Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic

Clément Hurlin

INRIA Sophia Antipolis – Méditerranée, France
Universiteit Twente, The Netherlands

Soutenance de thèse

Thèse dirigée par Marieke Huisman

Thèse effectuée au sein des équipes Everest et FMT

14 septembre 2009

Un programme, qu'est ce que c'est ?

- Un programme est une suite d'instructions.

Exemple de programme calculant la valeur d'une fraction:

```
lire n;
lire d;
affiche n / d;
```

Un programme, qu'est ce que c'est ?

- Un programme est une suite d'instructions.

Exemple de programme calculant la valeur d'une fraction:

```
lire n;
lire d;
affiche n / d;
```

Le but de cette thèse:

- S'assurer que les programmes fonctionnent correctement.

Un programme, qu'est ce que c'est ?

- Un programme est une suite d'instructions.

Exemple de programme calculant la valeur d'une fraction:

```
lire n;
lire d;
affiche n / d;
```

Le but de cette thèse:

- S'assurer que les programmes fonctionnent correctement.

↳ Par exemple, le programme ci-dessus est-il correct ?

↳ Hum, pas vraiment, on peut effectuer une division par zéro…

↳ C.à.d. que ce programme peut *planter*.

Un programme, qu'est ce que c'est ?

- Un programme est une suite d'instructions.

Exemple de programme calculant la valeur d'une fraction:

```
lire n;
lire d;
si d ≠ 0 alors affiche (n / d);
sinon affiche ‘‘Erreur : d doit etre different de zero.’’;
```

Le but de cette thèse:

- S'assurer que les programmes fonctionnent correctement.

↳ Par exemple, le programme ci-dessus est-il correct ?

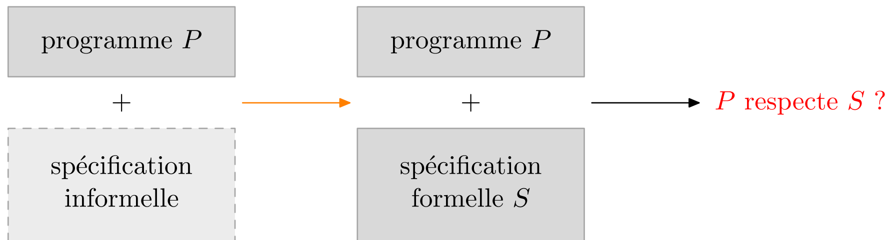
↳ Oui!

Vérifier des programmes

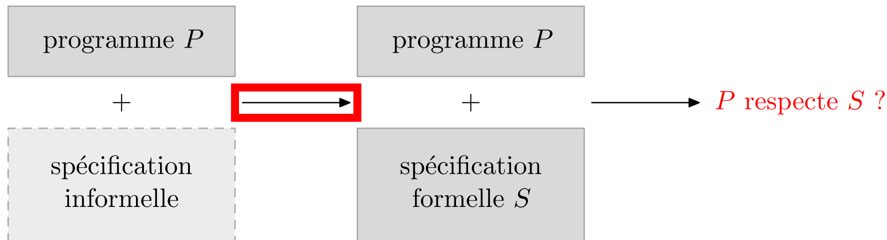
Pour s'assurer du bon fonctionnement des programmes, on les *vérifie*.

Vérifier un programme *P*, ça consiste à:

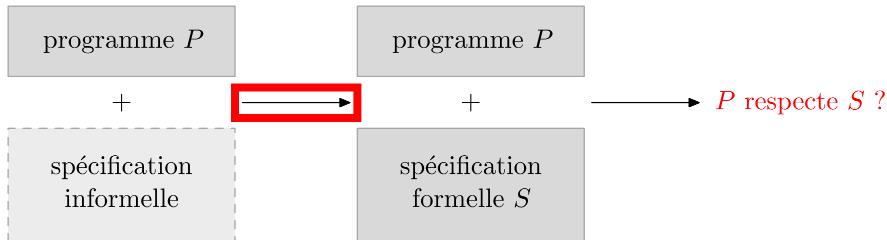
- Spécifier formellement *P*, c.à.d. exprimer ce que *P* est censé faire.
- Vérifier que *P* satisfait sa spécification.



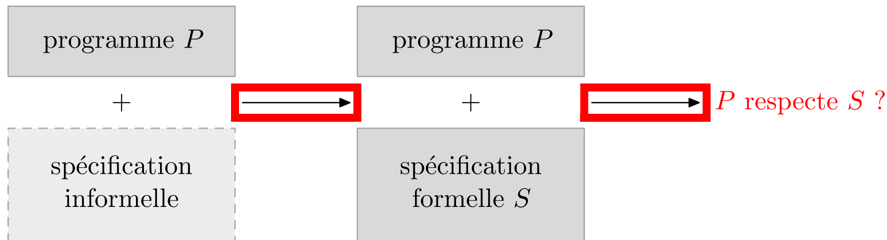
Sujets d'étude de cette thèse



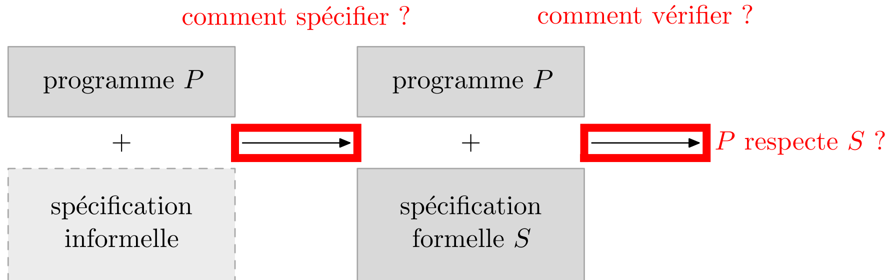
Sujets d'étude de cette thèse



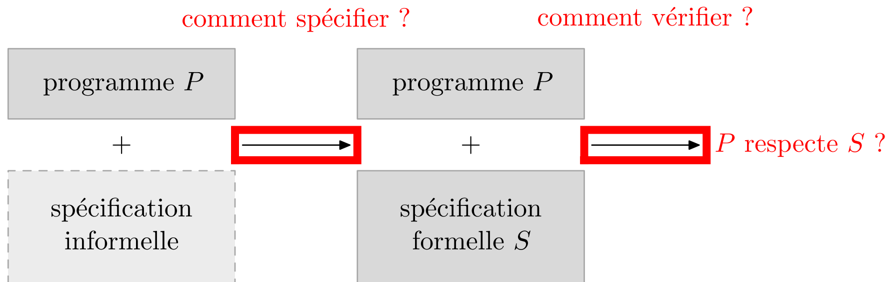
Sujets d'étude de cette thèse



Sujets d'étude de cette thèse



Sujets d'étude de cette thèse



Pour des programmes *objets* et *multi-processeurs*.

Objectives of the thesis

To adapt separation logic to multithreaded Java

↳ I.e. to support Java's primitives for multithreading:

(1) fork/join

(2) Reentrant locks

Objectives of the thesis

To adapt separation logic to multithreaded Java

↳ I.e. to support Java's primitives for multithreading:

(1) fork/join

(2) Reentrant locks

By using variants of separation logic [Reynolds'02]:

- Separation logic for while programs with a parallel operator $\parallel$ [O'Hearn'07]
- Separation logic for sequential Java programs [Parkinson'05]

Objectives of the thesis

To adapt separation logic to multithreaded Java

↳ I.e. to support Java's primitives for multithreading:

(1) fork/join

(2) Reentrant locks

By using variants of separation logic [Reynolds'02]:

- Separation logic for while programs with a parallel operator $\|$ [O'Hearn'07]
- Separation logic for sequential Java programs [Parkinson'05]

Side effects of the thesis:

- Three analyses based on separation logic

Our tool for reasoning: Separation Logic

Our assertion language is *permission accounting separation logic* [Reynolds'02,Bornat et al.'05].

- Formulas represent permissions to access the heap.
- Formula $x.f \stackrel{\pi}{\longmapsto} v$ has a dual meaning:
  - $x.f$ contains value v.
  - Permission π to access field $x.f$.
- Permissions π are fractions in $(0,1]$ [Boyland'03].
  - Permission 1 grants write and read access.
  - Any permission < 1 grants readonly access.

Our tool for reasoning: Separation Logic

Our assertion language is *permission accounting separation logic* [Reynolds'02,Bornat et al.'05].

- Formulas represent permissions to access the heap.
- Formula $x.f \stackrel{\pi}{\longmapsto} v$ has a dual meaning:
  - $x.f$ contains value v.
  - Permission π to access field $x.f$.
- Permissions π are fractions in $(0,1]$ [Boyland'03].
  - Permission 1 grants write and read access.
  - Any permission < 1 grants readonly access.

Abstract predicates represent complex formulas [Parkinson'05]:

- They are defined in classes.
- They have at least one parameter (the receiver)

Our tool for reasoning: Separation Logic

Our assertion language is *permission accounting separation logic* [Reynolds'02,Bornat et al.'05].

- Formulas represent permissions to access the heap.
- Formula $x.f \stackrel{\pi}{\longmapsto} v$ has a dual meaning:
  - $x.f$ contains value v.
  - Permission π to access field $x.f$.
- Permissions π are fractions in $(0,1]$ [Boyland'03].
  - Permission 1 grants write and read access.
  - Any permission < 1 grants readonly access.

Abstract predicates represent complex formulas [Parkinson'05]:

- They are defined in classes.
- They have at least one parameter (the receiver)

Compared to the literature:

- We mix object-orientation and permissions.
- Classes can be parameterized by specification values.

Objective 1: fork/join [AMAST'08]

fork and join are the two primitives used to create and wait threads
(in Java, C++, C, python, etc.):

- `t.fork()` starts a new thread `t`.
- `t.join()` waits until thread `t` terminates.

↳ fork and join are more general than $\|$.

Objective 1: fork/join [AMAST'08]

fork and join are the two primitives used to create and wait threads (in Java, C++, C, python, etc.):

- `t.fork()` starts a new thread `t`.
- `t.join()` waits until thread `t` terminates.

↳ fork and join are more general than $\|$.

In terms of resources (i.e. the heap), fork and join behave as follows:

- `t.fork()` consumes the resource needed by `t` to execute.

Objective 1: fork/join [AMAST'08]

fork and join are the two primitives used to create and wait threads (in Java, C++, C, python, etc.):

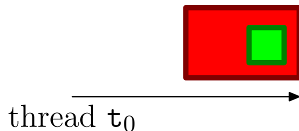
- `t.fork()` starts a new thread `t`.
- `t.join()` waits until thread `t` terminates.

↳ fork and join are more general than $\|$.

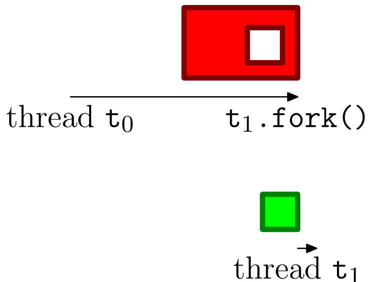
In terms of resources (i.e. the heap), fork and join behave as follows:

- `t.fork()` consumes the resource needed by `t` to execute.
- `t.join()` gets back [a part of] `t`'s resource when `t` terminates.

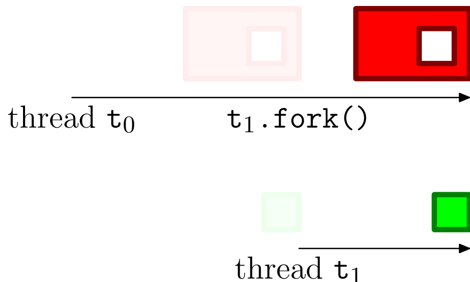
fork and join in terms of resources (1)



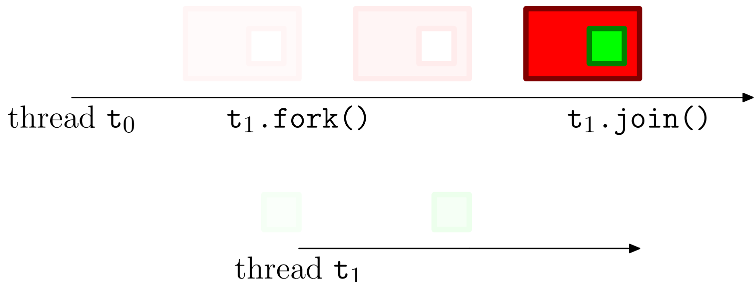
fork and join in terms of resources (1)



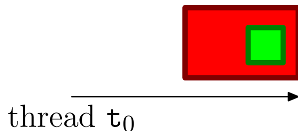
fork and join in terms of resources (1)

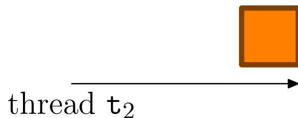


fork and join in terms of resources (1)

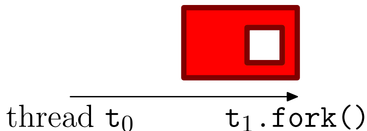


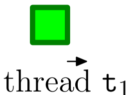
Fork and join in terms of resources (2)

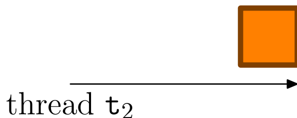




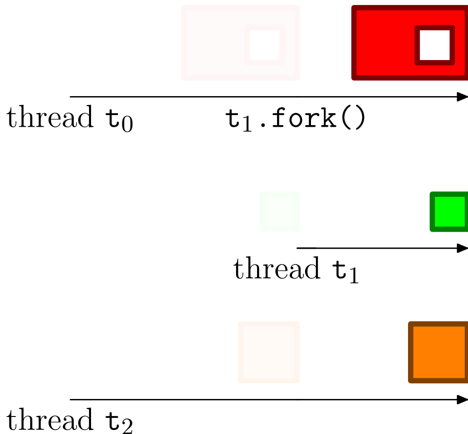
Fork and join in terms of resources (2)



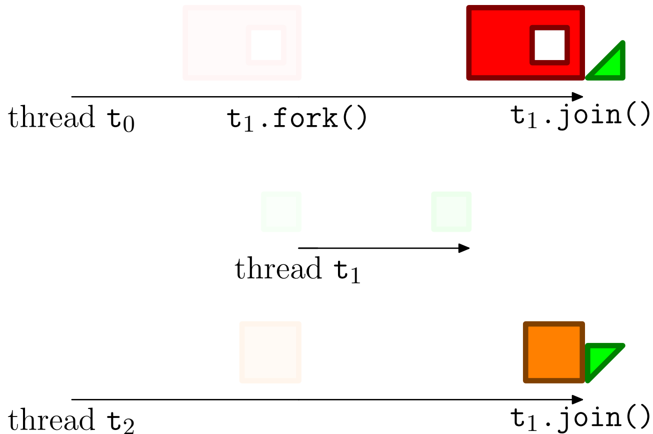




Fork and join in terms of resources (2)



Fork and join in terms of resources (2)



Hoare rules for fork and join

```
class Thread extends Object{




 void fork();


 void join();


 void run() { null }
}
```

Hoare rules for fork and join

```
class Thread extends Object{

 void fork();

 void join();

 void run() { null }
}
```

- When `t.fork()` is called, `t.run()` is executed in parallel.

Hoare rules for fork and join

```
class Thread extends Object{
 pred preFork = true; // to be extended in subclasses


 requires preFork; ensures true;
 void fork();


 void join();

 requires preFork; ensures true;
 void run() { null }
}
```

- When `t.fork()` is called, `t.run()` is executed in parallel.

Hoare rules for fork and join

```
class Thread extends Object{
 pred preFork = true;
 pred postJoin = true; // to be extended in subclasses

 requires preFork; ensures true;
 void fork();

 requires Join(this); ensures postJoin;
 void join();

 requires preFork; ensures postJoin;
 void run() { null }
}
```

- `t.join()` resumes when `t` terminates.
- `Join(t)`: the thread in which this formula appears can get back `t`'s postcondition when `t` terminates.

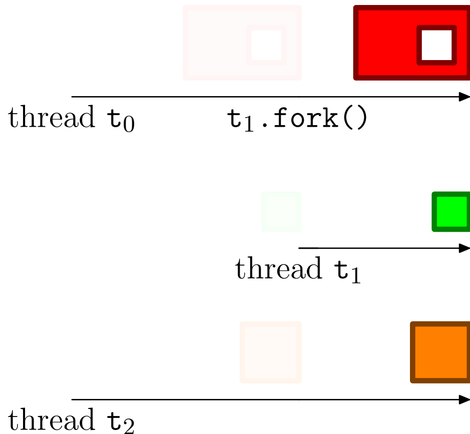
Hoare rules for fork and join

```
class Thread extends Object{
 pred preFork = true;
 pred postJoin = true; // to be extended in subclasses

 requires preFork; ensures true;
 void fork();

 requires Join(this); ensures postJoin;
 void join();

 requires preFork; ensures postJoin;
 void run() { null }
}
```

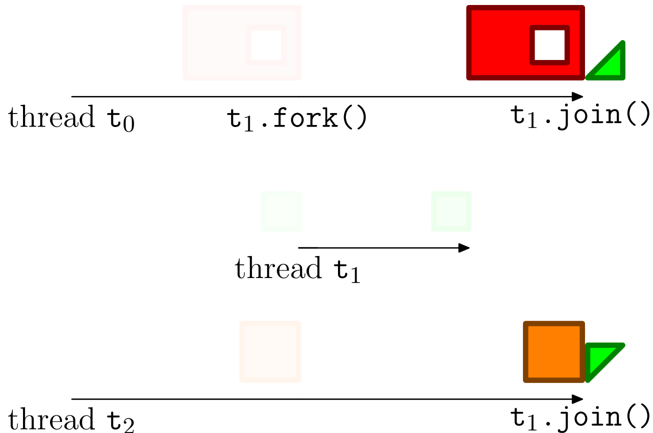
- `t.join()` resumes when `t` terminates.
- `Join(t)`: the thread in which this formula appears can get back `t`'s postcondition when `t` terminates.

↳ But this does not allow concurrent joiners.

Fork and join in terms of resources (2)



Fork and join in terms of resources (2)



Hoare rules for fork and join

```
class Thread extends Object{
 pred preFork = true;
 pred postJoin<perm p> = true;

 requires preFork; ensures true;
 void fork();

 requires Join(this,p); ensures postJoin<p>;
 void join();

 requires preFork; ensures postJoin<1>;
 void run() { null }
}
```

↳ We parameterize `Join` and `postJoin` by a permission.

- `Join(t,p)` give access to fraction `p` of thread `t`'s postcondition.

Hoare rules for fork and join

```
class Thread extends Object{
 pred preFork = true;
 group postJoin = true;

 requires preFork; ensures true;
 void fork();

 requires Join(this,p); ensures postJoin<p>;
 void join();

 requires preFork; ensures postJoin<1>;
 void run() { null }
}
```

- For soundness:
- `postJoin` is a special predicate: a group.

↳ It satisfies $\forall$`perm` p.`postJoin<p> *-* (postJoin<p/2> * postJoin<p/2>)`.

Objective 2: reentrant locks [APLAS'08]

Reentrant locks are the main primitive to acquire/release locks in Java.

- They can be acquired more than once (and released accordingly)

↳ Convenient for programmers (no need to acquire conditionally)

Objective 2: reentrant locks [APLAS'08]

Reentrant locks are the main primitive to acquire/release locks in Java.

- They can be acquired more than once (and released accordingly)

↳ Convenient for programmers (no need to acquire conditionally)

```
class Set{
  int size(){           // client and helper method





  }
  bool has(Element e){ // client method





  }
}
```

Objective 2: reentrant locks [APLAS'08]

Reentrant locks are the main primitive to acquire/release locks in Java.

- They can be acquired more than once (and released accordingly)

↳ Convenient for programmers (no need to acquire conditionally)

```
class Set{
  int size(){
    lock(this);
      ...
    unlock(this);
    return ...;
  }
  bool has(Element e){
    lock(this);
      bool result;
      if(size()==0) unlock(this); return false;
      else ...;     unlock(this); return ...;
  }
}
```

Objective 2: reentrant locks [APLAS'08]

Reentrant locks are the main primitive to acquire/release locks in Java.

- They can be acquired more than once (and released accordingly)

↳ Convenient for programmers (no need to acquire conditionally)

```
class Set{
  int size(){
    lock(this);
      ...
    unlock(this);
    return ...;
  }
  bool has(Element e){
    lock(this);
      bool result;
      if(size()==0) unlock(this); return false;
      else ...;     unlock(this); return ...;
  }
}
```

Objective 2: reentrant locks [APLAS'08]

In separation logic [O'Hearn'07]:

- Each lock guards a part of the heap called the lock's resource invariant.
- Resource invariants are exchanged between locks and threads:
  - When a lock is acquired, it lends its resource invariant to the acquiring thread.
  - When a lock is released, it claims back its resource invariant from the releasing thread.

Objective 2: reentrant locks [APLAS'08]

In separation logic [O'Hearn'07]:

- Each lock guards a part of the heap called the lock's resource invariant.
- Resource invariants are exchanged between locks and threads:
  - When a lock is acquired, it lends its resource invariant to the acquiring thread.
  - When a lock is released, it claims back its resource invariant from the releasing thread.

Resource invariants are represented by the distinguished *abstract predicate* `inv`:

```
class Object{
  pred inv = true;
}
```

Objective 2: reentrant locks [APLAS'08]

In separation logic [O'Hearn'07]:

- Each lock guards a part of the heap called the lock's resource invariant.
- Resource invariants are exchanged between locks and threads:
  - When a lock is acquired, it lends its resource invariant to the acquiring thread.
  - When a lock is released, it claims back its resource invariant from the releasing thread.

Objective 2: reentrant locks [APLAS'08]

In separation logic [O'Hearn'07]:

- Each lock guards a part of the heap called the lock's resource invariant.
- Resource invariants are exchanged between locks and threads:
  - When a lock is acquired, it lends its resource invariant to the acquiring thread.
  - When a lock is released, it claims back its resource invariant from the releasing thread.

- But this is unsound for reentrant locks!

↳ We need to distinguish between initial acquirements and reentrant acquirements.

Separation Logic for Reentrant Locks

4 formulas to speak about locks (where `S` is a *multiset*):

`Lockset(S)` | `S contains x` | `x.fresh` | `x.initialized`

For each thread, we track the set of currently held locks:

- `Lockset(S)`: `S` is the multiset of currently held locks.
- *S* `contains x`: lockset `S` contains lock `x`.

Separation Logic for Reentrant Locks

4 formulas to speak about locks (where `S` is a *multiset*):

`Lockset(S)` | `S contains x` | `x.fresh` | `x.initialized`

For each thread, we track the set of currently held locks:

- `Lockset(S)`: `S` is the multiset of currently held locks.
- *S* `contains x`: lockset `S` contains lock `x`.

For each lock, we track its abstract lock state:

- `x.fresh`: `x`'s resource invariant is not initialized
- `x.initialized`: `x`'s resource invariant is initialized.

Initializing Locks

$$\frac{C\texttt{<}\bar{\pi}\texttt{>} <: \Gamma(\texttt{x})}{\Gamma \vdash \begin{array}{c}\{\texttt{true}\}\\ \texttt{x} = \texttt{new}\ C\texttt{<}\bar{\pi}\texttt{>}\\ \{\texttt{x.init} * \texttt{C classof x} * \circledast_{\Gamma(\texttt{u}) <: \texttt{Object}}\ \texttt{x != u} * \texttt{x.fresh}\}\end{array}} \text{(New)}$$

↳ After creation a lock cannot be acquired: `x.initialized` fails to match (Lock)'s precondition.

Initializing Locks

$$\frac{C\texttt{<}\bar{\pi}\texttt{>} <: \Gamma(\texttt{x})}{\Gamma \vdash \begin{array}{c}\{\texttt{true}\} \\ \texttt{x} = \texttt{new}\ C\texttt{<}\bar{\pi}\texttt{>} \\ \{\texttt{x.init} * \texttt{C classof x} * \circledast_{\Gamma(\texttt{u})<:\texttt{Object}}\ \texttt{x != u} * \texttt{x.fresh}\}\end{array}} \text{(New)}$$

↳ After creation a lock cannot be acquired: `x.initialized` fails to match (Lock)'s precondition.

$$\frac{}{\Gamma \vdash \begin{array}{c}\{\texttt{Lockset(S)} * \texttt{x.inv} * \texttt{x.fresh}\} \\ \texttt{x.commit} \\ \{\texttt{Lockset(S)} * \neg(\texttt{S contains x}) * \texttt{x.initialized}\}\end{array}} \text{(Commit)}$$

↳ `x.commit` is a no-op.

↳ After being committed a lock can be acquired: (Commit)'s postcondition entails `x.initialized`.

Acquiring Locks

$$\frac{}{\Gamma \vdash \begin{array}{c}\{\texttt{Lockset(S)} * (\neg \texttt{S contains x}) * \texttt{x.initialized}\} \\ \texttt{lock(x)} \\ \{\texttt{Lockset(x} \cdot \texttt{S)} * \texttt{x.inv}\}\end{array}} \text{(Lock)}$$

↳ First acquirement: resource invariants obtained.

↳ Nothing special to handle subclassing.

$$\frac{}{\Gamma \vdash \{\texttt{Lockset(x} \cdot \texttt{S)}\}\texttt{lock(x)}\{\texttt{Lockset(x} \cdot \texttt{x} \cdot \texttt{S)}\}} \text{(Re-Lock)}$$

↳ Reentrant acquirement: x's resource invariant not obtained.

Releasing Locks

- The 2 rules for releasing locks are dual to the rules for acquirement.

↳ Hence, we do not discuss them.

Objectives 1 and 2: Achievements

- A sound verification system for realistic multithreaded Java programs.
- Usability tested against challenging case studies:
  - Concurrent iterator
  - Lock coupling algorithm (still some limitations)
- Algorithmic verification still to be developed

Objectives 1 and 2: Achievements

- A sound verification system for realistic multithreaded Java programs.
- Usability tested against challenging case studies:
  - Concurrent iterator
  - Lock coupling algorithm (still some limitations)
- Algorithmic verification still to be developed

After that:

- 3 new analyses based on separation logic

↳ 2 of these analyses are sketched in the next slides

1st Analysis: Fast Disproving of Entailment [IWACO'09]

Goal:

- Disprove entailment between separation logic formulas

↳ I.e. to prove $A \not\models B$

1st Analysis: Fast Disproving of Entailment [IWACO'09]

Goal:

- Disprove entailment between separation logic formulas

↳ I.e. to prove $A \nvdash B$

Usefulness:

- Program verifiers spend their time checking entailment.

↳ I.e. given the program's state A, and the next command's precondition B,

↳ program verifiers have to find a F such that $A \vdash B \star F$.

1st Analysis: Fast Disproving of Entailment [IWACO'09]

Goal:

- Disprove entailment between separation logic formulas

↳ I.e. to prove $A \nvdash B$

Usefulness:

- Program verifiers spend their time checking entailment.

↳ I.e. given the program's state A, and the next command's precondition B,

↳ program verifiers have to find a F such that $A \vdash B \star F$.

- In full separation logic, $\vdash$ is undecidable.
- If we can prove that $A \nvdash B$, then we know that F cannot be found.

↳ This avoids trying to prove unprovable programs.

Disproving Technique

Soundness of the proof system:

$$A \vdash B \text{ implies } (\forall h, h \models A \rightarrow h \models B)$$

Disproving Technique

Soundness of the proof system:

$$A \vdash B \text{ implies } (\forall h, h \models A \rightarrow h \models B)$$

Contraposition:

$$(\exists h, h \models A \wedge \neg h \models B) \text{ implies } A \nvdash B$$

Goal of this work:

- Take A and B and prove that $A \nvdash B$
- By discriminating models of A and B

Disproving Technique (classical semantics)

Objective:

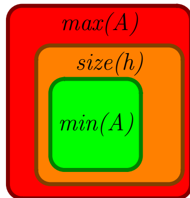
$$\text{Find } h \text{ such that } h \models A \text{ and } \neg h \models B$$

- We compute bounds on the size of models.
- *max* : Formula → Size
- *min* : Formula → Size
- *size* : Model → Size

Disproving Technique (classical semantics)

Objective:

Find h such that $h \models A$ and $\neg h \models B$



- We compute bounds on the size of models.
- *max* : Formula $\rightarrow$ Size
- *min* : Formula $\rightarrow$ Size
- *size* : Model $\rightarrow$ Size

Properties of *max* and *min* (classical semantics):

$$\forall h, h \models A \text{ implies } min(A) \leqslant size(h) \leqslant max(A)$$

Disproving Technique (classical semantics)

$$(\exists h, h \models A \wedge \neg h \models B) \text{ implies } A \not\models B$$

$$\forall h, h \models A \text{ implies } min(A) \leqslant size(h) \leqslant max(A)$$

$$\downarrow$$

$$max(A) < min(B) \text{ implies } A \not\models B$$

1st Analysis: Achievements

- A fast technique to disprove entailment.
- Two different trade offs between speed and precision

1st Analysis: Achievements

- A fast technique to disprove entailment.
- Two different trade offs between speed and precision

  (two ways to define Size)

1st Analysis: Achievements

- A fast technique to disprove entailment.
- Two different trade offs between speed and precision
  (two ways to define Size)
- Proven correct in Coq
- License-left proof scripts

2nd Analysis: Optimizations by Proof Rewriting [SAS'09]

- We parallelize and optimize proven programs.

To parallelize programs, you need to know:

- What data is accessed by programs.
- What data is not accessed by programs.

2nd Analysis: Optimizations by Proof Rewriting [SAS'09]

- We parallelize and optimize proven programs.

To parallelize programs, you need to know:

- What data is accessed by programs.
- What data is not accessed by programs.

The good thing is:

- Separation logic proofs exhibit how data is accessed (or not):
  - Antiframes exhibit data that is accessed. (explained next)
  - The (Frame) rule exhibits data that is not accessed. (explained next)

2nd Analysis: Optimizations by Proof Rewriting [SAS'09]

- We parallelize and optimize proven programs.

To parallelize programs, you need to know:

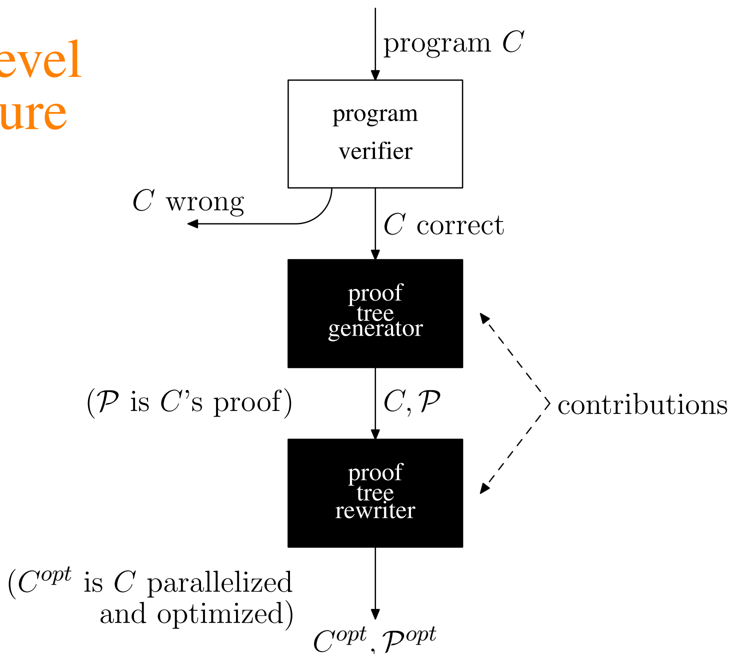
- What data is accessed by programs.
- What data is not accessed by programs.

The good thing is:

- Separation logic proofs exhibit how data is accessed (or not):
  - Antiframes exhibit data that is accessed. (explained next)
  - The (Frame) rule exhibits data that is not accessed. (explained next)

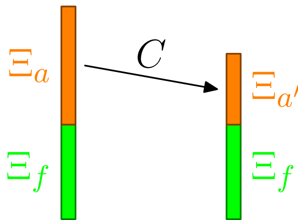
- Optimizations are expressed with a rewrite system between proof trees.
- Proof trees are derivations of Hoare triplets.

High-Level Procedure



Separation Logic: (Frame) rule

$$\frac{\{\Xi_a\}C\{\Xi_{a'}\}}{\{\Xi_a \star \Xi_f\}C\{\Xi_{a'} \star \Xi_f\}}\ (\text{Frame } \Xi_f)$$



- Ξ_a is the *antiframe* ← accessed data
- Ξ_f is the *frame* ← not-accessed data

- Later, (Fr) sometimes abbreviates (Frame).

With Frames: Parallelization Is Easy

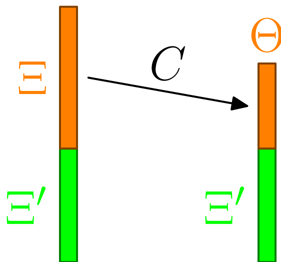
$$\dfrac{\dfrac{\{\Xi\}C\{\Theta\}}{\{\Xi \star \Xi'\}C\{\Theta \star \Xi'\}}\ (\text{Fr } \Xi') \quad \dfrac{\{\Xi'\}C'\{\Theta'\}}{\{\Theta \star \Xi'\}C'\{\Theta \star \Theta'\}}\ (\text{Fr } \Theta)}{\{\Xi \star \Xi'\}C;\ C'\{\Theta \star \Theta'\}}\ (\text{Seq})$$

$\downarrow$ Parallelize

$$\dfrac{\{\Xi\}C\{\Theta\} \qquad \{\Xi'\}C'\{\Theta'\}}{\{\Xi \star \Xi'\}C \parallel C'\{\Theta \star \Theta'\}}\ (\text{Parallel})$$

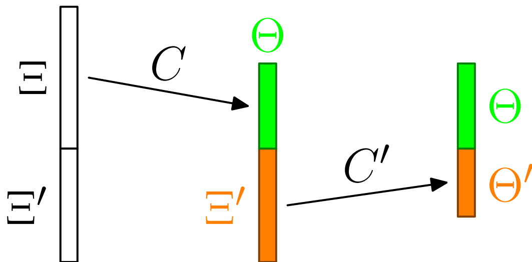
Parallelize's left hand side

$$\dfrac{\dfrac{\{\Xi\}C\{\Theta\}}{\{\Xi \star \Xi'\}C\{\Theta \star \Xi'\}}\text{(Fr } \Xi'\text{)} \quad \dfrac{\{\Xi'\}C'\{\Theta'\}}{\{\Theta \star \Xi'\}C'\{\Theta \star \Theta'\}}\text{(Fr } \Theta\text{)}}{\{\Xi \star \Xi'\}C;\, C'\{\Theta \star \Theta'\}}\text{(Seq)}$$



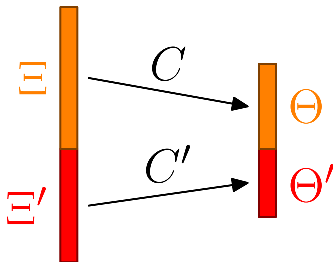
Parallelize's left hand side

$$\dfrac{\dfrac{\{\Xi\}C\{\Theta\}}{\{\Xi \star \Xi'\}C\{\Theta \star \Xi'\}}\,(\text{Fr } \Xi') \quad \dfrac{\{\Xi'\}C'\{\Theta'\}}{\{\Theta \star \Xi'\}C'\{\Theta \star \Theta'\}}\,(\text{Fr } \Theta)}{\{\Xi \star \Xi'\}C;\, C'\{\Theta \star \Theta'\}}\,(\text{Seq})$$



Parallelize's right hand side

$$\frac{\{\Xi\}C\{\Theta\} \qquad \{\Xi'\}C'\{\Theta'\}}{\{\Xi \star \Xi'\}C \parallel C'\{\Theta \star \Theta'\}}\ \text{(Parallel)}$$

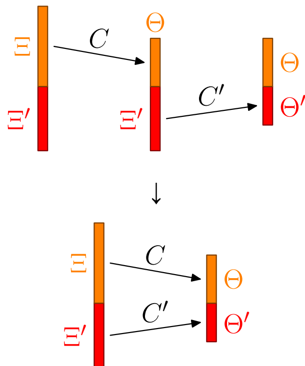


Parallelize

$$\dfrac{\dfrac{\{\Xi\}C\{\Theta\}}{\{\Xi \star \Xi'\}C\{\Theta \star \Xi'\}}\text{(Fr }\Xi'\text{)} \qquad \dfrac{\{\Xi'\}C'\{\Theta'\}}{\{\Theta \star \Xi'\}C'\{\Theta \star \Theta'\}}\text{(Fr }\Theta\text{)}}{\{\Xi \star \Xi'\}C;\, C'\{\Theta \star \Theta'\}}\text{(Seq)}$$

$\downarrow$ Parallelize

$$\dfrac{\{\Xi\}C\{\Theta\} \qquad \{\Xi'\}C'\{\Theta'\}}{\{\Xi \star \Xi'\}C \parallel C'\{\Theta \star \Theta'\}}\text{(Parallel)}$$



2nd Analysis: Achievements

- An entirely new technique to parallelize and optimize programs.
  - ↳ Other optimizations than parallelization have been studied.
- No ad-hoc analyses: separation logic proofs are taken as analyses.
- Can parallelize any code (i.e. not focused on loops).
- Soundness is easier to prove than for classical approaches.
- License-left prototype implementation.

Related Work

Program verification:

- Separation logic for sequential Java [Parkinson'05,Distefano et al.'08,Chin et al.'08]
- Separation logic for multithreaded C [Gotsman et al.'07,Appel et al.'07]
- Boogie for multithreaded C# [Barnett et al.'04,Jacobs et al.'06]
- ESC/Java2 for Java [Leino et al.'02,Kiniry et al.'04]

Related Work

Program verification:

- Separation logic for sequential Java [Parkinson'05,Distefano et al.'08,Chin et al.'08]
- Separation logic for multithreaded C [Gotsman et al.'07,Appel et al.'07]
- Boogie for multithreaded C# [Barnett et al.'04,Jacobs et al.'06]
- ESC/Java2 for Java [Leino et al.'02,Kiniry et al.'04]

Algorithms for entailment/disproving:

- Sound and complete entailment in Smallfoot [Berdine et al.'04]
- Sound entailment in JStar [Parkinson et al.'08]
- Sound and complete entailment and refutation [Galmiche et al.'08]

Related Work

Program verification:

- Separation logic for sequential Java [Parkinson'05,Distefano et al.'08,Chin et al.'08]
- Separation logic for multithreaded C [Gotsman et al.'07,Appel et al.'07]
- Boogie for multithreaded C# [Barnett et al.'04,Jacobs et al.'06]
- ESC/Java2 for Java [Leino et al.'02,Kiniry et al.'04]

Algorithms for entailment/disproving:

- Sound and complete entailment in Smallfoot [Berdine et al.'04]
- Sound entailment in JStar [Parkinson et al.'08]
- Sound and complete entailment and refutation [Galmiche et al.'08]

Automatic parallelization:

- Many "classical" approaches
- By using separation logic [Raza et al.'09]

Main Publications

- Separation Logic Contracts for a Java-like Language with Fork/Join; Haack and Hurlin; AMAST'08
- Reasoning about Java's Reentrant Locks; Haack, Huisman, and Hurlin; APLAS'08
- Specifying and Checking Protocols of Multithreaded Classes; Hurlin; SAC'09
- Resource Usage Protocols for Iterators; Haack and Hurlin; Journal of Object Technology'09
- Size Does Matter: Two Certified Abstractions to Disprove Entailment in Intuitionistic and Classical Separation Logic; Hurlin, Bobot, and Summers; IWACO'09
- Automatic Parallelization and Optimization of Programs by Proof Rewriting; Hurlin; SAS'09

Main Publications

- Separation Logic Contracts for a Java-like Language with Fork/Join; Haack and Hurlin; AMAST'08
- Reasoning about Java's Reentrant Locks; Haack, Huisman, and Hurlin; APLAS'08
- Specifying and Checking Protocols of Multithreaded Classes; Hurlin; SAC'09
- Resource Usage Protocols for Iterators; Haack and Hurlin; Journal of Object Technology'09
- Size Does Matter: Two Certified Abstractions to Disprove Entailment in Intuitionistic and Classical Separation Logic; Hurlin, Bobot, and Summers; IWACO'09
- Automatic Parallelization and Optimization of Programs by Proof Rewriting; Hurlin; SAS'09

Developments:

1. Some Coq proofs for the AMAST and APLAS papers.
2. ocaml implementation of some of the techniques described in the SAC paper.
3. Full Coq proofs for the IWACO paper.
4. ocaml and Java+tom prototype implementation of the SAS paper.

Conclusion

First, we developed:

- A sound verification system for multithreaded Java programs in separation logic,

Conclusion

First, we developed:

- A sound verification system for multithreaded Java programs in separation logic,

↳ that uses realistic primitives,

Conclusion

First, we developed:

- A sound verification system for multithreaded Java programs in separation logic,
- ↳ that uses realistic primitives,
- ↳ and that handles challenging examples (iterator, lock-coupling).

Conclusion

First, we developed:

- A sound verification system for multithreaded Java programs in separation logic,

↳ that uses realistic primitives,

↳ and that handles challenging examples (iterator, lock-coupling).

Second:

- We extended previous work on protocols. (omitted in this talk)
- We discovered a fast algorithm to disprove entailment.
- We showed how to parallelize and optimize programs by rewriting their proofs.

Future Work

For the verification system:

- Implementing it!
- Doing a large case study

For the disproving algorithm:

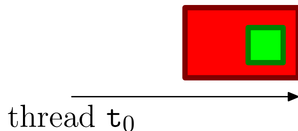
- Extension to object-orientation
- ↳ By keeping its simplicity and its usefulness (not straightforward)

For the parallelizing analysis:

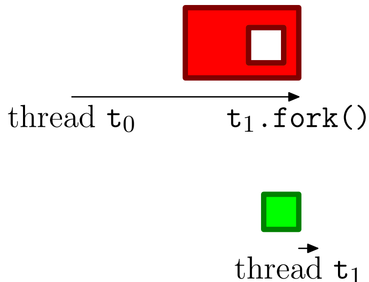
- Extension to object-oriented programs (easy)
- Extension to loops → to battle it out with classical parallelizers!

Thank you

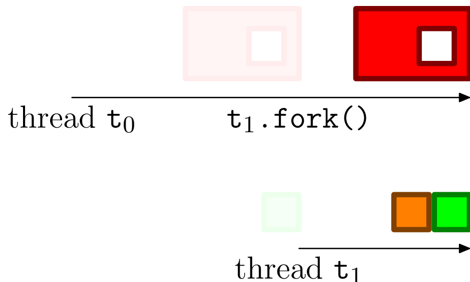
Fork and join in terms of resources (2)



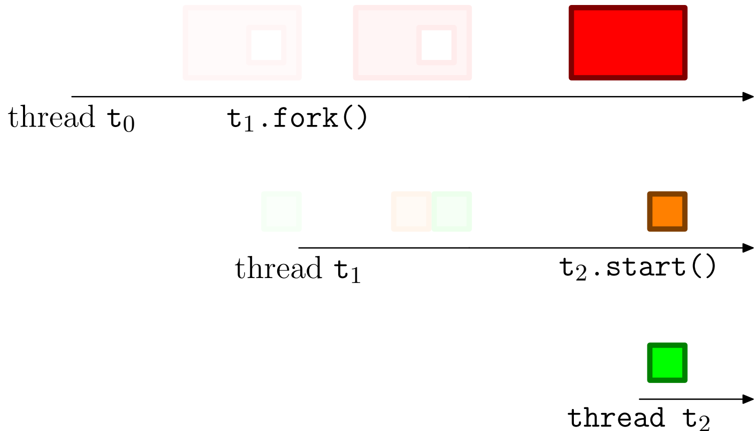
Fork and join in terms of resources (2)



Fork and join in terms of resources (2)



Fork and join in terms of resources (2)



Releasing Locks

$$\frac{}{\Gamma \vdash \{\texttt{Lockset(x}\cdot\texttt{x}\cdot\texttt{S)}\}\texttt{unlock}(x)\{\texttt{Lockset(x}\cdot\texttt{S)}\}}\ \text{(Re-Unlock)}$$

↳ Releasing x but x's reentrancy level > 1: invariant not abandoned.

$$\frac{}{\Gamma \vdash \{\texttt{Lockset(x}\cdot\texttt{S)} * x.\texttt{inv}\}\texttt{unlock}(x)\{\texttt{Lockset(S)}\}}\ \text{(Unlock)}$$

↳ x's reentrancy level not known to be > 1, x's resource invariant abandoned.

Disproving Technique (classical semantics)

$$(\exists h, h \models A \wedge \neg h \models B) \text{ implies } A \not\models B$$

$$\forall h, h \models A \text{ implies } \mathit{min}(A) \leqslant \mathit{size}(h) \leqslant \mathit{max}(A)$$

$$\downarrow$$

$$\mathit{max}(A) < \mathit{min}(B) \text{ implies } A \not\models B$$

Defining *size*

- $size(h) \triangleq$ sum of h's permissions
- $size$: Model $\rightarrow$ $Perm$

Defining *size*

- $size(h) \triangleq$ sum of h's permissions
- $size$: Model $\rightarrow Perm$

- Models h are lists of triples of an address, a permission, and a value.

↳ An example model is $(42, \pi, 3) :: (47, \pi', -5) :: (42, \pi'', 0) :: []$.

$$size((42, \pi, 3) :: (47, \pi', -5) :: (42, \pi'', 0) :: []) = \pi + \pi' + \pi''$$

Defining *max/min* (classical semantics)(excerpt)

$$max(_ \overset{\pi}{\mapsto} _) = \pi$$
$$max(A \star B) = max(A) +_{\pi} max(B)$$

$$min(_ \overset{\pi}{\mapsto} _) = \pi$$
$$min(A \star B) = min(A) +_{\pi} min(B)$$

$$\begin{array}{lll} h \models a \overset{\pi}{\mapsto} v & \text{iff} & h = (a, \pi, v) \\ h \models A \star B & \text{iff} & \exists h_A, h_B, h = h_A \uplus h_B, h_A \models A \text{ and } h_B \models B \end{array}$$

Defining *max/min* (classical semantics)(excerpt)

$max(A \wedge B) = min_{\pi}\,(max(A), max(B))$

$max(A \vee B) = max_{\pi}(max(A), max(B))$

$min(A \wedge B) = max_{\pi}(min(A), min(B))$

$min(A \vee B) = min_{\pi}\,(min(A), min(B))$

$$h \models A \wedge B \quad \text{iff} \quad h \models A \text{ and } h \models B$$
$$h \models A \vee B \quad \text{iff} \quad h \models A \text{ or } h \models B$$

Defining *max*/*min* (classical semantics)(excerpt)

$max(_ \overset{\pi}{\mapsto} _) = \pi$

$max(A \star B) = max(A) +_{\pi} max(B)$

$min(_ \overset{\pi}{\mapsto} _) = \pi$

$min(A \star B) = min(A) +_{\pi} min(B)$

$$h \models a \overset{\pi}{\mapsto} v \quad \text{iff} \quad h = (a, \pi, v)$$

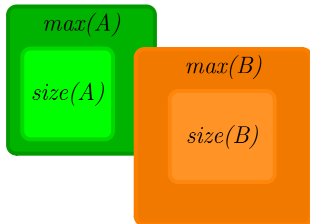
$$h \models A \star B \quad \text{iff} \quad \exists h_A, h_B, h = h_A \uplus h_B, h_A \models A \text{ and } h_B \models B$$

Defining *max/min* (classical semantics)(excerpt)

$max(_ \stackrel{\pi}{\mapsto} _) = \pi$
$max(A \star B) = max(A) +_\pi max(B)$

$min(_ \stackrel{\pi}{\mapsto} _) = \pi$
$min(A \star B) = min(A) +_\pi min(B)$

$$h \models a \stackrel{\pi}{\mapsto} v \quad \text{iff} \quad h = (a, \pi, v)$$
$$h \models A \star B \quad \text{iff} \quad \exists h_A, h_B, h = h_A \uplus h_B, h_A \models A \text{ and } h_B \models B$$

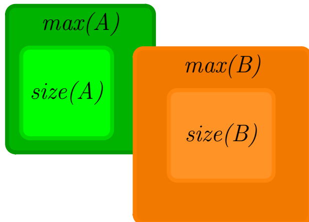


Defining *max/min* (classical semantics)(excerpt)

$max(_ \overset{\pi}{\mapsto} _) = \pi$
$max(A \star B) = max(A) +_{\pi} max(B)$

$min(_ \overset{\pi}{\mapsto} _) = \pi$
$min(A \star B) = min(A) +_{\pi} min(B)$

$$h \models a \overset{\pi}{\mapsto} v \quad \text{iff} \quad h = (a, \pi, v)$$
$$h \models A \star B \quad \text{iff} \quad \exists h_A, h_B, h = h_A \uplus h_B, h_A \models A \text{ and } h_B \models B$$



Defining *max/min* (classical semantics)(excerpt)

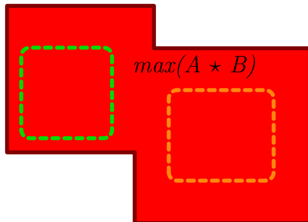
$max(_ \overset{\pi}{\mapsto} _) = \pi$

$max(A \star B) = max(A) +_\pi max(B)$

$min(_ \overset{\pi}{\mapsto} _) = \pi$

$min(A \star B) = min(A) +_\pi min(B)$

$$h \models a \overset{\pi}{\mapsto} v \quad \text{iff} \quad h = (a, \pi, v)$$
$$h \models A \star B \quad \text{iff} \quad \exists h_A, h_B, h = h_A \uplus h_B, h_A \models A \text{ and } h_B \models B$$



Defining *max*/*min* (classical semantics)

$$
\begin{aligned}
max(_ \overset{\pi}{\mapsto} _) &= \pi \\
max(A \star B) &= max(A) +_\pi max(B) \\
max(A \mathrel{-\!\!*} B) &= max(B) -_\pi min(A)
\end{aligned}
$$

$$
\begin{aligned}
min(_ \overset{\pi}{\mapsto} _) &= \pi \\
min(A \star B) &= min(A) +_\pi min(B) \\
min(A \mathrel{-\!\!*} B) &= min(B) -_\pi max(A)
\end{aligned}
$$

$$
\begin{array}{lll}
h \models a \overset{\pi}{\mapsto} v & \text{iff} & h = (a, \pi, v) \\
h \models A \star B & \text{iff} & \exists h_A, h_B, h = h_A \uplus h_B, h_A \models A \text{ and } h_B \models B \\
h \models A \mathrel{-\!\!*} B & \text{iff} & \forall h_A, h_A \models A \text{ and } h_A \text{ and } h \text{ are compatible} \\
 & & \quad \text{implies } h_A \uplus h \models A \star B
\end{array}
$$

Defining *max*/*min* (classical semantics)

$max(\forall\pi.A) = max(A[\pi_w/\pi])$
$max(\exists\pi.A) = max(A[\pi_w/\pi])$
$max(\forall v.A) = max(\exists v.A) = max(A)$

$min(\forall\pi.A) = min(A[\pi_w/\pi])$
$min(\exists\pi.A) = min(A[\varepsilon/\pi])$
$min(\forall v.A) = min(\exists v.A) = min(A)$

↳ Standard semantics of quantifiers (omitted)

- ε is an infinitely small permission

Defining *max*/*min* (classical semantics)

$max(b) = \infty$ $min(b) = \pi_0$

$$h \models b \quad \text{iff} \quad oracle(b)$$

- b is a pure formula, i.e., it does not depend on the heap h.
- π_0 is the minimal permission.

Toy Example

$$\Lambda^{n,m,k}_{x,y,z} \triangleq x \mapsto [f:n] \star y \mapsto [f:m] \star z \mapsto [f:k]$$

$$\dfrac{\dfrac{\dfrac{}{\{\Lambda^{-}_{x}\} x \to f = n \{\Lambda^{n}_{x}\}}\ (\text{Mutate})}{\{\Lambda^{-,-,-}_{x,y,z}\} x \to f = n \{\Lambda^{n,-,-}_{x,y,z}\}}\ (\text{Fr } \Lambda^{-,-}_{y,z}) \qquad \dfrac{\dfrac{\dfrac{}{\{\Lambda^{-}_{y}\} y \to f = m \{\Lambda^{m}_{y}\}}\ (\text{Mutate})}{\{\Lambda^{n,-,-}_{x,y,z}\} y \to f = m \{\Lambda^{n,m,-}_{x,y,z}\}}\ (\text{Fr } \Lambda^{n,-}_{x,z}) \quad \dfrac{\dfrac{}{\{\Lambda^{-}_{z}\} z \to f = k \{\Lambda^{k}_{z}\}}\ (\text{Mutate})}{\{\Lambda^{n,m,-}_{x,y,z}\} z \to f = k \{\Lambda^{n,m,k}_{x,y,z}\}}\ (\text{Fr } \Lambda^{n,m}_{x,y})}{\{\Lambda^{n,-,-}_{x,y,z}\} y \to f = m; z \to f = k \{\Lambda^{n,m,k}_{x,y,z}\}}\ (\text{Seq})}{\{\Lambda^{-,-,-}_{x,y,z}\} x \to f = n; y \to f = m; z \to f = k \{\Lambda^{n,m,k}_{x,y,z}\}}\ (\text{Seq})$$

$$\downarrow$$

$$\dfrac{\dfrac{}{\{\Lambda^{-}_{x}\} x \to f = n \{\Lambda^{n}_{x}\}}\ (\text{Mutate}) \qquad \dfrac{\dfrac{}{\{\Lambda^{-}_{y}\} y \to f = m \{\Lambda^{m}_{y}\}}\ (\text{Mutate}) \quad \dfrac{}{\{\Lambda^{-}_{z}\} z \to f = k \{\Lambda^{k}_{z}\}}\ (\text{Mutate})}{\{\Lambda^{-,-}_{y,z}\} y \to f = m \parallel z \to f = k \{\Lambda^{m,k}_{y,z}\}}\ (\text{Parallel})}{\{\Lambda^{-,-,-}_{x,y,z}\} x \to f = n \parallel (y \to f = m \parallel z \to f = k) \{\Lambda^{n,m,k}_{x,y,z}\}}\ (\text{Parallel})$$

Toy Example

$$\Lambda^{n,m,k}_{x,y,z} \triangleq x \mapsto [f:n] \star y \mapsto [f:m] \star z \mapsto [f:k]$$

$$\dfrac{\dfrac{\dfrac{}{\{\Lambda^{-}_{x}\}x \to f = n\{\Lambda^{n}_{x}\}}\,(\text{Mutate})}{\{\Lambda^{-,-,-}_{x,y,z}\}x \to f = n\{\Lambda^{n,-,-}_{x,y,z}\}}\,(\text{Fr } \Lambda^{-,-}_{y,z}) \quad \dfrac{\dfrac{\dfrac{}{\{\Lambda^{-}_{y}\}y \to f = m\{\Lambda^{m}_{y}\}}\,(\text{Mutate})}{\{\Lambda^{n,-,-}_{x,y,z}\}y \to f = m\{\Lambda^{n,m,-}_{x,y,z}\}}\,(\text{Fr } \Lambda^{n,-}_{x,z}) \quad \dfrac{\dfrac{}{\{\Lambda^{-}_{z}\}z \to f = k\{\Lambda^{k}_{z}\}}\,(\text{Mutate})}{\{\Lambda^{n,m,-}_{x,y,z}\}z \to f = k\{\Lambda^{n,m,k}_{x,y,z}\}}\,(\text{Fr } \Lambda^{n,m}_{x,y})}{\{\Lambda^{n,-,-}_{x,y,z}\}y \to f = m; z \to f = k\{\Lambda^{n,m,k}_{x,y,z}\}}\,(\text{Seq})}{\{\Lambda^{-,-,-}_{x,y,z}\}x \to f = n; y \to f = m; z \to f = k\{\Lambda^{n,m,k}_{x,y,z}\}}\,(\text{Seq})$$

$$\downarrow$$

$$\dfrac{\dfrac{}{\{\Lambda^{-}_{x}\}x \to f = n\{\Lambda^{n}_{x}\}}\,(\text{Mutate}) \quad \dfrac{\dfrac{}{\{\Lambda^{-}_{y}\}y \to f = m\{\Lambda^{m}_{y}\}}\,(\text{Mutate}) \quad \dfrac{}{\{\Lambda^{-}_{z}\}z \to f = k\{\Lambda^{k}_{z}\}}\,(\text{Mutate})}{\{\Lambda^{-,-}_{y,z}\}y \to f = m \parallel z \to f = k\{\Lambda^{m,k}_{y,z}\}}\,(\text{Parallel})}{\{\Lambda^{-,-,-}_{x,y,z}\}x \to f = n \parallel (y \to f = m \parallel z \to f = k)\{\Lambda^{n,m,k}_{x,y,z}\}}\,(\text{Parallel})$$

- Hypothesis: the left hand side is a valid proof tree.
- Soundness follows from the inclusion of the rhs's leaves in the lhs's leaves.

Guarantees of the Rewrite System

The rewrite system modifies programs but preserves specifications:

$$\frac{\mathscr{P}}{\{\Xi\}C\{\Theta\}}$$

$$\downarrow$$

$$\frac{\mathscr{Q}}{\{\Xi\}C'\{\Theta\}}$$

- The program and the proof are modified: $\mathscr{P}, C \rightarrow \mathscr{Q}, C'$,

↳ but specifications are preserved: $\Xi, \Theta \rightarrow \Xi, \Theta$.

Conjecture:

- Programs related with $\rightarrow$ are equivalent from a big step p.o.v.

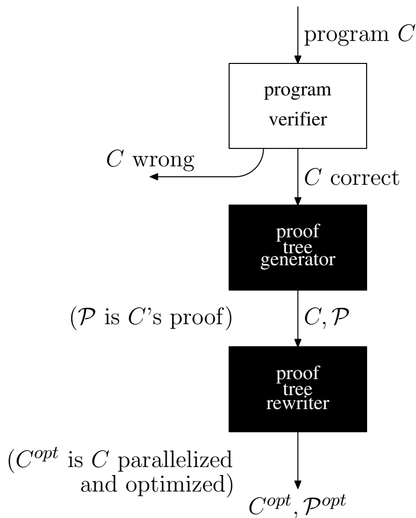
We Want the Frames ♫

- The (Frame) rule is the central ingredient of our procedure.
- Problem: Existing program verifiers (e.g. smallfoot) do not make frames explicit.

$$\frac{\Pi \vdash F = E \quad \ldots}{\{\Pi \mathrel{¦} F \mapsto [\rho]\} E \rightarrow f = G \{\Pi \mathrel{¦} F \mapsto [\rho']\}} \text{(Mutate)}$$

↳ Π is “too big”: there exists a “smaller” antiframe Π_a such that $\Pi_a \vdash F = E$.

Recall the big picture ?



In the proof tree generator:

- Proof rules with explicit frames.
- But still usage of the program's verifier normal rules for verification.

Next slides:

Proof rules with explicit {anti-,} frames

We Want the Frames

Berdine, Calcagno, and O'Hearn "Symbolic Execution with Separation Logic"

$$\frac{\Pi \vdash F = E \quad \ldots}{\{\Pi \mid F \mapsto [\rho]\} E \to f = G \{\Pi \mid F \mapsto [\rho']\}} \text{(Mutate)}$$

With explicit frames and antiframes

$$\frac{\dfrac{\Pi_a \vdash F = E \quad \ldots}{\{\Pi_a \mid F \mapsto [\rho]\} E \to f = G \{\Pi_a \mid F \mapsto [\rho']\}} \text{(Mutate)}}{\{(\Pi_a \mid F \mapsto [\rho]) \star \Xi_f\} E \to f = G \{(\Pi_a \mid F \mapsto [\rho']) \star \Xi_f\}} \text{(Frame } \Xi_f\text{)}$$

We Want the Frames

Berdine, Calcagno, and O'Hearn "Symbolic Execution with Separation Logic"

$$\frac{\ldots \quad x' \text{ fresh} \quad \Pi \vdash F = E}{\{\Pi \mid \Sigma \star F \mapsto [\rho]\} x := E \to f \{\ldots \wedge \Pi[x'/x] \mid (\Sigma \star F \mapsto [\rho])[x'/x]\}} \text{(Lookup)}$$

With explicit frames and antiframes

$$\frac{\dfrac{\begin{array}{l} x' \text{ fresh} \quad \ldots \\ \Pi_a \vdash F = E \\ \Xi = \Pi_a[x'/x] \wedge \ldots \mid (\Sigma_a \star F \mapsto [\rho])[x'/x] \end{array}}{\{\Pi_a \mid \Sigma_a \star F \mapsto [\rho]\} x := E \to f \{\Xi\}} \text{(Lookup)} \quad x \notin \Xi_f}{\{\underbrace{(\Pi_a \mid \Sigma_a \star F \mapsto [\rho])}_{\text{antiframe needed to prove } E \mapsto [\rho] \text{ and affected by [x'/x]}} \star \underbrace{\Xi_f}_{\text{frame unaffected by [x'/x]}}\} x := E \to f \{\Xi \star \Xi_f\}} \text{(Frame } \Xi_f)$$

We Want the Frames

Berdine, Calcagno, and O'Hearn "Symbolic Execution with Separation Logic"

$$\frac{\Pi \vdash \bot}{\{\Pi \mathrel{¦} \Sigma\} C \{\Theta\}} \text{ (Inconsistent)}$$

With explicit frames and antiframes

$$\frac{\dfrac{\Pi_a \vdash \bot}{\{\Pi_a \mathrel{¦} \texttt{emp}\} C \{\Theta\}} \text{ (Inconsistent)}}{\{\underbrace{\Pi_a \mathrel{¦} \texttt{emp}}_{\text{sufficient antiframe to prove } \bot} \star \underbrace{\Pi_f \mathrel{¦} \Sigma_f}_{\text{frame}}\} C \{\Theta\}} \text{ (Frame } \Pi_f \mathrel{¦} \Sigma_f\text{)}$$

- $\texttt{emp} \triangleq$ the heap is empty.

Shape of Generated Trees

- Because of the algorithm used in program verifiers,
↳ proof trees have a special shape.

For successive commands $C_0, C_1, C_2, \ldots$, proof trees have this shape:

$$\dfrac{\dfrac{..}{\{..\}C_0\{..\}}(\text{Fr } \Xi) \quad \dfrac{\dfrac{..}{\{..\}C_1\{..\}}(\text{Fr } \Theta) \quad \dfrac{\dfrac{..}{\{..\}C_2\{..\}}(\text{Fr } \Xi') \quad ..}{\{..\}C_2;\,..\{..\}}}{\{..\}C_1;\, C_2;\, ..\{..\}}(\text{Seq})}{\{..\}C_0;\, C_1;\, C_2;\, ..\{..\}}(\text{Seq})$$

- A (Frame) at each command.
↳ Problem: Frames are redundant (i.e., $\Xi \cup \Theta \cup \Xi' \neq \emptyset$).
↳ In practice, this must be avoided for optimizations to fire.

We Want the Frames ♫

- We rewrite proof trees to frame multiple commands,

↳ i.e., we *factorize frames*.

- Below, Ξ_c is the factorized frame.

Guard: $\Xi_f \Leftrightarrow \Xi_{f0} \star \Xi_c$ and $\Theta_f \Leftrightarrow \Theta_{f0} \star \Xi_c$

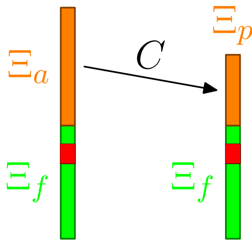
$$\dfrac{\dfrac{\{\Xi_a\}C\{\Xi_p\}}{\{\Xi_a \star \Xi_f\}C\{\Xi_p \star \Xi_f\}}\ (\text{Frame } \Xi_f) \quad \dfrac{\{\Theta_a\}C'\{\Theta_p\}}{\{\Theta_a \star \Theta_f\}C'\{\Theta_p \star \Theta_f\}}\ (\text{Frame } \Theta_f)}{\{\Xi_a \star \Xi_f\}C;\ C'\{\Theta_p \star \Theta_f\}}\ (\text{Seq})$$

$$\downarrow \underline{\text{FactorizeFrames}}$$

$$\dfrac{\dfrac{\dfrac{\{\Xi_a\}C\{\Xi_p\}}{\{\Xi_a \star \Xi_{f0}\}C\{\Xi_p \star \Xi_{f0}\}}\ (\text{Fr } \Xi_{f0}) \quad \dfrac{\{\Theta_a\}C'\{\Theta_p\}}{\{\Theta_a \star \Theta_{f0}\}C'\{\Theta_p \star \Theta_{f0}\}}\ (\text{Fr } \Theta_{f0})}{\{\Xi_a \star \Xi_{f0}\}C;\ C'\{\Theta_p \star \Theta_{f0}\}}\ (\text{Seq})}{\{\Xi_a \star \Xi_f\}C;\ C'\{\Theta_p \star \Theta_f\}}\ (\text{Frame } \Xi_c)$$

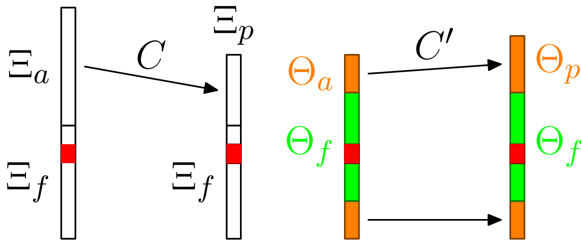
FactorizeFrames's left hand side

$$\dfrac{\dfrac{\{\Xi_a\}C\{\Xi_p\}}{\{\Xi_a \star \Xi_f\}C\{\Xi_p \star \Xi_f\}}\ (\text{Fr}\ \Xi_f) \quad \dfrac{\{\Theta_a\}C'\{\Theta_p\}}{\{\Theta_a \star \Theta_f\}C'\{\Theta_p \star \Theta_f\}}\ (\text{Fr}\ \Theta_f)}{\{\Xi_a \star \Xi_f\}C;\ C'\{\Theta_p \star \Theta_f\}}\ (\text{Seq})$$



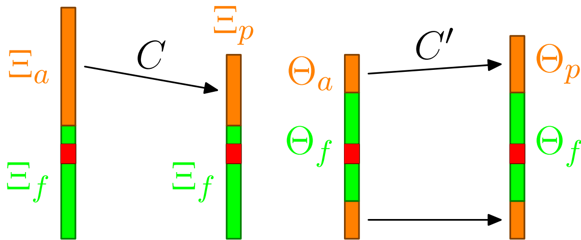
FactorizeFrames's left hand side

$$\dfrac{\dfrac{\{\Xi_a\}C\{\Xi_p\}}{\{\Xi_a \star \Xi_f\}C\{\Xi_p \star \Xi_f\}}\ (\text{Fr } \Xi_f) \quad \dfrac{\{\Theta_a\}C'\{\Theta_p\}}{\{\Theta_a \star \Theta_f\}C'\{\Theta_p \star \Theta_f\}}\ (\text{Fr } \Theta_f)}{\{\Xi_a \star \Xi_f\}C;\, C'\{\Theta_p \star \Theta_f\}}\ (\text{Seq})$$



FactorizeFrames's left hand side

$$\dfrac{\dfrac{\{\Xi_a\}C\{\Xi_p\}}{\{\Xi_a \star \Xi_f\}C\{\Xi_p \star \Xi_f\}}\ (\text{Fr } \Xi_f) \quad \dfrac{\{\Theta_a\}C'\{\Theta_p\}}{\{\Theta_a \star \Theta_f\}C'\{\Theta_p \star \Theta_f\}}\ (\text{Fr } \Theta_f)}{\{\Xi_a \star \Xi_f\}C;\, C'\{\Theta_p \star \Theta_f\}}\ (\text{Seq})$$



- Guard: $\Xi_f \Leftrightarrow \Xi_{f_0} \star \Xi_c$ and $\Theta_f \Leftrightarrow \Theta_{f_0} \star \Xi_c$
- The part of the heap framed twice (in red) is the common frame Ξ_c.

We Want the Frames ♫

Guard: $\Xi_f \Leftrightarrow \Xi_{f0} \star \Xi_c$ and $\Theta_f \Leftrightarrow \Theta_{f0} \star \Xi_c$

$$\dfrac{\dfrac{\{\Xi_a\}C\{\Xi_p\}}{\{\Xi_a \star \Xi_f\}C\{\Xi_p \star \Xi_f\}}\text{(Frame }\Xi_f) \qquad \dfrac{\{\Theta_a\}C'\{\Theta_p\}}{\{\Theta_a \star \Theta_f\}C'\{\Theta_p \star \Theta_f\}}\text{(Frame }\Theta_f)}{\{\Xi_a \star \Xi_f\}C;\, C'\{\Theta_p \star \Theta_f\}}\text{(Seq)}$$

$\downarrow$ FactorizeFrames

$$\dfrac{\dfrac{\dfrac{\{\Xi_a\}C\{\Xi_p\}}{\{\Xi_a \star \Xi_{f0}\}C\{\Xi_p \star \Xi_{f0}\}}\text{(Fr }\Xi_{f0}) \qquad \dfrac{\{\Theta_a\}C'\{\Theta_p\}}{\{\Theta_a \star \Theta_{f0}\}C'\{\Theta_p \star \Theta_{f0}\}}\text{(Fr }\Theta_{f0})}{\{\Xi_a \star \Xi_{f0}\}C;\, C'\{\Theta_p \star \Theta_{f0}\}}\text{(Seq)}}{\{\Xi_a \star \Xi_f\}C;\, C'\{\Theta_p \star \Theta_f\}}\text{(Frame }\Xi_c)$$

↳ The common part of Ξ_f and Θ_f (i.e., Ξ_c) is framed separately.

↳ The new application of (Frame) is on a longer command (C; C') than before.

Example

```
requires tree(x);
ensures  tree(x);
rotate_tree(x;){
  local x1,x2;
  if(x = nil){}
  else{
    x1 := x→l;
    x2 := x→r;
    x→l = x2;
    x→r = x1;
    rotate_tree(x1;);
    rotate_tree(x2;); }}
```

→

```
requires tree(x);
ensures  tree(x);
rotate_tree(x;){
  local x1,x2;
  if(x = nil){}
  else{
    x1 := x→l;
    x2 := x→r;
    (x→l = x2; x→r = x1) ||
    rotate_tree(x1;) ||
    rotate_tree(x2;); }}
```

Example

```
requires tree(x);
ensures  tree(x);
rotate_tree(x;){
  local x1,x2;
  if(x = nil){}
  else{
    x1 := x→l;
    x2 := x→r;
    x→l = x2;
    x→r = x1;
    rotate_tree(x1;);
    rotate_tree(x2;); }}
```

→

```
requires tree(x);
ensures  tree(x);
rotate_tree(x;){
  local x1,x2;
  if(x = nil){}
  else{
    x1 := x→l;
    x2 := x→r;
    (x→l = x2; x→r = x1) ||
    rotate_tree(x1;) ||
    rotate_tree(x2;); }}
```

Implementation:

- The rewrite rules have been implemented in Java+tom.
- tom extends Java to pattern match against tom/user-defined Java objects.
- Each rewrite rule is less than 75 lines of code (i.e. manageable).
- Use of tom's strategies to fine tune optimizations.

List of Optimizations

Optimizations include:

- parallelization (previous slides)
- Improvement of temporal locality (omitted in this talk)
- A generic optimization with 4 concrete applications: (omitted in this talk)
  - Early lock releasing
  - Late lock acquirement
  - Early disposal
  - Late allocation

GenericOptimization

$$\dfrac{\dfrac{\{\Xi_a\}C\{\Xi_p\}}{\{\Xi_a \star \Xi_f\}C\{\Xi_p \star \Xi_f\}}\text{(Frame } \Xi_f\text{)} \quad \dfrac{\{\Theta_a\}C'\{\Theta_p\}}{\{\Theta_a \star \Theta_f\}C'\{\Theta_p \star \Theta_f\}}\text{(Frame } \Theta_f\text{)}}{\{\Xi_a \star \Xi_f\}C;\, C'\{\Xi'\}}\text{(Seq)}$$

$\downarrow$ GenericOptimization

$$\dfrac{\dfrac{\{\Theta_a\}C'\{\Theta_p\}}{\{\Xi_a \star \Theta_a \star \Xi_r\}C'\{\Xi_a \star \Theta_p \star \Xi_r\}}\text{(Fr ...)} \quad \dfrac{\{\Xi_a\}C\{\Xi_p\}}{\{\ldots\}C\{\Xi_p \star \Theta_p \star \Xi_r\}}\text{(Fr ...)}}{\{\Xi_a \star \Xi_f\}C';\, C\{\Xi'\}}\text{(Seq)}$$

Guard: $\Theta_f \Leftrightarrow \Xi_p \star \Xi_r$

- This optimization changes the program order.

↳ The guard requires that C' frames the postcondition of C (Ξ_p).

Locks in Separation Logic

- Each lock guards a part of the heap called the lock's resource invariant.
- Resource invariants are exchanged between locks and threads:
  1. When a lock is acquired, it lends its resource invariant to the acquiring thread.
  2. When a lock is released, it claims back its resource invariant from the releasing thread.

Locks in Separation Logic

- Each lock guards a part of the heap called the lock's resource invariant.
- Resource invariants are exchanged between locks and threads:
  1. When a lock is acquired, it lends its resource invariant to the acquiring thread.
  2. When a lock is released, it claims back its resource invariant from the releasing thread.

Formally:

$$\frac{\Xi \text{ is } x\text{'s resource invariant}}{\{\texttt{emp}\}\texttt{lock}(x)\{\Xi\}} \text{ (Lock)}$$

$$\frac{\Xi \text{ is } x\text{'s resource invariant}}{\{\Xi\}\texttt{unlock}(x)\{\texttt{emp}\}} \text{ (Unlock)}$$

(emp represents the empty heap)

Locks in Separation Logic

- Each lock guards a part of the heap called the lock's resource invariant.
- Resource invariants are exchanged between locks and threads:
  1. When a lock is acquired, it lends its resource invariant to the acquiring thread.
  2. When a lock is released, it claims back its resource invariant from the releasing thread.

Formally:

$$\frac{\Xi \text{ is } x\text{'s resource invariant}}{\{\texttt{emp}\}\texttt{lock}(x)\{\Xi\}}\ (\text{Lock})$$

$$\frac{\Xi \text{ is } x\text{'s resource invariant}}{\{\Xi\}\texttt{unlock}(x)\{\texttt{emp}\}}\ (\text{Unlock})$$

(emp represents the empty heap)

Next slide:

↳ Instantiation of the generic optimization to optimize usage of locks

Instantiating GenericOpt.: EarlyUnlocking

$$\dfrac{\dfrac{\{\Xi_a\}C\{\Xi_p\}}{\{\Xi_a \star \Xi_f\}C\{\Xi_p \star \Xi_f\}}\ (\text{Fr } \Xi_f) \qquad \dfrac{\dfrac{\Theta_a \text{ is } x\text{'s resource invariant}}{\{\Theta_a\}\mathtt{unlock}(x)\{\mathtt{emp}\}}\ (\text{Unlock})}{\{\Theta_a \star \Theta_f\}\mathtt{unlock}(x)\{\Theta_f\}}\ (\text{Fr } \Theta_f)}{\{\Xi_a \star \Xi_f\}C;\ \mathtt{unlock}(x)\{\Theta_f\}}\ (\text{Seq})$$

$\downarrow$ EarlyUnlocking

$$\dfrac{\dfrac{\dfrac{\Theta_a \text{ is } x\text{'s resource invariant}}{\{\Theta_a\}\mathtt{unlock}(x)\{\mathtt{emp}\}}\ (\text{Unlock})}{\{\Xi_a \star \Theta_a \star \Xi_r\}\mathtt{unlock}(x)\{\Xi_a \star \Xi_r\}}\ (\text{Fr} \ldots) \qquad \dfrac{\{\Xi_a\}C\{\Xi_p\}}{\{\Xi_a \star \Xi_r\}C\{\Xi_p \star \Xi_r\}}\ (\text{Fr } \Xi_r)}{\{\Xi_a \star \Xi_f\}\mathtt{unlock}(x);\ C\{\Xi_p \star \Xi_r\}}\ (\text{Seq})$$

Guard: $\Theta_f \Leftrightarrow \Xi_p \star \Xi_r$

- ($\Xi_p \star \Xi_f \Leftrightarrow \Theta_a \star \Theta_f$ + guard) implies $\Xi_f \Leftrightarrow \Theta_a \star \Xi_r$.

↳ Command C does not access x's resource invariant:

- Better unlock x before executing C!

Example (2)

```
requires  x ↦ [val : _];
ensures   emp;
copy_and_dispose(x;){
  local v;
  lock(r_{c ↦ [val:_]});
    v := x → val;
    c → val = v;
    dispose(x);
  unlock(r_{c ↦ [val:_]});
}
```

→

```
requires  x ↦ [val : _];
ensures   emp;
copy_and_dispose(x;){
  local v;
  v := x → val;
  dispose(x);
  lock(r_{c ↦ [val:_]});
    c → val = v;
  unlock(r_{c ↦ [val:_]});
}
```

- r is a lock with resource invariant $c \mapsto [val : _]$, i.e., one cell c with field val.

Optimizations:

- The critical region is shortened.
- Memory is disposed as soon as possible.

TemporalLocality

- temporal locality $\stackrel{\Delta}{=}$ time between two accesses to the same heap cell

↳ the smaller the better (no need to free/load processors's caches)

TemporalLocality

- Intuition below: C and C'' access the same part of the heap

↳ Execute them successively

$$
\dfrac{
\dfrac{\{\Xi\}C\{\Xi'\}}{\{\Xi\star\Theta\}C\{\Xi'\star\Theta\}}\,(\text{Fr }\Theta)
\quad
\dfrac{\{\Theta\}C'\{\Theta'\}}{\{\Xi'\star\Theta\}C'\{\Xi'\star\Theta'\}}\,(\text{Fr }\Xi')
}{\{\Xi\star\Theta\}C;\,C'\{\Xi'\star\Theta'\}}\,(\text{Seq})
\quad
\dfrac{\{\Xi'\}C''\{\Xi''\}}{\{\Xi'\star\Theta'\}C''\{\Xi''\star\Theta'\}}\,(\text{Fr }\Theta')
$$
$$
\overline{\{\Xi\star\Theta\}C;\,C';\,C''\{\Xi''\star\Theta'\}}\,(\text{Seq})
$$

$\downarrow$ TemporalLocality

$$
\dfrac{
\dfrac{\dfrac{\{\Xi\}C\{\Xi'\}\qquad\{\Xi'\}C''\{\Xi''\}}{\{\Xi\}C;\,C''\{\Xi''\}}\,(\text{Seq})}{\{\Xi\star\Theta\}C;\,C''\{\Xi''\star\Theta\}}\,(\text{Fr }\Theta)
\quad
\dfrac{\{\Theta\}C'\{\Theta'\}}{\{\Xi''\star\Theta\}C'\{\Xi''\star\Theta'\}}\,(\text{Fr }\Xi'')
}{\{\Xi\star\Theta\}C;\,C'';\,C'\{\Xi''\star\Theta'\}}\,(\text{Seq})
$$

Instantiating GenericOpt.: LateLocking

$$\dfrac{\dfrac{\dfrac{\Xi_p \text{ is } x\text{'s resource invariant}}{\{\texttt{emp}\}\texttt{lock}(x)\{\Xi_p\}}\ (\text{Lock})}{\{\Xi_f\}\texttt{lock}(x)\{\Xi_p \star \Xi_f\}}\ (\text{Frame } \Xi_f) \qquad \dfrac{\{\Theta_a\}C'\{\Theta_p\}}{\{\Theta_a \star \Theta_f\}C'\{\Theta_p \star \Theta_f\}}\ (\text{Frame } \Theta_f)}{\{\Xi_f\}\texttt{lock}(x);\ C'\{\Theta_p \star \Theta_f\}}\ (\text{Seq})$$

$\downarrow$ LateLocking

$$\dfrac{\dfrac{\{\Theta_a\}C'\{\Theta_p\}}{\{\Theta_a \star \Xi_r\}C'\{\Theta_p \star \Xi_r\}}\ (\text{Frame } \Xi_r) \qquad \dfrac{\dfrac{\Xi_p \text{ is } x\text{'s resource invariant}}{\{\texttt{emp}\}\texttt{lock}(x)\{\Xi_p\}}\ (\text{Lock})}{\{\Theta_p \star \Xi_r\}\texttt{lock}(x)\{\Xi_p \star \Theta_p \star \Xi_r\}}\ (\text{Frame } \Theta_p \star \Xi_r)}{\{\Xi_f\}C';\ \texttt{lock}(x)\{\Xi_p \star \Theta_p \star \Xi_r\}}\ (\text{Seq})$$

Guard: $\Theta_f \Leftrightarrow \Xi_p \star \Xi_r$

↳ The guard means that command C' does not access x's resource invariant:

- Better lock x after executing C'!

Example (3)

```
requires tree(t); ensures emp;
disp_tree(t) {
 local i,j;
 if (t = nil) {} else {
  i := t→l; j := t→r;
  disp_tree(i); disp_tree(j);
  dispose(t);
 }
}
```

→

```
requires tree(t); ensures emp;
disp_tree(t) {
 local i,j;
 if (t = nil) {} else {
  i := t→l; j := t→r;
  dispose(t) ‖
  (disp_tree(i) ‖ disp_tree(j));
 }
}
```