The Static Analysis Framework OPAL

The 3-Address Code Representation

Statements and Expression

Software Technology Group
Department of Computer Science
Technische Universität Darmstadt
Dr. Michael Eichberg

> If you have questions don’t hesitate to join our public chat: Gitter
> If you find any issues, please directly report them: GitHub

Statements - Assignments

The most frequent statement is the `Assignment` statement:

```
case class Assignment[+V <: Var[V]](
  pc:        PC,
  targetVar: V,
  expr:      Expr[V])
```

`pc` is the pc of the underlying original bytecode instruction!

`targetVar` identifies the var which stores the result of the evaluation of the right-hand side's expression. In case of the TACAI based representation it is always a so-called `DVar`.

After generation, OPAL's three-address code is flat. That is, all expressions referred to by expressions are either `Var`s or `Consts`, but not further nested expressions. For example, if the right hand side is a binary expression then the operands are guaranteed to be either `Const`s or `Var`s.

If the result of an expression - which potentially has a side-effect – is ignored, an `ExprStmt` is used:

```
case class ExprStmt[+V <: Var[V]](
  pc: Int,
  expr: Expr[V])
```

Pattern matching is facilitate by the common super class: `AssignmentLikeStmt`.

Statements - Unconditional jumps

Unconditional jumps:

```
case class Goto(pc: PC, target: Int)
```

The target is the absolute address of the jump target in the statements array.

If you analyze pre Java-6 code, you may encounter the following statements which are used by old compilers when compiling try-finally statements:
`case class JSR(pc: PC, target: Int)`
`case class Ret(pc: PC, returnAddresses: PCs)`

Statements - Conditional jumps

```
case class If[+V <: Var[V]](
  pc:             PC,
  left:           Expr[V],
  condition:      RelationalOperator,
  right:          Expr[V],
  target:         Int)
```

```
case class Switch[+V <: Var[V]](
  pc:             PC,
  defaultTarget: Int,
  index:          Expr[V],
  npairs:         RefArray[IntIntPair])
```

The target is always given as an absolute address.

In case of switches the `IntIntPair`'s first value is the case value; the second value is the absolute jump target.

Statements - Normal return from method

```
case class ReturnValue[+V <: Var[V]](
  pc: Int,
  expr: Expr[V])
```

```
case class Return(pc: PC)
```

Recall that a return statement may throw an exception!

Statements - Handling exceptions

If the `exception` is `null` a new instance of a `NullPointerException` is generated and thrown, in general, however, the `exception` expression is a variable.

```
case class Throw[+V <: Var[V]](
  pc: PC,
  exception: Expr[V])
```

In case of the TACAI based representation OPAL makes it explicit if an exception is caught by adding a `CaughtException` statement before the handler statement.

CaughtException is the only three-address code specific statement which is not based on a Java bytecode statement. It is basically the (always necessary) use site of a thrown exception that is otherwise swallowed.

```
case class CaughtException[+V <: Var[V]](
  pc:            PC,
  exceptionType: Option[ObjectType],
  throwingStmts: IntTrieSet)
```

Statements - Method invocations

```scala
case class (Non)VirtualMethodCall[+V <: Var[V]](
  pc:             Int,
  declaringClass: ReferenceType,
  isInterface:    Boolean,
  name:           String,
  descriptor:     MethodDescriptor,
  receiver:       Expr[V],
  params:         Seq[Expr[V]])
```

```scala
case class StaticMethodCall[+V <: Var[V]](
  pc:             Int,
  declaringClass: ObjectType,
  isInterface:    Boolean,
  name:           String,
  descriptor:     MethodDescriptor,
  params:         Seq[Expr[V]])
```

`params` are the explicitly declared parameters..

Given that it is possible to also call all methods defined by `java.lang.Object` on arrays the declaring class of virtual method calls can either be a class type or an array type.

A non-virtual instance method call is a call where the call target is statically resolved. Such a call is either the call of a private method, a super call or a constructor call.

Note that Java interfaces can now also define static and/or private methods.

Statements - Writing fields

```scala
case class PutField[+V <: Var[V]](
  pc:                Int,
  declaringClass:    ObjectType,
  name:              String,
  declaredFieldType: FieldType,
  objRef:            Expr[V],
  value:             Expr[V])
```

```scala
case class PutStatic[+V <: Var[V]](
  pc:                PC,
  declaringClass:    ObjectType,
  name:              String,
  declaredFieldType: FieldType,
  value:             Expr[V])
```

Statements - Invokedynamic

```
case class InvokedynamicMethodCall[+V <: Var[V]](
  pc:              PC,
  bootstrapMethod: BootstrapMethod,
  name:            String,
  descriptor:      MethodDescriptor,
  params:          Seq[Expr[V]]
)
```

In general, it is recommended to let OPAL resolve `invokedynamic` based calls to avoid that analyses have to handle them explicitly. However, OPAL only provides resolution of Java/Scala `invokedynamic` calls at the moment. Therefore, it is always required to also be able to handle this statement.

Statements - Checkcast

A `Checkcast` as, e.g., in `Object o = ...; ((List<?>)o).size()` typically serves two purposes: (1) checking if a specific object has the respective type and (2) performing the cast. W.r.t. (2) a `Checkcast` is basically an expression. However, the underlying data-flow analysis propagates def-use information and a check cast statement does not change the identity of an object and therefore the information about the original def-site is propagated.

A check cast merely refines the available type information. Hence, if we would assign the result of the check cast to a new `DVar`, we would have not uses of it. But it is important to realize that the type information is of course appropriately refined after a check cast statement.

```
case class Checkcast[+V <: Var[V]](
  pc: PC,
  value: Expr[V],
  cmpTpe: ReferenceType
)
```

Statements - Array writes

```
case class ArrayStore[+V <: Var[V]](
  pc:       PC,
  arrayRef: Expr[V],
  index:    Expr[V],
  value:    Expr[V]
)
```

Writes the value in the referenced array at the given index.

Statements - Synchronization

```
case class MonitorEnter[+V <: Var[V]](pc: PC, objRef: Expr[V])
case class MonitorExit[+V <: Var[V]](pc: PC, objRef: Expr[V])
```

`MonitorEnter` and `MonitorExit` statements always need to occur inside one method and need to be balanced w.r.t. a specific object.

Statements - Nops

To facilitate an efficient conversion, OPAL sometimes inserts `NOP`s in the generated code.

```
case class Nop(pc: PC)
```

Expressions - Arrays

```
case class NewArray[+V <: Var[V]](
  pc: PC,
  counts: Seq[Expr[V]],
  tpe: ArrayType)
```

```
case class ArrayLength[+V <: Var[V]](
  pc: PC,
  arrayRef: Expr[V])
```

```
case class ArrayLoad[+V <: Var[V]](
  pc: PC,
  index: Expr[V],
  arrayRef: Expr[V])
```

Expressions - Arithmetic Binary Expressions

```
case class BinaryExpr[+V <: Var[V]](
        pc:   PC,
        cTpe: ComputationalType,
        op:   BinaryArithmeticOperator,
        left: Expr[V], right: Expr[V])
```

Expressions - Constants

The following types of constants have their own representations:

- Null
- *Class*
- *String*
- *Double*
- *Float*
- *Long*
- *Integer*
- *MethodHandle*
- *MethodType*

In all cases the class follows the following pattern `case class <TypeOfConstant>Const(pc,value)`

Expressions - Comparisons

```
case class Compare[+V <: Var[V]](
  pc:        PC,
  left:      Expr[V],
  condition: RelationalOperator,
  right:     Expr[V])
```

Please note, that the potential relational operators are the typical ones (`<`, `>`, `<=`, `>=`, `==`, `!=`) and `cmp(g|l)` to compare long, float and double values.

Expressions - Accessing Fields

```
case class GetField[+V <: Var[V]](
  pc:                 PC,
  declaringClass:     ObjectType,
  name:               String,
  declaredFieldType: FieldType,
  objRef:             Expr[V])
```

```
case class GetStatic(
  pc:                 PC,
  declaringClass:     ObjectType,
  name:               String,
  declaredFieldType: FieldType)
```

Expressions - Invokedynamic

```
case class InvokedynamicFunctionCall[+V <: Var[V]](
  pc:              PC,
  bootstrapMethod: BootstrapMethod,
  name:            String,
  descriptor:      MethodDescriptor,
  params:          Seq[Expr[V]])
```

Expressions - Method Calls

```
case class (Non)VirtualFunctionCall[+V <: Var[V]](
  pc:             PC,
  declaringClass: ObjectType,
  isInterface:    Boolean,
  name:           String,
  descriptor:     MethodDescriptor,
  receiver:       Expr[V],
  params:         Seq[Expr[V]])
```

```
case class StaticFunctionCall[+V <: Var[V]](
  pc:             PC,
  declaringClass: ObjectType,
  isInterface:    Boolean,
  name:           String,
  descriptor:     MethodDescriptor,
  params:         Seq[Expr[V]])
```

The semantics of the method call expressions reflect the semantics of the method call statements w.r.t. the type of called methods.

Expressions - Creating Objects

```
case class New(pc: PC, tpe: ObjectType)
```

Expressions - Primitive Type Casts

```
case class PrimitiveTypecastExpr[+V <: Var[V]](
  pc:         PC,
  targetTpe: BaseType,
  operand:    Expr[V])
```

Casts between the primitive values: `int`, `long`, `float`, and `double`.

Expressions - Prefix Expressions

```
case class PrefixExpr[+V <: Var[V]](
  pc:      PC,
  cTpe:    ComputationalType,
  op:      UnaryArithmeticOperator,
  operand: Expr[V])
```

At the three-address code level – due to optimizations – it may be possible to identify effective boolean negations.

Expressions - Type checks

```scala
case class InstanceOf[+V <: Var[V]](
  pc: PC,
  value: Expr[V],
  cmpTpe: ReferenceType)
```

Expressions - Accessing a Parameter

The `Param` expression is exclusive to the `TACNaive` representation and represents the access of a parameter.

```
case class Param(
  cTpe: ComputationalType,
  name: String)
```