

Verificación de sistemas de código que utilizan memoria dinámica

Renato Cherini

Facultad de Matemática, Astronomía y Física, UNC.

Córdoba - 6 de diciembre de 2010

El problema del *aliasing*

Consideremos el siguiente código:

{true}

[x] := 6

[y] := 7

{[x] ≠ [y]}

- Queremos saber que ciertas cosas son diferentes. ¿Cómo?

El problema del aliasing

Consideremos el siguiente código:

```
{true}
```

```
[x] := 6
```

```
[y] := 7
```

```
{[x] ≠ [y]}
```

Agregar una aserción

- Queremos saber que ciertas cosas son diferentes. ¿Cómo?

El problema del aliasing

Consideremos el siguiente código:

$$\{x \neq y\}$$

[x] := 6

[y] := 7

$$\{[x] \neq [y]\}$$

Agregar una aserción

- Queremos saber que ciertas cosas son diferentes. ¿Cómo?

El problema del *aliasing*

Consideremos el siguiente código:

$$\{x \neq y\}$$

$$[x] := 6$$

$$[y] := 7$$

$$\{[x] \neq [y] \wedge [z] = 5\}$$

- Queremos saber que ciertas cosas son diferentes.
- Queremos saber que ciertas cosas permanecen iguales. ¿Cómo?

El problema del aliasing

Consideremos el siguiente código:

$$\{x \neq y\}$$

[x] := 6

[y] := 7

$$\{[x] \neq [y] \wedge [z] = 5\}$$

Agregar una aserción

- Queremos saber que ciertas cosas son diferentes.
- Queremos saber que ciertas cosas permanecen iguales. ¿Cómo?

El problema del aliasing

Consideremos el siguiente código:

$$\{x \neq y \wedge [z] = 5 \wedge z \neq x \wedge z \neq y\}$$

[x] := 6

[y] := 7

$$\{[x] \neq [y] \wedge [z] = 5\}$$

Agregar una aserción

- Queremos saber que ciertas cosas son diferentes.
- Queremos saber que ciertas cosas permanecen iguales. ¿Cómo?

El problema del *aliasing*

Consideremos el siguiente código:

$$\{x \neq y \wedge ?\}$$

[x] := 6

[y] := 7

$$\{[x] \neq [y] \wedge \text{list}.z\}$$

- Queremos saber que ciertas cosas son diferentes.
- Queremos saber que ciertas cosas permanecen iguales.

Separation Logic

- Extiende las aserciones de la lógica de Hoare para razonar sobre programas con manejo de punteros de bajo nivel.
- Incorpora operadores espaciales ($*$, \rightarrow) que codifican la ausencia de aliases de punteros.
- Permite razonar localmente a través de la Regla de Frame:

$$\frac{\{P\} \ C \ \{Q\}}{\{P \wedge R\} \ C \ \{Q \wedge R\}} \quad \frac{\{P\} \ C \ \{Q\}}{\{P * R\} \ C \ \{Q * R\}}$$

- “*SL makes earlier attempts to prove pointer-mutating programs look ridiculously complicated and ad-hoc*”

Separation Logic

- Extiende las aserciones de la lógica de Hoare para razonar sobre programas con manejo de punteros de bajo nivel.
- Incorpora operadores espaciales ($*$, \rightarrow) que codifican la ausencia de aliases de punteros.
- Permite razonar localmente a través de la Regla de Frame:

$$\frac{\{P\} \ C \ \{Q\}}{\{P \wedge R\} \ C \ \{Q \wedge R\}} \quad \frac{\{P\} \ C \ \{Q\}}{\{P * R\} \ C \ \{Q * R\}}$$


- “*SL makes earlier attempts to prove pointer-mutating programs look ridiculously complicated and ad-hoc*”

Especificaciones con SL

{list.x.XS}

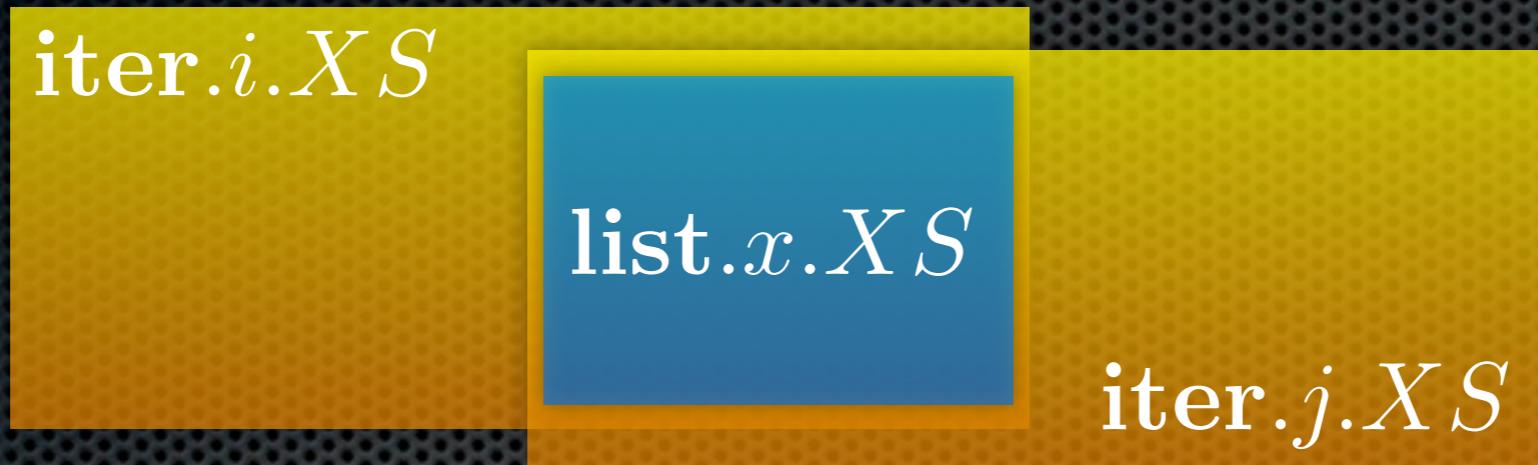
```
[p] := nil;  
while (x <> nil) {  
    q := p;  
    p := c;
```

$\{(list.q.XS_1 * list.c.XS_2) \wedge p = c \wedge rev.XS_1 ++ XS_2 = XS\}$

```
c := [c];  
[p] := q;  
}
```

{list.x.(rev.XS)}

Especificaciones con *sharing*



$(\text{iter}.i.XS * \text{true}) \wedge (\text{list}.x.XS * \text{true}) \wedge (\text{iter}.j.XS * \text{true})$

- **Inseguras desde el punto de vista metodológico:** es necesario apelar al ingenio o a “trucos” para especificar patrones de sharing.
- **Inadecuadas desde el punto de la teoria:** el uso del operador inhibe el uso de la regla de frame, forzando a razonar globalmente.

SL vs ShL

$(s, h) \models P * Q$ si i $\exists h_1, h_2$ tal que $h = h_1 \uplus h_2$
y $(s, h_1) \models P$ y $(s, h_2) \models Q$


$$h_1 \quad h_2 \quad \models \quad P * Q$$

$(s, h) \models P \langle * : R \rangle Q$ si i $\exists h_1, h_2, h_3$ tal que $h = h_1 \uplus h_2 \uplus h_3$
y $(s, h_1 \cup h_3) \models P$ y $(s, h_3) \models R$
y $(s, h_2 \cup h_3) \models Q$


$$h_1 \quad h_3 \quad h_2 \quad \models \quad P \langle * : R \rangle Q$$

Sharing Logic



- Subsume a la SL: $P * Q \doteq P \langle * : \mathbf{emp} \rangle Q$
- Permite la especificación de relaciones de *sharing* de forma sistemática, evitando las especificaciones *ad hoc*.
- Acepta una Regla de Frame General que permite razonar de forma composicional sin romper la abstracción:

$$\frac{\{P\} C \{Q\} \quad (R -\circledast I) * I' \Rightarrow R' \quad Q \Rightarrow R' * \mathbf{true}}{\{P \langle * : R \rangle I\} C \{Q \langle * : R' \rangle I'\}}$$

Shape Analysis con SL

- Shape Analysis es una forma de análisis estático para verificar la validez de propiedades sobre la “forma” de las estructuras dinámicas.
- Los Shape Analysis basados en SL son modulares y soportan programas de gran escala, programación concurrente y programación orientada a objetos.
- Los análisis actuales son capaces de verificar propiedades sobre programas que manipulan estructuras de datos lineales como listas ligadas (posiblemente combinadas de forma compleja).

Objetivo

- Extender un shape analysis basado en SL para soportar estructuras no-lineales como grafos.
- La clave fue obtener un predicado atomico que:
 - permitiera la especificación de las estructuras parciales que ocurren en la iteración de los ciclos.
 - exhibiera buenas propiedades sintácticas, permitiendo la composición de estructuras de datos (parciales).
 - no diera lugar a una inmanejable proliferación de ocurrencias de los predicados en las fórmulas.

Objetos

- Extender la clase `sopor`
- La clase `sopor` hereda de `parc`
- permite la inserción y eliminación de datos (para el tipo `dato`)
- excepciones que permiten manejar errores (para el tipo `error`)
- no se permite la modificación de los datos que ocurren en la memoria.

{*Pre:* graph.{*r*}. \emptyset }

```
m,p := 0,null;
if r != null {m := r.2} else {m := 1};
while p != null || (r != null && m = 0) {
    if r = null || m != 0 {
        c := p.2;
        if c = 2 {
            q,r := r,p;
            p := p.1;
            r.1 := q;
        } else {
            q := r;
            r := p.1;
            b := p.0;
            p.1 := b;
            p.0 := q;
            p.2 := 2;
        }
    } else {
        q,p := p,r;
        r := r.0;
        p.0 := q;
        p.2 := 1;
    };
    if r != null {m := r.2} else {m := 1}
}
```

{*Post:* ?}

Objetivo

- Extender un shape analysis basado en SL para soportar estructuras no-lineales como grafos.
- La clave fue obtener un predicado atomico que:
 - permitiera la especificación de las estructuras parciales que ocurren en la iteración de los ciclos.
 - exhibiera buenas propiedades sintácticas, permitiendo la composición de estructuras de datos (parciales).
 - no diera lugar a una inmanejable proliferación de ocurrencias de los predicados en las fórmulas.

Semántica Abstracta

Ejemplo de ejecución simbólica

```
v := p.2;  
if v == Val {  
    end := True  
} else {  
    q := p;  
    if v < Val {  
        d := 0;  
        p := p.0  
    } else {  
        d := 1;  
        p := p.1  
    }  
}
```

$\{\text{trees.}\{p\}\.\emptyset\}$

Semántica Abstracta

Ejemplo de ejecución simbólica

```
v := p.2;  
if v == Val {  
    end := True  
} else {  
    q := p;  
    if v < Val {  
        d := 0;  
        p := p.0  
    } else {  
        d := 1;  
        p := p.1  
    }  
}
```

Semántica Abstracta

Ejemplo de ejecución simbólica

```
v := p.2;  
if v == Val {  
    end := True  
} else {  
    q := p;  
    if v < Val {  $\{p \mapsto x', y', v' * \text{trees.}\{x', y'\}.\emptyset\}$   
        d := 0;  
        p := p.0  
    } else {  
        d := 1;  
        p := p.1  
    }  
}
```

Semántica Abstracta

Ejemplo de ejecución simbólica

```
v := p.2;  
if v == Val {  
    end := True  
} else {  
    q := p;  
    if v < Val {  $\{p \mapsto x', y', v' * \text{trees.}\{x', y'\}.\emptyset\}$   
        d := 0;  
    } else {  
        p := p.0  
    }  
    d := 1;  
    p := p.1  
}  
}
```



Semántica Abstracta

Ejemplo de ejecución simbólica

```
v := p.2;  
if v == Val {  
    end := True  
} else {  
    q := p;  
    if v < Val {  
        d := 0;  
    } else {  
        p := p.0  
    } else {  
        d := 1;  
        p := p.1  
    }  
}
```



Semántica Abstracta

Ejemplo de ejecución simbólica

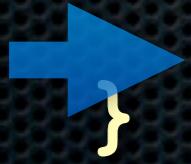
```
v := p.2;  
if v == Val {  
    end := True  
} else {  
    q := p;  
    if v < Val {  $\{q \mapsto p, y', v' * \text{trees.}\{p, y'\}.\emptyset, \dots$   
        d := 0;  
    } else {  
        p := p.0  
    } else {  
        d := 1;  
        p := p.1  
    }  
}
```



Semántica Abstracta

Ejemplo de ejecución simbólica

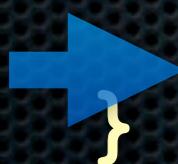
```
v := p.2;  
if v == Val {  
    end := True  
} else {  
    q := p;  
    if v < Val {  $\{q \mapsto p, y', v' * \text{trees.}\{p, y'\}.\emptyset, \dots$   
        d := 0;  
        p := p.0  
    } else {  
        d := 1;  
        p := p.1  
    }  
}
```



Semántica Abstracta

Ejemplo de ejecución simbólica

```
v := p.2;  
if v == Val {  
    end := True  
} else {  
    q := p;  
    if v < Val {  
        d := 0;  
        p := p.0  
    } else {  
        d := 1;  
        p := p.1  
    }  
}
```

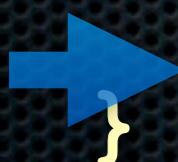


Semántica Abstracta

Ejemplo de ejecución simbólica

```
v := p.2;  
if v == Val {  
    end := True  
} else {  
    q := p;  
    if v < Val {  
        d := 0;  
        p := p.0  
    } else {  
        d := 1;  
        p := p.1  
    }  
}
```

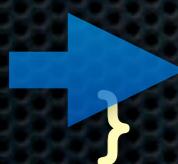
$\{q \mapsto p, y', v' * \text{trees.}\{p, y'\}\}. \emptyset,$
 $q \mapsto x', p, v' * \text{trees.}\{x', p\}\}$



Semántica Abstracta

Ejemplo de ejecución simbólica

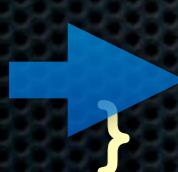
```
v := p.2;  
if v == Val {  
    end := True  
} else {  
    q := p;  
    if v < Val {  
        d := 0;  
        p := p.0  
    } else {  
        d := 1;  
        p := p.1  
    }  
}
```



Semántica Abstracta

Ejemplo de ejecución simbólica

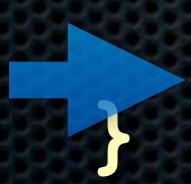
```
v := p.2;  
if v == Val {  
    end := True  
} else {  
    q := p;  
    if v < Val {  
        d := 0;  
        p := p.0  
    } else {  
        d := 1;  
        p := p.1  
    }  
}
```

$$\{q \mapsto p, y', v' * \text{trees.}\{p, y'\}\emptyset,$$
$$q \mapsto x', p, v' * \text{trees.}\{x', p\}\emptyset$$
$$q' \mapsto q, y', v' * q \mapsto p, y'', v'' * \text{trees.}\{y', p, y''\}\emptyset,$$
$$q' \mapsto q, y', v' * q \mapsto x'', p, v'' * \text{trees.}\{y', p, x''\}\emptyset,$$
$$q' \mapsto x', q, v' * q \mapsto p, y'', v'' * \text{trees.}\{x', p, y''\}\emptyset,$$
$$q' \mapsto x', q, v' * q \mapsto x'', p, v'' * \text{trees.}\{x', p, x''\}\emptyset\}$$


Semántica Abstracta

Ejemplo de ejecución simbólica

```
v := p.2;  
if v == Val {  
    end := True  
} else {  
    q := p;  
    if v < Val {  
        d := 0;  
        p := p.0  
    } else {  
        d := 1;  
        p := p.1  
    }  
}
```



Relación entre semánticas

Semántica
concreta

- Dominio:

$$D_c \doteq \mathcal{P}(\text{States} \cup \{\top\})$$

$$\xleftarrow{\gamma/\alpha}$$

- Semántica de programas:

$$[\![p]\!]_c : D_c \rightarrow D_c$$

$$\xleftarrow{\subseteq}$$

es estándar.

- El elemento Top denota un fallo de memoria.

Semántica
abstracta

- Dominio:

$$D_a \doteq \mathcal{P}(\text{SH} \cup \{\top\})$$

- Semántica de programas:

$$[\![p]\!]_a : D_a \rightarrow D_a$$

- rearrangement
- ejecución simbólica
- abstracción
- El elemento Top denota un posible fallo de memoria.

Semántica abstracta

El dominio de los Symbolic Heaps

SH	$::=$	$\Pi \upharpoonright \Sigma$
Π	$::=$	true $ e = e e \neq e \Pi \wedge \Pi$
Σ	$::=$	emp $ $ true $ $ junk $ v \mapsto e, e, e \Sigma * \Sigma $ struct. <i>ms.ms</i>
<i>ms</i>	$::=$	$\{e, e, \dots, e\}$
e	$::=$	$x x' \mathbf{nil} 0 1 \dots$

Semántica abstracta

El dominio de los Symbolic Heaps

$SH ::= \Pi \mid \Sigma$

$\boxed{\Pi} ::= \text{true} \mid e = e \mid e \neq e \mid \Pi \wedge \Pi$

$\Sigma ::= \text{emp} \mid \text{true} \mid \text{junk} \mid v \mapsto e, e, e \mid \Sigma * \Sigma \mid \text{struct}.ms.ms$

$ms ::= \{e, e, \dots, e\}$

$e ::= x \mid x' \mid \text{nil} \mid 0 \mid 1 \dots$

Semántica abstracta

El dominio de los Symbolic Heaps

$SH ::= \Pi ; \Sigma$

$\Pi ::= \text{true} \mid e = e \mid e \neq e \mid \Pi \wedge \Pi$

$\Sigma ::= \text{emp} \mid \text{true} \mid \text{junk} \mid v \mapsto e, e, e \mid \Sigma * \Sigma \mid \text{struct}.ms.ms$

$ms ::= \{e, e, \dots, e\}$

$e ::= x \mid x' \mid \text{nil} \mid 0 \mid 1 \dots$

Semántica abstracta

El dominio de los Symbolic Heaps

$SH ::= \Pi ; \Sigma$

$\Pi ::= \text{true} \mid e = e \mid e \neq e \mid \Pi \wedge \Pi$

$\Sigma ::= \text{emp} \mid \text{true} \mid \text{junk} \mid v \mapsto e, e, e \mid \Sigma * \Sigma \mid \text{struct}.ms.ms$

$ms ::= \{e, e, \dots, e\}$

$e ::= x \mid x' \mid \text{nil} \mid 0 \mid 1 \dots$

emp



stack



heap

Semántica abstracta

El dominio de los Symbolic Heaps

$SH ::= \Pi ; \Sigma$

$\Pi ::= \text{true} \mid e = e \mid e \neq e \mid \Pi \wedge \Pi$

$\Sigma ::= \text{emp} \mid \text{true} \mid \text{junk} \mid v \mapsto e, e, e \mid \Sigma * \Sigma \mid \text{struct}.ms.ms$

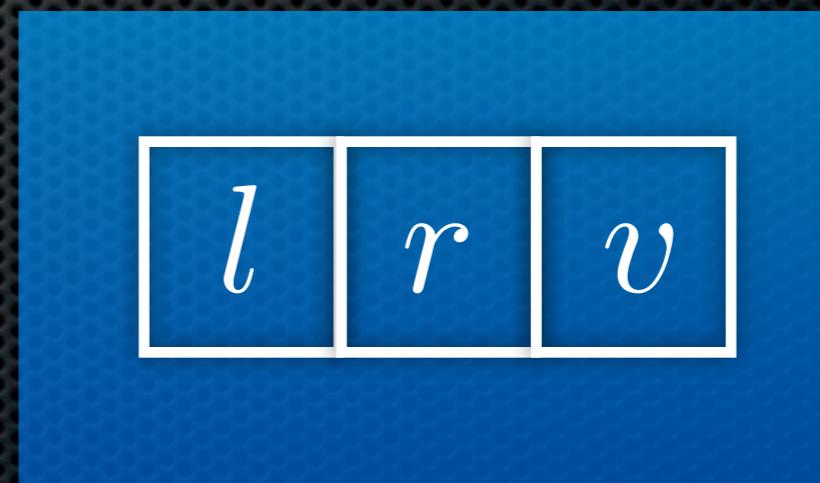
$ms ::= \{e, e, \dots, e\}$

$e ::= x \mid x' \mid \text{nil} \mid 0 \mid 1 \dots$

junk



stack



heap

Semántica abstracta

El dominio de los Symbolic Heaps

$SH ::= \Pi \upharpoonright \Sigma$

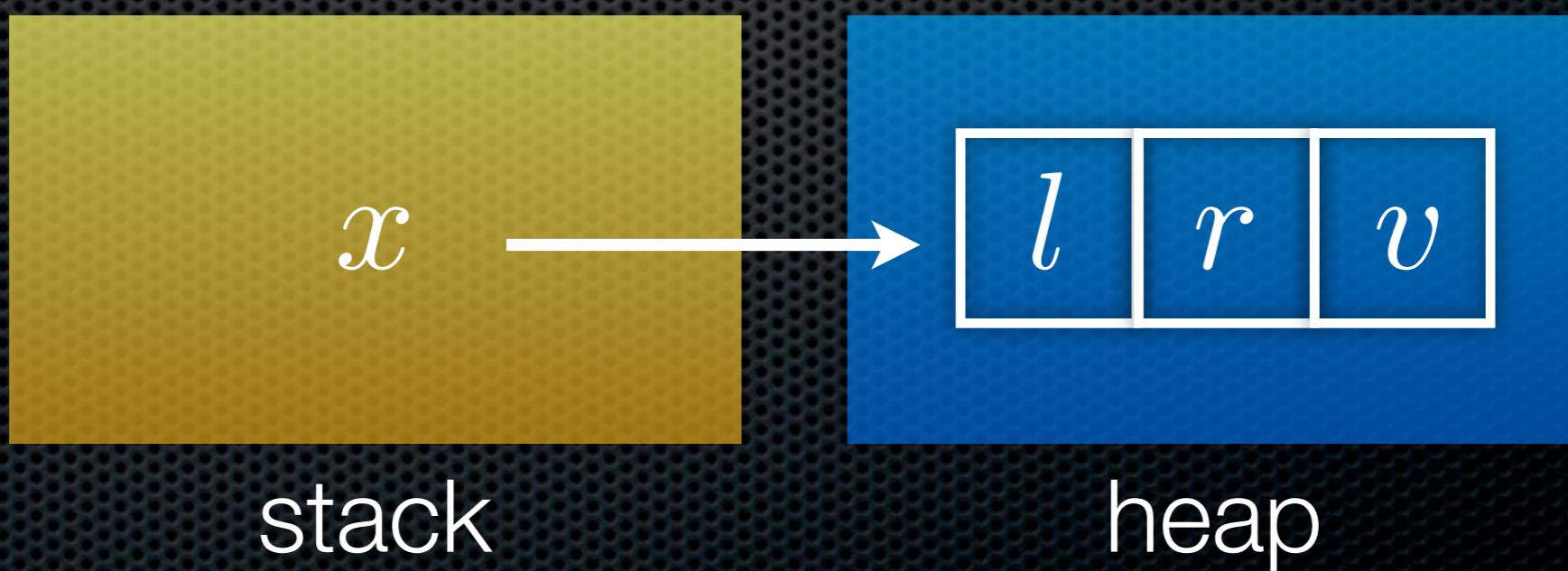
$\Pi ::= \mathbf{true} \mid e = e \mid e \neq e \mid \Pi \wedge \Pi$

$\Sigma ::= \mathbf{emp} \mid \mathbf{true} \mid \mathbf{junk} \mid v \mapsto e, e, e \mid \Sigma * \Sigma \mid \mathbf{struct}.ms.ms$

$ms ::= \{e, e, \dots, e\}$

$e ::= x \mid x' \mid \mathbf{nil} \mid 0 \mid 1 \dots$

$$x \mapsto (l, r, v)$$



Semántica abstracta

El dominio de los Symbolic Heaps

$SH ::= \Pi \upharpoonright \Sigma$

$\Pi ::= \mathbf{true} \mid e = e \mid e \neq e \mid \Pi \wedge \Pi$

$\Sigma ::= \mathbf{emp} \mid \mathbf{true} \mid \mathbf{junk} \mid v \mapsto e, e, e \mid \Sigma * \Sigma \mid \mathbf{struct}.ms.ms$

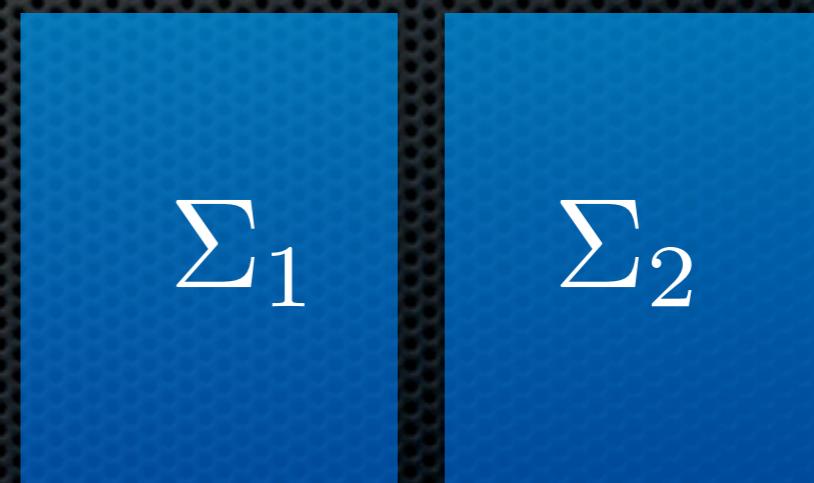
$ms ::= \{e, e, \dots, e\}$

$e ::= x \mid x' \mid \mathbf{nil} \mid 0 \mid 1 \dots$

$\Sigma_1 * \Sigma_2$



stack



heap

Semántica abstracta

El dominio de los Symbolic Heaps

$SH ::= \Pi \mid \Sigma$

$\Pi ::= \text{true} \mid e = e \mid e \neq e \mid \Pi \wedge \Pi$

$\Sigma ::= \text{emp} \mid \text{true} \mid \text{junk} \mid v \mapsto e, e, e \mid \Sigma * \Sigma \mid \text{struct}.ms.ms$

$ms ::= \{e, e, \dots, e\}$

$e ::= x \mid x' \mid \text{nil} \mid 0 \mid 1 \dots$

- **trees.** $\mathcal{C}.\mathcal{D}$ define una familia de arboles binarios.
 - Cada puntero de salida es alcanzado por un puntero de entrada.
 - Los punteros de salidas pueden ser compartidos.
- **graph.** $\mathcal{C}.\mathcal{D}$ define un grafo general.
 - \mathcal{C} representa los puntos de entrada de recorrido.
 - \mathcal{D} representa los nodos “visitados”.

Semántica abstracta

Abstracción para calcular invariantes

- Está organizada en tres pasos de aplicación de:
 - **1º:** (seis) reglas para componer predicados atómicos.
 - **2º:** (siete) reglas para “juntar la basura”.
 - **3º:** (cuatro) reglas para simplificar predicados triviales, y eliminar heaps inconsistentes.
- Se aplica antes de entrar a un ciclo, y después de cada iteración.
- Sólo se aplica si la variable a abstraer no provee información relevante para el futuro.

Relevancia de variables

- El análisis **relev** codifica la información prevista que se necesitará sobre cada variable en un programa.
 - Distinguimos entre:
 - Variables cuantificadas. x'
 - Variables cuyo valor es “pisado”. $x := e$
 - Variables en la precondición.
 - Variables cuyo valor es asignado. $y := x$
 - Variables usadas para observar el heap. $y := x.0$
 - Variables usadas para modificar el heap. $x.0 := e$
- 

Consistencia y terminación

- **Consistencia:** la semántica abstracta es una sobre-aproximación de la semántica concreta.

$$[\![p]\!]_c.(\gamma.S) \subseteq \gamma.([\![p]\!]_a.S) \quad \forall S \in \mathcal{P}(\text{SH} \cup \{\top\}).$$

- Cada regla de reescritura es una implicación válida en Separation Logic.
- **Terminación:** cada secuencia de reglas de abstracción termina, y el conjunto de fórmulas resultante es finito.

Ejemplos

Algorithm	Precondition / Postcondition	R.L.	Inv	Iter	Time
min/max	$\text{true} \mid \text{trees.}\{x\}.\emptyset$	2-4	2	2	0.003
	$x = \text{nil} \mid \text{emp}$				
	$x \neq \text{nil} \mid \text{trees.}\{x\}.\emptyset$				
destroy	$\text{true} \mid \text{trees.}\{x\}.\emptyset$	4-4	4	3	0.005
	$x = \text{nil} \mid \text{emp}$				
	$x \neq \text{nil} \mid \text{emp}$				
search	$\text{true} \mid \text{trees.}\{x\}.\emptyset$	2-4	4	3	0.006
	$x = \text{nil} \mid \text{emp}$				
	$x \neq \text{nil} \mid \text{trees.}\{x\}.\emptyset$				
toVine	$\text{true} \mid t \mapsto x', 0, 0 * \text{trees.}\{x\}.\emptyset$	3-4	8	6	0.061
	$\text{true} \mid t \mapsto \text{nil}, 0, 0$				
	$\text{true} \mid t \mapsto x', 0, 0 * x' \mapsto \text{nil}, \text{nil}, x''$				
	$\text{true} \mid t \mapsto x', 0, 0 * \text{trees.}\{x\}.\emptyset$				

Ejemplos

Algorithm	Precondition / Postcondition	R.L.	Inv	Iter	Time
insert	true $t \mapsto x', 0, 0 * \text{trees}.\{x\}.\emptyset$	3-3	12	4	0.036
	true $t \mapsto x', 0, 0 * x' \mapsto \text{nil}, \text{nil}, x''$				
	true $t \mapsto x', 0, 0 * \text{trees}.\{x\}.\emptyset$				
delete	true $t \mapsto x', 0, 0 * \text{trees}.\{x\}.\emptyset$	3-2	14	5	0.142
	true $t \mapsto \text{nil}, 0, 0$				
	true $t \mapsto x', 0, 0 * \text{trees}.\{x\}.\emptyset$				
Schorr	$r = x \text{trees}.\{x\}.\emptyset$	4-4	8	4	0.061
Waite	true emp				
	$x \neq \text{nil} \text{trees}.\{x\}.\emptyset$				
	$r \neq \text{nil} \text{trees}.\{r\}.\emptyset$				
Schorr	$r = x \text{graph}.\{x\}.\emptyset$	4-4	17	4	0.185
Waite	$x \neq \text{nil} \text{graph}.\{x\}.\emptyset$				
	$r \neq \text{nil} \text{graph}.\{r\}.\emptyset$				
	true emp				

Shape analysis basado en SL

- El análisis termina y es capaz de verificar propiedades interesantes y calcular invariantes de ciclos. Hereda todas las ventajas de la Separation Logic.
- Los nuevos predicados permiten la especificación simple de estructuras de datos (parciales) con forma de grafos generales, permitiendo la verificación de algoritmos complejos (Schorr-Waite, Cheney's)
- El nivel de relevancia de las variables introduce una mejora significativa en la compacidad de los estados abstractos sin modificar la precisión necesaria.

Conclusiones y Futuro

- La ShL permite “aumentar” la expresividad y definir un sistema de prueba composicional respetando el principio de modularidad e *information hiding*.
- Un Shape Analysis basado en SL permite la verificación automática de ciertas propiedades (mantenimiento del invariante de representación, ausencia de *memory leaks*, ausencia de dereferenciamiento a *null*, etc).
- Estamos trabajando en aumentar la capacidad expresiva de los Symbolic Heaps, definiendo una lógica **decidable** que permita la especificación de diversas estructuras de datos, *sharing*, etc.

Gracias.

¿Preguntas?