Verifying Data Structure Invariants in Device Drivers

Scott McPeak (smcpeak@cs)
George Necula (necula@cs)

Chess Review
May 10, 2004
Berkeley, CA

Chess

Motivation

- Want to verify programs that use pointers
- Need precise description of heap's shape
  - Traditional alias analysis won't do
  - Must be able to do strong updates, i.e. distinguish a particular object from the rest
- Examples
  - "This is a tree"
  - "These structures are disjoint"
  - "Node p is reachable from node q"
- BUT: Language should be simple, tractable
- Our approach: FODIL, a First-Order Data structure Invariant Language

The FODIL Language

Invariant
$$I \quad ::= \quad \text{true} \mid I \wedge Q$$
Quantifier
$$Q \quad ::= \quad \forall p.\ P$$
Predicate
$$P \quad ::= \quad A \mid \neg P \mid P \wedge P \mid P \vee P$$
Atom
$$A \quad ::= \quad T = T \mid T \neq T$$
Term
$$T \quad ::= \quad p \mid T\text{->}f$$

Full FODIL is undecidable

- Problem: lots of function symbols
- e.g., can reduce word problem:
  - given abc=def, de=s, sf=q, cba=q
  - is abc=cba ?
  - yes: abc ! def ! sf ! q ! cba
- to FODIL:
  - 8 p. p->a->b->c = p->d->e->f,
    8 p. p->d->e = p->s,
    8 p. p->s->f = p->q
  - is x->a->b->c = x->c->b->a ?

Ghost fields

- Verifier treats them like other fields
  - Added to assist description
  - Way of making global properties local
  - Like strengthening an inductive hypothesis
- Must be updated like other fields!
  - For now, this is done manually
  - But like other annotations, inference is possible
- Compiler ignores them
  - Hence, can't be inspected at run-time
  - Discarding ghost fields can be seen as an optimization

Injectivity Pattern

- Want to say: "these nodes form a tree"

```
struct Node {
  Node *left;
  Node *right;
};
```

Injectivity Pattern

- Want to say: "these nodes form a tree"
- Instead #1: "the child selector is injective"

```
struct Node {
  Node *left;
  Node *right;
};
```

`p->left` ≡ child(p, "left")
`p->right` ≡ child(p, "right")

Injectivity Pattern

- Want to say: "these nodes form a tree"
- Instead #1: "the child selector is injective"
- Instead #2: "child selector has an inverse"

```
struct Node {
  Node *left;
  Node *right;
  Node *parent;
  bool isLeft;
};
```

`p->left` ≡ child(p, true)
`p->right` ≡ child(p, false)

`p->parent` ≡ fst(child^{-1}(p))
`p->isLeft` ≡ snd(child^{-1}(p))

8 p. `p->left->isLeft == true &&`
`p->left->parent == p;`

Transitivity Pattern

- Want to say that all reachable nodes have some property
- Instead, associate the property with a ghost field
- Then say neighbor nodes' fields are equal

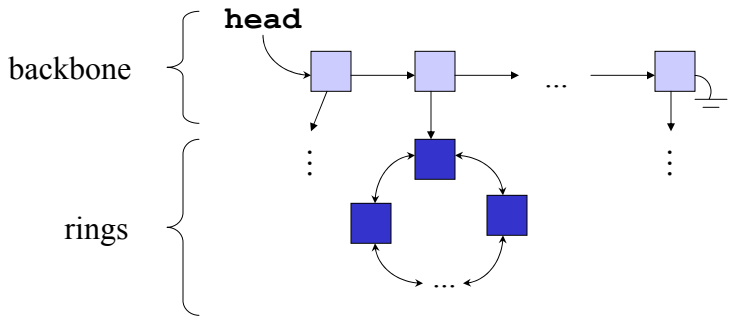
```
struct Node {
  Node *next;
  Node *head;
};
```

8 p. `p->next->head == p->head;`

Dynamic types

- Every static type has a corresponding dynamic type *tag*
- Every structure has a (ghost) tag field
- malloc sets the tag to the proper value
- free sets the tag to zero
- An object must have the proper tag for a field access to be safe (i.e. not a dangling reference)
- NULL's tag is zero, so a nonzero tag implies a pointer is not NULL

Example: Linked list of circular lists



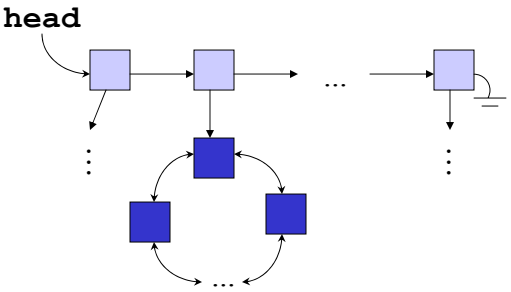
Example: Linked list of circular lists

```
struct BNode {
  BNode *next;
  BNode *prev;

  RNode *ring;
};

struct RNode {
  BNode *bnode;

  RNode *next;
  RNode *prev;
};
```



Example: Linked list of circular lists

```
struct BNode {
  BNode *next;
  BNode *prev;

  RNode *ring;
};

struct RNode {
  BNode *bnode;

  RNode *next;
  RNode *prev;
};
```

```
forall(BNode *b) {
  b->next != NULL ==>
    b->next->prev == b;        inj
  b->ring->tag == RNode;
  b->ring->bnode == b;         inj
}

forall(RNode *r) {
  r->next->tag == RNode;
  r->prev->tag == RNode;
  r->next->prev == r;          inj
  r->prev->next == r;          inj
  r->next->bnode == r->bnode;
};                             trans
```


Verification: deallocNode()

```
deallocNode(...) {
  for (BNode *b = head; b; b = b->next) {
    if (...) {
      RNode *r = b->ring;
      do {
        if (... && r != r->next) {
          // remove 'r' from its ring
          r->prev->next = r->next;
          r->next->prev = r->prev;
          if (r->bnode == b)
            b->ring = r->next;
          free(r); return;
        }
        r = r->next;
      } while (r != b->ring); }}}
```

Proof: No dangling references

Given:
- invariant held to begin with;
- r->bnode = b;
- b->ring ≠ r;
- r->tag = 0;

Goal:
- 8 b. b->ring->tag = RNode

:Goal, instantiated with fresh var:
- c->ring->tag ≠ RNode

i.e.
- c->ring->($tag_0\{r \mapsto 0\}$) ≠ RNode

```
forall(BNode *b) {
  ...
  b->ring->tag == RNode;
  b->ring->bnode == b;
}
```

If c->ring = r:
- then r->bnode = c
- so b = c, contr.

If c->ring ≠ r:
- then c->ring->tag_0 ≠ RNode
- contradicts orig. invariant

Decision procedure

- Key question: when to instantiate universally quantified facts?
- Our answer (for now): ad-hoc matching
  - 8 p. p->a->b = p, match on "p->a"
  - 8 p. p->a->b = p->b, match on "p->a" or "p->b"
- For these cases we can prove completeness
  - Relies on detailed reasoning about the e-dag, a data structure used by the theorem prover
- Open question: more general strategy?
  - Have explored variation of Knuth-Bendix completion, still unclear if it can work

Experimental Results

- Verified two linux drivers (~1kloc each)
  - scull: Rubini example, complicated data str.
  - pc_keyb: PC keyboard + mouse driver
- Verified several data structure kernels
  - lists, arrays, etc.
  - red-black trees
  - b+-trees (including balance + key properties)
- Annotation effort metrics
  - Between 50 and 100% of original code size
  - Takes time to learn how the code works

Related: Shape Types

- Fradet and Métayer POPL97
- Formalism using graph grammars
- Doubly-linked list:
  - Doubly ::= head x, pred x NULL, L x
    L x ::= next x y, pred y x, L y
    | next x NULL
- Undecidable in general (like FODIL); but practical decidable subset not apparent
- Arguably less natural …
- All examples in their paper are expressible in FODIL (with inj+trans only)

Related: Graph Types

- Moller and Schwartzbach PLDI01, Klarlund and Schwartzbach POPL93
- Invariants expressed as quantified formulas
- Notion of trees is built into their logic; i.e. injectivity is implicit (no circular lists..)
- Uses regular expressions to describe non-tree pointers' targets
- We can reduce *deterministic* graph types to FODIL (with inj+trans only)

Related: 3-Valued Logic (TVLA)

- (e.g.) Sagiv et. al TOPLAS02
- Abstract interpretation; heap abstraction has yes/no/maybe pointers ("3-Valued")
- Requires *instrumentation predicates*
  - Supplied by programmer, defined in terms of other fields, predicates
  - Many similarities to global invariants of ghost fields
- Approach favors automation over precision
- Not obvious how to extend (e.g. to specify a tree is balanced)

Future Work

- Generalize decidable FODIL forms
- More atomic predicates: partial orders, …
- Change isolation; some connections to bunched implication
  - e.g.: ok for module A to call into module B while A's invariant is broken, if B can't see it
- Annotation automation/inference
  - Existing invariant inference is simple, effective
  - Want annotation abstractions: "this kind of loop always has these invariants: …"
- More sophisticated proof failure diagnosis

Conclusion

- Device drivers use the heap nontrivially; must characterize that use precisely
- Injectivity and transitivity are key concepts in data structure description
- We can describe them using simple quantified equalities
  - No need to add trees or transitive closure to the logic
  - Ghost fields are a more tractable alternative, making global properties expressible locally