The Static Analysis Framework OPAL

Fixed Point Computations

The FPCF Framework (Subproject: `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 a project's call graph 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. 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.

Property Kinds

The property kind encodes (ex- or implicitly):

- the lattice regarding a property's 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.

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 dependees $\mathcal{D}$ which may still influence the property p. The set of dependees consists of the 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.

Basic Definitions

Basic type definitions:

```
final type Entity = AnyRef

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]
```

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.

```
sealed abstract class PropertyComputationResult {
case class Result(finalEP: FinalEP[Entity, Property]) extends FinalPropertyComputation
Result
final class InterimResult[P >: Null <: Property] private (
```

```
        val eps:       InterimEP[Entity, P],
        val dependees: Traversable[SomeEOptionP],
        val c:         ProperOnUpdateContinuation,
        val hint:      PropertyComputationHint
) extends ProperPropertyComputationResult
```

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]
```

Registering Static Analyses

Scheduling an eager analysis:

scala
def scheduleEagerComputationsForEntitiesE <: Entity(
c: PropertyComputation[E]): Unit

```
Registration of lazy analyses:
```

scala
def registerLazyPropertyComputationE <: Entity, P <: Property: Unit

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

---

# Executing Static Analyses

To execute a set of scheduled eager analysis:
```

scala
def waitOnPhaseCompletion(): Unit
```

Lazy analyses will be triggered when required.

---
```

Examples

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

- TypeImmutabilityAnalysis
- ClassImmutabilityAnalysis