# Verifying the Composite Pattern using Separation Logic


Bart Jacobs

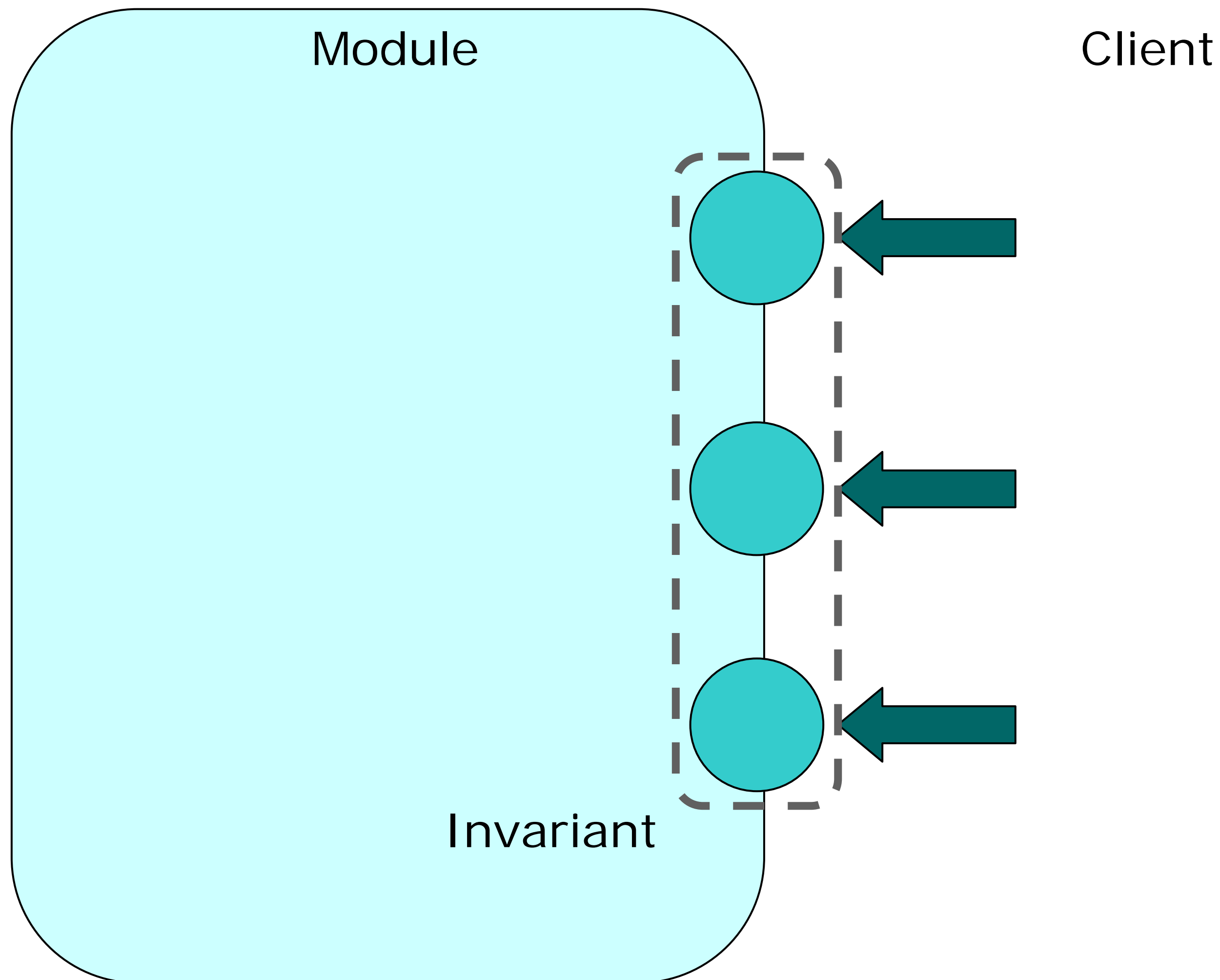
Jan Smans

Frank Piessens

*Katholieke Universiteit Leuven, Belgium*

- General Idea
- Example: Binary Tree
  - Interface
  - Client
  - Specification
  - Client Proof
  - Implementation and Implementation Proof
  - Non-contiguous Focus Changes
- Demonstration
- Conclusion

# General Idea

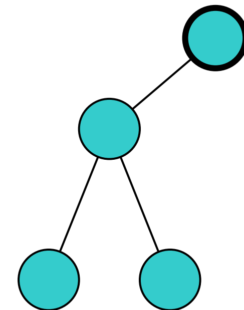


# Example: Binary Tree Interface

$$
\begin{array}{l}
\textbf{struct}\ node;\\
\textbf{typedef struct}\ node\ {*}node;\\
\\
node\ create_tree();\\
node\ tree_add_left(node\ node);\\
node\ tree_add_right(node\ node);\\
\textbf{int}\ tree_get_count(node\ node);\\
node\ tree_get_parent(node\ node);\\
\textbf{void}\ tree_dispose(node\ node);\\
\ldots
\end{array}
$$

# Example: Binary Tree Client

```
int main(void)
{
    node node := create_tree();
    node := tree_add_left(node);
    node := tree_add_right(node);
    node := tree_get_parent(node);
    node := tree_add_left(node);
    node := tree_get_parent(node);
    node := tree_get_parent(node);
    assert(tree_get_count(node) = 4);
    tree_dispose(node);
    return 0;
}
```



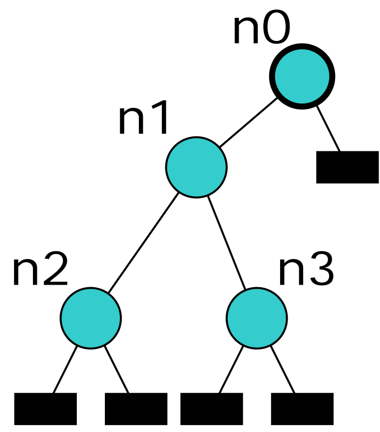
# Example: Binary Tree Specification

```
struct node;
typedef struct node *node;

inductive tree := nil | tree(node, tree, tree);

fixpoint int count(tree t) {
  switch (t) {
    case nil : return 0;
    case tree(n, l, r) : return 1 + count(l) + count(r);
  }
}

inductive context :=
  | root
  | left_context(context, node, tree)
  | right_context(context, node, tree);

predicate tree(node node, context c, tree subtree);

node create_tree();
  requires emp;
  ensures tree(result, root, tree(result, nil, nil));
```

```
node tree_add_left(node node);
  requires
    tree(node, ?c, ?t) *
    switch (t) {
      case nil : false;
      case tree(n0, l, r) : l = nil;
    };
  ensures
    switch (t) {
      case nil : false;
      case tree(n0, l, r) :
        tree(result, left_context(c, node, r),
          tree(result, nil, nil));
    };

node tree_add_right(node node);
  ... analogous ...

int tree_get_count(node node);
  requires tree(node, ?c, ?t);
  ensures tree(node, c, t) * result = count(t);

node tree_get_parent(node node);
  requires tree(node, ?c, ?t) * c ≠ root * t ≠ nil;
  ensures
    switch (c) {
      case root : false;
      case left_context(pns, p, r) :
        result = p * tree(p, pns, tree(p, t, r));
      case right_context(pns, p, l) :
        result = p * tree(p, pns, tree(p, l, t));
    };

void tree_dispose(node node);
  requires tree(node, root, _);
  ensures emp;
```

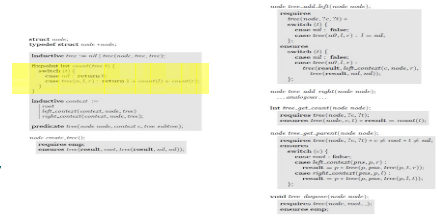
# Example: Binary Tree Specification: Datatype *tree*

$$\textbf{inductive}\ \mathit{tree} := \mathit{nil} \mid \mathit{tree}(\mathit{node}, \mathit{tree}, \mathit{tree});$$

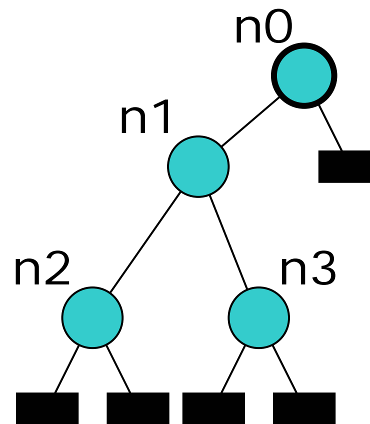



tree(n0, tree(n1, tree(n2, nil, nil), tree(n3, nil, nil)), nil)

# Example: Binary Tree Spec'n: Pure function *count*



```
fixpoint int count(tree t) {
  switch (t) {
    case nil : return 0;
    case tree(n, l, r) : return 1 + count(l) + count(r);
  }
}
```

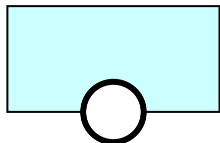



count(tree(n0, tree(n1, tree(n2, nil, nil), tree(n3, nil, nil)), nil)) = 4

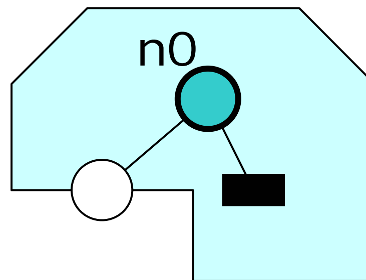
# Example: Binary Tree Spec'n: Datatype *context*

$$\begin{aligned}&\textbf{inductive}\ context\ := \\ &\quad |\ root \\ &\quad |\ left_context(context, node, tree) \\ &\quad |\ right_context(context, node, tree);\end{aligned}$$

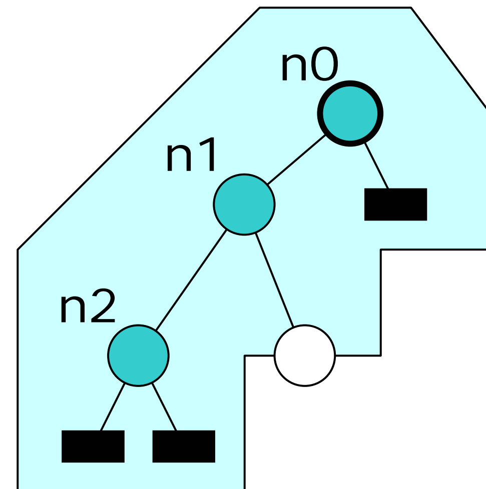
root



left_context(root, n0, nil)

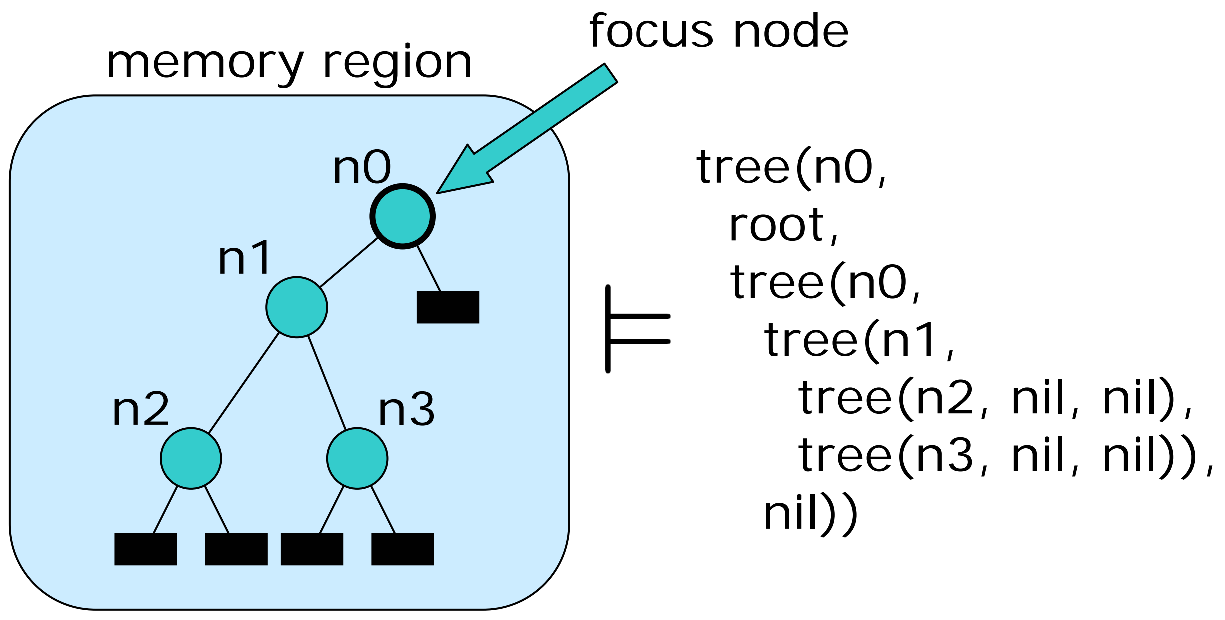



right_context(left_context(root, n0, nil), n1, tree(n2, nil, nil))



# Example: Binary Tree Spec'n: Predicate *tree*

$$\textbf{predicate}\ tree(node\ node,\ context\ c,\ tree\ subtree);$$



# Example: Binary Tree Spec'n: Predicate *tree*

$$\textbf{predicate}\ tree(node\ node,\ context\ c,\ tree\ subtree);$$

memory region

focus node

n0

n1

n2

n3

$\models$

tree(n1,
 left_context(root, n0, nil),
 tree(n1,
  tree(n2, nil, nil),
  tree(n3, nil, nil)))

# Example: Binary Tree Spec'n: Predicate *tree*

$$\textbf{predicate}\ tree(node\ node, context\ c, tree\ subtree);$$

memory region

n0

n1

n2

n3

focus node

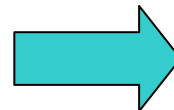
$\models$

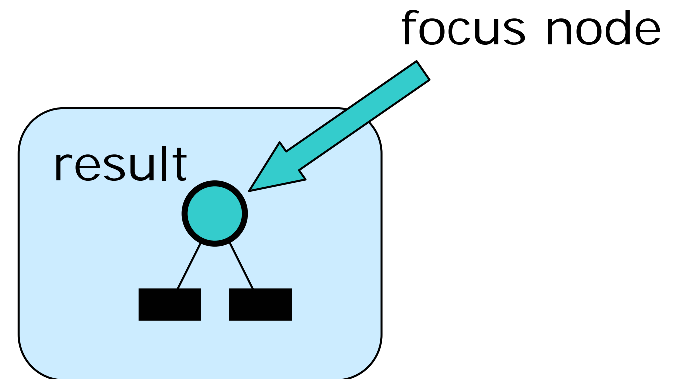
```
tree(n3,
  right_context(
    left_context(root, n0, nil),
    n1,
    tree(n2, nil, nil)),
  tree(n3, nil, nil))
```

# Example: Binary Tree Spec'n: Function *create_tree*

$$node\ create_tree();$$

**requires emp**;
**ensures** $tree(\textbf{result}, root, tree(\textbf{result}, nil, nil))$;

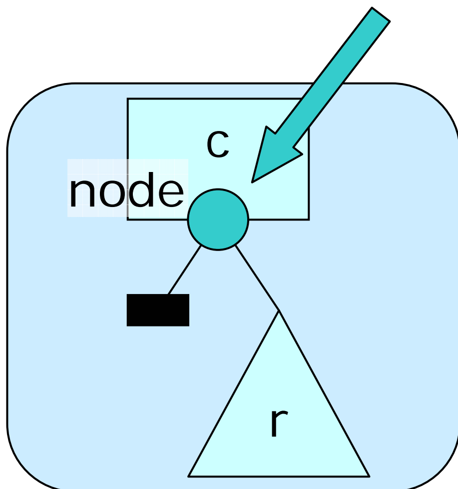


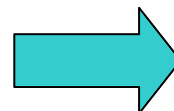


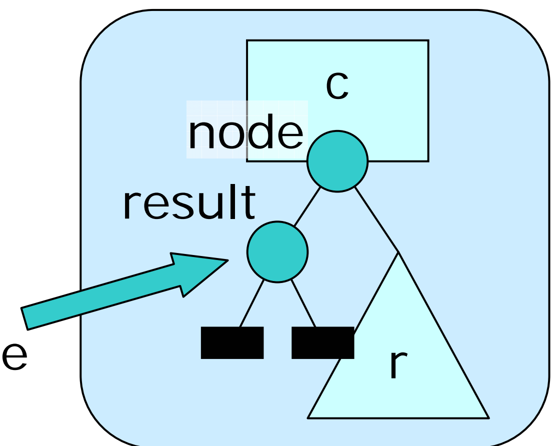
# Example: Binary Tree Spec'n: Function *tree_add_left*

$node\ tree_add_left(node\ node);$

**requires**
  $tree(node, ?c, ?t) *$
  **switch** $(t)$ {
    **case** $nil$ : **false**;
    **case** $tree(n0, l, r)$ : $l = nil$;
  };
**ensures**
  **switch** $(t)$ {
    **case** $nil$ : **false**;
    **case** $tree(n0, l, r)$ :
      $tree($**result**$, left_context(c, node, r),$
        $tree($**result**$, nil, nil));$
  };



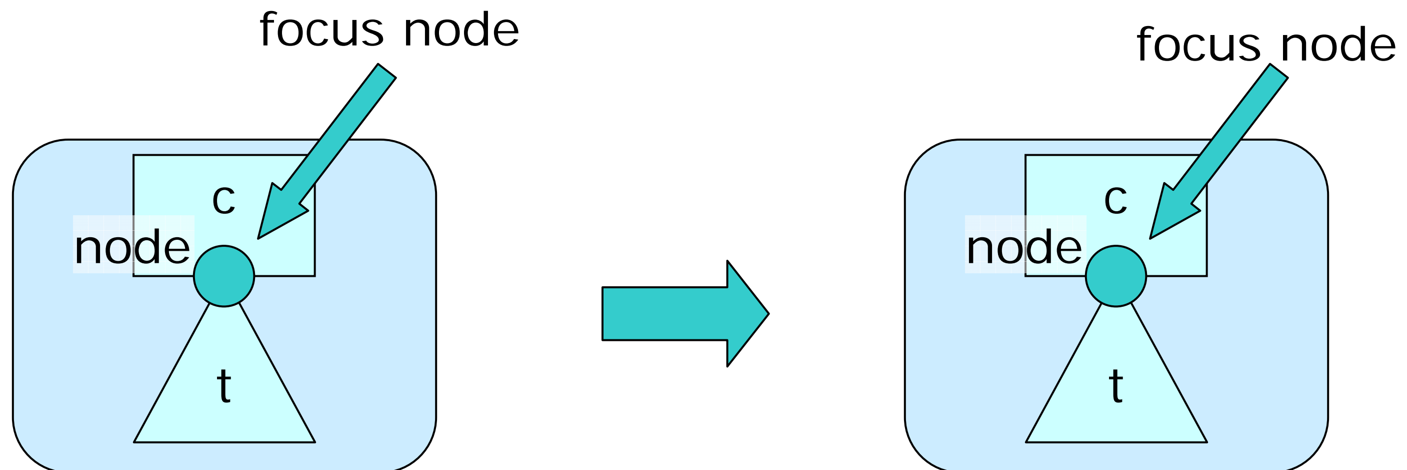





# Example: Binary Tree Spec'n: Function *tree_get_count*

$$\textbf{int}\ tree_get_count(node\ node);$$

$$\textbf{requires}\ tree(node, ?c, ?t);$$
$$\textbf{ensures}\ tree(node, c, t) * \textbf{result} = count(t);$$

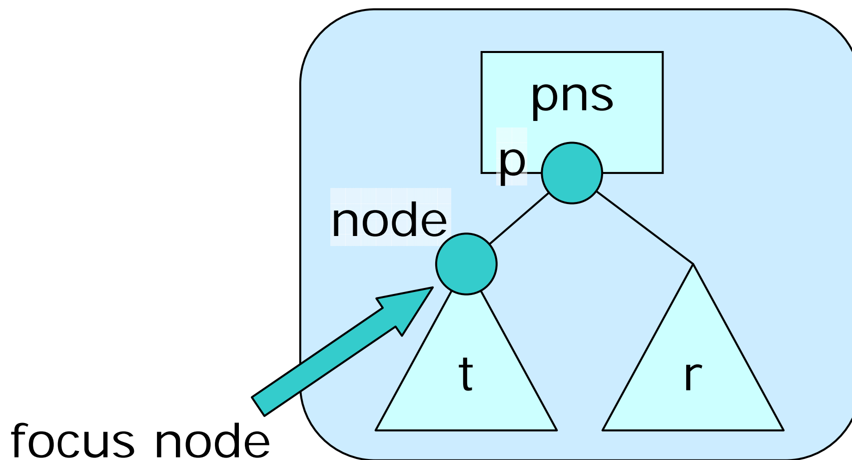


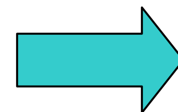
# Example: Binary Tree Spec'n: Function *tree_get_parent*

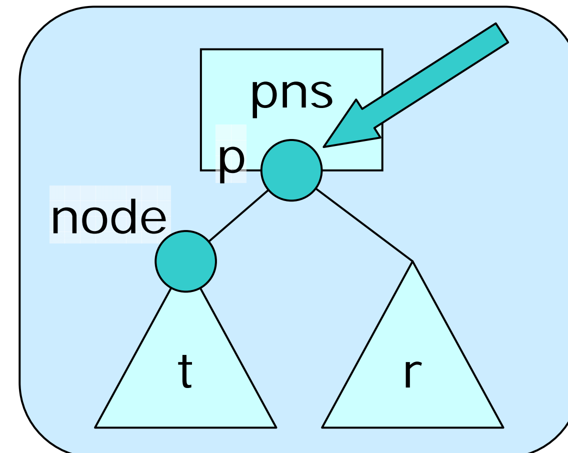
$node\ tree_get_parent(node\ node);$

**requires** $tree(node, ?c, ?t) * c \neq root * t \neq nil;$
**ensures**
  **switch** $(c)$ {
    **case** $root$ : **false**;
    **case** $left_context(pns, p, r)$ :
      **result** $= p * tree(p, pns, tree(p, t, r));$
    **case** $right_context(pns, p, l)$ :
      **result** $= p * tree(p, pns, tree(p, l, t));$
  };

(case left_context)



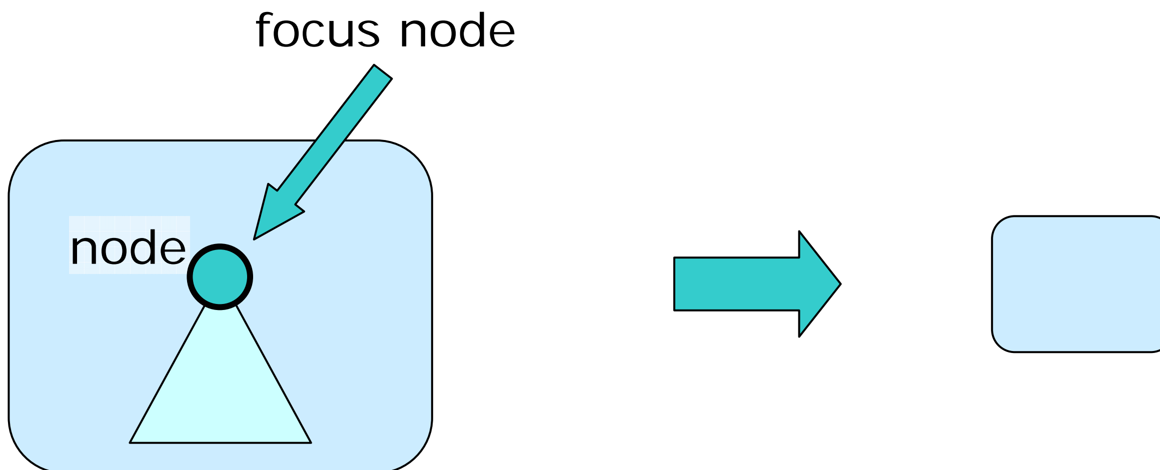





# Example: Binary Tree Spec'n: Function *tree_dispose*

$$\textbf{void}\ \mathit{tree_dispose}(\mathit{node}\ \mathit{node});$$

$$\textbf{requires}\ \mathit{tree}(\mathit{node}, \mathit{root}, _);$$
$$\textbf{ensures}\ \textbf{emp};$$



# Example: Binary Tree Client Proof

```
int main()
    requires emp;
    ensures emp;
{
    // {}
    node node := create_tree();
    // {tree(n0, root, tree(n0, nil, nil))}
    node := tree_add_left(node);
    // {tree(n1, left_context(root, n0, nil), tree(n1, nil, nil))}
    node := tree_add_right(node);
    // {tree(n2, right_context(left_context(root, n0, nil), n1,
    //     nil), tree(n2, nil, nil))}
    node := tree_get_parent(node);
    // {tree(n1, left_context(root, n0, nil),
    //     tree(n1, nil, tree(n2, nil, nil)))}
    node := tree_add_left(node);
    // {tree(n3, left_context(left_context(root, n0, nil), n1,
    //     tree(n2, nil, nil)), tree(n3, nil, nil))}
    node := tree_get_parent(node);
    // {tree(n1, left_context(root, n0, nil), tree(n1,
    //     tree(n3, nil, nil), tree(n2, nil, nil)))}
    node := tree_get_parent(node);
    // {tree(n0, root, tree(n0, tree(n1, tree(n3,
    //     nil, nil), tree(n2, nil, nil)), nil))}
    tree_dispose(node);
    // {}
    return 0;
}
```

# Example: Binary Tree Implementation: struct *node*

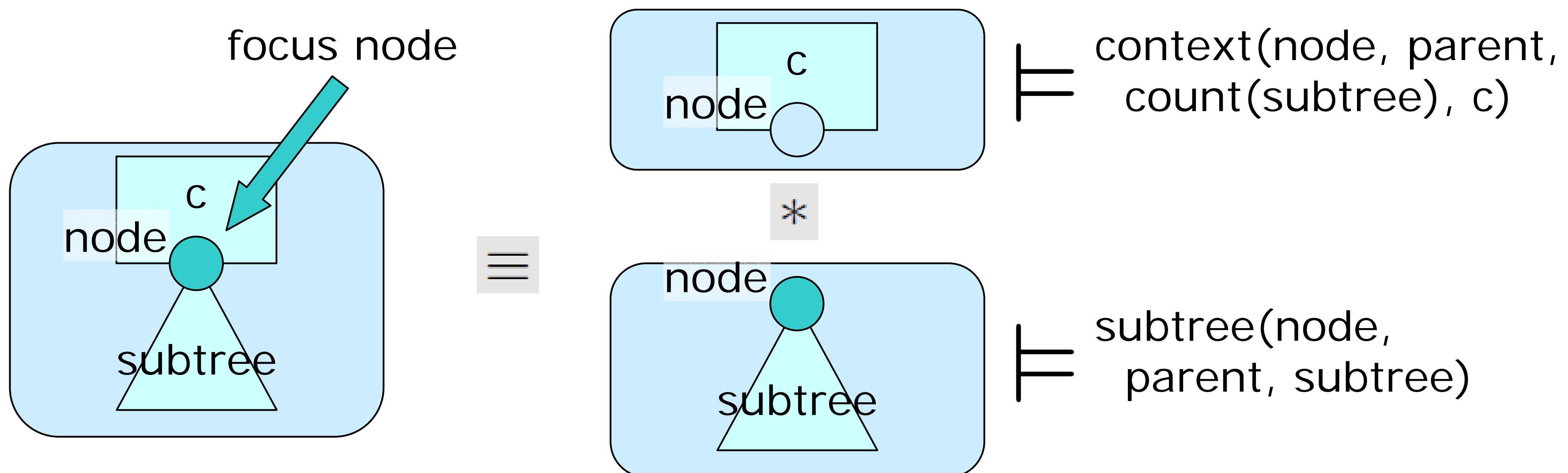
```
struct node {
    struct node *left;
    struct node *right;
    struct node *parent;
    int count;
};
```

# Example: Binary Tree Implementation: predicate *tree*

$$\textbf{predicate}\ subtree(node\ root, node\ parent, tree\ t)$$

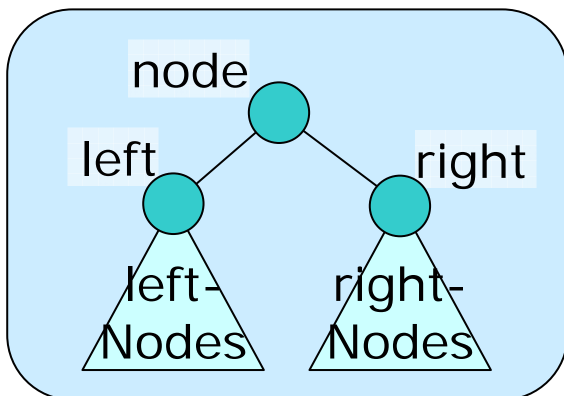
$$\textbf{predicate}\ context(node\ n, node\ p, \textbf{int}\ count, context\ c)$$

$$\begin{array}{l}\textbf{predicate}\ tree(node\ node, context\ c, tree\ subtree) \equiv \\ \quad context(node, ?parent, count(subtree), c) * \\ \quad subtree(node, parent, subtree);\end{array}$$

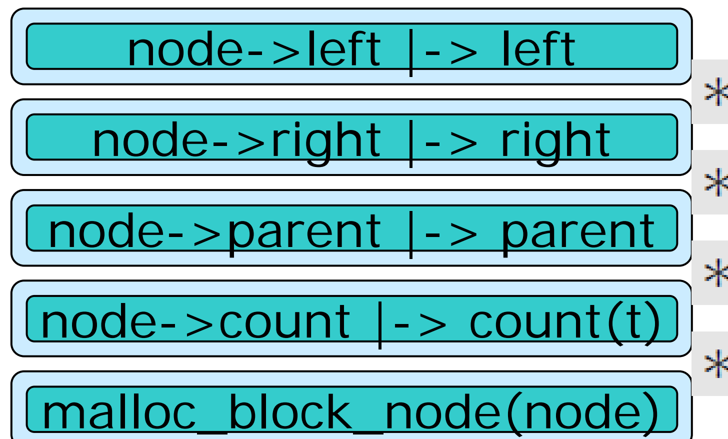


# Example: Binary Tree Implementation: predicate *subtree*

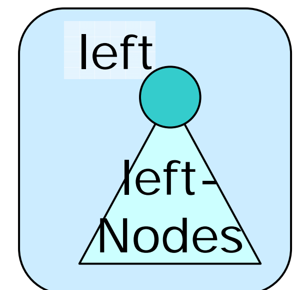
$$
\begin{array}{l}
\textbf{predicate}\ \mathit{subtree}(\mathit{node\ node}, \mathit{node\ parent}, \mathit{tree\ t})\ \equiv \\
\quad \textbf{switch}\ (t)\ \{ \\
\quad\quad \textbf{case}\ \mathit{nil} : \mathit{node} = 0; \\
\quad\quad \textbf{case}\ \mathit{tree}(\mathit{node0}, \mathit{leftNodes}, \mathit{rightNodes}) : \\
\quad\quad\quad \mathit{node} = \mathit{node0} * \mathit{node} \neq 0 * \\
\quad\quad\quad \mathit{node}{\rightarrow}\mathit{left} \mapsto ?\mathit{left} * \\
\quad\quad\quad \mathit{node}{\rightarrow}\mathit{right} \mapsto ?\mathit{right} * \\
\quad\quad\quad \mathit{node}{\rightarrow}\mathit{parent} \mapsto \mathit{parent} * \\
\quad\quad\quad \mathit{node}{\rightarrow}\mathit{count} \mapsto \mathit{count}(t) * \\
\quad\quad\quad \mathit{malloc_block_node}(\mathit{node}) * \\
\quad\quad\quad \mathit{subtree}(\mathit{left}, \mathit{node}, \mathit{leftNodes}) * \\
\quad\quad\quad \mathit{subtree}(\mathit{right}, \mathit{node}, \mathit{rightNodes}); \\
\quad \}; \\
\end{array}
$$



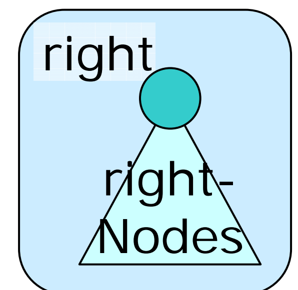

≡




*

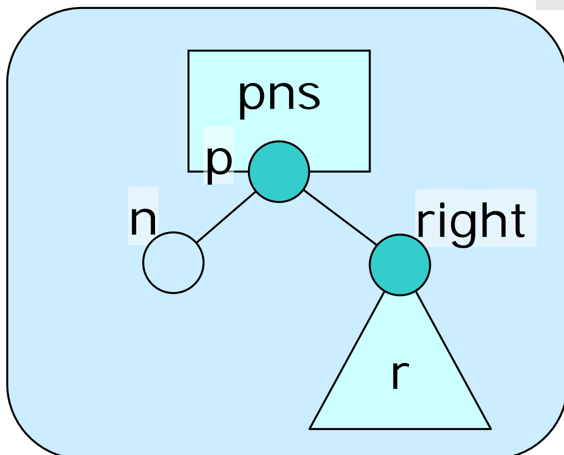



*

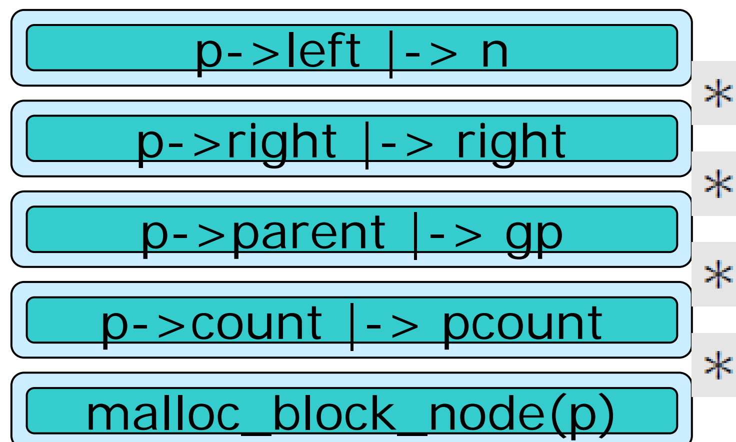


# Example: Binary Tree Implementation: predicate *context*

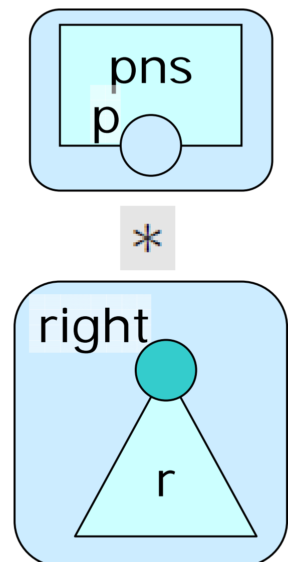
**predicate** $context(node\ n, node\ p,$ **int** $count, context\ c) \equiv$
  **switch** $(c)$ {
    **case** $root:\ p = 0;$
    **case** $left_context(pns, p0, r):$
      $p = p0 * p \neq 0 *$
      $p \to left \mapsto n *$
      $p \to right \mapsto ?right *$
      $p \to parent \mapsto ?gp *$
      $p \to count \mapsto ?pcount *$
      $malloc_block_node(p) *$
      $context(p, gp, pcount, pns) *$
      $subtree(right, p, r) *$
      $pcount = 1 + count + count(r);$
    **case** $right_context(pns, p0, l):$
      $\ldots analogous \ldots$
  };




≡




*



# Example: Binary Tree Implementation: function *create_tree*

```
node create_node(node p)
    requires emp;
    ensures subtree(result, p, tree(result, nil, nil));
{
    node n := malloc(sizeof(struct node));
    n→left := 0;  close subtree(0, n, nil);
    n→right := 0;  close subtree(0, n, nil);
    n→parent := p;
    n→count := 1;
    close subtree(n, p, tree(n, nil, nil));
    return n;
}

node create_tree()
    requires emp;
    ensures tree(result, root, tree(result, nil, nil));
{
    node n := create_node(0);
    close context(n, 0, 1, root);
    close tree(n, root, tree(n, nil, nil));
    return n;
}
```

# Example: Binary Tree Impl'n: function *subtree_get_count*

```
int subtree_get_count(node node)
    requires subtree(node, ?parent, ?nodes);
    ensures subtree(node, parent, nodes) *
        result = count(nodes);
{
    int result := 0;
    open subtree(node, parent, nodes);
    if (node ≠ 0) { result := node→count; }
    close subtree(node, parent, nodes);
    return result;
}
```

# Example: Binary Tree
# Impl'n: function *fixup_ancestors*

```
void fixup_ancestors(node n, node p, int count)
    requires context(n, p, _, ?c);
    ensures context(n, p, count, c);
{
    open context(n, p, _, c);
    if (p ≠ 0) {
        node left := p→left;
        node right := p→right;
        node grandparent := p→parent;
        int leftCount := 0;
        int rightCount := 0;
        if (n = left) {
            leftCount := count;
            rightCount := subtree_get_count(right);
        } else {
            leftCount := subtree_get_count(left);
            rightCount := count;
        }
        {
            int pcount := 1 + leftCount + rightCount;
            p→count := pcount;
            fixup_ancestors(p, grandparent, pcount);
        }
    }
    close context(n, p, count, c);
}
```

# Example: Binary Tree Impl'n: function *tree_add_left*

```
node tree_add_left(node node)
  requires
    tree(node, ?c, ?t) *
    switch (t) {
      case nil : false;
      case tree(n0, l, r) : l = nil;
    };
  ensures
    switch (t) {
      case nil : false;
      case tree(n0, l, r) :
        tree(result, left_context(c, node, r),
          tree(result, nil, nil));
    };
{
  open tree(node, c, t);
  node n := create_node(node);
  open subtree(node, ?parent, t);
  node nodeRight := node→right;
  assert subtree(nodeRight, node, ?r);
  {
    node nodeLeft := node→left;
    open subtree(nodeLeft, node, nil);
    node→left := n;
    close context(n, node, 0, left_context(c, node, r));
    fixup_ancestors(n, node, 1);
  }
  close tree(n, left_context(c, node, r), tree(n, nil, nil));
  return n;
}
```

# Example: Binary Tree
# Impl'n: function *tree_get_count*

$$
\begin{array}{l}
\textbf{int}\ \mathit{tree_get_count}(\mathit{node}\ n) \\
\quad \textbf{requires}\ \mathit{tree}(n, ?c, ?t); \\
\quad \textbf{ensures}\ \mathit{tree}(n, c, t) * \textbf{result} = \mathit{count}(t); \\
\{ \\
\quad \textbf{open}\ \mathit{tree}(n, c, t); \\
\quad \textbf{int}\ \mathit{result} := \mathit{subtree_get_count}(n); \\
\quad \textbf{close}\ \mathit{tree}(n, c, t); \\
\quad \textbf{return}\ \mathit{result}; \\
\}
\end{array}
$$

# Example: Binary Tree Impl'n: function *tree_get_parent*

```
node tree_get_parent(node node)
  requires tree(node, ?c, ?t) * c ≠ root * t ≠ nil;
  ensures
    switch (c) {
      case root : false;
      case left_context(pns, p, r) :
        result = p * tree(p, pns, tree(p, t, r));
      case right_context(pns, p, r) :
        result = p * tree(p, pns, tree(p, l, t));
    };
{
   open tree(node, c, t);
   open subtree(node, _, t);
  node p := node→parent;
   close subtree(node, p, t);
   open context(node, p, count(t), c);
   assert context(p, ?gp, ?pcount, ?pns);
   switch (c) {
     case root :
     case left_context(pns, p0, r) :
       close subtree(p, gp, tree(p, t, r));
     case right_context(pns, p0, l) :
       close subtree(p, gp, tree(p, l, t));
   }
   assert subtree(p, gp, ?pt);
   close tree(p, pns, pt);
  return p;
}
```

# Overview

- General Idea
- Example: Binary Tree
  - Interface
  - Client
  - Specification
  - Client Proof
  - Implementation and Implementation Proof
  - Non-contiguous Focus Changes
- Demonstration
- Conclusion

# Non-contiguous Focus Changes Example Client Program

```
int main()
{
    node root := create_tree();
    node l := tree_add_left(root);
    node lr := tree_add_right(l);
    node ll := tree_add_left(l);
    node lrr := tree_add_right(lr);
    tree_dispose(root);
    return 0;
}
```

# Non-contiguous Focus Changes
# Additional Specification Elements

**fixpoint** *tree combine*(*context c*, *tree t*) { . . . }
**inductive** *path* := *here* | *left*(*path*) | *right*(*path*);
**fixpoint bool** *contains_at*(*tree t*, *path p*, *node n*) { . . . }
**fixpoint** *context context_at*(*context c*, *tree t*, *path p*) { . . . }
**fixpoint** *tree subtree_at*(*tree t*, *path p*)) { . . . }

**lemma void** *change_focus*(*node n0*, *path p*, *node n*);
  **requires** *tree*(*n0*, ?*c*, ?*t*) ∗ *contains_at*(*combine*(*c*, *t*), *p*, *n*);
  **ensures** *tree*(*n*, *context_at*(*root*, *combine*(*c*, *t*), *p*),
    *subtree_at*(*combine*(*c*, *t*), *p*));

# Non-contiguous Focus Changes Proof of Example Client Program

```
int main()
    requires emp;
    ensures emp;
{
    node root := create_tree();
    node l := tree_add_left(root);
    node lr := tree_add_right(l);
    change_focus(lr, left(here), l);
    node ll := tree_add_left(l);
    change_focus(ll, left(right(here)), lr);
    node lrr := tree_add_right(lr);
    change_focus(lrr, here, root);
    tree_dispose(root);
    return 0;
}
```

# Non-contiguous Focus Changes Proof of lemma *change_focus*

```
lemma void go_to_root(node n, context c)
  requires tree(n, c, ?t);
  ensures tree(_, root, combine(c, t));
{
  switch (c) {
    case root :
    case left_context(pcn, p, r) :
      open tree(n, c, t);
      open context(n, _, _, _);
      assert context(p, ?gp, _, _);
      close subtree(p, gp, tree(p, t, r));
      go_to_root(p, pcn);
    case right_context(···) : ... analogous ...
  }
}

lemma void go_to_descendant(node n0, path p, node n)
  requires tree(n0, ?c, ?t) * contains_at(t, p, n);
  ensures tree(n, context_at(c, t, p), subtree_at(t, p));
{ ... }

lemma void change_focus(node n0, path p, node n)
  requires tree(n0, ?c, ?t) * contains_at(combine(c, t), p, n);
  ensures tree(n, context_at(root, combine(c, t), p),
    subtree_at(combine(c, t), p));
{
  go_to_root(c);
  assert tree(?rootNode, _, _);
  go_to_descendant(rootNode, p, n);
}
```

# Overview

- General Idea
- Example: Binary Tree
  - Interface
  - Client
  - Specification
  - Client Proof
  - Implementation and Implementation Proof
  - Non-contiguous Focus Changes
- Demonstration
- Conclusion

- General Idea
- Example: Binary Tree
  - Interface
  - Client
  - Specification
  - Client Proof
  - Implementation and Implementation Proof
  - Non-contiguous Focus Changes
- Demonstration
- Conclusion

# Conclusion

- Approach:
  - Structure = 1 seplogic predicate
  - In proof: Separate out focus node
  - In client: Change focus node using lemma
- VeriFast:

  www.cs.kuleuven.be/~bartj/verifast