Lecture 4

Towards a Verifying Compiler: Data Abstraction

Wolfram Schulte
Microsoft Research
Formal Methods 2006

Purity, Model fields, Inconsistency

Joint work with Rustan Leino, Mike Barnett, Manuel Fähndrich, Herman Venter, Rob DeLine, Wolfram Schulte (all MSR), and Peter Müller (ETH), Bart Jacobs (KU Leuven) and Bor-Yuh Evan Chung (Berkley) .

Slides based on a presentation of Peter Müller given at MSR 5/2006

Review: Verification of OO Programs with Invariants

- What were the 2 major tricks to support invariants?
- Which programs can we verify?
- What are the limitations?

Data Abstraction using Methods

Needed for

- Subtyping
- Information hiding

```
interface Shape {
 pure int Width( );
 void DoubleWidth( )
  ensures Width( ) == old( Width( ) ) * 2;
}
```

```
class Rectangle: Shape {
 int x1; y1; x2; y2;
 pure int Width( )
   private ensures result == x2 – x1;  { … }
 void DoubleWidth( )
   ensures Width( ) == old( Width( ) ) * 2;
 { … }
}
```

Encoding of Pure Methods

- Pure methods are encoded as functions

$$M\text{: Value} \times \text{Value} \times \text{Heap} \rightarrow \text{Value}$$

- Functions are axiomatized based on specifications

$$\forall \text{this,par,Heap:}$$
$$\quad \text{Requires}_M(\text{ this,par,Heap }) \Rightarrow$$
$$\qquad \text{Ensures}_M(\text{this,par,Heap}) [\ M(\text{this,par,Heap })\ /\ \textbf{result}\ ]$$

Problem 1: Inconsistent Specifications

- Flawed specifications potentially lead to *inconsistent axioms*

- How to guarantee consistency?

```
class Inconsistent {
  pure int Wrong( )
    ensures result == 1;
    ensures result == 0;
  { … }
}
```

```
class List {
  List next;
  pure int Len( )
    ensures result == Len( ) + 1;
  { … }
  … }
```

Problem 2: Weak Purity

- Weak purity can be observed through reference equality
- How to prevent tests for reference equality?

```
class C {
  pure C Alloc( )
    ensures fresh( result );
  { return new C( ); }

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

$$Alloc(\text{ this},\mathbf{H}\ ) = Alloc(\text{ this},\mathbf{H}\ )$$

Problem 3: Frame Properties

- Result of pure methods depends on the heap

Has(list, o, **H**)

- How to relate invocations that refer to *different heaps?*

```
class List {
  pure bool Has( object o ) { … }
  void Remove( object o )
    requires Has( o );
  { … }
… }
```

```
void Foo( List list, object o )
  requires list.Has( o );
{
  log.Log( “Message“ );
  list.Remove( o );
}
```

Data Abstraction using Model Fields

- Specification-only fields
- Value is determined by a mapping from concrete state
- Similar to parameterless pure methods

```
interface Shape {
 model int width;
 void DoubleWidth( )
  ensures width == old( width ) * 2;
}
```

```
class Rectangle implements Shape {
 int x1; y1; x2; y2;
 model int width | width == x2 – x1;
 void DoubleWidth( )
  ensures width == old( width ) * 2;
 { … }  }
```

Variant of Problem 3: Frame Properties

```
class Legend {
   Rectangle box;  int font;
   model int mc | mc==box.width / font;
  … }
```

```
class Rectangle {
  model int width | width == x2 – x1;
  void DoubleWidth( )
    modifies x2, width;
    ensures width = old( width ) * 2;
    { x2 := (x2 - x1) * 2 + x1; }
  }
```

- Assignment might change model fields of client objects
- Analogous problem for subtypes
- How to synchronize values of model fields with concrete fields?

Validity Principle

```
class List {
  List next;
  invariant list is acyclic;
  model int len | len == (next == null) ? 1 : next.len + 1;
  … }
```

- Only model fields of *valid objects* have to satisfy their constraints

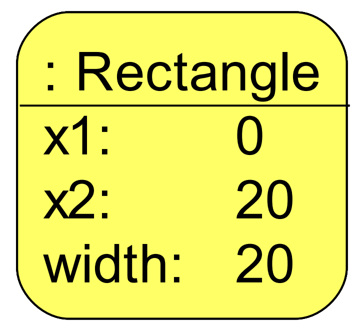
$$\forall X, m: \; X.inv = valid \Rightarrow R_m(X, X.m)$$

- *Avoids inconsistencies* due to invalid objects

Decoupling Principle

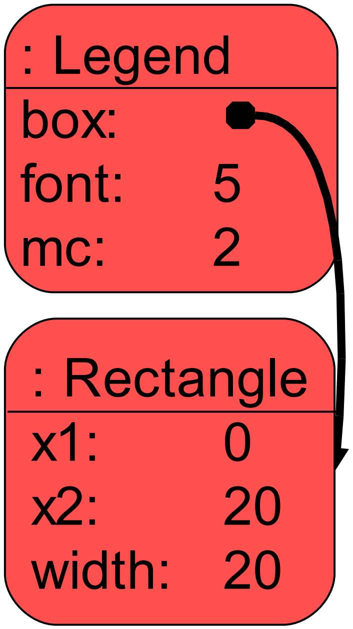
- Decoupling: Model fields are *not updated instantly* when dependee fields are modified
  - Values of model fields are *stored in the heap*
  - *Updated when* object is being *packed*

```
class Rectangle {
  model int width | width == x2 – x1;
  void DoubleWidth( ) requires inv==valid; {
    unpack this;
      x2 := (x2 – x1) * 2 + x1;
    pack this;
  }
```



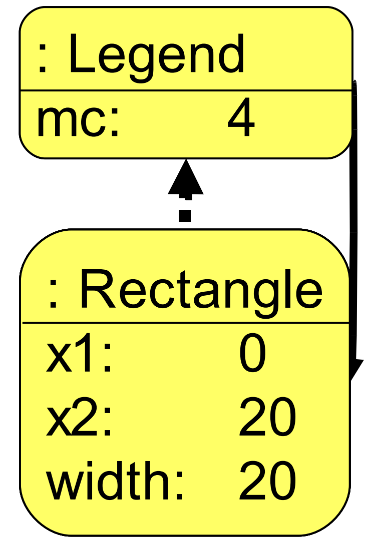
Mutable Dependent Principle

- Mutable Dependent: If a model field *o.m* depends on a field *x.f*, then *o must be mutable whenever x is mutable*



The Methodology in Action

```
class Rectangle {
 void DoubleWidth( )
  requires inv == valid &&
                 owner.inv == mutable;
  modifies width, x2;
 {
  expose(this) {
     x2 := (x2 – x1) * 2 + x1;
  }
 }
```




```
class Legend {
 rep Rectangle box;
 model int mc |
     mc == box.width / font;
 … }
```

Automatic Updates of Model Fields

pack X ≡
    assert $X \neq$ **null** $\wedge$ X.inv = mutable;
    assert Inv(X);
    assert $\forall$p: p.owner = X $\Rightarrow$ p.inv = valid; …
    X.inv := valid;
    foreach m of X:
        assert $\exists$r: R_m(X, r);
        X.m := **choose** r **such that** R_m(X, r);
    end

Soundness

- Theorem:

$$\forall X,m:\ \ X.inv = valid \Rightarrow R_m(\ X,\ X.m\ )$$

- Proof sketch
  - Object creation new:
    - new object is initially mutable
  - Field update X.f := E;
    - Model fields of X: asserts X.inv = mutable
    - Model fields of X's owners: mutable dependent principle
  - unpack X:
    - changes X.inv to mutable
  - pack X:
    - updates model fields of X

Problem 1 Revisited: Inconsistent Specifications

- Witness requirement for non-recursive specifications
- Ownership for traversal of object structures
- Termination measures for recursive specs

```
pure int Wrong( )
  ensures result == 1;
  ensures result == 0;
```

```
pure int Len( )
  ensures result == Len( ) + 1;
  measured_by height( this );
```

```
pure static int Fac( int n )
  requires n >= 0;
  ensures result ==
    ( n==0 ) ? 1 : Fac( n-1 ) * n;
  measured_by n;
```

Problem 2 Revisited: Restricted Weak Purity

- Pure methods must not return references to new objects (Compile time effect analysis)

```
pure C Alloc( )
{ return new C( ); }
```

- Provide value types for sets, sequences, etc.

Problem 3 Revisited: Frame Properties

- Model field solution does not work for methods with parameters
- Caching of values not possible for runtime checking
- Mutable dependent principle too strict

```
class List {
  pure bool Has( object o )
  { … }
  void Remove( object o )
    requires Has( o );
  { … }
… }
```

```
void Foo( List list, object o )
  requires list.Has( o );
{
  log.Log( “Message“ );
  list.Remove( o );
}
```

Summary

- Data abstraction is crucial to express functional correctness properties
- Verification methodology for model fields
  - Supports subtyping
  - Is modular and sound
  - *Key insight: model fields are reduced to ordinary fields with automatic updates*
- Verification methodology for methods (not yet ready)
  - Partial solution: encoding, weak purity, consistency
  - Future work: frame properties based on effects