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

The development of static analyses is centered around *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 that are generally of interest for static analyses are class and type declarations, methods, formal parameters, fields, call sites or allocation sites of new objects/arrays. Furthermore, artificial/virtual entities such as the project as a whole can also be the target of analyses.

Properties are, e.g., the immutability of (all) instances of a specific class or the immutability of (all) instances of a specific type. Another example is the purity of a method. Most common are, however, set based properties such as, the set of thrown exceptions, the set of callers and callees, the set of instantiated types etc.
In FPCF, it is a recurring pattern that a concrete analysis is implemented by two sub analysis: one analysis that derives a property related to a specific entity and a second analysis that basically just aggregates all results, e.g., over the type hierarchy.

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.)

For historical reasons, in OPAL the bottom value represents the sound over approximation and the top value the most precise one. Hence, the fallback value is typically the bottom value of the lattice if no analysis is scheduled and the top value if an analysis is scheduled, but no value for a specific entity was computed.

Being able to _not_schedule an analysis opens up the possibility of computing the effect of a static analysis on the overall result.

Analyses

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

The set of dependees consists of the intermediate results of querying the property store for the current property extensions of other entity/property kind pairings.

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

Technically and conceptually a static analysis could derive for a given entity e multiple properties ps with different property kinds pks where $|ps| = |pks|$, however, this can be thought of as being multiple analyses that are just executed concurrently.

While computing the property p for e the analysis generally requires knowledge of properties of other entities. For that, it queries the `PropertyStore` regarding the current extension of a property p' of a specific kind pk' for a specific entity e'.The result of that query will - w.r.t. the queried entity - either return a final result, which specifies the extension of the property of the respective kind, or an intermediate result. In the first case the result is just taken into consideration and the computation of p for e continues. In the latter case the property may change in the future and the analysis now has to decide if the current information is already sufficient or if it has to keep a dependency on the just queried entity e'. If the latter is the case, the analysis has to record the dependency and continue. As soon as the analysis has completed analyzing e's enclosing context it commits it results to the property store along with the open (refineable) dependencies and a function (called `continuation` function next) that continues the computation of p whenever a property p' of a dependee e' is updated. Note that in many cases certain intermediate results are actually sufficient and final results can be commit earlier which is a major factor w.r.t. the overall scalability and performance. For example, if an analysis just needs to know whether a certain value escapes at all or not, complete information is not required. Whenever the continuation function is called it will take the new information (the new extension of the property of kind pk' of entity e' along with the information if that information is final) into consideration and either store another intermediate property in the store or a final value for e.

Analyses - example

Let’s assume that we want to compute the purity of the methods.

```
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) { ... }
}
```

If the overall (lazy or eager) analysis would start with `Math.floor` then the result would be:

- The immutability of `Math.floor` would be immutable (the most precise result for which we didn’t found a counter example) along with the list of dependees: { `StrictMath.floor` } and a function (the so-called continuation function) that will update `Math.floor` ’s purity property depending on the property of the updated dependee. It is important that we can’t wait for `StrictMath.floor` ’s result. If we would do so, and we would have a cyclic dependency, we would end up in a dead lock!
- Additionally, (if the purity analysis is lazy) the computation of the purity property of `StrictMath.floor` would have been triggered.

As soon as the computation of `StrictMath.floor` ’s purity property has made progress, the continuation function related to the computation of `Math.floor` s purity property will be called again. Eventually, the result will be that both `StrictMath.floor` and `Math.floor` are (compile-time) pure.

Basic Definitions

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 }
```

Every object can be an `Entity`. A property key is created using the factory method provided by `PropertyKey`'s companion object. A `fastTrackPropertyComputation` is a function that handles extremely simple cases (e.g., empty methods) and by that reduces the workload on the property store.

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

`EOptionP` represents a pairing of an entity e and an optional property of a well defined kind pk.

Types of pairings:

```
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]
```

A property can be associated with an upper and/or a lower bound. The vast majority of analyses only use the upper bound which represents the most precise solution and refines that one. However, some analyses use both bounds to speed up analyses. E.g., imagine an analysis which determines a method's purity. This analysis may have determined that a specific method m is at least side-effect free but may also be pure. This knowledge may be sufficient for a client that *just* wants to know if m is at least side effect free.

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

The property store is the core component that handles the execution (and parallelization) of static analyses.

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]
```

Querying the property store will return the current extension of the property for the given entity. If the property is computed by a lazy analysis, the analysis will be started and an `EPK` object will be returned.

The property store offers various further query methods which are all *only* intended to be used after the analyses have been finished. (The property store has reached quiescence.)

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
```

A lazy analysis is an analysis that will only be executed for an entity e w.r.t. property kind pk when required. All base analysis should always be lazy analyses. However, some analyses, e.g., call graph algorithms strictly require an eager analysis.

Executing static analyses

To execute a set of scheduled eager analysis:

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

Lazy analyses will only be triggered when required. It is, however, possible to force the computation of a specific property (and thereby triggering the computation of further properties) using `force`.

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(DeclaredMethodsKe
y)

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

The `determinePurity` method performs the initial analysis of a method w.r.t. its purity.

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

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

       var referenceTypedParameters = method.parameterTypes.iterator.collect[ObjectTyp
e] {
            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) {
```

< 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, ImpureByAnaly
sis)
                ...
                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)
}   }
```

Important:

- You have to update the set of `dependees` (line 2); if you forget it the analysis will not terminate.

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.col
lect {
            case dm if dm.hasSingleDefinedMethod && dm.definedMethod.body.isDefined ⇒
                dm.asDefinedMethod
        }
        ps.scheduleEagerComputationsForEntities(methodsWithBody)(analysis.determinePur
ity)
        analysis
}   }
```

In this case, we specify that – at analysis time – both bounds: the lower and the upper bound are refined. However, the analysis only uses the upper bounds of the type immutability and field mutability analyses.

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