Automated Verification via Separation Logic

Cristina David

National University of Singapore

Advisor: Prof. Chin Wei-Ngan

Seoul National University, October 2009

Overview

- **Verification of shape, size and bag props**
  - Automated Verification of Shape, Size and Bag Properties, W.N.Chin, C.David, H.H.Nguyen, S.Qin, ICECCS'07
  - Automated Verification of Shape and Size Properties via Separation Logic, H.H.Nguyen, C.David, S.Qin, W.N.Chin, VMCAI'07
  - Multiple Pre/Post Specifications for Heap-Manipulating Methods, W.N.Chin, C.David, H.H.Nguyen, S.Qin, HASE'07

- **Verification of object-oriented programs**
  - Enhancing Modular OO Verification with Separation Logic, W.N.Chin, C.David, H.H.Nguyen, S.Qin, POPL'08

- **Verification of exception-handling programs**
  - Translation and Optimization for a Core Calculus with Exceptions, C.David, C,Gherghina, W.N.Chin, PEPM'09

Goal

- Verifying mutable data structures with invariants involving size properties

Challenges

- Strong updates in the presence of aliasing
- Entailment checking with inductive predicates

Separation logic

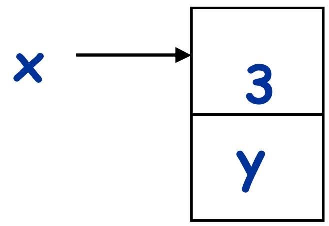
- Foundations:
  - O'Hearn and Pym, "The Logic of Bunched Implications", Bulletin of Symbolic Logic 1999
  - Reynolds, "Separation Logic: A Logic for Shared Mutable Data Structures", LICS 2002

- Extension to Hoare logic to reason about shared mutable data structures

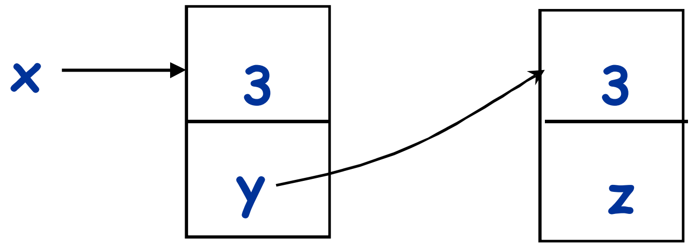
Examples (Reynolds LICS'02)

$p_1 * p_2$: the heap can be split into two disjoint parts (p_1 holds for one part and p_2 holds for the other)

$x \mapsto 3, y$



$x \mapsto 3, y * y \mapsto 3, z$



Local reasoning in separation logic

- Frame Rule

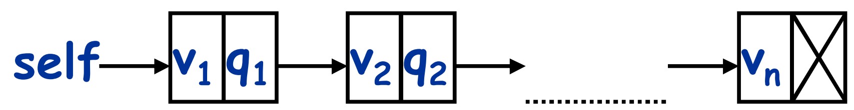
$$\frac{\{p\}\ c\ \{q\}}{\{p * r\}\ c\ \{q * r\}}$$

Reasoning about command *c* involves only parts of the heap that are actually used by *c*.

Our work

- Size properties
  - Data structure invariants involving arithmetic constraints.
  - Sortedness, length, height-balanced, etc.
- User-defined inductive predicates
  - Data structures are user custom design

Example predicates



$self::node\langle v_1,q_1\rangle * q_1::node\langle v_2,q_2\rangle$
$*\ldots*q_{n-1}::node\langle v_n,null\rangle$

Singly-linked list:

$ll\langle n\rangle \equiv self=null \wedge n=0$
$\vee\ \ self::node\langle\_,q\rangle * q::ll\langle n-1\rangle$

Non-empty sorted list:

$sortl\langle n,min\rangle \equiv self::node\langle min,null\rangle \wedge n=1$
$\vee\ \ self::node\langle min,q\rangle * q::sortl\langle n-1,k\rangle \wedge min \leq k$

Singly-linked list with bag of values:

$llB\langle B\rangle \equiv self=null \wedge B=\{\}$
$\vee\ \ self::node\langle v,q\rangle * q::llB\langle B_1\rangle \wedge B = B_1 \cup \{v\}$

What can be written?

- Heap part
  - Describes shapes using separation logic
- Pure part
  - Size properties (arithmetic constraints)
  - Pointer constraints
  - Bag constraints
- self is a pointer from which every node is reachable

Verification

- Methods are annotated with pre and postconditions
- Loop invariants are supplied
- Entailment checks
  - Precondition at call site
  - Postcondition at end of method

Insert Into a Sorted List

```
node insert(node x, node vn)
requires x points to a sorted list with length n and vn points to
a disjoint node
ensures res points to a sorted list with length n+1
{
        if (vn.val≤x.val) {
                vn.next = x;
                return vn; }
        else if (x.next=null) then {
                x.next = vn; vn.next = null;
                return x; }
        else {
                x.next = insert(x.next, vn);
                return x; }
}
```


12

Insert Into a Sorted List

```
node insert(node x, node vn)
requires x::sortl⟨n, sm⟩ * vn::node⟨v,_⟩
ensures res::sortl⟨n+1, min(v, sm)⟩

{
      if (vn.val≤x.val) {
            vn.next = x;
            return vn; }
      else if (x.next=null) then {
            x.next = vn; vn.next = null;
            return x; }
      else {
            x.next = insert(x.next, vn);
            return x; }
}
```

Sorted List with Bag of Values

- $\text{sortB}\langle B\rangle \equiv \text{self=null} \wedge B=\{\}$
  $\vee\ \text{self::node}\langle v,q\rangle * q\text{::sortB}\langle B_1\rangle$
  $\wedge\ B=B_1\cup\{v\} \wedge \forall a.\, a\in B_1 \Rightarrow v\leq a$

Insert Into a Sorted List

```
node insert(node x, node vn)
requires x::sortB⟨B⟩ * vn::node⟨v,_⟩
ensures res::sortl⟨B ∪ {v}⟩

{
        if (vn.val≤x.val) {
                vn.next = x;
                return vn; }
        else if (x.next=null) then {
                x.next = vn; vn.next = null;
                return x; }
        else {
                x.next = insert(x.next, vn);
                return x; }
}
```

Entailment

$$\Delta_1 \vdash \Delta_2 * \Delta_3$$

- Δ_1 “provides” all heap nodes “requested” by Δ_2
- Remaining nodes are kept in Δ_3

Entailment

- Matching
  - •Aliased data nodes/predicates are matched and their components/arguments equated
- Unfolding
  - •Replaces a predicate on LHS by its definition to match a node on the RHS
- Folding
  - •Recursive entailment call to check predicate on RHS

Unfolding and Matching

$x::ll\langle n\rangle \wedge n>1 \vdash$
$\exists r, m \cdot x::node\langle \_,r\rangle * r::ll\langle m\rangle \wedge m>0$

- **Unfolding**
$\exists q \cdot x::node\langle \_, q\rangle * q::ll\langle n-1\rangle \wedge n>1 \vdash$
$\exists r, m \cdot x::node\langle \_, r\rangle * r::ll\langle m\rangle \wedge m>0$

- **Matching**
$q::ll\langle n-1\rangle \wedge n>1 \wedge q=r \vdash \exists m \cdot r::ll<m> \wedge m>0$

18

Entailment $\Delta_1 \vdash \Delta_2 * \Delta_3$

Entailment between separation formulae

1. Successively remove heap nodes from Δ_2

2 . When Δ_2 is pure, the heap formula in Δ_1 is approximated by function XPure

Entailment between pure formulae

Approximation

- Each predicate is approximated by a pure constraint.

$XPure(x::node\langle\_,\_\rangle * y::node\langle\_,\_\rangle)$

$= \exists i,j.\ (x=i \wedge i>0 \wedge y=j \wedge j>0 \wedge i \neq j)$

$= x \neq y$

$XPure(x::ll\langle n\rangle)$

$= \exists i.\ (self=0 \wedge n=0 \vee self=i \wedge i>0 \wedge n>0)$

$= (self=0 \wedge n=0) \vee (self \neq 0 \wedge n>0)$

Experiments

Programs	Without size/bag	Omega Calculator	Isabelle Prover	MONA Prover	Isabelle Prover	MONA Prover
Linked List		verifies size/length			verifies bag/set	
delete	0.02	0.09	8.35	0.33	5.00	0.34
reverse	0.02	0.07	3.28	0.21	3.01	0.20
Circular Linked List		verifies size + cyclic structure			verifies bag/set + cyclic structure	
delete (first)	0.01	0.09	5.46	0.26	7.17	0.40
count	0.04	0.16	14.99	0.71	21.01	2.29
Doubly Linked List		verifies size + double links			verifies bag/set + double links	
append	0.05	0.16	28.18	0.83	23.73	0.93
flatten (from tree)	0.08	0.30	158.3	6.65	55.78	2.03
Sorted List		verifies size + min + max + sortedness			verifies bag/set + sortedness	
delete	0.02	0.13	34.09	26.68	51.39	0.60
insertion_sort	0.07	0.27	41.17	18.22	27.34	0.73
selection_sort	0.10	0.41	79.08	20.62	221.7	1.10
bubble_sort	0.16	0.64	358.7	9.36	221.2	2.84
merge_sort	0.11	0.61	342.9	105.1	150.1	21.75
quick_sort	0.19	0.59	642.0	*out of memory*	*failed*	3.40
Binary Search Tree		verifies min + max + sortedness			verifies bag/set + sortedness	
insert	0.03	0.20	*failed*	11.92	99.57	0.95
delete	0.06	0.38	97.5	6.86	943.5	3.03
Priority Queue Heap		verifies size + height + max-heap			verifies bag/set + size + max-heap	
insert	0.15	0.45	520.8	41.55	416.2	6.45
delete_max	0.55	7.17	*failed*	290.7	*failed*	626.1
AVL Tree		verifies size + height + height-balanced			verifies bag/set + height + height-balanced	
insert	1.04	5.06	*failed*	36.02	1973	7.38
Red-Black Tree		verifies size + black-height + height-balanced			verifies bag/set + black-height + height-balanced	
insert	0.44	1.53	2992	352.4	*failed*	392.8
delete	8.29	5.98	24335	*out of memory*	42416	1691

Experiments

Programs	Without size/bag	Omega Calculator	Isabelle Prover	MONA Prover	Isabelle Prover	MONA Prover
Linked List		verifies size/length			verifies bag/set	
delete	0.02	0.09	8.35	0.33	5.00	0.34
reverse	0.02	0.07	3.28	0.21	3.01	0.20
Circular Linked List		verifies size + cyclic structure			verifies bag/set + cyclic structure	
delete (first)	0.01	0.09	5.46	0.26	7.17	0.40
count	0.04	0.16	14.99	0.71	21.01	2.29
Doubly Linked List		verifies size + double links			verifies bag/set + double links	
append	0.05	0.16	28.18	0.83	23.73	0.93
flatten (from tree)	0.08	0.30	158.3	6.65	55.78	2.03
Sorted List		verifies size + min + max + sortedness			verifies bag/set + sortedness	
delete	0.02	0.13	34.09	26.68	51.39	0.60
insertion_sort	0.07	0.27	41.17	18.22	27.34	0.73
selection_sort	0.10	0.41	79.08	20.62	221.7	1.10
bubble_sort	0.16	0.64	358.7	9.36	221.2	2.84
merge_sort	0.11	0.61	342.9	105.1	150.1	21.75
quick_sort	0.19	0.59	642.0	*out of memory*	*failed*	3.40
Binary Search Tree		verifies min + max + sortedness			verifies bag/set + sortedness	
insert	0.03	0.20	*failed*	11.92	99.57	0.95
delete	0.06	0.38	97.5	6.86	943.5	3.03
Priority Queue Heap		verifies size + height + max-heap			verifies bag/set + size + max-heap	
insert	0.15	0.45	520.8	41.55	416.2	6.45
delete_max	0.55	7.17	*failed*	290.7	*failed*	626.1
AVL Tree		verifies size + height + height-balanced			verifies bag/set + height + height-balanced	
insert	1.04	5.06	*failed*	36.02	1973	7.38
Red-Black Tree		verifies size + black-height + height-balanced			verifies bag/set + black-height + height-balanced	
insert	0.44	1.53	2992	352.4	*failed*	392.8
delete	8.29	5.98	24335	*out of memory*	42416	1691

Verification of OO programs

Must support behavioral subtyping.
Must support class inheritance.
Must support casting.
Good to support class invariants.
Good to support super/direct calls.

- precision
- efficiency (minimize code re-verification)

Overview

- Verification of shape, size and bag props
- Verification of OO programs
  - Enhanced Spec Subsumption ⟵
  - Static & Dynamic Specs
  - Key Principles

Behavioral subtyping

- Liskov's Substitutivity Principle (1988) :

  – an object of a subclass can always be passed to a location where an object of its superclass is expected

Spec subsumption

```
class A {
        t mn(..) where preA *→ postA {...}
}
class B extends A {
        t mn(..) where preB *→ postB {...}
}
```

Spec (preB *→ postB) is a **subtype** of (preA *→ postA) if:

$$\frac{preA \Rightarrow preB \qquad \wedge type(this) <: B \qquad old(preA) \wedge postB \Rightarrow postA}{(preB \ {*\!\!\rightarrow}\ postB) <:_B (preA \ {*\!\!\rightarrow}\ postA)}$$

Castagna('95)

contravariance

Leavens&Naumann('06)

covariance

Enhanced spec subsumption

- With the help of frame rule

$$\frac{\vdash \{P\}\ c\ \{Q\}}{\vdash \{P * \Delta\}\ c\ \{Q * \Delta\}}$$

Spec (preB $*\!\!\rightarrow$ postB) is a subtype of (preA $*\!\!\rightarrow$ postA) if:

$$\frac{preA \wedge type(this) <: B \Rightarrow preB * \Delta \qquad \Delta * postB \Rightarrow postA}{(preB *\!\!\rightarrow postB) <:_B (preA *\!\!\rightarrow postA)}$$

Static and dynamic specs : example

```
class Cnt { int val;
  Cnt(int v) {this.val:=v}
  void tick() {this.val:=this.val+1}
  int get() {this.val}
  void set(int x) {this.val:=x} }

class FastCnt extends Cnt {
  FastCnt(int v) {this.val:=v}
  void tick() {this.val:=this.val+2} }

class PosCnt extends Cnt inv this.val≥0 {
  PosCnt(int v) {this.val:=v}
  void set(int x) {if x≥0 then this.val:=x else error()} }
```

static this::Cnt<n>$ *→ this::Cnt<n+1>$

Static and dynamic specs : example

```
class Cnt { int val;
  Cnt(int v) {this.val:=v}
  void tick() {this.val:=this.val+1}
  int get() {this.val}
  void set(int x) {this.val:=x} }

class FastCnt extends Cnt {
  FastCnt(int v) {this.val:=v}
  void tick() {this.val:=this.val+2} }

class PosCnt extends Cnt inv this.val≥0 {
  PosCnt(int v) {this.val:=v}
  void set(int x) {if x≥0 then this.val:=x else error()} }
```

static this::Cnt<n>\$ *→ this::Cnt<n+1>\$
dynamic this::Cnt<n>\$ ∧ n≥0
*→ this::Cnt<b>\$ ∧ b=n+1 / n+1≤b≤n+2

Cnt.tick is overridden ⇒ weaken the postcond

PosCnt has inv this.val≥0 ⇒ strengthen the precond

Static and dynamic spec

A static spec:

- describes just a single method
- used for statically-dispatched calls (e.g. super/direct)
- can be very precise

↕

A dynamic spec:

- describes a method and its overriding methods
- used for dynamically-dispatched calls
- less precise

Key principles

- Static spec must be given for each new method.
- Code verification is done only for static spec.
- Dynamic spec is either given or derived.
- Subsumption relations:

```
class A {
// defines mn
}
```

Static-Spec(A.mn) <: Dyn-Spec(A.mn)

min. re-verification (→ <:)

Key principles

- Static spec must be given for each new method.
- Code verification is done only for static spec.
- Dynamic spec is either given or derived.
- Subsumption relations:

```
class A {
// defines mn
}
class B extends A {
// overrides mn
}
```

min. re-verification

Static-Spec(A.mn) <: Dyn-Spec(A.mn)

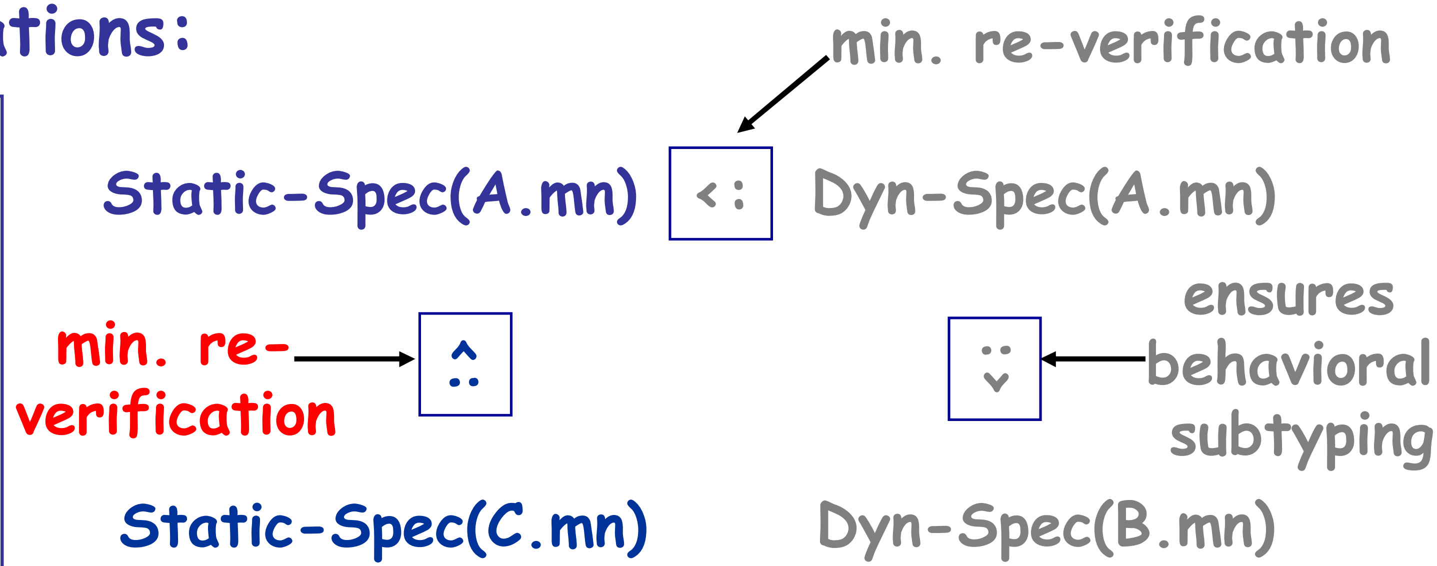
<: ensures behavioral subtyping

Dyn-Spec(B.mn)

Key principles

- Static spec must be given for each new method.
- Code verification is done only for static spec.
- Dynamic spec is either given or derived.
- Subsumption relations:

```
class A {
// defines mn
}
class B extends A {
// overrides mn
}
class C extends A {
// inherits mn
}
```



Initial experiment

- Code Verification > Spec Subsumption Checking

Class	Our system (secs)	With code re-verif. (secs)	Savings
Example 1			
A	0.04	0.05	20%
B	0.11	0.12	8.3%
Example 2			
A	0.08	0.12	33.3%
B	0.02	0.02	0%
The Cnt Example			
Cnt	0.05	0.09	44.4%
FastCnt	0.05	0.08	37.5%
PosCnt	0.11	0.13	15.3%
TwoCnt	0.09	0.14	35.7%

Conclusion

- Verification system based on separation logic
- User-definable shape predicates with size and bag properties
- Static and dynamic specs for OO verification

Thank you!

Questions?