Separation Logic, Abstraction and Inheritance
M. Parkinson, G. Bierman, in Proc. POPL, 2008

Timothée Martiel

Research Topics in Software Engineering
1. From Separation Logic to Inheritance

2. Beyond Separation Logic

3. What About Invariants?
1. From Separation Logic to Inheritance

2. Beyond Separation Logic

3. What About Invariants?
Separation Logic

- Extension of Hoare Logic
- Models heap manipulation
- Local reasoning: separate heap into disjoint parts
- No abstraction (modules, classes, dynamic method binding)

\[
\{ P \} C \{ Q \}
\]

\[
\{ Precondition \} Code \{ Postcondition \}
\]
Specifications

- Points to predicate: \( i \mapsto x \)
- \( \ast \) conjunction: \( i \mapsto x \ast j \mapsto y \)

Program

- Heap allocation: \( \text{cons}(x) \)
- Heap lookup: \( i = [x] \)
- Heap assignment: \( [x] = i \)
- Heap deallocation: \( \text{dispose}(x) \)
Frame Rule

\[
\frac{\{P\} C \{Q\}}{\{P \ast R\} C \{Q \ast R\}}
\]

Provided: free variables of \( R \) are not modified in \( C \)

- Aliasing control
- Local reasoning
Challenges of object-oriented languages:
- Heavy heap usage: object references
- Inheritance and dynamic dispatch

Separation logic
- Is a good framework for heap control
- Needs extension to support inheritance
1 From Separation Logic to Inheritance

2 Beyond Separation Logic

3 What About Invariants?
3 extensions

1. Abstract Predicate Families to abstract data types
2. Static and Dynamic method specifications for static or dynamic method calls
3. Verification rules: method body is verified exactly once
Example: Cell Class Hierarchy

```csharp
Example

class Cell {
    int val;

    public Cell() {}

    public virtual void set(int x)
    { this.val = x; }

    public virtual int get()
    { return this.val; }
}

class ReCell: Cell {
    int back;

    public Cell() {}

    public override void set(int x)
    { this.back = this.Cell::get();
        this.Cell::set(x); }

    public inherit int get();
    public virtual void undo(){...}
}
```
Extension 1: Abstract Predicate Family

- Abstract predicate describe abstract data types
- Class hierarchy gives a family of abstract predicates, one for each class
- Predicates accessible within the class hierarchy, predicate definition accessible within the class

Example

Family $Val(x, v)$:

$Val_{Cell}(x, v) \triangleq x.val \mapsto v$

$Val_{ReCell}(x, v, b) \triangleq Val_{Cell}(x, v) \land x.back \mapsto b$

Note: variable argument numbers are compensated by existential quantifiers
Extension 2: Method Specifications

- Two types of specifications: static ($\{S_C\}_C \{T_C\}$) and dynamic ($\{P_C\}_C \{Q_C\}$), for static and dynamic dispatch

4 elementary verifications

- **Body verification**: $\{S_C\}$ *method body* $\{T_C\}$
- **Dynamic dispatch**: $\{S_C\}_C \{T_C\}$ stronger than $\{P_C\}_C \{Q_C\}$
- **Behavioral subtyping**: with $D <: C$, $\{P_D\}_D \{Q_D\}$ stronger than $\{P_C\}_C \{Q_C\}$
- **Inheritance**: with $D <: C$, $\{S_C\}_C \{T_C\}$ stronger than $\{S_D\}_D \{T_D\}$
Specifications

- Dynamic: \{Val(this, \_)} \_ \{Val(this, x)\}
- Static: \{Val_{\text{Cell}}(this, \_)} \_ \{Val_{\text{Cell}}(this, x)\}

Verification: method implemented in the base class

- Body verification:
  \{Val_{\text{Cell}}(this, \_)} this.val = x; \{Val_{\text{Cell}}(this, x)\}

- Dynamic dispatch:
  \{Val_{\text{Cell}}(this, \_)} \_ \{Val_{\text{Cell}}(this, x)\}
  \Rightarrow \{Val(this, \_)} \_ \{Val(this, x)\}
Extension 3, Verifying Methods: ReCell::set(int x)

Specifications

- Dynamic: \{\textit{Val}(\textit{this}, v, \_)} \_ \{\textit{Val}(\textit{this}, x, v)\}
- Static: \{\textit{Val}_{ReCell}(\textit{this}, v, \_)} \_ \{\textit{Val}_{ReCell}(\textit{this}, x, v)\}

Verification: overridden method

- Behavioral subtyping:
  \{\textit{Val}(\textit{this}, v, \_)} \_ \{\textit{Val}(\textit{this}, x, v)\}
  \Rightarrow \{\textit{Val}(\textit{this}, \_)} \_ \{\textit{Val}(\textit{this}, x)\}

- Dynamic dispatch
- Body verification
Extension 3, Verifying Methods: ReCell::get()

Specifications

- **Dynamic:** \( \{ Val(this, v, o) \} \_ \{ Val(this, v, o) * ret = v \} \)
- **Static:** \( \{ Val_{ReCell}(this, v, o) \} \_ \{ Val_{ReCell}(this, v, o) * ret = v \} \)
- **Static for Cell:** \( \{ Val_{Cell}(this, v) \} \_ \{ Val_{Cell}(this, v) * ret = v \} \)

Verification: inherited (not overridden) method

- **Inheritance:**

  \[
  \{ Val_{Cell}(this, v) \} \_ \{ Val_{Cell}(this, v) * ret = v \} \\
  \Rightarrow \{ Val_{ReCell}(this, v, o) \} \_ \{ Val_{ReCell}(this, v, o) \}
  \]

- Behavioral subtyping
- Dynamic dispatch
1. From Separation Logic to Inheritance

2. Beyond Separation Logic

3. What About Invariants?
Object Invariants

- Invariant: explicit consistency criterion on an object
- When does it hold or not? How does an object tell that to a client?

- Drossopoulou et al., in *ECOOP*, 2008
- Spec#, Barnett et al., in *Proceedings of CASSIS*, 2005
Example

class DCell: Cell {
    public DCell(){}
    public override void set(int x) {
        this.Cell::set(2 * x);
    }
}

• Not a behavioral subtype: “copy-and-paste” inheritance
• Forbidden in invariant-based approaches
• With separation logic:

\[
Val_{DCell}(x, v) \triangleq false \\
DVal(x, v) \triangleq Val_{Cell}(x, v)
\]

works fine: DCell is not a (behavioral) subtype of Cell for the logic.
Conclusion: a Flexible Framework

Framework

- More expressive than most other approaches
- Requires more annotation: this can be automated
- Cannot use first-order SMT solvers
- Has been extended to a Java verifier (jStar, Distefano et al., in *OOPSLA*, 2008)

Article

- Self-contained, no other article required if you know separation logic
- Well explained: formalism, intuition, examples
- Gives an elegant solution in an elegant form
4 Formal Separation Logic Definitions

5 Bibliography
## Separation Logic Definitions: Stack and Heap

### Definition (Stack)

\[ S \triangleq \text{Variables} \rightarrow \text{Values} \]

### Definition (Heap)

\[ H \triangleq \text{Locations} \rightarrow \text{Values} \]

### Definition (Program State)

\[(S, H, I)\]

- \(I\): auxiliary variables stack
Definition (points to)

\[(S, H, I) \models E \leftrightarrow E' \overset{\Delta}{=} \text{dom}(H) = \{[E]_{S,I}\} \]
\[\land H([E]_{S,I}) = [E']_{S,I} \]

Definition (star)

\[(S, H, I) \models P \ast Q \overset{\Delta}{=} \exists H_1, H_2. H_1 \ast H_2 = H \]
\[\land (S, H_1, I) \models P \land (S, H_2, I) \models Q \]
Definition (Frame Rule)

\[
\frac{\vdash \{ P \} C \{ Q \}}{
\vdash \{ P \star R \} C \{ Q \star R \}}
\]

Provided: \( \text{modified}(C) \cap \text{FV}(R) = \emptyset \)
• D. Distefano and M. Parkinson “jStar: towards practical verification for java”, in OOPSLA 2008