An Abstract Interpretation Framework for Refactoring

P. Cousot, NYU, ENS, CNRS, INRIA
R. Cousot, ENS, CNRS, INRIA
F. Logozzo, M. Barnett, Microsoft Research

The problem

Refactoring is a very common programmer activity. Useful to maintain the code, avoid code bloats, etc. Examples: rename, re-order parameters, extract method, etc.

IDEs guarantee that the refactored program is:
1. a syntactically valid program
2. a semantically equivalent program

There is no guarantee about the
1. Preservation of the correctness proof
2. Interaction with the static analysis

Example: extract method

```java
public int Decrement(int x)
{
    Contract.Requires(x >= 5);
    Contract.Ensures(Contract.Result<int>() >= 0);
    while (x != 0) x--;
    return x;
}
```

```
private static int NewMethod(int x)
{
    while (x != 0) x--;
    return x;
}
```

and the (modular) proof?

```java
public int Decrement(int x)
{
    Contract.Requires(x >= 5);
    Contract.Ensures(Contract.Result<int>() >= 0);
    while (x != 0) x--;
    return x;
}
```

```
private static int NewMethod(int x)
{
    while (x != 0) x--;
    return x;
}
```
Simple solutions?

Method inlining: the reverse of extract method
May not scale up, how many levels should we inline?
Isolated analysis: infer pre- and postconditions of the extracted method
Too imprecise, without the context inferred contracts may be too generic
Invariant projection: project the pre/post-states on the parameters and return value
Too specific, cannot refactor unreached code
User assistance: User provides the contracts
Impractical, too many contracts to write
State of the art (before this paper:)

Extract method with contracts:
Requirements

Validity
The inferred contract should be valid
Counterexample:

```
public int Decrement(int x)
{
 Contract.Requires(x >= 5);
 Contract.Ensure
```
Safety

The precondition of the extracted method should advertise possible errors

Counterexample:

```java
public int Decrement(int x)
{
    Contract.Requires(x >= 5);
    Contract.Ensure(Contract.Result<int>() >= 0);
    x = NewMethod(x);
    return x;
}
```

```java
private static int NewMethod(int x)
{
    Contract.Requires(x >= 5);
    Contract.Ensure(Contract.Result<int>() <= 0);
    while (x != 0) x--;
    return x;
}
```

Generality

The inferred contract is the most general satisfying Validity, Safety, and Completeness

Counterexample: Valid, Safe, Complete but not General contract

```java
public int Decrement(int x)
{
    Contract.Requires(x >= 5);
    Contract.Ensure(Contract.Result<int>() >= 0);
    x = NewMethod(x);
    return x;
}
```

```java
private static int NewMethod(int x)
{
    Contract.Requires(x >= 0);
    Contract.Ensure(Contract.Result<int>() == 0);
    while (x != 0) x--;
    return x;
}
```

Completeness

The verification of the callee should still go through

Counterexample: Valid and safe contract, but not complete

```java
public int Decrement(int x)
{
    Contract.Requires(x >= 5);
    Contract.Ensure(Contract.Result<int>() >= 0);
    x = NewMethod(x);
    return x;
}
```

```java
private static int NewMethod(int x)
{
    Contract.Requires(x >= 0);
    Contract.Ensure(Contract.Result<int>() == 0);
    while (x != 0) x--;
    return x;
}
```

Our solution

Valid, Safe, Complete, and General contract

```java
public int Decrement(int x)
{
    Contract.Requires(x >= 5);
    Contract.Ensure(Contract.Result<int>() >= 0);
    x = NewMethod(x);
    return x;
}
```

```java
private static int NewMethod(int x)
{
    Contract.Requires(x >= 0);
    Contract.Ensure(Contract.Result<int>() == 0);
    while (x != 0) x--;
    return x;
}
```
Formalization

Algebraic Hoare Logic

We need to formalize what a static analyzer does, in particular method calls. Hoare Logic is the natural candidate. However, it is already an abstraction of the concrete semantics. We define a concrete Hoare logic where predicates are replaced by sets:

\[ \{ P \} S \{ Q \} \quad P \in \mathcal{P}(\Sigma) \text{ and } Q \in \mathcal{P}(\Sigma \times \Sigma) \]

The deduction rules are as usual. Details in the paper.

Orders on contracts

Covariant order \( \Rightarrow \)
Intuition: a stronger precondition is better for the callee
\[ P, Q \Rightarrow P', Q' \text{ iff } P \subseteq P' \text{ and } Q \subseteq Q' \]

Contravariant order \( \rightarrow \)
Intuition: a \( \rightarrow \)-stronger contract is more general (better for the caller)
\[ P, Q \rightarrow P', Q' \text{ iff } P' \subseteq P \text{ and } Q \subseteq Q' \]

Note: formal (and more correct) definition in the paper.

Some notation...

\( m \) is the refactored (extracted) method
\( S \) denotes the selected code (to be extracted)
It is the body of the extracted method \( m \)
\( P_m, Q_m \) is the most precise safety contract for a method \( m \)
See Cousot, Cousot & Logozzo VMCAI'11
\( P_s, Q_s \) is the projection of the abstract state before the selection, \( P_s \)
\( Q_s \) after the selection, \( Q_s \)
**Extract method with contracts problem**

The refactored contract $P^r, Q^r$ is a solution to the problem if it satisfies

- **Validity**
  
  \[ \{ P^r \} S \{ Q^r \} \]

- **Safety**
  
  $P^r, Q^r \Rightarrow P_m, Q_m$

- **Completeness**
  
  \[ \{ P_m \} m(\_\_) \{ Q_m \} \]

- **Generality**
  
  $\forall P'^r, Q'^r$ satisfying validity, safety, and completeness: $P^r, Q^r \Rightarrow P'^r, Q'^r$

Theorem: The 4 requirements above are mutually independent

---

**Iterative Solution**

Idea: give an iterative characterization of the declarative solution

- It is easier to abstract and compensates for the lose of precision

**Theorem:** Define

\[ F[S] \langle X, Y \rangle \equiv \langle P_m \cap \text{pre}[S] Y, Q_m \cap \text{post}[S] X \rangle \]

Then

\[ P^r, Q^r \Rightarrow \{ P_m \} S \{ \text{post}[S] P_m \} = \text{gfp}_{P_m, Q_m} F[S] \]

The order for the greatest fixpoint computation is $\rightarrow$

**Intuition:** generalize the contract at each iteration step

---

**Declarative Solution**

Theorem: There exists a unique solution for the problem:

$P^r, Q^r = \{ P_m \} S \{ \text{post}[S] P_m \}$

**Drawback:** It is not a feasible solution

- $P_m$ and $\text{post}[\_\_]$ are not computable (only for trivial cases of finite domains)
- We need to perform some abstraction to make it tractable
- The formulation above is ill-suited for abstraction

---

**Abstraction**
Abstract Hoare triples

Given abstract domains $A$ approximating $\mathcal{P}(\Sigma)$ and $B$ approximating $\mathcal{P}(\Sigma \times \Sigma)$
Define abstract Hoare triples
$$\{P\} S \{Q\} \iff \{\gamma_A(P)\} S \{\gamma_B(Q)\}$$
Idea: replace the concrete set operations with the abstract counterparts
Abstract Hoare triples generalize usual Hoare logic
Example: Fix $A, B$ to be first order logic predicates
Question: Are the usual rules of Hoare logic valid in the general case?

We are in trouble?

A similar result holds for the disjunction rule $\oplus$
We need some hypotheses on the abstract domains and the concretizations $\gamma$
Theorem: The abstract Hoare triples without the conjunction and disjunction are sound
But we need conjunction to model method call, product of analyses, etc.!
Theorem: If $\gamma_B$ is finite-meet preserving the conjunction rule is sound
A dual result holds for $\gamma_A$ and the disjunction rule

Details on the paper: formalization and some extra technical details

Counterexample: conjunction rule

$$\{x \geq 0\} x = -x \{x \leq 0\} \text{ and } \{x \leq 0\} x = -x \{x \geq 0\}$$

But
$$\{x \geq 0 \cap x \leq 0\} x = -x \{x \leq 0 \cap x \geq 0\}$$
$$\{x = 0\} x = -x \{\text{false}\}$$

And now?

We can define the problem of the extract method with contracts in the abstract
Define abstract contracts, the rule for abstract method call, etc.
Theorem: The abstract counterparts for validity, safety, and completeness are sound
However, abstraction introduces new problems
It is impossible to have a complete abstract refactoring in general
It did not manifest in our experiments
The iterated gfp computation balances for the loss of information
Details in the paper (or come to see me after the talk!)
Experiments

Implementation

We use the CodeContracts static checker (aka Clousot) as underlying static analyzer
Based on abstract interpretation
More then 75K downloads, widely used in industrial environments
We use the Roslyn CTP for C# language services and basic engine refactoring
Industrial strength C# compiler and services implementation
Integrates in Visual Studio

Inference Algorithm

Use the Roslyn refactoring service to detect the extracted method m
Use Clousot to infer $P_s$, $Q_s$
Project the entry state on the beginning of the selection($P_s$). Similarly for $Q_s$
Annotate the extracted method with $P_s$, $Q_s$
Use Clousot to infer $P_{inv}$, $Q_{inv}$
Add $P_{inv}$, $Q_{inv}$ to the extracted method and start the gfp computation
Weaken the precondition, strengthen the postcondition
Do not go below $P_s$, $Q_s$

Results

<table>
<thead>
<tr>
<th>Test</th>
<th>Extraction</th>
<th>Step 1</th>
<th>Step 2/3</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decrement</td>
<td>0.18</td>
<td>0.10</td>
<td>0.12</td>
<td>0.42</td>
</tr>
<tr>
<td>Generaliz</td>
<td>0.20</td>
<td>0.09</td>
<td>0.14</td>
<td>0.45</td>
</tr>
<tr>
<td>BinarySearch</td>
<td>0.23</td>
<td>0.14</td>
<td>0.32</td>
<td>0.70</td>
</tr>
<tr>
<td>Abv</td>
<td>0.23</td>
<td>0.07</td>
<td>0.12</td>
<td>0.43</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>0.20</td>
<td>0.07</td>
<td>0.28</td>
<td>0.56</td>
</tr>
<tr>
<td>Rcu</td>
<td>0.20</td>
<td>0.09</td>
<td>0.20</td>
<td>0.49</td>
</tr>
<tr>
<td>Gued</td>
<td>0.17</td>
<td>0.07</td>
<td>0.14</td>
<td>0.40</td>
</tr>
<tr>
<td>Loop</td>
<td>0.18</td>
<td>0.07</td>
<td>0.10</td>
<td>0.37</td>
</tr>
<tr>
<td>Exp</td>
<td>0.34</td>
<td>0.18</td>
<td>0.24</td>
<td>0.79</td>
</tr>
<tr>
<td>Man</td>
<td>0.20</td>
<td>0.14</td>
<td>0.20</td>
<td>0.56</td>
</tr>
<tr>
<td>Karr</td>
<td>0.35</td>
<td>0.09</td>
<td>0.14</td>
<td>0.71</td>
</tr>
<tr>
<td>Loop-2</td>
<td>0.28</td>
<td>0.18</td>
<td>1.09</td>
<td>2.43</td>
</tr>
<tr>
<td>Loop-3</td>
<td>0.21</td>
<td>0.10</td>
<td>0.14</td>
<td>0.46</td>
</tr>
<tr>
<td>Seek[4][40]</td>
<td>0.24</td>
<td>0.09</td>
<td>0.00</td>
<td>0.35</td>
</tr>
<tr>
<td>McMillan[13]</td>
<td>0.24</td>
<td>0.18</td>
<td>0.43</td>
<td>0.93</td>
</tr>
<tr>
<td>Bry[9][5]</td>
<td>0.34</td>
<td>0.18</td>
<td>0.28</td>
<td>0.82</td>
</tr>
<tr>
<td>Perotilu[26]</td>
<td>0.47</td>
<td>0.33</td>
<td>0.31</td>
<td>1.13</td>
</tr>
</tbody>
</table>
Conclusions

Have an abstract interpretation framework to define proof-preserving refactorings
  - En passant, generalized Hoare logic
  - Found counterintuitive examples
Instantiated to the problem of refactoring with contracts
  - In the concrete: One solution, two formulations
  - In the abstract: Completeness and generality only under some conditions
Implementation on the top of industrial strength tools
Come see our demo!!!