The Static Analysis Framework OPAL

The 3-Address Code Representation

Introduction

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

Getting the naive three-address code

```
object TACNaive {
  def apply(
    method:         Method,
    classHierarchy: ClassHierarchy,
    optimizations:  List[TACOptimization[Param, IdBasedVar]] = List.empty
  ): TACode[Param, IdBasedVar] = { ...}
}
```

Getting the naive three address code is always possible and does not require any kind of specialized data-flow analysis. However, only very few optimizations/transformations are performed.

In the very vast majority of cases it highly recommended to build your analysis on top of OPAL's SSA-like three-address code presented next.

Getting the SSA-Like three-address code (TAC(AI))

This requires the execution of a data-flow analysis because the SAA-like address code is parameterized over the result of the data-flow analysis (`AIResult`). The three-address code is `SSA-like` because it makes the `def-use` and `use-def` directly available, but does not use ϕ-statements.

```
object TACAI{
  def apply(
    project:  SomeProject,
    method:   Method,
    aiResult: AIResult { val domain: Domain with RecordDefUse }
  ): TACode[TACMethodParameter, DUVar[aiResult.domain.DomainValue]] = { ... }
}
```

A convenient way to get the SSA-like representation is to use one of the respective `ProjectInformationKey` s.

```
val tacaiKey = project.get(ComputeTACAIKey) // The Key: ComputedTACAIKey
val taCode = tacaiKey(m)
```

The `ComputeTACAIKey` returns a function object that – given a method – (re)computes the three-address code for the method on demand. I.e., the result is not cached. The link to the results of the underlying data-flow analysis are kept. The latter requires significant memory but generally provides more information.

The `Lazy|EagerDetachedTACAIKey` return function objects that compute the three-address code for a method lazily/eagerly and then detaches it from the results of the underlying data-flow analysis.

Analysis specific adaptation of (TAC(AI))

The data-flow analysis that is used as a foundation for generating the SSA-like three address code can be changed by updating the information about the so-called domain that should be used. A domain basically determines how the values are taken into account.

```
project.updateProjectInformationKeyInitializationData(
    ComputeTACAIKey) { // or Lazy|EagerDetachedTACAIKey
    _ ⇒ (method: Method) ⇒ new DefaultDomainWithCFGAndDefUse(project, method)
}
```

Alternative domains:

- `l0.PrimitiveTACAIDomain` - only basic type information is propagated.
- `l1.DefaultDomainWithCFGAndDefUse` - type information is computed more precisely.
- `l2.DefaultPerformInvocationsDomainWithCFGAndDefUse` - monomorphic calls are inlined (depth:1).

The l0 domain is the fast domain that can be used. Depending on the use-case it may be sufficient and it provides SSA-like three-address code that is widely comparable to SSA code used by other frameworks.

The l1 domain additionally tracks `Class` and `String` objects (intra-procedurally) and tries to compute the ranges of integer variables.

Using the l2 domain makes it possible to, e.g., identify that the receiver objects of call chains such as `myStringBuilder.append(x).append(y)...` is actually always the same object.

TACode

The `TACode` object is the general entry point.

```
class TACode[P <: AnyRef, V <: Var[V]](
  val params:            Parameters[P],
  val stmts:             Array[Stmt[V]],
  val pcToIndex:         Array[Int],
  val cfg:               CFG[Stmt[V], TACStmts[V]],
  val exceptionHandlers: ExceptionHandlers
)
```

- `params` provides information about the parameters passed to the method; in particular about their use-sites.
- `stmts` contains the three-address code statements. (Manipulation is strictly prohibited.)
- `pcToIndex` contains the mapping between the pcs of the original bytecode instructions and the corresponding statements. It may be the case that multiple pcs are mapped to a single TAC statement. This data-structure is required if you want to use further code attributes which are not rewritten when we generate the `TACode` and, hence, need to be updated on-demand.
- `cfg` the control-flow graph.
- `exceptionHandlers` specifies the try-catch blocks and handlers in terms of TAC statements.
- to get the line number of a statement the TACode object provides respective helper methods.

OPAL's three-address code

On the origin of values

When we analyze a method it may happen that a single expression/statement gives rise to different values: the value that is computed if the expression completes successfully and the value that is computed if the evaluation throws an exception!

In general, a def site can be a value in the ranges:

Range	Semantics
[0... `stmts.length`]	The `Assignment` statement at the given index (def-site) initialized the variable.
[-256...-1]	Identifies the respective parameter.
[-165535...-100,000]	Identifies an exception that was created by the JVM because the evaluation of the instruction failed.

OPAL's three-address code

Variables

The left-hand side of an assignment is always a unique `DVar`. A `DVar` provides information about the value, the origin of the value (the statement index) as well as the useSites. The kind of information provided about the value depends on the underlying data-flow analysis:

```
class DVar[+Value <: ValueInformation] (
  origin:   ValueOrigin,
  value:    Value,
  useSites: IntTrieSet
) extends DUVar[Value]
```

`

Expressions always use a `UVar` which makes the def-sites explicit and which provides (additional) information about the value. E.g., due to a type check.

```
class UVar[+Value <: ValueInformation] (
  value:    Value,
  defSites: IntTrieSet
) extends DUVar[Value]
```