Translating Java bytecode to BoogiePL

Master project - Final presentation

Alex Suzuki

September 29, 2006

Outline

Recap

What has changed?

Formalization of the translation

Boogie modifications

Conclusion

Translating Java bytecode to BoogiePL

What is this project about?

Translating Java bytecode to BoogiePL

What is this project about?

- Providing a translation from bytecode to BoogiePL

Translating Java bytecode to BoogiePL

What is this project about?

- Providing a translation from bytecode to BoogiePL
  - Including exceptions

Translating Java bytecode to BoogiePL

What is this project about?

- Providing a translation from bytecode to BoogiePL
  - Including exceptions
- Formalizing the translation in Coq

Translating Java bytecode to BoogiePL

What is this project about?

- Providing a translation from bytecode to BoogiePL
  - Including exceptions
- Formalizing the translation in Coq
  - Basis for a soundness proof

Translating Java bytecode to BoogiePL

What is this project about?

- Providing a translation from bytecode to BoogiePL
  - Including exceptions
- Formalizing the translation in Coq
  - Basis for a soundness proof
- Extending Boogie to support exceptions

Translating an example

- Java source

```
public class Account {
    private int balance;
    public void deposit(int amount) { balance = balance + amount; }
}
```

Translating an example

- Java source

public class Account {
    private int balance;
    public void deposit(**int** amount) { balance = balance + amount; }
}

- JVM bytecode

```
public void deposit(int);
0:   aload_0
1:   aload_0
2:   getfield   #2; //Field balance:I
5:   iload_1
6:   iadd
7:   putfield   #2; //Field balance:I
10:  return
```

Translating an example (cont'd)

```
implementation Account.deposit(this: ref, param1: int) {
 init:
   assume this ≠ null ∧ typ(rval(this)) = Account;
   reg0r := this;    // argument to register transfer
   reg1i := param1;
   goto block_0;
 block_0:
   stack0r := reg0r; // aload_0
   stack1r := reg0r; // aload_0
   stack1i := toint(get(heap, fieldLoc(stack1r, Account.balance)));
   stack2i := reg1i; // iload_1
   stack1i := stack1i + stack2i; // iadd
   heap := update(heap, fieldLoc(stack0r, Account.balance), ival(stack1i));
   return;
}
```

Overview

What has changed?

Overview

What has changed?

- Simplified translation

Overview

What has changed?

- Simplified translation
- Revised heap axiomatization

Overview

What has changed?

- Simplified translation
- Revised heap axiomatization
- Method frame conditions

Overview

What has changed?

- Simplified translation
- Revised heap axiomatization
- Method frame conditions
- Static fields and methods

Simplified translation

Generic stack manipulation instructions (e.g. `dup`, `swap`) relied on type of stack elements.

Simplified translation

Generic stack manipulation instructions (e.g. `dup`, `swap`) relied on type of stack elements.

- Unnecessary complexity

  // dup
  $\text{stack}(h+1)[\mathit{stackType}(h)] := \text{stack}(h)[\mathit{stackType}(h)];$

Simplified translation

Generic stack manipulation instructions (e.g. `dup`, `swap`) relied on type of stack elements.

- Unnecessary complexity

  // dup
  stack$(h+1)[\mathit{stackType}(h)]$:= stack$(h)[\mathit{stackType}(h)]$;

- Solution: just perform operation on all types

  // dup
  stack$(h+1)$i := stack(h)i; *// primitive*
  stack$(h+1)$r := stack(h)r; *// reference*

Heap axiomatization

Still based on Poetzsch-Heffter formalization, but...

Heap axiomatization

Still based on Poetzsch-Heffter formalization, but...

- Support for arrays

```
function fieldLoc(ref, name) returns (Location);
function arrayLoc(ref, int) returns (Location);
```

Heap axiomatization

Still based on Poetzsch-Heffter formalization, but...

- Support for arrays

```
function fieldLoc(ref, name) returns (Location);
function arrayLoc(ref, int) returns (Location);
```

- Data model closer to Java (uses values now, not objects)

```
function update(Store, Location, Value) returns (Store);
function add(Store, Allocation) returns (Store);
function get(Store, Location) returns (Value);
function alive(Value, Store) returns (bool);
function new(Store, Allocation) returns (Value);
```

Method invocation

Previously heap variable was **havoc**ed and heap state was lost.

Method invocation

Previously heap variable was **havoc**ed and heap state was lost.

- Now we preserve the heap state in a variable

```
pre_heap := heap; // preserve heap
havoc heap;
```

Method invocation

Previously heap variable was **havoc**ed and heap state was lost.

- Now we preserve the heap state in a variable

```
pre_heap := heap; // preserve heap
havoc heap;
```

- Values that were alive in the *preheap* remain alive

assume $(\forall$ v: Value $\circ$ alive(v, pre_heap) $\Rightarrow$ alive(v, heap));

Method invocation

Previously heap variable was **havoc**ed and heap state was lost.

- Now we preserve the heap state in a variable

  pre_heap := heap; *// preserve heap*
  havoc heap;

- Values that were alive in the *preheap* remain alive

  assume ($\forall$ v: Value $\circ$ alive(v, pre_heap) $\Rightarrow$ alive(v, heap));

- Locations not in callee **modifies** remain unchanged

  assume ($\forall$ h: Store, l: Location, v:Value $\circ$ l $\neq$ Account.balance $\Rightarrow$
  get(heap, l) = get(pre_heap, l));

Static fields and methods

Added support for static fields and methods.

Static fields and methods

Added support for static fields and methods.

- Static fields: Introduce *type object*

function typeObject(t: **name**) **returns** (**ref**);
axiom ($\forall$ t: **name** $\circ$ typeObject(t) $\neq$ **null**);

// getstatic int C.f
stack($h+1$)i := toint(get(heap, fieldLoc(typeObject(C), C.f)));

Static fields and methods

Added support for static fields and methods.

- Static fields: Introduce *type object*

```
function typeObject(t: name) returns (ref);
axiom (∀ t: name ∘ typeObject(t) ≠ null);

// getstatic int C.f
stack(h + 1)i := toint(get(heap, fieldLoc(typeObject(C), C.f)));
```

- Treat static methods same as instance, just omit implicit **this** parameter

Formalization in Coq

I have produced a formalization in Coq, a theorem prover.

Formalization in Coq

I have produced a formalization in Coq, a theorem prover.

- Includes formalization of BoogiePL language

Formalization in Coq

I have produced a formalization in Coq, a theorem prover.

- Includes formalization of BoogiePL language
- Depends on bytecode formalization library *Bicolano*

Formalization in Coq

I have produced a formalization in Coq, a theorem prover.

- Includes formalization of BoogiePL language
- Depends on bytecode formalization library *Bicolano*
- Type-checkable $\Rightarrow$ guarantees syntactically correct BoogiePL

Formalization in Coq

I have produced a formalization in Coq, a theorem prover.

- Includes formalization of BoogiePL language
- Depends on bytecode formalization library *Bicolano*
- Type-checkable $\Rightarrow$ guarantees syntactically correct BoogiePL
- Basis for a soundness proof

Layout of development

Layout of Coq development:

Layout of development

Layout of Coq development:

- `BoogiePL.v`: Formalization of BoogiePL language

Layout of development

Layout of Coq development:

- `BoogiePL.v`: Formalization of BoogiePL language
  - Basically a description of the grammar

Layout of development

Layout of Coq development:

- `BoogiePL.v`: Formalization of BoogiePL language
  - Basically a description of the grammar
- `BoogieUtils.v`: Helper library for building expressions

Layout of development

Layout of Coq development:

- `BoogiePL.v`: Formalization of BoogiePL language
  - Basically a description of the grammar
- `BoogieUtils.v`: Helper library for building expressions
  - Grammar makes it hard to create expressions directly

Layout of development

Layout of Coq development:

- `BoogiePL.v`: Formalization of BoogiePL language
  - Basically a description of the grammar
- `BoogieUtils.v`: Helper library for building expressions
  - Grammar makes it hard to create expressions directly
- `Translation.v`: Translation of bytecode methods to BoogiePL programs

Example: Translating swap

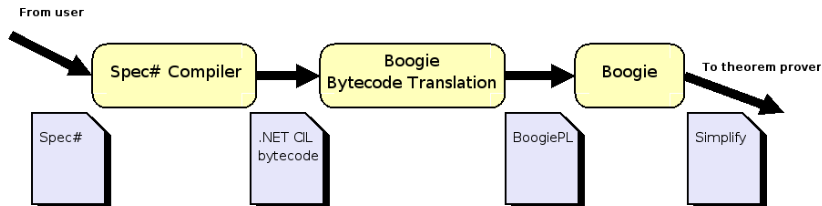
```
Definition TrInstruction (bm: BytecodeMethod) (pc: PC)
    (cur: Boogie.Block) : ( list Boogie.Block * Boogie.Block) :=

match BYTECODEMETHOD.instructionAt bm pc with
 ...
| Swap ⇒ match cur with
  | (blockid ,cmds,toc) ⇒ (blockid, cmds++(
    Assign (swapvar Ref) (BPL.IdentExpr (stvar (h−1) Ref))::
    Assign (swapvar Int) (BPL.IdentExpr (stvar (h−1) Int))::
    Assign (stvar (h−1) Ref) (BPL.IdentExpr (stvar h Ref))::
    Assign (stvar (h−1) Int) (BPL.IdentExpr (stvar h Int ))::
    Assign (stvar h Ref) (BPL.IdentExpr (swapvar Ref))::
    Assign (stvar h Int) (BPL.IdentExpr (swapvar Int ))::
     nil
   ))
  end
 ...
```

Example: Boogie declarations

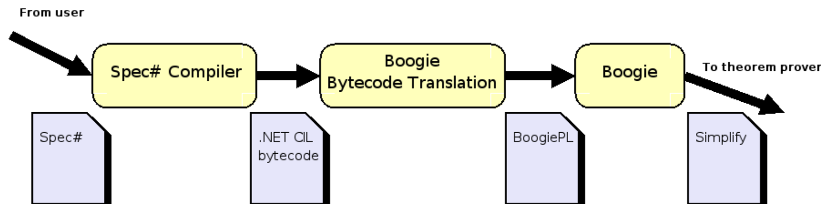
```
(* Declarations *)
Inductive Declaration : Set :=
| TypeDecl (ids: list Identifier)
| ConstantDecl (idtypes: list (Identifier * BPLType))
| FunctionDecl (ids: list Identifier)
    (params: list (option Identifier * BPLType))
| AxiomDecl (e: Expression)
| VariableDecl (idtypes: list (Identifier * BPLType))
| ProcedureDecl (id: Identifier) (params: list (Identifier * BPLType))
    (returnType: option BPLType) (specs: list Specification)
| ImplementationDecl (id: Identifier)
    (params: list (Identifier * BPLType))
    (returnType: option BPLType) (body: Body).
```

Boogie modifications: Overview



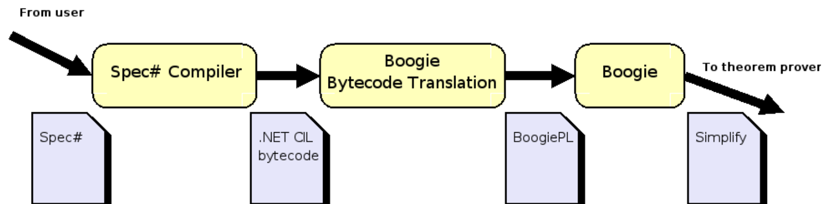
- Boogie does not translate exceptions

Boogie modifications: Overview



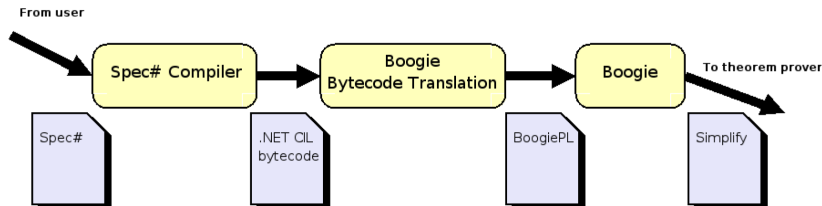
- Boogie does not translate exceptions
- Modifications to incorporate our methdology

Boogie modifications: Overview



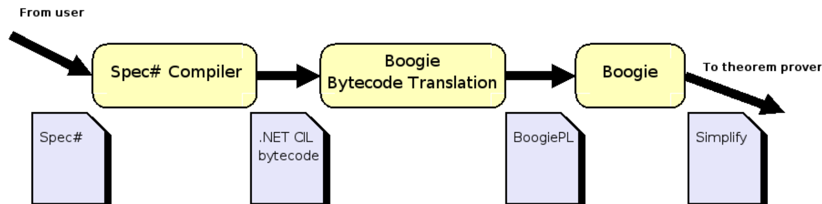
- Boogie does not translate exceptions
- Modifications to incorporate our methdology
  - Code in `catch` blocks

Boogie modifications: Overview



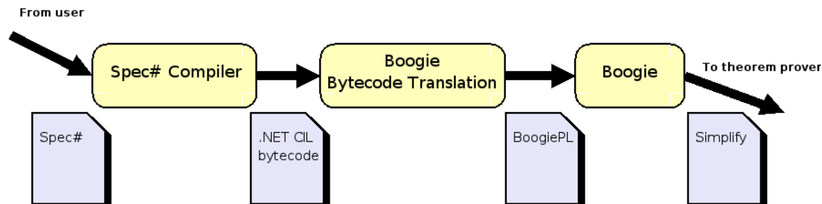
- Boogie does not translate exceptions
- Modifications to incorporate our methdology
  - Code in `catch` blocks
  - Exceptional method termination

Boogie modifications: Overview



- Boogie does not translate exceptions
- Modifications to incorporate our methdology
  - Code in `catch` blocks
  - Exceptional method termination
  - Throwing exceptions

Boogie modifications: Overview



- Boogie does not translate exceptions
- Modifications to incorporate our methdology
  - Code in `catch` blocks
  - Exceptional method termination
  - Throwing exceptions
- New commandline switch `/experimentalExceptions`

Code in catch blocks: Before

Code in catch blocks: Before

- CFG contains information about `catch` blocks

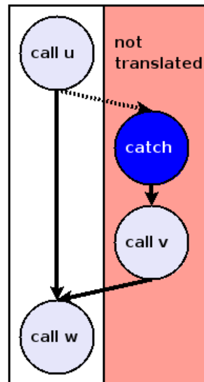
Code in catch blocks: Before

- CFG contains information about `catch` blocks
- `catch` handlers are exceptional successors of other nodes

Code in catch blocks: Before

- CFG contains information about `catch` blocks
- `catch` handlers are exceptional successors of other nodes
- But: translation only considers normal successors

```
try {
    u();
} catch (Exception){
    v();
}
w();
```



Code in catch blocks: After

Code in catch blocks: After

- Exceptional successors are no longer ignored

Code in catch blocks: After

- Exceptional successors are no longer ignored
- `catch` (InsufficientFundsException) translates to

  block_[*catch*]:
  havoc stack0o;
  assume cast(Heap[stack0o, allocated], **bool**) = true
  $\wedge$ stack0o $\neq$ **null**
  $\wedge$ typeof(stack0o) <: InsufficientFundsException ;
  // branch to handler code
  goto block_[*handler*];

Method invocation: Before

Method invocation: Before

- **call** statement desugars into a *state command*

Method invocation: Before

- **call** statement desugars into a *state command*
  - Corresponds to a let-binding, own set of local variables

Method invocation: Before

- **call** statement desugars into a *state command*
  - Corresponds to a let-binding, own set of local variables
- Methods are assumed to always terminate normally

Method invocation: Before

- **call** statement desugars into a *state command*
  - Corresponds to a let-binding, own set of local variables
- Methods are assumed to always terminate normally
- No additional blocks/execution paths generated

Method invocation: Before

- **call** statement desugars into a *state command*
  - Corresponds to a let-binding, own set of local variables
- Methods are assumed to always terminate normally
- No additional blocks/execution paths generated

```
Desugar[[result := call C.m(x,y)]] := {
    var call@formal@0: int, call@formal@1: int;
    call@formal@0 := x;
    call@formal@1 := y;

    assert Precondition(C.m, prestate);
    havoc Heap, result;
    assume Postcondition(C.m, prestate, poststate);
}
```

Method invocation: After

Desugar **call** statement “early”

Method invocation: After

Desugar **call** statement “early”

- Split desugaring over multiple blocks

Method invocation: After

Desugar **call** statement “early”

- Split desugaring over multiple blocks
- Successor blocks for normal and exceptional execution

Method invocation: After

Desugar **call** statement “early”

- Split desugaring over multiple blocks
- Successor blocks for normal and exceptional execution
  - Spec# doesn’t have exceptional postcondtions yet

Method invocation: After

Desugar **call** statement “early”

- Split desugaring over multiple blocks
- Successor blocks for normal and exceptional execution
  - Spec# doesn’t have exceptional postcondtions yet
  - Exceptional case not really useful right now

Method invocation: After

Desugar **call** statement “early”

- Split desugaring over multiple blocks
- Successor blocks for normal and exceptional execution
  - Spec# doesn’t have exceptional postcondtions yet
  - Exceptional case not really useful right now
  - `throws` clause not serialized ($\Rightarrow$ next release)

Method invocation: After (cont'd)

```
block_2042:
    call@formal@0 := x;
    call@formal@1 := y;
    call@old@Heap := Heap;
    assert Precondition(C.m, prestate);
    havoc Heap, result;
    goto block_2042_N, block_2042_X;

block_2042_N: // normal termination
    assume Postcondition(C.m, prestate, poststate);
    ...          // normal continuation

block_2042_X: // exceptional termination
    assume ExcpPostcondition(C.m, prestate, poststate);
    goto block_[handler];
```

Throwing exceptions: Before

Throwing exceptions: Before

- Throwing an exception generates the following code:

  assert stack(h)r $\neq$ **null**;
  assume false; *// !*
  return;

Throwing exceptions: Before

- Throwing an exception generates the following code:

  assert stack(h)r $\neq$ **null**;
  assume false; *// !*
  return;

- After an **assume** false; Boogie will “verify” anything, postcondition violations are ignored.

Throwing exceptions: After

Case distinction:

Throwing exceptions: After

Case distinction:

- Exception is caught (fully or partially):

  assert stack(h)r $\neq$ **null**;
  goto block_[*handler*];

Throwing exceptions: After

Case distinction:

- Exception is caught (fully or partially):

  assert stack(h)r $\neq$ **null**;
  goto block_[*handler*];

- No handler: old behavior (no exceptional postconditions yet)

  assert stack(h)r $\neq$ **null**;
  assume false;
  return;

Demo

```
public static void transfer (Account! src, Account! dest, int amount)
     requires amount > 0;
    throws TransferFailedException
{
    try {
        src.withdraw(amount);
    } catch ( InsufficientFundsException e) {
        throw new TransferFailedException();
    }
    dest.deposit(amount);
}
```

Conclusion

This project produced:

Conclusion

This project produced:

- Translation of Java bytecode to BoogiePL, including exceptions

Conclusion

This project produced:

- Translation of Java bytecode to BoogiePL, including exceptions
  - BoogiePL version of the Poetzsch-Heffter heap model with support for arrays

Conclusion

This project produced:

- Translation of Java bytecode to BoogiePL, including exceptions
  - BoogiePL version of the Poetzsch-Heffter heap model with support for arrays
- Formalization of the translation for the Coq theorem prover

Conclusion

This project produced:

- Translation of Java bytecode to BoogiePL, including exceptions
  - BoogiePL version of the Poetzsch-Heffter heap model with support for arrays
- Formalization of the translation for the Coq theorem prover
  - Basis for a soundness proof

Conclusion

This project produced:

- Translation of Java bytecode to BoogiePL, including exceptions
  - BoogiePL version of the Poetzsch-Heffter heap model with support for arrays
- Formalization of the translation for the Coq theorem prover
  - Basis for a soundness proof
- Boogie modifications for our approach of dealing with exceptions

Future work

Future work

- Support for floating-point numbers

Future work

- Support for floating-point numbers
  - Axiomatization in Boogie needed.

Future work

- Support for floating-point numbers
  - Axiomatization in Boogie needed.
- Implementation of the translation (use JML suite?)

Future work

- Support for floating-point numbers
  - Axiomatization in Boogie needed.
- Implementation of the translation (use JML suite?)
- Exception methodology for Spec# (exceptional postconditions)

Questions

? ? ?

Sushi apéro

Sushi apéro @ RZ F11, 15.00
Feel free to join!

Frame conditions

```
// requires amount > 0;
// modifies this.balance;
// ensures balance + amount == old(balance);
// when InsufficientFundsException ensures balance == old(balance);
public void withdraw(int amount) throws InsufficientFundsException {
    ...
}

public static void transfer(Account src, Account dest, int amount) {
    try {
        src.withdraw(amount);
    } catch (InsufficientFundsException e) {
        ...
    }
    ...
}
```

Frame conditions (cont'd)

```
  assume (forall v: Value ∘ alive(v, pre_heap) ⇒ alive(v, heap));
  assume (forall h: Store, l: Location, v:Value ∘
    l ≠ fieldLoc(arg0r, Account.balance)
    ⇒ get(heap, l) = get(pre_heap, l));
  goto block_2_N, block_2_InsufficientFundsException ;
block_2_N:
  assume stack1i > 0 ⇒ // normal postcondition
        toint(get(heap, fieldLoc(arg0r, Account.balance))) + stack1i
        = toint(get(pre_heap, fieldLoc(arg0r, Account.balance)));
   ...
 block_2_InsufficientFundsException :
  havoc stack0r; // exceptional postcondition
  assume alive(rval(stack0r), heap);
  assume typ(rval(stack0r)) <: InsufficientFundsException ;
  assume true ⇒ get(pre_heap, fieldLoc(arg0r, Account.balance)) =
                                   get(heap, fieldLoc(arg0r, Account.balance));
   ...
```

Normal and exceptional postconditions

Normal and exceptional postconditions:

```
// ensures balance + amount == old(balance);
// when InsufficientFundsException ensures balance == old(balance);
public void withdraw(int amount) { ...}

post:
    // old(this.balance) == this.balance + param1
    assert toint(get(old_heap, fieldLoc(this, Account.balance))) ==
           toint(get(heap, fieldLoc(this, Account.balance))) + param1;
    return;

 post_X_InsufficientFundsException :
    // old(this.balance) == this.balance
    assert toint(get(heap, fieldLoc(this, Account.balance))) ==
           toint(get(old_heap, fieldLoc(this, Account.balance)));
    return;
}
```