Lecture 2

Towards a Verifying Compiler: Logic of Object oriented Programs

Wolfram Schulte
Microsoft Research
Formal Methods 2006

Objects, references, heaps,
Subtyping and dynamic binding,
Pre- and postconditions, method framing

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)
Review: Boogie PL

Source language
(eg. Spec#)

Translate source language features
using particular programming methodology

Intermediate language for verification

BoogiePL

Translate Boogie PL code using
particular VC generation

Formulas
Review Boogie PL

• What components does Boogie PL have, and what does it not have?

• What is the purpose of assert, assume and havoc?

• What’s the meaning of a procedure and its modifies clause?

• What do we need to translate an OO language into Boogie PL?
Mapping Spec# to BoogiePL

• Axiomatizing Spec#’s class and field declarations
• The storage model
• Translating methods and code
• Method framing (simplified)
• Loop framing
Axiomatizing the Spec# Type System

On notation:

We use the following C# class

```csharp
class C : object {
    object f = null;
    C(){
    }
}
```

to describe the result of the axiomatization.

We use the function

```
Tr (anslate)
```

to translate Spec# statements into BoogiePL
Axiomatizing the Spec# Type System

Introduce a typename for each Spec# type

```csharp
type C : name;
```

Assert subtyping relationship for program types

```csharp
axiom C <: System.Object;
```

by using a predefined partial order operation `<:`
Axiomatizing C#' Type Declarations

Introduce field names as constants

```csharp
const C.f : name;
```

Assert field properties (kind, type etc).

```csharp
axiom IsRefField(C.f, System.Object);
```

by using the appropriate functions

```csharp
function IsRefField(field:name, type:name) returns bool
```
Storage Model

Use Boogie’s type \texttt{ref} to denote runtime object references.

A Heap maps object references and field names to values:

\begin{verbatim}
var Heap: [ref, name] any;  // Heap : ref ×name →any
\end{verbatim}

Allocatedness is represented as another field of the heap:

\begin{verbatim}
const allocated: name;
\end{verbatim}

Access to an instance field $f$ declared in $C$ is translated as:

\begin{verbatim}
assert o ≠ null; x := Heap[ o, C.f ]
\end{verbatim}

\begin{verbatim}
assert o ≠ null; Heap[ o, C.f ] := x
\end{verbatim}

\begin{verbatim}
Tr[[ x = o.f; ]] = assert o ≠ null ; Heap[ o, C.f ] := x
\end{verbatim}

\begin{verbatim}
Tr[[ o.f = x; ]] =
\end{verbatim}
Allocation

\[
\text{Tr}[x = \text{new } T()] = \\
\{ \text{var } o: \text{ref}; \\
\text{assume } o \neq \text{null } \land \text{typeof}(o) = T; \\
\text{assume } \text{Heap}[o, \text{allocated}] = \text{false}; \\
\text{Heap}[o, \text{allocated}] := \text{true}; \\
\text{call } T..\text{ctor}(o); \} 
\]
Methods

Recall: Boogie PL

• has only procedures, no instance methods
  → Add *this* as first parameter to generated proc

• is weakly typed (just int, bool, ref)
  → Spec# types must be preserved via contracts

• has no idea of heap properties
  → Allocatedness must be preserved via contracts

• has no inheritance
  → Strengthening of postconditions must be implemented via multiple procedures
Constructors and Non-Virtual Methods

\[ \text{Tr } [\text{C}()] \] =

\begin{verbatim}
proc C..ctor(this: ref);
    requires this != null \land typeof(this) <: C;
    modifies Heap;

impl C..ctor(this: ref)
{
    assume Heap[this, allocated] == true;

    //for constructors only
    assume Heap[this, C.f] == null;
    call System.Object..ctor(this);
    ...
}
\end{verbatim}

Preserve type information

Preserve initialization semantics
Virtual Methods: Example

class Cell{
    public int x;

    protected virtual void Set(int x)
        modifies this.*;
        ensures this.x == x;
    {
        this.x = x;
    }

    public void Inc(int x)
        modifies this.*;
        ensures this.x==old(this.x)+x;
    {
        this.Set(Get()+x);
    }
}

class BackupCell: Cell{
    int b;

    protected override void Set(int x)
        ensures this.b == old(this.x);
    {
        this.b = this.x;
        base.Set(x);
    }
}
Behavioral Subtyping should guarantee *substitutability*

- Wherever an object of type $T$ is expected an object of type $S$, where $S <: T$, should do without changing the program’s behavior expressed in $wp$

Sufficient conditions: Let $M_1$ be a virtual method and $M_2$ be its overridden method, then

- $M_2$ can *weaken* $M_1$’s precondition
- $M_2$ can *strengthen* $M_1$’s postcondition
Virtual Methods

Translate each *method* \( m \) declared in \( C \) into a

\[
\text{proc } m.C \text{ (this, …) requires this } \neq \text{ null } \land \text{ typeof(this) } \subsetneq C; \\
\]

\[
\ldots
\]

The precondition of the overriding method is inherited from the
overridden method; additional postconditions are conjoined

Translate *calls* of the form \( o.m() \) to the method on \( o \)’s most
specific static type
Method Framing

• For sound verification we assume that every method modifies the heap

• *Modifies clauses* in Spec# express which locations (evaluated in the method’s prestate) a method is allowed to modify

• Modifies clauses for an object o or array a have the form:
  - o.f allows modification of o’s f field
  - o.* allows modification of all of o’s fields
  - a[k] allows modification of a’s array location k
  - a[*] allows modification of all of a’s array locations
Let $W$ denote all locations a method is allowed to modify

- The Boogie PL post condition for a Spec# modifies clause

\[
\text{Tr}[[W]] = \\
(\forall o: \text{ref}, f: \text{name} :: \text{old}(\text{Heap}[o, \text{allocated}]) \\
\Rightarrow (o, f) \in \text{old}(W) \lor \text{old}(\text{Heap}[o, f]) = \text{Heap}[o, f])
\]
Virtual Methods: Example Translation

**Spec#**

```spec#
protected virtual void Set(int x)
    modifies this.*;
```

**Boogie**

```boogie
proc Cell.Set(this : Cell, x : int)
    requires this != null ∧ typeof(this) <: Cell;
    modifies Heap;
    ensures (∀o:ref, f: name :: old(Heap[o,allocated]))
        ⇒ o = this ∨ old(Heap[o,f]) = Heap[o,f]);
```
Loop Framing

• Loops might change the heap. Let $W$ denote the set of locations potentially changed by the loop

• For sound verification we havoc the heap. We add as loop invariant the assertion that *fields not written to don’t change*

$$\text{Tr } [[W]] =$$

$$(\forall o : \text{ref}, f: \text{name} :: \text{Heap}_{\text{entry}}[o, \text{allocated}]$$

$$\Rightarrow f \in W \lor \text{Heap}_{\text{entry}}[o,f] = \text{Heap}_{\text{current}}[o,f])$$

where $\text{Heap}_{\text{entry/current}}$ denote the entry/current incarnations of the Heap variable in the loop
Summary

Verifying object-oriented programs requires to

• axiomatize the declaration environment
  – to keep enough information around for verification

• decide on a storage model
  – to model updates and framing

• translate the method bodies, paying particular attention to
  – partiality of operations
  – virtual dispatch
  – method and loop frames