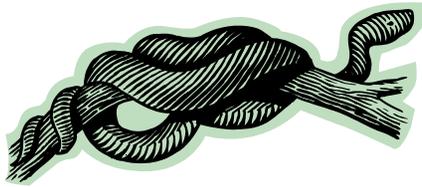


# **Ivy: Modernizing C**

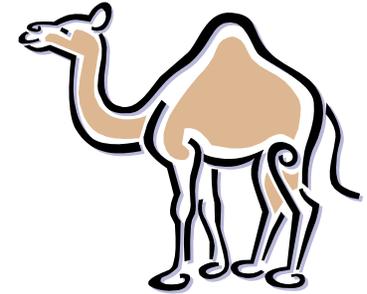
David Gay, Intel Labs Berkeley

*Joint work between UCB and Intel Berkeley: George Necula, Eric Brewer, Jeremy Condit, Zachary Anderson, Matt Harren, Feng Zhou, Bill McCloskey, Robert Ennals, Mayur Naik*

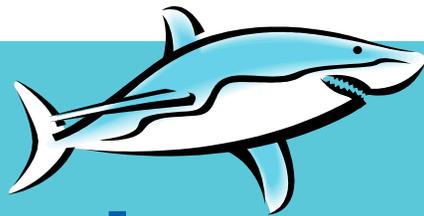
# Why C?



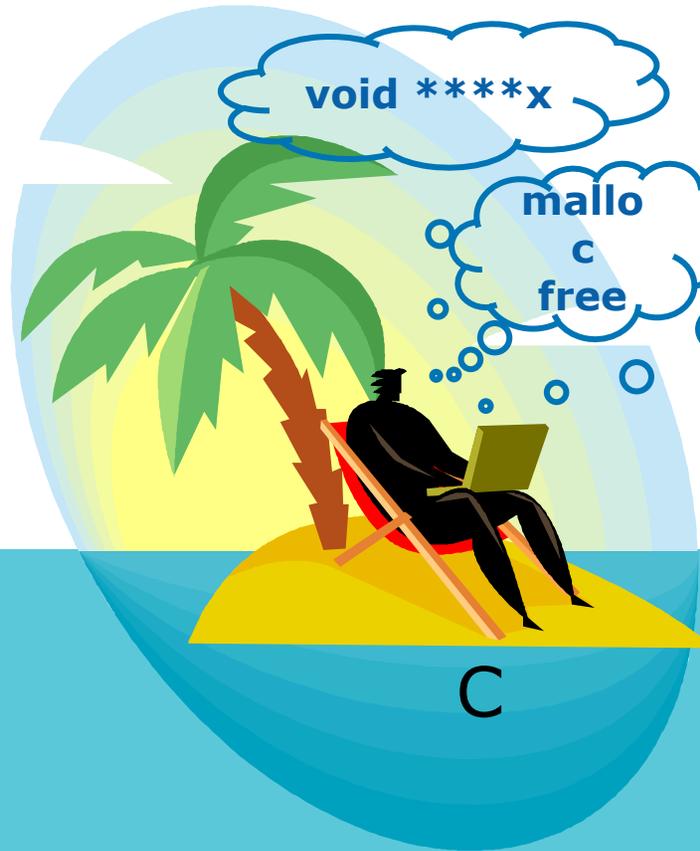
Python



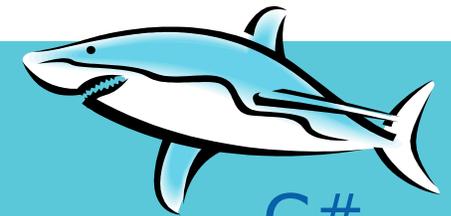
OCaml



Java



C



C#



# C Code is Mostly Correct

C is the predominant systems programming language

- In addition: 12% of projects on sourceforge.net
  - (Java: 18%, C++: 15%, PHP: 11 %, Python: 5%, Ocaml: 0.1%)

C code is mostly correct

- But, no reliable way for programmer to check

## Bug-finding

Use **heuristics** and **approximations** to find common bugs

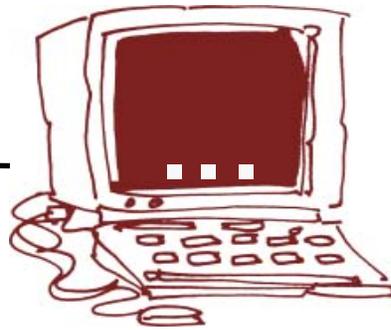
## Soundness

**Prevent the occurrence** of a particular bug class



# Bringing Soundness to C

Horrifying,  
blood-curdling  
C program



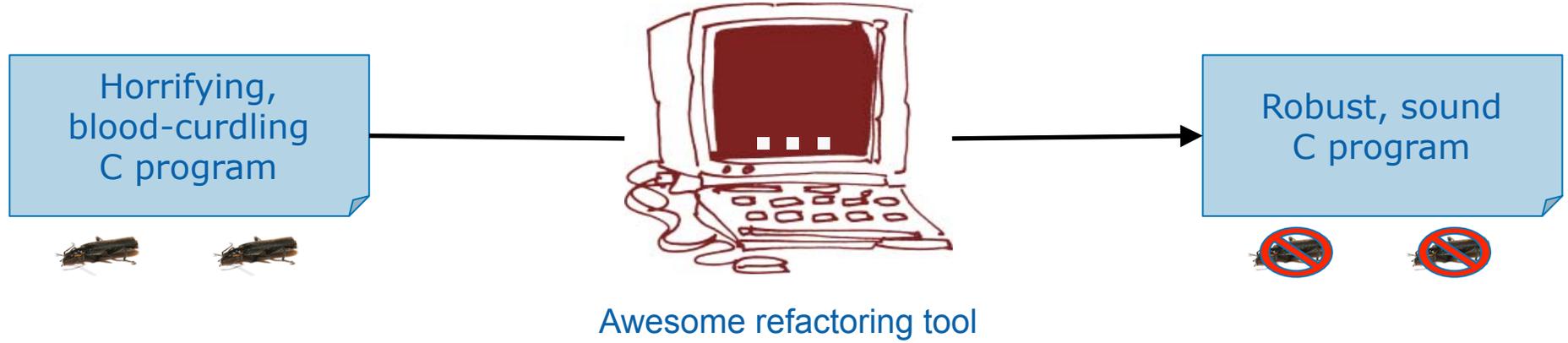
Robust, sound  
C program



Awesome refactoring tool



# Ivy: Bringing Soundness to C



The middle box is hard. The Ivy approach:

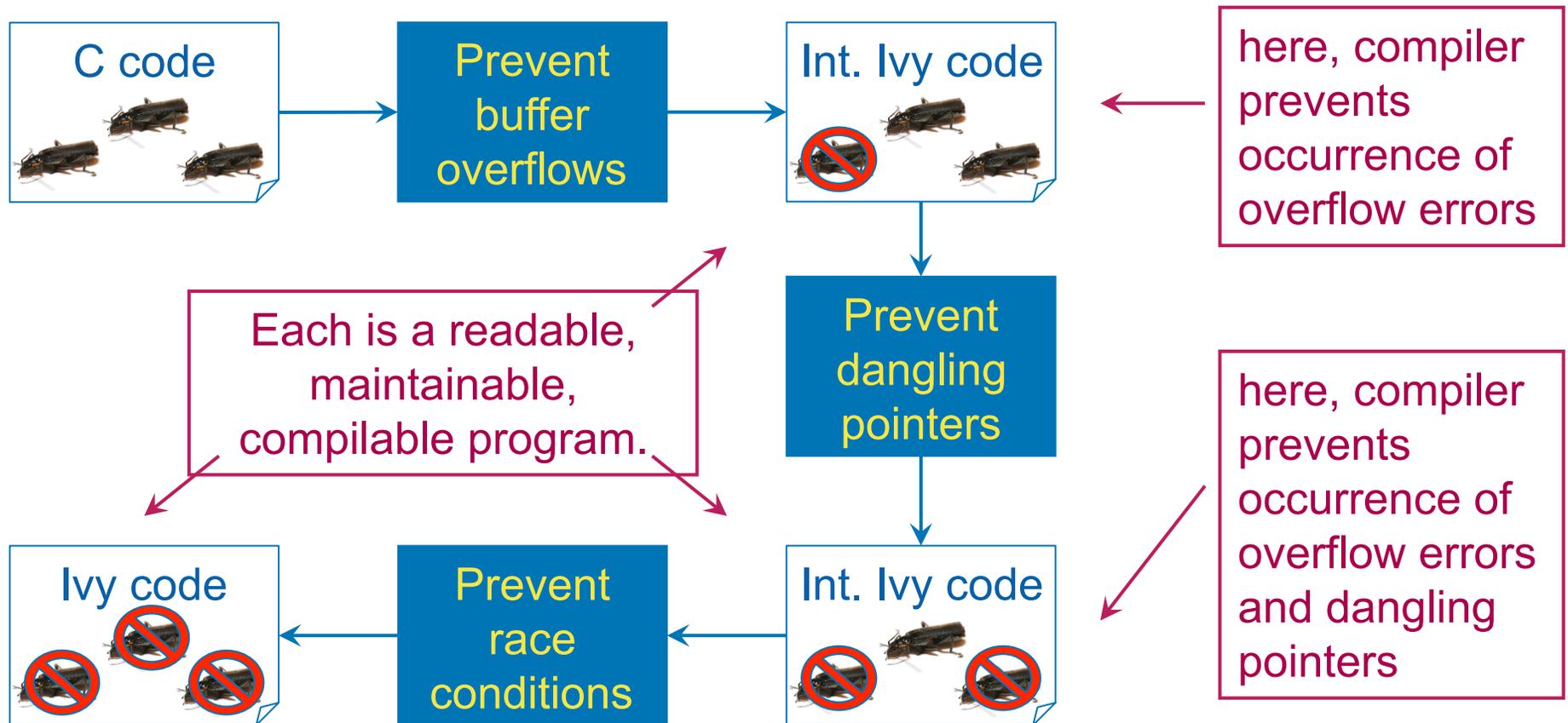
- Let programmers explain why their C is safe
  - Identify code
  - Design goals
- Compiler check
  - Statically
  - Efficiently otherwise
- Support incremental translation, a module, an idiom at a time
  - “trusted” code helps translation, provides escape hatch

**Old: Maximize bugs found**

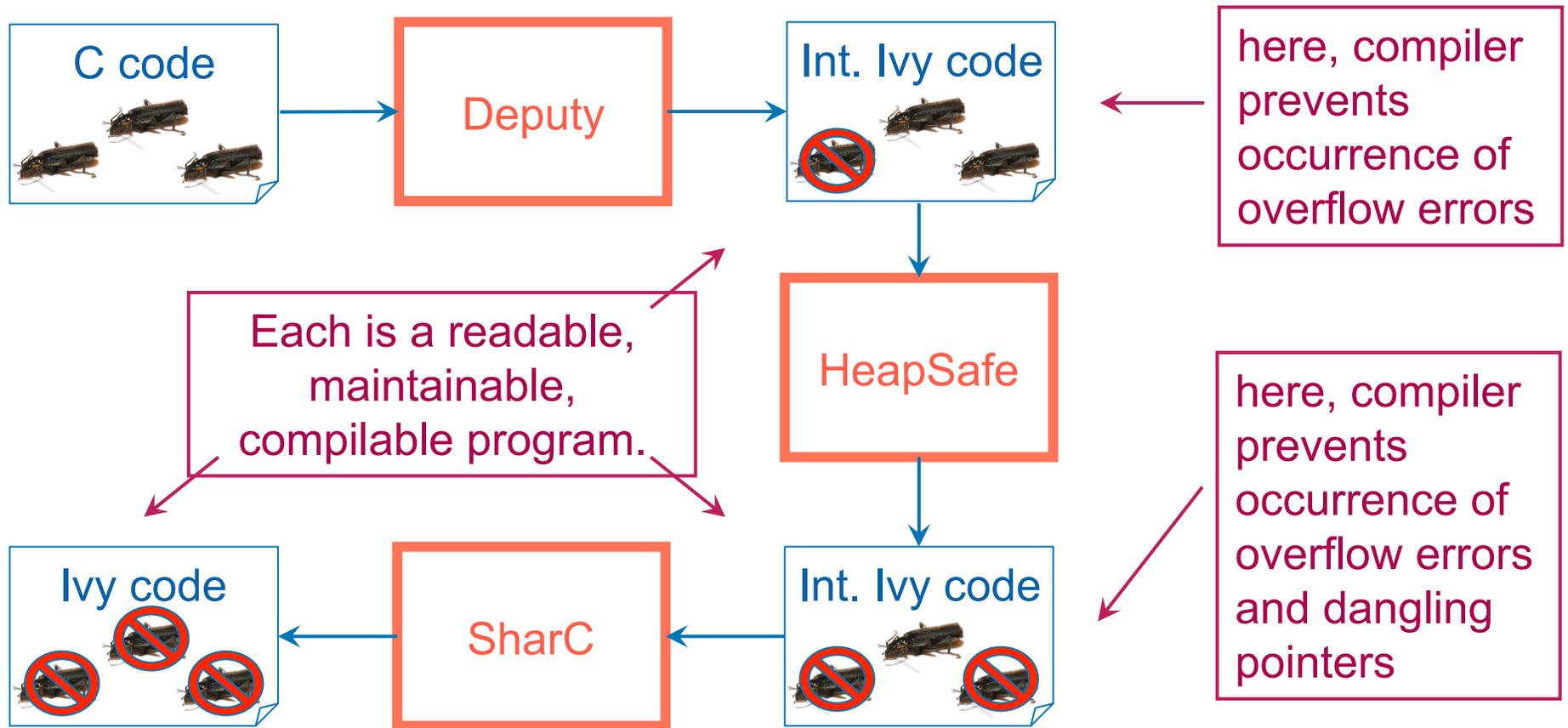
**New: Minimize trusted code**



# Ivy: Bringing Soundness to C



# Ivy: Bringing Soundness to C



# A Proof of Concept – the Linux kernel

Linux 2.6.15.5 kernel

- Small but “real” configuration (435 kLOC)
- Runs on our test machine and in VMware

Analyses

- Deputy: Type safety
- HeapSafe: Memory safety



# Deputy – Type Safety

Deputy enforces type safety

- Programmer inserts buffer size, union, etc annotations and tweaks code
- Compiler performs static checks and inserts run-time checks

Key Observation: *most C code is already safe*

- Rules are usually simple, describable in terms of locally available information!
- Including explanation in source code *improves code readability*



# Deputy Examples

Annotate and check any invariants that are important for type safety

```
struct buffer {  
    char *data;  
    int len;  
};
```

```
void ioctl(int req,  
           void *ptr);
```

```
struct message {  
    int tag;  
    union {  
        int num;  
        char *str;  
    } u;  
};
```



# Deputy Examples

Annotate and check any invariants that are important for type safety

```
struct buffer {  
    char * count(len) data;  
    int len; ←  
};
```

```
void ioctl(int req, ←  
           void * type(req) ptr);
```

```
struct message {  
    int tag; ←  
    union {  
        int num    when(tag == 1);  
        char *str  when(tag == 2);  
    } u;  
};
```



# Deputy Usage

Programmers annotate:

- type definitions
- global variable declarations
- function definitions

Deputy:

- infers local variable annotations
- checks annotated type compatibility
- inserts runtime checks for to ensure that:
  - reads and writes are in bounds
  - type soundness is maintained
- an optimizer attempts to remove unnecessary checks

# HeapSafe – Deallocation Safety

HeapSafe enforces safe use of free

- Programmer tweaks code to use HeapSafe's extended memory management API
- Compiler maintains reference counts for all objects, checks that count is 0 at deallocation time

Key observation: references to dead objects typically deleted *soon after* the dead object itself

- observation exploited by *delayed free scopes*, that delay frees until the end of the scope

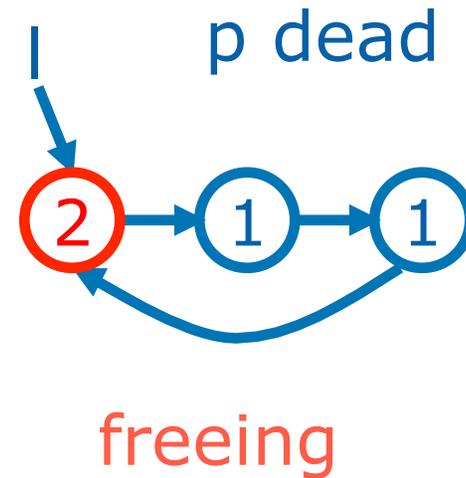


# HeapSafe Example – Delayed Frees

```
struct list {
    struct list *next;
    int data;
};

void free_list(struct list *l) {
    struct list *p = l, *t;

    do {
        t = p->next;
        free(p);
        p = t;
    } while (p != l);
}
```



No-dangling-pointer problem:

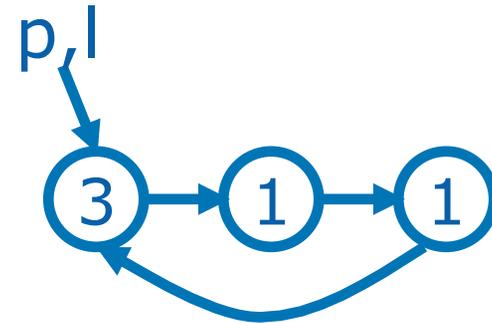
- l still points to object
- last object still points to it too



# HeapSafe Example – Delayed Frees

```
struct list {
    struct list *next;
    int data;
};

void free_list(struct list *l) {
    struct list *p = l;
    delayed_free_start();
    do {
        free(p);
        p = p->next;
    } while (p != l);
    delayed_free_end();
}
```



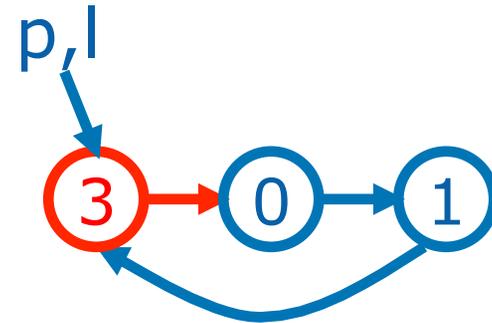
Delayed free scopes:

- make no-dangling-pointer rule practical



# HeapSafe Example – Delayed Frees

```
struct list {  
    struct list *next;  
    int data;  
};  
  
void free_list(struct list *l) {  
    struct list *p = l;  
    delayed_free_start();  
    do {  
        free(p);  
        p = p->next;  
    } while (p != l);  
    delayed_free_end();  
}
```



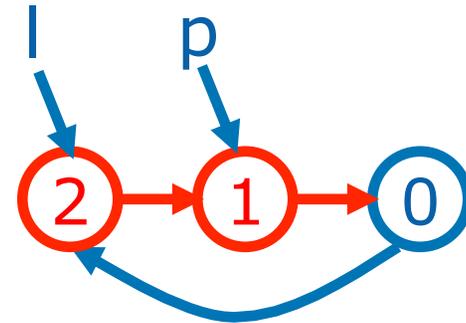
Delayed free scopes:

- make no-dangling-pointer rule practical



# HeapSafe Example – Delayed Frees

```
struct list {  
    struct list *next;  
    int data;  
};  
  
void free_list(struct list *l) {  
    struct list *p = l;  
    delayed_free_start();  
    do {  
        free(p);  
        p = p->next;  
    } while (p != l);  
    delayed_free_end();  
}
```



Delayed free scopes:

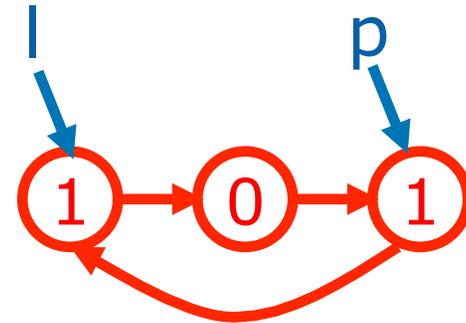
- make no-dangling-pointer rule practical



# HeapSafe Example – Delayed Frees

```
struct list {
    struct list *next;
    int data;
};

void free_list(struct list *l) {
    struct list *p = l;
    delayed_free_start();
    do {
        free(p);
        p = p->next;
    } while (p != l);
    delayed_free_end();
}
```



Delayed free scopes:

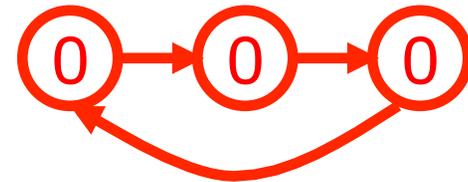
- make no-dangling-pointer rule practical



# HeapSafe Example – Delayed Frees

```
struct list {  
    struct list *next;  
    int data;  
};  
  
void free_list(struct list *l) {  
    struct list *p = l;  
    delayed_free_start();  
    do {  
        free(p);  
        p = p->next;  
    } while (p != l);  
    delayed_free_end();  
}
```

p, l dead



Delayed free scopes:

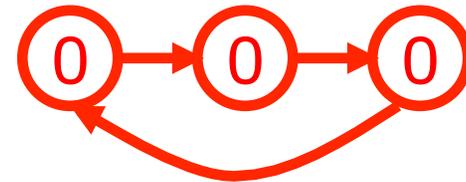
- make no-dangling-pointer rule practical



# HeapSafe Example – Delayed Frees

```
struct list {  
    struct list *next;  
    int data;  
};  
  
void free_list(struct list *l) {  
    struct list *p = l;  
    delayed_free_start();  
    do {  
        free(p);  
        p = p->next;  
    } while (p != l);  
    delayed_free_end();  
}
```

p, l dead



Delayed free scopes:

- make no-dangling-pointer rule practical
- p, l now dead
- no live object refers to freed objects



# Linux Summary (435k LOC)

## Programmer

- inserts **type annotations** and **delayed free scopes**, and **tweaks code**
- Deputy: 2626 LOC annotated (0.6%), 3273 LOC trusted (0.8%)
- HeapSafe: ~160 small, local changes
- Approximately **13 person-weeks** of work

## Compiler

- performs **static checks**
- inserts **reference count updates** and **run-time checks**, trying to avoid unnecessary work

## Runtime

- **maintains** reference counts, and delayed free scopes
- **checks** array bounds, union types, calls to free(), etc
- Linux overhead (hbench): 0-48% for Deputy, 0-173% for HeapSafe



# Other Deputy and HeapSafe Results

**Deputy:** TinyOS sensor network operating system (w/ U. Utah)

- 0.74% LOC changed, 5.2% runtime overhead
- several bugs found
- adopted as part of TinyOS 2.1 release

**Deputy:** 19 applications (from SPEC, Olden, Ptrdist, Mediabench)

- upto 30 kLOC, 2.2% lines annotated
- typical (14/19) overhead below 30%, worst case 98%

**HeapSafe:** 22 applications (mostly from SPEC)

- upto 168 kLOC (perl), 0.6% lines annotated
- typical (20/22) overhead below 22%, worst case 84% (perl, again)



# Concurrency Safety

Threads and locks widely used, but lots can go wrong

- Data Races
- Deadlocks
- Atomicity violations
- etc.

Key Observation: People have a strategy for correct concurrency

- Race detectors don't know about it
- So they try to infer what the programmer wants
- This inference could be wrong



# SharC

Goal: Allow sharing of data only how/when programmer *intends*

Programmer declares the *high-level sharing strategy*

- *what are the sharing rules (e.g. lock-protected) for a given object?*
- *where do the sharing rules change?*

SharC *soundly* checks the sharing strategy through static and dynamic analysis

- Violations of the sharing strategy **include** real data races

Good performance on real programs

- up to ~1 million LOC
- few annotations
- reasonable overheads, generally below 20%



# Sharing Rules and Strategies

Sharing Strategy = Sharing Modes + Mode Changes:

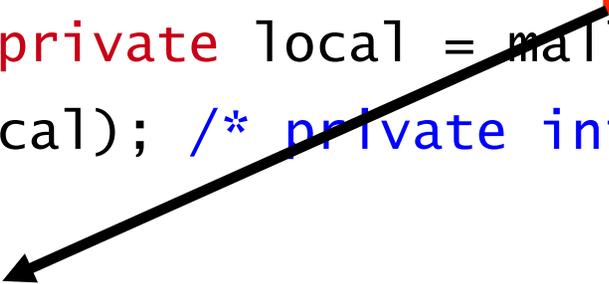
- Lock protected counters/queues/caches
  - **locked(I)**
- Data initialized, then read-shared
  - **private -> readonly**
- Ownership of data transferred via locked queue
  - **private -> locked(I) -> private**
- Trusted objects
  - **racy**
- etc.



# Example: data initialized, then lock-protected

```
mutex_t *L;  
/* a global buffer protected by mutex L */  
char locked(L) *locked(L) buffer;  
void init_buffer() {  
    char private *private local = malloc(SIZE);  
    init_buffer(local); /* private initialization */  
    mutex_lock(L);  
    buffer = local;  
    mutex_unlock(L);  
}
```

Type Error!  
char **locked(L)** \* vs.  
char **private** \*



# Single-object mode change

```
mutex_t *L;  
/* a global buffer protected by mutex L */  
char locked(L) *locked(L) buffer;  
void init_buffer() {  
    char private *private local = malloc(SIZE);  
    init_buffer(local); /* private initialization */  
    mutex_lock(L);  
    buffer = SCAST(char locked(L)*, local);  
    mutex_unlock(L);  
}
```

Explicit, checked  
*Sharing Cast*  
expresses  
mode change!



# SharC: Sharing Casts

Key Insight: If  $p$  is the sole pointer to object  $O$ , then you can change the sharing strategy for  $O$  by changing  $p$ 's type.

SharC:

- maintains reference counts (already used in HeapSafe)
- checks that the reference count is one at a cast
- atomically nulls-out the old value (it has the wrong type):
- So:

```
buffer = SCAST(char locked(L) *, local);
```

is effectively (atomically)

```
buffer = local; local = NULL;
```

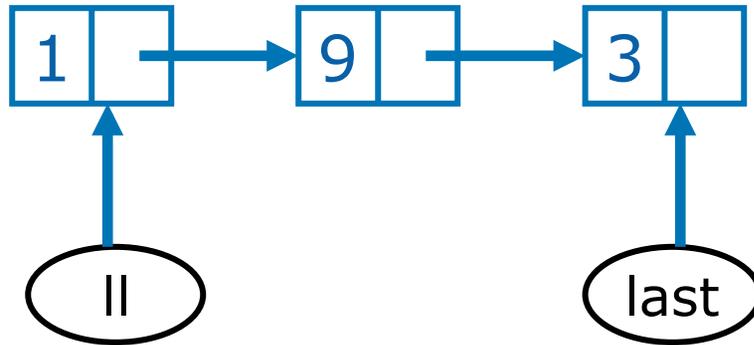


## A slight modification

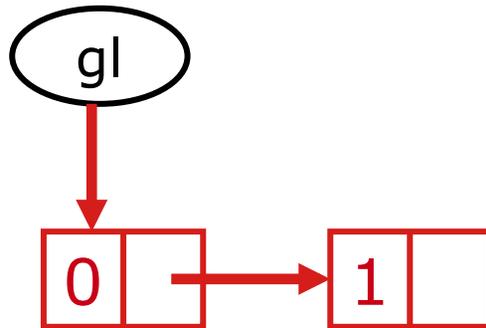
```
struct list {int data; struct list *next;};
mutex_t *L;
/* a global list protected by mutex L */
struct list locked(L) *locked(L) gl;
void init_list() {
    struct list private *private ll, *private last;
    ll = malloc(...);
    last = init_list(ll); /* private initialization */
    mutex_lock(L);
    gl = SCAST(struct list locked(L)*, ll);
    mutex_unlock(L);
}
```



# The problem with data structures



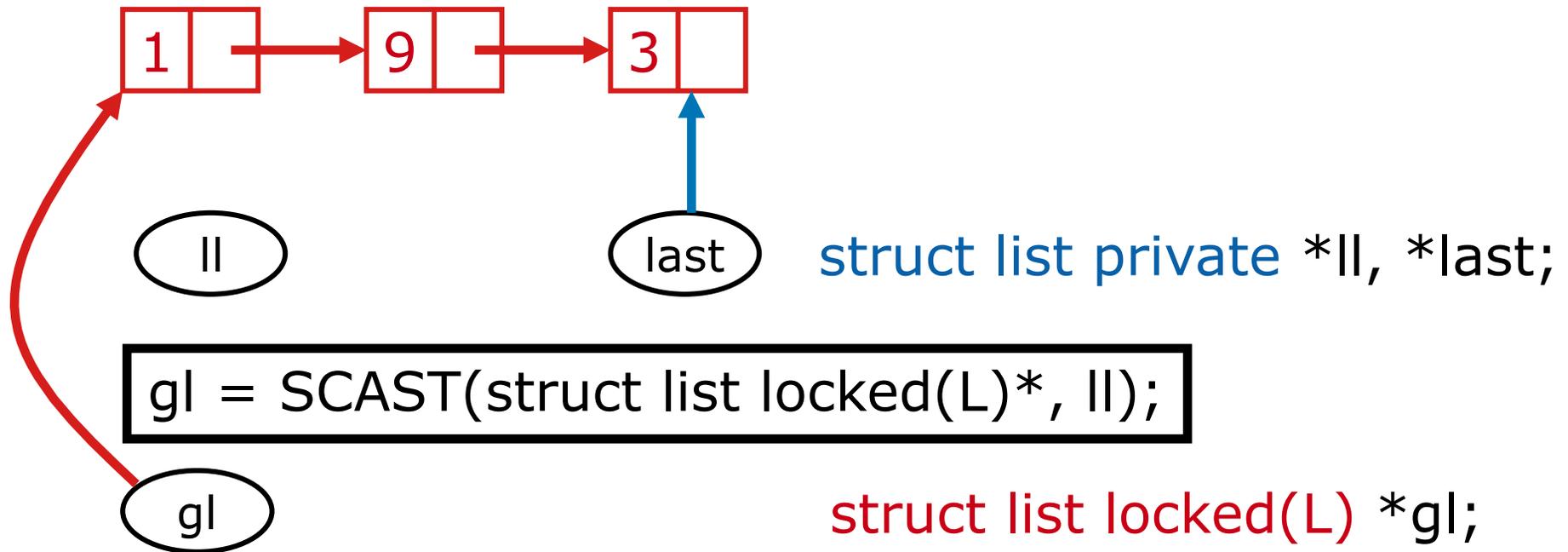
```
struct list private *ll, *last;
```



```
struct list locked(L) *gl;
```



# The problem with data structures



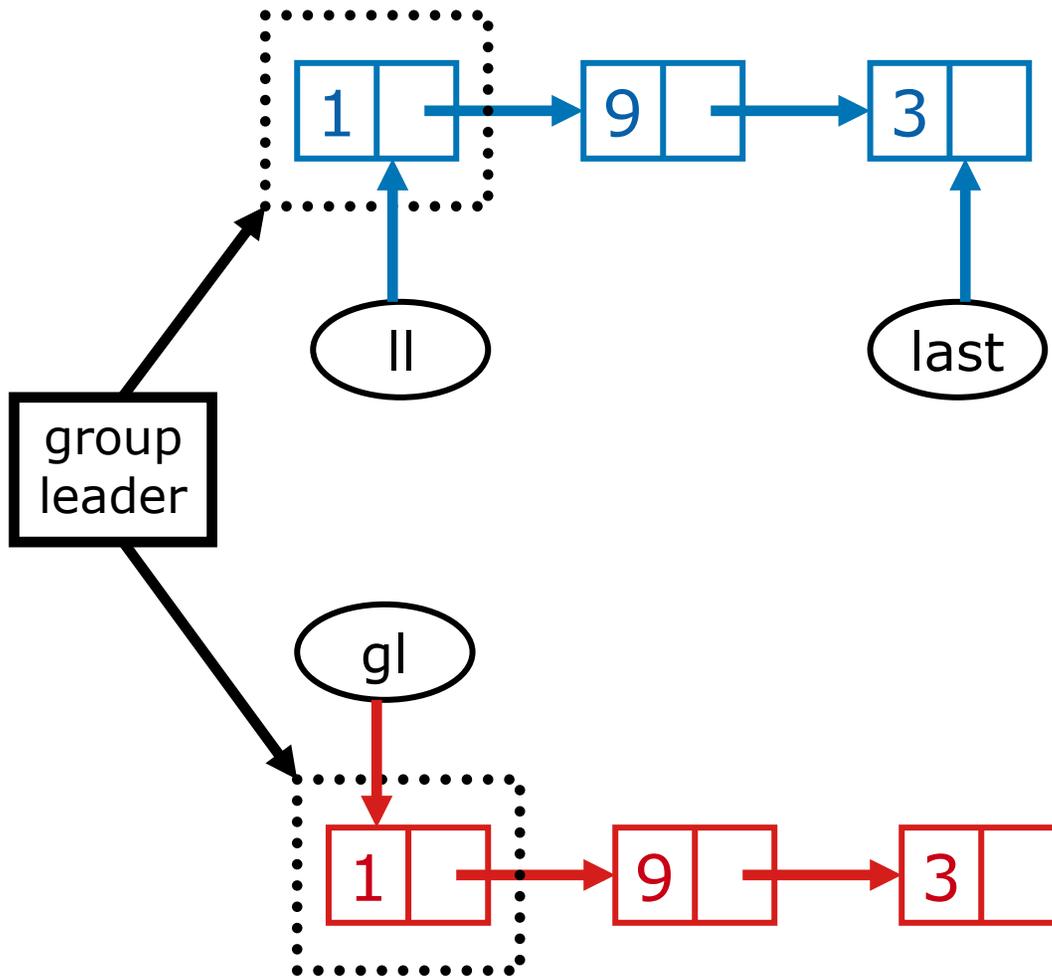
# Solution: Groups

A *group* is a collection of objects identified by a distinguished *group leader*

- e.g. the head of a list, or the root of a tree.



# Groups



```
struct list private *ll;  
struct list private *last;  
struct list locked(L) *gl;
```



# Solution: Groups

A *group* is a collection of objects identified by a distinguished *group leader*

- e.g. the head of a list, or the root of a tree.

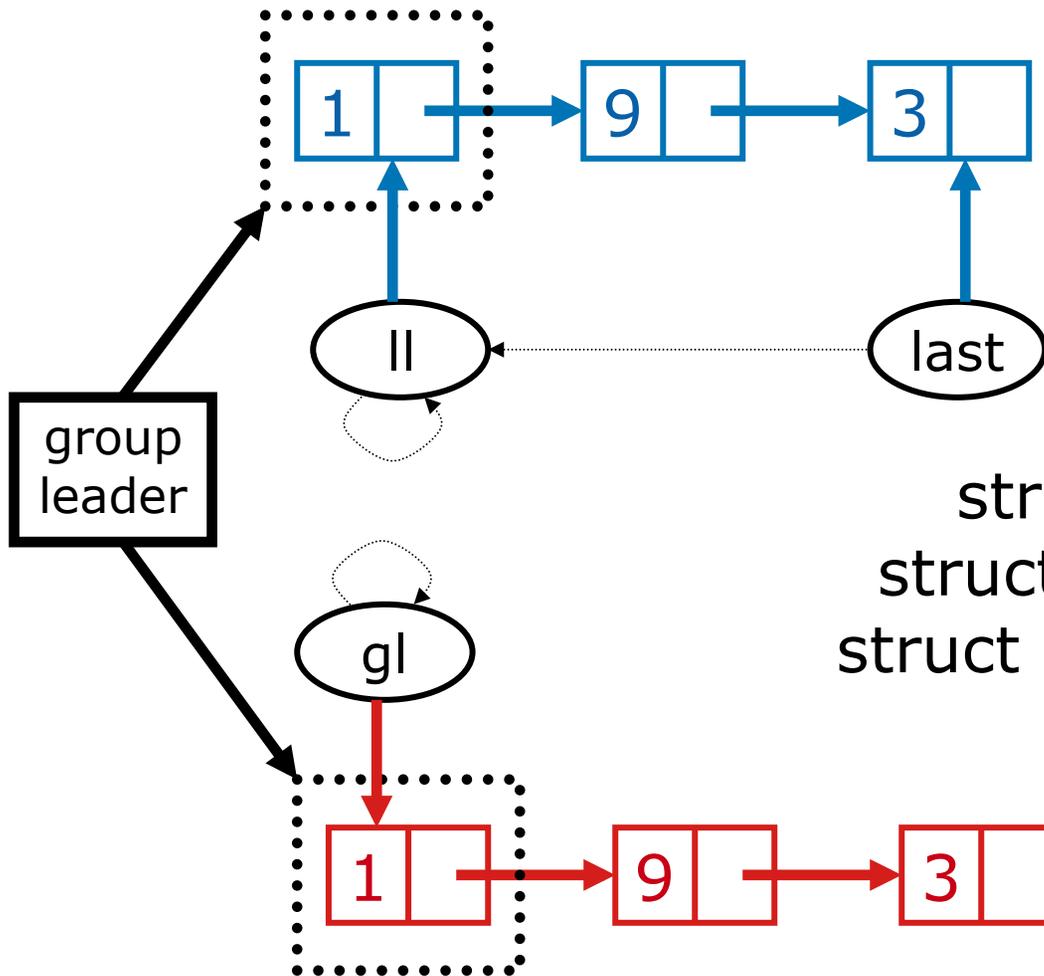
We identify *external pointers* into a data structure and require that they **depend** on a pointer to the group leader

- We use a dependent-type annotation, `group(g)`, that means a “pointer into the group whose leader is `*g`”, e.g.:

```
struct list group(g1) *g1;
```



# Groups



```
struct list group(ll) private *ll;  
struct list group(ll) private *last;  
struct list group(gl) locked(L) *gl;
```



# Solution: Groups

A *group* is a collection of objects identified by a distinguished *group leader*

- e.g. the head of a list, or the root of a tree.

We identify *external pointers* into a data structure and require that they **depend** on a pointer to the group leader

- We use a dependent-type annotation, `group(g)`, that means a “pointer into the group whose leader is `*g`”, e.g.:

```
struct list group(g1) *g1;
```

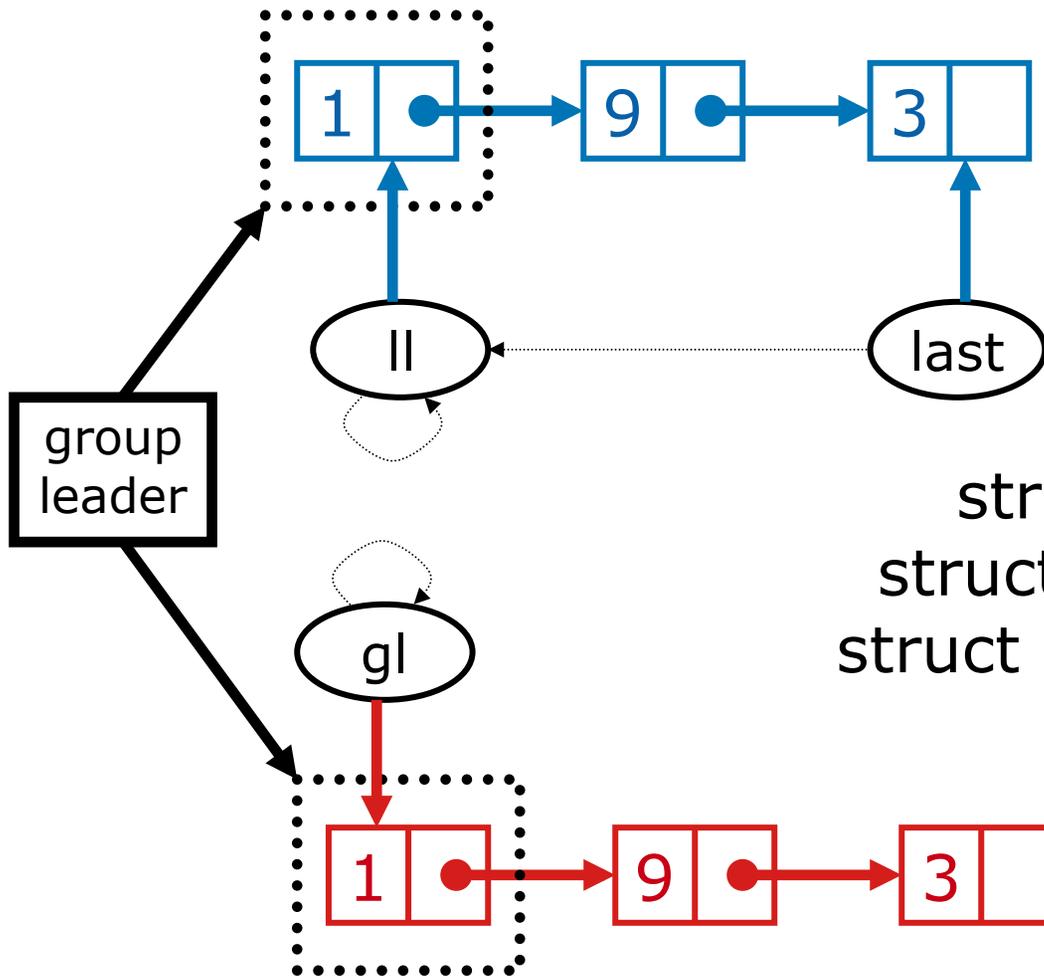
We identify *internal pointers* in a data structure

- We use an annotation on the pointer types of structure fields, `same`, that means “a pointer into the same group”, e.g.:

```
struct list {  
    int data;  
    struct list same *next;  
};
```



# Groups



```
struct list group(ll) private *ll;  
struct list group(ll) private *last;  
struct list group(gl) locked(L) *gl;
```



# Groups

```
struct list {int data; struct list *same next;};
mutex_t *L;
/* a global list protected by mutex L */
struct list group(g1) Locked(L) * Locked(L) g1;
void init_list() {
    struct list group(l1) private *l1, *last;
    l1 = malloc( );
    last =
    mutex_
    g1 = GCAST(struct list group(g1) Locked(L)*, l1);
    mutex_unlock(L);
}
```

**Goal:** Use knowledge about internal/external pointers, to make **GCAST** safe.



# Group Enforcement

Goal: check that the argument to GCAST is the only external pointer into the data structure

Consider **GCAST(struct list group(gl) locked(L) \*, ll)**

- The type of ll is **struct list group(ll) private \***
- If there is another external pointer, either it depends on ll or on another pointer to the group leader (\*ll).
- If it depends on ll, then it must be in scope. We can check that it is NULL at the cast.
- If it depends on another group leader pointer, then the reference count for \*ll will be bigger than one



# Groups

```
struct list {int data; struct list *same next};
```

```
mutex_t *L;
```

```
/* a global list protected by L */
```

```
struct list group(g1) locked(L);
```

```
void init_list() {
```

```
    struct list group(l1) private *l1, *last;
```

```
    l1 = malloc(...);
```

```
    last = init_list(l1); /* private initialization */
```

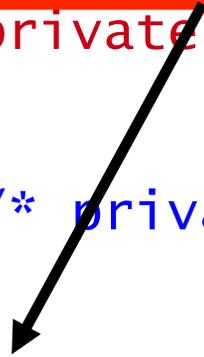
```
    mutex_lock(L);
```

```
    g1 = GCAST(struct list group(g1) locked(L)*, l1);
```

```
    mutex_unlock(L);
```

```
}
```

In this example, the cast fails unless **last** is nulled out first, as last depends on l1.



# SharC Evaluation

Goal: Investigate overheads, prove applicability to real programs

- Not a bug-finding mission
- Instead: Proving the absence of a class of bugs:  
Benchmarks run with one benign race and no false positives

9 Linux applications, 2 scientific kernels

- 1.1k - 936k LOC, at most  $\sim 100$  annotations and casts / program
  - intuition: annotation complexity scales with concurrency complexity
- 5 need groups, groups range from 8 - 100k objects
- $< 20\%$  time overhead for applications,  $\sim 40\%$  time overhead for kernels
- Time overhead stays independent of thread count (tested up to 8 threads)



# SharC Summary

Incorrect sharing leads to races and other bugs

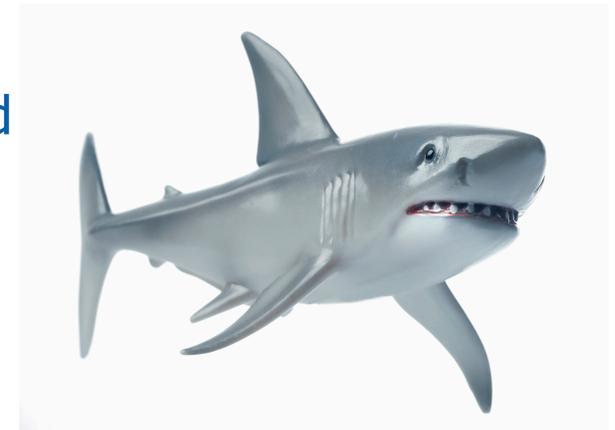
With a few simple annotations on types:

- Programmers specify *data structures* and *sharing strategy*
- And where the strategy for an object or data structure *changes*

SharC checks that the program respects the declared strategy

SharC works on real programs

- Evaluated on over one million lines
- Manageable slowdown and memory overhead



# Ivy Availability

Ivy compiler is open source, contains SharC, Deputy (type safety) and HeapSafe (memory safety)

- <http://ivy.cs.berkeley.edu>
- Ivy 1.0 (no groups) is available for
  - Linux, Mac OS X
  - 32-bit only
- Earlier versions of Deputy, HeapSafe available as separate tools
- A 1.1 release with groups and 64-bit support should happen soon



```
char *GROUP(y) *COUNT(3) x
```

```
delayed_free_start  
()  
_delayed_free_end()
```

```
char private  
*readonly x
```

Ivy



# BACKUP



# Deputy Results

Deputy processes the 435 KLOC kernel

- 2626 lines annotated (0.6%)
- 3273 lines trusted (0.8%)

Approximately 7 person-weeks of work  
...and the second derivative is positive!



# Deputy Performance

## HBench-OS

kernel  
benchmarks

[Brown &  
Seltzer '97]

Bandwidth Tests	Ratio	Latency Tests	Ratio
bzero	1.01	connect	1.10
file_rd	0.98	ctx	1.15
mem_cp	1.00	ctx2	1.35
mem_rd	1.00	fs	1.35
mem_wr	1.06	fslayer	1.04
mmap_rd	0.85	mmap	1.41
pipe	0.98	pipe	1.14
tcp	0.83	proc	1.29
		rpc	1.37
		sig	1.31
		syscall	0.74
		tcp	1.41
		udp	1.48

# HeapSafe Results

Few annotations and changes

- 32 type changes, 27 RTTI insertions
- 50 uses of memcpy / memset changed
- 27 changes for null pointers
- 26 delayed free scopes

Bad frees reported

- Boot: 0 in 107K frees
- HBench: 677 in 100M frees

# HeapSafe Performance

## HBench-OS

kernel  
benchmarks

[Brown &  
Seltzer '97]

Bandwidth Tests	Ratio	Latency Tests	Ratio
bzero	1.00	connect	2.10
file_rd	0.99	ctx	1.17
mem_cp	1.00	ctx2	1.13
mem_rd	1.00	fs	2.73
mem_wr	0.99	fslayer	0.98
mmap_rd	0.93	mmap	1.21
pipe	0.97	pipe	1.12
tcp	0.66	proc	1.30
		rpc	2.01
		sig	1.23
		syscall	1.22
		tcp	2.55
		udp	1.82

# Sharing Rules are Type Qualifiers

int **readonly** \* **private** X;

- **private** means that X can only be accessed by owning thread
- **readonly** means that the *target* of X can only be read
- Well-formedness rule:
  - No non-**private** pointers to **private** objects
    - Allowing these would be unsound
- Programmer makes a few key annotations
  - SharC infers the rest via:
    - Simple systematic rules
    - Global sharing analysis to infer **dynamic, private**
      - On unannotated declarations

# SharC Guarantees

- If there are no compile time errors:
  - **private** objects are only accessed by the owning thread
  - **readonly** objects aren't written
    - except for initialization
  - Sharing modes agree at assignments/for arguments(i.e. type safety)
- If there are no runtime errors:
  - no races on **dynamic** objects
  - **locked** objects always protected by the right lock
  - no references with wrong type after a Sharing Cast
- Sharing strategy violations include all data races on a run
  - Except on objects marked **racy** by the programmer
- If there is a race between two running threads
  - SharC will generate an error regardless of the thread schedule
- Proved sound for **private** and **dynamic** modes with Sharing Casts
  - Formalism readily extendable to include other modes

# SharC Evaluation

- Goal: Investigate overheads
  - Not a bug-finding mission
  - Instead: Proving the absence of a class of bugs:  
Benchmarks run with one benign race and no false positives
- No groups needed:
  - pfsfan - Parallel file scanning similar to find and grep
  - pbzip2 - Parallel bzip2 implementation
  - aget - Download accelerator
  - dillo - Simple web browser
  - fftw - FFT computation
  - stunnel - TCP connection encryption
- Groups needed:
  - ebarnes - n-body simulation, oct-tree
  - em3d-s - EM wave propagation, bipartite graph
  - knot - webserver, in-memory cache
  - eog - image viewer, complex "task" structures
  - GIMP - image manipulator, image "tiles"



# Evaluation

Benchmark name	Lines	Annots +SCASTs	Other Mods <sup>*</sup>	# Thread	Space overhead	Time overhead
pfscan	<b>1.1k</b>	8	11	3	0.8%	12%
aget	<b>1.1k</b>	7	7	3	30.8%	~0%
pbzip2	<b>10k</b>	10	36	5	1.6%	11%
dillo	<b>49k</b>	8	8	4	78.8%**	14%
fftw	<b>197k</b>	7	39	3	1.2%	7%
stunnel	<b>361k</b>	20	22	3	43.5%	2%

\*Casts that strip our qualifiers were manually removed

\*\*High due to pointer/int confusion interacting with reference counting



# Evaluation

Benchmark name	Lines	Annots +SCASTs	Other Mods <sup>*</sup>	# Thread	Space overhead	Time overhead
pfscan	1.1k	<b>8</b>	11	3	0.8%	12%
aget	1.1k	<b>7</b>	7	3	30.8%	~0%
pbzip2	10k	<b>10</b>	36	5	1.6%	11%
dillo	49k	<b>8</b>	8	4	78.8%**	14%
fftw	197k	<b>7</b>	39	3	1.2%	7%
stunnel	361k	<b>20</b>	22	3	43.5%	2%

Run with one benign race and no false positives

\*Casts that strip our qualifiers were manually removed

\*\*High due to pointer/int confusion interacting with reference counting



# Evaluation

Benchmark name	Lines	Annots +SCASTs	Other Mods	# Thread	Space overhead	Time overhead
pfscan	1.1k	8	11	3	0.8%	<b>12%</b>
aget	1.1k	7	7	3	30.8%	<b>~0%</b>
pbzip2	10k	10	36	5	1.6%	<b>11%</b>
dillo	49k	8	8	4	78.8%*	<b>14%</b>
fftw	197k	7	39	3	1.2%	<b>7%</b>
stunnel	361k	20	22	3	43.5%	<b>2%</b>

\*High due to pointer/int confusion interacting with reference counting



# Evaluation

Benchmark name	Lines	Annots + SCASTs	Max Group Size	# Threads	Time overhead
ebarnes	3k	76	16k	2	38%
em3d-s	1k	44	100k	2	42%
knot	5k	77	75	3	19%
eog	38k	105	20	4	6%
GIMP	936k	69	8	4	13%

In our experiments with these benchmarks, overhead does *not* increase with increased concurrency up to 8 threads



# Why is the runtime overhead low?

- No dynamic analysis needed for **private** or **readonly** data
- Thanks to sharing casts, typically only a small proportion of accesses are to **dynamic** data
  - ~20% on average in our benchmarks--typically less
- Lock-free reference counting algorithm is very efficient