Pointer Analysis and Separation Logic

Élodie-Jane Sims
Elodie-Jane.Sims@polytechnique.edu

Thèse pour l’obtention du Doctorat de l’École Polytechnique 1er décembre 2007
Outline

1. Introduction
   (a) General
   (b) Introduction to separation logic

2. Contents of the thesis
   (a) Results
   (b) Adding fixpoints to separation logic
   (c) Pointer analysis: an abstract language to translate separation logic formulae

3. Comparisons

4. Conclusions
Analysing programs

Why?

- **Safety:** Programs are used in spaceships, public transportations, powerplants, banking,...

- **Money:** Debugging (finding errors) is a big part of the effort of programming

How?

- The industry usually does **testing** (trying to run the program in various situations) but this is not safe:
  - one cannot test a program on an infinity of values to cover all behaviours;
  - one cannot run a program forever before insuring it behaves properly forever.

- **Formal methods** try to address the problem by providing mathematically **sound techniques** that guarantee a full coverage of all program behaviours.
Requirements for our analyses

- are always **safe**: if we say no error, there are indeed no error possible, we cover all possible behaviours of the programs;

- can be **unprecise**;
  From undecidablity theorems: for any analyzer, there always exist programs for which it will answer “I don’t know” (or not terminates).

- always **terminates**;

- are **automatised**, we do not want to make proofs by hand.
The methodology: Abstraction

Example, the program

\[ x := y + 3; z := 3/x; \]

runs to a division error if \( x = 0 \) that is if \( y = -3 \).

We can not try all integer for \( y \) to find this \(-3\).

So we build an abstract domain, for example the sign domain and we get that:

<table>
<thead>
<tr>
<th>if ( y ) is</th>
<th>then ( x ) is</th>
<th>and the result is</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &gt; 0 )</td>
<td>( &gt; 0 )</td>
<td>no ERROR</td>
</tr>
<tr>
<td>( = 0 )</td>
<td>( &gt; 0 )</td>
<td>no ERROR</td>
</tr>
<tr>
<td>( &lt; 0 )</td>
<td>DONT_KNOW</td>
<td>DONT_KNOW</td>
</tr>
<tr>
<td>DONT_KNOW</td>
<td>DONT_KNOW</td>
<td>DONT_KNOW</td>
</tr>
</tbody>
</table>

If the answers are too imprecise, we refine our abstract domain, which means create or use a domain for which the answer is more costly to get but more precise.
Using logics to analyse programs

In the history of program analysis, people have often used Hoare logics as abstract domains.

Take a short program:

\[
\begin{align*}
x & := 3; \quad y := x;
\end{align*}
\]

You can run it starting with \(x\) and \(y\) equal to \(0\)

\[
\begin{align*}
[x \mapsto 0 \mid y \mapsto 0] \xrightarrow{x:=3;} [x \mapsto 3 \mid y \mapsto 0] \xrightarrow{y:=x;} [x \mapsto 3 \mid y \mapsto 3]
\end{align*}
\]

or you could also have

\[
\begin{align*}
[x \mapsto 5 \mid y \mapsto 2] \xrightarrow{x:=3;} [x \mapsto 3 \mid y \mapsto 2] \xrightarrow{y:=x;} [x \mapsto 3 \mid y \mapsto 3]
\end{align*}
\]

So people started to use logic to characterise the state before and after running a program:

\[
\begin{align*}
\{\text{true}\} x:=3; \quad y:=x; \quad \{x = 3 \land y = x\}
\end{align*}
\]
Automatisation and Search for precision:

We wrote

$$\{\text{true}\} \ x:=3; \ y:=x; \ \{x = 3 \land y = x\}$$

- we could also have written

$$\{x = 5\} \ x:=3; \ y:=x; \ \{x = 3 \land y = x\}$$

**Weakest precondition:**  $$\{\ ? \ } C \{Q\}$$

for a formula $Q$, and a program $C$ what is the least restrictive formula $P$ such that $\{P\} C \{Q\}$ is correct?

- we could also have written

$$\{\text{true}\} \ x:=3; \ y:=x; \ \{x = 3\}$$

**Strongest postcondition:**  $$\{P\} C \{\ ? \ }$$

for a formula, $P$, and a program, $C$, what is the most precise formula $Q$ such that $\{P\} C \{Q\}$ is correct?
The choice of the *abstract domain* is driven by the kind of program to analyse and the kind of property we want to prove.

We focused on programs using pointers, for what is called *pointer analysis*: check dereferencing errors, aliases, ...

Example of a pointer program with a bug, where \((a \leftarrow b, c)\) asserts that \(a\) points to a cons cell whose head value is \(b\) and tail value is \(c\):

\[
\begin{align*}
\uparrow & \{ \exists z_1, z_2. (\text{nil} \leftarrow z_1, z_2) \equiv \text{FALSE} \} \\
& \quad x := \text{nil} ; \\
& \quad \{ \exists z_1, z_2. (x \leftarrow z_1, z_2) \} \\
& \quad z := x ; \\
& \quad \{ \exists z_1, z_2. (z \leftarrow z_1, z_2) \} \\
& \quad y := z \cdot 1 ; \\
& \quad \{ \text{TRUE} \}
\end{align*}
\]
Pointers analyses: Shape/alias analyses

- **Shape analyses**: the analysis builds a graph where
  - the nodes represent locations in the heap
  - the edges represent fields between locations

  The analysis usually does approximation (represent more or less nodes/fields than what is in the heap) and computes some more informations about the nodes or edges of the graph.

- **Alias analyses**: a point-to analysis which computes sets of variables

  There have been and there are still tons of work on pointers: TVLA [POPL’99, SAS’00], Smallfoot, SpaceInvader [FMCO’05, SAS’07], Magill [SPACE’06], Whaley Rinard, Vivien Rinard [PLDI’01], Salcianu, Yang [ESOP’05], Rival [SAS’07], Andersen, Steensgaard, Heintze, Tzolovski, Foster Aiken [SAS’00], Ryder Landi, Emilianov, Deutsch, Jonkers, Møller, Reddy, ...
Separation logic: a logic for pointer analysis

Separation logic allows easy descriptions of memory states, e.g.

- $x$ points to a list of [1;2;3]
  \[ \exists x_2, x_3. (x \leftarrow 1, x_2) \ast (x_2 \leftarrow 2, x_3) \ast (x_3 \leftarrow 3, \text{nil}) \]

- $x$ and $y$ are aliased pointers
  \[ x = y \land \exists x_1, x_2. (x \leftarrow x_1, x_2) \]

- Partitioning: $x$ and $y$ belong to two disjoint parts of the heap which have no pointers from one to the other...
We wanted to use separation logic as an **interface** language for **modular analysis**.

$F, F'$: separation logic formulae; $D, D'$ other analysis's domain elements.

So we wanted to characterise programs with pre- and post-conditions in separation logic, and translate formulae into and from other domains. For this last point, we created an intermediate language into which we translate separation logic formulae.
Outline

1. Introduction
   (a) General
   (b) Introduction to separation logic

2. Contents of the thesis
   (a) Results
   (b) Adding fixpoints to separation logic
   (c) Pointer analysis: an abstract language to translate separation logic formulae

3. Comparisons

4. Conclusions
Example for a piece of code inserting a cell in a linked list
\{(x \mapsto 1, y) \ast (y \mapsto 3, \text{nil})\}

t := \text{cons}(2, y);

\{(x \mapsto 1, y) \ast (y \mapsto 3, \text{nil}) \ast (t \mapsto 2, y)\}

x \cdot 2 := t;

\{(x \mapsto 1, t) \ast (y \mapsto 3, \text{nil}) \ast (t \mapsto 2, y)\}
Local reasoning
\[
\left\{
\begin{array}{l}
(x \mapsto 1, y) \ast (y \mapsto 3, \text{nil}) \\
\ast (z \mapsto 4, y)
\end{array}
\right\}
\]

\[t := \text{cons}(2, y);\]

\[
\left\{
\begin{array}{l}
(x \mapsto 1, y) \ast (y \mapsto 3, \text{nil}) \ast (t \mapsto 2, y) \\
\ast (z \mapsto 4, y)
\end{array}
\right\}
\]

\[x \cdot 2 := t;\]

\[
\left\{
\begin{array}{l}
(x \mapsto 1, t) \ast (t \mapsto 2, y) \ast (y \mapsto 3, \text{nil}) \\
\ast (z \mapsto 4, y)
\end{array}
\right\}
\]
Separation logic

**Classical connectives**

- $E = E'$
- $P \Rightarrow Q$
- $\exists x. P$

**Spatial connectives**

- $\text{emp}$: Empty heap
- $P \ast Q$: Spatial conj.
- $E \leftrightarrow E_1, E_2$: Points to
- $P \rightarrow\!\ast Q$: Spatial imp.

Expressions can be:

- $x$
- $n$
- $\text{nil}$
- $\text{True}$
- $\text{False}$
- $E_1 \text{ op } E_2$
**Domain of interpretation: State**

We have a set of variables $Var$.

\[ Val = \text{Int} \cup \text{Bool} \cup \text{Atoms} \cup \text{Loc} \quad \text{Values} \]
\[ S = Var \rightarrow Val \quad \text{Stacks} \]
\[ H = Loc \rightarrow Val \times Val \quad \text{Heaps} \]
\[ State = S \times H \]

**Semantics of $\ast$**

\[
[P \ast Q]_\rho = \left\{ s, h_0 \cdot h_1 \mid
\begin{align*}
& \text{dom}(h_0) \cap \text{dom}(h_1) = \emptyset \\
& s, h_0 \in [P]_\rho \\
& s, h_1 \in [Q]_\rho
\end{align*}
\right\}
\]
Examples of formulae

Ex. 1

\[ s = [x \rightarrow l_1, y \rightarrow l_2] \]
\[ h_1 = [l_1 \rightarrow \langle 3, l_2 \rangle] \]
\[ \models (x \mapsto 3, y) \]

Ex. 2

\[ s = [x \rightarrow l_1, y \rightarrow l_2] \]
\[ h_2 = [l_2 \rightarrow \langle 4, l_1 \rangle] \]
\[ \models (y \mapsto 4, x) \]

Ex. 3

\[ s = [x \rightarrow l_1, y \rightarrow l_2] \]
\[ h_1 \cdot h_2 = \begin{bmatrix} l_1 \rightarrow \langle 3, l_2 \rangle, \\ l_2 \rightarrow \langle 4, l_1 \rangle \end{bmatrix} \]
\[ \models (x \mapsto 3, y) \ast (y \mapsto 4, x) \]
\[ \not\models (x \mapsto 3, y) \land (y \mapsto 4, x) \]
Outline

1. Introduction

2. Contents of the thesis
   (a) Results
   (b) Adding fixpoints to separation logic
   (c) Pointer analysis: an abstract language to translate separation logic formulae

3. Comparisons

4. Conclusions
**Result 1**

- When starting the work, recursive formulae could not be expressed within separation logic, and moreover pre-conditions \((wlp)\) and post-conditions \((sp)\) for while loops could not be expressed.

- We have extended separation logic such that we can express recursive formulae, and use them to instantiate existing triples rules and new ones.

- We have a backward \((wlp)\) and forward \((sp)\) analysis with their soundness proofs for any formula and any command, in particular for *while-loops*.

- We have proved *various properties* of the extended logic.
Result 2

We have built an intermediate language such that:

- it is similar to the existing shape/alias analysis domains to allow translations of our intermediate language from and to those existing domains

- it comes with a concrete semantics in term of sets of states which is the same domain as for the formulae’s semantics

- we translated the separation logic formulae into our intermediate language and proved sound those translations

- it is a partially reduced product of different subdomains so that we can cheaply tune the precision depending on the needs (for example, the language is parametrised by a numerical domain which can be ignored if we do not care about numericals)
Outline

1. Introduction

2. Contents of the thesis
   (a) Results
   (b) Adding fixpoints to separation logic
   (c) Pointer analysis: an abstract language to translate separation logic formulae

3. Comparisons

4. Conclusions
The logic: $BI^{\mu \nu}$

**Classical connectives**
- $E = E'$
- $P \Rightarrow Q$

**false**
- $\exists x. P$

**Spatial connectives**
- $\text{emp}$: Empty heap
- $P \ast Q$: Spatial conj.

**Points to**
- $E \mapsto E_1, E_2$
- $P \rightarrow* Q$

**Fixpoints connectives**
- $X_v$: Variable for formulae
- $\nu X_v . P$: Greatest fixpoint

**Postponed substitution**
- $P[E/x]$
- $\mu X_v . P$

**Least fixpoint**

$\text{Var}_v = \{ X_v, Y_v, \ldots \}$ infinite set of variables of formulae
Fixpoint connectives semantics

\( \rho \) is an environment mapping formula variables to sets of State
\[
\llbracket X_v \rrbracket_\rho = \rho(X_v) \text{ if } X_v \in \text{dom}(\rho)
\]

\[
\llbracket \mu X_v. P \rrbracket_\rho = \text{lfp}_\emptyset \lambda Y. \llbracket P \rrbracket_{[\rho|X_v \rightarrow Y]}
\]

\[
\llbracket \nu X_v. P \rrbracket_\rho = \text{gfp}_\emptyset \lambda Y. \llbracket P \rrbracket_{[\rho|X_v \rightarrow Y]}
\]

\[
\llbracket P[E/x] \rrbracket_\rho = \begin{cases} 
\{ s, h \mid \llbracket E \rrbracket^s \text{ exists and } \llbracket s \mid x \rightarrow \llbracket E \rrbracket^s\}, h \in \llbracket P \rrbracket_\rho \}
\end{cases}
\]

\( \llbracket \rrbracket \) may be undefined: if the formula is not closed for variables of formulae (e.g. \( \llbracket X_v \rrbracket_\emptyset \)) or if the fixpoint does not exists (e.g. \( \llbracket \mu X_v. \neg X_v \rrbracket_\rho \))
is not \{ / \}

\[ [\text{true}[y/x]] = \{s, h \mid [y]^s \text{ exists} \} \quad \text{Postponed substitution connective} \]

\[ [\text{true}\{y/x\}] = [\text{true}] = \text{State} \quad \text{Capture-avoiding substitution} \]

\{\text{true}\}x := y\{\text{true}\} \text{ is false since the command will be stuck from a state that has no value on its stack for } y

but \{is(y)\}x := y\{\text{true}\} \text{ is true}

so \{P\{y/x\}\}x := y\{P\} \text{ is unsound}

but \{P[y/x]\}x := y\{P\} \text{ is sound}

With \text{is}(E) \triangleq (E = E), \text{ since } [E = E]_\rho = \{s, h \mid [E]^s \text{ has a value}\}
List formula

“\(x\) points to a finite non-cyclic list of True”

\[
nclist(x) \triangleq \mu X_v. ((x = \text{nil}) \lor \exists x_2. (x \mapsto \text{true}, x_2 \ast X_v[x_2/x]))
\]

Notice the combination of fixpoint and postponed substitution to write recursive definitions

“\(nclist(x) = (x = \text{nil}) \lor \exists x_2. (x \mapsto \text{true}, x_2 \ast nclist(x_2))\)”

(to define finite or infinite lists replace \(\mu\) by \(\nu\))
Tree formula

“x points to a tree of True”

\[ \text{tree}(x) \triangleq \mu X_v . \ (x = \text{nil}) \lor \exists x_l, x_r, x'. \]
\[ ((x \mapsto \text{true}, x') \ast (x' \mapsto x_l, x_r) \ast X_v[x_l/x] \ast X_v[x_r/x]) \]
Unfolding theorems

As usual, the following theorems hold

\[ \mu X_v. P \equiv P\{\mu X_v. P / X_v\} \]

\[ \nu X_v. P \equiv P\{\nu X_v. P / X_v\} \]

We have proved some other theorems like variable renaming, variable substitution, equivalence of \( \mu \) and \( \nu \) using \( \neg \), simplifications of \([ / ]\) by equivalent formulae.
**Backward Analysis: wlp**

\( \text{wlp} \) : weakest liberal precondition, such that

\[ \{ \text{wlp}(P, C) \} C \{ P \} \text{ true} \]

\( \text{wlp} \) is expressed and proved sound for any \( P \) and any \( C \)

\( \forall \{ P[E/x] \} x := E \{ P \} \)

\( \forall (E = \text{true} \land \text{wlp}(X_v, C)) \lor (E = \text{false} \land P)) \}

\text{while } E \text{ do } C \{ P \} \)
**Forward analysis: $sp$**

$sp$ : strongest postcondition, such that

$$[[sp(P,C)]_\emptyset = \{m' \mid \exists m \in [[P]]_\emptyset. C, m \leadsto^* m'\}$$

$sp$ are expressed and proved sound for all $P$ and all $C$

♦ $sp(P, x := E) = \exists x'. P[x'/x] \land x = E\{x'/x\}$ with $x' \notin FV(E, P)$

♦ $sp(P, while E do C) = (E = false) \land (\mu X_v.sp(X_v \land E = true, C) \lor P)$
Outline

1. Introduction

2. Contents of the thesis
   (a) Results
   (b) Adding fixpoints to separation logic
   (c) **Pointer analysis: an abstract language to translate separation logic formulae**

3. Comparisons

4. Conclusions
Elements of the domain are tuples

Elements are 7-tuples \((sg, hu, ho, sn, sn^\infty, t, d)\)

- \(sg \in SG\)  
  A kind of shape graph
- \(hu \in \mathcal{P}(TVa\text{r})\)  
  Under approximation of heap domain
- \(ho \in \mathcal{P}(TVa\text{r}) \uplus \text{full}\)  
  Over approximation of heap domain
- \(sn \in \mathcal{P}(TVa\text{r})\)  
  Set of finite summary nodes
- \(sn^\infty \in \mathcal{P}(TVa\text{r})\)  
  Set of infinite summary nodes
- \(t \in TB\)  
  Tabular expressing inclusions on the concrete values represented
- \(d \in D\)  
  Numerical domain
Simple abstract values (Nilt, Truet,...) and disjunction

<table>
<thead>
<tr>
<th>Formulae</th>
<th>$x = \text{nil}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semantics</td>
<td>${s, h \mid s(x) = \text{nil}}$, ...</td>
</tr>
</tbody>
</table>
| Translation    | $\begin{array}{c}
                      \xrightarrow{x} \\
                      \text{Nilt}
                    \end{array}$ |

<table>
<thead>
<tr>
<th>Formulae</th>
<th>$(x = \text{nil} \lor x = \text{true})$</th>
</tr>
</thead>
</table>
| Translation    | $\begin{array}{c}
                      \xrightarrow{x} \\
                      \text{Nilt} \\
                      \xrightarrow{\text{Truet}}
                    \end{array}$ |
Aliasing and Conjunction
We want cheap translation of $\wedge: T(A \wedge B) \triangleq T'(\top, A, B)$

<table>
<thead>
<tr>
<th>Formula</th>
<th>$x = y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraints</td>
<td>refine the information for one variable while also refining the information of the second one in a cheap way</td>
</tr>
<tr>
<td>Adds</td>
<td>infinite set of auxiliary variables $TVar$</td>
</tr>
<tr>
<td>$VAR \triangleq Var \uplus TVar$</td>
<td></td>
</tr>
</tbody>
</table>

Translation

\[
\begin{pmatrix}
  x & \alpha & \top \\
  y & & \\
\end{pmatrix}
\]

Formula $x = y \wedge x = \text{nil}$

Translation

\[
\begin{pmatrix}
  x & \alpha & \text{Nilt} \\
  y & & \\
\end{pmatrix}
\]
### Quantifier

<table>
<thead>
<tr>
<th>Formula</th>
<th>$x = y \land x = \text{nil}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translation</td>
<td>![Translation Diagram](Translation Diagram)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Formula</th>
<th>$(\exists x. x = y \land x = \text{nil}) \equiv (y = \text{nil})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translation</td>
<td>![Translation Diagram](Translation Diagram)</td>
</tr>
</tbody>
</table>
### Numericals

<table>
<thead>
<tr>
<th>Formula</th>
<th>((x &lt; y + 3))</th>
</tr>
</thead>
</table>
| **Translation** | \[
\begin{align*}
  x &\rightarrow \alpha & \text{Numt} \\
  y &\rightarrow \beta & \text{Numt} \\
  d &\in \mathcal{D} & \text{encodes that } \alpha < \beta + 3
\end{align*}
\] |

\(d \in \mathcal{D}\) encodes that \(\alpha < \beta + 3\)
## Dangling pointers

<table>
<thead>
<tr>
<th>Formula</th>
<th>“x is a location not allocated”</th>
</tr>
</thead>
<tbody>
<tr>
<td>isdangling(x) (\equiv) isloc(x) (\wedge) (\neg)isinheap(x)</td>
<td></td>
</tr>
</tbody>
</table>

| Semantics | \(\{s, h \mid s(x) \in \text{Loc} \wedge s(x) \notin \text{dom}(h)\}\) |

| Translation | \(\left(\begin{array}{c} x \\
\xrightarrow{\text{Dangling\_Loc}} \end{array}\right)\) |

where

isint(x) \(\equiv\) \(\exists n. n = x + 1\)

isloc(x) \(\equiv\) \(\neg(x = \text{nil}) \wedge \neg(x = \text{true}) \wedge \neg(x = \text{false}) \wedge \neg(\text{isint}(x))\)

isinheap(x) \(\equiv\) \(\exists x_1, x_2. (x \leftrightarrow x_1, x_2)\)
**emp, approximation of the heap**

<table>
<thead>
<tr>
<th>Formula</th>
<th>emp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semantics</td>
<td>{ s, h \mid \text{dom}(h) = \emptyset }</td>
</tr>
</tbody>
</table>
| Subdomains | \( HU \triangleq \mathcal{P}(TVar) \)
\( HO \triangleq \mathcal{P}(TVar) \uplus \text{full} \) |
| Translation | \((\top, \emptyset, \emptyset, \_, \_, \_, \_)\) |
## Heap locations

<table>
<thead>
<tr>
<th>Formula</th>
<th>((x \mapsto \text{true}, \text{nil}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semantics</td>
<td>({s, h</td>
</tr>
<tr>
<td>Translation</td>
<td><img src="image" alt="Translation Diagram" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Formula</th>
<th>((x \mapsto \text{true}, \text{nil}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semantics</td>
<td>({s, h</td>
</tr>
<tr>
<td>Translation</td>
<td><img src="image" alt="Translation Diagram" /></td>
</tr>
</tbody>
</table>
### Summary nodes

Variables represent at most one value. To allow approximation we introduce summary nodes which can represent several values.

<table>
<thead>
<tr>
<th>Formula</th>
<th>approx. of ((x = \text{true} \land y = \text{nil})) by ((x = \text{true} \lor x = \text{nil}) \land (y = \text{true} \lor y = \text{nil}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translation</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>
**Finite acyclic list of True starting from x**

<table>
<thead>
<tr>
<th>Formula</th>
<th>$\mu X_v. \left( (x = \text{nil}) \lor \exists x_2. \left( x \mapsto (\text{true}, x_2) \ast X_v[x_2/x] \right) \right)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translation</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>

∅ is the set of infinite summary nodes, for infinite list $\mu$ would be replaced by $\nu$ and ∅ by \{α\}.
Tabular to increase precision of union

<table>
<thead>
<tr>
<th>Formula</th>
<th>( (x = \text{nil} \land y = \text{true}) \lor (x = \text{true} \land y = \text{nil}) )</th>
</tr>
</thead>
</table>
| Translation |<table><tr><td>\( \alpha_1 \) \{\hat{eq} \}
\( \alpha_2 \) \{\hat{eq} \}
\( \alpha_3 \) \{\hat{eq} \}
\( \alpha_4 \) \{\hat{eq} \}</td></tr></table>|

The dashed arrows are drawn to represent the tabular:

<table>
<thead>
<tr>
<th>( \alpha_1 )</th>
<th>( \alpha_2 )</th>
<th>( \alpha_3 )</th>
<th>( \alpha_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( {\hat{eq}, = eq} )</td>
<td>( {\hat{eq} } )</td>
<td>( {\hat{eq} } )</td>
<td>( \top )</td>
</tr>
<tr>
<td>( \alpha_2 )</td>
<td>( {\hat{eq}, = eq} )</td>
<td>( \top ) eq</td>
<td>( {\hat{eq} } )</td>
</tr>
<tr>
<td>( \alpha_3 )</td>
<td>( {\hat{eq}, = eq} )</td>
<td>( {\hat{eq} } )</td>
<td>( {\hat{eq} } )</td>
</tr>
<tr>
<td>( \alpha_4 )</td>
<td>( {\hat{eq}, = eq} )</td>
<td>( {\hat{eq} } )</td>
<td>( {\hat{eq}, = eq} )</td>
</tr>
</tbody>
</table>
Operations

We have proved soundness of the operations we use, in particular:

- **union, intersection**

- **extension** (replace $v_1 \to v \to Nilt$ by $v_1 \to \alpha \to Nilt$)

  or by

  $v_1 \to v \to \alpha \to Nilt$

  used to tune the precision of the union

- **merging** (replace $[v_1 \to S_1 \mid v_2 \to S_2]$ by $[v_2 \to (S_1 \cup S_2)]$)

  used with the **widening**

- **translations** from formulae to the abstract language
Comparisons

- the \( \bullet \rightarrow \) represent nodes in the usual shape graphs
- summary nodes, as for other shape graphs, seems to give more possibilities than predicate abstraction (with each time a specific predicate for list, etc...) but the framework of predicate abstraction and their algorithm/heuristics (like folding/unfolding) could probably also be use on our graphs
- a lonely outgoing edge can be seen as a “must” arrow (or valued 1), several outgoing edges from a variable can be seen as a “may” arrow (or valued 1/2, but it is a bit more precise because we know that one of them should exist), and an edge to \( \emptyset \) can be seen as a “must not” arrow (or valued 0)
- we deal with numerical information (not many works do, for example Magill & al. also do)
- we have a formal semantic of our domain, the semantics of auxiliary variables are formally defined and formally used in the proofs. We don’t have to check for equalities of variables
- we directly have in the domain the “Dangling” information which is suitable for cleaning checking
We have built a prototype implementation:

- Data of type `Prog.t` defined in `program.ml` is computed by `prog2sl.ml`.
- Data of type `Form.t` defined in `formula.ml` is computed by `sl2ar.ml`.
- Data of type `AR.t` defined in `ar.ml` uses subtypes defined in `ad.ml`, `heap.ml`, `sn.ml`, `tb.ml`, `cleq.ml`, `numerical_domain.ml`, and they all use `all_domain.ml`.

Implement the computation of pre- and post-conditions in the extended separation logic and the translation of the formulae into the abstract language.
Conclusions

✓ We added fixpoints to separation logic, which provides a way to express recursive formulae and while-loops pre- and post-conditions.

✓ We proved useful properties about the extended logic

✓ We gave a precise semantics of the abstract language for separation formulae in terms of sets of states. We gave a semantics to auxiliary variables and did not leave this as an implementation design question.

✓ We designed the abstract language as a partially reduced product of subdomains. We combined the domain’s heap analysis with a numerical domain which could be chosen from existing ones (e.g. polyhedra, octogons).

✓ We designed a novel tabular data structure which allows extra precision by using a graph of sets instead of sets of graphs.

✓ We expressed and proved the translation of separation logic formulae into our abstract language and implemented it in a prototype.
Future work

• finish the prototype implementation, profiling studies with standard example programs, experiment various strategies to build summary nodes
• improve the abstract language with labels indicating where we do overapproximation
• add sugar structures to do abstract language like lists (or a system to add those structures) and functions to use the mechanisms of folding/unfolding those structures when needed
• adding labels to *1 and *2 to allow to have several families of uncycling edges instead of one
• it could be fun to design a program analysis directly in our abstract language
• actually do interface the abstract language with some existing pointer analysis
End