The Static Analysis Framework OPAL

Fixed Point Computations

The FPCF Framework
(*OPAL - Static Analysis Infrastructure*)

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

Overview

OPAL implements a general-purpose static analysis framework that facilitates a strictly modularized implementation of a wide-range of (potentially mutually dependent) static analyses.

The framework inherently supports fixed point computations and transparently handles cyclic dependencies.

Entities and Properties

- *Entities* represent (virtual) elements of the source code that are of particular interest
- *Properties* store the results of static analyses in relation to the entities. Every property belongs to exactly one property kind

Entities and Properties - example

Entity	Property Kind	(Final) Property
`java.lang.String`	Immutability	Immutable
`java.util.ArrayList`	Immutability	Mutable
`scala.collection.immutable.HashSet`	Immutability	Container Immutable
`Math.abs(...)`	Thrown Exceptions	None
`Math.abs(...)`	Purity	Compile-time pure

Property Kinds

The property kind encodes (ex- or implicitly):

- the *lattice* regarding a property's potential extensions and
- explicitly encodes the fallback behavior if
  - no analysis is scheduled or
  - an analysis is scheduled but no value is computed for a specific entity (e.g., if a method is deemed not reachable.)

Analyses

A static analysis is a function that - given an entity $\boldsymbol{e}$ - derives a property $\boldsymbol{p}$ of property kind $\boldsymbol{pk}$ along with the set of non-final dependees $\boldsymbol{\mathcal{D}}$ which may still influence the property $\boldsymbol{p}$.

The programming model is to always complete the analysis of the current entity and to *just* record the relevant dependencies to other entities.

Analyses - example

```
class Math {
   public static double floor(double a) {
     return StrictMath.floor(a);
}  }

class StrictMath {
       public static double floor(double a) {
           return floorOrCeil(a, -1.0, 0.0, -1.0);
       }
       private static double floorOrCeil(double a,
                                         double negativeBoundary,
                                         double positiveBoundary,
                                         double sign) { ... }
}
```

Basic Definitions

```
final type Entity = AnyRef

object PropertyKey {
    def create[E <: Entity, P <: Property](
        name:                           String,
        fallbackPropertyComputation:    FallbackPropertyComputation[E, P],
        fastTrackPropertyComputation: (PropertyStore, E) ⇒ Option[P]
    ): PropertyKey[P] }

trait Property  {
    def key: PropertyKey[Self]
    override def equals(other: Any): Boolean }

trait EOptionP[+E <: Entity, +P <: Property] {
    val e: E
    def pk: PropertyKey[P] }
```

```scala
trait EPS[+E <: Entity, +P <: Property] extends EOptionP[E, P]

final class FinalEP[+E <: Entity, +P <: Property](val e: E, val p: P) extends EPS[E, P]

sealed trait InterimEP[+E <: Entity, +P <: Property] extends EPS[E, P] {
final class InterimELUBP[+E <: Entity, +P <: Property](
         val e:  E,
         val lb: P,
         val ub: P ) extends InterimEP[E, P]
final class InterimELBP[+E <: Entity, +P <: Property](
        val e:  E,
        val lb: P ) extends InterimEP[E, P]
final class InterimEUBP[+E <: Entity, +P <: Property](
        val e:  E,
        val ub: P ) extends InterimEP[E, P]

final class EPK[+E <: Entity, +P <: Property](
        val e:  E,
        val pk: PropertyKey[P] ) extends EOptionP[E, P]
```

An analysis consists of two functions. An initial function which analyzes an entity and a second function (the continuation function) which is called whenever a dependee is updated.

```
final type PropertyComputation[E <: Entity] = E ⇒ PropertyComputationResult
final type OnUpdateContinuation = SomeEPS ⇒ PropertyComputationResult
```

A `PropertyComputationResult` encapsulates the (intermediate) information about an entity's property.

```
sealed abstract class PropertyComputationResult

case class Result(
        finalEP: FinalEP[Entity, Property]
) extends PropertyComputationResult

final class InterimResult[P >: Null <: Property] private (
        val eps:       InterimEP[Entity, P],
        val dependees: Traversable[SomeEOptionP],
        val c:         ProperOnUpdateContinuation,
        val hint:      PropertyComputationHint
) extends ProperPropertyComputationResult
```

Getting the PropertyStore

The simplest way to create the property store is to use the `PropertyStoreKey`.

```
val project : SomeProject = ...
val propertyStore = project.get(PropertyStoreKey)
```

Querying the PropertyStore

Querying the property store can be done using the property store's **`apply`** method:

```
def apply[E <: Entity, P <: Property](
    e: E,
    pk: PropertyKey[P]
): EOptionP[E, P]
```

Registering Static Analyses

Scheduling an eager analysis:

```
def scheduleEagerComputationsForEntities[E <: Entity](
      es: TraversableOnce[E] )(
      c: PropertyComputation[E] ): Unit
```

Registration of lazy analyses:

```
def registerLazyPropertyComputation[E <: Entity, P <: Property](
     pk: PropertyKey[P],
     pc: ProperPropertyComputation[E] ): Unit
```

Executing static analyses

To execute a set of scheduled eager analysis:

```
def waitOnPhaseCompletion(): Unit
```

Example - a very simple purity analysis

```
class PurityAnalysis ( final val project: SomeProject) extends FPCFAnalysis {

    import project.nonVirtualCall
    import project.resolveFieldReference

    private[this] val declaredMethods: DeclaredMethods = project.get(DeclaredMethodsKey)

    def determinePurity(definedMethod: DefinedMethod): ProperPropertyComputationResult = {
        val method = definedMethod.definedMethod
        if (method.body.isEmpty || method.isSynchronized)
            return Result(definedMethod, ImpureByAnalysis);
        determinePurityStep1(definedMethod.asDefinedMethod)
    }
    ...
```

All parameters either have to be base types or have to be immutable:

```
def determinePurityStep1(definedMethod: DefinedMethod): ProperPropertyComputationResult = {
    val method = definedMethod.definedMethod

   var referenceTypedParameters = method.parameterTypes.iterator.collect[ObjectType] {
        case t: ObjectType ⇒ t
        case _: ArrayType  ⇒ return Result(definedMethod, ImpureByAnalysis);
    }
    var dependees: Set[EOptionP[Entity, Property]] = Set.empty
    referenceTypedParameters foreach { e ⇒
        propertyStore(e, TypeImmutability.key) match {
            case FinalP(ImmutableType) ⇒ /*everything is Ok*/
            case _: FinalEP[_, _] ⇒
                return Result(definedMethod, ImpureByAnalysis);
            case InterimUBP(ub) if ub ne ImmutableType ⇒
                return Result(definedMethod, ImpureByAnalysis);
            case epk ⇒ dependees += epk
        }
    }

    doDeterminePurityOfBody(method, dependees)
}
```

```
def doDeterminePurityOfBody(
    method:              Method,
    initialDependees: Set[EOptionP[Entity, Property]]
): ProperPropertyComputationResult = {

    var dependees = initialDependees

    val maxPC = instructions.length
    var currentPC = 0
    while (currentPC < maxPC) {
```

$$< \textbf{analyze instructions and collect dependencies} >$$

```
        currentPC = body.pcOfNextInstruction(currentPC)
    }

    if (dependees.isEmpty) return Result(definedMethod, Pure);
```

```
    def c(eps: SomeEPS): ProperPropertyComputationResult = {
        dependees = dependees.filter(_.e ne eps.e)
        (eps: @unchecked) match {
            case _: InterimEP[_, _] ⇒
                dependees += eps
                InterimResult(definedMethod, ImpureByAnalysis, Pure, dependees, c)

            case FinalP(_: FinalField | ImmutableType) ⇒
                if (dependees.isEmpty) {
                    Result(definedMethod, Pure)
                } else {
                    InterimResult(definedMethod, ImpureByAnalysis, Pure, dependees, c)
                }
            case FinalP(_: NonFinalField)    ⇒ Result(definedMethod, ImpureByAnalysis)
            ...
            case FinalP(CompileTimePure | Pure) ⇒
                if (dependees.isEmpty)
                    Result(definedMethod, Pure)
                else {
                    InterimResult(definedMethod, ImpureByAnalysis, Pure, dependees, c)
                }

            case FinalP(_: Purity) ⇒ Result(definedMethod, ImpureByAnalysis)
    }   }

    InterimResult(definedMethod, ImpureByAnalysis, Pure, dependees, c)
}   }
```

Example - specifying meta information

```
object EagerPurityAnalysis extends BasicFPCFEagerAnalysisScheduler {

    final override def uses: Set[PropertyBounds] = {
        Set(PropertyBounds.ub(TypeImmutability), PropertyBounds.ub(FieldMutability))
    }
    final def derivedProperty: PropertyBounds = PropertyBounds.lub(Purity)
    override def derivesEagerly: Set[PropertyBounds] = Set(derivedProperty)
    override def derivesCollaboratively: Set[PropertyBounds] = Set.empty

    override def start(p: SomeProject, ps: PropertyStore, unused: Null): FPCFAnalysis = {
        val analysis = new PurityAnalysis(p)
        val methodsWithBody = p.get(DeclaredMethodsKey).declaredMethods.toIterator.collect {
            case dm if dm.hasSingleDefinedMethod && dm.definedMethod.body.isDefined ⇒
                dm.asDefinedMethod
        }
        ps.scheduleEagerComputationsForEntities(methodsWithBody)(analysis.determinePurity)
        analysis
}   }
```

Examples

To get a deeper understanding how to instantiate the framework consider studying concrete implementations. In OPAL, we have implemented multiple analyses using the framework:

- TypeImmutabilityAnalysis
- ClassImmutabilityAnalysis