Practical reasoning about invocations and implementations of pure methods
Ádám Darvas and K. Rustan M. Leino

Pascal Zimmermann

27.05.2013

Motivation

```
class Job{
  List<Job> predecessors;
  int producedOutput, totalOutput;
  void Start()
    requires forall j in predecessors
      j.IsFinished();
  {...}
  bool IsFinished()
  {return producedOutput == totalOutput; }
}
```

Motivation

Methods are a means of abstraction and reuse.

Using methods in specifications seems natural. However, specifications must not have side effects.

Motivation

Methods are a means of abstraction and reuse.

Using methods in specifications seems natural. However, specifications must not have side effects.

Definition

A method m is called pure if

- execution of m does not change the visible state
- the return value of m is the same for all executions in the same state

Contributions

- Axiomatization of pure methods as mathematical functions
- Preconditions and frame conditions for pure methods
- Implementation in the Spec# programming system

Encoding of pure methods as mathematical functions

```
[Pure] int Exmpl(int x)
  requires this.f <= x;
  ensures result <= x - this.f;
{return (x - this.f) / 2; }
```

$$\#\mathsf{Exmpl} : \mathit{ref} \times \mathit{int} \times \mathit{heap} \rightarrow \mathit{int}$$

$$(\forall t, x, h | \underbrace{h[t, f]}_{\text{'t.f'}} \leq x \Rightarrow \#\mathsf{Exmpl}(t, x, h) \leq x - h[t, f])$$

Unsound axioms

```
[pure] int Unsound()
  requires this == this;
  ensures result != result;
{return 1; }
```

$\#\text{Unsound} : \mathit{ref} \times \mathit{heap} \rightarrow \mathit{int}$

$(\forall t, h | t = t \Rightarrow \#\text{Unsound}(t, h) \neq \#\text{Unsound}(t, h))$

Unsound axioms

```
[pure] int Unsound()
  requires this == this;
  ensures result != result;
{return 1; }
```

$$\#\text{Unsound} : ref \times heap \rightarrow int$$

$$(\forall t, h | t = t \Rightarrow (\exists w | w \neq w))$$
$$\Rightarrow (\forall t, h | t = t \Rightarrow \#\text{Unsound}(t, h) \neq \#\text{Unsound}(t, h))$$

Problems with newly allocated objects

```
[pure] O Alloc()
  ...
{return new O(); }

int Foo()
  ensures Alloc() == Alloc();
{...}
```

$$\#\text{Alloc}(t, h) = \#\text{Alloc}(t, h) \Leftrightarrow \mathit{true}$$

Problems with newly allocated objects

The simple solution of disallowing pure methods to return newly allocated objects is too restrictive.
Consider a pure function that returns a tuple.

Solution: Annotations

- ResultNotNewlyAllocated
- NoReferenceComparison

Default: No annotation/restriction

1. Only one reference comparison operand may be allocating
2. At most one parameter to a method call without NoReferenceComparison annotation may be allocating

Boogie methodology

Definition

If an object is **consistent** its invariant must hold. If an object is **exposed** its invariant may temporarily be broken.

Definition

An object is **committed** if its owner is consistent.

Precondition and frame condition for general methods

Precondition

requires `this` is consistent and not committed

Frame condition

$$(\forall o, f \,|\, h[o, f] = \mathbf{old}(h)[o, f] \vee (o, f) \in \mathbf{old}(W) \vee \neg \mathbf{old}(h)[o, \mathit{allocated}] \vee (o \text{ is committed in } \mathbf{old}(h)))$$

for a modifies clause W

Pure methods cannot establish the precondition

```
class Tree{
  int value;
  [Rep] Tree[] children;
  [Pure] int Sum()
    requires this is consistent and not committed;
  { int result = value;
    for(int i=0; i<children.length; i++)
      {result += children[i].Sum(); }
    return result; }
}
```

Relaxed precondition for pure methods

- Precondition for general methods requires the receiver to be not committed.

- A pure method could not call a pure method on rep objects because it cannot expose the receiver.

Solution: Let the precondition make no statement about whether or not the object is committed.

Frame condition for pure methods

$$(\forall o, f \,|\, h[o, f] = \mathbf{old}(h)[o, f] \vee (o, f) \in \mathbf{old}(W) \vee \\ \neg\mathbf{old}(h)[o, \mathit{allocated}] \vee (o \text{ is committed in } \mathbf{old}(h)))$$

```
[Pure] int M()
  requires this is consistent;
{int a = this.x; int b = N();
 assert a == this.x; return b-a; }
[Pure] int N()
  requires this is consistent;
{return 5; }
```

Cannot verify assertion. `N()` is seen as having arbitrary effect on committed objects.

Effects on committed objects

The methodology enforces the owner to be exposed before exposing and modifying an object.

Record a *snapshot* of an objects ownership cone everytime the object transitions from exposed to consistent by adding a field snapshot.

o.snapshot = (o.x,o.y,...,o.r.snapshot,o.s.snapshot,...)
where x,y are non-snapshot fields of o and r,s are rep objects.

Effects on committed objects

Encode FirstNonCommittedOwner (fnco) using a field.
fnco is allocated and not committed.

$$(\forall o, h | h[o, \mathit{allocated}] \wedge (o \text{ is committed in } h) \Rightarrow$$
$$h[o, \mathit{fnco}] \neq \textbf{null} \wedge h[h[o, \mathit{fnco}], \mathit{allocated}] \wedge$$
$$(h[o, \mathit{fnco}] \text{ not committed in } h))$$

Effects on committed objects

Requirements:

- A change to a field of a committed object requires change in the snapshot of the object's fnco. (axiom)
- o.fnco can only change if o.fnco.snapshot does. (postcondition)

Conclusion

- Using pure methods in specifications makes the specifications more readable and more robust to code changes.
- To not break information hiding pure methods may be needed to not expose implementation specific fields.
- Encoding pure methods in the verifier is needed to be able to make use of pure methods.

The paper

- Self contained. Builds on previous work but shortly explains prerequisites.
- Not all explanations clear.
- They could have used more or improved examples.

Related work

2007: Á.Darvas and K.R.M. Leino, Practical reasoning about invocations and implementations of pure methods.

- 2005: D.R.Cok, Reasoning with specifications containing method calls and model fields.
- 2005: Á.Darvas and P.Müller, Reasoning about method calls in JML specifications
- 2006: Á.Darvas and P.Müller, Reasoning about method calls in interface specifications.
- 2008: K.R.M.Leino and P.Müller, Verification of equivalent-result methods.
- 2009: K.R.M.Leino and R.Middelkoop, Proving consistency of pure methods and model fields.

Questions?

Finding a witness

```
[pure] int Unsound()
  requires this == this;
  ensures result != result;
{return 1; }
```

$\#\text{Unsound} : \mathit{ref} \times \mathit{heap} \rightarrow \mathit{int}$

$$(\forall t, h | t = t \Rightarrow (\exists w | w \neq w))$$
$$\Rightarrow (\forall t, h | t = t \Rightarrow \#\text{Unsound}(t, h) \neq \#\text{Unsound}(t, h))$$

SMT-Solver fails to find w even for simple examples.

Syntactic approximation

If a pure method has exactly one postcondition of the form

result *op* E
with $op \in \{=, <, >, \leq, \geq\}$,
and **result** $\notin$ E,

then w exists.

Types of pure methods

Distinction between three types of pure methods:

- state-independent pure methods
- read-confined pure methods that read only the heap locations in the ownership cone of their receiver
- read-anything pure methods

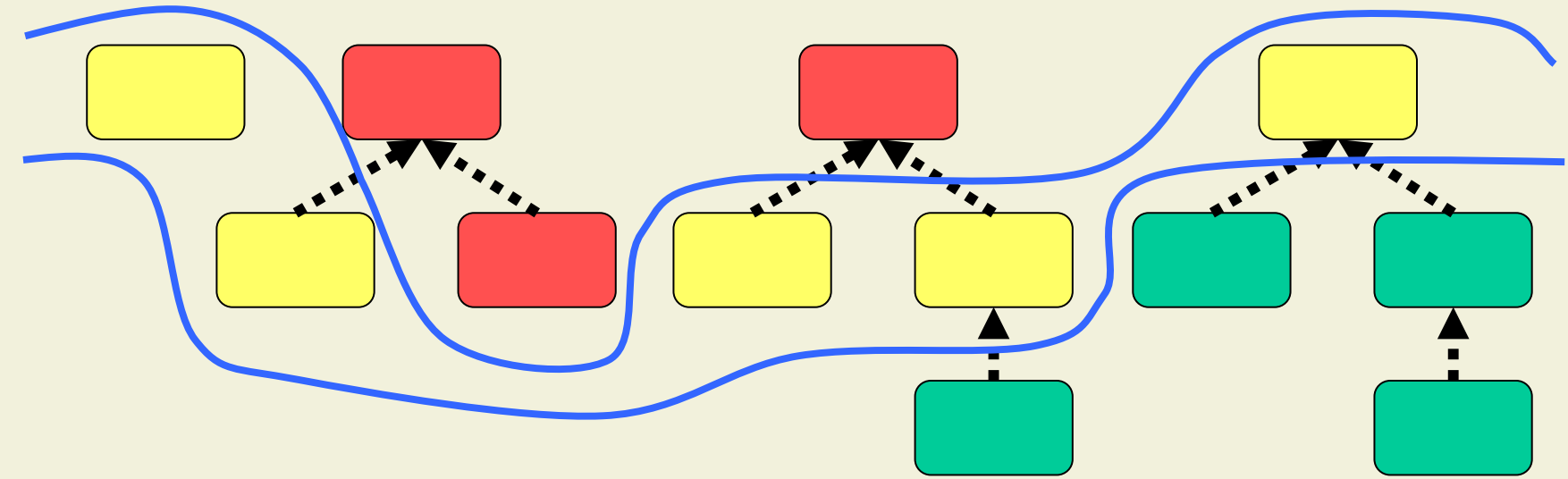
Modifications in the encoding

If M is a state-independent pure method, the receiver of M is not an input of #M

Provide an axiom that states that if a read-confined pure method is called on two different object stores where the heap locations of the ownership cone are the same, then the two executions return the same result.

Use snapshot to generate axiom

$$(\forall o, p, h, k | (o \text{ is consistent in } h) \wedge (o \text{ is consistent in } k) \wedge h[o, snapshot] = k[o, snapshot] \Rightarrow \#M(o, p, h) = \#M(o, p, k))$$



mutable

valid, mutable owner

valid, valid owner