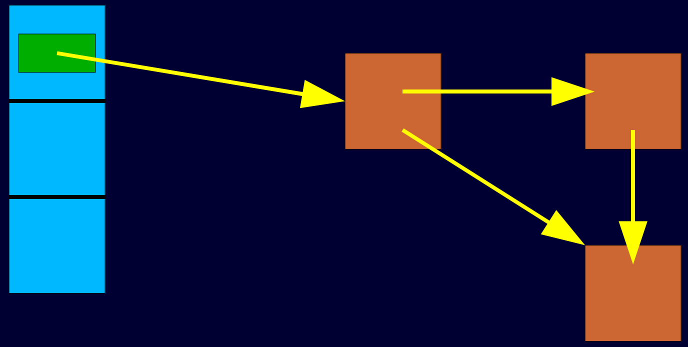
# Automatic Inference of Reference Count Invariants

David Detlefs
Sun Microsystems Laboratories
SPACE – Jan. 2004

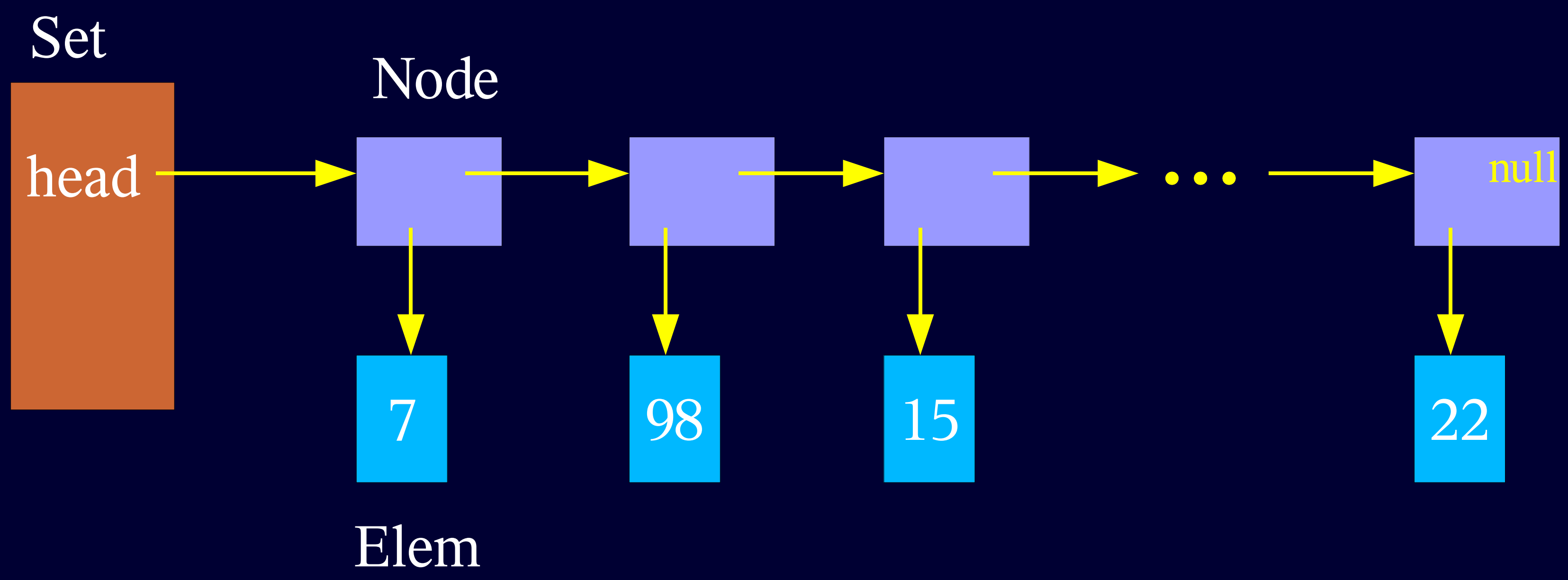
# Goal of this Work

- A form of compile-time GC.
  - Escape analysis:
  - Region inference:
    - Mostly short-lived objects.
    - Reachability rooted in local variables of stack frames.



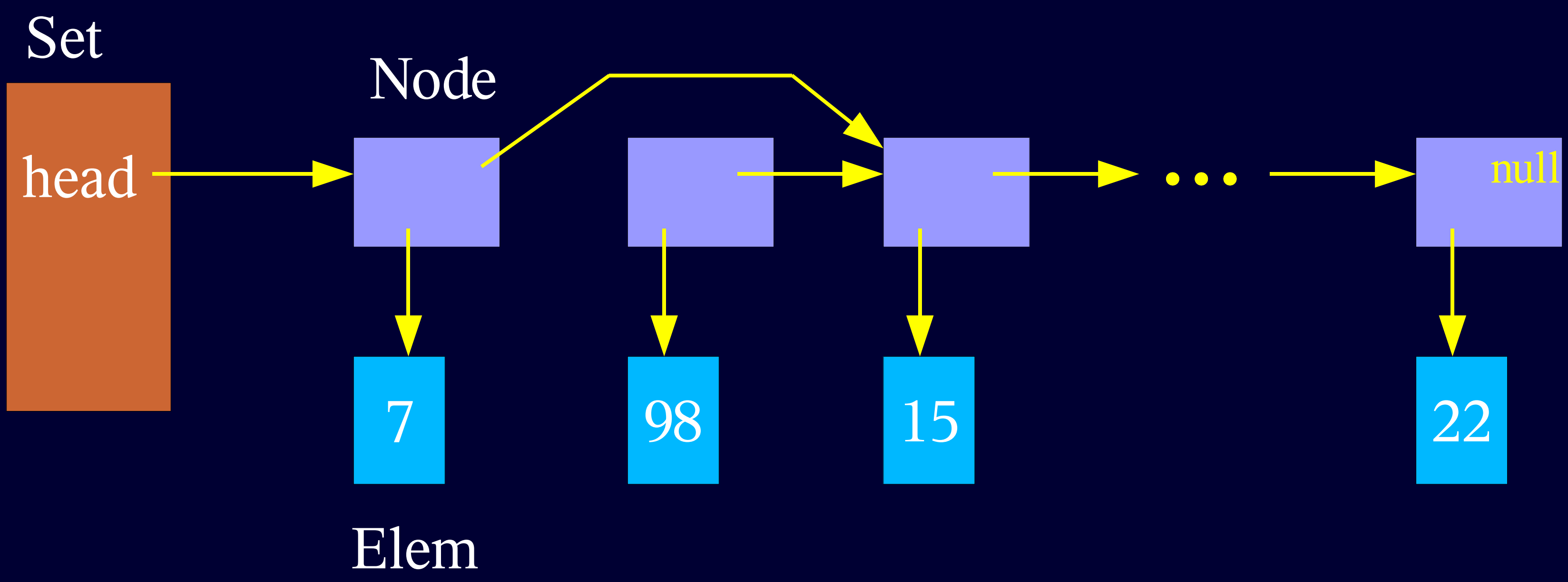
- In contrast: automatically infer correct explicit deallocation of elements of long-lived data structures.

# Motivating Example: Set via Linked List



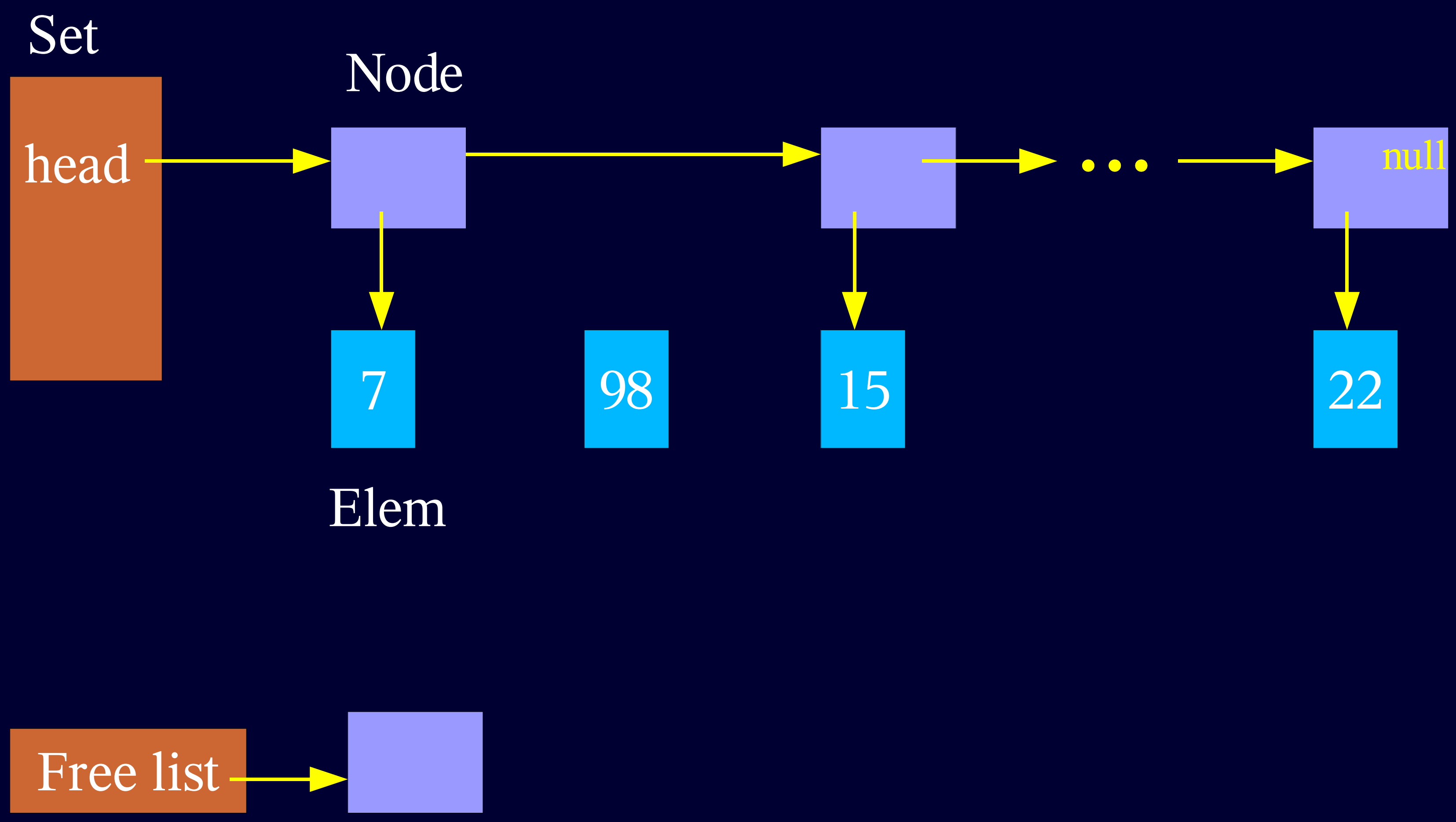
# Motivating Example: Set via Linked List

delete(98):



# Motivating Example:
# Set via Linked List

delete(98):



# Outline of the Remainder

- When can you safely free something?
- What *class invariant* do you need?
- The role of ownership in class invariants.
- How we might infer such invariants.
  - "Whole-class" abstract interpretation.
  - "Owned-by-this" abstraction.
- Other related work.
- Current implementation status.
- Future work.

# When can you safely free something?

- When its reference count is zero!

  { v == r & rc[r] == 1 }
  <last use of v:  "v = null">
  { rc[r] == 0 }
  .
  .
  .
  <v goes out of scope>

# When can you safely free something?

- When its reference count is zero!

  { v == r & rc[r] == 1 }
  free(v); { v == null, rc[r] == 0 }
  .
  .
  .
  <v goes out of scope>

- (This is what escape analysis does...)

# What does this mean in the example?

- ```
  void delete(Elem o) {
    Node hd = head;
    Node prev = null;
    while (hd != null) {
      if (o.equals(hd.elem)) {
        if (prev == null) head = hd.next;
        else prev.next = hd.next;
        return;
      } else {
        prev = hd; hd = hd.next;
  }}}
  ```
  ```

# What does this mean in the example?

- void delete(Elem o) {

```
Node hd = head;
Node prev = null;
while (hd != null) {
  if (o.equals(hd.elem)) {
    if (prev == null) head = hd.next;
    else prev.next = hd.next;
    free(hd);
    return;
  } else {
    prev = hd; hd = hd.next;
}}}
```

# What Class Invariant do you need?

- Need to know that all Nodes making up a Set representation have reference count 1:
- $\forall$ s: Set ::

$$
\begin{array}{l}
(\quad s.head = null \\
\quad \vee (\quad s.head \neq null \wedge owner[s.head] = s \\
\qquad \wedge (\forall\ n: Node :: (n \neq null \wedge owner[n] = s) \Rightarrow \\
\qquad\quad (\quad rc[n] = 1 \\
\qquad\qquad \wedge (\quad n.next = null \\
\qquad\qquad\quad \vee (n.next \neq null \wedge owner[n.next] = s) \\
\qquad\qquad ))))))
\end{array}
$$

# The role of ownership

- A nasty problem in program semantics:
  - Which objects are "subobjects" of others objects?
  - (or…) what object fields may contribute to the abstract state of object *x*?
    - Reachability?  Very hard, not always the right concept.
- Ownership makes this explicit: objects owned by x may contribute to abstract state of *x*.
- For this talk:
  - new objects are unowned.
  - can only set the owner of unowned objects.
  - Heuristic: if *y* is unowned,   *x*.f = *y*   sets owner[*y*] to
    - owner[*x*],     if that is known, else
    - *x*     (so   *n* = new Node; ... ; *s*.head = *n*
      sets   owner[*n*] = *s* )

# Treatment of Reference Count

- “rc” is a state variable.
- Translation of source program (to *guarded command* program) elaborates with updates of rc:

lhs = rhs

# Treatment of Reference Count

- "rc" is an implicit state variable.
- Translation of source program (to *guarded command* program) elaborates with updates of rc:

```
rc[lhs] = rc[lhs] – 1;
{ tmp :
  tmp = rhs;
lhs = tmp;
  rc[tmp] = rc[tmp] + 1 }
```

# Invariant Inference

- "Whole-class" abstract interpretation.
  - Initially: there are no Set's allocated.
  - Create a state in which the first Set is allocated, and execute its constructor(s).
  - Abstract the state(s) back to an invariant true of all Set's seen so far:

    Set() { head = null; }

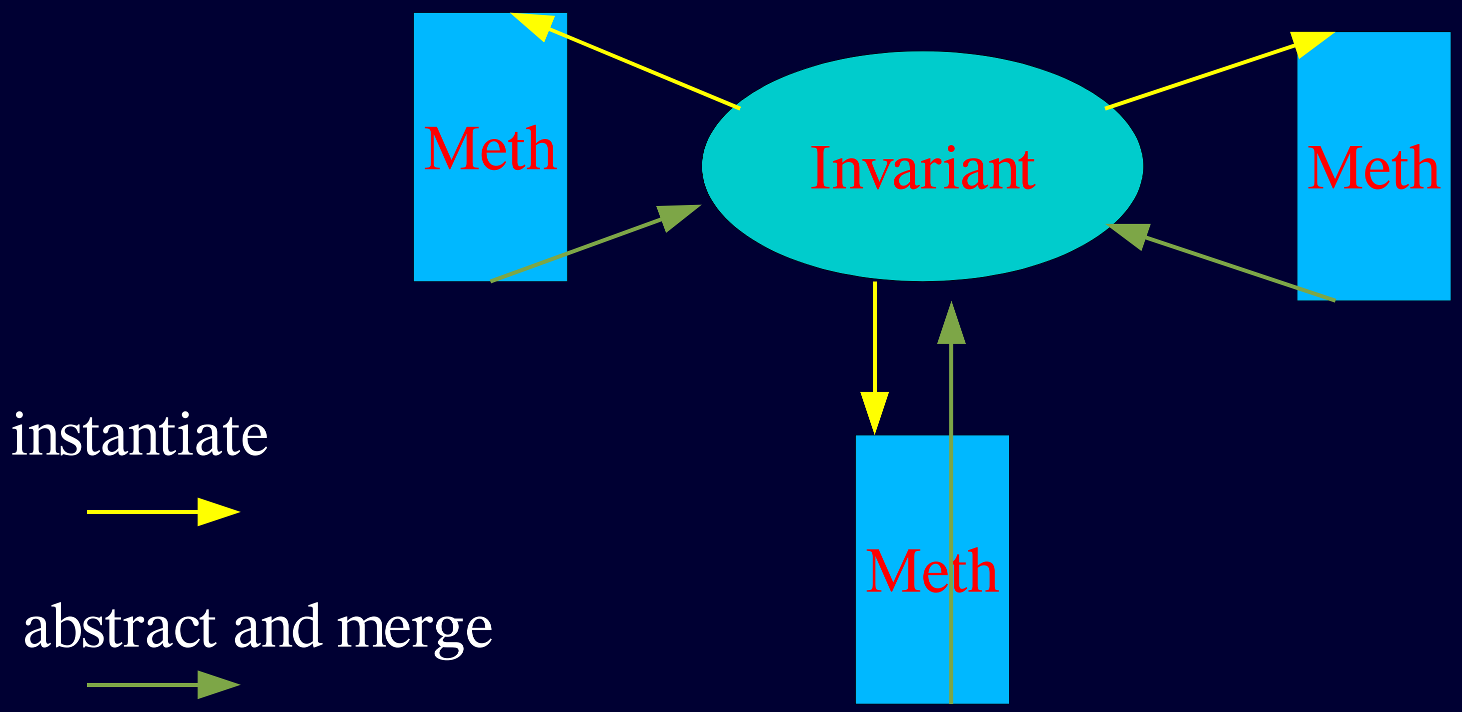
    leads to

    $\forall$ s: Set :: s.head = null

  - This is our tentative invariant.

# Invariant Inference

- Interpret methods to a fixed point:



# Invariant Inference for Set Example

- Constructor leaves us with 0-elem state.
- Run insert: get 1-elem post-state.
- Merge: Sets have 0 or 1 elements.
- Run insert again: concretize to two start states (0 and 1 element).
- Get two post states $0 \rightarrow 1$, $1 \rightarrow \{1, 2\}$.
- Merge: Sets have 0, 1, or 2 elements.
- How can we reach a fixed point?

# Ownership abstraction

- When we've elaborated the possible states enough, try to abstract out by finding an invariant that applies to all objects owned by "this."
- In our case, all Nodes *n* owned by Set *s* have
  - rc[*n*] = 1
  - *n*.next = null or else owner[*n*.next] = *s*
- This will be a fixed point: maintained by insert and delete.
- And is sufficient to justify insertion of "free".

# Other related work

- RC GC:
  - [Bacon et. al],[Levanoni&Petrank],[Blackburn&McKinley].
- Escape analysis:
  - [Park&Goldberg],[Blanchet],[Choi et al.],[Whaley&Rinard]
- Linear types:
  - [Wadler],[Baker],[Fandrich&Deline]
- Region inference: [Tofte&Birkedal]
- Shape analysis: [Sagiv&Reps&Wilhelm]
- Role analysis: [Kuncak&Lam&Rinard]
- Program verification:
  - [Detlefs et al.],[Bush&Pincus&Sielaff]
- Ownership types:
  - [Boyapati&Liskov&Shrira],[Boyapati&Lee&Rinard]

# Current Status: Can do...

- I've started an implementation.
  - Typed Guarded Commands with classes.
  - State =
    - current variable values.
    - eqNull, neqNull.
    - a general "predicate" describing other known facts.
- Can do:
  - Run constructor.
  - Run insert once, get right "invariant".
  - Concretize these states.
  - Run insert on these states.

# Current Status: Working On, To do…

- Abstracting the result state from second insert correctly:
  - Issue: predicates over variables not in scope.
    - Local vars of methods, or:
    - $\{ P \}\ x := x+1\ \{ P \wedge x = x\$0 + 1 \}$
    - Should a predicate mentioning x$0 be part of an invariant?
- To do:
  - Ownership abstraction.
  - Quantified formulas in the state. $(\forall\ n: Node :: \ldots)$
  - Making sure merge reaches a fixed point.

# Future Work, Conclusions

- Actually getting this to work :-)
- Other examples:
  - Binary trees.
  - rc[n] > 1 examples:
    - Doubly linked lists.
    - Trees with parent pointers.
- Less ad-hoc implementation:
  - Egraph for equalities.
  - Simplex for integer inequalities.

- This is a promising technique:
  - For compile-time GC.
  - For program analysis in general.