Translating Java Bytecode to BoogiePL

Alex Suzuki

Master Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

http://sct.inf.ethz.ch/

October 2006

Supervised by:
Hermann Lehner
Prof. Dr. Peter Müller

Software Component Technology Group
inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

BoogiePL is a typed, procedural language tailored to modular verification. It serves as the input language of the Boogie program verifier, a core component of the Spec# programming system. The Spec# compiler generates annotated CIL (Common Intermediate Language) bytecode, which Boogie translates to BoogiePL and then uses the weakest precondition calculus to generate verification conditions, which are ultimately passed to an automatic theorem prover, currently Simplify.

While Boogie is a part of the Spec# programming system, there is no reason why it should not be leveraged to verify programs written in other object-oriented languages, namely Java. To this end, a suitable translation from Java bytecode to BoogiePL has to be found.

This project presents a translation from Java bytecode to BoogiePL, including a formalization for the Coq theorem prover. Our translation includes support for exceptions, and we show how Boogie can be modified to support our methodology of dealing with exceptions.

Contents

Chapter 1

Introduction

1.1 Translating Java bytecode to BoogiePL

The goal of this project is to provide a translation from Java bytecode, the "assembly language" of Java, to BoogiePL [4], an imperative language with constructs suitable for modular verification of object-oriented programs. BoogiePL is the input language of the Boogie program verifier [7], which is used in the Spec# [11] development environment to verify a superset of C#, enriched with specification machinery. Boogie transforms a BoogiePL program into a representation suitable for the generation of compact verification conditions, which are then passed to a theorem prover, currently Simplify [5]. Translating Java bytecode to BoogiePL allows us to use the program verifier to verify compiled Java code. We provide a translation routine that sequentially translates bytecode methods into BoogiePL. Using unstructured bytecode, as opposed to Java, enables us to dodge the many complexities of structured Java code and reason about the actual instructions that are executed on the virtual machine machine, while also simplifying the translation itself.

We model the operand stack and registers with variables, and use an axiomatic model which supports arrays for the heap. Our translation includes exceptions, which are modeled by non-deterministic branches to blocks representing normal and exceptional executions. Runtime exceptions can be dealt with in the same way, but in our translation we use assertions to guarantee that such exceptions do not occur. We present our translation in both an informal fashion and as a type-checkable development for the Coq [1] theorem prover. We also show what kind of changes have to be made to Boogie to incorporate our approach for dealing with exceptions.

In the next sections, a brief overview of Java bytecode is given, along with an introduction to BoogiePL and the Spec# programming system. Chapter 2 presents the translation as a collection of translation functions that produce BoogiePL code from bytecode. Chapter 3 contains the formalization for the Coq theorem prover. The development includes a formalization of the BoogiePL language, and relies on the Bicolano[2] library for the Java bytecode formalization. In Chapter 4 we describe the extensions we implemented for Boogie to incorporate our methodology of modeling exceptions. Chapter 5 concludes and outlines some possible areas for future work.

1.2 The Java Virtual Machine

The JVM (Java Virtual Machine) is the execution environment for Java programs, and is described in detail in the Java Virtual Machine Specification [10]. The JVM loads and executes class files on a stack-based machine. Class files contain bytecode, the compiled representation of Java methods.

1.2.1 Instruction set

Java bytecode is the instruction set executed by a JVM. Instructions may manipulate the operand stack, the registers (these correspond to the local variables) and the heap. Most instructions are

typed, meaning the types of the operands are known beforehand. For instance, the `iadd` instruction (integer addition) pops two integers from the stack and pushes back their sum. This instruction is an integer instruction (note the `i`-prefix), as opposed to its floating-point counterpart `fadd`, which performs the same operation for floating-point numbers. However, there are also some instructions that do not operate on a specific type, but perform generic stack operations, such as duplicating the top item (`dup`) or swapping the two top items (`swap`).

Instructions are allowed to make assumptions on the types of values on the operand stack and in the registers since the JVM will only execute code that has passed a process called *bytecode verification*, which Leroy describes very well in [9]. Essentially the code is executed in an abstract fashion (only types are considered, not actual values). The execution is iterated until no more type changes are incurred and a fixed-point is reached. This process is known as abstract interpretation.

1.2.2 Exceptions

Code ranges can be protected by an exception handler. The type of the exception being caught (or no type, which indicates that any exception is caught, this is used to represent **finally**), the protected code range, and the location of the handler are stored as entries in the method's exception table. If an exception occurs within the specified code range, the operand stack is cleared and control is transferred to the exception handler. Execution then resumes and the only item on the stack is a reference to an object of the caught type or a subtype.

1.2.3 An example

The following example shows a simple class Account and the bytecode resulting from a compilation[1]. Withdrawing money from an account may result in an exception, the withdraw method indicates this by mentioning the class InsufficientFundsException in its **throws** clause. The transfer method calls both withdraw and deposit and catches the exception that might occur when calling withdraw. The example is used in chapter 2 to illustrate our translation and in chapter 4 to demonstrate how Spec# code is compiled to CIL and to show the differences generated by our modifications to Boogie.

```
1  public class Account {
2      Account(int initial) {
3          balance = initial;
4      }
5
6      public void deposit(int amount) {
7          balance = balance + amount;
8      }
9
10     public void withdraw(int amount) throws InsufficientFundsException {
11         if (balance < amount) {
12             throw new InsufficientFundsException();
13         } else {
14             balance = balance   amount;
15         }
16     }
17
18     public static void transfer(Account src, Account dest, int amount)
19         throws TransferFailedException
20     {
21         try {
22             src.withdraw(amount);
23         } catch (InsufficientFundsException e) {
24             throw new TransferFailedException();
```

[1] The Sun Java 1.5 compiler is used in the example

```
25          }
26          dest.deposit(amount);
27      }
28
29      private int balance;
30  }
```

In the bytecode output produced by `javap` we have omitted the constant pool and "inlined" references such as field identifiers or string constants. Note the exception table of transfer which indicates that the program counter range $[0, 5)$ is protected by a handler for the type InsufficientFundsException.

```
public class Account extends java.lang.Object {

Account(int);
  Code:
   Stack=2, Locals=2, Args_size=2
   0:   aload_0
   1:   invokespecial  java.lang.Object.<init>();
   4:   aload_0
   5:   iload_1
   6:   putfield       balance;
   9:   return

public void deposit(int);
  Code:
   Stack=3, Locals=2, Args_size=2
   0:   aload_0
   1:   aload_0
   2:   getfield       balance;
   5:   iload_1
   6:   iadd
   7:   putfield       balance;
   10:  return

public void withdraw(int);
   throws InsufficientFundsException
  Code:
   Stack=3, Locals=2, Args_size=2
   0:   aload_0
   1:   getfield       balance;
   4:   iload_1
   5:   if_icmpge      16
   8:   new    InsufficientFundsException;
   11:  dup
   12:  invokespecial  InsufficientFundsException.<init>()
   15:  athrow
   16:  aload_0
   17:  aload_0
   18:  getfield       balance;
   21:  iload_1
   22:  isub
   23:  putfield       balance;
   26:  return
  Exceptions:
   throws InsufficientFundsException

public static void transfer(Account,Account,int);
   throws TransferFailedException
```

```
  Code:
   Stack=2, Locals=4, Args_size=3
   0:   aload_0
   1:   iload_2
   2:   invokevirtual  withdraw(int);
   5:   goto    17
   8:   astore_3
   9:   new     TransferFailedException;
   12:  dup
   13:  invokespecial  TransferFailedException.<init>();
   16:  athrow
   17:  aload_1
   18:  iload_2
   19:  invokevirtual  deposit(int);
   22:  return
  Exception table:
   from   to  target type
     0     5     8   Class InsufficientFundsException
  Exceptions:
   throws TransferFailedException
}
```

1.3 The Boogie program verifier

Boogie is a modular program verifier for object-oriented programs and is a core component of the Spec# programming system. The Spec# compiler compiles a superset of C# to CIL (.NET bytecode) and serializes the contract information into metadata attributes. Boogie then processes this binary and in a first step transforms it into its internal representation, a BoogiePL program. From this program verification conditions are generated using the weakest precondition calculus and are then passed to a theorem prover which attempts to find a counterexample. If one is found, it is interpreted and passed back up the layers, and the erroneous piece of Spec# code is highlighted for the user to correct. The flow of information in the Spec# programming system is depicted in figure 1.1.

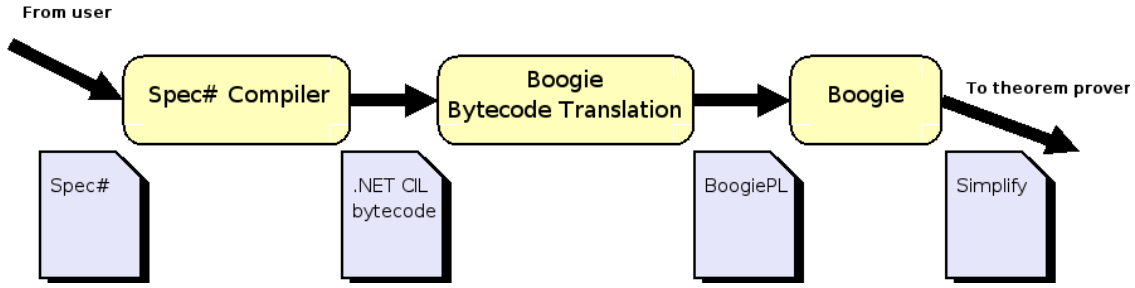

Figure 1.1: Flow of information in Spec#

1.3.1 The BoogiePL language

BoogiePL is a typed, procedural language tailored to modular verification. It contains statements that represent assumptions, assertions and axioms. The language is described in detail in [4], however we summarize the core parts of the language briefly here.

The language offers a range of built-in types, namely the numeric type (**int**), including the basic arithmetic facilities, the boolean type (**bool**), a type for object references (**ref**), and array types. The type **any** is provided as well, as a common supertype. Additionally the user is free to

declare new types using **type** declarations. There also exists a special type **name** which is used for unique identifiers such as field or class names. The language guarantees that such names are unique and provides equality and a partial order <: on **name**s.

Conditions can be tested and assumed with **assert** and **assume** statements, respectively. Conditions and rules that are assumed to always hold can be specified using **axiom** statements. Methods can be annotated with pre- and postconditions by using the **requires** and **ensures** clauses. Frame conditions that indicate the fields a method might modify are supplied with the **modifies** clause.

Variables can be assigned arbitrary values using the **havoc** statement, and **old** expressions can be used to reason about differences in an objects pre- and poststate.

User-defined functions can be introduced with the **function** statement. Usually they operate on a user-defined **type** and their behavior is described with axioms.

Methods are represented by BoogiePL **procedure**s, and consist of basic blocks (representing control flow graph nodes) which are connected by non-deterministic **goto** statements. During verification, all possible paths through the graph of basic blocks are checked.

Examples of BoogiePL programs will be given in section 2.2.

Chapter 2

Translation of Java bytecode to BoogiePL

In this chapter we present a translation from Java bytecode to BoogiePL. The unit of translation is a bytecode method, resulting in a BoogiePL procedure. The translation is presented in an informal way, the exact formalization for the Coq theorem prover is provided in chapter 3.

2.1 Main ideas of the translation

In the following sections we briefly show how we translate several aspects of Java bytecode. In section 2.2 we then illustrate the concepts by translating the Account class we introduced in section 1.2.3.

2.1.1 Operand stack and registers

The JVM operand stack is modeled by a set of BoogiePL variables of the form stack i t, where i denotes the depth of the stack and t is the type of the stack element, which can be either **int** or **ref**. If for example a binary arithmetic operation on integers is found at a given location with stack height two, then stack0i and stack1i are the operands of the instruction and stack0i holds the result of the arithmetic operation after the instruction has been processed.

Registers are treated similarly to stack elements. We use variables reg i t to represent the i-th register when its type is primitive or a reference type. If for example an `aload_0` instruction is encountered, and the current stack height is two, then the variable stack2r is assigned the value of reg0r after the instruction has been processed.

2.1.2 Axiomatic heap

We use the heap model described in [12], extend it to support arrays, and translate it into a BoogiePL representation. In BoogiePL, we describe the heap through a variable of type Store, axiomatize its behavior with axioms and provide access through functions. In this section we only show excerpts of the formalization, the complete listing can be found in appendix A.

// Müller/Poetzsch Heffter BoogiePL store axiomatization
type Store;

var heap: Store;

Types are expressed using the built-in **name** type, where the partial order <: expresses subtyping. The constant $int is used to denote the integer type. The axiom IsValueType($int) states

that the integer type is a value type. A one-dimensional array type is generated from another type by using the arrayType(**name**) function.

```
function IsClassType(name) returns (bool);
function IsValueType(name) returns (bool);
function IsArrayType(name) returns (bool);

// primitive types
const $int: name;
axiom IsValueType($int);

// array types
function arrayType(name) returns (name);
axiom (∀ t:name ∘ IsArrayType(arrayType(t)));

function elementType(name) returns (name);
axiom (∀ t:name ∘ elementType(arrayType(t)) = t);
```

Data on the heap is represented as *values*, which can be primitive (integer[1]), reference (class) or arrays. Functions are provided to convert these types to BoogiePL **int** and **ref** types and vice versa. For arrays a function arrayLength(Value) is introduced to denote the length of the array.

```
// Values (objects, primitive values, arrays)
type Value;

// integer values
function ival(int) returns (Value);
axiom (∀ x1: int, x2: int ∘ ival(x1) = ival(x2) ⟺ x1 = x2);
axiom (∀ v: Value ∘ ival(toint(v)) = v);

function toint(Value) returns (int);
axiom (∀ x: int ∘ toint(ival(x)) = x);

// reference values
function rval(ref) returns (Value);
axiom (∀ o1: ref, o2: ref ∘ rval(o1) = rval(o2) ⟺ o1 = o2);
axiom (∀ v: Value ∘ rval(toref(v)) = v);

function toref(Value) returns (ref);
axiom (∀ o: ref ∘ toref(rval(o)) = o);

// array values
function aval(ref) returns (Value); // array value
axiom (∀ o1:ref, o2:ref ∘ aval(o1) = aval(o2) ⟺ o1 = o2);
axiom (∀ o:ref, t:name ∘ toref(aval(o)) = o);

// array length
function arrayLength(Value) returns (int);
```

Values have a type, denoted by the function typ(Value). Uninitialized objects have a type-dependent default value. For integers it is 0, for reference and array types **null**.

```
// type of a value
function typ(Value) returns (name);
axiom (∀ x: int ∘ typ(ival(x)) = $int);

// uninitialized (default) value
function init(name) returns (Value);
```

[1] The boolean type is omitted since the JVM represents boolean values as integers.

```
axiom init($int) = ival(0);
axiom (∀ ct: name ∘ IsClassType(ct) ⇒ init(ct) = rval(null));
axiom (∀ at: name ∘ IsArrayType(at) ⇒ init(at) = rval(null));
```

Values can be *static* (in which case they are always alive). Primitive values and the null reference are static values.

```
function static(Value) returns (bool);
axiom (∀ x: Value ∘ static(x) ⟺ (IsValueType(typ(x)) ∨ x = rval(null)));
```

Values on the heap reside in *locations*. A location can either be an instance variable (*field location*), or an element of an array (*array location*). Instance variables are qualified by an object reference and a field identifier (**name** constants are used to uniquely identify fields). An array location is qualified by the reference pointing to the array and the array index. Locations have a type, for field locations the location type is the field type, for array locations the location type is the array element type.

```
// Locations (fields and array elements)
type Location;

// An instance field (use typeObject for static fields)
function fieldLoc(ref, name) returns (Location);
axiom (∀ o1: ref, o2: ref, f1: name, f2: name ∘
    (fieldLoc(o1, f1) = fieldLoc(o2, f2)) ⟺ ((f1 = f2) ∧ (o1 = o2)));

// An array element
function arrayLoc(ref, int) returns (Location);

// The object reference referring to an array element or instance variable
function obj(Location) returns (ref);
axiom (∀ o: ref, f: name ∘ obj(fieldLoc(o, f)) = o);
axiom (∀ o: ref, n: int ∘ obj(arrayLoc(o, n)) = o);

// Type of a location
function ltyp(Location) returns (name);
axiom (∀ o: ref, f: name ∘ ltyp(fieldLoc(o, f)) = fieldType(f));
axiom (∀ o: ref, i: int ∘ ltyp(arrayLoc(o, i)) = elementType(typ(aval(o))));
```

Fields are declared by introducing a **name** constant for the field, and specifying its type with the fieldType(**name**) function.

```
// Field declaration
function fieldType(name) returns (name);
```

Static fields are supported by introducing a function which for a given class type returns a reference to a type object.

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

An *allocation* denotes the nature of the object being allocated, either class or array type. This abstraction, along with the notion of locations, allows uniform access to the heap.

```
type Allocation;

// An object of class type
function objectAlloc(name) returns (Allocation);

// An array of given element type and size
```

```
function arrayAlloc(name, int) returns (Allocation);

function allocType(Allocation) returns (name);
axiom (∀ t: name ∘ allocType(objectAlloc(t)) = t);
axiom (∀ t: name, n: int ∘ allocType(arrayAlloc(t,n)) = arrayType(t));
```

Five functions are provided to access to the heap. Their meanings are given in the comments.

```
// Return the heap after storing a value in a location.
function update(Store, Location, Value) returns (Store);

// Returns the heap after an object of the given type has been allocated.
function add(Store, Allocation) returns (Store);

// Returns the value stored in a location.
function get(Store, Location) returns (Value);

// Returns true if a value is alive in a given heap.
function alive(Value, Store) returns (bool);

// Returns a newly allocated object of the given type.
function new(Store, Allocation) returns (Value);
```

The rules governing the heap's behavior are expressed by thirteen axioms, their meanings are given in the comments. These axioms are translated directly from the axioms in [12].

```
// Field stores do not affect the values stored in other fields.
axiom (∀ l1: Location, l2: Location, h: Store, x: Value ∘
    (l1 ≠ l2) ⇒ get(update(h, l1, x), l2) = get(h, l2));

// Field stores are persistent.
axiom (∀ l: Location, h: Store, x: Value ∘
    (alive(rval(obj(l)), h) ∧ alive(x, h)) ⇒ get(update(h, l, x), l) = x);

// Reading a field from a non alive object yields a type dependent default value.
axiom (∀ l: Location, h: Store ∘ ¬alive(rval(obj(l)), h) ⇒ get(h, l) = init(ltyp(l)));

// Updates through non living objects do not affect the heap.
axiom (∀ l: Location, h: Store, x: Value ∘ ¬alive(x, h) ⇒ (update(h, l, x) = h));

// Object allocation does not affect the existing heap.
axiom (∀ l: Location, h: Store, a: Allocation ∘ get(add(h, a), l) = get(h, l));

// Field stores do not affect object liveness.
axiom (∀ l: Location, h: Store, x: Value, y: Value ∘
    alive(x, update(h, l, y)) ⟺ alive(x, h));

// An object is alive if it was already alive or if it is the new object.
axiom (∀ h: Store, x: Value, a: Allocation ∘
    alive(x, add(h, a)) ⟺ alive(x, h) ∨ x = new(h, a));

// Values held stored in fields are alive.
axiom (∀ l: Location, h: Store ∘ alive(get(h, l), h));

// Static values are always alive.
axiom (∀ h: Store, x: Value ∘ static(x) ⇒ alive(x, h));

// A newly allocated object is not alive in the heap it was created in.
axiom (∀ h: Store, a: Allocation ∘ ¬alive(new(h, a), h));
```

```
// Allocated objects retain their type.
axiom (∀ h: Store, a: Allocation ∘ typ(new(h, a)) = allocType(a));

// Creating an object of a given type in two heaps yields the same result if liveness of
// all objects of that type is identical in both heaps.
axiom (∀ h1: Store, h2: Store, a: Allocation ∘
    (new(h1,a) = new(h2,a)) ⟺
    (∀ x: Value ∘ (typ(x) = allocType(a)) ⇒ (alive(x,h1) ⟺ alive(x,h2))));

// Two heaps are equal if they are indistinguishable by the alive and get functions.
axiom (∀ h1: Store, h2: Store ∘
    (∀ x: Value ∘ alive(x, h1) ⟺ alive(x, h2)) ∧
    (∀ l: Location ∘ get(h1, l) = get(h2, l)) ⇒ h1 = h2);
```

Three additional axioms have been added to the heap axiomatization that are not part of the Poetzsch-Heffter formalization.

```
// Get always returns a value whose type is a subtype of the (static) field type.
axiom (∀ h: Store, o:ref, f: name ∘ typ(get(h, fieldLoc(o, f))) <: fieldType(f));

// Transitivity of the IsClassType predicate
axiom (∀ t1: name, t2: name ∘ IsClassType(t1) && (t2 <: t1) ⇒ IsClassType(t2));

// New arrays have the allocated length
axiom (∀ h: Store, t: name, n: int ∘ arrayLength(new(h, arrayAlloc(t, n))) = n);
```

2.1.3 Control flow

Conditional branches

Conditional branches generate a non-deterministic branch to two successor blocks that assume the condition to be true or false. The true-block is then connected to the branch target (the true continuation), while the the false-block is connected to the block that starts with the instruction immediately following the conditional branch instruction (the false continuation).

Method calls

Method calls are translated to asserting the method's precondition, **havoc**ing the heap, and then branching non-deterministically to successor blocks that represent either normal or exceptional termination. In the case of normal termination, the called method's postcondition is assumed. In case the method terminates with an exception, its exceptional postcondition is assumed. Frame conditions can be used to express that fields not mentioned in the modifies clause remain unchanged.

In our translation we can not use the BoogiePL **call** statement because it will be desugared into a series of assertions and assumptions with no model of exceptional termination. The desugaring is illustrated in section 4.2.1.

Exceptions

When an exception occurs during the execution of a bytecode method, control is transferred from the instruction that caused the exception to an exception handler. The handler may be located in the same method as the instruction that caused the exception, or in one of the parent call frames. If no handler is found, a default top-level handler is called (a method of the currently executing ThreadGroup that prints the exception's stack trace). Since we are doing modular verification, we must distinguish between exceptions caught in the current method, and exceptions that are caught in parent frames. If an exception is not caught in the current method, it must appear in the method's **throws** clause. This condition is enforced at compile-time by Java compilers.

In our translation, instructions that can throw exceptions lead to the creation of additional blocks which represent the *normal* and possibly several *exceptional* executions of the instruction. In such an exceptional block, the heap is transformed to contain an exception object of a given type, and the top stack item is assumed to reference it. If a handler for the exception exists in the current method, a branch to the block containing the handler is added. If the exception is caught in a parent frame, a branch to the block that asserts the exceptional postcondition for the exception is added. The name of the target block is determined by the *lookupHandler* function.

$lookupHandler : BytecodeMethod \to PC \to ClassName \to BlockId$
Returns for a given program counter location and exception type, the name of the block that handles the exception, or the name of the block that asserts the exceptional postcondition depending on the exception type.

2.1.4 Bytecode formalization

JVM instructions

We support a reasonable subset of JVM instructions. Some instruction families (such as the `iconst` instructions) are represented by a single instruction (e.g. `iconst` Int). Instructions that reference the constant pool (such as field access and method invocation) are modeled explicitly, for instance `getfield` #3 is modeled as `getfield` $FieldSig$. Furthermore we restrict ourselves to integers, and do not consider **long**s and floating point numbers. We also introduce functions to navigate through the bytecode and fetch instructions to be translated.

$instructionAt : BytecodeMethod \to PC \to Instr$
Returns the bytecode instruction at a given program counter location.

$firstAddress : BytecodeMethod \to PC$
Returns the first program counter location of the given method.

$nextAddress : BytecodeMethod \to PC \to PC$
Returns the sequentially next program counter location.

$jump : PC \to Offset \to PC$
Returns the program counter location resulting from a conditional or non-conditional jump offset.

Method specifications

We assume that methods are annotated with a *precondition* and a *normal postcondition* and optionally one or more *exceptional pre- and postconditions*.

We introduce a function *getSpec* that retrieves a specification as a BoogiePL expression. The function takes as arguments the method for which the specification should be retrieved, the type of specification, and the set of variables that hold the pre- and poststate.

$SpecType := PreCond \mid PostCond \mid PreXCond(ClassName) \mid PostXCond(ClassName)$

$getSpec : MethodSig \to SpecType \to list\,Identifier \to BoogiePL\ expression$
Returns a method specification as a BoogiePL expression.

Bytecode verifier information

We can safely assume that our input method passed bytecode verification, so we can rely on the following to hold:

1. The height of the operand stack and its type (i.e. the type of all elements on the stack) is known for every instruction.
2. The height of the operand stack is always bounded by $MaxStackSize$ which is a property of the method.
3. The types of the values in the (possibly uninitialized) registers are known for every instruction.
4. The control flow graph of the method is known.

Using this information, we can introduce helper functions which support our translation. $isEdge$ and $isBlockStart$ allow us to translate the bytecode sequentially, while $getStackHeight$ and $getStackType$ free us from keeping track of the stack contents, otherwise we would have to perform an abstract interpretation.

$getStackHeight : BytecodeMethod \to PC \to Int$
Returns the height of the operand stack at the given program counter location.

$getStackType : BytecodeMethod \to PC \to Int \to Type$
Returns the type of the stack element at the given height and program counter location.

$isBlockStart : BytecodeMethod \to PC \to Boolean$
Returns if the given program counter location is the target of a CFG edge, or the first PC.

$isEdge : BytecodeMethod \to PC \to PC \to Boolean$
Returns if there is an edge between two given program counter locations in the CFG.

2.2 Translating the example

To illustrate the ideas shown in the previous sections, we present the translation of the Account class we presented in the introduction. To make things interesting, we introduce method specifications such as preconditions, postconditions and modifies clauses in comments, and assume that there is a mechanism (the *getSpec* function) to retrieve these specifications. The annotated source code is shown below.

```
public class Account {

    // ensures this.balance == initial;
    Account(int initial) {
        balance = initial;
    }

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

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

    // requires src != null && dest != null && amount > 0;
    // when TransferFailedException ensures true;
    public static void transfer(Account src, Account dest, int amount)
        throws TransferFailedException
    {
        try {
            src.withdraw(amount);
        } catch (InsufficientFundsException e) {
            throw new TransferFailedException();
        }
        dest.deposit(amount);
    }

    private int balance;
}
```

2.2.1 Translating types and fields

The class types and their field names are translated to **name** constant declarations. An axiom is introduced that states that Account.balance is an **int** field.

```
// Class and field declarations

const Account: name;
axiom Account <: java.lang.Object;

const Account.balance: name;
axiom fieldType(Account.balance) == $int;

const AccountException: name;
axiom AccountException <: java.lang.Exception;

const InsufficientFundsException: name;
axiom InsufficientFundsException <: AccountException;

const TransferFailedException: name;
axiom TransferFailedException <: AccountException;
```

2.2.2 Translating deposit

The deposit method serves as an illustration of how we translate access to the operand stack, registers and the heap.

```
public void deposit(int);
  Code:
```

```
Stack=3, Locals=2, Args_size=2
0:  aload_0
1:  aload_0
2:  getfield   Account.balance;
5:  iload_1
6:  iadd
7:  putfield   Account.balance;
10: return
```

The method has a precondition (amount > 0) which is assumed in the pre block, and a postcondition that says the new balance is the old balance plus the amount that has been deposited. The postcondition is asserted in the post block. Since deposit is an instance method of class Account, we can assume that the first argument is a non-**null** reference to an object of type Account and that the object is alive in the heap. To translate the postcondition which reasons about the prestate by using an old expression we preserve the heap variable by assigning it to the variable old_heap in the init block, which is also the place where the method arguments are transferred to the registers, as described in the JVM specification. The field accesses are translated to function applications on the heap model, including an assertion on the target reference being non-**null**.

```
procedure Account.deposit(this: ref, param1: int);
    modifies heap;

implementation Account.deposit(this: ref, param1: int) {
    var stack0r: ref, stack0i: int;
    var stack1r: ref, stack1i: int;
    var stack2r: ref, stack2i: int;
    var reg0r: ref, reg0i: int;
    var reg1r: ref, reg1i: int;
    var swapr: ref, swapi: int;
    var arg0r: ref, arg0i: int;
    var old_heap: Store;

init:
    assume this != null;                    // free requires
    assume typ(rval(this)) == Account; // free requires
    assume alive(rval(this), heap);         // free requires

    old_heap := heap;
    reg0r := this;
    reg1i := param1;
    goto pre;

pre:
    assume param1 > 0; // userdefined precondition
    goto block_0;

block_0:
    stack0r := reg0r;
    stack1r := reg0r;

    assert stack1r != null;
    stack1i := toint(get(heap, fieldLoc(stack1r, Account.balance)));
    stack2i := reg1i;
    stack1i := stack1i + stack2i;

    assert stack0r != null;
    heap := update(heap, fieldLoc(stack0r, Account.balance), ival(stack1i));
    goto post;
```

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

2.2.3 Translating withdraw

The translation of withdraw shows our methodology for control flow such as conditional branches, method invocation and exceptions. It also contains an object allocation (the `new` instruction). Furthermore the method has an exceptional postcondition specifying that the balance remains unchanged should an InsufficientFundsException be thrown.

```
public void withdraw(int);
  throws InsufficientFundsException
 Code:
  Stack=3, Locals=2, Args_size=2
  0:  aload_0
  1:  getfield  Account.balance;
  4:  iload_1
  5:  if_icmpge  16
  8:  new InsufficientFundsException;
  11: dup
  12: invokespecial InsufficientFundsException.<init>();
  15: athrow
  16: aload_0
  17: aload_0
  18: getfield  Account.balance;
  21: iload_1
  22: isub
  23: putfield  Account.balance;
  26: return
 Exceptions:
  throws InsufficientFundsException
```

The conditional branch at $PC = 5$ is translated into a non-deterministic branch to successor blocks block_5T and block_5F which assume the condition to be true or false.

In block_8 we construct an object of type InsufficientFundsException by assuming the top stack item to hold a reference to an object that has been added to the heap. The allocation is followed by a call to its default constructor, which has a trivial pre- and postcondition and does not throw an exception.

The `athrow` instruction at $PC = 15$ is translated to an assertion that the thrown reference is non-**null** and a branch to the block that asserts the exceptional postcondition, since no handler is present in the method.

```
procedure Account.withdraw(this: ref, param1: int);
    modifies heap;

implementation Account.withdraw(this: ref, param1: int) {
    var stack0r: ref, stack0i: int;
    var stack1r: ref, stack1i: int;
    var stack2r: ref, stack2i: int;
    var reg0r: ref, reg0i: int;
    var reg1r: ref, reg1i: int;
    var swapr: ref, swapi: int;
```

```
    var arg0r: ref, arg0i: int;
    var old_heap: Store;

init:
    assume this != null;                    // free requires
    assume typ(rval(this)) == Account;      // free requires
    assume alive(rval(this), heap);         // free requires

    old_heap := heap;
    reg0r := this;
    reg1i := param1;
    goto pre;

pre:
    assume param1 > 0; // userdefined precondition
    goto block_0;

block_0:
    stack0r := reg0r;
    assert stack0r != null;
    stack0i := toint(get(heap, fieldLoc(stack0r, Account.balance)));
    stack1i := reg1i;
    goto block_5T, block_5F; // if_icmpge

block_5T:
    assume stack0i >= stack1i;
    goto block_16;

block_5F:
    assume !(stack0i >= stack1i);
    goto block_8;

block_8:
    // new InsufficientFundsException
    havoc stack0r;
    assume rval(stack0r) == new(heap, objectAlloc(InsufficientFundsException));
    heap := add(heap, objectAlloc(InsufficientFundsException));
    stack1r := stack0r;

    // invokespecial InsufficientFundsException.<init>()
    arg0r := stack1r;
    assert arg0r != null;
    assert true; // precondition of InsufficientFundsException.<init>()
    goto block_12_N;

block_12_N:
    assume true; // postcondition of InsufficientFundsException.<init>()
    goto block_15;

block_15:
    // athrow
    assert stack0r != null;
    goto post_X_InsufficientFundsException;

block_16:
    stack0r := reg0r;
    stack1r := reg0r;
    assert stack1r != null;
```

```
    stack1i := toint(get(heap, fieldLoc(stack1r, Account.balance)));
    stack2i := reg1i;
    stack1i := stack1i    stack2i;
    assert stack0r != null;
    heap := update(heap, fieldLoc(stack0r, Account.balance), ival(stack1i));
    goto post;

post:
    // user defined postcondition: 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:
    // user defined exceptional postcondition: old(this.balance) == this.balance
    assert toint(get(heap, fieldLoc(this, Account.balance))) ==
           toint(get(old_heap, fieldLoc(this, Account.balance)));
    return;
}
```

2.2.4 Translating transfer

To conclude, the static transfer method features a method call to withdraw which might terminate with an exception. Both the calls to withdraw and deposit illustrate how we translate frame conditions, by assuming that locations not mentioned in the modifies clause remain unchanged. The method also contains a **catch** block which is reached in case of an exceptional execution of withdraw.

```
public static void transfer(Account,Account,int);
   throws TransferFailedException
  Code:
   Stack=2, Locals=4, Args_size=3
   0:   aload_0
   1:   iload_2
   2:   invokevirtual  Account.withdraw(int);
   5:   goto    17
   8:   astore_3
   9:   new TransferFailedException;
   12:  dup
   13:  invokespecial  TransferFailedException.<init>()
   16:  athrow
   17:  aload_1
   18:  iload_2
   19:  invokevirtual  Account.deposit(int)'
   22:  return
  Exception table:
   from   to  target type
     0     5     8   Class InsufficientFundsException

  Exceptions:
   throws TransferFailedException
```

The call to withdraw generates a block block_2_N which represents the method's normal execution and assumes the postcondition. The block representing an exceptional termination assumes that the only item on the stack is a reference to an object of type InsufficientFundsException or a subtype, and then assumes the callee's exceptional postcondition to hold. It then branches to block_8 which contains is the catch handler for the PC range $[0, 5)$.

```
procedure Account.transfer(param0: ref, param1: ref, param2: int);
    modifies heap;

implementation Account.transfer(param0: ref, param1: ref, param2: int) {
    var stack0r: ref, stack0i: int;
    var stack1r: ref, stack1i: int;
    var reg0r: ref, reg0i: int;
    var reg1r: ref, reg1i: int;
    var reg2r: ref, reg2i: int;
    var reg3r: ref, reg3i: int;
    var swapr: ref, swapi: int;
    var arg0r: ref, arg0i: int;
    var old_heap: Store, pre_heap: Store;

init:
    old_heap := heap;
    reg0r := param0;
    reg1r := param1;
    reg2i := param2;
    goto pre;

pre:
    assume param0 != null; // userdefined precondition
    assume param1 != null; // userdefined precondition
    assume param2 > 0;    // user defined precondition
    goto block_0;

block_0:
    stack0r := reg0r;
    stack1i := reg2i;
    // invokevirtual Account.withdraw()

    arg0r := stack0r;
    assert arg0r != null;
    assert stack1i > 0; // precondition of Account.withdraw(int)
    pre_heap := heap;
    havoc heap;

    // frame conditions
    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:
    // postcondition of Account.withdraw(int)
    assume stack1i > 0 ==>
            toint(get(heap, fieldLoc(arg0r, Account.balance))) + stack1i ==
            toint(get(pre_heap, fieldLoc(arg0r, Account.balance)));
    goto block_5;

block_2_InsufficientFundsException:
    // user defined exceptional postcondition
    havoc stack0r;
    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));
```

```
    goto block_8; // branch to handler

block_5:
    goto block_17;

block_8: // catch handler [0,5) InsufficientFundsException
    reg3r := stack0r;

    // new TransferFailedException
    havoc stack0r;
    assume rval(stack0r) == new(heap, objectAlloc(TransferFailedException));
    heap := add(heap, objectAlloc(TransferFailedException));

    stack1r := stack0r;
    // invokespecial TransferFailedException.<init>()

    assert stack1r != null;
    pre_heap := heap;
    assert true; // precondition of TransferFailedException.<init>()
    havoc heap;
    assume (forall v: Value :: alive(v, pre_heap) ==> alive(v, heap));
    goto block_13_N;

block_13_N:
    assume true; // postcondition of TransferFailedException.<init>()
    goto block_16;

block_16:
    // athrow
    assert stack0r != null;
    goto post_X_TransferFailedException;

block_17:
    stack0r := reg1r;
    stack1i := reg2i;

    // invokevirtual Account.deposit(int)
    arg0r := stack0r;
    assert arg0r != null;
    pre_heap := heap;
    assert stack1i > 0; // precondition of Account.deposit(int)
    havoc heap;

    // frame conditions
    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_19_N;

block_19_N:
    // postcondition of Account.deposit(int)
    assume stack1i > 0 ==>
            toint(get(heap, fieldLoc(arg0r, Account.balance))) ==
            toint(get(pre_heap, fieldLoc(arg0r, Account.balance))) + stack1i;

    goto block_22;

block_22:
```

```
    goto post;

post:
    assert true; // user defined postcondition
    return;

post_X_TransferFailedException:
    assert true; // user defined exceptional postcondition
    return;
}
```

2.3 The translation function

In the following functions we use a `monospace` font for literals of the translation. Lines beginning with a # character are interpreted and not part of the output. The symbol ↩ is used to indicate line breaks.

The Tr function is the root of the translation. The BoogiePL **procedure** and **implementation** headers, the local variable declarations and the initialization block are generated, and the function for translating the method body is applied.

$Tr[\![m : Method]\!] =$
  `procedure` $TrSig[\![m]\!]$ ↩
      `modifies heap;`↩
  `implementation` $TrSig[\![m]\!]$ ↩
  $TrVars[\![m]\!]$ ↩
  $TrInit[\![m]\!]$ ↩
  $TrBody[\![m]\!]$ ↩

The $TrSig$ function translates the signature of the method to a BoogiePL procedure signature. For instance methods, the first parameter is named this and represents the receiver object. Subsequent parameters are numbered from param1 to param$n$. If the method is static no implicit **this** parameter is added[2].

$TrSig[\![m : Method]\!] =$
  `(`
  #if $\neg isStatic(m)$
      `this: ref,`
  #end if
  #for $i := 0, i < |parameters(signature(m))|, i := i + 1$
      `param` i `:` $TrType[\![parameters(signature(m))[i]]\!]$ `,`
  #end for
  `)`
  #if $result(signature(m)) \neq VoidType$
      `returns ( result:` $TrType[\![result(signature(m))]\!]$ `)`
  #end if

As described in the previous sections, we use variables to model the stack and the heap. From the bytecode verifier we get the maximum size of the operand stack and the number of registers needed for the given method.

[2]The reason we do not just omit **this** and use param0 is because it is consistent with Bicolano.

$maxLocals : BytecodeMethod \to Int$
Returns the number of locals (registers) used by the method.

$maxOperandStackSize : BytecodeMethod \to Int$
Returns the maximum size of the operand stack.

With this information we can just create the variables needed for integer and reference types. Most likely some will remain unused, but Boogie will ignore unused variables, so it does not hurt to have these variables around. We also introduce variables for translating the `swap` instruction, for preserving the heap state and to store the first argument of a method invocation, since that stack element may be **havoc**ed if the method has a non-void return type.

$TrVars[\![m : Method]\!] =$
  $bm = body(m)$
  #for $i := 0, i < maxOperandStackSize(bm), i := i + 1$
    `var stack` i `r: ref, stack` i `i: int;`↩
  #end for
  #for $i := 0, i < maxLocals(bm), i := i + 1$
    `var reg` i `r: ref, reg` i `i: int;`↩
  #end for
    `var swapr: ref, swapi: int;`↩
    `var old_heap: Store, pre_heap: Store;`↩
    `var arg0r: ref, arg0i: int;`↩

$TrInit$ creates a block `init` that transfers the method arguments to the corresponding registers and assumes that the arguments are of the correct type and alive in the heap. Also the heap prestate is stored in the variable `old_heap` to support expressions referring to the method's prestate. For instance methods, the first register always holds the reference to the invocation target, **this**.

$TrInit[\![m : Method]\!] =$
  `init:`↩
    #if $\neg isStatic(m)$
    `assume this != null;`↩
    `assume typ(rval(this)) ==` $declaringClass(signature(m))$`;`↩
    `assume alive(rval(this), heap);`↩
    `reg0r := this;`↩
    #endif
    #for $i := 0, i < |parameters(signature(m))|, i := i + 1$
      #if $parameters(signature(m))[i]$ is $RefType$
      `assume typ(rval(param` i `r)) ==` $parameters(signature(m))[i]$`;`↩
      `assume alive(rval(param` i `r), heap);`↩
      #end if
      #if $\neg isStatic(m)$
        `reg` $(i+1)$ $TrTypeAbbrev[\![parameters(signature(m))[i]]\!]$ `:= param` i `;`↩
      #else
        `reg` i $TrTypeAbbrev[\![parameters(signature(m))[i]]\!]$ `:= param` i `;`↩
      #end if
    #end for
    `old_heap := heap;`↩
    `goto pre;`↩

The method body is translated by the $TrBody$ function, which begins by creating a block pre that assumes the precondition, and then branches to the first block of the the translated body. The translation of the JVM instructions then starts at the first program counter location ($PC = 0$) with the application of $TrInstructions$. At the end of the translation, the blocks that assert the normal postcondition and any exceptional postconditions are generated.

$TrBody[\![m : Method]\!] =$
  $bm := body(m)$
  `pre:`↩
      `assume` $getSpec(bm, PreCond, [heapVar, params])$ `;`↩
      `goto block_`$firstAddress(bm)$`;`↩
      ↩
  $TrInstructions[\![bm, firstAddress(bm)]\!]$ ↩
  `post:` ↩
      `assert` $getSpec(bm, PostCond, [heapVar, oldHeapVar, params, returnValue])$ `;`↩
      `return;`↩
      ↩
  #for $excType$ in $throws(bm)$
  `post_X_`$excType$`:` ↩
      `assert` $getSpec(bm, PostXCond(excType), [heapVar, oldHeapVar, params])$ `;`↩
      `return;`↩
  #end for

$TrInstructions$ is a wrapper for the translation of a single instruction. It uses the control flow graph information from the bytecode verifier to determine if a new block needs to be started. This is the case for the first instruction, any instructions that are targets of jumps, and additionally instructions that follow method calls or `athrow` instructions. If the successor instruction can be reached by other locations, the block has to be ended. This allows us to translate the body of the method sequentially.

$TrInstructions[\![m : BytecodeMethod, pc : PC]\!] =$
  #if $isBlockstart(m, pc)$
      `block_` pc `:` ↩
  #end
  $TrInstruction[\![m, pc]\!]$
  #if $isEdge(m, pc, nextAddress(m, pc))$
      `goto block_` $nextAddress(m, pc)$ `;`↩↩
  #end if
  $TrInstructions[\![m, nextAddress(m, pc)]\!]$

$TrInstruction$ translates a single JVM instruction. Certain instructions may cause successor blocks to be created (e.g. for assuming conditions after a conditional jump), note that a **goto** statement is added only to the block representing the branch decision, the **goto** to the block starting at the sequentially next program counter is added by $TrInstructions$.

For the sake of readability, we introduce some extra notation to describe stack variables. We write `stack`h to denote the stack variable representing the stack item at height h, the current stack height at the given program counter location, which can be retrieved by the function $getStackHeight$ described earlier.

2.3.1 Integer binary arithmetic

$TrInstruction[\![bm : BytecodeMethod, pc : PC]\!] =$
  $\#h := getStackHeight(bm, pc)$
  #switch $instructionAt(bm, pc)$

Integer binary arithmetic operations are done by performing simple arithmetic on the corresponding integer stack variables.

#case ***ibinop*** *op* : *AddInt*
`stack`$(h-1)$`i := stack`$(h-1)$`i + stack`h`i;`↩

#case ***ibinop*** *op* : *SubInt*
`stack`$(h-1)$`i := stack`$(h-1)$`i - stack`h`i;`↩

#case ***ibinop*** *op* : *MulInt*
`stack`$(h-1)$`i := stack`$(h-1)$`i * stack`h`i;`↩

Integer division and remainder operations cause an arithmetic runtime exception when dividing by zero. We take this fact into consideration by asserting that the divisor is not zero.

#case ***ibinop*** *op* : *DivInt*
`assert stack`h`i != 0;`↩
`stack`$(h-1)$`i := stack`$(h-1)$`i / stack`h`i;`↩

#case ***ibinop*** *op* : *RemInt*
`assert stack`h`i != 0;`↩
`stack`$(h-1)$`i := stack`$(h-1)$`i % stack`h`i;`↩

Since BoogiePL does not offer bitwise operations on integers, we use BoogiePL functions and axioms that describe the effect of the bitwise operation. For example, the effect of the bitwise left shift operation can be described with a function bit_shl as shown below.

---

function bit_shl(**int**, **int**) **returns** (**int**);

axiom (∀ i: **int** ∘ bit_shl(i, 0) = i);
axiom (∀ i: **int**, j: **int** ∘ 0 ≤ j ⇒ bit_shl(i, j + 1) = bit_shl(i, j) ∗ 2);

The other operations can be described similarly.

#case ***ibinop*** *op* : *AndInt*
`stack`$(h-1)$`i := bit_and(stack`$(h-1)$`i, stack`h`i);`↩

#case ***ibinop*** *op* : *OrInt*
`stack`$(h-1)$`i := bit_or(stack`$(h-1)$`i, stack`h`i);`↩

#case ***ibinop*** *op* : *XorInt*
`stack`$(h-1)$`i := bit_xor(stack`$(h-1)$`i, stack`h`i);`↩

#case ***ibinop*** *op* : *ShlInt*
`stack`$(h-1)$`i := bit_shl(stack`$(h-1)$`i, stack`h`i);`↩

#case ***ibinop*** *op* : *ShrInt*
`stack`$(h-1)$`i := bit_shr(stack`$(h-1)$`i, stack`h`i);`↩

#case ***ibinop*** *op* : *UshrInt*
`stack`$(h-1)$`i := bit_ushr(stack`$(h-1)$`i, stack`h`i);`↩

Integer negation is a unary operation on integers and is performed by negating the top integer stack item.

#case ***ineg***
```
        stackhi := -stackhi;↩
```

2.3.2 Pushing constants

Pushing an integer constant or the **null** reference is done by assigning a the constant to the appropriate stack variable of depth one plus the current depth.

#case ***iconst*** $n : int$
```
        stack(h + 1)i := n;↩
```

#case ***aconst_null***
```
        stack(h + 1)r := null;↩
```

2.3.3 Generic stack manipulation

The generic stack manipulation instructions `swap` and all variations of the `dup` instructions are untyped, so we simply perform the operation on both the integer and reference stack variables, this relieves us from keeping track of the types of all the items on the stack.

Since we do not support **long** integers or floating-point values, we can assume that all values fit into one register (these are called *category 1 computational types* in the JVM specification) and therefore simplify the translation of these instructions.

#case ***dup***
```
        stack(h + 1)r := stackhr;↩
        stack(h + 1)i := stackhi;↩
```

#case ***dup_x1***
```
        stack(h + 1)r := stackhr;↩
        stack(h + 1)i := stackhi;↩
        stackhr := stack(h − 1)r;↩
        stackhi := stack(h − 1)i;↩
        stack(h − 1)r := stack(h + 1)r;↩
        stack(h − 1)i := stack(h + 1)i;↩
```

#case ***dup_x2***
```
        stack(h + 1)r := stackhr;↩
        stack(h + 1)i := stackhi;↩
        stackhr := stack(h − 1)r;↩
        stackhi := stack(h − 1)i;↩
        stack(h − 1)r := stack(h − 2)r;↩
        stack(h − 1)i := stack(h − 2)i;↩
        stack(h − 2)r := stack(h + 1)r;↩
        stack(h − 2)i := stack(h + 1)i;↩
```

#case ***dup2***
```
        stack(h + 2)r := stackhr;↩
        stack(h + 2)i := stackhi;↩
        stack(h + 1)r := stack(h − 1)r;↩
        stack(h + 1)i := stack(h − 1)i;↩
```

#case ***dup2_x1***

```
        stack(h + 2)r := stackhr;↩
        stack(h + 2)i := stackhi;↩
        stack(h + 1)r := stack(h − 1)r;↩
        stack(h + 1)i := stack(h − 1)i;↩
        stackhr := stack(h − 2)r;↩
        stackhi := stack(h − 2)i;↩
        stack(h − 1)r := stack(h + 2)r;↩
        stack(h − 1)i := stack(h + 2)i;↩
        stack(h − 2)r := stack(h + 1)r;↩
        stack(h − 2)i := stack(h + 1)i;↩
```

#case ***dup2_x2***

```
        stack(h + 2)r := stackhr;↩
        stack(h + 2)i := stackhi;↩
        stack(h + 1)r := stack(h − 1)r;↩
        stack(h + 1)i := stack(h − 1)i;↩
        stackhr := stack(h − 2)r;↩
        stackhi := stack(h − 2)i;↩
        stack(h − 1)r := stack(h − 3)r;↩
        stack(h − 1)i := stack(h − 3)i;↩
        stack(h − 2)r := stack(h + 2)r;↩
        stack(h − 2)i := stack(h + 2)i;↩
        stack(h − 3)r := stack(h + 1)r;↩
        stack(h − 3)i := stack(h + 1)i;↩
```

#case ***swap***

```
        swapr := stack(h − 1)r;↩
        swapi := stack(h − 1)i;↩
        stack(h − 1)r := stackhr;↩
        stack(h − 1)i := stackhi;↩
        stackhr := swapr;↩
        stackhi := swapi;↩
```

Popping an item off the stack does not lead to any statements, **havoc**ing the stack variables is not necessary since they are assigned new values before they are read again.

#case ***pop***
#case ***pop2***

2.3.4 Register manipulation

Register loads and stores are translated to assigning the value of the top stack variable to the register variable and vice versa.

#case ***iload*** $n : RegNum$

```
        stack (h + 1)i := regni;↩
```

#case ***aload*** $n : RegNum$

```
        stack (h + 1)r := regnr;↩
```

#case ***istore*** $n : RegNum$

```
        regni := stackhi;↩
```

#case ***astore*** $n : RegNum$
    `reg`n`r := stack`h`r;`↩

Incrementing a register by an integer constant is done by adding the numerical constant to the register variable.

#case ***iinc*** $n : RegNum\ x : int$
    `reg`n`i := reg`n`i +` x`;`↩

2.3.5 Field access

Field read and write operations are performed by applying the heap functions *get* and *update*. Depending on the type of the field, helper functions that convert BoogiePL types to values stored in the heap (and vice versa) are applied. The heap location is defined by the object reference on the stack and the field identifier (a **name** constant). Assertions prevent accessing a field through a **null** reference.

#case ***getfield*** $f : FieldSig$
    `assert stack`h`r != null;`↩
    #switch $type(f)$
    #case int
        `stack`h`i := toint(get(heap, fieldLoc(stack`h`r,` f`)));`↩
    #case $RefType$
        `stack`h`r := toref(get(heap, fieldLoc(stack`h`r,` f`)));`↩
    #end switch

#case ***putfield*** $f : FieldSig$
    `assert stack(`$h-1$`)r != null;`↩
    #switch $type(f)$
    #case int
        `heap := update(heap, fieldLoc(stack(`$h-1$`)r,` f`), ival(stack`h`i));`↩
    #case $RefType$
        `heap := update(heap, fieldLoc(stack(`$h-1$`)r,` f`), rval(stack`h`r));`↩
    #end switch

Static fields are supported by accessing fields of a (non-**null**) type object, which is returned by the function typeObject(**name**). Otherwise they are treated in the same way as instance field accesses.

#case ***getstatic*** $f : FieldSig$
    $\#dt := declaringType(f)$
    #switch $type(f)$
    #case int
        `stack(`$h+1$`)i := toint(get(heap, fieldLoc(typeObject(`dt`),` f`)));`↩
    #case $RefType$
        `stack(`$h+1$`)r := toref(get(heap, fieldLoc(typeObject(`dt`),` f`)));`↩
    #end switch

#case ***putstatic*** $f : FieldSig$
    $\#dt := declaringType(f)$
    #switch $type(f)$
    #case int

```
        heap := update(heap, fieldLoc(typeObject(dt), f), ival(stackhi));↩
    #case RefType
        heap := update(heap, fieldLoc(typeObject(dt), f), rval(stackhr));↩
    #end switch
```

2.3.6 Array access

Array access is handled similarly to field access, both operations simply refer to heap locations. Access is uniform through the *get* and *update* functions. Array access is guarded by asserting that the array reference is not **null**.

```
#case iaload
    assert stack(h − 1)r != null;↩
    stack(h − 1)i := toint(get(heap, arrayLoc(stack(h − 1)r, stackhi)));↩

#case iastore
    assert stack(h − 2)r != null;↩
    heap := update(heap, arrayLoc(stack(h − 2)r, stack(h − 1)i), ival(stackhi));↩

#case aaload
    assert stack(h − 1)r != null;↩
    stack(h − 1)r := toint(get(heap, arrayLoc(stack(h − 1)r, stackhi)));↩

#case aastore
    assert stack(h − 2)r != null;↩
    heap := update(heap, arrayLoc(stack(h − 2)r, stack(h − 1)i), rval(stackhr));↩
```

2.3.7 Object allocation

Allocating an object with the `new` instruction is translated to applications of the heap functions *new* and *add*. The top stack item is assumed to hold a reference to an object of the given reference type after the instruction completes.

```
#case new t : RefType
    havoc stack(h + 1)r;↩
    assume rval(stack(h + 1)r) == new(heap, objectAlloc(t));↩
    heap := add(heap, objectAlloc(t));↩
```

2.3.8 Array allocation

Allocating an array with the `newarray` or `anewarray` instructions is treated in the same way as allocating an object of class type. The top stack item (which holds the array length before the instruction has executed) is assumed to hold a reference to an array of the given type after the instruction completes. The length of the array is the value given in the allocation.

```
#case newarray t : Type
    havoc stackhr;↩
    assume rval(stackhr) == new(heap, arrayAlloc(t, stackhi));↩
    heap := add(heap, arrayAlloc(t, stackhi));↩
    assume arrayLength(rval(stackhr)) == stackhi;↩
```

Note that we do not support multi-dimensional arrays (`multianewarray`) in our translation. In the underlying Bicolano formalization, the `newarray` and `anewarray` instructions are merged into a single instruction.

2.3.9 Conditional and unconditional branches

Conditional branch instructions on integers and references are translated to non-deterministic branches to blocks that contain the respective assumptions.

#case ***if_icmp*** $cond : IntCond,\ o : Offset$
```
        goto block_pcT, goto block_pcF;↩
        ↩
    block_pcT:↩
        assume stack(h − 1)i TrCond⟦cond⟧ stackhi;↩
        goto block_jump(pc, o);↩
        ↩
    block_pcF:↩
        assume !(stack(h − 1)i TrCond⟦cond⟧ stackhi);↩
```

#case ***if*** $cond : IntCond,\ o : Offset$
```
        goto block_pcT, block_pcF;↩
        ↩
    block_pcT:↩
        assume stackhi TrCond⟦cond⟧ 0;↩
        goto block_jump(pc, o);↩
        ↩
    block_pcF:↩
        assume !(stackhi TrCond⟦cond⟧ 0);↩
```

#case ***if_acmpeq*** $o : Offset$
```
        goto block_pcT, block_pcF;↩
        ↩
    block_pcT:↩
        assume stack(h − 1)r == stackhr;↩
        goto block_jump(pc, o);↩
        ↩
    block_pcF:↩
        assume stack(h − 1)r != stackhr;↩
```

#case ***if_acmpne*** $o : Offset$
```
        goto block_pcT, block_pcF;↩
        ↩
    block_pcT:↩
        assume stack(h − 1)r != stackhr;↩
        goto block_jump(pc, o);↩
        ↩
    block_pcF:↩
        assume stack(h − 1)r == stackhr;↩
```

#case ***ifnull*** $o : Offset$
```
        goto block_pcT, block_pcF;↩
        ↩
    block_pcT:↩
        assume stackhr == null;↩
```

```
        goto block_jump(pc, o);↩
        ↩
    block_pcF:↩
        assume stackhr != null;↩

#case ifnonnull o : Offset
        goto block_pcT, block_pcF;↩
        ↩
    block_pcT:↩
        assume stackhr != null;↩
        goto block_jump(pc, o);↩
        ↩
    block_pcF:↩
        assume stackhr == null;↩
```

Return statements assign to a designated result variable and branch to the block that asserts the normal postcondition. For methods that do not return anything, only a branch is generated.

```
#case ireturn
        result := stackhi;↩
        goto post;↩

#case areturn
        result := stackhr;↩
        goto post;↩

#case return
        goto post;↩
```

The `goto` instruction leads to a branch to the block that contains the translation of the instructions starting at the resulting offset.

```
#case goto o : Offset
        goto block_jump(pc, o);↩
```

The `tableswitch` and `lookupswitch` instructions are translated to a non-determinstic branch to all offsets stored in the instruction.

```
#case tableswitch default : Offset, low : Int, high : Int, targets : list Offset
        goto block_jump(pc, default)
        #for offset in targets
            , block_jump(pc, offset)
        #end for
        ;↩

#case lookupswitch default : Offset, table : list (Int, Offset)
        goto block_jump(pc, default)
        #for key, offset in table
            , block_jump(pc, offset)
        #end for
        ;↩
```

2.3.10 Method calls

The translation of an instance method call through the `invokevirtual` or `invokespecial` instructions depends on the method called, particularly its *throws* clause. Exceptions thrown by the target method which are caught in the current method lead to a branch to the block containing the handler. Exceptions which are not caught in the current method (and are thus in the **throws** clause of the current method) lead to the creation of a branch to the method's exceptional post-condition for that exception. To determine the set of exceptions we have to consider we introduce a function *invocationExceptions* which for a given program counter location and method to be invoked returns to us a list of possible candidates.

The heap variable is **havoc**ed to indicate that the method may have altered the heap. If the method supplies a modifies clause, we can assume that fields not mentioned in the clause remain unchanged. Object liveness is not affected by method calls, all objects that were alive in the heap prior to the method call will be alive in the heap after the method has executed.

If the method has a non-**void** return type, the top stack item is **havoc**ed to indicate that it has changed to something arbitrary.

#case ***invokevirtual*** $target : MethodSig$
#case ***invokespecial*** $target : MethodSig$
    `arg0r := stack(`$h - |parameters(target)|$`)r;`↩
    `assert arg0r != null;`↩
    `pre_heap := heap;`↩
    $\#params := [stack(h - |parameters(target)|)...stack(h)]$
    `assert` $getSpec(target, PreCond, [heap, params])$ `|| ...;`↩
    `havoc heap;`↩
    `assume (forall v: Value :: alive(v, pre_heap) ==> alive(v, heap));`↩
    `goto` [...]↩*// branch to all blocks created below*
    ↩

  #for T in $invocationExceptions(bm, pc, target)$
  `block_`*pc*`_`T`:`↩
    `havoc stack0r;`↩
    `assume alive(rval(stack0r), heap);`↩
    `assume typ(rval(stack0r)) <:` T`;`↩
    `assume` $getSpec(target, PreXCond(T), [old(heap), params])$ `==>`
        $getSpec(target, PostXCond(T), [heap, old(heap), params, returnval])$ `;`↩
    `goto` $lookupHandler(bm, pc, T)$`;`↩
    ↩
  #end for

  `block_`*pc*`_N:`↩
    #if $returnType(target) \neq Void$
    `havoc stack(`$h - |parameters(target)|$`)`$TrTypeAbbrev[\![returnType(target)]\!]$`;`↩
    #end if
    `assume` $getSpec(target, PreCond, [old(heap), params])$ `==>`
        $getSpec(target, PostCond, [heap, old(heap), params, returnval])$;↩

Static methods invoked by the `invokestatic` instruction are treated similarly.

2.3.11 Throwing exceptions

The `athrow` instruction throws an object whose type is known[3] through bytecode verification. An assertion is inserted to verify that the top stack item does not contain a **null** reference. The

[3] Actually, only the smallest common supertype is known, the actual object may be a subtype

current block is then ended with a branch to the handler or the block that asserts the exceptional postcondition if no handler is present. The details of looking up the correct handler in the exception table or the method's exceptional postcondition are left to the *lookupHandler* function.

#case ***athrow***
`assert stack`h`r != null;`↩
`stack0r := stack`h`r;`↩
`goto` $lookupHandler(bm, pc, getStackType(bm, pc, h))$`;`↩

2.3.12 Helper functions

TrCond translates an integer conditional operator to its BoogiePL counterpart.

$TrCond[\![cond : IntCond]\!] =$
#switch *cond*
#case *eq*
`==`
#case *ne*
`!=`
#case *lt*
`<`
#case *le*
`<=`
#case *gt*
`>`
#case *ge*
`>=`
#end switch

TrType translates a type to its BoogiePL representation (e.g. **ref** for reference types).

$TrType[\![t : Type]\!] =$
#switch t
#case $RefType$
`ref`
#case Int
`int`
#end switch

TrTypeAbbrev translates a type to its type abbreviation (e.g. `r` for reference types) for use in variable names.

$TrTypeAbbrev[\![t : Type]\!] =$
#switch t
#case $RefType$
`r`
#case Int
`i`
#end switch

2.4 Limitations of our translation

Currently we only support a subset of Java bytecode. Instructions related to threading (`monitorenter`, `monitorexit`) and wide data types (e.g. **long**, **float**, **double**) are not supported. Support for floating-point numbers has been omitted because the BoogiePL language does not offer a built-in floating-point type, and the translation function becomes unnecessarily complicated with the introduction of values that span more than one stack or register slot. Furthermore, control flow instructions for subroutines (`jsr`, `jsr_w` and `ret`) are not supported. These instructions are not generated anymore by recent compilers when compiling **try finally** statements. Instead, the **finally** blocks are duplicated. This removes another complexity, namely dealing with branch addresses as values on the operand stack.

Chapter 3

Formalization in Coq

3.1 Formalization of BoogiePL

3.1.1 Language core

The formalization of the BoogiePL language is basically a mapping from the grammar of the language, which is given in [4], to inductive types, and is contained in the BoogiePL library. The formalization includes all parts of the language, including array types, even though we do not use them for our representation of arrays.

```
(* Formalization of the BoogiePL language. Follows the grammar of the language as
   defined in "BoogiePL: a typed procedural language for checking object oriented
   programs" *)

Require Import List.
Require Import ZArith.

Open Scope type_scope.

Module Type BOOGIEPL.

  (* Identifiers *)
  Parameter Identifier: Set.

  (* Types *)
  Inductive BPLType: Set :=
    | Ref | Int | Bool | Any | Name | UserDef (id: Identifier)
    | OneDimArray (it: BPLType) (et: BPLType)
    | TwoDimArray (it1: BPLType) (it2: BPLType) (et: BPLType).

  (* Expressions, cumbersome to use directly. Use BoogieUtils library to
     build expressions instead. *)
  Inductive Expression: Set :=
    | EquivExprU (impl: ImplicationExpression)
    | EquivExpr (impl: ImplicationExpression) (equiv: Expression)
  with ImplicationExpression: Set :=
    | ImplicationU (e: LogicalExpression)
    | Implication (e: LogicalExpression) (impl: ImplicationExpression)
  with LogicalExpression: Set :=
    | LogicalExprU (r: Relation)
    | LogicalExprAnd (r: Relation) (e: AndExpression)
    | LogicalExprOr (r: Relation) (e: OrExpression)
  with AndExpression: Set :=
```

```
  | AndExprU (r: Relation)
  | AndExpr (r: Relation) (e: AndExpression)
with OrExpression: Set :=
  | OrExprU (r: Relation)
  | OrExpr (r: Relation) (e: OrExpression)
with Relation: Set :=
  | RelationU (t: Term)
  | RelationEq (a: Term) (b: Term) (* == *)
  | RelationNe (a: Term) (b: Term) (* != *)
  | RelationLt (a: Term) (b: Term) (* < *)
  | RelationLe (a: Term) (b: Term) (* <= *)
  | RelationGe (a: Term) (b: Term) (* >= *)
  | RelationGt (a: Term) (b: Term) (* > *)
  | RelationPo (a: Term) (b: Term) (* <: *)
with Term: Set :=
  | TermU (f: Factor)
  | TermAdd (t: Term) (f: Factor)
  | TermSub (t: Term) (f: Factor)
with Factor: Set :=
  | FactorU (e: UnaryExpression)
  | FactorMul (f: Factor) (e: UnaryExpression) (* f * e *)
  | FactorDiv (f: Factor) (e: UnaryExpression) (* f / e *)
  | FactorMod (f: Factor) (e: UnaryExpression) (* f % e *)
with UnaryExpression: Set :=
  | UnaryExprArray (e: ArrayExpression)
  | UnaryExprNot (e: UnaryExpression) (* !e *)
  | UnaryExprMinus (e: UnaryExpression) (* e *)
with ArrayExpression: Set :=
  | ArrayExprU (a: Atom)
  | ArrayExpr (i: Index) (* e[i] *)
with Index: Set :=
  | Index1D (e: Expression)
  | Index2D (e1: Expression) (e2: Expression)
with Atom: Set :=
  | False | True | Null
  | Number (n: Z)
  | Ident (id: Identifier)
  | FunctionCall (id: Identifier) (args: list Expression)
  | Old (e: Expression)
  | Cast (e: Expression) (t: BPLType)
  | Quant (q: Quantification)
  | Expr (e: Expression)
with Quantification: Set :=
  | Forall (ids: list Identifier) (e: Expression)
  | Exists (ids: list Identifier) (e: Expression).

(* Method Specifications *)
Inductive Specification: Set :=
  | Requires (e: Expression)
  | Modifies (vars: list Identifier)
  | Ensures (e: Expression).

(* Commands *)
Inductive Command: Set :=
  | Assign (id: Identifier) (e: Expression)
  | ArrayAssign (id: Identifier) (i: Expression) (e: Expression)
  | Assert (e: Expression)
  | Assume (e: Expression)
```

```
  | Havoc (ids: list Identifier )
  | Call (out: list Identifier ) (id: Identifier ) (args: list Expression).

(* Basic Blocks *)
Inductive TocManifesto: Set :=
  | Goto (blocks: list Identifier )
  | Return.

(* Block is a 3 tuple: identifier , list of commands and the
   transfer command (return or goto) *)
Definition Block : Set := (Identifier * list Command * TocManifesto).

(* Implementation provides local variables and body *)
Definition LocalVarDecl: Set := list ( Identifier * BPLType).
Definition Body: Set := (LocalVarDecl * list Block).

(* 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).

(* At the top level , a BoogiePL program is a sequence of declarations *)
Definition Program: Set := list Declaration.

End BOOGIEPL.
```

3.1.2 The BoogieUtils library

The BoogieUtils library provides shortcuts for commonly used expressions, which would otherwise be cumbersome to create using the grammar defined in the Boogie library.

```
(* Helper library to build BoogiePL expressions *)

Require Import List.
Require Import ZArith.

Add LoadPath "../Bicolano".
Require Import BoogiePL.
Require Import Program.

Open Scope type_scope.

Module BoogieHelper (Boogie: BOOGIEPL) (Prog: PROGRAM).
  Import Prog.
  Import Boogie.

  (* Helper routine that retrieves a block's identifier *)
  Definition blockId (b: Block) : Identifier :=
    match b with
      | (id,_,_) => id
    end.
```

```
(* Helper routine that retrieves a block identifiers from a list of blocks *)
Fixpoint blockIds (blocks: list Block) : list Identifier :=
  match blocks with
      | nil => nil
      | (id,_,_)::l => id::blockIds l
  end.

(* Integer literal *)
Definition IntConstant (z:Z): Expression :=
  EquivExprU (ImplicationU (LogicalExprU (RelationU (TermU (FactorU (UnaryExprArray (ArrayExprU (Number z))))))

(* The null reference *)
Definition NullConstant: Expression :=
  EquivExprU (ImplicationU (LogicalExprU (RelationU (TermU (FactorU (UnaryExprArray (ArrayExprU Null)))))))).

(* An identifier expression (e.g. a variable name) *)
Definition IdentExpr (id: Identifier ): Expression :=
  EquivExprU (ImplicationU (LogicalExprU (RelationU (TermU (FactorU (UnaryExprArray (ArrayExprU (Ident id)))))))

(* Comparison of two references *)
Definition CompRefExpr (e1 e2: Expression) (cmp: CompRef): Expression :=
  match cmp with
    | EqRef =>
      EquivExprU (ImplicationU (LogicalExprU (RelationEq
          (TermU (FactorU (UnaryExprArray (ArrayExprU (Expr e1)))))
          (TermU (FactorU (UnaryExprArray (ArrayExprU (Expr e2)))))
      )))
    | NeRef =>
      EquivExprU (ImplicationU (LogicalExprU (RelationNe
          (TermU (FactorU (UnaryExprArray (ArrayExprU (Expr e1)))))
          (TermU (FactorU (UnaryExprArray (ArrayExprU (Expr e2)))))
      )))
  end.

(* Comparison of two integers *)
Definition CompIntExpr (e1 e2: Expression) (cmp: CompInt): Expression :=
  match cmp with
    | EqInt =>
      EquivExprU (ImplicationU (LogicalExprU (RelationEq
          (TermU (FactorU (UnaryExprArray (ArrayExprU (Expr e1)))))
          (TermU (FactorU (UnaryExprArray (ArrayExprU (Expr e2)))))
      )))
    | NeInt =>
      EquivExprU (ImplicationU (LogicalExprU (RelationNe
          (TermU (FactorU (UnaryExprArray (ArrayExprU (Expr e1)))))
          (TermU (FactorU (UnaryExprArray (ArrayExprU (Expr e2)))))
      )))
    | LtInt =>
      EquivExprU (ImplicationU (LogicalExprU (RelationLt
          (TermU (FactorU (UnaryExprArray (ArrayExprU (Expr e1)))))
          (TermU (FactorU (UnaryExprArray (ArrayExprU (Expr e2)))))
      )))
    | LeInt =>
      EquivExprU (ImplicationU (LogicalExprU (RelationLe
          (TermU (FactorU (UnaryExprArray (ArrayExprU (Expr e1)))))
          (TermU (FactorU (UnaryExprArray (ArrayExprU (Expr e2)))))
      )))
```

```
    | GtInt =>
        EquivExprU (ImplicationU (LogicalExprU (RelationGt
            (TermU (FactorU (UnaryExprArray (ArrayExprU (Expr e1)))))
            (TermU (FactorU (UnaryExprArray (ArrayExprU (Expr e2)))))
        )))
    | GeInt =>
        EquivExprU (ImplicationU (LogicalExprU (RelationGe
            (TermU (FactorU (UnaryExprArray (ArrayExprU (Expr e1)))))
            (TermU (FactorU (UnaryExprArray (ArrayExprU (Expr e2)))))
        )))
  end.

(*  e *)
Definition UnaryMinusExpr (e: Expression): Expression :=
  EquivExprU (ImplicationU (LogicalExprU (RelationU
      (TermU (FactorU (UnaryExprMinus (UnaryExprArray (ArrayExprU (Expr e))))))
  ))).

(* !e *)
Definition UnaryNotExpr (e: Expression): Expression :=
  EquivExprU (ImplicationU (LogicalExprU (RelationU
      (TermU (FactorU (UnaryExprNot (UnaryExprArray (ArrayExprU (Expr e))))))
  ))).

(* e1 + e2 *)
Definition AddIntExpr (e1 e2: Expression): Expression :=
  EquivExprU (ImplicationU (LogicalExprU (RelationU (TermAdd
      (TermU (FactorU (UnaryExprArray (ArrayExprU (Expr e1)))))
      (FactorU (UnaryExprArray (ArrayExprU (Expr e2))))
  )))).

(* e1   e2 *)
Definition SubIntExpr (e1 e2: Expression): Expression :=
  EquivExprU (ImplicationU (LogicalExprU (RelationU (TermSub
      (TermU (FactorU (UnaryExprArray (ArrayExprU (Expr e1)))))
      (FactorU (UnaryExprArray (ArrayExprU (Expr e2))))
  )))).

(* e1 * e2 *)
Definition MulIntExpr (e1 e2: Expression): Expression :=
  EquivExprU (ImplicationU (LogicalExprU (RelationU (TermU (FactorMul
      (FactorU (UnaryExprArray (ArrayExprU (Expr e1))))
      (UnaryExprArray (ArrayExprU (Expr e2)))
  ))))).

(* e1 / e2 *)
Definition DivIntExpr (e1 e2: Expression): Expression :=
  EquivExprU (ImplicationU (LogicalExprU (RelationU (TermU (FactorDiv
      (FactorU (UnaryExprArray (ArrayExprU (Expr e1))))
      (UnaryExprArray (ArrayExprU (Expr e2)))
  ))))).

(* e1 % e2 *)
Definition RemIntExpr (e1 e2: Expression): Expression :=
  EquivExprU (ImplicationU (LogicalExprU (RelationU (TermU (FactorMod
      (FactorU (UnaryExprArray (ArrayExprU (Expr e1))))
      (UnaryExprArray (ArrayExprU (Expr e2)))
  ))))).
```

```
(* e1 == e2 *)
Definition EqualsExpr (e1 e2: Expression): Expression :=
  EquivExprU (ImplicationU (LogicalExprU (RelationEq
      (TermU (FactorU (UnaryExprArray (ArrayExprU (Expr e1)))))
      (TermU (FactorU (UnaryExprArray (ArrayExprU (Expr e2)))))
  ))).

(* e1 != e2 *)
Definition NotEqualsExpr (e1 e2: Expression): Expression :=
  EquivExprU (ImplicationU (LogicalExprU (RelationNe
      (TermU (FactorU (UnaryExprArray (ArrayExprU (Expr e1)))))
      (TermU (FactorU (UnaryExprArray (ArrayExprU (Expr e2)))))
  ))).

(* e1 ==> e2 *)
Definition ImpliesExpr (e1 e2: Expression): Expression :=
  EquivExprU (Implication
      (LogicalExprU (RelationU (TermU (FactorU (UnaryExprArray (ArrayExprU (Expr e1)))))))
      (ImplicationU (LogicalExprU (RelationU (TermU (FactorU (UnaryExprArray (ArrayExprU (Expr e2))))))))
  ).

(* e1 <: e2 *)
Definition PartialOrderExpr (e1 e2: Expression): Expression :=
  EquivExprU (ImplicationU (LogicalExprU (RelationPo
      (TermU (FactorU (UnaryExprArray (ArrayExprU (Expr e1)))))
      (TermU (FactorU (UnaryExprArray (ArrayExprU (Expr e2)))))
  ))).

(* func(args) *)
Definition FunctionCallExpr (func: Identifier) (args: list Expression) :=
  EquivExprU (ImplicationU (LogicalExprU (RelationU (TermU (FactorU
      (UnaryExprArray (ArrayExprU (FunctionCall func args)))
  ))))).

(* forall ids: e *)
Definition ForallExpr (ids: list Identifier ) (e: Expression) :=
  EquivExprU (ImplicationU (LogicalExprU (RelationU (TermU (FactorU
      (UnaryExprArray (ArrayExprU (Quant (Forall ids e))))
  ))))).

End BoogieHelper.
```

3.2 Formalization of the translation

The actual translation relies on the Bicolano formalization and is contained in the Translation library. The root of the translation is the Tr function and takes as argument a method and returns the translation as a BoogiePL procedure and implementation. The development is type-checkable so we have a guarantee that our translation generates valid (at least syntactically) BoogiePL code.

```
(* Formalization of the translation from Java bytecode to BoogiePL.
   Builds upon Bicolano and BoogiePL libraries *)

Require Import List.
Require Import ListSet.
Require Import ZArith.
```

```
(* Include the Bicolano library *)
Add LoadPath "../Bicolano".
Require Import Program.

(* Include the BoogiePL library *)
Require Import BoogiePL.
Require Import BoogieUtils.

Module Translation (Boogie: BOOGIEPL) (Prog: PROGRAM).
  Module BPL := BoogieHelper Boogie Prog.

  Import Prog.

  (* Bytecode verifier information *)

  (* The stack height at the specified PC *)
  Parameter sth: BytecodeMethod > PC > nat.

  (* The type on the stack at the specified height and PC
     Note: this is only used for athrow, to determine the type of exception on the stack. *)
  Parameter stt: BytecodeMethod > PC > nat > type.

  (* CFG information *)
  Parameter isBlockStart: BytecodeMethod > PC > bool.
  Parameter isEdge: BytecodeMethod > PC > PC > bool.

  (* Exceptions *)
  Parameter throws: MethodSignature > list ClassName.
  Parameter lookupHandler: BytecodeMethod > PC > ClassName > Boogie.Identifier.
  Parameter invocationExceptions: BytecodeMethod > PC > MethodSignature > list ClassName.

  (* Method specifications *)
  Inductive SpecType: Set := PreCond | PostCond | PreXCond (cn: ClassName) | PostXCond (cn: ClassName).

  (* Magic function which returns to us a first order logic BoogiePL expression which expresses
     a pre or postcondition of a method, dependent on a list of variables, e.g. the heap in its
     pre and poststate. *)
  Parameter getSpec: MethodSignature > SpecType > list Boogie.Identifier > Boogie.Expression.

  (* Procedure name (e.g. Sample.swap) *)
  Parameter procName: MethodName > Boogie.Identifier.

  (* Block identifiers *)
  Parameter PreCondBlock PostCondBlock InitBlock: Boogie.Identifier.
  Parameter blockPC: PC > Boogie.Identifier.

  (* Method continuation, None = normal, ClassName = exceptional termination *)
  Parameter blockPCE: PC > option ClassName > Boogie.Identifier.

  (* Block that asserts exceptional post condition *)
  Parameter blockPostX: ClassName > Boogie.Identifier.

  (* Continuation block for branches *)
  Parameter blockPCB: PC > bool > Boogie.Identifier.

  (* Translation environment: heap type and variables *)
  Parameters heapType: Boogie.Identifier.
  Parameters heapVar oldHeapVar preHeapVar: Boogie.Identifier.
```

```
(* Heap functions *)
Parameters h_get h_alive h_new h_update h_add: Boogie.Identifier.
Parameters h_fieldLoc h_arrayLoc h_toref h_rval h_toint h_ival h_typ h_typeObject: Boogie.Identifier.
Parameters h_objectAlloc h_arrayAlloc: Boogie.Identifier.

(* Type and field variables (e.g. java.lang.Exception, C.f) *)
Parameter fieldVar: FieldSignature > Boogie.Identifier.
Parameter typeVar: type > Boogie.Identifier.

(* Functions for bitwise operations *)
Parameters bo_and bo_or bo_xor bo_shl bo_shr bo_ushr: Boogie.Identifier.

(* Variables for stack, register and temporary variables *)
Parameter stvar: nat > Boogie.BPLType > Boogie.Identifier.
Parameter regvar: nat > Boogie.BPLType > Boogie.Identifier.
Parameter swapvar: Boogie.BPLType > Boogie.Identifier.
Parameter argvar: nat > Boogie.BPLType > Boogie.Identifier.

(* Variable used for quantification over heap values in method invocation *)
Parameter valueVar: Boogie.Identifier.

(* Placeholders *)
Parameter emptyBlock: Boogie.Block.
Parameter emptyToc: Boogie.TocManifesto.

(* Our simple type mapping *)
Definition TrType (t: type): Boogie.BPLType :=
  match t with
    | ReferenceType rt => Boogie.Ref
    | PrimitiveType pt => Boogie.Int
  end.

(* variables for operand stack *)
Fixpoint TrStackVars (n: nat) : list (Boogie.Identifier * Boogie.BPLType) :=
  match n with
    | O => nil
    | S p => (stvar p Boogie.Ref, Boogie.Ref)::
             (stvar p Boogie.Int, Boogie.Int)::
             TrStackVars p
  end.

(* variables for registers *)
Fixpoint TrRegVars (n: nat): list (Boogie.Identifier * Boogie.BPLType) :=
  match n with
    | O => nil
    | S p => (regvar p Boogie.Ref, Boogie.Ref)::
             (regvar p Boogie.Int, Boogie.Int)::
             TrRegVars p
  end.

(* temporary variables for swap instruction *)
Definition TrSwapVars: list (Boogie.Identifier * Boogie.BPLType) :=
  (swapvar Boogie.Ref, Boogie.Ref)::
  (swapvar Boogie.Int, Boogie.Int):: nil.


(* Declare all local variables *)
```

```
Definition TrVars (body: BytecodeMethod) : Boogie.LocalVarDecl :=
  TrStackVars (BYTECODEMETHOD.max_operand_stack_size body) ++
  TrRegVars (BYTECODEMETHOD.max_locals body) ++
  TrSwapVars ++
  (oldHeapVar, Boogie.UserDef heapType)::
  (preHeapVar, Boogie.UserDef heapType)::
  nil .

(* Method parameters and return variable *)
Parameter this: Boogie.Identifier.
Parameter param: nat > Boogie.Identifier.
Parameter result: Boogie.Identifier.

Fixpoint TrParams0 (params: list type) (i: nat) {struct params}
    : list (Boogie. Identifier * Boogie.BPLType) :=
  match params with
    | nil => nil
    | cons t l => (param i, TrType t)::TrParams0 l (S i)
  end.

(* "this" is implicit *)
Definition TrParams (m: Method): list (Boogie.Identifier * Boogie.BPLType) :=
  if METHOD.isStatic m then
    TrParams0 (METHODSIGNATURE.parameters (METHOD.signature m)) 0
  else
    (this , Boogie.Ref)::TrParams0 (METHODSIGNATURE.parameters (METHOD.signature m)) 0.

(* Procedure declaration *)
Definition TrProcDecl (cl:ClassName) (m: Method): Boogie.Declaration :=
  let ms := METHOD.signature m in
      Boogie.ProcedureDecl (procName (cl,METHODSIGNATURE.name ms))
          (TrParams m)
          match METHODSIGNATURE.result ms with
            | None => None
            | Some t => Some (TrType t)
          end
          (Boogie.Modifies (heapVar::nil ):: nil ).

(* Argument to register transfer *)
Fixpoint TrRegs (params: list type) (static: bool) (i: nat) {struct params}: list Boogie.Command :=
  match params with
    | nil => nil
    | t :: l =>
      let
        j := if static then i else (plus i 1)
      in
          (* assign register , shift by 1 if instance method *)
          Boogie.Assign (regvar j (TrType t)) (BPL.IdentExpr (param i))::
          match t with
            | ReferenceType rt =>
              match rt with
                | ClassType cl =>
                  (* assume typ(rval(param(i))) == [declaringClass] *)
                  Boogie.Assume (BPL.EqualsExpr
                    (BPL.FunctionCallExpr h_typ (
                      BPL.FunctionCallExpr h_rval (BPL.IdentExpr (param i)::nil)
                      :: nil ))
                    (BPL.IdentExpr (typeVar t)))::
```

```
                    (* assume alive(rval(param(i)), heap); *)
                    Boogie.Assume (BPL.FunctionCallExpr h_alive (
                      BPL.FunctionCallExpr h_rval (BPL.IdentExpr (param i)::nil)::
                      BPL.IdentExpr heapVar::nil)
                    )::nil
                | _ => nil
              end
            | _ => nil
          end
          ++
          TrRegs l static (plus i 1)
  end.

(* Initialization block *)
Definition TrInit (thisClass:ClassName) (m: Method): Boogie.Block :=
  let ms := METHOD.signature m in
      (InitBlock,

       if METHOD.isStatic m then
          (* assume this != null; *)
          Boogie.Assume (BPL.EqualsExpr (BPL.IdentExpr this) BPL.NullConstant)::
          (* assume typ(rval(this)) == [declaringClass] *)
          Boogie.Assume (BPL.EqualsExpr
              (BPL.FunctionCallExpr h_typ (
                  BPL.FunctionCallExpr h_rval (
                      BPL.IdentExpr this::nil
                )):: nil ))
              (BPL.IdentExpr (typeVar (ReferenceType (ClassType thisClass))))
          )::
          (* assume alive(rval(this), heap); *)
          Boogie.Assume (BPL.FunctionCallExpr h_alive (
              BPL.FunctionCallExpr h_rval (BPL.IdentExpr this::nil)::
              BPL.IdentExpr heapVar::nil)
          )::
          (* reg0r := this; *)
          Boogie.Assign (regvar 0 Boogie.Ref) (BPL.IdentExpr this)::
          TrRegs (METHODSIGNATURE.parameters ms) true 1
       else
          TrRegs (METHODSIGNATURE.parameters ms) false 0
       , Boogie.Goto (PreCondBlock::nil)).


(* Precondition block *)
Definition TrPreCondBlock (cl:ClassName) (m: Method) (body: BytecodeMethod): Boogie.Block :=
  (PreCondBlock,
   Boogie.Assume (getSpec (cl,METHOD.signature m) PreCond (heapVar::nil))::nil,
   Boogie.Goto (blockPC (BYTECODEMETHOD.firstAddress body)::nil)).

(* Exceptional postcondition blocks *)
Definition TrPostXCondBlock (cl:ClassName) (m: Method) (exc: ClassName): Boogie.Block :=
  (blockPostX exc,
   (Boogie.Assert (getSpec (cl,METHOD.signature m) (PostXCond exc) (heapVar::oldHeapVar::nil)))::nil,
   Boogie.Return).

Fixpoint TrPostXConds (cl:ClassName) (m: Method) (excs: list ClassName) {struct excs}
    : list Boogie.Block :=
  match excs with
    | nil => nil
```

```
    | cons exception l => TrPostXCondBlock cl m exception::TrPostXConds cl m l
  end.

(* Branch targets for tableswitch instruction *)
Fixpoint TrTableswitchTargets (pc: PC) (offsets: list OFFSET.t) {struct offsets}
    : list Boogie.Identifier :=
  match offsets with
    | nil => nil
    | o :: l => blockPC (OFFSET.jump pc o)::TrTableswitchTargets pc l
  end.

(* Branch targets for lookupswitch instruction *)
Fixpoint TrLookupswitchTargets (pc: PC) (offsets: list (Z * OFFSET.t)) {struct offsets}
    : list Boogie.Identifier :=
  match offsets with
    | nil => nil
    | (key,o):: l => blockPC (OFFSET.jump pc o)::TrLookupswitchTargets pc l
  end.

(* Generate the blocks that model an exceptional termination of a method invocation *)
Definition TrExceptionBlock (bm: BytecodeMethod) (pc: PC) (ms: MethodSignature)
  (exc: ClassName) : Boogie.Block :=
    (blockPCE pc (Some exc),
      (* havoc stack0r *)
      Boogie.Havoc (stvar O Boogie.Ref::nil)::
      (* assume alive(rval(stack0r), heap) *)
      Boogie.Assume (BPL.FunctionCallExpr h_alive (
          BPL.FunctionCallExpr h_rval (BPL.IdentExpr (stvar O Boogie.Ref)::nil)::
          BPL.IdentExpr heapVar::nil)
      )::
      (* assume typ(rval(stack0r)) <: cl *)
      Boogie.Assume (BPL.PartialOrderExpr
          (BPL.FunctionCallExpr h_typ (
              BPL.FunctionCallExpr h_rval (BPL.IdentExpr (stvar O Boogie.Ref)::nil)::
              nil
          ))
          (BPL.IdentExpr (typeVar (ReferenceType (ClassType exc))))
      )::
      (* assume PreCondX(exc) ==> PostCondX(exc) *)
      Boogie.Assume (BPL.ImpliesExpr
          (getSpec ms (PreXCond exc) (heapVar::nil))
          (getSpec ms (PostXCond exc) (heapVar::preHeapVar::nil))
      )::
      nil,
     Boogie.Goto (lookupHandler bm pc exc::nil)
    ).

Fixpoint TrExceptionBlocks (bm: BytecodeMethod) (pc: PC) (ms: MethodSignature)
    (excs: list ClassName) {struct excs}: list Boogie.Block :=
  match excs with
    | nil => nil
    | cl :: l => TrExceptionBlock bm pc ms cl :: TrExceptionBlocks bm pc ms l
  end.

(* Copy method arguments to local variables (to reason about old values) *)
(*  arg j <i|r> := stack (h  |parameters(ms)| +1 + j) <i|r> *)
Fixpoint TrCopyArgs (bm: BytecodeMethod) (pc: PC) (argc i: nat) {struct i}: list Boogie.Command :=
  let h := sth bm pc in
```

```
  match i with
    | O => nil
    | S p =>
        TrCopyArgs bm pc argc p++
        Boogie.Assign (argvar p Boogie.Int) (BPL.IdentExpr (stvar (h  argc + p + 1) Boogie.Int))::
        Boogie.Assign (argvar p Boogie.Ref) (BPL.IdentExpr (stvar (h  argc + p + 1) Boogie.Ref))::
        nil
  end.

(* Translate a modifies clause *)
Parameter TrModifies: BytecodeMethod > PC > MethodSignature > Boogie.Expression.

(* Return the top N stack variables, used to pass state to getSpec *)
Parameter stackvars: BytecodeMethod > PC > nat > list Boogie.Identifier.

(* Translate a method invocation (instance and static) *)
Definition TrMethodInvocation (bm: BytecodeMethod) (pc: PC) (ms: MethodSignature) (static: bool)
    (cur: Boogie.Block) : (list Boogie.Block * Boogie.Block) :=
  let argc := length (METHODSIGNATURE.parameters (snd ms)) in
  let h := sth bm pc in
  let params := stackvars bm pc argc in
  match cur with
    | (curBlockId,cmds,toc) =>
              let excBlocks := TrExceptionBlocks bm pc ms (invocationExceptions bm pc ms) in
                (
                 (curBlockId, cmds ++ (
                   (* preserve first argument (might be havoced if method has nonvoid return type *)
                   Boogie.Assign (argvar 0 Boogie.Ref) (BPL.IdentExpr (stvar (h  argc) Boogie.Ref))::
                   Boogie.Assign (argvar 0 Boogie.Int) (BPL.IdentExpr (stvar (h  argc) Boogie.Int))::
                   if (static) then
                       nil
                   else
                       (* assert arg0r != null; *)
                       Boogie.Assert (BPL.NotEqualsExpr
                          (BPL.IdentExpr (argvar 0 Boogie.Ref)) BPL.NullConstant)::nil
                   ++
                   (* assert precondition *)
                   Boogie.Assert (getSpec ms PreCond (heapVar::params))::
                   (* pre_heap := heap; *)
                   Boogie.Assign preHeapVar (BPL.IdentExpr (heapVar))::
                   (* havoc heap; *)
                   Boogie.Havoc (heapVar::nil)::
                   (* objects alive in old heap are also alive in new heap *)
                   Boogie.Assume (BPL.ForallExpr (valueVar::nil)
                       (BPL.ImpliesExpr
                           (BPL.FunctionCallExpr h_alive
                               (BPL.IdentExpr valueVar::BPL.IdentExpr preHeapVar::nil))
                           (BPL.FunctionCallExpr h_alive
                               (BPL.IdentExpr valueVar::BPL.IdentExpr heapVar::nil))
                       )
                   )::
                   (* modifies clause *)
                   Boogie.Assume (TrModifies bm pc ms)::
                   nil
                  ),
                 (* branch to normal and exceptional blocks *)
                 Boogie.Goto (blockPCE pc None::BPL.blockIds excBlocks))::
```

```
                    (* Exceptional blocks > see TrExceptionBlocks *)
                    excBlocks,

                    (* Normal execution block *)
                    (blockPCE pc None,
                       match METHODSIGNATURE.result (snd ms) with
                         | None => nil
                         | Some t =>
                           (* Nonvoid return type > havoc returned value *)
                           if static then
                               Boogie.Havoc (stvar (h argc+1) (TrType t)::nil ):: nil
                           else
                               Boogie.Havoc (stvar (h argc) (TrType t)::nil ):: nil
                       end ++
                        (* Assume Precondition ==> Postcondition *)
                        (Boogie.Assume (BPL.ImpliesExpr
                           (getSpec ms PreCond (heapVar::params))
                           (getSpec ms PostCond (heapVar::preHeapVar::params)))::
                         nil)
                     , emptyToc)
                   )
  end.

(* Translate a single JVM instruction *)
Definition TrInstruction (bm: BytecodeMethod) (pc: PC) (blocks: list Boogie.Block) (cur: Boogie.Block)
    : (list Boogie.Block * Boogie.Block) :=
  match BYTECODEMETHOD.instructionAt bm pc with
    | Some instr =>
      let h := sth bm pc in
      match instr with
        | Ibinop op => match cur with
            | (blockid,cmds,toc) => match op with
                (* Addition, substraction and multiplication: nothing special here *)
                | AddInt => (nil,(blockid, cmds++(
                    (Boogie.Assign (stvar (h  1) Boogie.Int) (BPL.AddIntExpr
                        (BPL.IdentExpr (stvar (h  1) Boogie.Int))
                        (BPL.IdentExpr (stvar h Boogie.Int)))
                     )::nil),toc))
                | SubInt => (nil,(blockid, cmds++(
                    (Boogie.Assign (stvar (h  1) Boogie.Int) (BPL.SubIntExpr
                        (BPL.IdentExpr (stvar (h  1) Boogie.Int))
                        (BPL.IdentExpr (stvar h Boogie.Int)))
                     )::nil),toc))
                | MulInt => (nil,(blockid, cmds++(
                    (Boogie.Assign (stvar (h  1) Boogie.Int) (BPL.MulIntExpr
                        (BPL.IdentExpr (stvar (h  1) Boogie.Int))
                        (BPL.IdentExpr (stvar h Boogie.Int)))
                     )::nil),toc))
                (* Division, remainder: Assertions handle division by zero *)
                | DivInt => (nil, (blockid,cmds++(
                    (Boogie.Assert (BPL.NotEqualsExpr
                        (BPL.IdentExpr (stvar h Boogie.Int))
                        (BPL.IntConstant 0)))::
                     Boogie.Assign (stvar (h  1) Boogie.Int) (BPL.DivIntExpr
                        (BPL.IdentExpr (stvar (h  1) Boogie.Int))
                        (BPL.IdentExpr (stvar h Boogie.Int)))::
                     nil
                    ), toc))
```

```
        | RemInt => (nil, (blockid,cmds++(
            (Boogie.Assert (BPL.NotEqualsExpr
                (BPL.IdentExpr (stvar h Boogie.Int))
                (BPL.IntConstant 0)))::
           Boogie.Assign (stvar (h  1) Boogie.Int) (BPL.RemIntExpr
                (BPL.IdentExpr (stvar (h  1) Boogie.Int))
                (BPL.IdentExpr (stvar h Boogie.Int)))::
           nil
          ), toc))
        (* Bitwise operations: BoogiePL function calls *)
        | ShlInt => (nil, (blockid,cmds++(
            (Boogie.Assign (stvar (h  1) Boogie.Int) (BPL.FunctionCallExpr bo_shl
                (BPL.IdentExpr (stvar (h  1) Boogie.Int)::
                 BPL.IdentExpr (stvar h Boogie.Int)::nil)
          )):: nil,toc))
        | ShrInt => (nil, (blockid,cmds++(
            (Boogie.Assign (stvar (h  1) Boogie.Int) (BPL.FunctionCallExpr bo_shr
                (BPL.IdentExpr (stvar (h  1) Boogie.Int)::
                 BPL.IdentExpr (stvar h Boogie.Int)::nil)
          )):: nil,toc))
        | AndInt => (nil, (blockid,cmds++(
            (Boogie.Assign (stvar (h  1) Boogie.Int) (BPL.FunctionCallExpr bo_and
                (BPL.IdentExpr (stvar (h  1) Boogie.Int)::
                 BPL.IdentExpr (stvar h Boogie.Int)::nil)
          )):: nil,toc))
        | OrInt => (nil, (blockid,cmds++(
            (Boogie.Assign (stvar (h  1) Boogie.Int) (BPL.FunctionCallExpr bo_or
                (BPL.IdentExpr (stvar (h  1) Boogie.Int)::
                 BPL.IdentExpr (stvar h Boogie.Int)::nil)
          )):: nil,toc))
        | UshrInt => (nil, (blockid,cmds++(
            (Boogie.Assign (stvar (h  1) Boogie.Int) (BPL.FunctionCallExpr bo_ushr
                (BPL.IdentExpr (stvar (h  1) Boogie.Int)::
                 BPL.IdentExpr (stvar h Boogie.Int)::nil)
          )):: nil,toc))
        | XorInt => (nil, (blockid,cmds++(
            (Boogie.Assign (stvar (h  1) Boogie.Int) (BPL.FunctionCallExpr bo_xor
                (BPL.IdentExpr (stvar (h  1) Boogie.Int)::
                 BPL.IdentExpr (stvar h Boogie.Int)::nil)
          )):: nil,toc))
      end
    end
  | Ineg => match cur with
      | (blockid,cmds,toc) => (nil, (blockid,cmds++(
          Boogie.Assign (stvar h Boogie.Int)
              (BPL.UnaryMinusExpr (BPL.IdentExpr (stvar h Boogie.Int)))
      ):: nil,toc))
    end
  | Iinc i z => match cur with
      | (blockid,cmds,toc) => (nil, (blockid, cmds++(
          Boogie.Assign (regvar (Var_toN i) Boogie.Int)
              (BPL.AddIntExpr (BPL.IdentExpr (regvar (Var_toN i) Boogie.Int))
                              (BPL.IntConstant z))
      ):: nil,toc))
    end
  | Dup =>
      match cur with
        | (blockid,cmds,toc) => (nil, (blockid,cmds++(
```

```
            (Boogie.Assign (stvar (h + 1) Boogie.Ref) (BPL.IdentExpr (stvar h Boogie.Ref)))::
            (Boogie.Assign (stvar (h + 1) Boogie.Int) (BPL.IdentExpr (stvar h Boogie.Int)))::
            nil
        ),toc))
    end
| Dup_x1 =>
    match cur with
      | (blockid,cmds,toc) => (nil, (blockid,cmds++
            Boogie.Assign (stvar (h+1) Boogie.Ref) (BPL.IdentExpr (stvar h Boogie.Ref))::
            Boogie.Assign (stvar (h+1) Boogie.Int) (BPL.IdentExpr (stvar h Boogie.Int))::
            Boogie.Assign (stvar h Boogie.Ref) (BPL.IdentExpr (stvar (h1) Boogie.Ref))::
            Boogie.Assign (stvar h Boogie.Int) (BPL.IdentExpr (stvar (h1) Boogie.Int ))::
            Boogie.Assign (stvar (h 1) Boogie.Ref) (BPL.IdentExpr (stvar (h+1) Boogie.Ref))::
            Boogie.Assign (stvar (h 1) Boogie.Int) (BPL.IdentExpr (stvar (h+1) Boogie.Int))::
            nil
        ,toc))
    end
| Dup_x2 =>
    match cur with
      | (blockid,cmds,toc) => (nil, (blockid,cmds++
          Boogie.Assign (stvar (h+1) Boogie.Ref) (BPL.IdentExpr (stvar h Boogie.Ref))::
          Boogie.Assign (stvar (h+1) Boogie.Int) (BPL.IdentExpr (stvar h Boogie.Int))::
          Boogie.Assign (stvar h Boogie.Ref) (BPL.IdentExpr (stvar (h1) Boogie.Ref))::
          Boogie.Assign (stvar h Boogie.Int) (BPL.IdentExpr (stvar (h1) Boogie.Int ))::
          Boogie.Assign (stvar (h 1) Boogie.Ref) (BPL.IdentExpr (stvar (h2) Boogie.Ref))::
          Boogie.Assign (stvar (h 1) Boogie.Int) (BPL.IdentExpr (stvar (h2) Boogie.Int ))::
          Boogie.Assign (stvar (h 2) Boogie.Ref) (BPL.IdentExpr (stvar (h+1) Boogie.Ref))::
          Boogie.Assign (stvar (h 2) Boogie.Int) (BPL.IdentExpr (stvar (h+1) Boogie.Int))::
          nil
        ,toc))
    end
| Dup2 =>
    match cur with
      | (blockid,cmds,toc) => (nil, (blockid,cmds++
          Boogie.Assign (stvar (h+2) Boogie.Ref) (BPL.IdentExpr (stvar h Boogie.Ref))::
          Boogie.Assign (stvar (h+2) Boogie.Int) (BPL.IdentExpr (stvar h Boogie.Int))::
          Boogie.Assign (stvar (h+1) Boogie.Ref) (BPL.IdentExpr (stvar (h1) Boogie.Ref))::
          Boogie.Assign (stvar (h+1) Boogie.Int) (BPL.IdentExpr (stvar (h1) Boogie.Int ))::
          nil
        ,toc))
    end
| Dup2_x1 =>
    match cur with
      | (blockid,cmds,toc) => (nil, (blockid,cmds++
          Boogie.Assign (stvar (h+2) Boogie.Ref) (BPL.IdentExpr (stvar h Boogie.Ref))::
          Boogie.Assign (stvar (h+2) Boogie.Int) (BPL.IdentExpr (stvar h Boogie.Int))::
          Boogie.Assign (stvar (h+1) Boogie.Ref) (BPL.IdentExpr (stvar (h1) Boogie.Ref))::
          Boogie.Assign (stvar (h+1) Boogie.Int) (BPL.IdentExpr (stvar (h1) Boogie.Int ))::
          Boogie.Assign (stvar h Boogie.Ref) (BPL.IdentExpr (stvar (h2) Boogie.Ref))::
          Boogie.Assign (stvar h Boogie.Int) (BPL.IdentExpr (stvar (h2) Boogie.Int ))::
          Boogie.Assign (stvar (h 1) Boogie.Ref) (BPL.IdentExpr (stvar (h+2) Boogie.Ref))::
          Boogie.Assign (stvar (h 1) Boogie.Int) (BPL.IdentExpr (stvar (h+2) Boogie.Int))::
          Boogie.Assign (stvar (h 2) Boogie.Ref) (BPL.IdentExpr (stvar (h+1) Boogie.Ref))::
          Boogie.Assign (stvar (h 2) Boogie.Int) (BPL.IdentExpr (stvar (h+1) Boogie.Int))::
          nil
        ,toc))
    end
| Dup2_x2 =>
```

```
    match cur with
      | (blockid,cmds,toc) => (nil, (blockid,cmds++
          Boogie.Assign (stvar (h+2) Boogie.Ref) (BPL.IdentExpr (stvar h Boogie.Ref))::
          Boogie.Assign (stvar (h+2) Boogie.Int) (BPL.IdentExpr (stvar h Boogie.Int))::
          Boogie.Assign (stvar (h+1) Boogie.Ref) (BPL.IdentExpr (stvar (h1) Boogie.Ref))::
          Boogie.Assign (stvar (h+1) Boogie.Int) (BPL.IdentExpr (stvar (h1) Boogie.Int ))::
          Boogie.Assign (stvar h Boogie.Ref) (BPL.IdentExpr (stvar (h2) Boogie.Ref))::
          Boogie.Assign (stvar h Boogie.Int) (BPL.IdentExpr (stvar (h2) Boogie.Int ))::
          Boogie.Assign (stvar (h 1) Boogie.Ref) (BPL.IdentExpr (stvar (h3) Boogie.Ref))::
          Boogie.Assign (stvar (h 1) Boogie.Int) (BPL.IdentExpr (stvar (h3) Boogie.Int ))::
          Boogie.Assign (stvar (h 2) Boogie.Ref) (BPL.IdentExpr (stvar (h+2) Boogie.Ref))::
          Boogie.Assign (stvar (h 2) Boogie.Int) (BPL.IdentExpr (stvar (h+2) Boogie.Int))::
          Boogie.Assign (stvar (h 3) Boogie.Ref) (BPL.IdentExpr (stvar (h+1) Boogie.Ref))::
          Boogie.Assign (stvar (h 3) Boogie.Int) (BPL.IdentExpr (stvar (h+1) Boogie.Int))::
          nil
        ,toc))
    end
| Swap =>
    match cur with
      | (blockid,cmds,toc) => (nil,
          (blockid, cmds++(
            Boogie.Assign (swapvar Boogie.Ref) (BPL.IdentExpr (stvar (h1) Boogie.Ref))::
            Boogie.Assign (swapvar Boogie.Int) (BPL.IdentExpr (stvar (h1) Boogie.Int ))::
            Boogie.Assign (stvar (h 1) Boogie.Ref) (BPL.IdentExpr (stvar h Boogie.Ref))::
            Boogie.Assign (stvar (h 1) Boogie.Int) (BPL.IdentExpr (stvar h Boogie.Int))::
            Boogie.Assign (stvar h Boogie.Ref) (BPL.IdentExpr (swapvar Boogie.Ref))::
            Boogie.Assign (stvar h Boogie.Int) (BPL.IdentExpr (swapvar Boogie.Int))::
            nil
          ), toc)
        )
    end
| Pop => (nil,cur)
| Pop2 => (nil,cur)
| Getfield fs => match cur with
    | (blockid,cmds,toc) => (nil,
        (blockid, cmds++
          Boogie.Assert (BPL.NotEqualsExpr
              (BPL.IdentExpr (stvar h Boogie.Ref))
              BPL.NullConstant
          )::
          match FIELDSIGNATURE.type (snd fs) with
            | PrimitiveType pt =>
              (* stack(h)i := toint(get(heap, fieldLoc(stack(h)r, fs))) *)
              Boogie.Assign (stvar h Boogie.Int) (BPL.FunctionCallExpr h_toint
                (BPL.FunctionCallExpr h_get
                  (BPL.IdentExpr heapVar::
                   BPL.FunctionCallExpr h_fieldLoc
                     (BPL.IdentExpr (stvar h Boogie.Ref)::
                      BPL.IdentExpr (fieldVar fs):: nil )::
                    nil )::
                  nil
                )):: nil
            | ReferenceType rt => match rt with
                | ClassType cl =>
                  (* stack(h)r := toref(get(heap, fieldLoc(stack(h)r, fs))) *)
                  Boogie.Assign (stvar h Boogie.Ref) (BPL.FunctionCallExpr h_toref
                    (BPL.FunctionCallExpr h_get
                      (BPL.IdentExpr heapVar::
```

```
                                BPL.FunctionCallExpr h_fieldLoc (
                                  BPL.IdentExpr (stvar h Boogie.Ref)::
                                  BPL.IdentExpr (fieldVar fs):: nil )::
                                 nil )::
                                nil
                               )):: nil
                          | _ => nil (* not supported *)
                        end
                    end
                    , toc)
            )
        end
    | Putfield fs => match cur with
        | (blockid,cmds,toc) => (nil,
              (blockid, cmds++
               Boogie.Assert (BPL.NotEqualsExpr
                   (BPL.IdentExpr (stvar (h1) Boogie.Ref))
                   (BPL.NullConstant)
               )::
               match FIELDSIGNATURE.type (snd fs) with
                 | PrimitiveType pt =>
                   (* heap := update(heap, fieldLoc(stack(h 1) r, fs), ival(stack(h)i)); *)
                   Boogie.Assign heapVar (BPL.FunctionCallExpr h_update
                     (BPL.IdentExpr heapVar::
                      BPL.FunctionCallExpr h_fieldLoc (
                        BPL.IdentExpr (stvar (h1) Boogie.Ref)::
                        BPL.IdentExpr (fieldVar fs):: nil )::
                      BPL.FunctionCallExpr h_ival (BPL.IdentExpr (stvar h Boogie.Int)::nil)
                       :: nil)
                     ):: nil
                 | ReferenceType rt => match rt with
                   | ClassType cl =>
                     (* heap := update(heap, fieldLoc(stack(h 1) r, fs), rval(stack(h)r)); *)
                     Boogie.Assign heapVar (BPL.FunctionCallExpr h_update
                       (BPL.IdentExpr heapVar::
                        BPL.FunctionCallExpr h_fieldLoc (
                          BPL.IdentExpr (stvar (h1) Boogie.Ref)::
                          BPL.IdentExpr (fieldVar fs):: nil )::
                        BPL.FunctionCallExpr h_rval (BPL.IdentExpr (stvar h Boogie.Int)::nil)
                         :: nil)
                       ):: nil
                   | _ => nil (* not supported *)
                 end
               end
               , toc)
        )
    end
| Getstatic fs => match cur with
    | (blockid,cmds,toc) =>
        match fs with
          | (className, sfs) =>
            let fieldName := FIELDSIGNATURE.name sfs in

          (nil,
          (blockid, cmds++
           match FIELDSIGNATURE.type sfs with
             | PrimitiveType pt =>
               (* stack(h+1)i := toint(get(heap, fieldLoc(typeObject(fs), fs)))) *)
```

```
                Boogie.Assign (stvar (h+1) Boogie.Int) (BPL.FunctionCallExpr h_toint
                  (BPL.FunctionCallExpr h_get
                    (BPL.IdentExpr heapVar::
                     BPL.FunctionCallExpr h_fieldLoc
                       (BPL.FunctionCallExpr h_typeObject
                         (BPL.IdentExpr (typeVar (ReferenceType (ClassType className)))::nil)::
                        BPL.IdentExpr (fieldVar fs):: nil )::
                      nil )::
                    nil
                   )):: nil
              | ReferenceType rt => match rt with
                  | ClassType cl =>
                    (* stack(h+1)r := toref(get(heap, fieldLoc (typeoObject(fs), fs)))) *)
                    Boogie.Assign (stvar (h+1) Boogie.Ref) (BPL.FunctionCallExpr h_toref
                      (BPL.FunctionCallExpr h_get
                        (BPL.IdentExpr heapVar::
                         BPL.FunctionCallExpr h_fieldLoc
                           (BPL.FunctionCallExpr h_typeObject
                             (BPL.IdentExpr (typeVar (ReferenceType (ClassType className)))::nil)::
                            BPL.IdentExpr (fieldVar fs):: nil )::
                          nil )::
                        nil
                       )):: nil
                  | _ => nil (* not supported *)
                end
            end
            , toc)
        )
      end
    end
| Putstatic fs => match cur with
    | (blockid,cmds,toc) =>
      match fs with
        | (className, sfs) => let fieldName := FIELDSIGNATURE.name sfs in
        (nil,
        (blockid, cmds++

         match FIELDSIGNATURE.type sfs with
           | PrimitiveType pt =>
             (* heap := update(heap, instvar(typeObject(fs), fs), ival(stack(h)i)); *)
             Boogie.Assign heapVar (BPL.FunctionCallExpr h_update
               (BPL.IdentExpr heapVar::
                BPL.FunctionCallExpr h_fieldLoc (
                  BPL.FunctionCallExpr h_typeObject
                    (BPL.IdentExpr (typeVar (ReferenceType (ClassType className)))::nil)::
                  BPL.IdentExpr (fieldVar fs):: nil )::
                BPL.FunctionCallExpr h_ival (BPL.IdentExpr (stvar h Boogie.Int)::nil)
                :: nil)
              :: nil
           | ReferenceType rt => match rt with
             | ClassType cl =>
               (* heap := update(heap, instvar(typeObject(fs), fs), rval(stack(h)r)); *)
               Boogie.Assign heapVar (BPL.FunctionCallExpr h_update
                 (BPL.IdentExpr heapVar::
                  BPL.FunctionCallExpr h_fieldLoc (
                    BPL.FunctionCallExpr h_typeObject
                      (BPL.IdentExpr (typeVar (ReferenceType (ClassType className)))::nil)::
                    BPL.IdentExpr (fieldVar fs):: nil )::
```

```
                    BPL.FunctionCallExpr h_rval (BPL.IdentExpr (stvar h Boogie.Int)::nil)
                    :: nil )
                  ):: nil
                | _ => nil (* not supported *)
              end
            end
            , toc)
        )
      end
    end
| Vaload k => match cur with
    | (blockid,cmds,toc) => match k with
      | Aarray => (* reference arrays *)
        (nil, (blockid,cmds++(
          Boogie.Assert (BPL.NotEqualsExpr
              (BPL.IdentExpr (stvar (h1) Boogie.Ref))
              BPL.NullConstant
          )::
          Boogie.Assign (stvar (h 1) Boogie.Ref) (BPL.FunctionCallExpr h_toint
              (BPL.FunctionCallExpr h_get
                (BPL.IdentExpr heapVar::
                 BPL.FunctionCallExpr h_arrayLoc
                   (BPL.IdentExpr (stvar (h1) Boogie.Ref)::
                    BPL.IdentExpr (stvar h Boogie.Int)::nil )::
                 nil )::
               nil
              )):: nil
        ),toc))
      | _ => (* primitive arrays *)
        (nil, (blockid,cmds++(
          Boogie.Assert (BPL.NotEqualsExpr
              (BPL.IdentExpr (stvar (h1) Boogie.Ref))
              BPL.NullConstant
          )::
          Boogie.Assign (stvar (h 1) Boogie.Int) (BPL.FunctionCallExpr h_toint
              (BPL.FunctionCallExpr h_get
                (BPL.IdentExpr heapVar::
                 BPL.FunctionCallExpr h_arrayLoc
                   (BPL.IdentExpr (stvar (h1) Boogie.Ref)::
                    BPL.IdentExpr (stvar h Boogie.Int)::nil )::
                 nil )::
               nil
              )):: nil
        ),toc))
    end
  end
| Vastore k => match cur with
    | (blockid,cmds,toc) => match k with
      | Aarray => (* reference arrays *)
        (nil, (blockid,cmds++(
          Boogie.Assert (BPL.NotEqualsExpr
              (BPL.IdentExpr (stvar (h2) Boogie.Ref))
              BPL.NullConstant
          )::
          (* heap := update(heap, arrayLoc(stack(h 2) r, stack(h 1) r), ival(stack(h)i)); *)
          Boogie.Assign heapVar (BPL.FunctionCallExpr h_update
            (BPL.IdentExpr heapVar::
             BPL.FunctionCallExpr h_arrayLoc (
```

```
                BPL.IdentExpr (stvar (h2) Boogie.Ref)::
                BPL.IdentExpr (stvar (h1) Boogie.Int ):: nil )::
              BPL.FunctionCallExpr h_rval (BPL.IdentExpr (stvar h Boogie.Ref)::nil)
              :: nil )
            ):: nil
          ),toc))
      | _ => (* primitive arrays *)
        (nil, (blockid,cmds++(
          Boogie.Assert (BPL.NotEqualsExpr
              (BPL.IdentExpr (stvar (h2) Boogie.Ref))
              BPL.NullConstant
          )::
          (* heap := update(heap, arrayLoc(stack(h2)r, stack(h1)i), ival(stackhi)); *)
          Boogie.Assign heapVar (BPL.FunctionCallExpr h_update
            (BPL.IdentExpr heapVar::
            BPL.FunctionCallExpr h_arrayLoc (
              BPL.IdentExpr (stvar (h2) Boogie.Ref)::
              BPL.IdentExpr (stvar (h1) Boogie.Int ):: nil )::
            BPL.FunctionCallExpr h_ival (BPL.IdentExpr (stvar h Boogie.Int)::nil)
            :: nil )
          )):: nil
        ),toc))
    end
  end
| New cl => match cur with
    | (blockid,cmds,toc) => (nil,
        (blockid,cmds++(
          (* havoc stack(h+1)r *)
          Boogie.Havoc (stvar (h+1) Boogie.Ref::nil )::
          (* assume rval(stack(h+1)r) == new(heap, objectAlloc(cl)) *)
          Boogie.Assume (BPL.EqualsExpr
            (BPL.FunctionCallExpr h_rval
              (BPL.IdentExpr (stvar (h+1) Boogie.Ref)::nil))
            (BPL.FunctionCallExpr h_new
              (BPL.IdentExpr heapVar::
              BPL.FunctionCallExpr h_objectAlloc
                (BPL.IdentExpr (typeVar (ReferenceType (ClassType cl)))::nil)::
              nil)
            ))::
          (* heap := add(heap, objectAlloc(cl)) *)
          Boogie.Assign heapVar (BPL.FunctionCallExpr h_add
              (BPL.IdentExpr heapVar::
              BPL.FunctionCallExpr h_objectAlloc
                (BPL.IdentExpr (typeVar (ReferenceType (ClassType cl)))::nil)::
              nil ))::
          nil
        ), toc)
    )
  end
| Newarray t => match cur with
    | (blockid,cmds,toc) => (nil,
        (blockid,cmds++(
          (* havoc stack(h)r *)
          Boogie.Havoc (stvar h Boogie.Ref:: nil )::
          (* assume rval(stackhr) == new(heap, arrayAlloc(cl, stackhi)) *)
          Boogie.Assume (BPL.EqualsExpr
            (BPL.FunctionCallExpr h_rval
              (BPL.IdentExpr (stvar (h+1) Boogie.Ref)::nil))
```

```
                    (BPL.FunctionCallExpr h_new
                      (BPL.IdentExpr heapVar::
                       BPL.FunctionCallExpr h_arrayAlloc
                         (BPL.IdentExpr (typeVar t)::
                          BPL.IdentExpr (stvar h Boogie.Int)::
                          nil )::
                       nil )
                     ))::
                  (* heap := add(heap, arrayAlloc(cl, stackhi)) *)
                  Boogie.Assign heapVar (BPL.FunctionCallExpr h_add
                      (BPL.IdentExpr heapVar::
                       BPL.FunctionCallExpr h_arrayAlloc
                         (BPL.IdentExpr (typeVar t)::
                          BPL.IdentExpr (stvar h Boogie.Int)::
                          nil )::
                       nil ))::
                  nil
              ), toc)
      )
  end
| If_acmp cmp o => match cur with
    | (blockid,cmds,toc) =>
      (* finish current block with TocManifesto *)
      ((blockid,cmds, Boogie.Goto (blockPCB pc true::blockPCB pc false::nil))::
      (* new block that assumes condition holds *)
       (blockPCB pc true,
        Boogie.Assume (BPL.CompRefExpr
            (BPL.IdentExpr (stvar (h  1) Boogie.Ref))
            (BPL.IdentExpr (stvar h Boogie.Ref))
            cmp)::nil,
        Boogie.Goto (blockPC (OFFSET.jump pc o)::nil))::nil,
      (* new current block assumes condition does not hold *)
       (blockPCB pc false,
        Boogie.Assume (BPL.UnaryNotExpr (BPL.CompRefExpr
            (BPL.IdentExpr (stvar (h  1) Boogie.Ref))
            (BPL.IdentExpr (stvar h Boogie.Ref))
            cmp))::nil,
        emptyToc))
  end
| If_icmp cmp o => match cur with (* finish current block with TocManifesto *)
    | (blockid,cmds,toc) =>
      ( (blockid,cmds, Boogie.Goto (blockPCB pc true::blockPCB pc false::nil))::
        (* new block that assumes condition holds *)
        (blockPCB pc true, Boogie.Assume (BPL.CompIntExpr
            (BPL.IdentExpr (stvar (h  1) Boogie.Int))
            (BPL.IdentExpr (stvar h Boogie.Int))
            cmp)::nil,
         Boogie.Goto (blockPC (OFFSET.jump pc o)::nil))::nil,
        (* new current block assumes condition does not hold *)
        (blockPCB pc false,
         Boogie.Assume (BPL.UnaryNotExpr (BPL.CompIntExpr
             (BPL.IdentExpr (stvar (h  1) Boogie.Int))
             (BPL.IdentExpr (stvar h Boogie.Int))
             cmp))::nil,
         emptyToc))
  end
| If0 cmp o => match cur with (* finish current block with TocManifesto *)
    | (blockid,cmds,toc) =>
```

```
        ( (blockid,cmds, Boogie.Goto (blockPCB pc true::blockPCB pc false::nil))::
          (* new block that assumes condition holds *)
          (blockPCB pc true, Boogie.Assume (BPL.CompIntExpr
              (BPL.IdentExpr (stvar (h  1) Boogie.Int))
              (BPL.IntConstant 0)
              cmp)::nil,
           Boogie.Goto (blockPC (OFFSET.jump pc o)::nil))::nil,
          (* new current block assumes condition does not hold *)
          (blockPCB pc false,
           Boogie.Assume (BPL.UnaryNotExpr (BPL.CompIntExpr
              (BPL.IdentExpr (stvar (h  1) Boogie.Int))
              (BPL.IntConstant 0)
              cmp))::nil,
           emptyToc))
  end
| Ifnull  cmp o => match cur with (* finish current block with TocManifesto *)
    | (blockid,cmds,toc) =>
        ( (blockid,cmds, Boogie.Goto (blockPCB pc true::blockPCB pc false::nil))::
          (* new block that assumes condition holds *)
          (blockPCB pc true, Boogie.Assume (BPL.CompRefExpr
              (BPL.IdentExpr (stvar h Boogie.Ref))
              BPL.NullConstant
              cmp)::nil,
            Boogie.Goto (blockPC (OFFSET.jump pc o)::nil))::nil,
          (* new current block assumes condition does not hold *)
          (blockPCB pc false,
           Boogie.Assume (BPL.UnaryNotExpr (BPL.CompRefExpr
              (BPL.IdentExpr (stvar h Boogie.Ref))
              BPL.NullConstant
              cmp))::nil,
           emptyToc))
  end
| Goto o => match cur with
    | (blockid,cmds,toc) => (nil, (blockid,cmds,
                                   Boogie.Goto (blockPC (OFFSET.jump pc o)::nil)))
  end
| Vreturn kind => match cur with
    | (blockid,cmds,toc) => match kind with
        | Ival  => (nil, (blockid,cmds++
            (Boogie.Assign result  (BPL.IdentExpr (stvar h Boogie.Int)))::nil,
            Boogie.Goto (PostCondBlock::nil)))
        | Aval => (nil, (blockid,cmds++
            (Boogie.Assign result  (BPL.IdentExpr (stvar h Boogie.Ref)))::nil,
            Boogie.Goto (PostCondBlock::nil)))
    end
  end
| Invokevirtual  ms => TrMethodInvocation bm pc ms false cur
| Invokespecial  ms => TrMethodInvocation bm pc ms false cur
| Invokestatic  ms => TrMethodInvocation bm pc ms true cur
| Tableswitch def low high l  => match cur with
    | (blockid,cmds,toc) => (nil, (blockid,cmds,
                                   Boogie.Goto (blockPC (OFFSET.jump pc def)::
                                                TrTableswitchTargets pc l)))
  end
| Lookupswitch def l => match cur with
    | (blockid,cmds,toc) => (nil, (blockid,cmds,
                                   Boogie.Goto (blockPC (OFFSET.jump pc def)::
                                                TrLookupswitchTargets pc l)))
```

```
    end
  | Athrow => match cur with
      | (blockid,cmds,toc) => match stt bm pc h with
          | ReferenceType rt => match rt with
              | ClassType cl => (nil, (blockid,cmds++
                  (* assert stack(h)r != null; *)
                  (Boogie.Assert (BPL.NotEqualsExpr
                      (BPL.IdentExpr (stvar h Boogie.Ref))
                      (BPL.NullConstant)
                   )::
                   (* stack0r := stackhr; *)
                   Boogie.Assign (stvar O Boogie.Ref) (BPL.IdentExpr (stvar h Boogie.Ref))::
                   nil)
                  , Boogie.Goto (lookupHandler bm pc cl::nil)))
              | _ => (nil, cur) (* Shouldn't happen *)
            end
          | PrimitiveType _ => (nil, cur) (* Shouldn't happen *)
        end
    end
  | Vload kind n => match cur with (* load value from register *)
      | (blockid,cmds,toc) => match kind with
          | Ival => (nil, (blockid, cmds++
                      (Boogie.Assign (stvar (h + 1) Boogie.Int)
                          (BPL.IdentExpr (regvar (Var_toN n) Boogie.Int)))::nil
                     ,toc))
          | Aval => (nil, (blockid, cmds++
                      (Boogie.Assign (stvar (h + 1) Boogie.Ref)
                          (BPL.IdentExpr (regvar (Var_toN n) Boogie.Ref)))::nil
                     ,toc))
        end
    end
  | Vstore kind n => match cur with (* store value into register *)
      | (blockid,cmds,toc) => match kind with
          | Ival => (nil, (blockid, cmds++
                      (Boogie.Assign (regvar (Var_toN n) Boogie.Int)
                          (BPL.IdentExpr (stvar h Boogie.Int)) ):: nil
                     ,toc))
          | Aval => (nil, (blockid, cmds++
                      (Boogie.Assign (regvar (Var_toN n) Boogie.Ref)
                          (BPL.IdentExpr (stvar h Boogie.Ref)) ):: nil
                     ,toc))
        end
    end
  | Const pt z => match cur with (* push an int constant *)
      | (blockid,cmds,toc) => (nil, (blockid, cmds++
                                  (Boogie.Assign (stvar (h + 1) Boogie.Int)
                                      (BPL.IntConstant z))::nil
                                 ,toc))
    end
  | Aconst_null => match cur with (* push the null reference *)
      | (blockid,cmds,toc) => (nil, (blockid, cmds++
                                  (Boogie.Assign (stvar (h + 1) Boogie.Ref)
                                      BPL.NullConstant)::nil
                                 ,toc))
    end
  | Nop => (nil, cur) (* do nothing *)
  | _ => (nil, cur) (* not supported *)
end
```

```
    | None => (nil, cur)
  end.

(* This is the fixpoint iteration that translates a bytecode method to a list of
   BoogiePL blocks. There are two case distinctions:
     a) is this PC the start of a CFG block?
     b) is this the last instruction *)
Fixpoint TrInstructions (bm: BytecodeMethod) (pcs: list PC) (blocks: list Boogie.Block)
    (cur: Boogie.Block) {struct pcs}: list Boogie.Block :=
  match pcs with
    | nil => nil
    | pc::l =>
      match l with
        | nil => (* this is the last instruction, should always be a return *)
          if isBlockStart bm pc then
            match TrInstruction bm pc blocks (blockPC pc, nil, emptyToc) with
              | (blocks', cur') => blocks ++ (cur::nil) ++ blocks' ++ (cur'::nil)
            end
          else (* we are not starting a new block *)
            match TrInstruction bm pc blocks cur with
              | (blocks', cur') => blocks ++ blocks' ++ (cur'::nil)
            end
        | pc'::_ =>
          if isBlockStart bm pc then
            if isEdge bm pc pc' then (* CFG edge for pc > pc', end the block and add TocManifesto *)
              match TrInstruction bm pc blocks (blockPC pc, nil, emptyToc) with
                | (blocks', cur') =>
                  match cur with
                    | (blockid, cmds, toc) => TrInstructions bm l
                        (blocks++((blockid,cmds,(Boogie.Goto ((blockPC pc')::nil)))::nil)++blocks')
                        cur'
                  end
              end
            else
              match TrInstruction bm pc blocks (blockPC pc, nil, emptyToc) with
                | (blocks', cur') => TrInstructions bm l (blocks++(cur::nil)++blocks') cur'
              end
          else (* we are not starting a new block *)
            if isEdge bm pc pc' then (* CFG edge for pc > pc', end the block and add TocManifesto *)
              match TrInstruction bm pc blocks cur with
                | (blocks', cur') =>
                  match cur' with
                    | (blockid, cmds, toc) => TrInstructions bm l
                        (blocks++blocks'++(cur::nil))
                        (blockid, cmds, (Boogie.Goto ((blockPC pc')::nil)))
                  end
              end
            else
              match TrInstruction bm pc blocks cur with
                | (blocks', cur') => TrInstructions bm l (blocks++blocks') cur'
              end
      end
  end.

(* Helper, sequential list of all PCs. *)
Parameter listPC: BytecodeMethod > list PC.

(* Translate a method body *)
```

```
Definition TrBody (cn:ClassName) (m: Method) (body: BytecodeMethod): list Boogie.Block :=
  (TrPreCondBlock cn m body::
   TrInstructions body (listPC body) nil emptyBlock) ++
   TrPostXConds cn m (throws (cn,METHOD.signature m)).

(* Implementation = Local variables, initialization and body *)
Definition TrBodyWrapper (cn:ClassName) (m: Method) (body: BytecodeMethod): Boogie.Body :=
  (TrVars body, TrInit cn m::TrBody cn m body).

(* Translate method implementation *)
Definition TrImplDecl (cn:ClassName) (m: Method): list Boogie.Declaration :=
  let ms := METHOD.signature m in
  match METHOD.body m with
    | None => nil
    | Some bm =>
      Boogie.ImplementationDecl (procName (cn,METHODSIGNATURE.name ms))
        (TrParams m)
        match METHODSIGNATURE.result ms with
          | None => None
          | Some t => Some (TrType t)
        end
        (TrBodyWrapper cn m bm)::nil
  end.

(* Top level translation function *)
Definition Tr (cn:ClassName) (m: Method): list Boogie.Declaration :=
  match METHOD.body m with
    | None => nil
    | Some bm => TrProcDecl cn m::TrImplDecl cn m
  end.

End Translation.
```

3.3 Towards an executable specification

Coq provides facilities for the extraction of functional programs from a given specification. We have decided against using this feature, since some parts of the translation, especially the data-flow analysis (size and type of the operand stack and registers) and the translation of method specifications to BoogiePL expressions (*getSpec*) have been assumed to exist. To actually perform the translation of a bytecode method, these parameters would have to be implemented. We think that the primary usefulness of the specification stems from a precise description of the translation, and extracting a functional program that does not work out of the box but needs to be modified to work is too much of a hassle and not really useful in the end. All the examples shown in this report have been translated by hand, faithful to the translation function.

Chapter 4

Exception Handling in Spec#

4.1 Differences in C# and Java exception handling

Contrary to Java, the C# language does not offer a distinction between *checked* and *unchecked* exceptions[1]. In Java, checked exceptions are exceptions that the caller must handle either by catching the exception or declaring that the exception should be handled by its caller via the **throws** clause. This requirement is enforced by Java compilers. Unchecked exceptions (also called runtime exceptions) do not have to be caught or declared in a **throws** clause since they might occur in many places, so checking and declaring them would lead to code bloat. Some unchecked exceptions are also in a way not meant to be caught since the error they signal is fatal, for instance when the JVM runs out of memory (java.lang.OutOfMemoryException). In C#, all exceptions are unchecked and no **throws** clause exists, thus the programmer has to rely on documentation to find out which exceptions a method might throw. Spec# retrofits C# with checked and unchecked exceptions, as described in [8]. In short, checked exceptions implement a marker interface (ICheckedException) and a `throws` keyword is introduced for methods.

4.2 Current state of Boogie

The Boogie program verifier is written in Spec# and grouped into multiple projects. The part that is concerned with the translation from the .NET intermediate language (CIL) to BoogiePL is located in the Microsoft.Boogie.Translator class in the file `Translate.ssc`. The translator is a visitor for control flow graph nodes, which have *normal* and *exceptional successors*. The information about the exceptional control flow is present in the control flow graph, but is not currently translated to BoogiePL, the translator only visits the normal successors of the nodes. This means that CFG nodes that represent code in catch blocks are never reached.

4.2.1 Method invocation

Calling a method that might throw an exception does not lead to the creation of additional BoogiePL blocks which represent an exceptional execution of the callee. Instead, a **call** command is generated which is desugared[2] in a later stage into a series of assertions (callee preconditions), havoc statements (frame conditions) and assumptions (callee postconditions). The **call** command assumes a method to always terminate normally.

[1] Checked or unchecked exceptions are a topic of debate, see [6]

[2] Details of the desugaring can be found in the method ComputeDesugaring of class Microsoft.Boogie.CallCmd

4.2.2 Throwing exceptions

Throwing an exception currently leads to the generation of the following BoogiePL commands, and the termination of the current block.

```
block42:
    ...
    assert stackho != null;
    assume false;
    return;
```

The reason for the **assume** false; statement is for the method to pass verification, without having to satisfy its postcondition[3].

4.3 Implemented changes

We have added experimental support for our model of dealing with exceptions to Boogie, which can be activated by supplying the command-line switch `/experimentalExceptions`. The next sections explain what exactly has been modified.

All relevant modifications are confined to the class Microsoft.Boogie.Translator, which is a visitor on a pre-processed IL representation of the method to be translated. The translation method Translate iterates over the method's control flow graph blocks and creates BoogiePL blocks as output.

4.3.1 Method invocation

Method invocation is translated in the VisitCall method. When our modification is active, the current BoogiePL block is ended and successor blocks are generated that represent the callee's normal and exceptional executions. We can not use the **call** command, since it is desugared into a *state command*, which is essentially a sequence of commands with its own set of local variables (used for argument copying and reasoning about old expressions, much like in our translation), and can not be split up over multiple blocks. Therefore, we manually desugar the **call** command and split its contents over multiple blocks. The local variables introduced by the state command are declared as local variables of the procedure instead.

At the time of writing, there was no release available that serialized the information held in the `throws` clause. We use a conservative assumption, namely that the method can throw an object of type System.Exception or a subtype, which leads us to consider every present handler, since all exceptions derive from System.Exception. The developers have been notified of the situation, and will incorporate serialization of the `throws` clause in the next release of Spec#.

4.3.2 Code in catch blocks

To take code in catch blocks into consideration, we modify the translation method to also visit the exceptional successor of a block, i.e. the enclosing handler. Exceptional successor blocks are translated unless they are the method's exceptional exit point.

4.3.3 Throwing exceptions

Throwing an exception is translated in the VisitThrow method. When our modifications are active, the type of the thrown exception is determined and an appropriate handler is searched. If no handler is present, we simply insert an **assume** false; and return. As soon as Spec# has a notion of exceptional postconditions, we could branch to a block that asserts the exceptional postcondition, or use some special keyword (e.g. returnx) to denote an exceptional exit.

[3] Boogie will "verify" anything at this point

4.4 Example

To illustrate the effect of our modifications to the translation method, consider the following example. The Spec# code below implements the Account class we have used earlier in this report. The following sections focus on the transfer method.

```
1  using System;
2  using Microsoft.Contracts;
3
4  namespace ahs.examples {
5      // exception hierarchy
6      public class AccountException: Exception, ICheckedException {}
7      public class InsufficientFundsException: AccountException {}
8      public class TransferFailedException: AccountException {}
9
10     // simple account class
11     public class Account {
12
13         public int balance;
14
15         public Account(int initial)
16             requires initial >= 0;
17             ensures balance == initial;
18         {
19             balance = initial ;
20         }
21
22         public void deposit(int amount)
23             requires amount > 0;
24             modifies this.balance;
25             ensures old(balance) + amount == balance;
26         {
27             balance = balance + amount;
28         }
29
30         public /*virtual*/ void withdraw(int amount)
31             requires amount > 0;
32             modifies this.balance;
33             ensures balance + amount == old(balance);
34             throws InsufficientFundsException;
35         {
36             if (balance < amount)
37                 throw new InsufficientFundsException();
38
39             balance = balance   amount;
40         }
41
42         public static void transfer(Account! src, Account! dest, int amount)
43             requires amount > 0;
44             throws TransferFailedException;
45         {
46             try {
47                 src.withdraw(amount);
48             } catch (InsufficientFundsException) {
49                 throw new TransferFailedException();
50             }
51             dest.deposit(amount);
52         }
53     }
```

```
54 }
```

To understand the BoogiePL translations, it is useful to first review the generated CIL code. This can be done easily with the `ildasm` tool, which is part of the .NET Framework SDK. The disassembly of the transfer method is shown below. Note that the RequiresAttribute which contains the serialized precondition has been omitted.

```
.method public hidebysig static void transfer(
    class ahs.examples.Account modopt([System.Compiler.Runtime]Microsoft.Contracts.NonNullType) src,
    class ahs.examples.Account modopt([System.Compiler.Runtime]Microsoft.Contracts.NonNullType) dest,
    int32 amount) cil managed
{
  // Code size     110 (0x6e)
  .maxstack 4
  .locals init ([0] class [System.Compiler.Runtime]Microsoft.Contracts.ContractMarkerException V_0,
          [1] class ahs.examples.InsufficientFundsException V_1)
  .try
  {
    IL_0000: ldarg.0
    IL_0001: ldnull
    IL_0002: beq       IL_000c
    IL_0007: br        IL_0017
    IL_000c: ldstr     "src"
    IL_0011: newobj    instance void [mscorlib]System.ArgumentNullException::.ctor(string)
    IL_0016: throw
    IL_0017: ldarg.1
    IL_0018: ldnull
    IL_0019: beq       IL_0023
    IL_001e: br        IL_002e
    IL_0023: ldstr     "dest"
    IL_0028: newobj    instance void [mscorlib]System.ArgumentNullException::.ctor(string)
    IL_002d: throw
    IL_002e: ldarg.2
    IL_002f: ldc.i4.0
    IL_0030: ble       IL_003a
    IL_0035: br        IL_0045
    IL_003a: ldstr     "Precondition violated from method 'ahs.examples.Ac"
    + "count.transfer(optional(Microsoft.Contracts.NonNullType) ahs.examples.A"
    + "ccount,optional(Microsoft.Contracts.NonNullType) ahs.examples.Account,S"
    + "ystem.Int32)'"
    IL_003f: newobj    instance void [System.Compiler.Runtime]
                          Microsoft.Contracts.RequiresException::.ctor(string)
    IL_0044: throw
    IL_0045: leave     IL_004d
  }  // end .try
  catch [System.Compiler.Runtime]Microsoft.Contracts.ContractMarkerException
  {
    IL_004a: stloc.0
    IL_004b: rethrow
  }  // end handler
  IL_004d: nop
  .try
  {
    IL_004e: ldarg.0
    IL_004f: ldarg.2
    IL_0050: call      instance void ahs.examples.Account::withdraw(int32)
    IL_0055: leave     IL_0066
  }  // end .try
  catch ahs.examples.InsufficientFundsException
```

```
  {
    IL_005a: stloc.1
    IL_005b: newobj     instance void ahs.examples.TransferFailedException::.ctor()
    IL_0060: throw
    IL_0061: leave      IL_0066
  }  // end handler
  IL_0066: ldarg.1
  IL_0067: ldarg.2
  IL_0068: call       instance void ahs.examples.Account::deposit(int32)
  IL_006d: ret
} // end of method Account::transfer
```

We can observe that a large part of the generated IL code actually consists of runtime checks inserted by the compiler. In the example, the two Account arguments are compared to `null`, and the precondition (amount > 0) is checked. If any of these checks fail, an appropriate exception is thrown.

The actual method body starts at label `IL_004e`, which marks the beginning of a protected region of code. The arguments are loaded on to the stack and the withdraw method is called. If withdraw terminates exceptionally, control is transferred to the handler at `IL_005a`, which in turn throws an exception to indicate that the transfer has failed. If withdraw terminates normally, execution resumes at `IL_0066` where the deposit method is called, and the method returns.

4.4.1 Translation prior to modifications

Translating the transfer method by invoking Boogie with
`boogie /translate:transfer /print:Account.orig.bpl Account.dll`
yields the following BoogiePL code.

```
1  implementation ahs.examples.Account.transfer$ahs.examples.Account$notnull$ahs.examples.Account$notnull$
2      System.Int32(src$in: ref, dest$in: ref, amount$in: int)
3  {
4    var src: ref where $IsNotNull(src, ahs.examples.Account),
5        dest: ref where $IsNotNull(dest, ahs.examples.Account),
6        amount: int where InRange(amount, System.Int32),
7        stack0i: int;
8
9    entry:
10     src := src$in;
11     dest := dest$in;
12     amount := amount$in;
13     goto block1751;
14
15   block1751:
16     goto block1768;
17
18   block1768:
19     goto block1785, block1802;
20
21   block1785:
22     goto block1819;
23
24   block1802:
25     assume false;
26     return;
27
28   block1819:
29     goto block1836, block1853;
30
```

```
31    block1836:
32      goto block1870;
33
34    block1853:
35      assume false;
36      return;
37
38    block1870:
39      goto block1887, block1904;
40
41    block1887:
42      goto block1921;
43
44    block1904:
45      assume false;
46      return;
47
48    block1921:
49      goto block1955;
50
51    block1955:
52      //        nop
53      goto block1972;
54
55    block1972:
56      //        copy          Account.ssc(51,17)
57      stack0i := amount;
58      //        call          Account.ssc(51,17)
59      assert src != null;
60      call ahs.examples.Account.withdraw$System.Int32$.Virtual.$(src, stack0i);
61      //        branch
62      goto block2023;
63
64    block2023:
65      //        copy          Account.ssc(55,13)
66      stack0i := amount;
67      //        call          Account.ssc(55,13)
68      assert dest != null;
69      call ahs.examples.Account.deposit$System.Int32$.Virtual.$(dest, stack0i);
70      //        return
71      return;
72  }
```

The first series of blocks (block1751 to block1955) are generated from the instrumentation code. The actual instrumentation code is not translated, only the CFG nodes and edges. Boogie identifies instrumentation code by checking if the block is protected by a catch handler for the type Microsoft.Contracts.ContractMarkerException. The **assume** false; statements are generated for blocks whose continuation is a throw statement.

The blocks `block1972` and `block2023` contain the calls to withdraw and deposit. The CFG block containing the `catch` handler is ignored by the translation method and thus not present in the output. The two **call** statements are later desugared into a series of assumptions, havoc statements and assertions, as described in the previous section. The BoogiePL code after the desugaring of the two **call** statements, along with other useful information, can be obtained by invoking Boogie with the following command:

```
boogie /translate:transfer /traceverify /print:Account.trace.bpl Account.dll
```

The output is shown below. For sake of brevity, we only show the translation of the block containing the first **call** statement, whose desugaring is highlighted in the listing.

```
1  {
2      // ...
3
4    block1700:
5      assume true;
6      //        copy          Account.ssc(47,17)
7      stack0i := amount;
8      //        call          Account.ssc(47,17)
9      assert src != null;
10     call ahs.examples.Account.withdraw$System.Int32(src, stack0i);
11     {
12       var call1183formal@this: ref;
13       var call1183formal@amount$in: int;
14       var call1183old@$Heap: [ref,name]any;
15       call1183formal@this := src;
16       call1183formal@amount$in := stack0i;
17       assert call1183formal@amount$in > 0;
18       assert (cast($Heap[call1183formal@this, $ownerFrame],name) == $PeerGroupPlaceholder
19                 ||
20               !(cast($Heap[cast($Heap[call1183formal@this, $ownerRef],ref), $inv],name)
21                 <: cast($Heap[call1183formal@this, $ownerFrame],name))
22                 ||
23               cast($Heap[cast($Heap[call1183formal@this, $ownerRef],ref), $localinv],name)
24                 == $BaseClass(cast($Heap[call1183formal@this, $ownerFrame],name))
25             ) &&
26             (forall $pc: ref :: $pc != null && $Heap[$pc, $allocated] == true
27                &&
28                cast($Heap[$pc, $ownerRef],ref) == cast($Heap[call1183formal@this, $ownerRef],ref)
29                &&
30                cast($Heap[$pc, $ownerFrame],name)
31                    == cast($Heap[call1183formal@this, $ownerFrame],name)
32              ==>
33                cast($Heap[$pc, $inv],name) == $typeof($pc)
34                &&
35                cast($Heap[$pc, $localinv],name) == $typeof($pc)
36             );
37       call1183old@$Heap := $Heap;
38       havoc $Heap;
39       assume cast($Heap[call1183formal@this, ahs.examples.Account.balance],int)
40                  + call1183formal@amount$in
41           == cast(call1183old@$Heap[call1183formal@this, ahs.examples.Account.balance],int);
42       assume (forall $o: ref :: $o != null && call1183old@$Heap[$o, $allocated] != true &&
43                  $Heap[$o, $allocated] == true
44           ==> cast($Heap[$o, $inv],name) == $typeof($o)
45                  && cast($Heap[$o, $localinv],name) == $typeof($o));
46       assume (forall $o: ref :: $o != null && call1183old@$Heap[$o, $allocated] == true
47           ==> cast(call1183old@$Heap[$o, $ownerRef],ref) == cast($Heap[$o, $ownerRef],ref)
48                &&
49                cast(call1183old@$Heap[$o, $ownerFrame],name)
50                  == cast($Heap[$o, $ownerFrame],name));
51       assume (forall $o: ref, $f: name :: { $Heap[$o, $f] } $f != $inv && $f != $localinv &&
52                  (! IsStaticField($f) || !IsDirectlyModifiableField($f)) && $o != null &&
53                  call1183old@$Heap[$o, $allocated] == true &&
54                  (cast(call1183old@$Heap[$o, $ownerFrame],name) == $PeerGroupPlaceholder
55                    ||
56                   !(cast(call1183old@$Heap[cast(call1183old@$Heap[$o, $ownerRef],ref), $inv],name)
57                       <: cast(call1183old@$Heap[$o, $ownerFrame],name))
58                    ||
```

```
59                  cast(call1183old@$Heap[cast(call1183old@$Heap[$o, $ownerRef],ref), $localinv],name)
60                          == $BaseClass(cast(call1183old@$Heap[$o, $ownerFrame],name))
61                      )
62                      &&
63                      ($o != call1183formal@this || $f != ahs.examples.Account.balance)
64                          ==> call1183old@$Heap[$o, $f] == $Heap[$o, $f]);
65      assume (forall $o: ref :: (call1183old@$Heap[$o, $inv] == $Heap[$o, $inv] &&
66                  call1183old@$Heap[$o, $localinv] == $Heap[$o, $localinv])
67                  || call1183old@$Heap[$o, $allocated] != true);
68      assume (forall $o: ref :: call1183old@$Heap[$o, $allocated] == true ==>
69                  $Heap[$o, $allocated] == true)
70              &&
71          (forall $ot: ref :: call1183old@$Heap[$ot, $allocated] == true
72                  && cast(call1183old@$Heap[$ot, $ownerFrame],name)
73                      != $PeerGroupPlaceholder
74                  ==> cast($Heap[$ot, $ownerRef],ref)
75                          == cast(call1183old@$Heap[$ot, $ownerRef],ref)
76                      && cast($Heap[$ot, $ownerFrame],name)
77                          == cast(call1183old@$Heap[$ot, $ownerFrame],name)
78          );
79      assume (forall $o: ref :: call1183old@$Heap[$o, $sharingMode] == $Heap[$o, $sharingMode]);
80  }
80  // branch
81  assume true;
82  goto block1751;
83
84  // ...
85 }
```

As can be seen the **call** statement desugars into a state command containing its own set of local variables (call1183formal@this, call1183formal@amount$in and call1183old@$Heap) for storing the actual arguments and holding on to the frame. The block consists of transferring the actual arguments to the variables, asserting the precondition of the method, then **havoc**ing the frame and assuming the postcondition.

4.4.2 Modified translation

By invoking Boogie with the command
`boogie /experimentalExceptions /translate:transfer /print:Account.bpl Account.dll`
we get the output of our modified translation. The differences between the original output are highlighted in the listing.

```
1  implementation ahs.examples.Account.transfer$ahs.examples.Account$notnull$ahs.examples.Account$notnull$
2      System.Int32(src$in: ref, dest$in: ref, amount$in: int)
3  {
4    var src: ref where $IsNotNull(src, ahs.examples.Account),
5        dest: ref where $IsNotNull(dest, ahs.examples.Account),
6        amount: int where InRange(amount, System.Int32),
7        stack0i: int,
8        call4760formal$this: ref,
9        call4760formal$amount$in: int,
10       call4760old$$Heap: [ref,name]any,
11       call4896formal$this: ref,
12       call4896formal$amount$in: int,
13       call4896old$$Heap: [ref,name]any,
14       stack0o: ref,
15       local1: ref where $Is(local1, ahs.examples.InsufficientFundsException),
16       stack50000o: ref,
```

```
17          call4828formal$this: ref,
18          call4828old$$Heap: [ref,name]any;
19      entry:
20        src := src$in;
21        dest := dest$in;
22        amount := amount$in;
23        goto block4301;
24
25      block4301:
26        goto block4318;
27
28      block4318:
29        goto block4335, block4352;
30
31      block4335:
32        goto block4369;
33
34      block4352:
35        assume false;
36        return;
37
38      block4369:
39        goto block4386, block4403;
40
41      block4386:
42        goto block4420;
43
44      block4403:
45        assume false;
46        return;
47
48      block4420:
49        goto block4437, block4454;
50
51      block4437:
52        goto block4471;
53
54      block4454:
55        assume false;
56        return;
57
58      block4471:
59        goto block4505;
60
61      block4505:
62        //         nop
63        goto block4522;
64
65      block4522:
66        //         copy          Account.ssc(47,17)
67        stack0i := amount;
68        //         call          Account.ssc(47,17)
69        assert src != null;
70        // call  withdraw
71        call4760formal$this := src;
72        call4760formal$amount$in := stack0i;
73        assert call4760formal$amount$in > 0;
74        assert (cast($Heap[call4760formal$this, $ownerFrame],name) == $PeerGroupPlaceholder
```

```
75              ||
76              !(cast($Heap[cast($Heap[call4760formal$this, $ownerRef],ref), $inv],name)
77                  <: cast($Heap[call4760formal$this, $ownerFrame],name))
78              ||
79              cast($Heap[cast($Heap[call4760formal$this, $ownerRef],ref), $localinv],name)
80                  == $BaseClass(cast($Heap[call4760formal$this, $ownerFrame],name))
81            )
82            &&
83            (forall $pc: ref :: $pc != null && $Heap[$pc, $allocated] == true &&
84                cast($Heap[$pc, $ownerRef],ref)
85                    == cast($Heap[call4760formal$this, $ownerRef],ref) &&
86                cast($Heap[$pc, $ownerFrame],name)
87                    == cast($Heap[call4760formal$this, $ownerFrame],name)
88                ==>
89                cast($Heap[$pc, $inv],name) == $typeof($pc) &&
90                cast($Heap[$pc, $localinv],name) == $typeof($pc));
91    call4760old$$Heap := $Heap;
92    havoc $Heap;
93    goto block4522_N, block4522_X;
94
95  block4522_N:
96    // normal termination of withdraw
97    assume cast($Heap[call4760formal$this, ahs.examples.Account.balance],int)
98                + call4760formal$amount$in
99            == cast(call4760old$$Heap[call4760formal$this, ahs.examples.Account.balance],int);
100   assume (forall $o: ref :: $o != null && call4760old$$Heap[$o, $allocated] != true &&
101               $Heap[$o, $allocated] == true
102           ==>
103           cast($Heap[$o, $inv],name) == $typeof($o)
104               && cast($Heap[$o, $localinv],name) == $typeof($o));
105   assume (forall $o: ref :: $o != null && call4760old$$Heap[$o, $allocated] == true
106           ==>
107           cast(call4760old$$Heap[$o, $ownerRef],ref) == cast($Heap[$o, $ownerRef],ref)
108           &&
109           cast(call4760old$$Heap[$o, $ownerFrame],name)
110               == cast($Heap[$o, $ownerFrame],name));
111   assume (forall $o: ref, $f: name :: { $Heap[$o, $f] } $f != $inv && $f != $localinv &&
112               (! IsStaticField($f) || !IsDirectlyModifiableField($f)) &&
113               $o != null && call4760old$$Heap[$o, $allocated] == true &&
114               (cast(call4760old$$Heap[$o, $ownerFrame],name) == $PeerGroupPlaceholder
115                   ||
116                !(cast(call4760old$$Heap[cast(call4760old$$Heap[$o, $ownerRef],ref), $inv],name)
117                       <: cast(call4760old$$Heap[$o, $ownerFrame],name))
118                   ||
119                cast(call4760old$$Heap[cast(call4760old$$Heap[$o, $ownerRef],ref), $localinv],name)
120                       == $BaseClass(cast(call4760old$$Heap[$o, $ownerFrame],name)))
121               &&
122               ($o != call4760formal$this || $f != ahs.examples.Account.balance)
123           ==> call4760old$$Heap[$o, $f] == $Heap[$o, $f]);
124   assume (forall $o: ref :: (call4760old$$Heap[$o, $inv] == $Heap[$o, $inv] &&
125               call4760old$$Heap[$o, $localinv] == $Heap[$o, $localinv])
126               ||
127               call4760old$$Heap[$o, $allocated] != true);
128   assume (forall $o: ref :: call4760old$$Heap[$o, $allocated] == true
129               ==> $Heap[$o, $allocated] == true)
130         &&
131         (forall $ot: ref :: call4760old$$Heap[$ot, $allocated] == true
132          &&
```

```
133                 cast(call4760old$$Heap[$ot, $ownerFrame],name) != $PeerGroupPlaceholder
134                     ==> cast($Heap[$ot, $ownerRef],ref) == cast(call4760old$$Heap[$ot, $ownerRef],ref)
135                         && cast($Heap[$ot, $ownerFrame],name)
136                                 == cast(call4760old$$Heap[$ot, $ownerFrame],name));
137       assume (forall $o: ref :: call4760old$$Heap[$o, $sharingMode] == $Heap[$o, $sharingMode]);
137       //        branch
138       goto block4573;
139     block4522_X:
140       // exceptional termination of withdraw
141       // branch to the enclosing handler
142       goto block4539;
143     block4573:
144       //        copy          Account.ssc(51,13)
145       stack0i := amount;
146       //        call          Account.ssc(51,13)
147       assert dest != null;
148       // call deposit
149       call4896formal$this := dest;
150       call4896formal$amount$in := stack0i;
151       assert call4896formal$amount$in > 0;
152       assert (cast($Heap[call4896formal$this, $ownerFrame],name) == $PeerGroupPlaceholder
153               ||
154               !(cast($Heap[cast($Heap[call4896formal$this, $ownerRef],ref), $inv],name)
155                   <: cast($Heap[call4896formal$this, $ownerFrame],name))
156               || cast($Heap[cast($Heap[call4896formal$this, $ownerRef],ref), $localinv],name)
157                   == $BaseClass(cast($Heap[call4896formal$this, $ownerFrame],name))
158               )
159               &&
160               (forall $pc: ref :: $pc != null && $Heap[$pc, $allocated] == true &&
161                   cast($Heap[$pc, $ownerRef],ref)
162                       == cast($Heap[call4896formal$this, $ownerRef],ref) &&
163                   cast($Heap[$pc, $ownerFrame],name)
164                       == cast($Heap[call4896formal$this, $ownerFrame],name)
165                   ==>
166                   cast($Heap[$pc, $inv],name) == $typeof($pc)
167                   && cast($Heap[$pc, $localinv],name) == $typeof($pc));
168       call4896old$$Heap := $Heap;
169       havoc $Heap;
170       goto block4573_N, block4573_X;
171
172     block4573_N:
173       // normal termination of deposit
174       assume cast(call4896old$$Heap[call4896formal$this, ahs.examples.Account.balance],int)
175                   + call4896formal$amount$in
176               == cast($Heap[call4896formal$this, ahs.examples.Account.balance],int);
177       assume (forall $o: ref :: $o != null && call4896old$$Heap[$o, $allocated] != true &&
178                   $Heap[$o, $allocated] == true
179               ==> cast($Heap[$o, $inv],name) == $typeof($o)
180                   && cast($Heap[$o, $localinv],name) == $typeof($o));
181       assume (forall $o: ref :: $o != null && call4896old$$Heap[$o, $allocated] == true
182               ==> cast(call4896old$$Heap[$o, $ownerRef],ref) == cast($Heap[$o, $ownerRef],ref)
183                   && cast(call4896old$$Heap[$o, $ownerFrame],name)
184                       == cast($Heap[$o, $ownerFrame],name));
185       assume (forall $o: ref, $f: name :: { $Heap[$o, $f] } $f != $inv && $f != $localinv &&
186                   (! IsStaticField ($f) || ! IsDirectlyModifiableField ($f)) &&
187                   $o != null && call4896old$$Heap[$o, $allocated] == true &&
188                   (cast(call4896old$$Heap[$o, $ownerFrame],name) == $PeerGroupPlaceholder
189                   || !(cast(call4896old$$Heap[cast(call4896old$$Heap[$o, $ownerRef],ref), $inv],name)
```

```
190                         <: cast(call4896old$$Heap[$o, $ownerFrame],name))
191                 || cast(call4896old$$Heap[cast(call4896old$$Heap[$o,$ownerRef],ref),$localinv],name)
192                         == $BaseClass(cast(call4896old$$Heap[$o, $ownerFrame],name))
193                 ) &&
194                 ($o != call4896formal$this || $f != ahs.examples.Account.balance)
195                 ==> call4896old$$Heap[$o, $f] == $Heap[$o, $f]);
196   assume (forall $o: ref :: (call4896old$$Heap[$o, $inv] == $Heap[$o, $inv] &&
197                 call4896old$$Heap[$o, $localinv] == $Heap[$o, $localinv])
198                 ||
199                 call4896old$$Heap[$o, $allocated] != true);
200   assume (forall $o: ref :: call4896old$$Heap[$o, $allocated] == true
201                 ==> $Heap[$o, $allocated] == true) &&
202                     (forall $ot: ref :: call4896old$$Heap[$ot, $allocated] == true &&
203                         cast(call4896old$$Heap[$ot, $ownerFrame],name) != $PeerGroupPlaceholder
204                             ==> cast($Heap[$ot, $ownerRef],ref)
205                                     == cast(call4896old$$Heap[$ot, $ownerRef],ref)
206                                 && cast($Heap[$ot, $ownerFrame],name)
207                                     == cast(call4896old$$Heap[$ot, $ownerFrame],name)
208                     );
209   assume (forall $o: ref :: call4896old$$Heap[$o, $sharingMode] == $Heap[$o, $sharingMode]);
210   //        return
211   return;
212 block4573_X:
213   // exceptional termination of deposit
214   // handler is exceptional exit block
215   assume false;
216   return;
217
218 block4539:
219   //        catch ahs.examples.InsufficientFundsException
220   havoc stack0o;
221   assume cast($Heap[stack0o, $allocated],bool) == true && stack0o != null
222               && $typeof(stack0o) <: ahs.examples.InsufficientFundsException;
223   goto block4607;
224
225 block4607:
226   //        copy
227   local1 := stack0o;
228   //        new object          Account.ssc(49,17)
229   havoc stack50000o;
230   assume cast($Heap[stack50000o, $allocated],bool) == false && stack50000o != null
231               && $typeof(stack50000o) == ahs.examples.TransferFailedException;
232   assume cast($Heap[stack50000o, $ownerRef],ref) == stack50000o
233               && cast($Heap[stack50000o, $ownerFrame],name) == $PeerGroupPlaceholder;
234   $Heap[stack50000o, $allocated] := true;
235   //        call          Account.ssc(49,17)
236   assert stack50000o != null;
237   // call .ctor
238   call4828formal$this := stack50000o;
239   call4828old$$Heap := $Heap;
240   havoc $Heap;
241   goto block4607_N, block4607_X;
242
243 block4607_N:
244   // normal termination of .ctor
245   assume (cast($Heap[call4828formal$this, $ownerFrame],name) == $PeerGroupPlaceholder
246           || !(cast($Heap[cast($Heap[call4828formal$this, $ownerRef],ref), $inv],name)
247               <: cast($Heap[call4828formal$this, $ownerFrame],name))
```

```
248                 || cast($Heap[cast($Heap[call4828formal$this, $ownerRef],ref), $localinv],name)
249                     == $BaseClass(cast($Heap[call4828formal$this, $ownerFrame],name))
250                 )
251             && cast($Heap[call4828formal$this, $inv],name) == ahs.examples.TransferFailedException
252             && cast($Heap[call4828formal$this, $localinv],name) == $typeof(call4828formal$this);
253     assume $Heap[call4828formal$this, $ownerRef]
254                     == call4828old$$Heap[call4828formal$this, $ownerRef]
255                 && $Heap[call4828formal$this, $ownerFrame]
256                     == call4828old$$Heap[call4828formal$this, $ownerFrame];
257     assume $Heap[call4828formal$this, $sharingMode] == $SharingMode_Unshared;
258     assume (forall $o: ref :: $o != null && call4828old$$Heap[$o, $allocated] != true
259                     && $Heap[$o, $allocated] == true
260                 ==> cast($Heap[$o, $inv],name) == $typeof($o)
261                     && cast($Heap[$o, $localinv],name) == $typeof($o));
262     assume (forall $o: ref :: $o != null && call4828old$$Heap[$o, $allocated] == true
263                 ==> cast(call4828old$$Heap[$o, $ownerRef],ref) == cast($Heap[$o, $ownerRef],ref)
264                     && cast(call4828old$$Heap[$o, $ownerFrame],name)
265                         == cast($Heap[$o, $ownerFrame],name));
266     assume (forall $o: ref, $f: name :: { $Heap[$o, $f] } $f != $inv && $f != $localinv
267                 && (!IsStaticField($f) || !IsDirectlyModifiableField($f))
268                 && $o != null && call4828old$$Heap[$o, $allocated] == true
269                 && (cast(call4828old$$Heap[$o, $ownerFrame],name) == $PeerGroupPlaceholder
270                     || !(cast(call4828old$$Heap[cast(call4828old$$Heap[$o, $ownerRef],ref), $inv],name)
271                         <: cast(call4828old$$Heap[$o, $ownerFrame],name))
272                     || cast(call4828old$$Heap[cast(call4828old$$Heap[$o,$ownerRef],ref),$localinv],name)
273                         == $BaseClass(cast(call4828old$$Heap[$o, $ownerFrame],name))
274                     )
275                 && ($o != call4828formal$this || !(ahs.examples.TransferFailedException <: DeclType($f)))
276                 && true ==> call4828old$$Heap[$o, $f] == $Heap[$o, $f]);
277     assume (forall $o: ref :: $o == call4828formal$this
278                 || (call4828old$$Heap[$o, $inv] == $Heap[$o, $inv]
279                     && call4828old$$Heap[$o, $localinv] == $Heap[$o, $localinv])
280                 || call4828old$$Heap[$o, $allocated] != true);
281     assume (forall $o: ref :: call4828old$$Heap[$o, $allocated] == true
282                 ==> $Heap[$o, $allocated] == true)
283                     && (forall $ot: ref :: call4828old$$Heap[$ot, $allocated] == true
284                         && cast(call4828old$$Heap[$ot, $ownerFrame],name) != $PeerGroupPlaceholder
285                         ==> cast($Heap[$ot, $ownerRef],ref)
286                                 == cast(call4828old$$Heap[$ot, $ownerRef],ref)
287                             && cast($Heap[$ot, $ownerFrame],name)
288                                 == cast(call4828old$$Heap[$ot, $ownerFrame],name));
289     assume (forall $o: ref :: $o == call4828formal$this
290                 || call4828old$$Heap[$o, $sharingMode] == $Heap[$o, $sharingMode]);
291     //         copy          Account.ssc(49,17)
292     stack0o := stack50000o;
293     //         throw          Account.ssc(49,17)
294     assert stack0o != null;
295     // no handler matching handler for ahs.examples.TransferFailedException, exceptional exit
296     assume false;
297     return;
298
299   block4607_X:
300     // exceptional termination of .ctor
301     // handler is exceptional exit block
302     assume false;
303     return;
304 }
```

The translation for the instrumentation code blocks remains the same. The real changes occur in the translation of the two method calls and the constructor call. We can observe that the call to withdraw in block4522 now generates successor blocks block4522_N and block4522_X which correspond to normal and exceptional termination of the callee. In the exceptional case, we now honor the possibility of a control-flow transfer to the exception handler. We can see that the **call** statement is no longer used, instead its desugaring is inserted directly, and new local variables are introduced in the procedure scope.

The `catch` block is no longer ignored, its translation is contained in block4607, whose predecessor block is block4539 which assumes that an exception of the caught type or a subtype is now allocated on the heap and that the only stack item holds a reference to it.

In block4607_N we can observe the translation of the `throw` statement, which leads to a branch to a handler, if one is present, or to an **assume** false; and a **return**; statement, in case the exception is not handled. Since Spec# does not yet have a concept of exceptional postconditions, there is nothing which should hold in an exceptional state. The **assume** false; statement is there merely to please the verifier.

Chapter 5

Conclusion and future work

5.1 Conclusion

In this report we have presented a sequential translation of Java bytecode to BoogiePL, the input language of the Boogie program verifier. The translation supports a reasonable subset of Java bytecode, including a methodology for exceptions. We have used built-in types to model the operand stack and the registers, and an axiomatic heap that includes support for one-dimensional arrays. We have illustrated the translation in an informal and easily understandable way, and also provided a precise, type-checkable specification of the translation for the Coq theorem prover. The latter could serve as the basis of a soundness proof. We have also shown how one could extend Boogie to use our methodology for exceptions, by introducing non-deterministic branches after method invocations to model exceptional control flow.

5.2 Future work

5.2.1 Floating-point arithmetic

The current translation does not include a treatment of floating-point numbers. Support could either be built into Boogie directly (resulting in a built-in floating-point type) or as a user-defined type with proper axiomatization. The latter seems more likely, since it is an aim of the BoogiePL language to be as compact as possible, yet still remain expressive. The translation needs to be modified to also consider values that span two stack items or registers instead of just one. Some instructions (e.g. the `dup` family) become more difficult to translate since the effects differ depending on the size of the involved operands.

5.2.2 Implementation of the translation

An executable implementation that translates actual Java `class` files to BoogiePL files would be desirable. Such a translation could be implemented by extending the MultiJava compiler suite [3] with a translator that performs a data-flow analysis (to determine the stack height and types) and then translates the bytecode sequentially as presented in this project.

5.2.3 Exception methodology for Spec#

In this project we have presented a way of modeling exceptional termination of methods through the use of non-deterministic branches. The necessary alterations to the control flow in the BoogiePL output have been made, but as there is no notion of exceptional postconditions in Spec# yet, the extensions are merely a proof of concept. It would be desirable to have exceptional postconditions as part of the methodology of Spec# and Boogie.

Acknowledgment I would like to thank my supervisor Hermann Lehner and Prof. Peter Müller for a socially and professionally fruitful collaboration, and the whole lab staff for the friendly atmosphere during my project.

Bibliography

[1] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[2] F. Besson and D. Pichardie. Bicolano: Bytecode language in coq. http://www-sop.inria.fr/everest/personnel/David.Pichardie/bicolano/main.html.

[3] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: Design rationale, compiler implementation, and user experience. Technical Report 04-01, Iowa State University, Dept. of Computer Science, January 2004. Submitted for publication.

[4] R. DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report 70, Microsoft Research, 2005.

[5] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, Systems Research Center, HP Laboratories, Palo Alto, 2003.

[6] B. Eckel. Does Java need Checked Exceptions? http://www.mindview.net/Etc/Discussions/CheckedExceptions.

[7] K. Rustan M. Leino. Boogie: a modular reusable verifier for object-oriented programs. 2006.

[8] K. Rustan M. Leino and W. Schulte. Exception safety for C#. Technical report, Microsoft Research, 2004.

[9] X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3–4):235–269, 2003.

[10] T. Lindholm and F. Yellin. The Java Virtual Machine Specification. http://java.sun.com/docs/books/vmspec/.

[11] K. Rustan M. Leino M. Barnett and W. Schulte. The Spec# Programming System: An Overview. Technical report, Microsoft Research, 2004.

[12] A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, 1997.

Appendix A

Heap formalization in BoogiePL

The complete heap formalization in BoogiePL file is shown below.

```
1  //
2  // Müller/Poetzsch Heffter BoogiePL store axiomatization
3  //
4  type Store;
5
6  //
7  // Types
8  //
9  function IsClassType(name) returns (bool);
10 function IsValueType(name) returns (bool);
11 function IsArrayType(name) returns (bool);
12
13 // primitive types
14 const $int: name;
15 axiom IsValueType($int);
16
17 // array types
18 function arrayType(name) returns (name);
19 axiom (forall t:name :: IsArrayType(arrayType(t)));
20
21 function elementType(name) returns (name);
22 axiom (forall t:name :: elementType(arrayType(t)) == t);
23
24 //
25 // Values (objects, primitive values, arrays)
26 //
27 type Value;
28
29 // integer values
30 function ival(int) returns (Value);
31 axiom (forall x1: int, x2: int :: ival(x1) == ival(x2) <==> x1 == x2);
32 axiom (forall v: Value :: ival(toint(v)) == v);
33
34 function toint(Value) returns (int);
35 axiom (forall x: int :: toint(ival(x)) == x);
36
37 // reference values
38 function rval(ref) returns (Value);
39 axiom (forall o1: ref, o2: ref :: rval(o1) == rval(o2) <==> o1 == o2);
40 axiom (forall v: Value :: rval(toref(v)) == v);
41
```

```
42 function toref(Value) returns (ref);
43 axiom (forall o: ref :: toref(rval(o)) == o);
44
45 // array values
46 function aval(ref) returns (Value); // array value
47 axiom (forall o1:ref, o2:ref :: aval(o1) == aval(o2) <==> o1 == o2);
48 axiom (forall o:ref, t:name :: toref(aval(o)) == o);
49
50 // type of a value
51 function typ(Value) returns (name);
52 axiom (forall x: int :: typ(ival(x)) == $int);
53
54 // uninitialized (default) value
55 function init(name) returns (Value);
56
57 axiom init($int) == ival(0);
58 axiom (forall ct: name :: IsClassType(ct) ==> init(ct) == rval(null));
59 axiom (forall at: name :: IsArrayType(at) ==> init(at) == aval(null));
60
61 // static values
62 function static(Value) returns (bool);
63 axiom (forall x: Value :: static(x) <==> (IsValueType(typ(x)) || x == rval(null)));
64
65 // array length
66 function arrayLength(Value) returns (int);
67
68 //
69 // Locations (fields and array elements)
70 //
71
72 type Location;
73
74 // An instance field (use typeObject for static fields)
75 function fieldLoc(ref, name) returns (Location);
76 axiom (forall o1: ref, o2: ref, f1: name, f2: name ::
77     (fieldLoc(o1, f1) == fieldLoc(o2, f2)) <==> ((f1 == f2) && (o1 == o2)));
78
79 // An array element
80 function arrayLoc(ref, int) returns (Location);
81
82 // The object reference referring to an array element or instance variable
83 function obj(Location) returns (ref);
84 axiom (forall o: ref, f: name :: obj(fieldLoc(o, f)) == o);
85 axiom (forall o: ref, n: int :: obj(arrayLoc(o, n)) == o);
86
87 // Type of a location
88 function ltyp(Location) returns (name);
89 axiom (forall o: ref, f: name :: ltyp(fieldLoc(o, f)) == fieldType(f));
90 axiom (forall o: ref, i: int :: ltyp(arrayLoc(o, i)) == elementType(typ(aval(o))));
91
92 // Field declaration
93 function fieldType(fieldSig: name) returns (name);
94
95 // Static fields
96 function typeObject(className: name) returns (ref);
97 axiom (forall t: name :: typeObject(t) != null);
98 axiom (forall h: Store, t: name :: alive(rval(typeObject(t)), h));
99
```

```
100 //
101 // An allocation is either an object of a specified class type or an array
102 // of a specified element type
103 //
104 type Allocation;
105
106 function objectAlloc(name) returns (Allocation);
107 function arrayAlloc(name, int) returns (Allocation);
108
109 function allocType(Allocation) returns (name);
110 axiom (forall t: name :: allocType(objectAlloc(t)) == t);
111 axiom (forall t: name, n: int :: allocType(arrayAlloc(t,n)) == arrayType(t));
112
113
114 //
115 // Heap functions
116 //
117
118 // Return the heap after storing a value in a location.
119 function update(Store, Location, Value) returns (Store);
120
121 // Returns the heap after an object of the given type has been allocated.
122 function add(Store, Allocation) returns (Store);
123
124 // Returns the value stored in a location.
125 function get(Store, Location) returns (Value);
126
127 // Returns true if a value is alive in a given heap.
128 function alive(Value, Store) returns (bool);
129
130 // Returns a newly allocated object of the given type.
131 function new(Store, Allocation) returns (Value);
132
133 //
134 // Heap axioms
135 //
136
137 // Field stores do not affect the values stored in other fields.
138 axiom (forall l1: Location, l2: Location, h: Store, x: Value ::
139       (l1 != l2) ==> get(update(h, l1, x), l2) == get(h, l2));
140
141 // Field stores are persistent.
142 axiom (forall l: Location, h: Store, x: Value ::
143     (alive(rval(obj(l)), h) && alive(x, h)) ==> get(update(h, l, x), l) == x);
144
145 // Reading a field from a non alive object yields a type dependent default value.
146 axiom (forall l: Location, h: Store :: !alive(rval(obj(l)), h) ==> get(h, l) == init(ltyp(l)));
147
148 // Updates through non living objects do not affect the heap.
149 axiom (forall l: Location, h: Store, x: Value :: !alive(x, h) ==> (update(h, l, x) == h));
150
151 // Object allocation does not affect the existing heap.
152 axiom (forall l: Location, h: Store, a: Allocation :: get(add(h, a), l) == get(h, l));
153
154 // Field stores do not affect object liveness.
155 axiom (forall l: Location, h: Store, x: Value, y: Value ::
156     alive(x, update(h, l, y)) <==> alive(x, h));
157
```

```
158 // An object is alive if it was already alive or if it is the new object.
159 axiom (forall h: Store, x: Value, a: Allocation ::
160     alive(x, add(h, a)) <==> alive(x, h) || x == new(h, a));
161
162 // Values held stored in fields are alive.
163 axiom (forall l: Location, h: Store :: alive(get(h, l), h));
164
165 // Static values are always alive.
166 axiom (forall h: Store, x: Value :: static(x) ==> alive(x, h));
167
168 // A newly allocated object is not alive in the heap it was created in.
169 axiom (forall h: Store, a: Allocation :: !alive(new(h, a), h));
170
171 // Allocated objects retain their type.
172 axiom (forall h: Store, a: Allocation :: typ(new(h, a)) == allocType(a));
173
174 // Creating an object of a given type in two heaps yields the same result if liveness of
175 // all objects of that type is identical in both heaps.
176 axiom (forall h1: Store, h2: Store, a: Allocation ::
177     (new(h1,a) == new(h2,a)) <==>
178     (forall x: Value :: (typ(x) == allocType(a)) ==> (alive(x,h1) <==> alive(x,h2))));
179
180 // Two heaps are equal if they are indistinguishable by the alive and get functions.
181 axiom (forall h1: Store, h2: Store ::
182     (forall x: Value :: alive(x, h1) <==> alive(x, h2)) &&
183     (forall l: Location :: get(h1, l) == get(h2, l)) ==> h1 == h2);
184
185 // Get always returns a value whose type is a subtype of the (static) field type.
186 axiom (forall h: Store, o:ref, f: name :: typ(get(h, fieldLoc(o, f))) <: fieldType(f));
187
188 // Transitivity of the IsClassType predicate
189 axiom (forall t1: name, t2: name :: IsClassType(t1) && (t2 <: t1) ==> IsClassType(t2));
190
191 // New arrays have the allocated length
192 axiom (forall h: Store, t: name, n: int ::
193     arrayLength(new(h, arrayAlloc(t, n))) == n);
```