The Relationship Between Separation Logic and Implicit Dynamic Frames

Alex Summers (ETH Zurich)

based on joint work with Matthew Parkinson (MSR-Cambridge)

Introduction

- Alex: a postdoc in Peter Müller’s Programming Methodology group at ETH Zurich
- Area: modular verification of (usually) concurrent, (usually) object-oriented software
- The group’s interests include developing new formalisms which can be implemented in automatic verification tools
- This talk is concerned with specification logics for concurrent heap-based programs

Baking your own automatic verifier

Ingredients:

- 1 Assertion Logic
- 1 Language semantics (weakest preconditions)
- Annotated code:

Method:

- Work backwards from the post-condition
- Check entailment: (ask SMT solver)

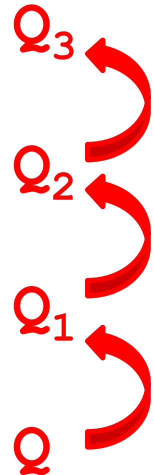
```
void m()
requires P
ensures Q
{

   this.x := 2;

   call n();

   this.x += 1;

}
```



$P \Rightarrow Q_3$?

Main problems

- Framing
  - how do we preserve heap information across method calls?
- Concurrency
  - how do we reason about heap values if other threads could interfere?
- Encoding to prover
  - how do we check entailments with a first-order SMT solver?

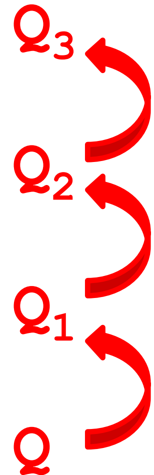
```
void m()
requires P
ensures Q
{

   this.x := 2;

   call n();

   this.x += 1;

}
```



$P \Rightarrow Q_3$?

Permissions to the Rescue (mostly)

- Idea: use specifications to explicitly allow/forbid certain heap accesses by program
- Assign a *permission* to each heap location, and only allow a thread to access with permission
- Similarly, heap values can only be *preserved* if permission is held on to (framing is easier)
- Distribute permissions between threads to avoid interference (concurrency is easier)
- We need a logic (or two) with these features...

Overview

Separation Logic (SL) ⊂ Total Permissions Logic (TPL) ⊃ Implicit Dynamic Frames (IDF)

Kripke semantics over partial heaps

… ? …

?

Weakest pre-condition definitions

Chalice : verification condition generation

Overview

Separation Logic (SL) ⊂ Total Permissions Logic (TPL) ⊃ Implicit Dynamic Frames (IDF)

Kripke semantics over partial heaps

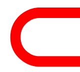
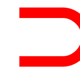
Kripke semantics over total heaps

?

Weakest pre-condition definitions

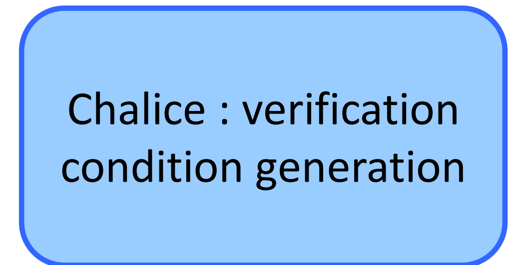
Chalice : verification condition generation

Overview

Separation Logic (SL)  Total Permissions Logic (TPL)  Implicit Dynamic Frames (IDF)

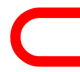
Kripke semantics over partial heaps Kripke semantics over total heaps Kripke semantics over total heaps

Weakest pre-condition definitions

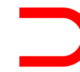


Overview

Separation Logic (SL)



Total Permissions Logic (TPL)



Implicit Dynamic Frames (IDF)

Kripke semantics over **partial** heaps

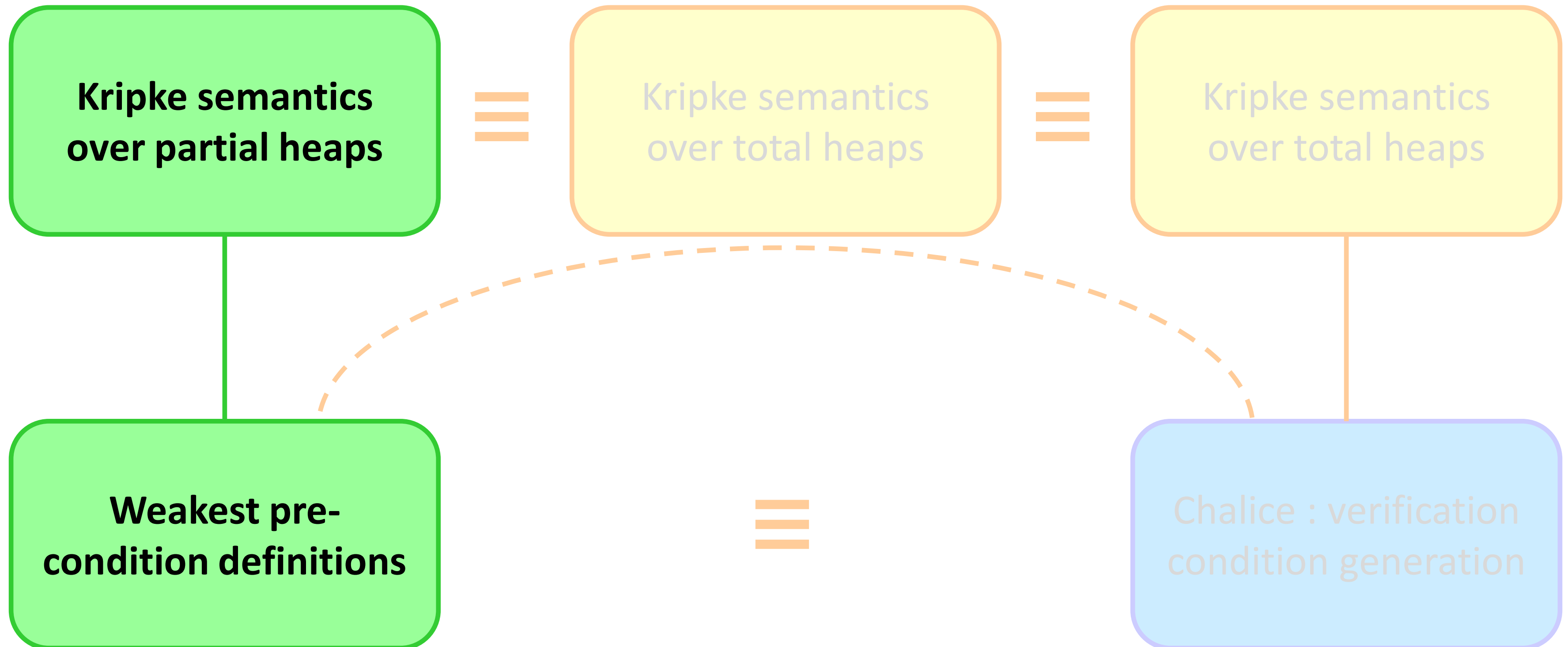
Kripke semantics over **total** heaps

Kripke semantics over total heaps

Weakest pre-condition definitions

Chalice : verification condition generation

Overview



What is separation logic?

- A specification logic with explicit connectives for accessing heap locations and dividing the heap
- Assertion semantics is based on *partial heaps*
- $A * B$: Splitting of heap into disjoint parts

$$\frac{\{\, P \,\}\ C\ \{\, Q \,\}}{\{\, R * P \,\}\ C\ \{\, Q * R \,\}}$$

: Frame rule (if vars in R unmodified)

- $x.f \mapsto v$ (“points-to predicate”)
  - Permission to access location x.f
  - Specifies value v currently stored at the location
- $A \mathrel{-\!\!\star} B$: Hypothetical addition of disjoint part

Intuitionistic separation logic

- For garbage collected languages, we want to be able to "forget" parts of our heap
  - e.g., intentionally leave certain heap locations out of a method post-condition.
- This can be reflected in the logic by ensuring that truth is closed under heap extension
  - i.e., $h \models A \;\Rightarrow\; h \uplus h' \models A$
  - this way, we can choose to check a weaker assertion than actually holds in our current heap
- This is easy to do for most of the connectives...

Intuitionistic separation logic

- For implication, it is a little tricky:
  - $h \vDash A \Rightarrow B$ iff ($h \vDash A \Rightarrow h \vDash B$) doesn't give a semantics closed under heap extension
  - For example, take the assertion ($x.f \mapsto 2 \Rightarrow$ false). This would be true in the empty heap (no access to x.f). But it is false in an extension of the empty heap, in which x.f = 2.
  - Instead, one builds in checking all extensions of the state:
- $h \vDash A \Rightarrow B$ iff $\forall h'(h \uplus h' \vDash A \Rightarrow h \uplus h' \vDash B)$
  - i.e., an implication A⇒B holds iff, in all extensions of the current state, if A is true then B is true.
  - More on this later...

Weakest preconditions in SL

- First-order logic weakest pre-condition world:

  wp(assume A, B) = $A \Rightarrow B$

  wp(assert A, B) = $A \wedge B$

- Other verification features can be "compiled" to assume/assert statements (e.g., method calls)
- In SL, there are two analogous commands

  wp(assume* A, B) = $A -\!\!\star B$ *(add A)*

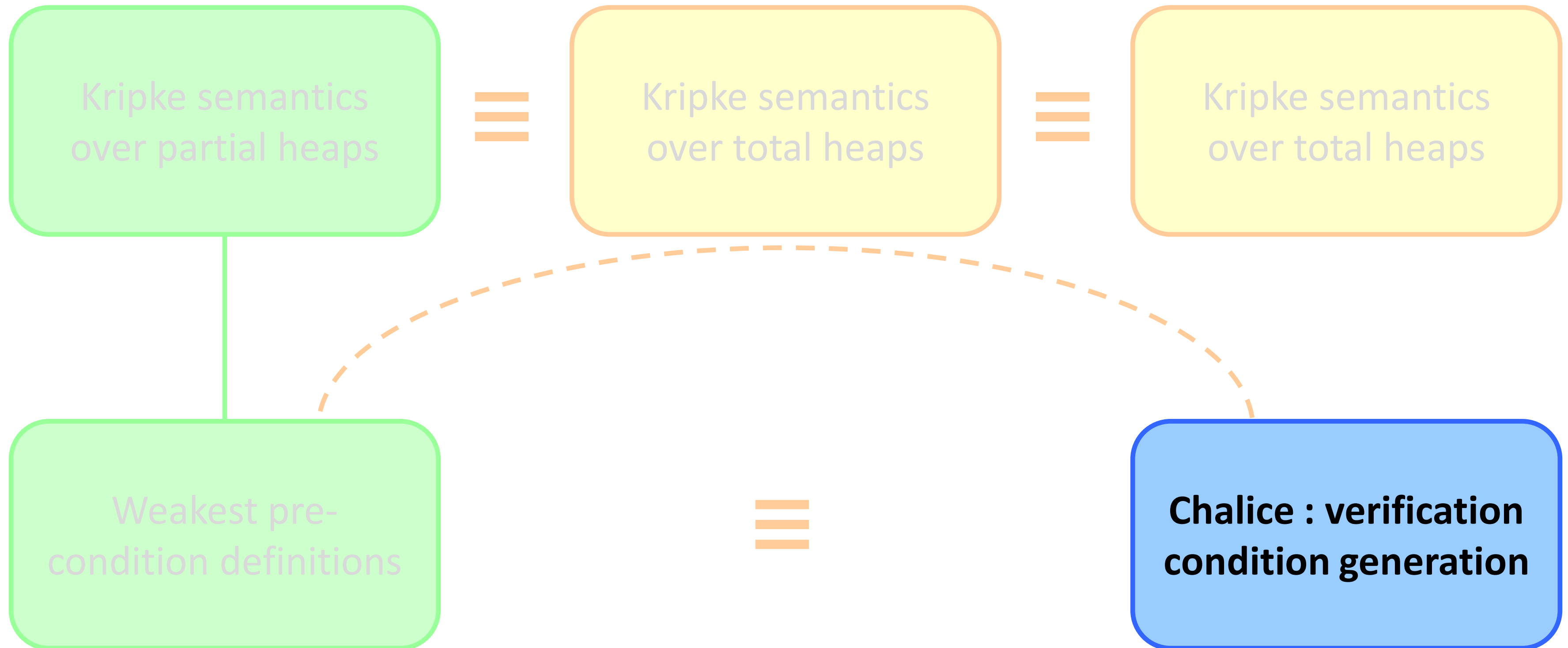
  wp(assert* A, B) = A * B *(remove A)*

Overview

Separation Logic (SL)

Total Permissions Logic (TPL)

Implicit Dynamic Frames (IDF)



Implicit Dynamic Frames

- Extend first-order assertions to additionally include “accessibility predicates”:

  acc(x.f) is an assertion; we have permission to x.f
- Assertions can also include heap-dependent expressions: e.g., x.f > 3
- Existing tool support is based on *total heaps*
  - every thread sees a value for every heap location
  - *but* these values are only guaranteed meaningful if the thread also holds permission to the location

Implicit Dynamic Frames

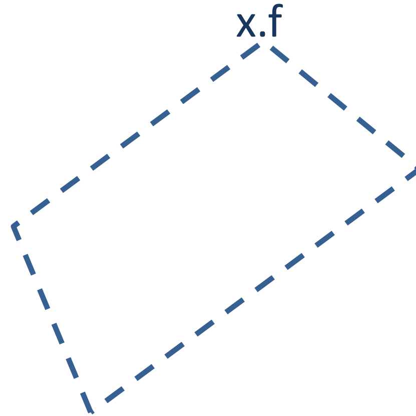
- For example

  acc(x.f) * x.f == 4 * acc(x.g)

Implicit Dynamic Frames

- For example

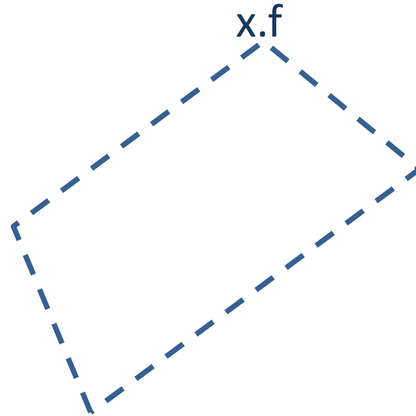
  acc(x.f) * x.f == 4 * acc(x.g)



Implicit Dynamic Frames

- For example

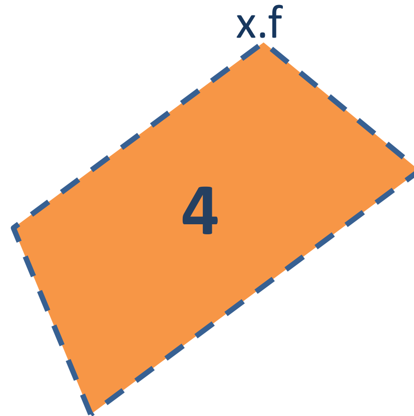
  acc(x.f) * **x.f == 4** * acc(x.g)



Implicit Dynamic Frames

- For example

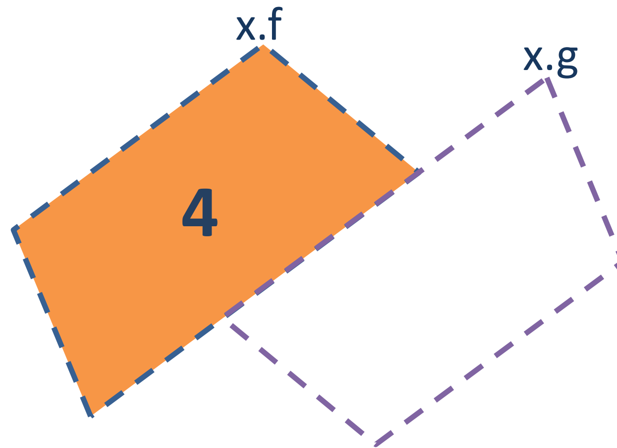
  acc(x.f) * **x.f == 4** * acc(x.g)



Implicit Dynamic Frames

- For example

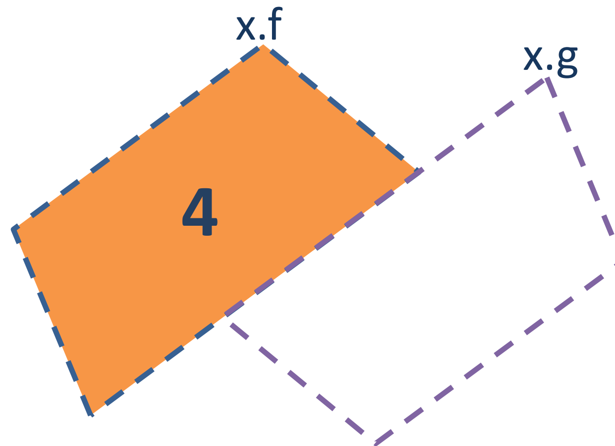
  acc(x.f) * x.f == 4 * **acc(x.g)**



Implicit Dynamic Frames

- For example

  acc(x.f) * **x.f == 4** * acc(x.g)

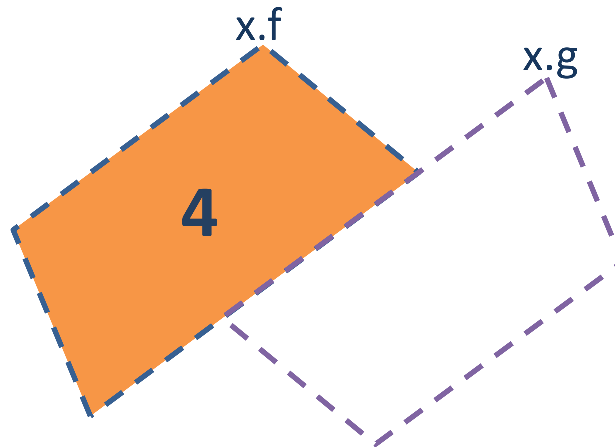


- Expressions include heap dereferences

Implicit Dynamic Frames

- For example

  acc(x.f) * x.f == 4 * acc(x.g)

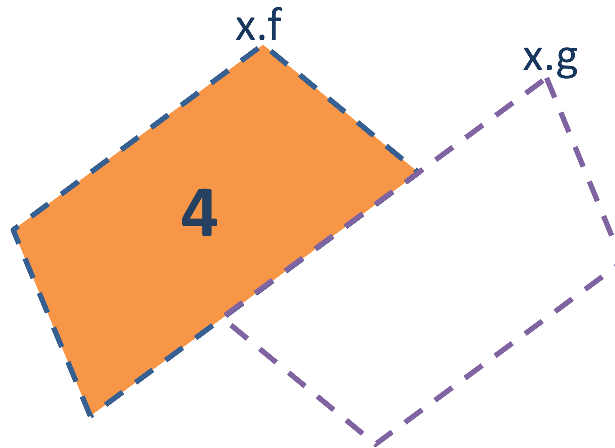


- Expressions include heap dereferences
- Permissions need not match “read footprint”

Implicit Dynamic Frames

- For example

  acc(x.f) * x.f == 4 * acc(x.g) *** y.f == 3**

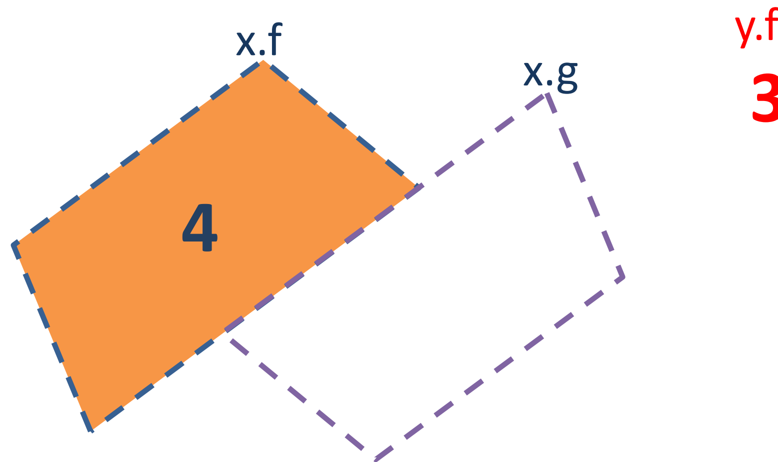


- Expressions include heap dereferences
- Permissions need not match “read footprint”

Implicit Dynamic Frames

- For example

  acc(x.f) * x.f == 4 * acc(x.g) *** y.f == 3**



- Expressions include heap dereferences
- Permissions might not match “read footprint”

Inhale and Exhale

- "inhale p" and "exhale p" are used in Chalice to encode transfers between threads/calls
- "inhale p" means:
  - *assume* heap properties in p
  - gain permissions in p
  - havoc newly-readable locations
- "exhale p" means:
  - *assert* heap properties in p
  - check and give up permissions

```
void m()
requires p
ensures q
{


}
```

Inhale and Exhale

- “inhale p” and “exhale p” are used in Chalice to encode transfers between threads/calls
- “inhale p” means:
  - *assume* heap properties in p
  - gain permissions in p
  - havoc newly-readable locations
- “exhale p” means:
  - *assert* heap properties in p
  - check and give up permissions

```
void m()
requires p
ensures q
{

   ...

   call m()

   ...

}
```

Inhale and Exhale

- "inhale p" and "exhale p" are used in Chalice to encode transfers between threads/calls
- "inhale p" means:
  - *assume* heap properties in p
  - gain permissions in p
  - havoc newly-readable locations
- "exhale p" means:
  - *assert* heap properties in p
  - check and give up permissions

```
void m()
requires p
ensures q
{
  // inhale p
  ...

  call m()


  ...

}
```

Inhale and Exhale

- "inhale p" and "exhale p" are used in Chalice to encode transfers between threads/calls
- "inhale p" means:
  - *assume* heap properties in p
  - gain permissions in p
  - havoc newly-readable locations
- "exhale p" means:
  - *assert* heap properties in p
  - check and give up permissions

```
void m()
requires p
ensures q
{
  // inhale p
  ...
  // exhale p
  call m()

  ...

}
```

Inhale and Exhale

- "inhale p" and "exhale p" are used in Chalice to encode transfers between threads/calls
- "inhale p" means:
  - *assume* heap properties in p
  - gain permissions in p
  - havoc newly-readable locations
- "exhale p" means:
  - *assert* heap properties in p
  - check and give up permissions

```
void m()
requires p
ensures q
{
  // inhale p
  ...
  // exhale p
  call m()
  // inhale q
  ...

}
```

Inhale and Exhale

- "inhale p" and "exhale p" are used in Chalice to encode transfers between threads/calls
- "inhale p" means:
  - *assume* heap properties in p
  - gain permissions in p
  - havoc newly-readable locations
- "exhale p" means:
  - *assert* heap properties in p
  - check and give up permissions

```
void m()
requires p
ensures q
{
  // inhale p
  ...
  // exhale p
  call m()
  // inhale q
  ...
  // exhale q
}
```

Inhale and Exhale

- "inhale p" and "exhale p" are used in Chalice to encode transfers between threads/calls
- "inhale p" means:
  - *assume* heap properties in p
  - gain permissions in p
  - havoc newly-readable locations
- "exhale p" means:
  - *assert* heap properties in p
  - check and give up permissions

```
void m()
requires p
ensures q
{
  // inhale p
  ...
  // exhale p
  call m()
  // inhale q
  ...
  // exhale q
}
```

Self-framing

- Inhaled assertions model new information passed from another thread/method call etc.
- But, this information must be “framed” by suitable permissions, to be sound to assume
- Inhaled/exhaled assertions are required to be “self-framing”:
  - essentially, they include enough permissions to preserve the truth of their heap assertions
  - e.g., acc(x.f)*x.f==4 but not x.f==4 alone
  - x.f==4 is meaningful only along with permission

Overview

Separation Logic (SL)

Implicit Dynamic Frames (IDF)

Kripke semantics over partial heaps

?

Weakest pre-condition definitions

Chalice : verification condition generation

A common semantics?

Separation Logic

- Controls access to heap locations along with values
- Semantics defined in terms of partial heaps
- Key connectives defined by adding/removing heap fragments

Chalice

- Controls permissions and values separately
- Semantics defined via translation, and total heaps
- Encoding defined by modification of global maps for heap and permissions

How can we formally relate the two?

Total Heaps Permission Logic (TPL)

Basically, a union of the syntaxes of SL and IDF

Semantics defined over *total* heaps...

Expressions (can access the heap, as in IDF)

E ::= E.f | E + E | n | x | …

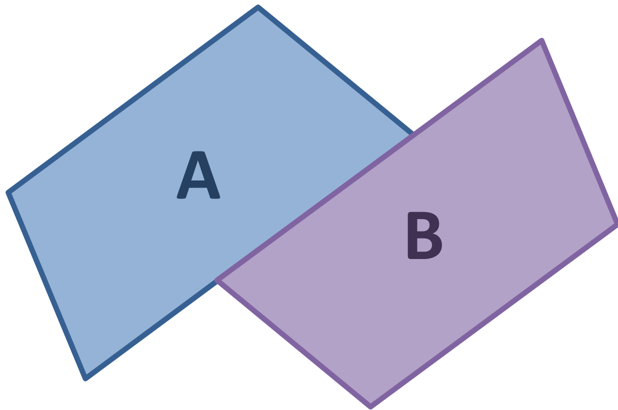
Assertions (both acc and “points to” predicates)

A,B ::= acc(x.f) | $x.f \mapsto v$ | E = E | A * B | …

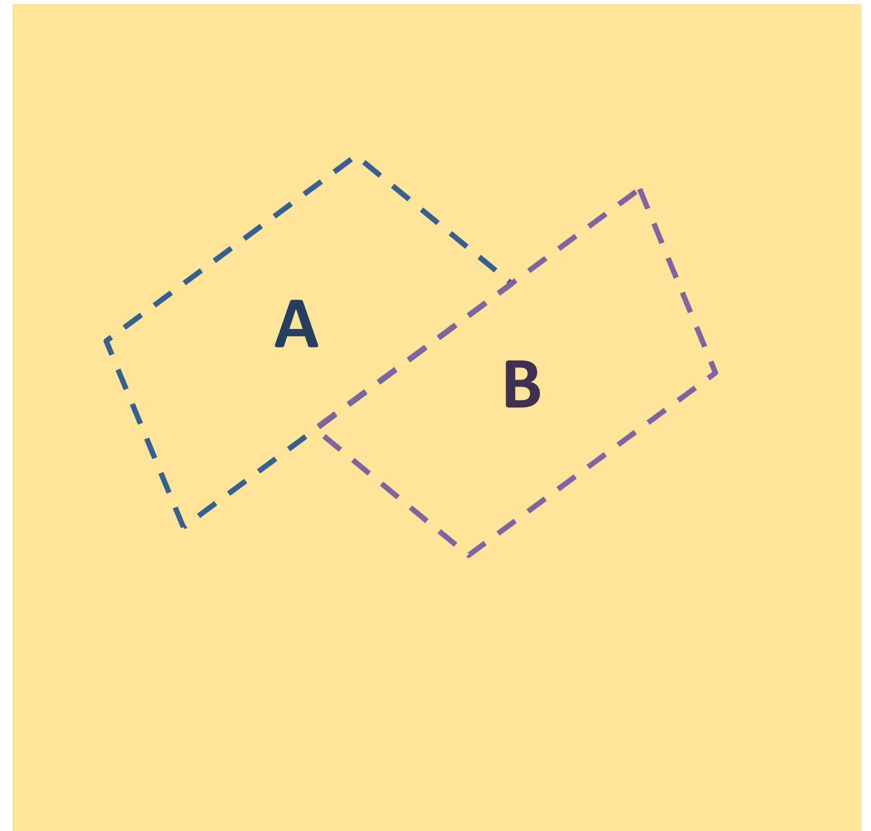
Intuition: $x.f \mapsto v \iff acc(x.f) * x.f = v$

A*B

Separation Logic partial heap

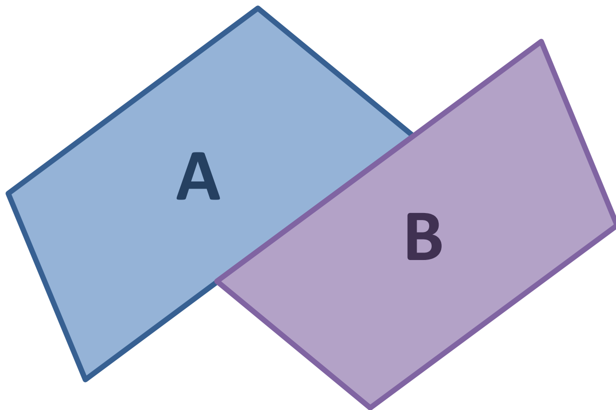


IDF total heap

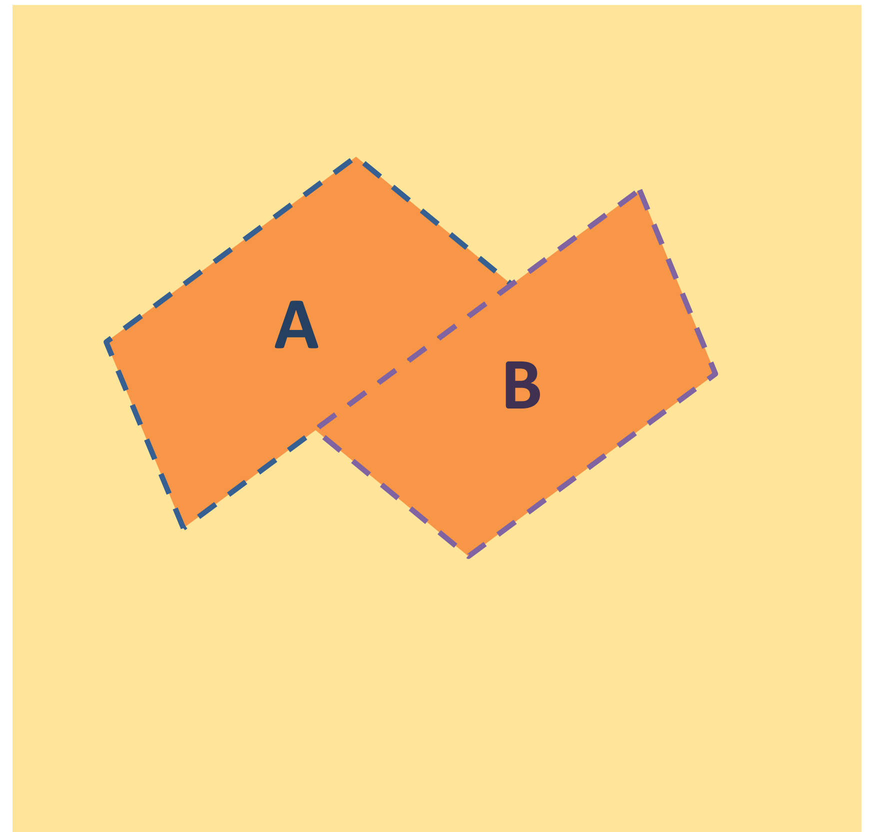


A*B

Separation Logic partial heap

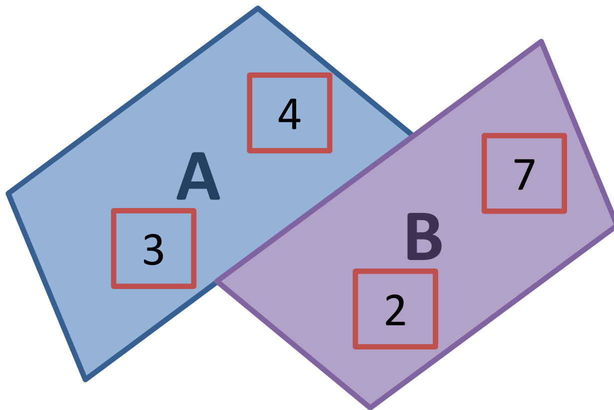


IDF total heap

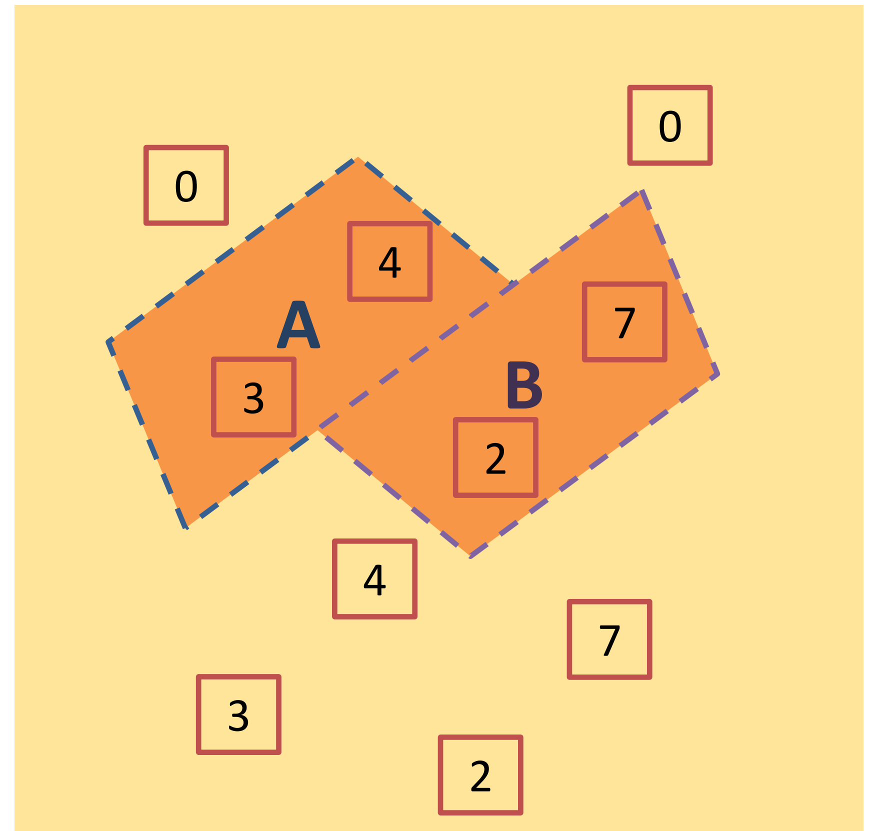


A*B

Separation Logic partial heap

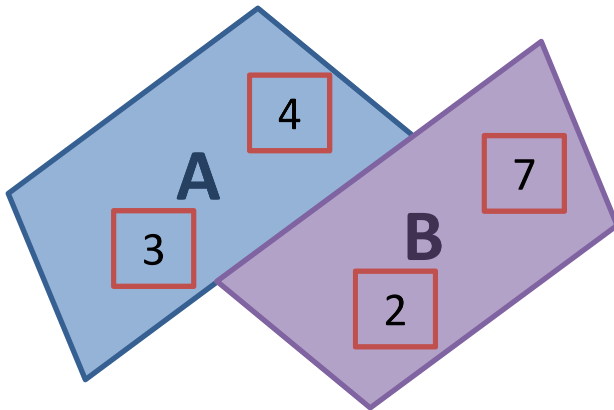


IDF total heap

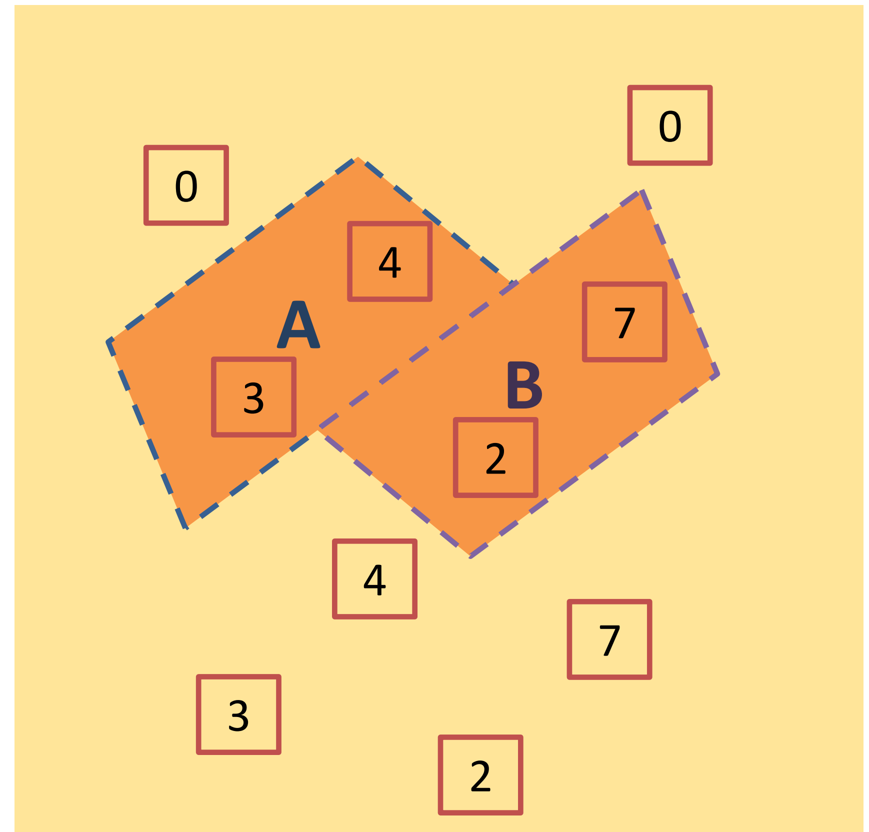


A*B

Separation Logic partial heap

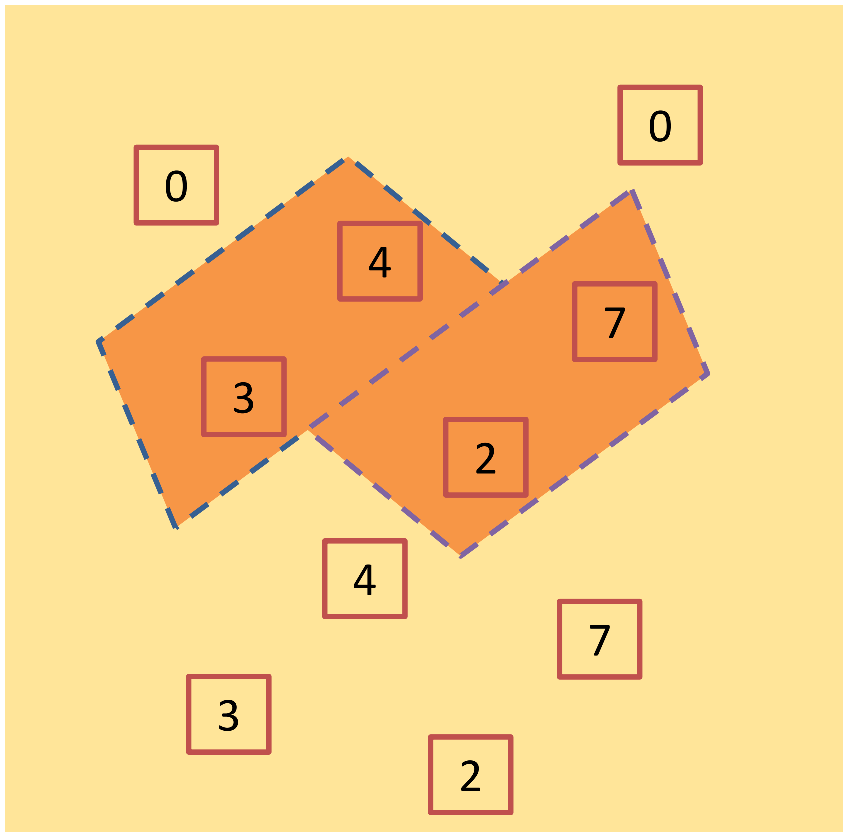


IDF total heap

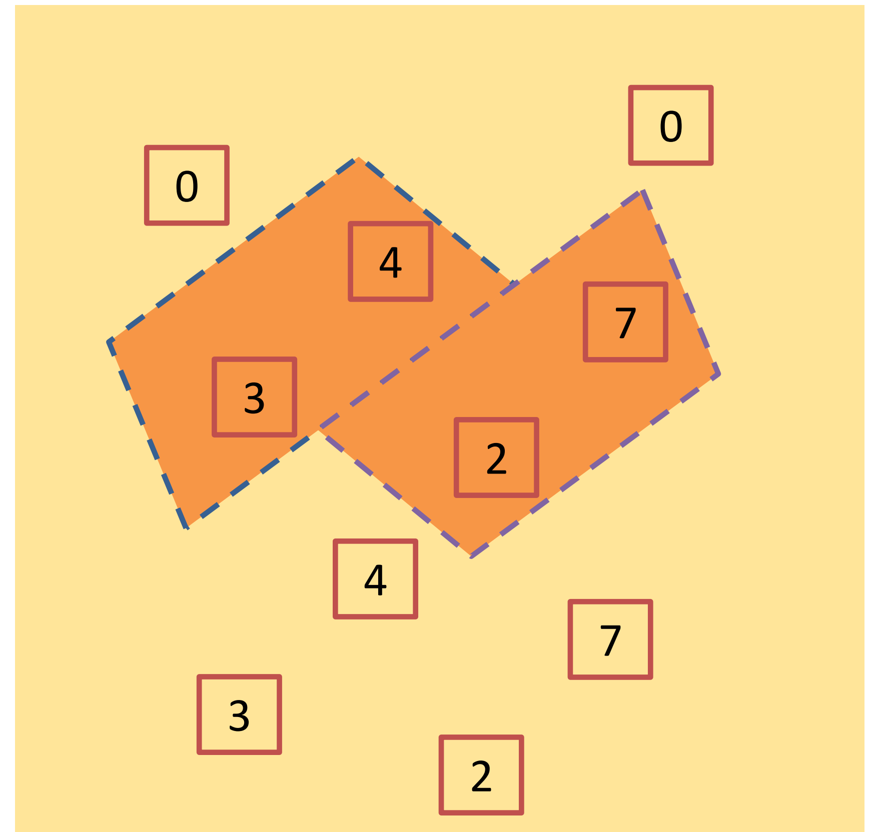


Heap agreement up to permissions

IDF total heap

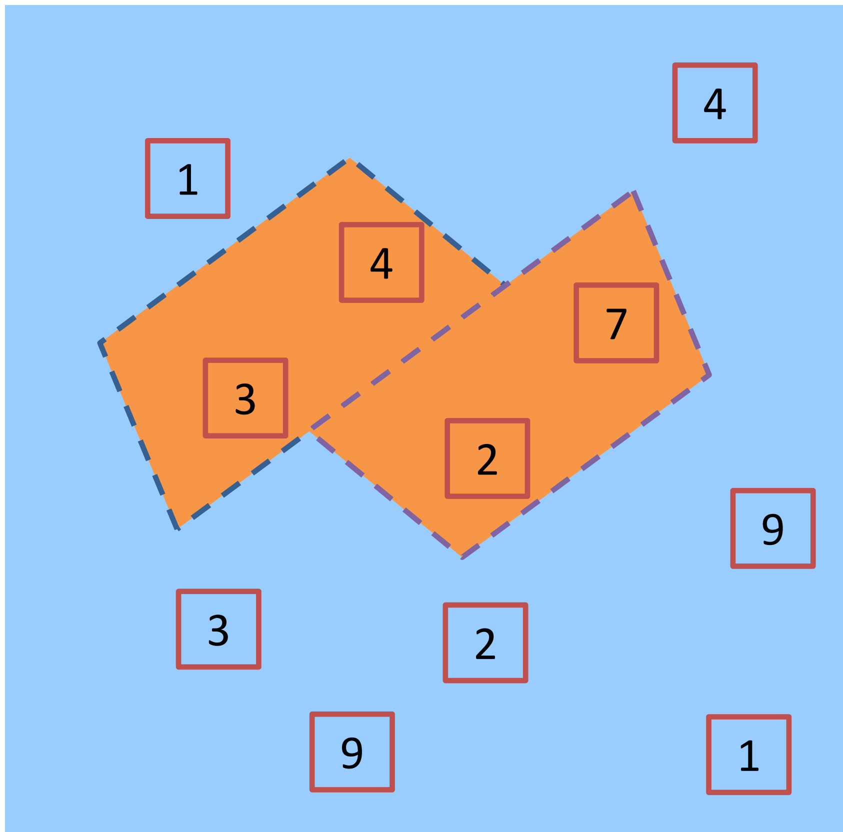


IDF total heap

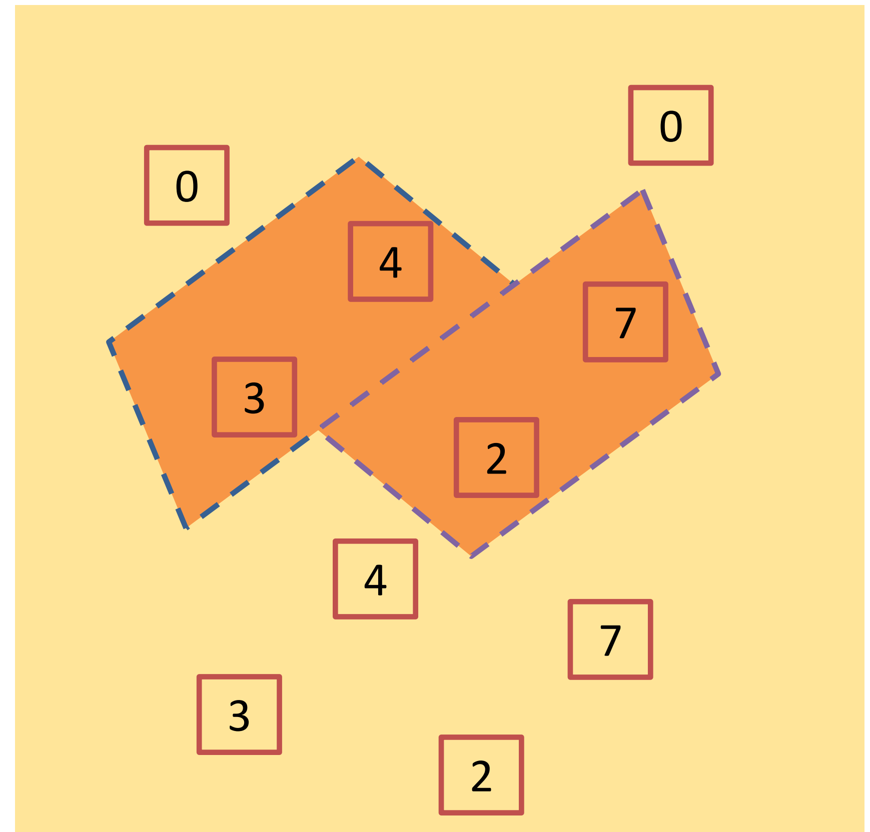


Heap agreement up to permissions

IDF total heap



IDF total heap

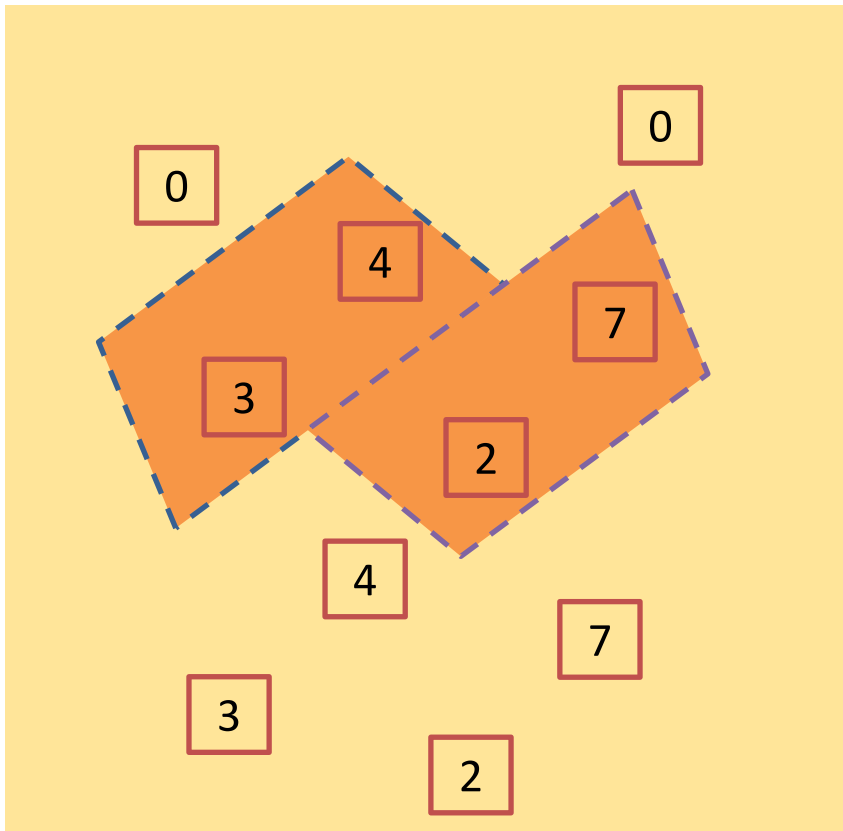


Self-framing revisited

An assertion is self-framing if:

Self-framing revisited

IDF total heap



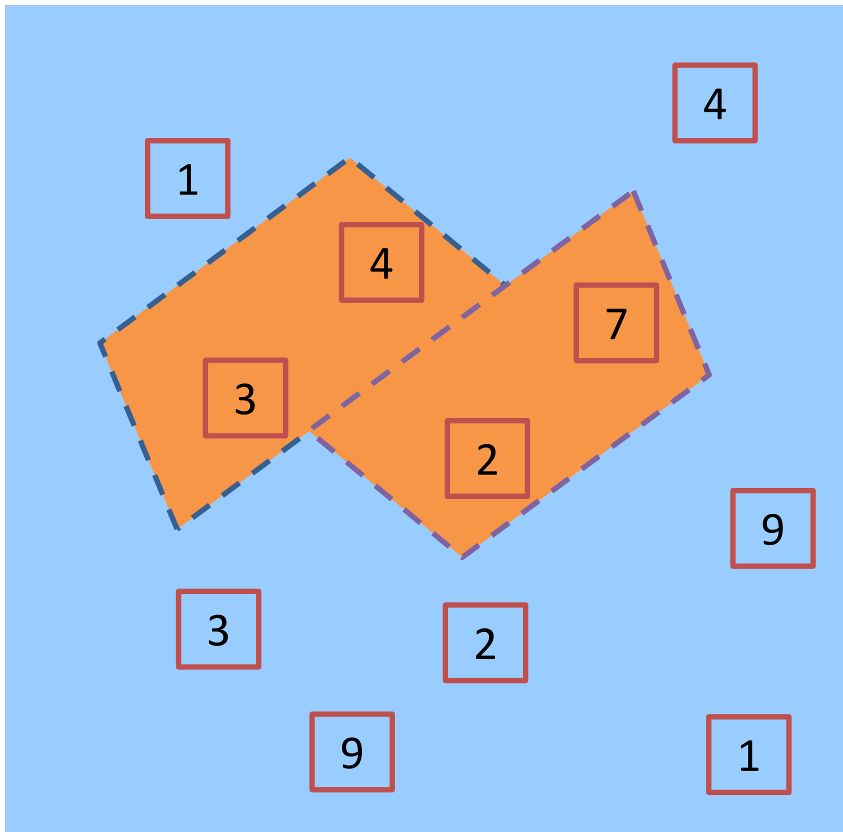
An assertion is self-framing if:

For any heap and permission mask satisfying it,

assertion remains true if we replace the heap with any that agrees on the permissions

Self-framing revisited

IDF total heap



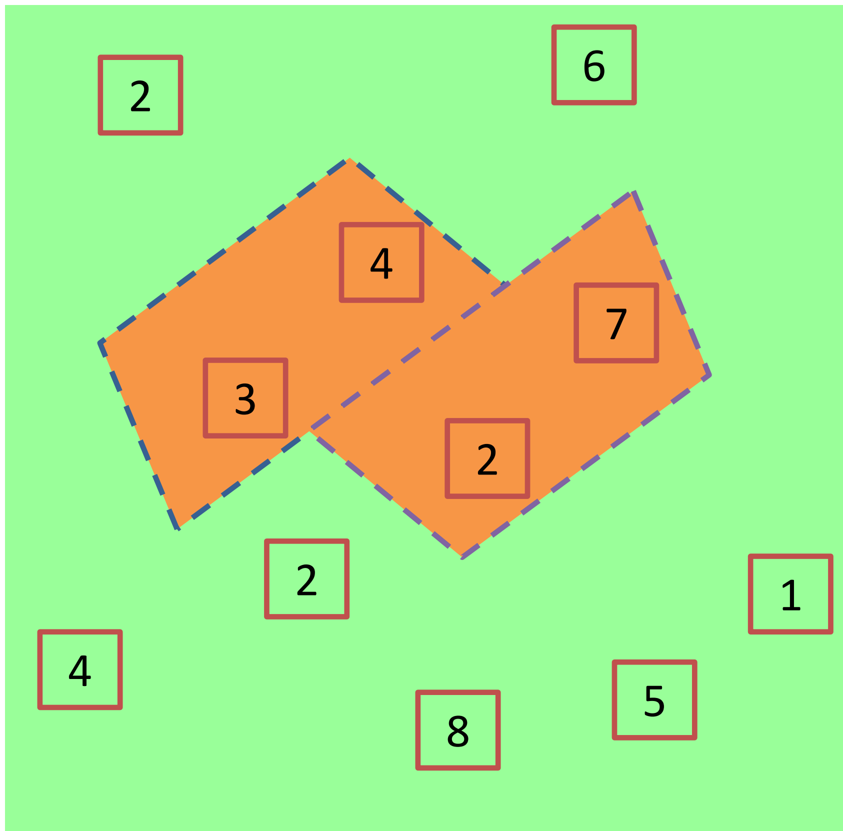
An assertion is self-framing if:

For any heap and permission mask satisfying it,

assertion remains true if we replace the heap with any that agrees on the permissions

Self-framing revisited

IDF total heap



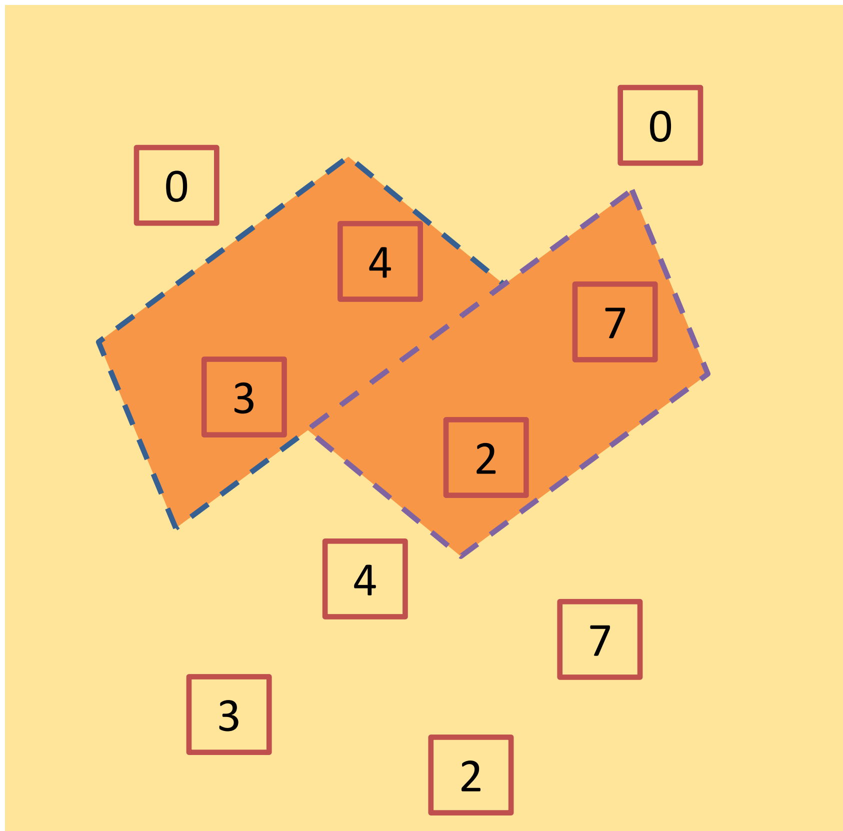
An assertion is self-framing if:

For any heap and permission mask satisfying it,

assertion remains true if we replace the heap with any that agrees on the permissions

Self-framing revisited

IDF total heap



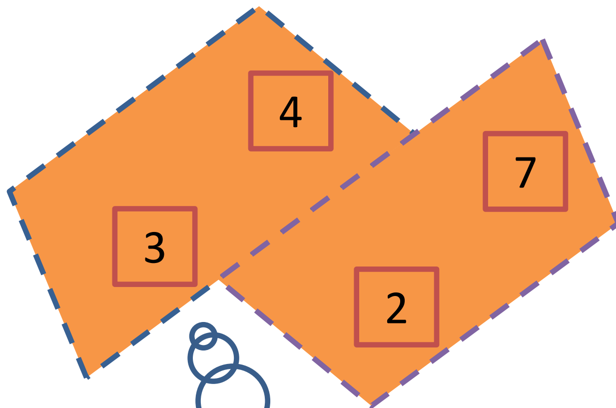
An assertion is self-framing if:

For any heap and permission mask satisfying it,

assertion remains true if we replace the heap with any that agrees on the permissions

Self-framing revisited

Separation Logic partial heap



In Separation Logic, there would be partial heap which canonically represents all the total ones in our semantics

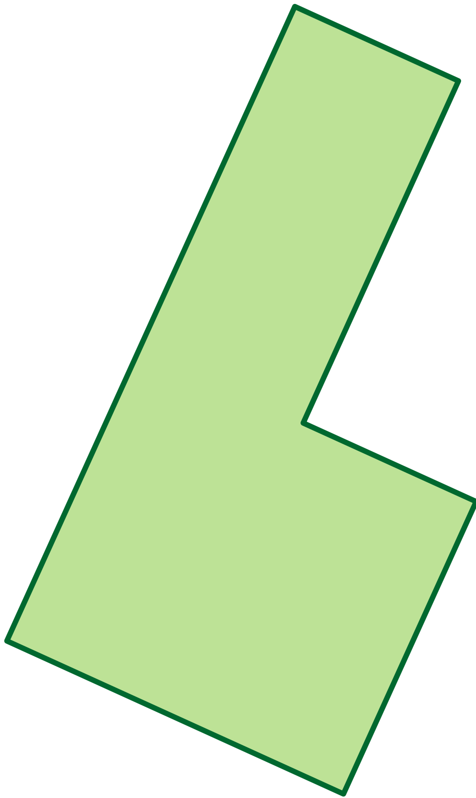
An assertion is self-framing if:

For any heap and permission mask satisfying it,

assertion remains true if we replace the heap with any that agrees on the permissions

$P \mathrel{-\!\!*} Q$

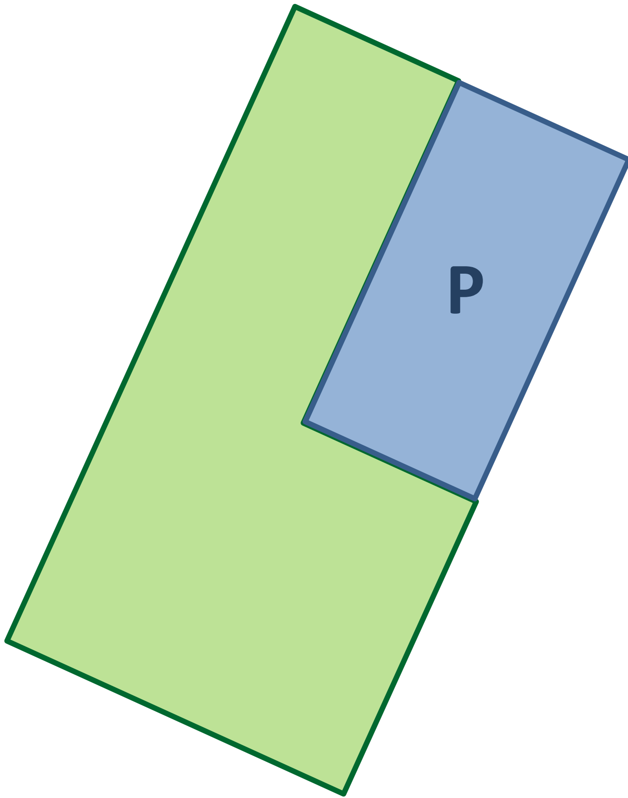
Separation Logic partial heap



IDF total heap

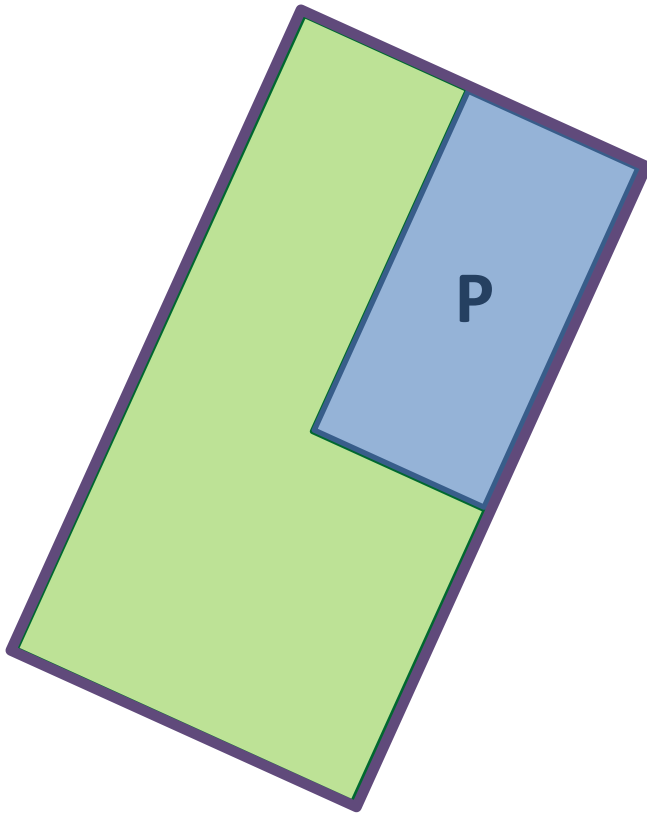
$P -\!\!* \; Q$

Separation Logic partial heap **IDF total heap**



$P -\!\!\ast\ Q$

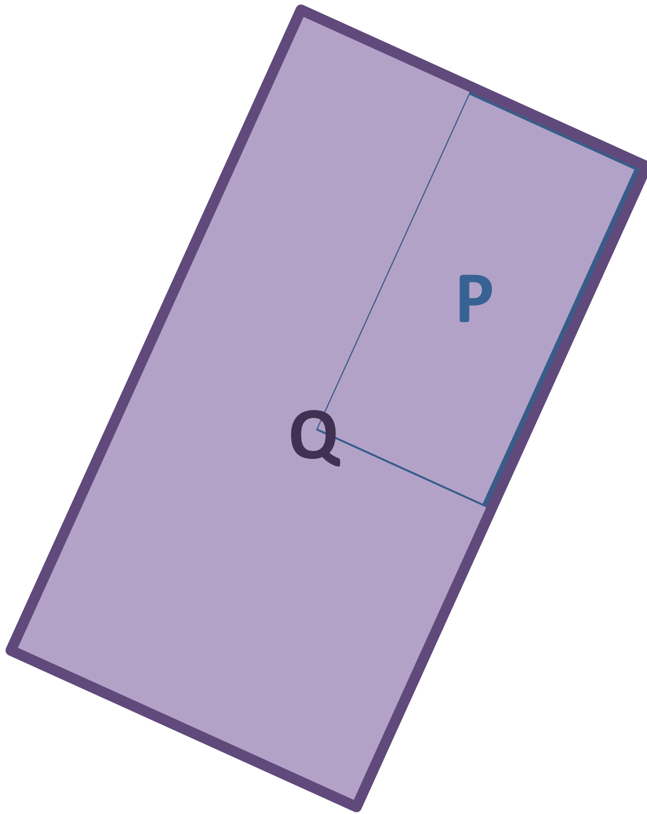
Separation Logic partial heap



IDF total heap

$P -\!\!\ast\ Q$

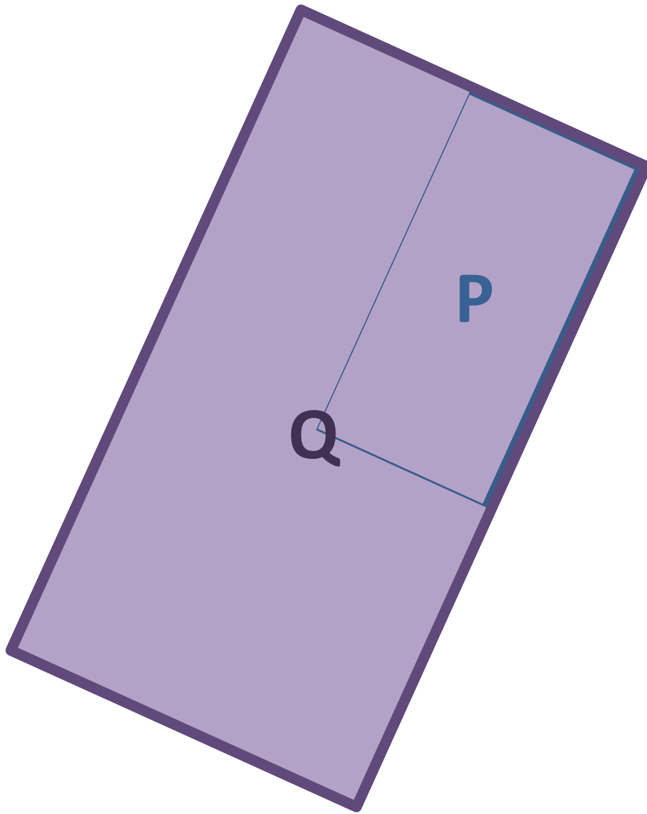
Separation Logic partial heap



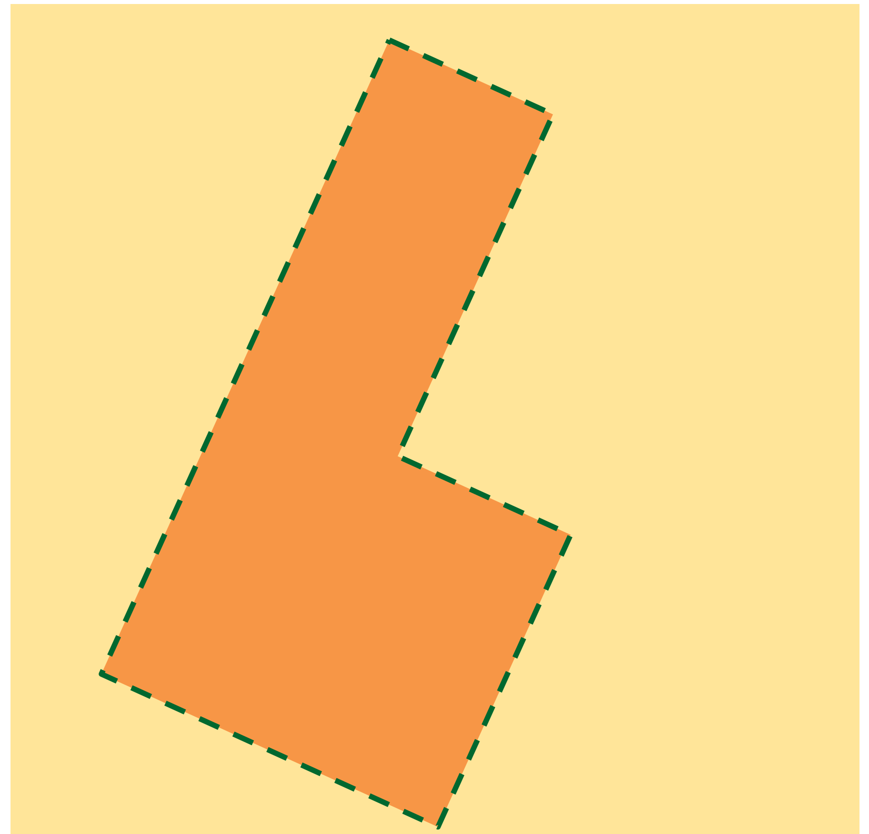
IDF total heap

$P \mathrel{-\!\!*} Q$

Separation Logic partial heap

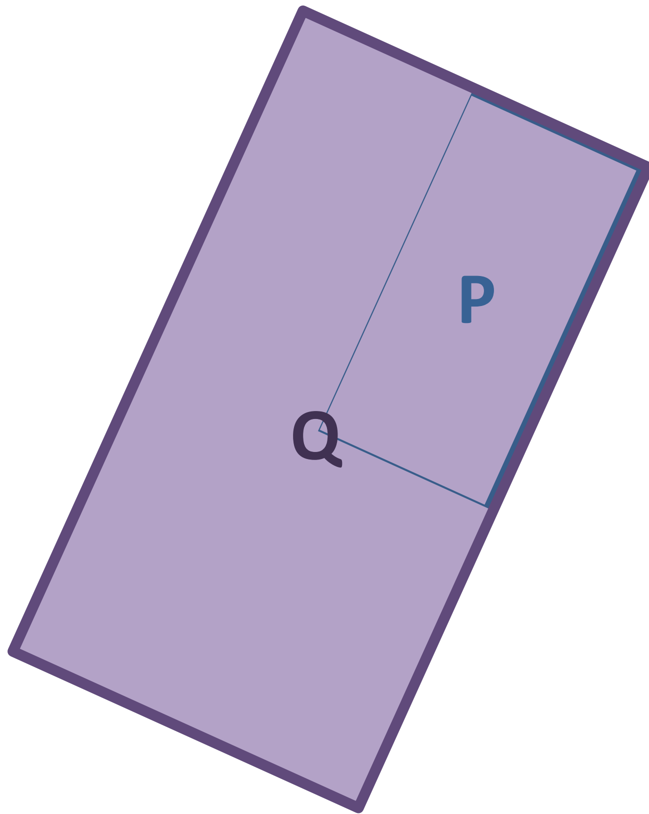


IDF total heap

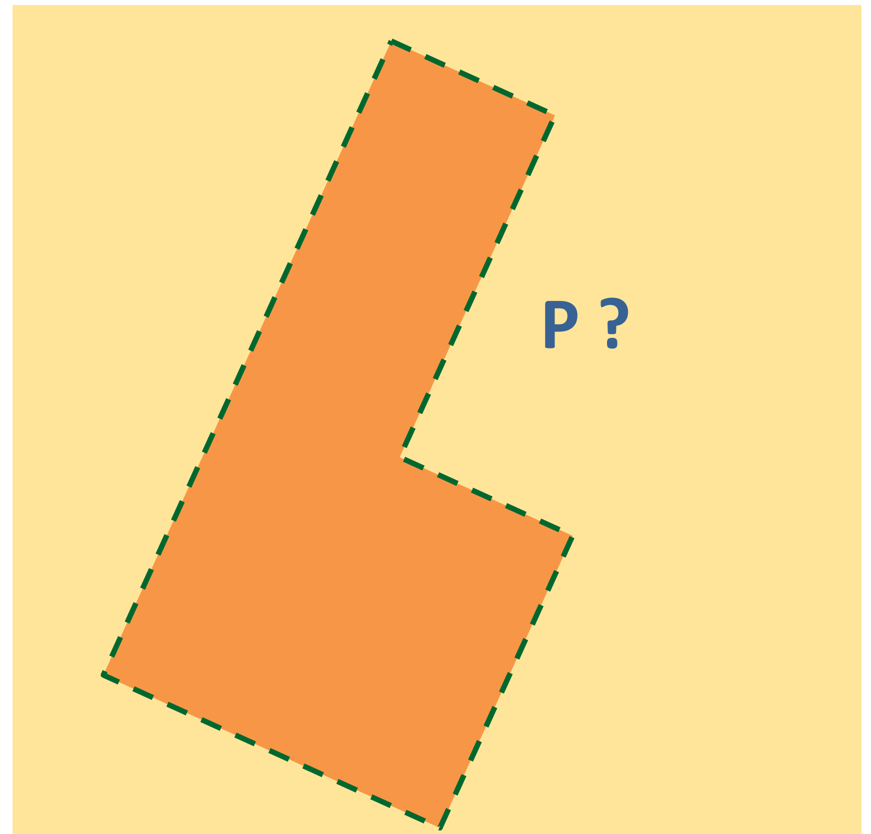


P −✶ Q

Separation Logic partial heap



IDF total heap



How to model heap extension?

Separation Logic

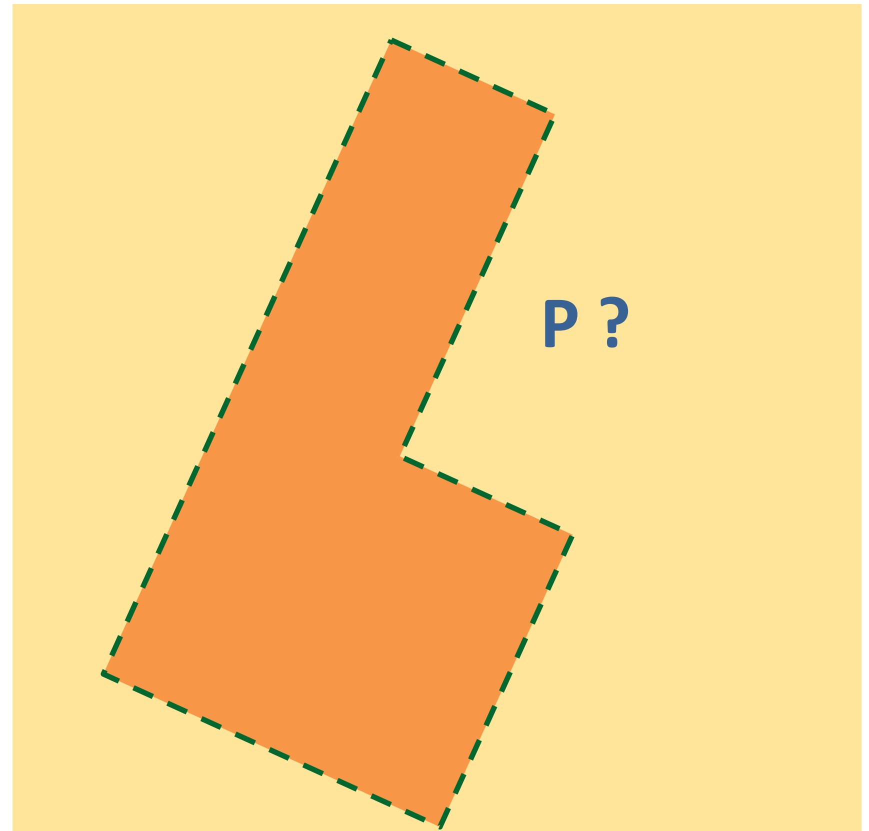
The semantics of implication and magic wand connectives are defined in terms of *partial heap extensions.*

$h \vDash A \Rightarrow B$ iff

$\forall h' (h \uplus h' \vDash A \;\Rightarrow\; h \uplus h' \vDash B)$

What should this mean for our total heaps model?

IDF total heap



How to model heap extension?

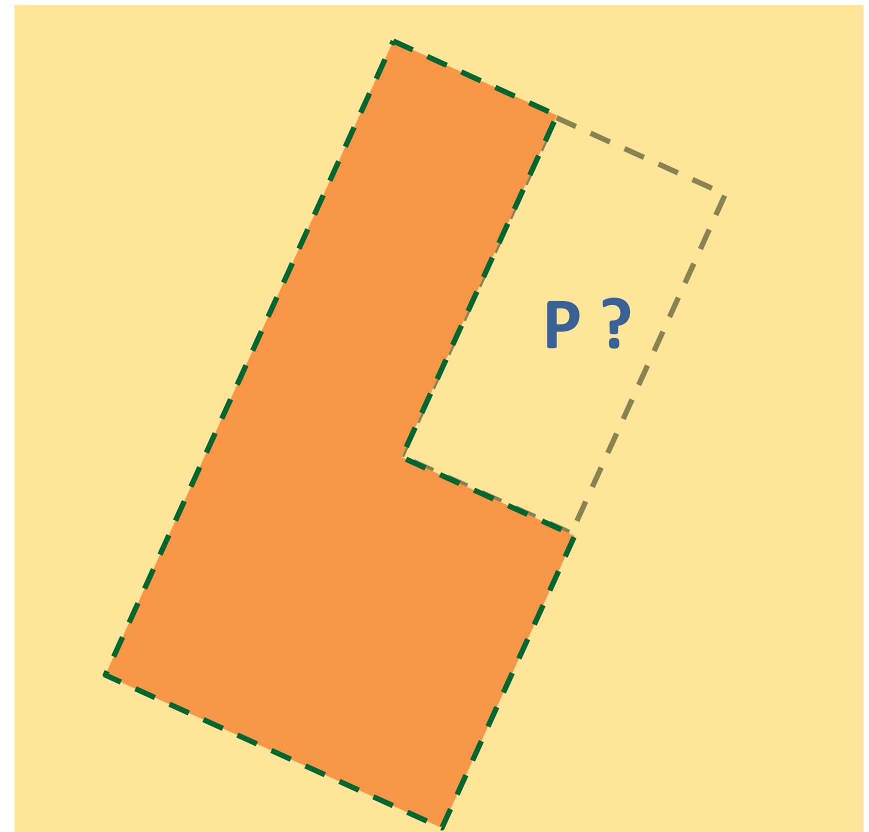
Idea 1:

Just add extra permissions to the original state

Problem:

We attach significance to the values that were previously stored in the heap, at the new locations

IDF total heap

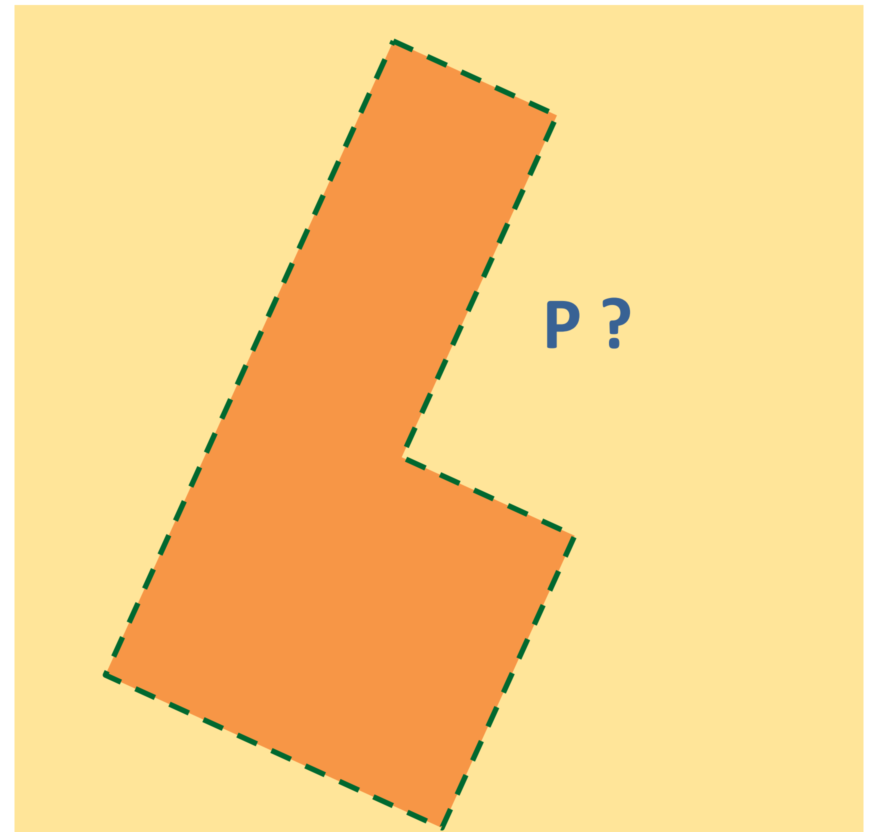


How to model heap extension?

Idea 2:

Assign new (arbitrary) values to unreadable heap locations, and *then* add new permissions

IDF total heap

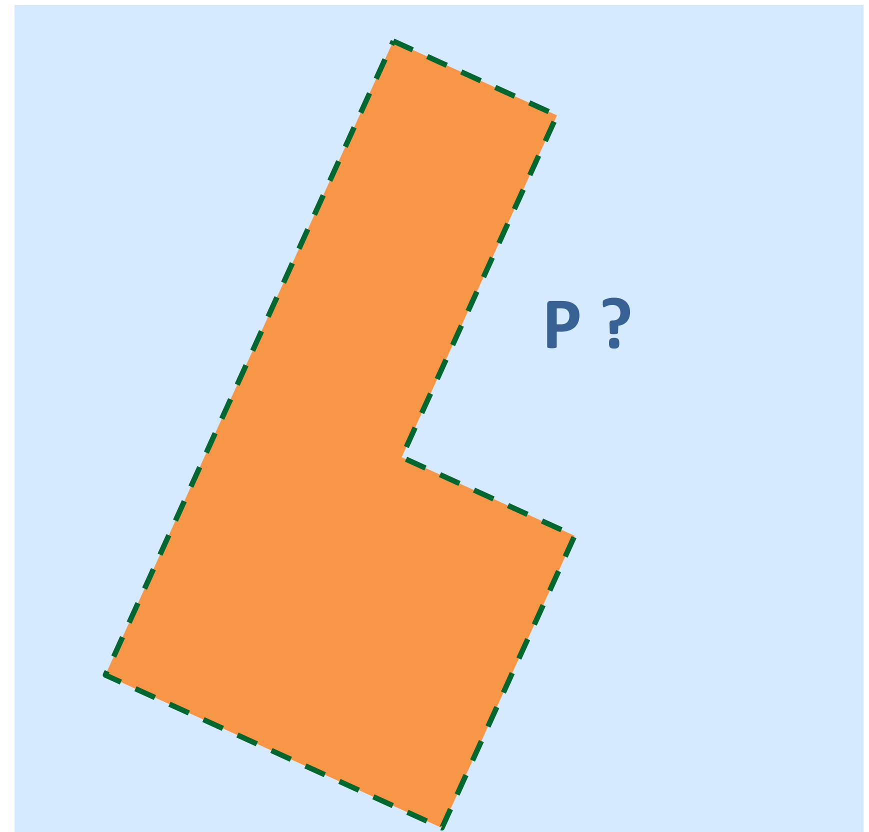


How to model heap extension?

Idea 2:

Assign new (arbitrary) values to unreadable heap locations, and *then* add new permissions

IDF total heap

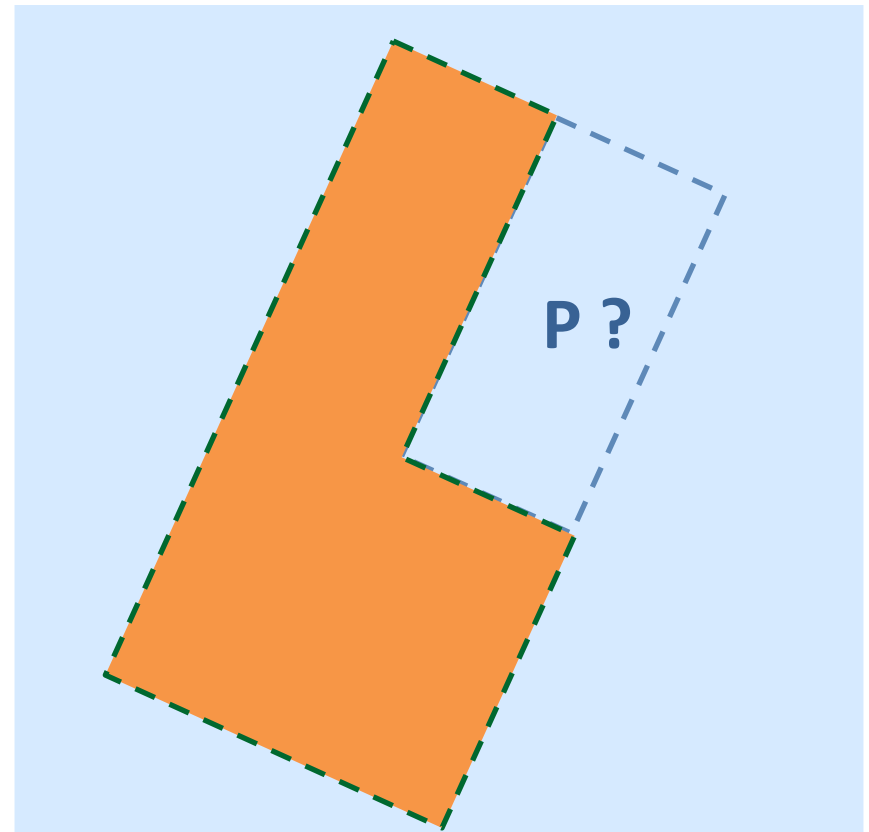


How to model heap extension?

Idea 2:

Assign new (arbitrary) values to unreadable heap locations, and *then* add new permissions

IDF total heap

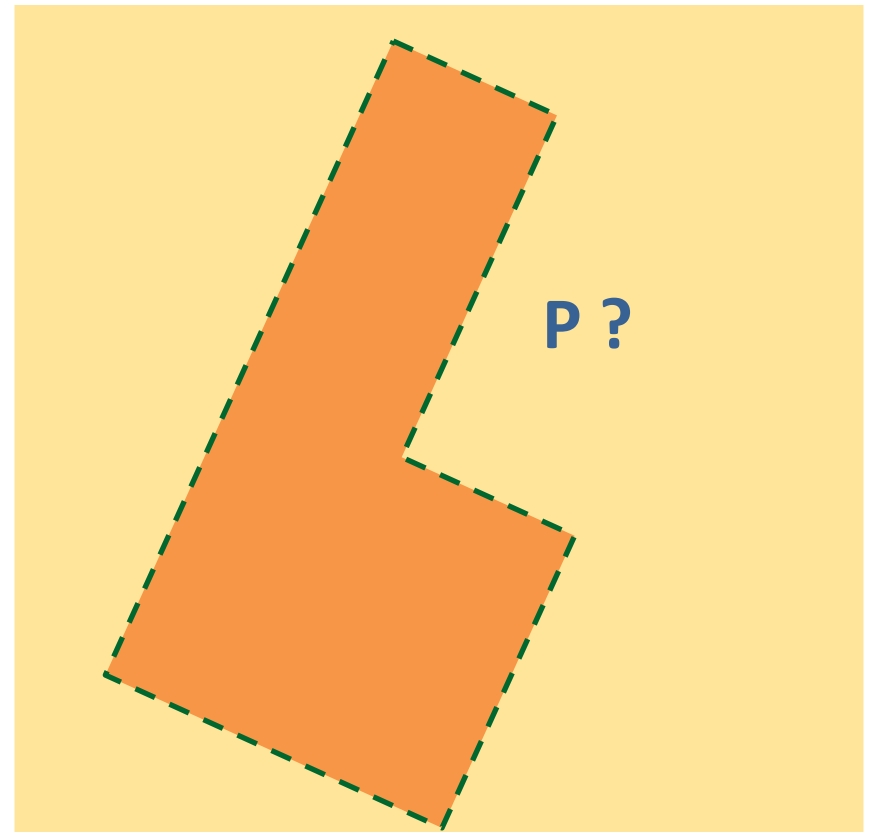


How to model heap extension?

Idea 2:

Assign new (arbitrary) values to unreadable heap locations, and *then* add new permissions

IDF total heap

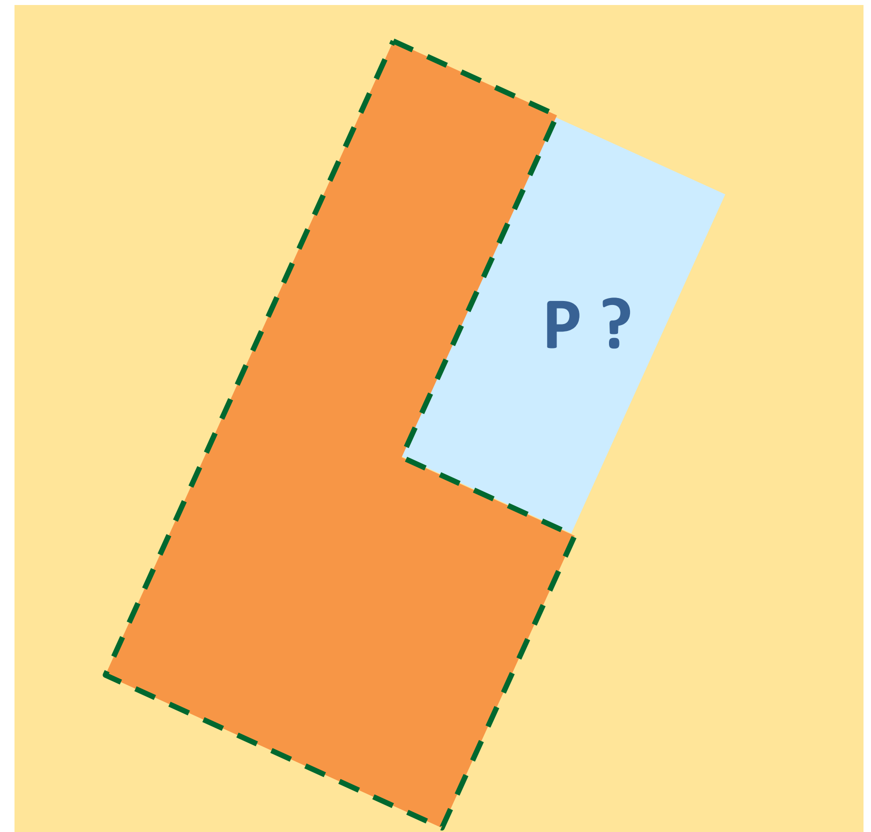


How to model heap extension?

Idea 2:

Assign new (arbitrary) values to unreadable heap locations, and *then* add new permissions

IDF total heap

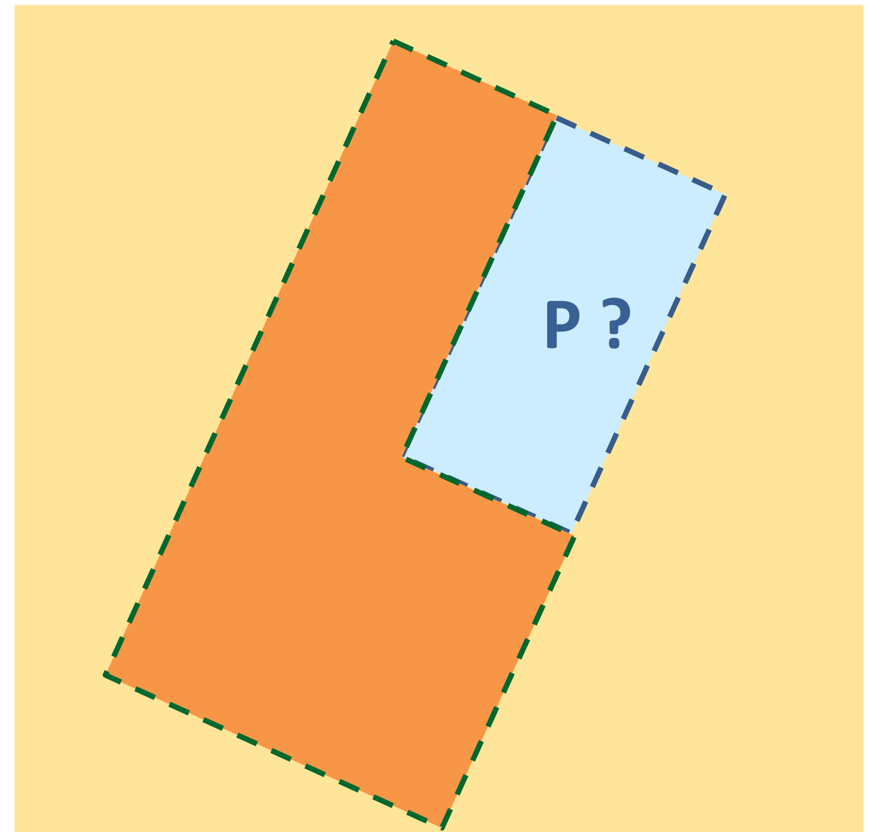


How to model heap extension?

Idea 2:

Assign new (arbitrary) values to unreadable heap locations, and *then* add new permissions

IDF total heap

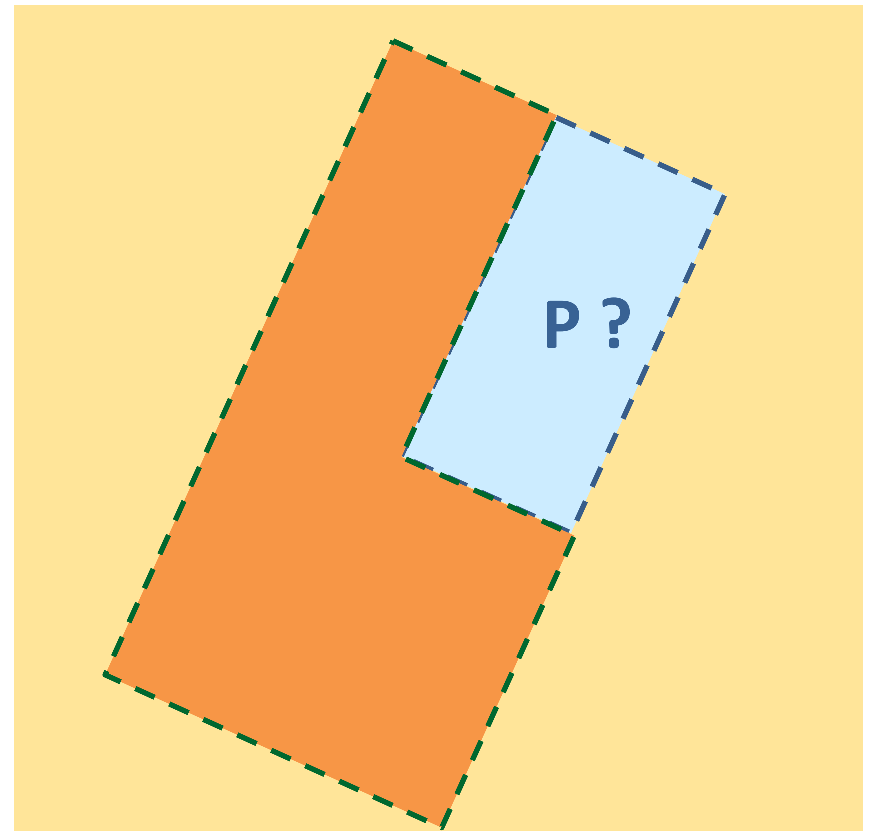


How to model heap extension?

Idea 2:

Assign new (arbitrary) values to unreadable heap locations, and *then* add new permissions

IDF total heap



Problem:

In IDF, P may involve heap-dependent expressions

How to model heap extension?

Idea 2:

Assign new (arbitrary) values to unreadable heap locations, and *then* add new permissions

Problem:

In IDF, P may involve heap-dependent expressions

IDF total heap

For example,

acc(x.f) * (x.f ≠ null ⇒ acc(x.f.g))

The intention is that the meaning of x.f is fixed by the permission “elsewhere”

But, when we judge the implication, if we consider assigning arbitrary values to x.f, then we lose the meaning

Minimal Permission Extensions

- Idea: only consider "extending" the state by the smallest amount possible

We modify the SL semantics:

$h \models A \Rightarrow B$ iff

$\forall h'(h \uplus h' \models A \;\; \Rightarrow \;\; h \uplus h' \models B)$

Minimal Permission Extensions

- Idea: only consider "extending" the state by the smallest amount possible

We modify the SL semantics to be:

$h \vDash A \Rightarrow B$ iff

$\forall h' (h \uplus h' \vDash A \wedge \forall h'' (h'' \subset h' \Rightarrow h \uplus h' \nvDash A)$

$\Rightarrow h \uplus h' \vDash B)$

We only consider adding the minimal extensions

For (intuitionistic) SL, this makes no difference

But this adapts well to our total heaps model...

Faithfully represents separation logic

Definition: the *restriction of a total heap H to permissions P*, is a partial heap $H\restriction P$ defined by:

$(H\restriction P)(x,f) = H(x,f)$ provided $P(x,f) > 0$,

$(H\restriction P)(x,f)$ is undefined otherwise.

Theorem

If A is a separation logic assertion, then:

$$H,P \vDash A \text{ in TPL} \iff H\restriction P \vDash A \text{ in SL}$$

Overview

Separation Logic (SL)

Total Permissions Logic (TPL)

Implicit Dynamic Frames (IDF)

Kripke semantics over partial heaps

Kripke semantics over total heaps

Kripke semantics over total heaps

Weakest pre-condition definitions

?

Chalice : verification condition generation

Chalice Weakest Pre-conditions

Encoded using two global map variables

- H : represents the values in the heap
- P : represents the permissions to access heap

For example,

wp_{ch}(inhale(acc(E.f)), A)

= wp_{ch}(P[E,f] += 1, A)

wp_{ch}(inhale(a*b), A)

= wp_{ch}(inhale(a);inhale(b), A)

Translating TPL to many sorted FOL

Expressions

$$[\![x]\!] = x \qquad [\![E.f]\!] = H[\, [\![E]\!], f \,]$$

Formulae

$$[\![acc(E.f)]\!] = P[[\![E]\!],f] == 1$$

$$[\![A*B]\!] = \exists P1,P2.\ [\![A]\!][P1/P] \wedge [\![B]\!][P2/P] \wedge P1 * P2 = P$$

P1 * P2 = P

$$\Leftrightarrow \forall i.\ P1[i] + P2[i] = P[i] \wedge P[i] \leq 1$$

Key points of the proof

All existentials of array type introduced by $[\![\quad]\!]$ are witnessed in Chalice VCs.

Self-framing is checked in Chalice by a syntactic (left-to-right) criterion, which is stronger than the semantic notion.

Note: asymmetry (left-to-right checking) of self-framing is essential for Chalice VC for inhale.

Faithfully representing Chalice

Theorem

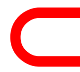
$$wp_{ch}\,(\text{exhale } p, \lfloor\lfloor A \rfloor\rfloor) \Leftrightarrow \lfloor\lfloor wp_{sl}(\text{assert* } p, A) \rfloor\rfloor$$

If p is (syntactically) self-framing, then

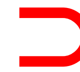
$$wp_{ch}(\text{inhale } p, \lfloor\lfloor A \rfloor\rfloor) \Leftrightarrow \lfloor\lfloor wp_{sl}(\text{assume* } p, A) \rfloor\rfloor$$

Summary

Separation Logic (SL)



Total Permissions Logic (TPL)



Implicit Dynamic Frames (IDF)

Kripke semantics over **partial** heaps

Kripke semantics over **total** heaps

Kripke semantics over total heaps

Weakest pre-condition definitions

Chalice : verification condition generation

Summary

- We defined the first direct semantics for IDF
- We defined a total heaps semantics for SL
- We have formally connected the two logics
  - We can encode SL assertions as IDF assertions
- We have defined a novel semantics for SL implication and magic wand connectives
- We have proven equivalence between weakest pre-conditions in SL and IDF

Advantages for Separation Logic

- We can use our work to provide a new way of verifying separation logic specifications
- Apply our encoding to convert specification to IDF, then feed it to e.g., Chalice
- Allows the verification of separation logic directly with an SMT solver
- Requires (ongoing) extension to handle abstract predicates
- New semantics with minimal extensions may be more easily implementable in automatic tools.

Advantages for Implicit Dynamic Frames

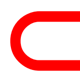
- The existence of a formal semantics helps with evaluating potential extensions to the logic
- Also facilitates soundness proofs for the methodology and (ultimately) the tools
- We've defined a compatible semantics for many previously-unsupported connectives
- Useful connectives such as the magic wand and logical disjunction could be added to tools

Future/Ongoing work

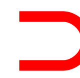
- Improving the encoding of abstract predicates (and heap functions) in Chalice
- Formalisation of the extended logic, and soundness proof based on this semantics
- Extending Chalice with more connectives – aided by our new formal semantics
- Mapping separation logic examples to implicit dynamic frames, for automatic verification

Thank you for listening..

Separation Logic (SL)



Total Permissions Logic (TPL)



Implicit Dynamic Frames (IDF)

Kripke semantics over partial heaps

Kripke semantics over total heaps

Kripke semantics over total heaps

Weakest pre-condition definitions

Chalice : verification condition generation