Toward Interprocedural Pointer and Effect Analysis for Scala

Etienne Kneuss

Laboratory for Automated Reasoning and Analysis
School of Computer and Communication Sciences
EPFL

Thesis Supervisor: Viktor Kuncak
Thesis Supervisor: Philippe Suter

July 11, 2011

Outline

insane!

Interprocedural Static Analysis Engine for Scala

- Precise pointer and effect analysis
  - Whole-Program but compositional
  - Based on abstract interpretation
  - Interprocedural
- Working Implementation
  - Provided as a compiler plug-in
  - Accepts any Scala code
  - Requires no annotations

Definitions

Pointer Analysis

Static analysis technique that builds information on the relations between pointers and allocated objects.

Effect Analysis

Static analysis technique that summarizes the side effects of procedures in a certain domain.

Analysis Phases

The analysis currently consists of five main phases:

1. Abstract Syntax Tree Extraction
2. Control Flow Graphs Generation
3. Class Analysis & Call Graph Generation
4. Effect Graphs Generation
5. Purity Analysis

Class Analysis

Class Analysis aims at establishing the types of runtime values. In the presence of dynamic dispatch, this information is used to compute a precise call-graph.

```
object Test {
    def run1(obj: A) {
        obj.f()
    }
    def run2() {
        val obj = new A
        obj.f()
    }
}
```

```
class A {
    def f() { ... }
}
class B extends A {
    override def f() { ... }
}
```

Class Analysis

- Analysis is flow sensitive, based on abstract interpretation
- For each local variable, we assign an abstract value of the form:

  $$\langle T_{sub}, T_{exact} \rangle$$

  where T_{sub} and T_{exact} are two sets of types

Expression *ex*	Abstract Value $\alpha(ex)$
`new A`	$\langle \emptyset, \{A\} \rangle$
`null`	$\langle \emptyset, \emptyset \rangle$
`obj.f`	$\langle \{type(\texttt{obj.f})\}, \{type(\texttt{obj.f})\} \rangle$
`rec.meth(..)`	$\langle \{type(\texttt{rec.meth})\}, \{type(\texttt{rec.meth})\} \rangle$

Class Analysis

Once we collected type information for every local variables, we generate the call graph.
In the presence of a call rec.meth(), with $\langle T_{sub}, T_{exact} \rangle$ as type information for rec, we consider calls to:

$$\{C.meth \mid (C \in T_{exact} \vee C \sqsubset T_{sub}) \wedge meth \in declMethods(C)\}$$

The call graph is then used to compute sets of mutually-dependant procedures (strongly connected components), and then to order the effect analysis (topological sort).

Effect/Alias Analysis

- Analysis based on abstract interpretation
- Graphs as representation
- Compositional: One graph per method, with "connecting" nodes.

Graphs are defined as:

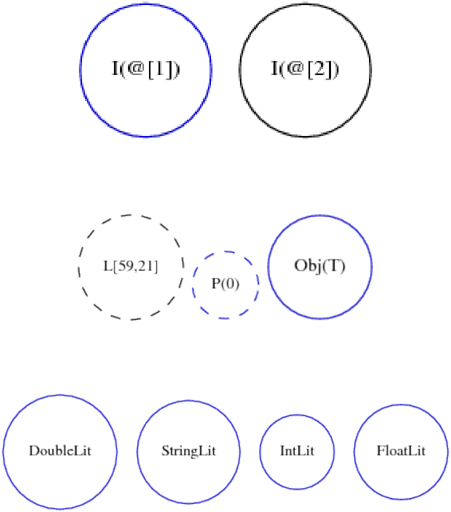
$$
\begin{aligned}
G := \quad \langle \quad & N \subseteq \mathit{Nodes}, \\
& E \subseteq \mathit{Edges}, \\
& \mathit{locVar} \subseteq \mathit{Variables} \times \mathcal{P}(N), \\
& \mathit{RetNodes} \subseteq N \rangle
\end{aligned}
$$

Graph Semantics
Nodes

- We define different kinds of nodes:
  - Inside Nodes (INodes) represent allocated objects.
  - Parameter Nodes (PNodes) represent the current object (**this**) as well as the method parameters.
  - Load Nodes (LNodes) represent nodes that are not yet fully determined.
  - Object Nodes (OBNodes) represent global objects, constructed using the **object** Scala construct.
  - Null Node (NNode) represents the special **null** value.
  - Literal Nodes represent literals of primitive types, such as int, boolean or string
- We also use the following conventions:
  - Nodes that are returned from the procedure have a double border.
  - Nodes that represent singleton objects are drawn in blue.

Graph Semantics

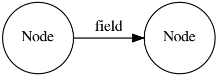
Nodes



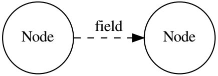
Graph Semantics

Edges

- **Inside Edges** represent write operations on the corresponding fields, they are represented as full edges, with the field's name as label.



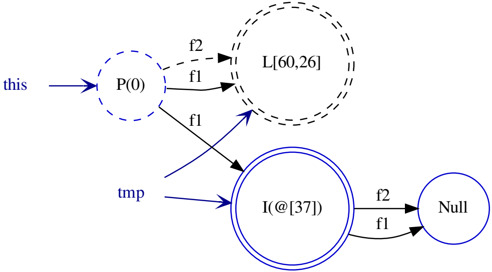
- **Outside Edges** represent the path to follow to reach *load nodes*. They are represented as dashed edges, with the field's name as label.



Graph Semantics

Example

```
class T(var f1: T, var f2: T) {
  def test() = {
    var tmp = this.f2
    if (tmp != null) {
      tmp = new T(null, null)
    }
    this.f1 = tmp
    tmp
  }
}
```



Effects Analysis

Transfer function

Statement st	Transfer Function f
`r = v`	$\langle N, E, locVar[r \mapsto locVar(v)], R\rangle$
`r = new C @p`	$alloc(G, r, C, @p)$
`r = null`	$\langle N \cup \{NNode\}, E,$ $locVar[r \mapsto \{NNode\}], R\rangle$
`r.f = v @p`	$write(G, locVar(r), f, locVar(v), @p)$
`r = v.f @p`	$read(G, locVar(v), f, r, @p)$
`r = v.meth(a1, .., an) @p`	$call(G, r, v, meth, (a_1, ..., a_n), @p)$
`return v`	$\langle N, E, locVar, locVar(v)\rangle$

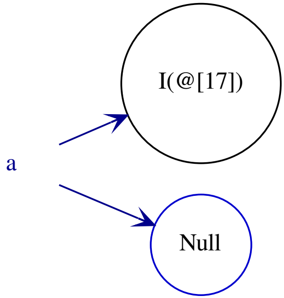
Effects Analysis

Allocations

We use allocation site abstraction

- One node per program point
- We need to determine if this node might represent multiple objects
- We detect loops around allocation sites

```
def test() {
    var a = null
    while(..) {
        a = new A
    }
}
```

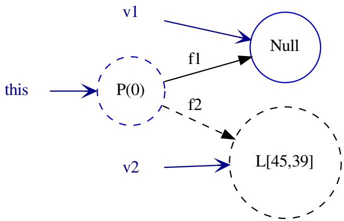


Effects Analysis
Field Reads

When analyzing a field read, we have two cases:

1. The targeted nodes are already determined, and we simply use them
2. The targeted nodes are not determined, we need to introduce a load node

```
class A(var f1: A, var f2: A) {
    def test() {
        this.f1 = null
        val v1 = this.f1
        val v2 = this.f2
    }
}
```



Effects Analysis

Field Writes

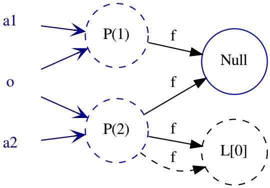
Strong/Weak Update

When a field update completely overwrites the old value, it is said to be **strong**.

One variable may represent multiple objects, we thus cannot always do strong updates. The criteria for a strong update on obj.f = v is

$$|locVar(obj)| = 1 \wedge \forall n \in locVar(obj) \ . \ n.isSingleton$$

```
class A(var f: A) {
    def test(a1: A, a2: A) {
        val o = if (..) a1 else a2
        a1.f = null
        o.f = null
    }
}
```

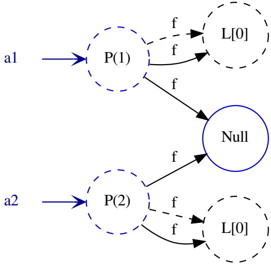


Join Operation

We need to take specific care when joining back two branches:

- Write operations that occur only on one branch are similar to weak updates.
- As usual, if no old value is found, we introduce a load node

```
class A(var f: A) {
    def test(a1: A, a2: A) {
        if (..) {
            a1.f = null
        } else {
            a2.f = null
        }
    }
}
```



Method Inlining

To analyze the effects of a function call, we need to *inline* the callee into the caller.

This inlining is made of two steps:

- Map nodes from the callee to the caller
- Apply write operations

```
class A {
  var f: A = null
  def test(a1: A, a2: A, a3: A) {
    a2.f = this.f
    a1.f = a2
    a1.f = a3
  }
```

```
  def plop() {
    val o1 = new A
    val o2 = new A
    val o3 = if (this!=null) o1 else o2
    o1.test(o3, o2, o2)
  }
}
```

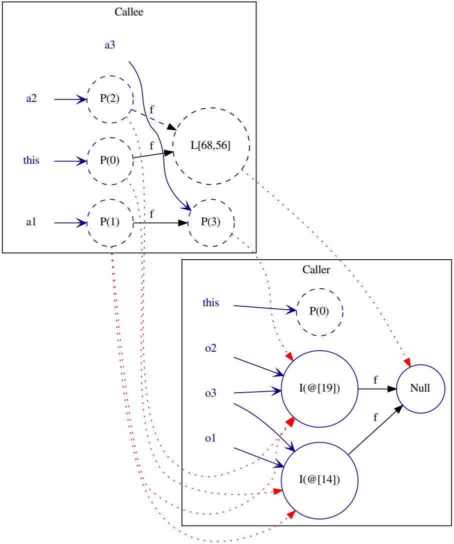
Method Inlining

Mapping Nodes

- First parameter node is mapped to the receiver
- Other parameter nodes are mapped to nodes passed as argument at the call site
- Inside node are mapped to themselves, but with a composed program point.
- Load nodes are mapped by following first inside and then outside edges.
- Literal and object nodes are mapped to themselves.
- Return nodes are mapped to the node presenting the return value.

Method Inlining

Mapping Nodes

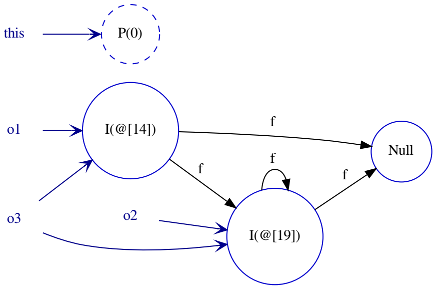


Method Inlining

Applying Writes

- In the callee graph, we have no information about the order in which write operations are performed.
- We thus apply write operations until the graph reaches a fix-point.

Inlining Result:



Purity Analysis

- We check for *observable* purity
  - Look for inside edges (writes) reachable from nodes accessible from *outside* (PNodes, OBNodes, ...)
- If not pure, we compute *modifies clauses*

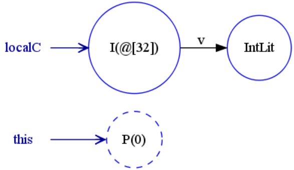
```
class Counter {
  var v = 0
}

class Test1 {
  def fun() {
    val localC = new Counter
    localC.v = 42
  }
}
```

```
class Test2 {
  val c = new Counter
  def fun() {
    c.v = 42
  }
}
```

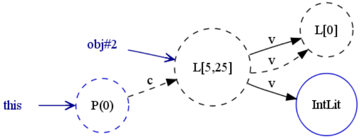
Purity Analysis

```
class Test1 {
  def fun() {
    val localC = new Counter
    localC.v = 42
  }
}
```



Test1.fun: Pure

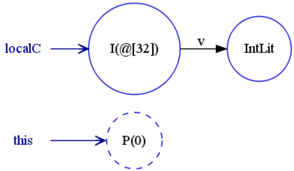
```
class Test2 {
  val c = new Counter
  def fun() {
    c.v = 42
  }
}
```



Test2.fun: Modifies(**this**.c.v)

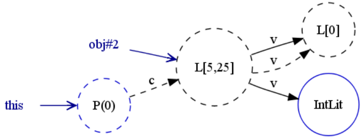
Purity Analysis

```
class Test1 {
  def fun() {
    val localC = new Counter
    localC.v = 42
  }
}
```



Test1.fun: Pure

```
class Test2 {
  val c = new Counter
  def fun() {
    c.v = 42
  }
}
```



Test2.fun: Modifies(**this**.c.v)

Implementation

- Implemented as a Scala compiler plug-in
- Database storage for:
  - Class Hierarchy
  - Intermediate graphs
- Specifying unanalyzable methods:

```
@AbstractsClass("java.lang.Long")
class javalangLong {
  @AbstractsMethod("java.lang.Long.hashCode(()Int)")
  def __hashCode(): Int = {
    42
  }
}
```

Implementation

Difficulties

- Our phase is late in the compiling process: code has been expanded/rewritten.
- No trivial way to access the class hierarchy, required for generating the call graph.
- Compiler becomes brittle for tasks out of its initial scope.
- Dependencies to the Java library are unanalyzable.
- Custom serialization procedure is required to store compiler objects in a database.

Limitations and Future Work

- Exceptions
  - Simple but unsound handling
  - Require some further analysis to handle them correctly without cluttering the CFG.
- Concurrency
  - No support
  - Scala encourages concurrency based on message-passing (actors)
- Higher order functions
  - Currently very imprecise
  - Possible solution: graph-based delaying of method calls
- Annotations
  - Support for more annotations

Related Work I

- Patrice Chalin and Perry R. James. Non-null references by default in java: Alleviating the nullity annotation burden.
  In *ECOOP*, pages 227–247, 2007.
- Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: Beyond strong vs. weak updates.
  In *ESOP*, pages 246–266, 2010.
- Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing.
  ACM Trans. Softw. Eng. Methodol., 17(2), 2008.
- Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing.
  In *ECOOP*, pages 354–378, 2010.

Related Work II

Thomas P. Jensen and Fausto Spoto. Class analysis of object-oriented programs through abstract interpretation.
In *FoSSaCS*, pages 261–275, 2001.

Alexandru D. Salcianu. Pointer analysis and its applications to Java programs.
Master's thesis, MIT, 2001.

Alexandru D. Salcianu. *Pointer Analysis for Java Programs: Novel Techniques and Applications*.
PhD thesis, MIT, 2006.

Olin Shivers. Control-flow analysis in scheme.
In *PLDI*, pages 164–174, 1988.

insane!

Interprocedural Static Analysis Engine for Scala

- Precise pointer and effect analysis
  - Whole-Program but compositional
  - Based on abstract interpretation
  - Interprocedural
- Working Implementation
  - Provided as a compiler plug-in
  - Accepts any Scala code
  - Requires no annotations

Thanks

Questions ?

Additional Material

Class Analysis

Transfer Function

Statement	Transfer Function
`v1 = v2`	$\mathit{facts} = \mathit{facts}[v1 \mapsto \mathit{facts}[v2]]$
`v1 = ex`	$\mathit{facts} = \mathit{facts}[v1 \mapsto \alpha(ex)]$
`obj.f = ex`	*ignore*
`...`	*ignore*

Expression ex	Abstract Value $\alpha(ex)$
`new A`	$\langle \emptyset, \{A\} \rangle$
`null`	$\langle \emptyset, \emptyset \rangle$
`obj.f`	$\langle \{\mathit{type}(\texttt{obj.f})\}, \{\mathit{type}(\texttt{obj.f})\} \rangle$
`rec.meth(..)`	$\langle \{\mathit{type}(\texttt{rec.meth})\}, \{\mathit{type}(\texttt{rec.meth})\} \rangle$

Effect Analysis

Algorithm 1 Lattice Join Operation

function $\bigsqcup(graphs = \{G_1, .., G_n\})$
  if $|graphs| = 1$ **then**
    return x s.t. $x \in graphs$
  else
    $N_{common} \leftarrow \bigcap_i N_i$
    $Pairs_{all} \leftarrow \bigcup_i \{\langle ie.v1, ie.f \rangle \mid ie \in G_i.E \wedge ie \text{ is IEdge}\}$
    $Pairs_{common} \leftarrow \bigcap_i \{\langle ie.v1, ie.f \rangle \mid ie \in G_i.E \wedge ie \text{ is IEdge}\}$
    $N_{load} \leftarrow \{safeLNode(p.v1, p.f, @0) \mid p \in Pairs_{all} - Pairs_{common} \wedge p.v1 \in N_{common}\}$
    $E_{load} \leftarrow \{IEdge(in.v1, in.f, in) \mid in \in N_{load}\} \cup \{OEdge(in.v1, in.f, in) \mid in \in N_{load}\}$
    return $\langle \bigcup_i G_i.N \cup N_{load}, \bigcup_i G_i.E \cup E_{load}, \bigcup_i G_i.locVar, \bigcup_i G_i.R \rangle$
  end if
end function

Effect Analysis

Types associated to nodes

Node Type	Types Associated
INode(A)	$\langle \emptyset, \{A\} \rangle$
LNode(a, f)	$\langle \{type(a.f)\}, \{type(a.f)\} \rangle$
PNode(arg)	$\langle \{type(arg)\}, \{type(arg)\} \rangle$
OBNode(A)	$\langle \emptyset, \{A\} \rangle$
NNode	$\langle \emptyset, \emptyset \rangle$
GBNode	$\langle \{Object\}, \{Object\} \rangle$ (all)
Literal Nodes	$\langle \emptyset, \{type(Literal)\} \rangle$

Effect Analysis

Algorithm 2 Allocations

function $\textsc{Alloc}(\langle N, E, locVar, R\rangle, r, C, @p)$
  $n \leftarrow INode(C, false, @p)$
  $n_{sgt} \leftarrow INode(C, true, @p)$
  if $n_{sgt} \in N$ **then**
    $N_{new} \leftarrow (N \cup \{n\}) - \{n_{sgt}\}$
    $locVar_{new} \leftarrow \{v \mapsto v_{nodes}[n_{sgt} \mapsto n] \mid (v \mapsto v_{nodes}) \in locVar\}$
    $locVar_{new} \leftarrow locVar_{new}[r \mapsto \{n\}]$
  else
    $N_{new} \leftarrow N \cup \{n_{sgt}\}$
    $locVar_{new} \leftarrow locVar[r \mapsto \{n_{sgt}\}]$
  end if
  return $\langle N_{new}, E, locVar_{new}, R\rangle$
end function

Effect Analysis

Algorithm 3 Field Updates

```
function WRITE(⟨N, E, locVar, R⟩, from, f, to, @p, allowStrong)
    isStrong ← ∀n ∈ from. n.isSingleton ∧ |from| = 1 ∧ allowStrong
    N_new ← N
    if isStrong then
        E_new ← E − {ie | ie ∈ E ∧ ie is IEdge ∧ ie.v1 ∈ from ∧ ie.f = f}
        E_new ← E_new ∪ {IEdge(v_from, f, v_to) | v_from ∈ from ∧ v_to ∈ to}
    else
        for n_from ← from do
            previous ← {ie.v2 | ie ∈ E ∧ ie is IEdge ∧ ie.v1 = n_from ∧ ie.f = f}
            E_new ← E
            if previous = ∅ then
                previous ← {ie.v2 | oe ∈ E ∧ ie is OEdge ∧ oe.v1 = n_from ∧ oe.f = f}
            end if
            if previous = ∅ then
                lNode ← safeLNode(n_from, f, @p)
                E_new ← E_new ∪ {IEdge(n_from, f, lNode), OEdge(n_from, f, lNode)}
                N_new ← N_new ∪ {lNode}
            end if
            E_new ← E_new ∪ {IEdge(n_from, f, v_to) | v_to ∈ (previous ∪ to)}
        end for
    end if
    return ⟨N_new, E_new, locVar, R⟩
end function
```

Effect Analysis

Algorithm 4 Field Reads

function $\textsc{Read}(\langle N, E, locVar, R\rangle, from, f, to, @p)$
  $N_{new} \leftarrow N$
  $E_{new} \leftarrow E$
  $pointed \leftarrow \emptyset$
  for $n_{from} \leftarrow from$ **do**
    $previous \leftarrow \{ie.v2 \mid ie \in E \land ie \text{ is IEdge } \land ie.v1 = n_{from} \land ie.f = f\}$
    if $previous = \emptyset$ **then**
      $previous \leftarrow \{ie.v2 \mid oe \in E \land ie \text{ is OEdge } \land oe.v1 = n_{from} \land oe.f = f\}$
    end if
    if $previous = \emptyset$ **then**
      $lNode \leftarrow safeLNode(n_{from}, f, @p)$
      $E_{new} \leftarrow E_{new} \cup \{OEdge(n_{from}, f, lNode)\}$
      $N_{new} \leftarrow N_{new} \cup \{lNode\}$
      $pointed \leftarrow pointed \cup \{lNode\}$
    else
      $pointed \leftarrow pointed \cup previous$
    end if
  end for
  return $\langle N_{new}, E_{new}, locVar[to \mapsto pointed, R\rangle$
end function