# *jStar: Towards Practical Verification for Java (OOPSLA 2008)*

**Paper**: Dino Distefano & Matthew Parkinson

**Speaker**: Nels E. Beckman

# Overview

- jStar is an automated tool for verifying separation logic predicates in Java.
- **Interesting Because:**
  - Automated
  - Must handle
    - inheritance,
    - multi-object properties,
    - call-backs
  - Uses cool abstract predicates*

*Parkinson & Bierman, POPL 2005.

# Overview: Approach

- jStar works by
  - Requiring method pre/post conditions
    - Static & dynamic
  - Combining a theorem prover & abstract interpretation
    - Requires ‘abstraction rules’ from the users to ensure termination

# Outline

1. Really simple example
2. More interesting example
3. Symbolic Execution: Straight-Line
4. Proving separation logic predicates
5. Symbolic Execution: Fixed Point Computation

# Syntax

$E ::= \quad x \mid \hat{x} \mid nil \mid \ldots$

$P ::= \quad E = F \mid E \neq F \mid p(\overline{E})$

$S ::= \quad s(\overline{E})$

$\prod ::= \quad true \mid P \wedge \prod$

$\Sigma ::= \quad emp \mid S * \Sigma$

$H ::= \quad \prod \wedge \Sigma$

# Syntax

$E ::= \; x \mid \hat{x} \mid nil \mid \ldots$

$P ::= \; E = F \mid E \neq F \mid p(\bar{E})$

$S ::= \; s(\bar{E})$

$\prod ::= \; true \mid P \wedge \prod$

$\Sigma ::= \; emp \mid S * \Sigma$

$H ::= \; \prod \wedge \Sigma$

$s(\bar{E})$?

Basically always of the form,
$x \mapsto E$

$p(\bar{E})$?

Not sure this is used.

# Really Simple Example

```
class Cell {
  int val;

  void set(int x) {
    this.val = x;
  }

  int get() {
    return this.val;
  }
}
```

# Really Simple Example

```
class Cell {
  int val;

  void set(int x) {
    this.val = x;
  }

  int get() {
    return this.val;
  }
}
```

**Property of Interest:**

**define** Val$Cell(c, {content=y}) =

**true** | $c.val \mapsto y$

# Really Simple Example

```
class Cell {
  int val;

  void set(int x) {
    this.val = x;
  }

  int get() {
    return this.val;
  }
}
```

**Property of Interest:**

**define** Val$Cell(c, {content=y}) =

**true** | $c.val \mapsto y$

Pred. over stack (**true**)

Pred. over heap ($c.val \mapsto y$)

# Really Simple Example

```
class Recell extends Cell {
  int bak;

  void set(int x) {
    this.bak = super.get();
    super.set(x);
  }

  int get() {
    return super.get();
  }
}
```

# Really Simple Example

**Property of Interest:**

**define** Val\$Recell(c, {content=y; old=z}) =

**true** | Val\$Cell(x, {content=y}) $*$

*c.bak* $\mapsto$ *z*

```
class Recell extends Cell {
  int bak;

  void set(int x) {
    this.bak = super.get();
    super.set(x);
  }

  int get() {
    return super.get();
  }
}
```

# Abstracting Val$XXX to Val

# Abstracting Val$XXX to Val

type(x, Cell) $\Longrightarrow$
    Val(x, {contents=y}) $\Longleftrightarrow$ Val$Cell(x,{contents=y}))

# Abstracting Val$XXX to Val

$\text{type}(x, \text{Cell}) \Longrightarrow$
$\quad \text{Val}(x, \{\text{contents=y}\}) \Longleftrightarrow \text{Val\$Cell}(x,\{\text{contents=y}\}))$

$\text{type}(x, \text{Recell}) \Longrightarrow$
$\quad \text{Val}(x, \{\text{contents=y, old=z}\}) \Longleftrightarrow \text{Val\$Recell}(x,\{\text{contents=y, old=z}\})$

# Abstracting Val$XXX to Val

$\text{type}(x, \text{Cell}) \Longrightarrow$
$\quad \text{Val}(x, \{\text{contents}=y\}) \Longleftrightarrow \text{Val\$Cell}(x,\{\text{contents}=y\}))$

$\text{type}(x, \text{Recell}) \Longrightarrow$
$\quad \text{Val}(x, \{\text{contents}=y, \text{old}=z\}) \Longleftrightarrow \text{Val\$Recell}(x,\{\text{contents}=y, \text{old}=z\})$

**Q. What if I have:**
type(x, Recell) & Val(x, {contents=y}) ?

# Abstracting Val$XXX to Val

type(x, Cell) $\Longrightarrow$
 Val(x, {contents=y}) $\Longleftrightarrow$ Val$Cell(x,{contents=y}))

type(x, Recell) $\Longrightarrow$
 Val(x, {contents=y, old=z}) $\Longleftrightarrow$ Val$Recell(x,{contents=y, old=z})

**Q. What if I have:**
type(x, Recell) & Val(x, {contents=y}) ?

**A. Implicit Existential Quantification:**
Val(x, {contents=y) $\Longleftrightarrow$ Val$Recell(x,{contents=y, old=$\hat{o}$})

# Now, Let's Specify

```
class Cell {
  int get() :
    static
      pre: {true | Val$Cell(this, {content=Ê})}
      post: {Ê = return | Val$Cell(this, {content=Ê})}
    dynamic
      pre: {true | Val(this, {content=Ê})}
      post: {Ê = return | Val(this, {content=Ê})}
  {
    return this.val;
  }
  …
}
```

# Now, Let's Specify

```
class Cell {
  int get() :
    static
      pre: {true | Val$Cell(this, {content=Ê})}
      post: {Ê = return | Val$Cell(this, {content=Ê})}
    dynamic
      pre: {true | Val(this, {content=Ê})}
      post: {Ê = return | Val(this, {content=Ê})}
  {
    return this.val;
  }
  …
}
```

Client knows exact type, **super** or **private**

# Now, Let's Specify

```
class Cell {
 int get() :
   static
     pre: {true | Val$Cell(this, {content=Ê})}
     post: {Ê = return | Val$Cell(this, {content=Ê})}
   dynamic
     pre: {true | Val(this, {content=Ê})}
     post: {Ê = return | Val(this, {content=Ê})}
 {
   return this.val;
 }
 …
}
```

Client knows exact type, **super** or **private** (static)

Client uses dynamic dispatch (dynamic)

# Feature: No Class Invariants

```
class Cell {
  int get() :
    static
      pre: {true | Val$Cell(this, {content=Ê})}
      post: {Ê = return | Val$Cell(this, {content=Ê})}
    dynamic
      pre: {true | Val(this,
      post: {Ê = return | V
  {
    return this.val;
  }
  …
}
```

- In most OO verification systems, Val would be a class invariant.
  - Given at method start. Must be reestablished at method return.
- jStar does not have these.
  - For verifying OO patterns with call-backs

# More Specification

```
class Recell {
  void set(int x) :
    static
      pre: {true | Val$Recell(this, {content=Ê, old=Ô})}
      post: {true | Val$Recell(this, {content=x, old=Ê})}
    dynamic
      pre: {true | Val(this, {content=Ê, old=Ô})}
      post: {true | Val(this, {content=x, old=Ê})}
  {
    this.bak = super.get();
    super.set(x);
  }
  …
}
```

# More Specification

```
class Recell {
  void set(int x) :
    {true | Val$(this, {content=Ê, old=Ô})}
    {true | Val$(this, {content=x, old=Ê})}
  {
    this.bak = super.get();
    super.set(x);
  }
  …
}
```

# More Specification

```
class Recell {
  void set(int x) :
    {true | Val$(this, {content=Ê, old=Ô})}
    {true | Val$(this, {content=x, old=Ê})}
  {
    this.bak = super.get();
    super.set(x);
  }

  Recell(int x) :
    {true | emp}
    {true | Val$(this, {content=x, old=-1}) }
   {
    super(x);
    this.bak = -1;
   }
```

# Verification Preview

**void** set(**int** x) :
  {**true** | *Val*$(**this**, {content=Ê, old=Ô})}
  {**true** | *Val*$(**this**, {content=x, old=Ê})}
 {
   …
 }

{**true** | *Val*(**this**, {content=Ê, old=Ô}) ^
      type(**this**,Recell))}
{**true** | *Val*$Recell(**this**, {content=Ê, old=Ô})}}
{**true** | **this**.bak $\mapsto$ Ô $*$
      *Val*$Cell(**this**, {content=Ê})}
temp = **super**.get();
{temp = Ê | **this**.bak $\mapsto$ Ô $*$
      *Val*$Cell(**this**, {content=Ê})}
**this**.bak = temp;
{temp = Ê | **this**.bak $\mapsto$ Ê $*$
      *Val*$Cell(**this**, {content=Ê})}
**super**.set(x);
{temp = Ê | **this**.bak $\mapsto$ Ê $*$
      *Val*$Cell(**this**, {content=x})}

# Observer Example

```
interface Subject {
  void addObsrvr(Observer o);
  void remObsrvr(Observer o);
}

interface Observer {
  void update(Subject s);
}
```

# Observer Example

```
interface Subject {
  void addObsrvr(Observer o);
  void remObsrvr(Observer o);
}

interface Observer {
  void update(Subject s);
}
class IntegerList implements
  Subject {
  List ints = ...;
  List observers = ...;

  void addObsrvr(Observer o){
    this.observers.add(o);
  }
```

```
  void remObsrvr(Observer o){
    this.observers.remove(o);
  }

  void beginModification() {}
  void endModification() {
    notifyObservers();
  }

  private
  void notifyObservers() {
    for(o : observers) {
      o.update(this);
    }
  }
}
```

# Observer & Properties

```
class SizeKeeper implements
   Observer {
  IntegerList subj;
  int size;

  SizeKeeper(IntegerList s){
    s.addObsrvr(this);
    this.subj = s;
  }

  void update(Subject o) {
    if(o==subj)
      size=subj.list.size();
  }
}
```

# Observer & Properties

```
class SizeKeeper implements
   Observer {
  IntegerList subj;
  int size;

  SizeKeeper(IntegerList s){
    s.addObsrvr(this);
    this.subj = s;
  }

  void update(Subject o) {
    if(o==subj)
      size=subj.list.size();
  }
}
```

**define** *Subject*(s,{obs=*O*;vals=*V*}) =
*SubjectInternal*\$IntegerList(*s*,{obs=*O*}) $*$
*SubjectData*(*s*,{vals=*V*})

# Observer & Properties

```
class SizeKeeper implements
   Observer {
  IntegerList subj;
  int size;

  SizeKeeper(IntegerList s){
    s.addObsrvr(this);
    this.subj = s;
  }

  void update(Subject o) {
    if(o==subj)
      size=subj.list.size();
  }
}
```

**define** *Subject*(s,{obs=*O*;vals=*V*}) =
*SubjectInternal*\$IntegerList(*s*,{obs=*O*}) $*$
*SubjectData*(*s*,{vals=*V*})

**define** *SubjectInternal*(*s*,{obs=*O*}) =
s.observers $\mapsto$ *ô* $*$ *LinkedList*(ô,O)

# Observer & Properties

```
class SizeKeeper implements
   Observer {
  IntegerList subj;
  int size;

  SizeKeeper(IntegerList s){
    s.addObsrvr(this);
    this.subj = s;
  }

  void update(Subject o) {
    if(o==subj)
      size=subj.list.size();
  }
}
```

**define** *Subject*(s,{obs=*O*;vals=*V*}) =
*SubjectInternal*\$IntegerList(*s*,{obs=*O*}) $*$
*SubjectData*(*s*,{vals=*V*})

**define** *SubjectInternal*(*s*,{obs=*O*}) =
s.observers $\mapsto$ *ô* $*$ *LinkedList*(ô,O)

**define** *SubjectData*(s,{vals=V}) =
s.list $\mapsto$ ê $*$ *LinkedList*(î,V)

# Observer & Properties

```
class SizeKeeper implements
   Observer {
  IntegerList subj;
  int size;

  SizeKeeper(IntegerList s){
    s.addObsrvr(this);
    this.subj = s;
  }

  void update(Subject o) {
    if(o==subj)
      size=subj.list.size();
  }
}
```

**define** *Subject*(s,{obs=*O*;vals=*V*}) =
*SubjectInternal*$IntegerList(*s*,{obs=*O*}) $*$
*SubjectData*(*s*,{vals=*V*})

**define** *SubjectInternal*(*s*,{obs=*O*}) =
s.observers $\mapsto$ *ô* $*$ *LinkedList*(ô,O)

**define** *SubjectData*(s,{vals=V}) =
s.list $\mapsto$ ê $*$ *LinkedList*(î,V)

**define** *SubjectObs*(s, {obs=O;vals=V}) =
*Subject*$IntegerList(*s*, {obs=O;vals=V})
$*$ *ObsSet*(O,V,s)

# Method Specifications

IntegerList ---

**void** addObsrvr(Observer o) :

{|*SubjectObs*$(**this**, {obs=Ô; vals=Ê}) $*$ *Observer*(o,{vals=$Ê_2$;subject=**this**})}

{|*SubjectObs*$(**this**, {obs=add(o,Ô); vals=Ê})}

# Method Specifications

IntegerList ---

**void** addObsrvr(Observer o) :

{|*SubjectObs*$(**this**, {obs=Ô; vals=Ê}) $*$ *Observer*(o,{vals=$Ê_2$;subject=**this**})}

{|*SubjectObs*$(**this**, {obs=add(o,Ô); vals=Ê})}

**void** beginModification() :

{|*SubjectObs*$(**this**, {obs=Ô; vals=Ê}) }

{|*SubjectInternal*$(**this**, {obs=Ô}) $*$

*SubjectData*(**this**, {val=Ê}) $*$ *ObsSet*(Ô,Ê,**this**) }

# Method Specifications

IntegerList ---

**void** addObsrvr(Observer o) :

{|*SubjectObs*$(**this**, {obs=Ô; vals=Ê}) $*$ *Observer*(o,{vals=$Ê_2$;subject=**this**})}

{|*SubjectObs*$(**this**, {obs=add(o,Ô); vals=Ê})}

**void** beginModification() :

{|*SubjectObs*$(**this**, {obs=Ô; vals=Ê}) }

{|*SubjectInternal*$(**this**, {obs=Ô}) $*$

*SubjectData*(**this**, {val=Ê}) $*$ *ObsSet*(Ô,Ê,**this**) }

**void** notifyObservers() :

{|*Subject*$(**this**, {obs=Ô;vals=Ê}) $*$ *ObsSet*(Ô,$Ê_2$,**this**) }

{|*Subject*$(**this**, {obs=Ô;vals=Ê}) $*$ *ObsSet*(Ô,Ê,**this**) }

# Method Specifications

SizeKeeper ---

**void** update(Subject s) :

{|*Observer*(**this**,{vals=Ê; subject=s}) $*$ *SubjectData*(s, {vals=$\hat{E}_2$}) }

{|*Observer*(**this**,{vals=$\hat{E}_2$; subject=s}) $*$ *SubjectData*(s, {vals=$\hat{E}_2$}) }

# Method Specifications

SizeKeeper ---

**void** update(Subject s) :

{|*Observer*(**this**,{vals=Ê; subject=s}) $*$ *SubjectData*(s, {vals=$\hat{E}_2$}) }

{|*Observer*(**this**,{vals=$\hat{E}_2$; subject=s}) $*$ *SubjectData*(s, {vals=$\hat{E}_2$}) }

Here's another neat feature! **Aliasing!**

- Each observer has an additional aliased reference to the subject.
- However it can't access it unless it is given the SubjectData predicate by the subject.

# Outline

1. Really simple example
2. More interesting example
3. Symbolic Execution: Straight-Line
4. Proving separation logic predicates
5. Symbolic Execution: Fixed Point Computation

# Symbolic Execution: Straight Line

- Step through methods, one at a time.
- Update a "symbolic heap" based on evaluation rules.
  - Symbolic heap = separation logic assertion + typing information
  - exec : $\mathit{Stmts} \times \mathit{Heaps} \rightarrow \mathrm{P}(\mathit{Heaps}) \cup \{ \top \}$
- At method call sites, talk to the theorem prover.

# Exec Rules

$$\frac{}{H, \quad x = E \;\longrightarrow\; x = E[\hat{x}/x] \wedge H[\hat{x}/x]} \text{ Assignment 1}$$

$$\frac{}{H * x.\langle C\colon t\ f\rangle \mapsto E_1, \quad x.\langle C\colon t\ f\rangle = E_2 \;\longrightarrow\; H * x.\langle C\colon t\ f\rangle \mapsto E_2} \text{ Mutation}$$

$$\frac{}{H * E.\langle C\colon t\ f\rangle \mapsto E_1, \quad x = E.\langle C\colon t\ f\rangle \;\longrightarrow\; x = E_1[\hat{x}/x] \wedge (H * E.\langle C\colon t\ f\rangle \mapsto E_1)[\hat{x}/x]} \text{ Look-up}$$

$$\frac{}{H, \quad \mathsf{return}\ E \;\longrightarrow\; ret = E \wedge H} \text{ Return}$$

$$\frac{S \in \mathsf{spec}_{\mathsf{invoke}}(C, t, m) \qquad \mathsf{jsr}(S, H, v) = H'}{H, \quad \mathsf{invoke}\ x.\langle C\colon t\ m\rangle(v) \;\longrightarrow\; H'} \text{ Invoke}$$

$$\frac{H,\ \mathsf{invoke}\ y.\langle C\colon t\ m\rangle(v) \;\longrightarrow\; H'}{H, \quad x = \mathsf{invoke}\ y.\langle C\colon t\ m\rangle(v) \;\longrightarrow\; H'[x/ret]} \text{ Assignment 2}$$

$$\frac{H[\hat{x}/x],\ \mathtt{virtualinvoke}\ x.\langle C\colon void\ \mathsf{init}\rangle(v) \;\longrightarrow\; H'}{H, \quad x = \mathsf{new}\ C(v) \;\longrightarrow\; H'} \text{ Allocation}$$

# Method Call Sites

$$\frac{S \in \mathsf{spec}_{\mathsf{invoke}}(C, t, m) \qquad \mathsf{jsr}(S, H, v) = H'}{H, \quad \mathsf{invoke}\; x.\langle C\colon t\; m\rangle(v) \;\longrightarrow H'}\;\text{Invoke}$$

# Method Call Sites

$$\frac{S \in \mathsf{spec}_{\mathsf{invoke}}(C, t, m) \qquad \mathsf{jsr}(S, H, v) = H'}{H, \quad \mathsf{invoke}\; x.\langle C\colon t\; m\rangle(v) \;\longrightarrow H'}\ \text{Invoke}$$

$$\mathrm{jsr}(\{P\}\ \mathrm{m(ps)}\ \{Q\}, H, \mathrm{args}) = \begin{cases} H' * \mathrm{Q[args/ps]} \text{ if } H \vdash \mathrm{P[args/ps]} * H' \\ \top \text{ otherwise} \end{cases}$$

# Method Call Sites

$$\frac{S \in \mathsf{spec}_{\mathsf{invoke}}(C, t, m) \qquad \mathsf{jsr}(S, H, v) = H'}{H, \quad \mathsf{invoke}\ x.\langle C\colon t\ m\rangle(v) \ \longrightarrow H'} \ \text{Invoke}$$

$$\mathrm{jsr}(\{P\}\ m(ps)\ \{Q\}, H, args) = \begin{cases} H' * Q[args/ps] \text{ if } H \vdash P[args/ps] * H' \\ \top \text{ otherwise} \end{cases}$$

We pose the following question to the theorem prover:
"Can you find H' such that…"
$H \vdash P[args/ps] * H'$

# Example

Call:

x.set(7)

Symbolic Heap:

Val(x, {content=3}) *

Val(y, {content=9})

Cell: **void** set(**int** x):

{|Val(**this**, {content=Ê})}

{|Val(**this**, {content=x})}

# Example

Call:

x.set(7)

Symbolic Heap:

Val(x, {content=3}) $*$

Val(y, {content=9})

Cell: **void** set(**int** x):

{|Val(**this**, {content=Ê})}

{|Val(**this**, {content=x})}

"Theorem prover, find H' such that:"

Val(x,{content=3}) $*$
Val(y,{content=9}) $\vdash$

Val(x,{content=Ê}) $*$ H'

Response:

H' = Ê=3 $\wedge$ Val(y,{content=9})

# Proving Predicates

- Theorem Prover
  - Called by symbolic execution
  - Decides implications (entailment checking)
  - Performs frame inference
  - Based on Smallfoot*

*Berdine, Calcagno, O'Hearn. FMCO 2005.

# Entailment Checking

- Solves sequents of the form

  $\Sigma_f \mid \prod_1 \mid \Sigma_1 \vdash \prod_2 \mid \Sigma_2$
- Whose "semantics" are

  $\prod_1 \wedge (\Sigma_f * \Sigma_1) \Longrightarrow \prod_2 \wedge (\Sigma_f * \Sigma_2)$
- Unfortunately, details are a little light in this section…
  - Unification and basic axioms of separation logic built into prover (e.g., comm. over $*$)
  - Otherwise, programmers add simplification rules

# Entailment Checking

- Solves sequents of the form

  $\Sigma_f \mid \prod_1 \mid \Sigma_1 \vdash \prod_2 \mid \Sigma_2$

- Whose "semantics" are

  $\prod_1 \wedge (\Sigma_f * \Sigma_1) \Longrightarrow \prod_2 \wedge (\Sigma_f * \Sigma_2)$

- Unfortunately, details are a little light in this section…
  – Unification and basic axioms of separation logic built into prover (e.g., comm. over $*$)
  – Otherwise, programmers add simplification rules

$$\frac{}{\Sigma_f \mid \prod_1 \mid \text{emp} \vdash \text{true} \mid \text{emp}}$$

# Simplification Rules

- The user can (must?) provide jStar with rules for simplifying proof rules.
- E.g,

$$\frac{\Sigma_f * S \mid \Pi_1 \mid \Sigma_1 \vdash \Pi_2 \mid \Sigma_2}{\Sigma_f \mid \Pi_1 \mid \Sigma_1 * S \vdash \Pi_2 \mid \Sigma_2 * S}$$

$$\frac{\Sigma_f[E/x] \mid \Pi_1[E/x] \mid \Sigma_1[E/x] \vdash \Pi_2[E/x] \mid \Sigma_2[E/x]}{\Sigma_f \mid \Pi_1 \wedge x = E \mid \Sigma_1 \vdash \Pi_2 \mid \Sigma_2}$$

# Frame Inference

- A key part of the theorem prover's job is frame inference:
  - Given $H_1$ and $H_2$ find $H_3$ s.t.
  - $H_1 \Longrightarrow H_2 * H_3$
- Finding the heap:
  1. Prove the formula with the whole heap
  2. Collect all the left over predicates from each proof tree
  3. Their disjunction is the frame

# Frame Inference

- A key part of the theorem prover's job is frame inference:
  - Given $H_1$ and $H_2$ find $H_3$ s.t.
  - $H_1 \Longrightarrow H_2 * H_3$
- Finding the heap:
  1. Prove the formula with the whole heap
  2. Collect all the left over predicates from each proof tree
  3. Their disjunction is the frame

$$\frac{\text{addToFrame}(\prod_1, \Sigma_1)}{\Sigma_{fl} \mid \prod_1 \mid \Sigma_1 \vdash \text{true} \mid \text{emp}}$$

# Outline

1. Really simple example
2. More interesting example
3. Symbolic Execution: Straight-Line
4. Proving separation logic predicates
5. Symbolic Execution: Fixed Point Computation

# Fixed Point Computation & Abstraction

- As expected in symbolic execution the **heap** predicate will explode around **loops** unless we **abstract**.

```
void create() {
  head = null;
  while( /* cond */ ) {
    Node n = new Node();
    n.next = head;
    head = n;
  }
}
```

$H_1 =$ head = n $\wedge$ Node(n,nil,v)

$H_2 =$ head = n $\wedge$ Node(n,ê,$v_1$) $*$ Node(ê,nil,$v_2$)

$H_3 =$ head = n $\wedge$ Node(n,ê,$v_1$) $*$ …

# Abstraction Rules

- In jStar, programmers provide abstraction rules
  - On a per program basis!
  - Tells theorem prover it can collapse a heap
  - Tried after every step

$$\frac{\text{condition}}{H * H' \rightsquigarrow H' * H''}$$

# Abstraction Rules

- In jStar, programmers provide abstraction rules
  - On a per program basis!
  - Tells theorem prover it can collapse a heap
  - Tried after every step

$$\frac{\text{condition}}{H * H' \rightsquigarrow H' * H''}$$

$$\frac{\hat{e} \notin Var(H, x)}{H * Node(x,\hat{e},\hat{\imath}) * lseg(\hat{e},nil,\hat{\imath}_2) \rightsquigarrow H' * lseg(x,nill,\hat{\imath}_3)}$$

# Abstraction Rules Used in Observer

```
Rule LS_OBS:
| | ls(?x,?z,cons(?w,?r))* Observer(?w,{val=?v; subject=?s}) ⊢|
if
| | lspe( f,?z,?r) * lsObs(?x, f,cons(?w,empty()),?v,?s) ⊢|


Rule LS_OBS_APP1:
| | lsObs(?x, f,?l,?v,?s) * lsObs( f,nil(),?l2,?v,?s) ⊢ |
where
_f notincontext;
_f notin ?x, ?l, ?v, ?s, ?l2;
if
| | lsObs(?x,nil(),app(?l,?l2),?v,?s) ⊢ |
```