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 programing

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

$$\boldsymbol{x := y + 3; z := 3/x;}$$

runs to a division error if $\boldsymbol{x = 0}$ that is if $\boldsymbol{y = -3}$.

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:

if y is	then x is	and the result is
> 0	> 0	no ERROR
$= 0$	> 0	no ERROR
< 0	DONT_KNOW	DONT_KNOW
DONT_KNOW	DONT_KNOW	DONT_KNOW

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:

$$\boldsymbol{x := 3;\ y := x;}$$

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

$$[x \mapsto 0 \mid y \mapsto 0] \xrightarrow{\texttt{x:=3;}} [x \mapsto 3 \mid y \mapsto 0] \xrightarrow{\texttt{y:=x;}} [x \mapsto 3 \mid y \mapsto 3]$$

or you could also have

$$[x \mapsto 5 \mid y \mapsto 2] \xrightarrow{\texttt{x:=3;}} [x \mapsto 3 \mid y \mapsto 2] \xrightarrow{\texttt{y:=x;}} [x \mapsto 3 \mid y \mapsto 3]$$

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

$$\{\texttt{true}\}\ \texttt{x:=3; y:=x;}\ \{\boldsymbol{x = 3 \wedge y = x}\}$$

Automatisation and Search for precision:

We wrote

$$\{\texttt{true}\}\ \texttt{x:=3; y:=x;}\ \{x = 3 \wedge y = x\}$$

- we could also have written

$$\{x = 5\}\ \texttt{x:=3; y:=x;}\ \{x = 3 \wedge y = x\}$$

  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

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

  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 ?

Pointer programs

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 $(a \hookrightarrow b, c)$ asserts that a points to a cons cell whose head value is b and tail value is c:

$$
\uparrow \quad
\begin{array}{c}
\{\exists z_1, z_2.\ (\mathtt{nil} \hookrightarrow z_1, z_2) \equiv \boldsymbol{FALSE}\} \\
\boldsymbol{x := \mathtt{nil};} \\
\{\exists z_1, z_2.\ (x \hookrightarrow z_1, z_2)\} \\
\boldsymbol{z := x;} \\
\{\exists z_1, z_2.\ (z \hookrightarrow z_1, z_2)\} \\
\boldsymbol{y := z \cdot 1;} \\
\{\boldsymbol{TRUE}\}
\end{array}
$$

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

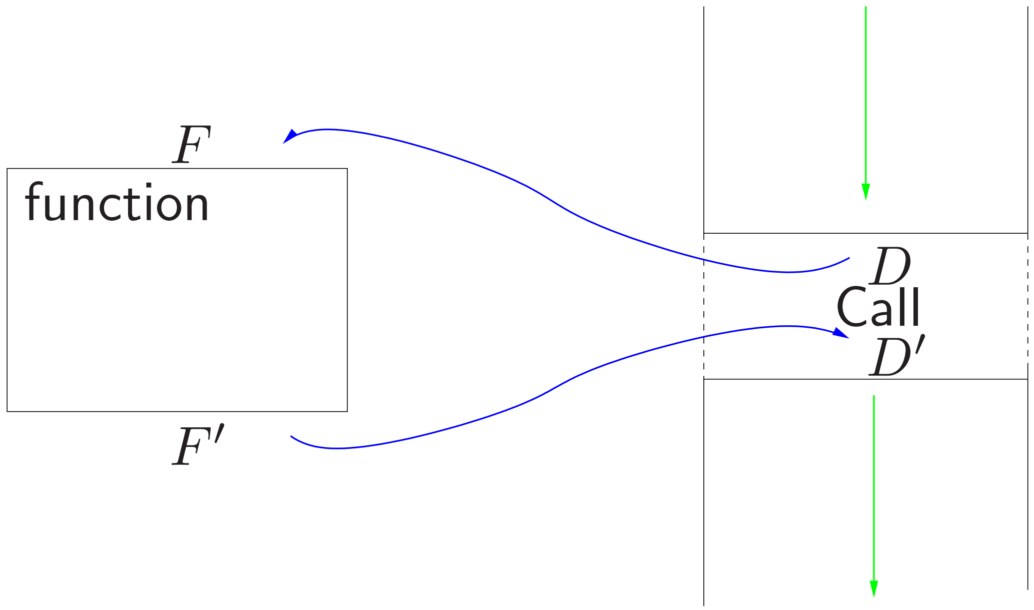
Separation logic allows easy descriptions of memory states, e.g.

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

- x and y are aliased pointers
  $x = y \wedge \exists x_1, x_2.\ (x \hookrightarrow x_1, x_2)$

- 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



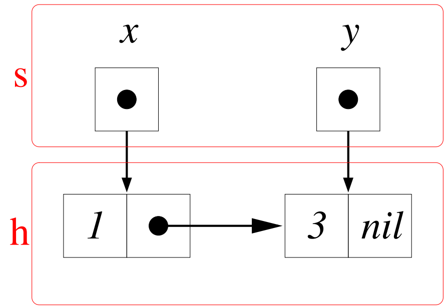
F, F': sep. logic formulae; D, D' other analysis's domain elements.

So we wanted to characterise programs with pre- and post-conditions in sep. 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

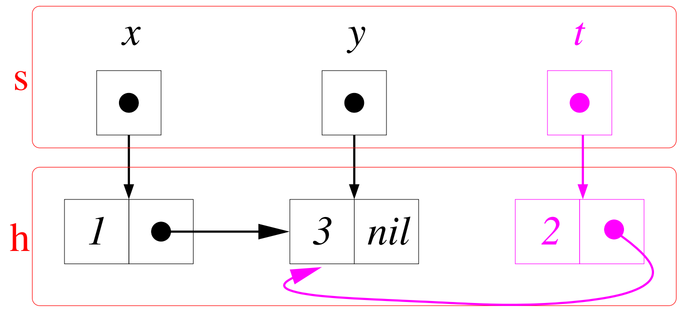
1. Introduction
   (a) General
   (b) **Introduction to separation logic**
2. Contents of the thesis
   (a) Results
   (b) Adding fixpoints to separation logic
   (c) Pointer analysis: an abstract language to translate separation logic formulae
3. Comparisons
4. Conclusions

Example for a piece of code inserting a cell in a linked list



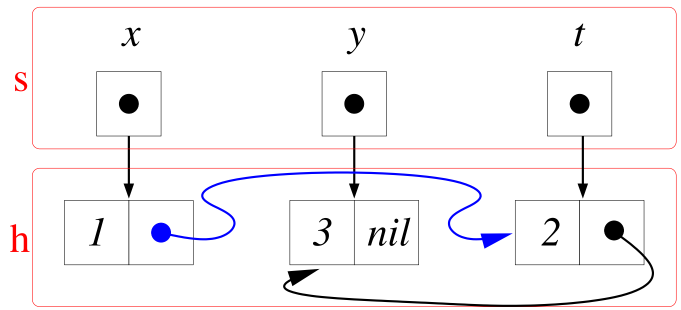
$$\{(x \mapsto 1, y) * (y \mapsto 3, \mathtt{nil})\}$$

$$\boldsymbol{t := \mathtt{cons}(2, y)};$$



$$\{(x \mapsto 1, y) * (y \mapsto 3, \mathtt{nil}) \boldsymbol{* (t \mapsto 2, y)}\}$$

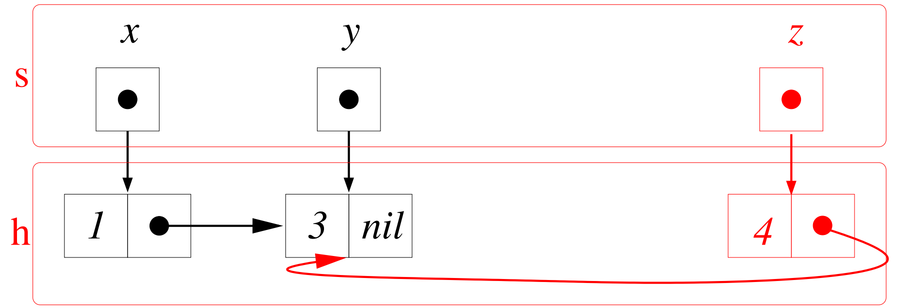
$$\boldsymbol{x \cdot 2 := t};$$



$$\{(x \mapsto 1, \boldsymbol{t}) * (y \mapsto 3, \mathtt{nil}) * (t \mapsto 2, y)\}$$

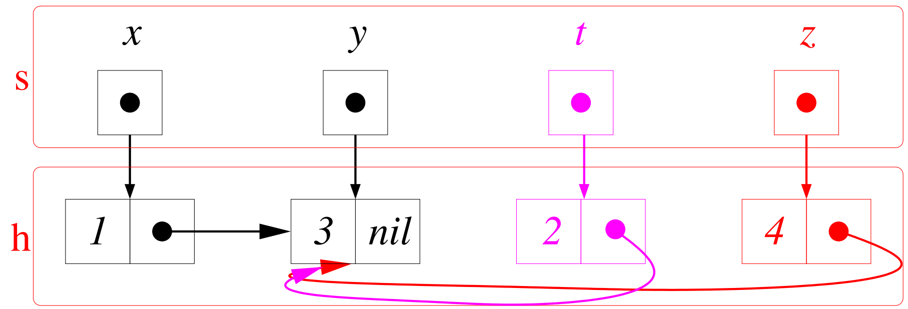
Local reasoning

$$\left\{ \begin{array}{l} (x \mapsto 1, y) * (y \mapsto 3, \mathtt{nil}) \\ \boldsymbol{*(z \mapsto 4, y)} \end{array} \right\}$$



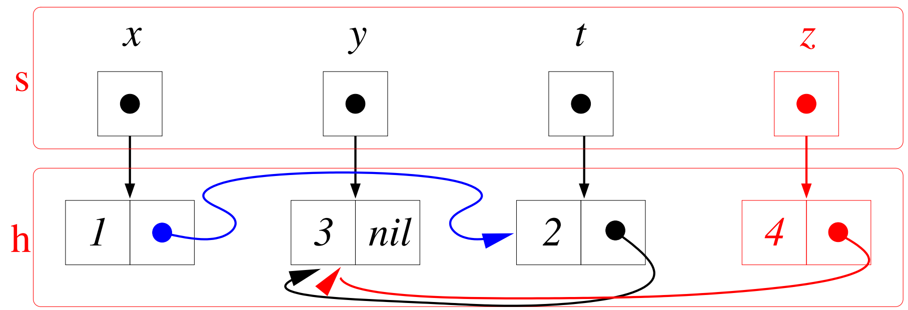
$$\boldsymbol{t := \mathtt{cons}(2, y)};$$

$$\left\{ \begin{array}{l} (x \mapsto 1, y) * (y \mapsto 3, \mathtt{nil}) \boldsymbol{*(t \mapsto 2, y)} \\ \boldsymbol{*(z \mapsto 4, y)} \end{array} \right\}$$



$$\boldsymbol{x \cdot 2 := t};$$

$$\left\{ \begin{array}{l} (x \mapsto 1, \boldsymbol{t}) * (t \mapsto 2, y) * (y \mapsto 3, \mathtt{nil}) \\ \boldsymbol{*(z \mapsto 4, y)} \end{array} \right\}$$



Separation logic

$$
\begin{array}{llllll}
 & \textit{Classical} & \textit{connectives} & & & \\
 & E = E' & & \mid & \texttt{false} & \\
\mid & P \Rightarrow Q & & \mid & \exists x.P & \\
 & & & & & \\
 & \textit{Spatial} & \textit{connectives} & & & \\
\mid & \texttt{emp} & \text{Empty heap} & \mid & E \mapsto E_1, E_2 & \text{Points to} \\
\mid & P * Q & \text{Spatial conj.} & \mid & P -\!\!* \; Q & \text{Spatial imp.}
\end{array}
$$

Expressions can be:

$$x \mid n \mid \texttt{nil} \mid True \mid False \mid E_1 \;\; op \;\; E_2$$

Domain of interpretation: $State$

We have a set of variables Var.

$$
\begin{array}{rcll}
Val & = & Int \cup Bool \cup Atoms \cup Loc & Values \\
S & = & Var \rightharpoonup Val & Stacks \\
H & = & Loc \rightharpoonup Val \times Val & Heaps \\
State & = & S \times H &
\end{array}
$$

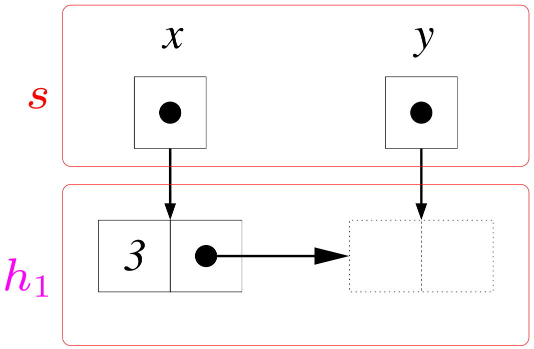
Semantics of $*$

$$
[\![P * Q]\!]_\rho = \left\{ s, h_0 \cdot h_1 \;\middle|\; \begin{array}{ll} \bullet & \boldsymbol{dom(h_0) \cap dom(h_1) = \emptyset} \\ \bullet & s, h_0 \in [\![P]\!]_\rho \\ \bullet & s, h_1 \in [\![Q]\!]_\rho \end{array} \right\}
$$

Examples of formulae

Ex. 1

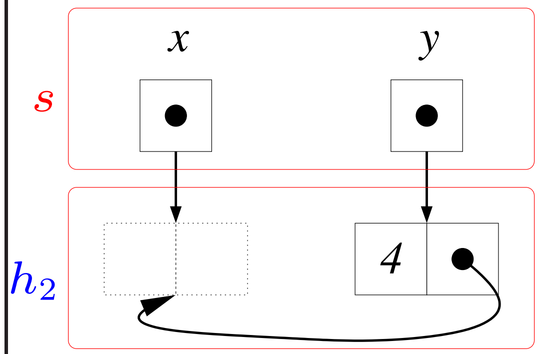
$$s = [x \to l_1, y \to l_2]$$
$$h_1 = [l_1 \to \langle 3, l_2 \rangle]$$



$$\models \quad (x \mapsto 3, y)$$

Ex. 2

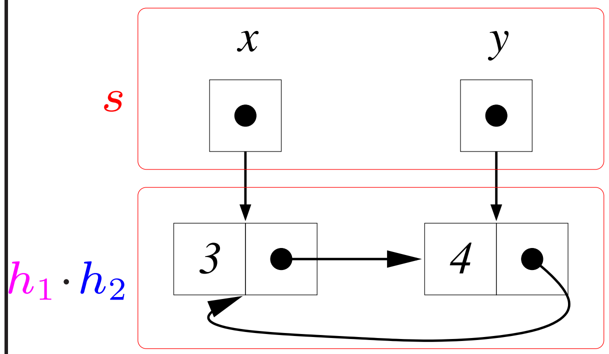
$$s = [x \to l_1, y \to l_2]$$
$$h_2 = [l_2 \to \langle 4, l_1 \rangle]$$



$$\models (y \mapsto 4, x)$$

Ex. 3

$$s = [x \to l_1, y \to l_2]$$
$$h_1 \cdot h_2 = \begin{bmatrix} l_1 \to \langle 3, l_2 \rangle, \\ l_2 \to \langle 4, l_1 \rangle \end{bmatrix}$$



$$\models (x \mapsto 3, y) * (y \mapsto 4, x)$$
$$\not\models (x \mapsto 3, y) \wedge (y \mapsto 4, x)$$

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*			
	$E = E'$		$\mid$	`false`	
$\mid$	$P \Rightarrow Q$		$\mid$	$\exists x.P$	
	Spatial	*connectives*			
$\mid$	`emp`	Empty heap	$\mid$	$E \mapsto E_1, E_2$	Points to
$\mid$	$P * Q$	Spatial conj.	$\mid$	$P \mathrel{-\!\!*} Q$	Spatial imp.
	Fixpoints	*connectives*			
$\mid$	X_v	Variable for formulae	$\mid$	$P[E/x]$	Postponed substitution
$\mid$	$\nu X_v.P$	Greatest fixpoint	$\mid$	$\mu X_v.P$	Least fixpoint

$Var_v = \{X_v, Y_v, ...\}$ infinite set of variables of formulae

Fixpoint connectives semantics

ρ is an environment mapping formula variables to sets of State

$$\llbracket X_v \rrbracket_\rho = \rho(X_v) \text{ if } X_v \in dom(\rho)$$

$$\llbracket \mu X_v.P \rrbracket_\rho = \mathrm{lfp}^{\subseteq}_{\emptyset} \lambda Y.\, \llbracket P \rrbracket_{[\rho | X_v \to Y]}$$

$$\llbracket \nu X_v.P \rrbracket_\rho = \mathrm{gfp}^{\subseteq}_{\emptyset} \lambda Y.\, \llbracket P \rrbracket_{[\rho | X_v \to Y]}$$

$$\llbracket P[E/x] \rrbracket_\rho = \left\{ s, h \mid \begin{array}{l} \llbracket E \rrbracket^s \text{ exists and} \\ [s \mid x \to \llbracket E \rrbracket^s], h \in \llbracket P \rrbracket_\rho \end{array} \right\}$$

$\llbracket\ \rrbracket$ 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 $\{\ /\ \}$

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

$[\![\mathtt{true}\{y/x\}]\!] = [\![\mathtt{true}]\!] = \mathit{State}$ Capture-avoiding substitution

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

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

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

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

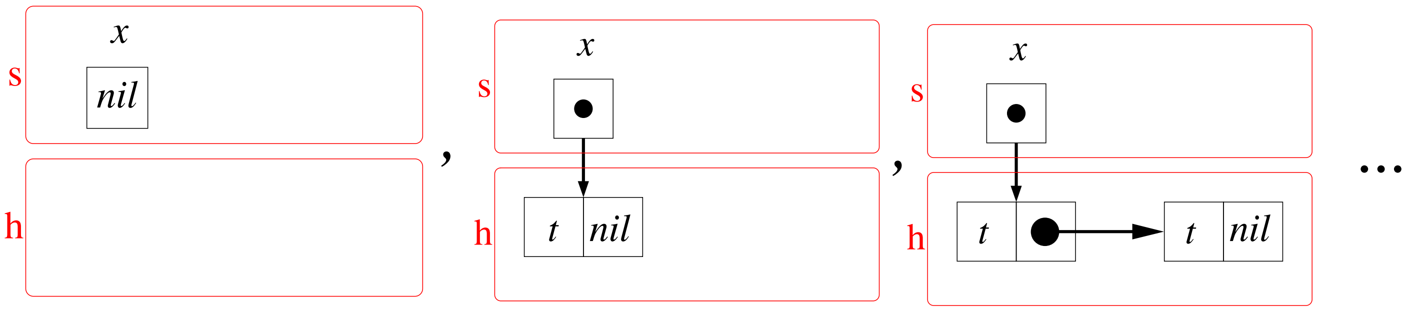
With $is(E) \triangleq (E = E)$, since $[\![E = E]\!]_\rho = \{s, h \mid [\![E]\!]^s \text{ has a value}\}$

List formula

"x points to a finite non-cyclic list of True"

$$\mathtt{nclist}(x) \triangleq \boldsymbol{\mu X_v}.\ ((x = \mathtt{nil}) \vee \exists x_2.\ (x \mapsto \mathtt{true}, x_2\ *\ \boldsymbol{X_v[x_2/x]}))$$

Notice the combination of fixpoint and postponed substitution to write recursive definitions
"$\mathtt{nclist}(x) = (x = \mathtt{nil}) \vee \exists x_2.(x \mapsto \mathtt{true}, x_2 * \mathtt{nclist}(x_2))$"



(to define finite or infinite lists replace $\boldsymbol{\mu}$ *by* $\boldsymbol{\nu}$*)*

Tree formula

"x points to a tree of True"

$$\texttt{tree}(x) \triangleq \boldsymbol{\mu X_v}. \begin{array}{l}(x = \texttt{nil}) \vee \exists x_l, x_r, x'. \\ ((x \mapsto \texttt{true}, x') \ * \ (x' \mapsto x_l, x_r) \ * \ \boldsymbol{X_v[x_l/x]} \ * \ \boldsymbol{X_v[x_r/x]})\end{array}$$

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 μ and ν using $\neg$, simplifications of $[\ /\ ]$ by equivalent formulae.

Backward Analysis: wlp

wlp : weakest liberal precondition, such that

$$\{wlp(P, C)\}C\{P\} \text{ true}$$

wlp is expressed and proved sound for any P and any C

- $\{P[E/x]\}x := E\{P\}$

- $\{\boldsymbol{\nu X_v}.\ ((E = \texttt{true} \wedge wlp(\boldsymbol{X_v}, C)) \vee (E = \texttt{false} \wedge P))\}$
  $\texttt{while}\ E\ \texttt{do}\ C\ \{P\}$

Forward analysis: sp

sp : strongest postcondition, such that

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

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

- $sp(P, x := E) = \exists x'.\ P[x'/x] \wedge x = E\{x'/x\}$ with $x' \notin FV(E, P)$
- $sp(P, \texttt{while}\ E\ \texttt{do}\ C) =$
  $(E = \texttt{false}) \wedge (\boldsymbol{\mu X_v}.sp(\boldsymbol{X_v} \wedge E = \texttt{true}, C) \vee P)$

Outline

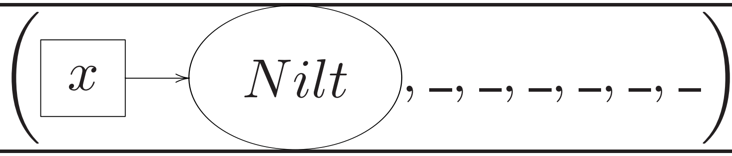
1. Introduction
2. Contents of the thesis
   (a) Results
   (b) Adding fixpoints to separation logic
   (c) **Pointer analysis: an abstract language to translate separation logic formulae**
3. Comparisons
4. Conclusions

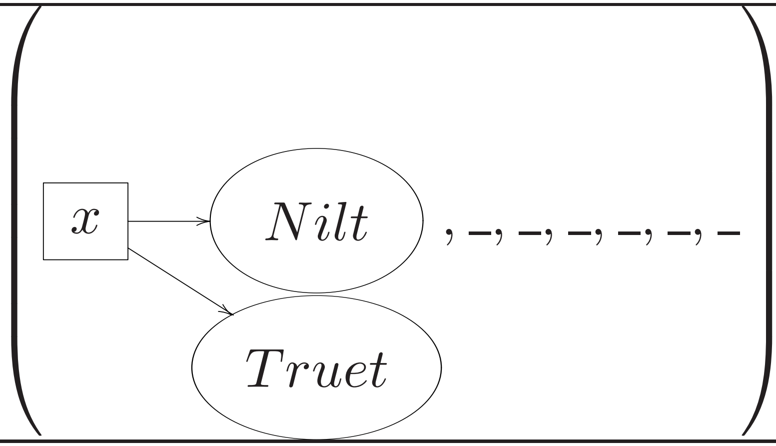
Elements of the domain are tuples

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

$sg \in SG$	A kind of shape graph
$hu \in \mathcal{P}(TVar)$	Under approximation of heap domain
$ho \in \mathcal{P}(TVar) \uplus \texttt{full}$	Over approximation of heap domain
$sn \in \mathcal{P}(TVar)$	Set of finite summary nodes
$sn^\infty \in \mathcal{P}(TVar)$	Set of infinite summary nodes
$t \in TB$	Tabular expressing inclusions on the concrete values represented
$d \in \mathcal{D}$	Numerical domain

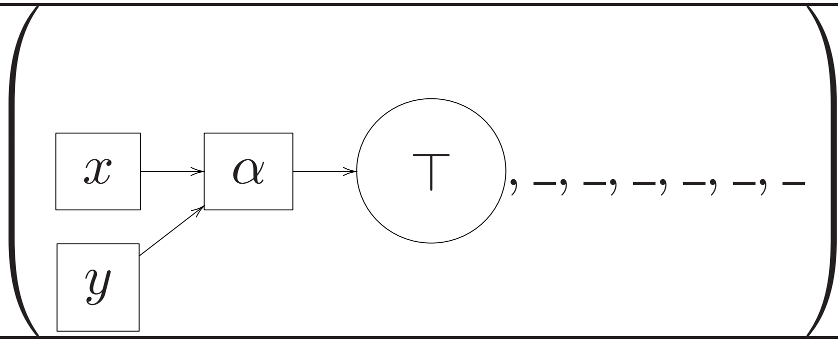
Simple abstract values (Nilt, Truet,...) and disjunction

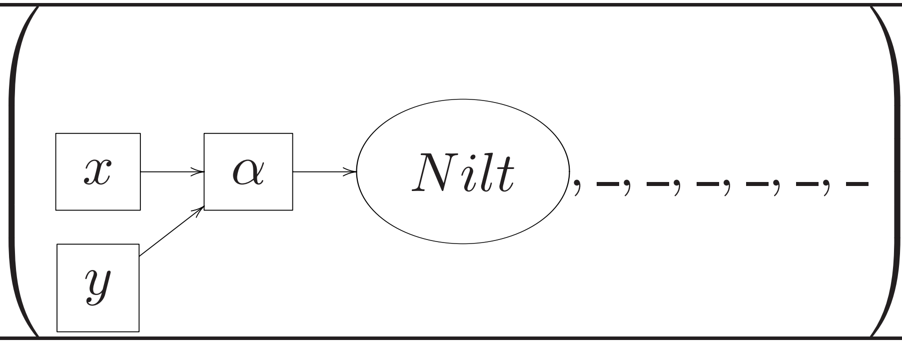
Formulae	$x = \texttt{nil}$
Semantics	$\{s, h \mid s(x) = \texttt{nil}\}$,...
Translation	

Formulae	$(x = \texttt{nil} \vee x = \texttt{true})$
Translation	

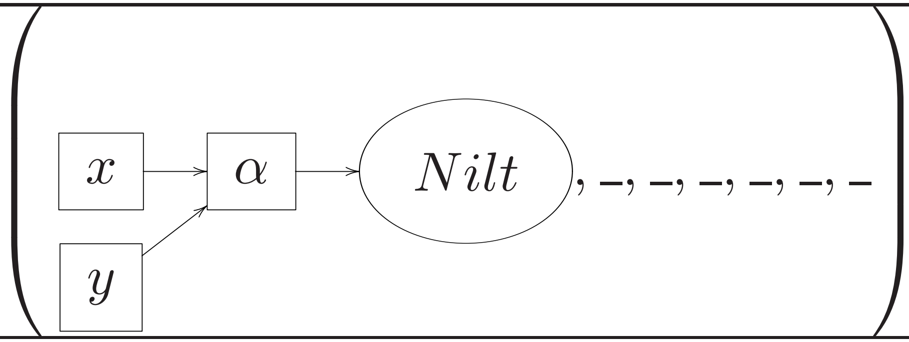
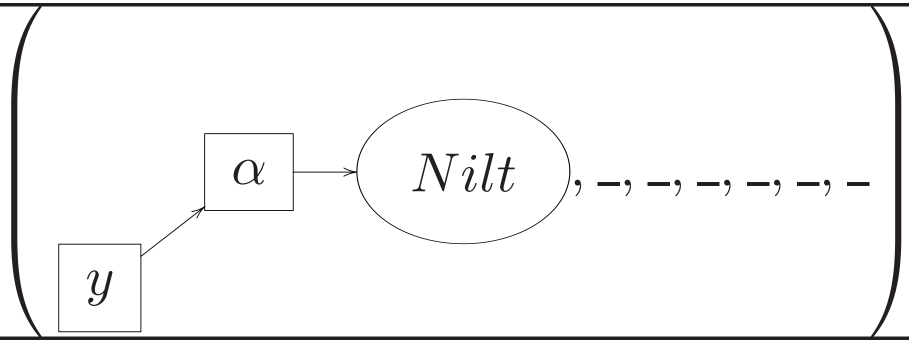
Aliasing and Conjunction

We want cheap translation of $\wedge$: $T(A \wedge B) \triangleq T'(T'(\top, A), B)$

Formula	$x = y$
Constraints	refine the information for one variable while also refining the information of the second one in a cheap way
Adds	infinite set of auxiliary variables $TVar$ $VAR \triangleq Var \uplus TVar$
Translation	

Formula	$x = y \wedge x = \texttt{nil}$
Translation	

Quantifier

<table>
<tr><td>Formula</td><td>$x = y \wedge x = \texttt{nil}$</td></tr>
<tr><td>Translation</td><td>
</td></tr>
<tr><td>Formula</td><td>$(\exists x.\ x = y \wedge x = \texttt{nil}) \equiv (y = \texttt{nil})$</td></tr>
<tr><td>Translation</td><td>
</td></tr>
</table>

Numericals

Formula	$(x < y + 3)$
Translation	$\left(\begin{array}{l} \boxed{x} \to \boxed{\alpha} \to \textcircled{Numt} \\ \boxed{y} \to \boxed{\beta} \to \textcircled{Numt} \end{array}, \_, \_, \_, \_, \_, d \right)$ $d \in \mathcal{D}$ encodes that $\alpha < \beta + 3$

Dangling pointers

Formula	"x is a location not allocated" $\texttt{isdangling}(x) \equiv \texttt{isloc}(x) \wedge \neg\texttt{isinheap}(x)$
Semantics	$\{s, h \mid s(x) \in Loc \wedge s(x) \notin dom(h)\}$
Translation	$\left(\boxed{x} \rightarrow \textit{Dangling\_Loc}, \_, \_, \_, \_, \_, \_\right)$

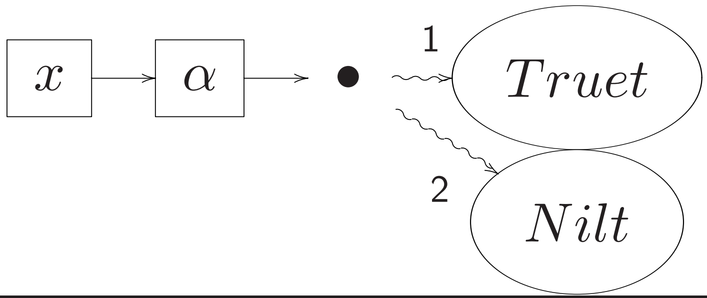
where

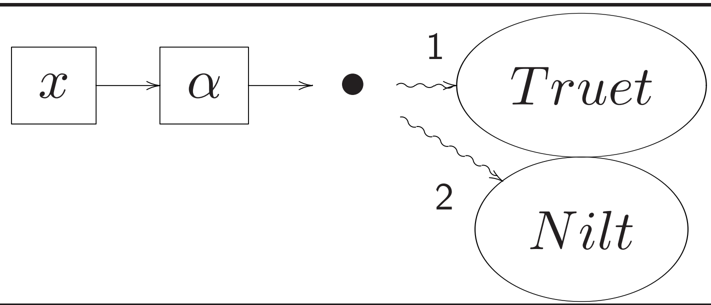
$$
\begin{aligned}
\texttt{isint}(x) &\equiv \exists n.\, n = x + 1 \\
\texttt{isloc}(x) &\equiv \neg(x = \texttt{nil}) \wedge \neg(x = \texttt{true}) \wedge \neg(x = \texttt{false}) \wedge \neg(\texttt{isint}(x)) \\
\texttt{isinheap}(x) &\equiv \exists x_1, x_2.\, (x \hookrightarrow x_1, x_2)
\end{aligned}
$$

emp, approximation of the heap

Formula	emp
Semantics	$\{s, h \mid dom(h) = \emptyset\}$
Subdomains	$HU \triangleq \mathcal{P}(TVar)$ $HO \triangleq \mathcal{P}(TVar) \uplus \texttt{full}$
Translation	$(\top, \emptyset, \emptyset, \_, \_, \_, \_)$

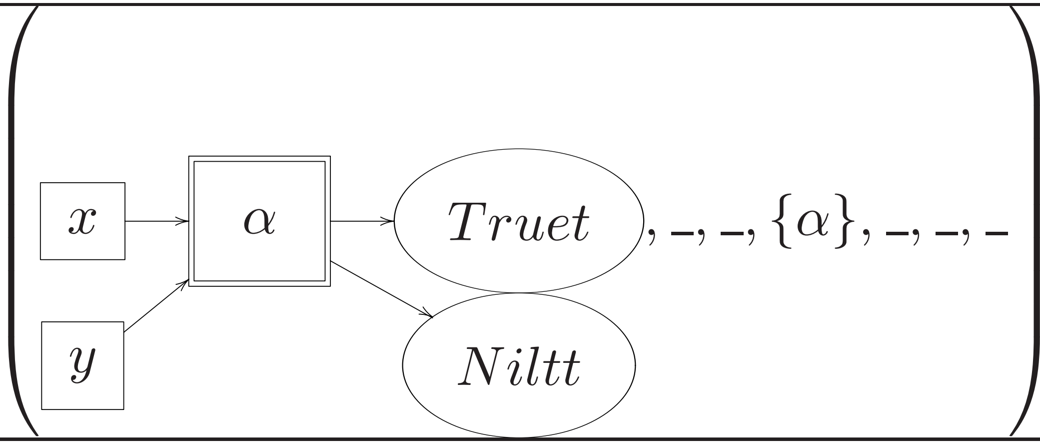
Heap locations

Formula	$(x \mapsto \texttt{true}, \texttt{nil})$
Semantics	$\{s, h \vert [s(x) \to \langle True, \texttt{nil} \rangle] = \boldsymbol{h}\}$
Translation	( $, \{\alpha\}, \{\alpha\}, \_, \_, \_, \_$)

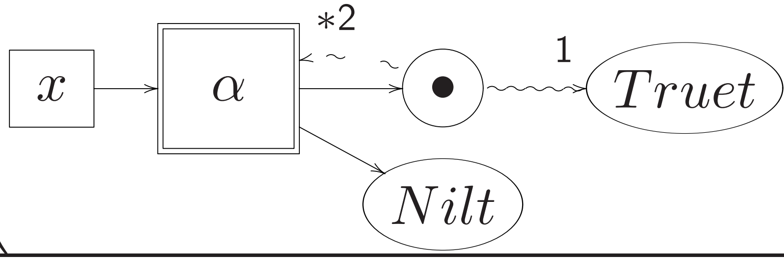
Formula	$(x \hookrightarrow \texttt{true}, \texttt{nil})$
Semantics	$\{s, h \vert [s(x) \to \langle True, \texttt{nil} \rangle] \subseteq \boldsymbol{h}\}$
Translation	( $, \{\alpha\}, \texttt{full}, \_, \_, \_, \_$)

Summary nodes

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

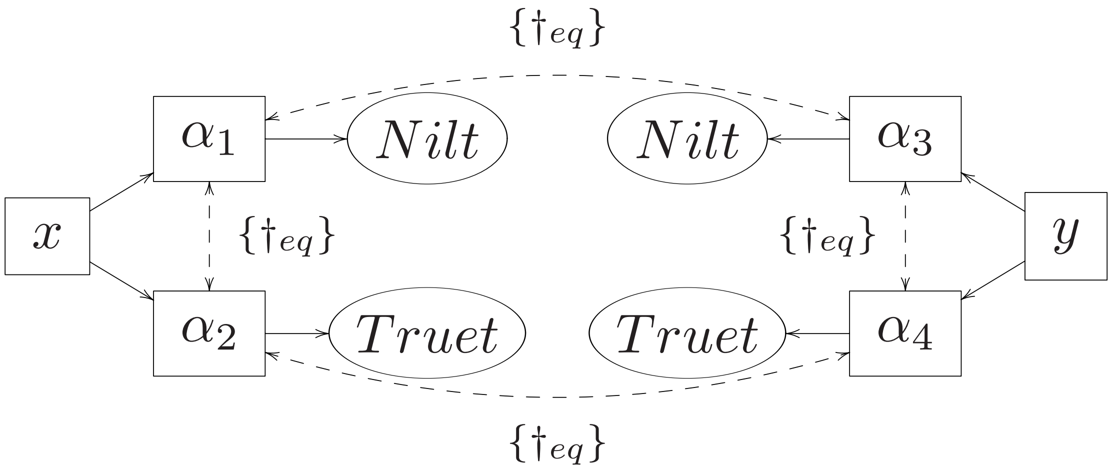
Formula	approx. of $(x = \texttt{true} \wedge y = \texttt{nil})$ by $\begin{array}{l}(x = \texttt{true} \ \vee \ x = \texttt{nil}) \\ \wedge (y = \texttt{true} \ \vee \ y = \texttt{nil})\end{array}$
Translation	

Finite acyclic list of $True$ starting from x

Formula	$\mu X_v.\left(\begin{array}{c}(x = \mathtt{nil}) \vee \exists x_2.\\ x \hookrightarrow (\mathtt{true}, x_2) \; * \; X_v[x_2/x]\end{array}\right)$
Translation	$\Big($  $, \_, \_, \{\alpha\}, \emptyset, \_, \_ \Big)$

$\emptyset$ is the set of infinite summary nodes, for infinite list μ would be replaced by ν and $\emptyset$ by $\{\alpha\}$.

Tabular to increase precision of union

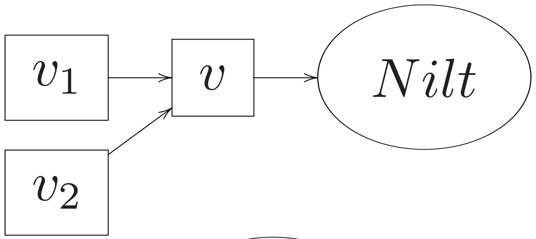
Formula	$\begin{pmatrix} x = \texttt{nil} \\ \wedge y = \texttt{true} \end{pmatrix} \vee \begin{pmatrix} x = \texttt{true} \\ \wedge y = \texttt{nil} \end{pmatrix}$
Translation	

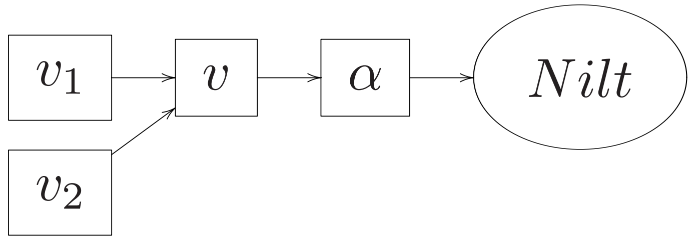
The dashed arrows are drawned to represent the tabular:

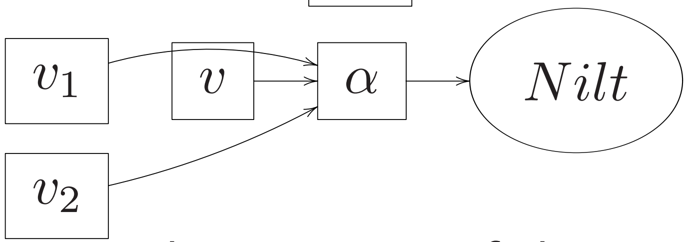
	α_1	α_2	α_3	α_4
α_1	$\{\ddagger_{eq}, =_{eq}\}$	$\{\dagger_{eq}\}$	$\{\dagger_{eq}\}$	$\top_{eq}$
α_2		$\{\ddagger_{eq}, =_{eq}\}$	$\top_{eq}$	$\{\dagger_{eq}\}$
α_3			$\{\ddagger_{eq}, =_{eq}\}$	$\{\dagger_{eq}\}$
α_4				$\{\ddagger_{eq}, =_{eq}\}$

Operations

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

- **union**, **intersection**
- **extension** (replace 

by 

or by 

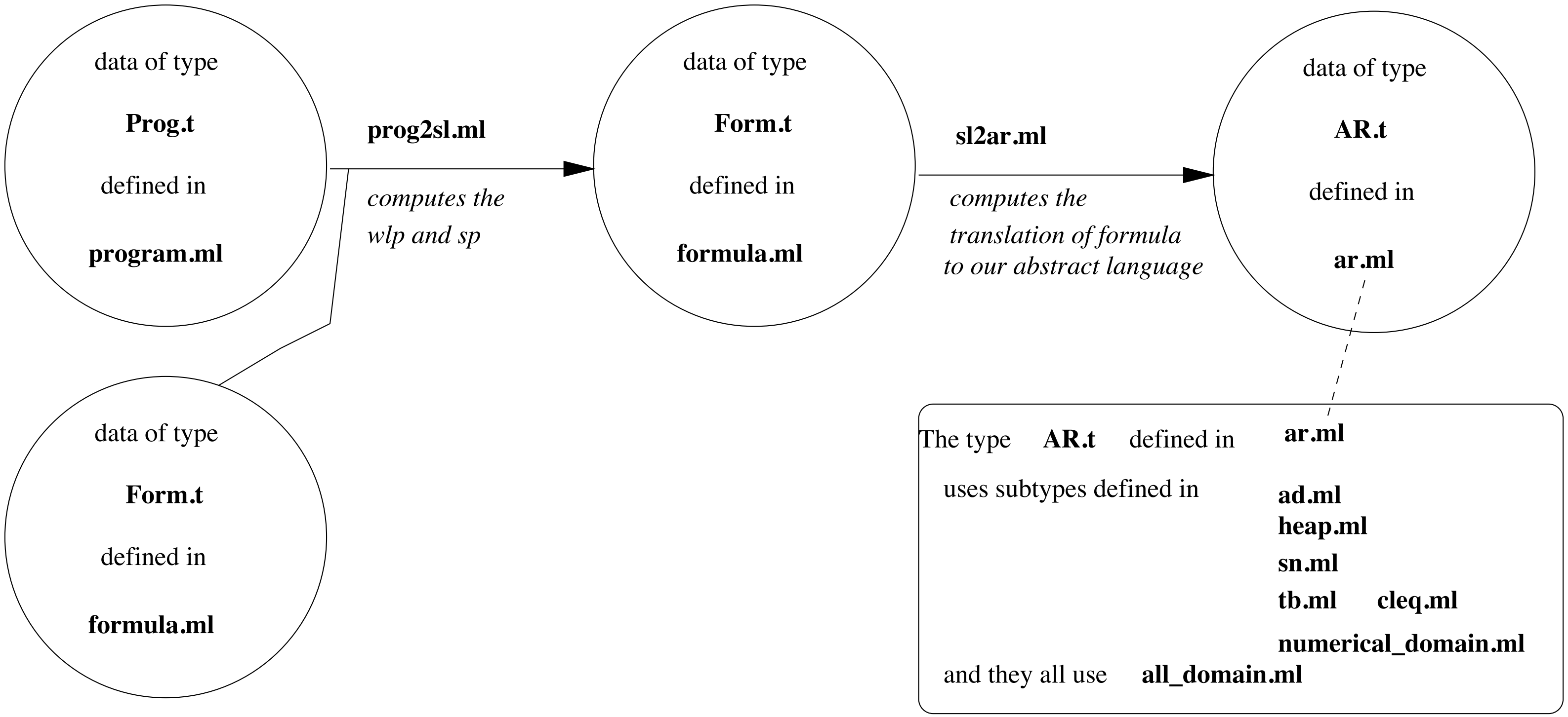
)
used to tune the precision of the union
- **merging** (replace $[v_1 \rightarrow S_1 \mid v_2 \rightarrow S_2]$ by $[v_2 \rightarrow (S_1 \cup S_2)]$)
used with the **widening**
- **translations** from formulae to the abstract language

Comparisons

- the $\bullet \begin{smallmatrix} \overset{1}{\rightsquigarrow} \\ \underset{2}{\rightsquigarrow} \end{smallmatrix}$ 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

Prototype

We have build a prototype implementation:



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

Conclusions

- $\surd$ We added fixpoints to separation logic, which provides a way to express recursive formulae and while-loops pre- and post-conditions.
- $\surd$ We proved useful properties about the extended logic

- $\surd$ 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
- $\surd$ 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)
- $\surd$ We designed a novel tabular data structure which allows extra precision by using a graph of sets instead of sets of graphs
- $\surd$ 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