Separation Logic, Abstraction and Inheritance

M. Parkinson, G. Bierman, in *Proc. POPL*, 2008

Timothée Martiel

Research Topics in Software Engineering

- Extension of Hoare Logic
- Models heap manipulation
- Local reasoning: separate heap into disjoint parts

- No abstraction (modules, classes, dynamic method binding)

$$\{P\}C\{Q\}$$

$$\{Precondition\}Code\{Postcondition\}$$

Separation Logic: Specification and Program Constructs

Specifications

- Points to predicate: $i \mapsto x$
- $*$ conjunction: $i \mapsto x * j \mapsto y$

Program

- Heap allocation: $\mathrm{cons}(x)$
- Heap lookup: $i = [x]$
- Heap assignment: $[x] = i$
- Heap deallocation: $\mathrm{dispose}(x)$

Frame Rule

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

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

Outline

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

```
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 $Val(x, v)$:

$$\begin{array}{rcl} Val_{Cell}(x, v) & \hat{=} & x.val \mapsto v \\ Val_{ReCell}(x, v, b) & \hat{=} & Val_{Cell}(x, v) \wedge x.back \mapsto b \end{array}$$

Note: variable argument numbers are compensated by existential quantifiers

Extension 2: Method Specifications

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

4 elementary verifications

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

Extension 3, Verifying Methods: `Cell::set(int x)`

Specifications

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

Verification: method implemented in the base class

- Body verification:

$$\{Val_{Cell}(this, \_)\} this.val = x; \{Val_{Cell}(this, x)\}$$

- Dynamic dispatch:

$$\{Val_{Cell}(this, \_)\}\_\{Val_{Cell}(this, x)\}$$
$$\Rightarrow \{Val(this, \_)\}\_\{Val(this, x)\}$$

Extension 3, Verifying Methods: `ReCell::set(int x)`

Specifications

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

Verification: overridden method

- Behavioral subtyping:

$$\{Val(this, v, \_)\}\_\{Val(this, x, v)\}$$
$$\Rightarrow \{Val(this, \_)\}\_\{Val(this, x)\}$$

- 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

- 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:

$$Val_{DCell}(x, v) \hat{=} false$$
$$DVal(x, v) \hat{=} Val_{Cell}(x, v)$$

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

Separation Logic Definitions: Stack and Heap

Definition (Stack)

$$S \hat{=} Variables \rightharpoonup Values$$

Definition (Heap)

$$H \hat{=} Locations \rightharpoonup Values$$

Definition (Program State)

$$(S, H, I)$$

- I: auxiliary variables stack

Separation Logic Definitions: Specifications

Definition (points to)

$$
\begin{aligned}
(S, H, I) \models E \mapsto E' \quad &\hat{=} \quad dom(H) = \{[E]_{S,I}\} \\
&\quad\ \wedge H([E]_{S,I}) = [E']_{S,I}
\end{aligned}
$$

Definition (star)

$$
\begin{aligned}
(S, H, I) \models P * Q \quad &\hat{=} \quad \exists H_1, H_2 . H_1 * H_2 = H \\
&\quad\ \wedge (S, H_1, I) \models P \wedge (S, H_2, I) \models Q
\end{aligned}
$$

Definition (Frame Rule)

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

Provided: $\mathrm{modified}(C) \cap \mathrm{FV}(R) = \emptyset$

- M. Barnett, K. R. M. Leino and W. Schulte. "The Spec# Programming System: An Overview". In *Proceedings of CASSIS*, 2005
- D. Distefano and M. Parkinson "jStar: towards practical verification for java", in *OOPSLA* 2008
- S. Drossopoulou, A. Francalanza, P. Müller and A. J. Summers. "A Unified Framework for Verification Techniques for Object Invariants". In *ECOOP* 2008