an attempt to combine expressive power and ease of use in a graphics interface. The user browses through a network of views, each in its own window with a specialized pop-up command menu and keyboard language. The user selects a window by pointing at it with a mouse or by following a menu or keyboard connection from a related window. All the information within a window is textual, i.e., symbolic and numeric rather than analog or geometric. Moreover, each line of text is also part of the browsing network; it defines its own menu and keyboard interface.

In Smalltalk's "integrated environment"… there is little distinction between the compiler, interpreter, browser and debugger, [all of which] cooperate through shared data structures.… Pi is an isolated tool in a [Unix] "toolkit environment" [and] interacts with graphics, external data and other processes through *explicit interfaces*.

T.A. Cargill
Pi: a case study in object-oriented programming
OOPSLA '86

A simple meta-level allocator protocol

```
struct uniqtype;                                        /* type descriptor */
struct allocator;                                       /* reified allocator */
uniqtype *   alloc_get_type       (void *obj);          /* what type? */
allocator *  alloc_get_allocator  (void *obj);          /* heap/stack? etc */
void *       alloc_get_site       (void *obj);          /* where allocated? */
void *       alloc_get_base       (void *obj);          /* base address? */
void *       alloc_get_limit      (void *obj);          /* end address? */
Dl_info      alloc_dladdr         (void *obj);          /* dladdr-like */
```

An object model, but not as we know it:

- (ideally) implemented across whole process
- embrace *plurality* (many heaps)
- embrace *diversity* (native, VMs, . . .)

Introducing node-banoffi

All native objects appear under JS process object

- dynamically load, explore, experiment
- no need to repeat interface definitions

Grand challenge

- recover “JavaScript-y view” of arbitrary code
- treat problem as *API mapping*, not glue coding

Smaller challenge

- make it do something sensible in common cases

A (slightly) more complex example

X11 toolkit defines:

```
Widget XtInitialize (String shell_name, String application_class,
    XrmOptionDescRec* options, Cardinal num_options,
    int* argc, char**argv);
```

. . . so we want to be able to do this:

```
process.lm.dlopen("/usr/local/lib/libXt.so.6", 257)
var toplvl = process.lm. XtInitialize (
    process.argv[0], "simple", null , 0,
    [process.argv.length], process.argv
);
```

Key mechanism: externalize arbitrary JS objects

V8 hackery (1)

- generic object template for “native proxy”
- two internal fields: data ptr, type ptr
- implement JS callbacks via liballocs metaprotocol

To externalize an arbitrary object as uniqtype T:

- allocate a T in malloc() heap
- copy properties by name
- detect & handle arrays
- leave behind a native proxy in JS heap
- move is one-way (for now)
- move all reachable objects too! (maybe?)

V8 hackery (2)

Enforce 5-word minimum object size

- proxy = 3-word JSObject + 2 internal fields
- $\rightarrow$ can reinitialize any JS object as native proxy

Many limitations remaining:

- still a bit crashy!
- JS semantics unclear
- GC semantics not good
- want "externalize on demand", not transitive move
- can't externalize code (e.g. to make C function pointers)
- want to unify Maps with uniqtypes somehow

How it *should* work (perhaps?)

Property access via inline cache, currently:

```
cmp [ebx,<class offset>],<cached class>; test
jne <inline cache miss>                ; miss? bail
mov eax,[ebx, <cached x offset>]       ; hit; do load
```

Same but "allocator-guarded" + slow/general path:

```
xor ebx,<allocator mask>               ; get allocator
cmp ebx,<cached allocator prefix>      ; test
jne <allocator miss>                   ; miss? bail
cmp [ebx,<class offset>],<cached class>; test class
jne <cached cache miss>                ; miss? bail
mov eax,[ebx, <cached x offset>]       ; hit! do load
```

Slow path goes via liballocs metaprotocol

Conclusions

- I built a thing to check pointer casts
- really implemented a whole-process object model
- seeking a world without FFIs, with cross-VM tools, . . .

Want to *retrofit* VMs like V8 onto liballocs

- making some early progress
- how *should* it work?

Code is here: https://github.com/stephenrkell

- + lots of related stuff I didn't mention. . .

Thanks for your attention. Questions?

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

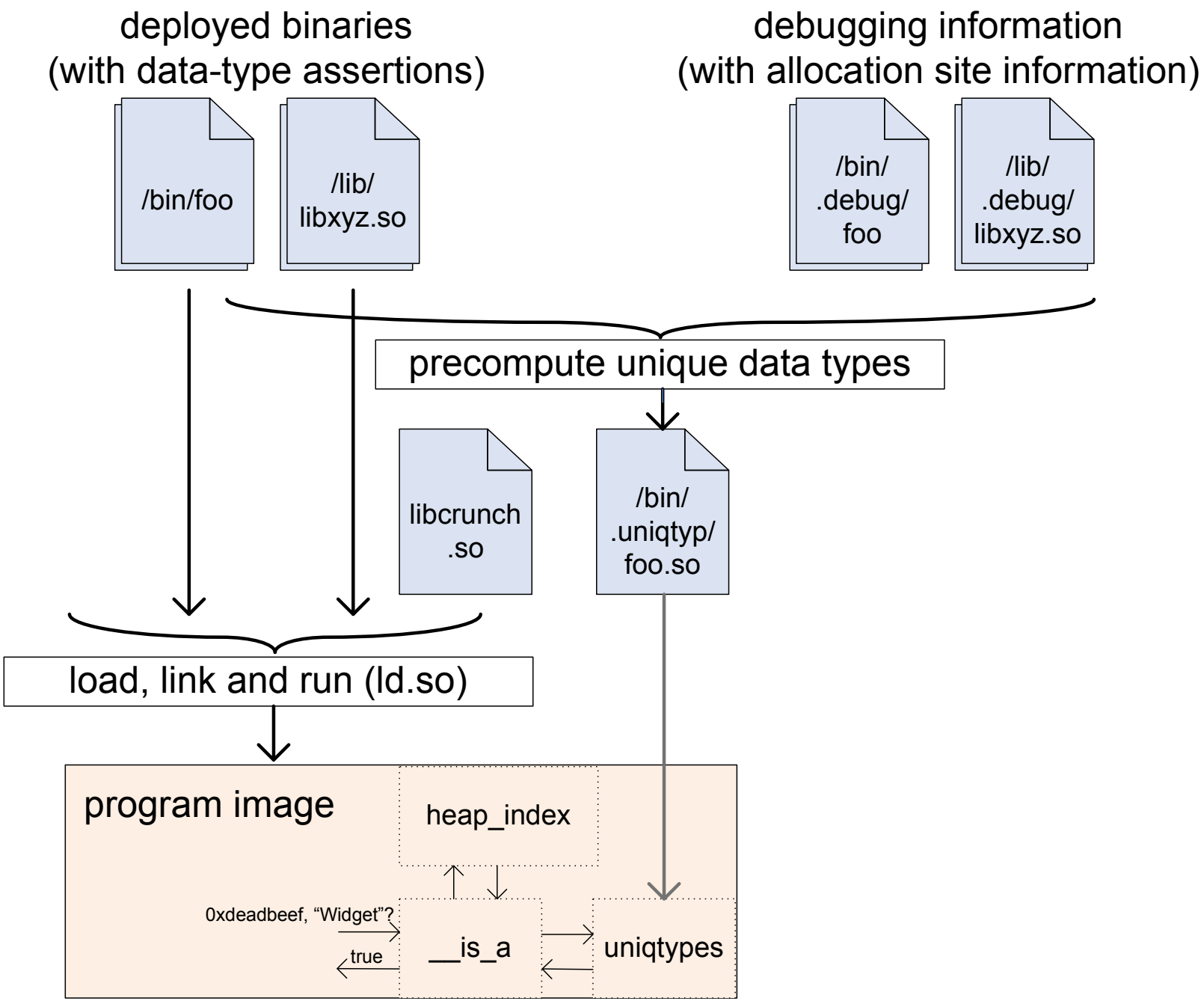
- `$ crunchcc -o myprog ...   # + other front-ends`
- `$ ./myprog                          # runs normally`
- `$ LD_PRELOAD=libcrunch.so ./myprog # does checks`
- `myprog:  Failed __is_a_internal(0x5a1220, 0x413560 a.k.a.  "uint$32") at 0x40dade, allocation was a heap block of int$32 originating at 0x40daa1`

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`

```
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]); // use SoftBound
return &z;                 // use CETS
```

Idealised view of libcrunch toolchain



A small departure from standard C

6 The *effective type* of an object for an access to its stored value is the declared type of the object, if any.[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 `e c` or `e ove`, 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.

A small departure from standard C

6 The *effective type* of an object for an access to its stored value is the declared type of the object, if any.[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 `e c` or `e ove`, 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 ≤ 1 effective type
- stack, locals / actuals: use declared types
- heap, alloca(): use *allocation site* (+ finesse)
- trap memcpy() and reassign type

Plenty of existing tools do bounds checking

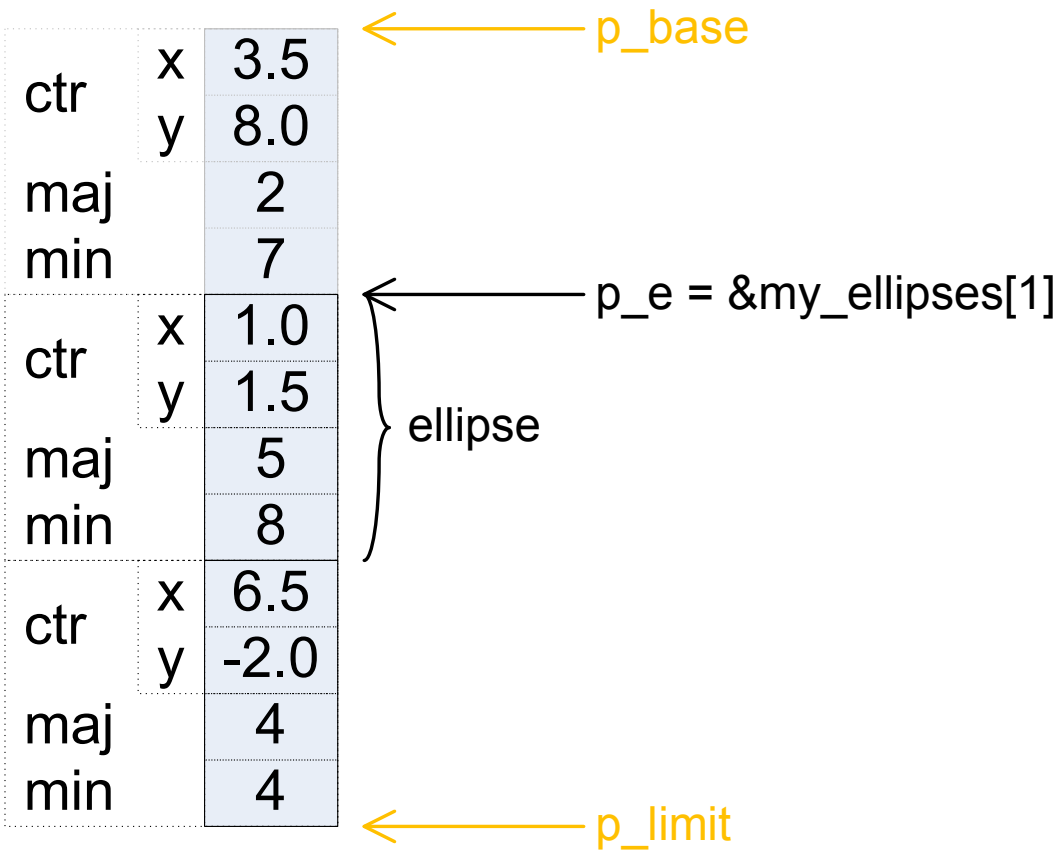
Memcheck (coarse), ASan (fine-ish), SoftBound (fine) . . .

- detect out-of-bounds pointer/array use
- first two also catch some temporal errors
- can run under libcrunch and [then] . . .

Problems remaining:

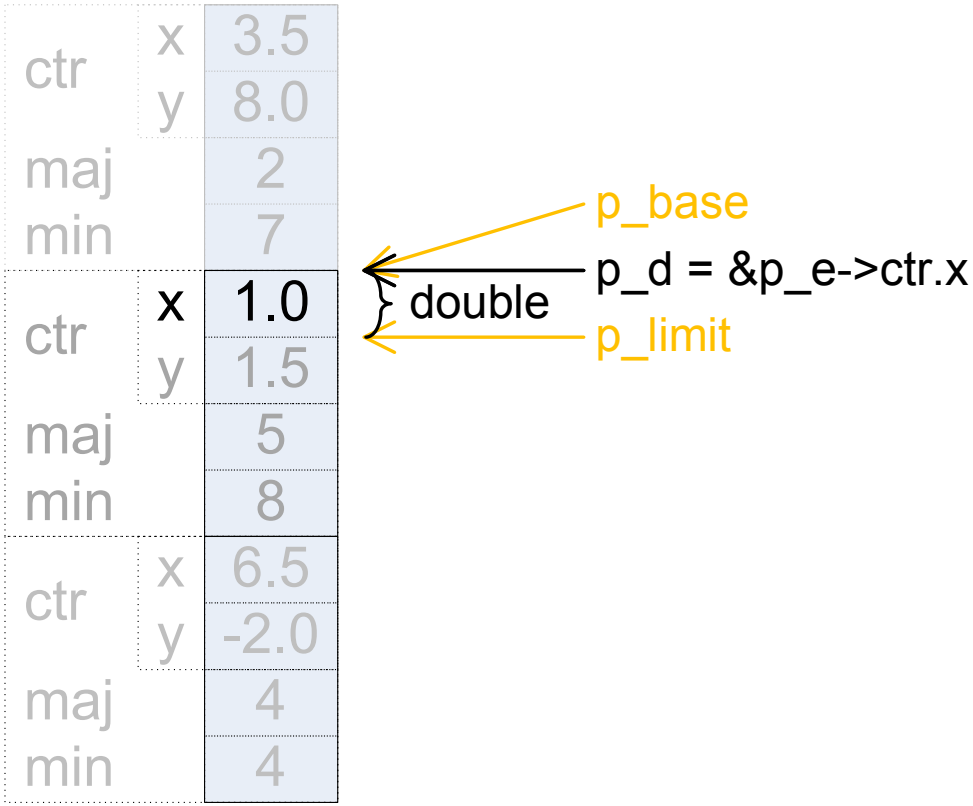
- overhead at best 50–100% (ASan & SoftBound)
- problems mixing uninstrumented code (libraries)
- *false positives for some idiomatic code!*

Existing bounds checkers use per-pointer metadata



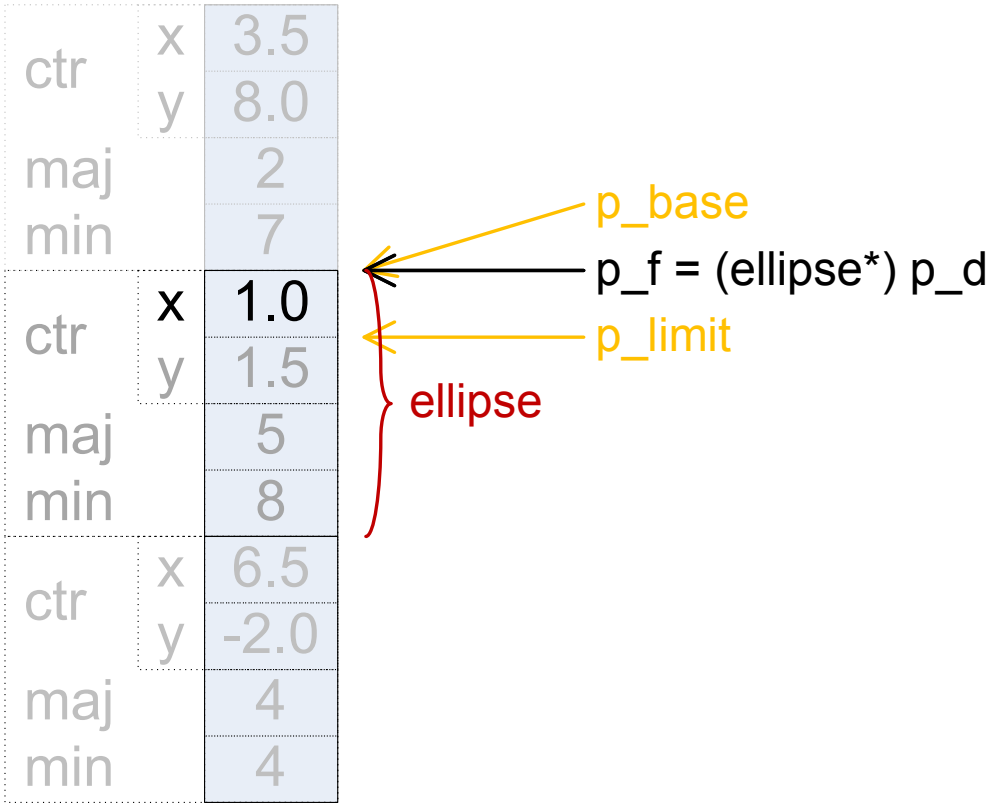

```
struct ellipse {
  struct point {
    double x, y;
  } ctr;
  double maj;
  double min;
} my_ellipses[3];
```

Existing bounds checkers use per-pointer metadata



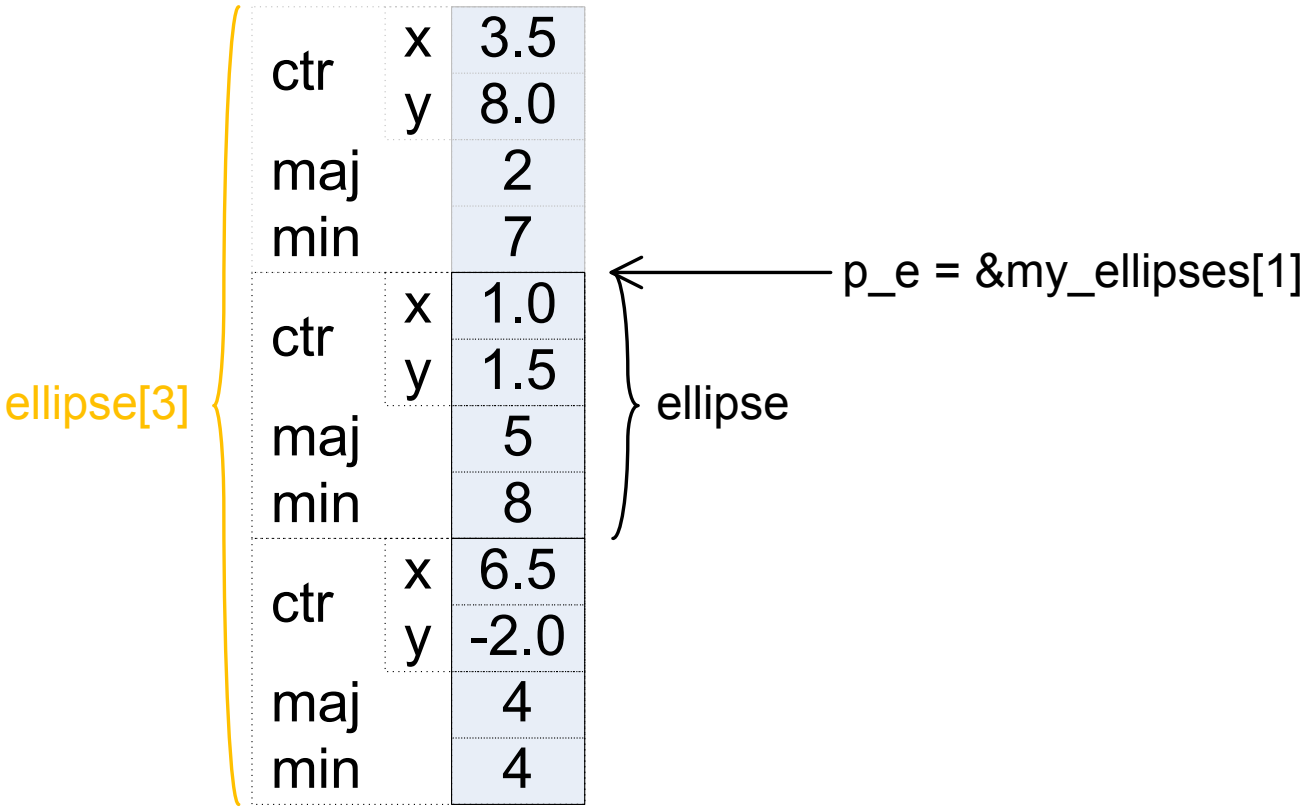

```
struct ellipse {
  struct point {
    double x, y;
  } ctr;
  double maj;
  double min;
} my_ellipses[3];
```

Without type information, pointer bounds lose precision



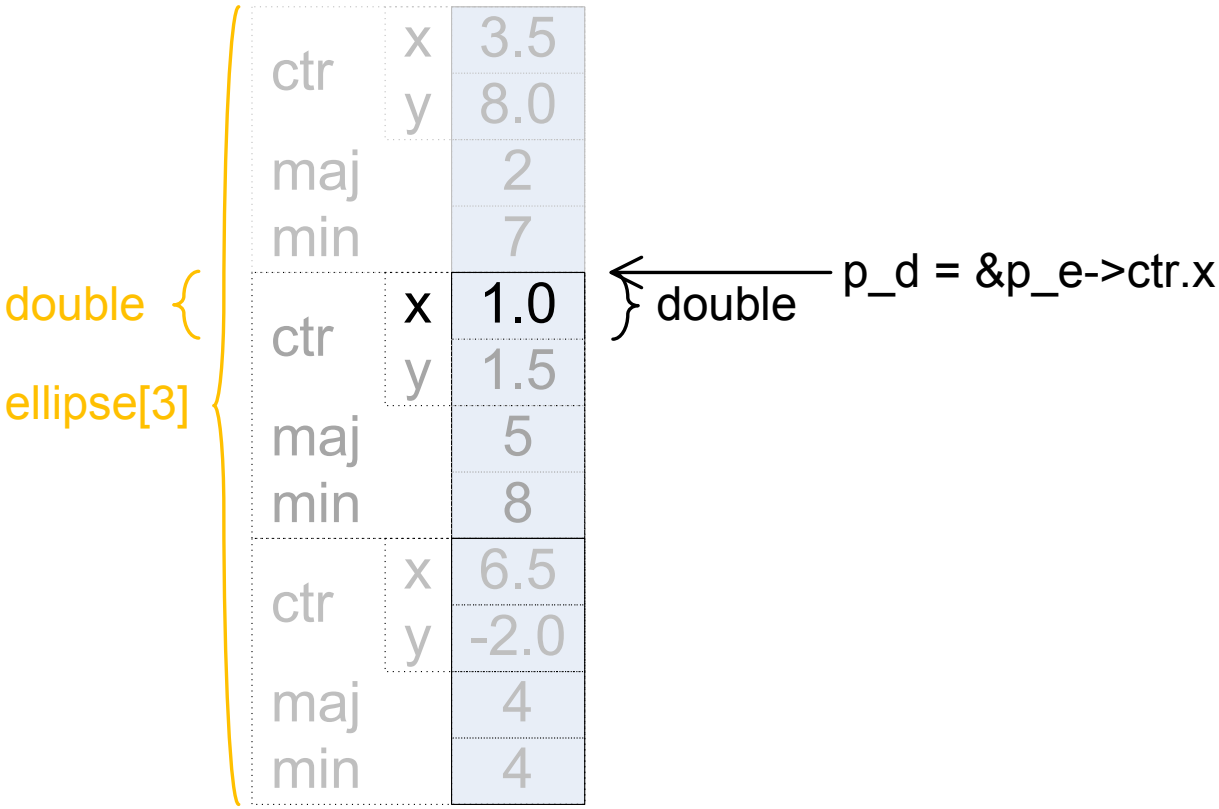

```
struct ellipse {
  struct point {
    double x, y;
  } ctr;
  double maj;
  double min;
} my_ellipses[3];
```

Given allocation type and pointer type, bounds are implicit



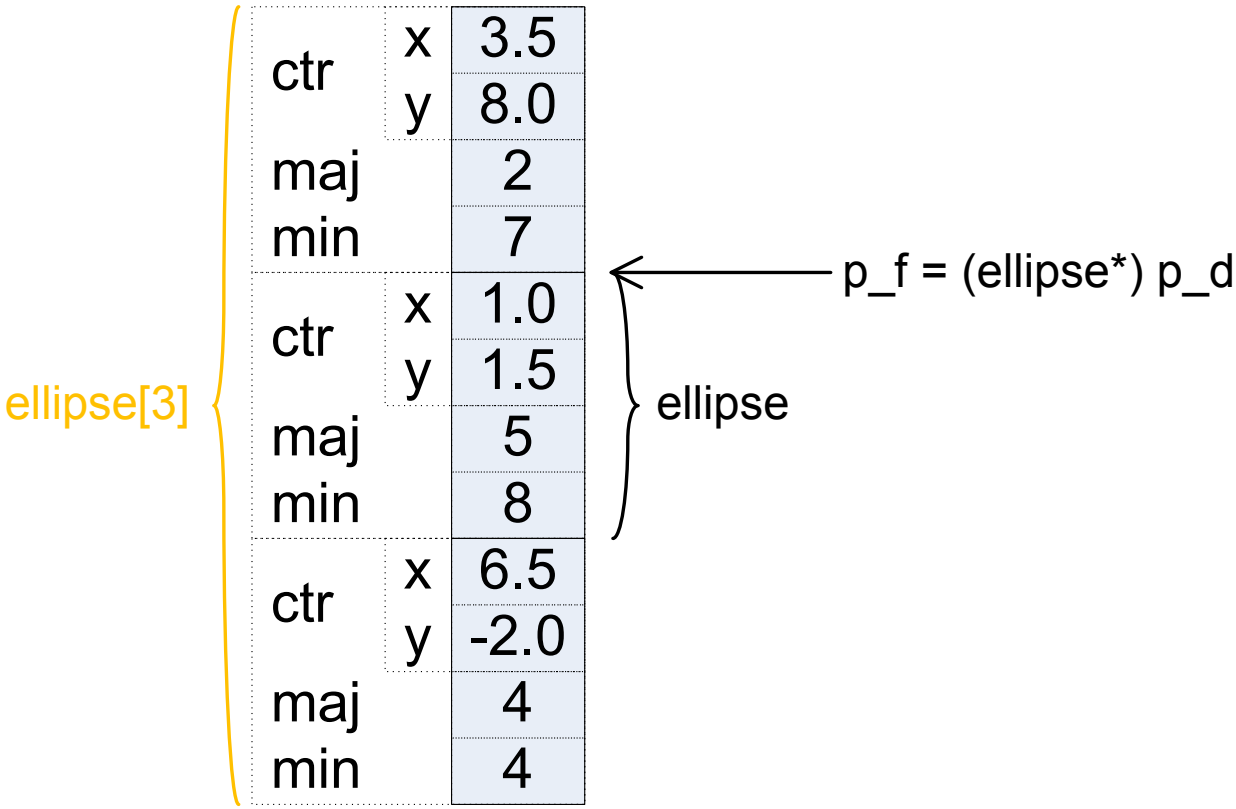

```
struct ellipse {
  struct point {
    double x, y;
  } ctr;
  double maj;
  double min;
} my_ellipses[3];
```

Given allocation type and pointer type, bounds are implicit




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

Given allocation type and pointer type, bounds are implicit




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

The importance of being type-aware (when bounds-checking)

```
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);
```

The importance of being type-aware (when bounds-checking)

```
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);
```

SoftBound is oblivious to casts, even though they matter:

- bounds of d: just the smaller struct
- bounds of the char*: the whole allocation
- bounds of i2c_drv: the bigger struct

If only we knew the *type* of the storage!

Idea

Write a bounds-checker consuming per-allocation metadata

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

Making it fast:

- cache bounds: make pointers "locally fat, globally thin"
- only check *derivation*, not *use*

```
inline int __check_derive_ptr(const void **p_derived,
    const void *derivedfrom, struct uniqtype *t,
    __libcrunch_bounds_t *opt_derivedfrom_bounds);
```

Handling one-past pointers

(diagram: Vladsinger, CC-BY-SA 3.0)

On x86-64, use noncanonical addresses as trap reps

(ask me!)

Status of the bounds checking extension

Does it work?

- yes! . . . modulo a few bugs right now
- several to-dos to make it fast (caching)

How fast will it be?

- no idea yet, but hopeful it can be competitive (or. . .)
- checks per-derive less frequent than per-deref

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

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*)

State of play

- libcrunch is now pretty good at run-time type checking
- supports idiomatic C, source- and binary-compatibly
- *does not check memory correctness*

State of play

- libcrunch is now pretty good at run-time type checking
- supports idiomatic C, source- and binary-compatibly
- *does not check memory correctness*

```
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]);   // use SoftBound
return &z;                      // use CETS
```

State of play

- libcrunch is now pretty good at run-time type checking
- supports idiomatic C, source- and binary-compatibly
- *does not check memory correctness*

```
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;                     // use CETS
```

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 ~~{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)
```

Remaining awkwards

- alloca
- unions
- varargs
- generic use of non-generic pointers (void**, . . .)
- casts of function pointers
  - to a supertype: no check needed
  - to a non-supertype: check against actual function sig

Remaining awkwards

- alloca
- unions
- varargs
- generic use of non-generic pointers (void**, . . .)
- casts of function pointers
  - to a supertype: no check needed
  - to a non-supertype: check against actual function sig

All solved/solvable with some extra instrumentation

- supply our own alloca
- instrument writes to unions
- check va_arg(ap, *type*)…

Handling one-past pointers

```
#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”

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) x
LIBCRUNCH_LAZY_HEAP_TYPES="__PTR_void"
```