Extending Sample with a semantic analysis

Pietro Ferrara
ETH Zurich
pietro.ferrara@inf.ethz.ch

November 15, 2010

Abstract

Sample can be plugged with several heterogeneous analyses. This document presents and explains the interface that has to be implemented in order to develop a new semantic analysis. This is a generic analysis that defines the lattice's structure and the basic semantic operators. The semantic analysis does not have to take care of the heap abstraction.

1 Lattice

```
package ch.ethz.inf.pm.AbstractInterpreter.AbstractDomain

trait Lattice[T <: Lattice[T]] {
  def factory() : T
  def top() : T
  def bottom() : T
  def lub(left : T, right : T) : T
  def glb(left : T, right : T) : T
  def widening(left : T, right : T) : T
  def lessEqual(r : T) : Boolean
}
```

The methods of trait Lattice have the usual meaning of the operators on lattices. Formally,

- **this**.lessEqual(a) returns **true** $\Leftrightarrow$ **this** $\leq$ a
- widening(a, b) returns a ∇ b
- glb(a, b) returns a $\sqcap$ b
- lub(a, b) returns a $\sqcup$ b
- top() returns $\top$

- bottom() returns $\bot$

In addition, factory () returns a new instance of the current domain.

2 SemanticDomain

Trait SemanticDomain is aimed at providing an interface for a generic domain that defines the main semantic operators, e.g., assignment of variables. The user can develop an analysis (e.g., about string values) implementing the methods of this traits. Note that this domain has not to take care about the structure of the heap, since it is combined with an heap analysis that preprocesses the program and substitutes heap accesses with abstract identifiers.

package ch.ethz.inf.pm.AbstractInterpreter.AbstractDomain

```
trait SemanticDomain[T <: SemanticDomain[T]] extends Lattice[T] {
  def getStringOfId(id : Identifier ) : String;
  def setToTop(variable : Identifier ) : T;
  def assign( variable : Identifier , expr : Expression) : T;
  def setParameter( variable : Identifier , expr : Expression) : T;
  def assume(expr : Expression) : T;
  def createVariable ( variable : Identifier , typ : Type) : T;
  def createVariableForParameter( variable : Identifier , typ : Type) : T;
  def removeVariable( variable : Identifier ) : T;
  def access( field : Identifier ) : T;
  def backwardAccess(field : Identifier ) : T;
  def backwardAssign(variable : Identifier , expr : Expression) : T;
}
```

- getStringOfId(x) returns a (short) string representing the abstract value of variable x. This method is used to show the abstract state of a variable to an user.
- setToTop(x) havocs the information tracked on variable x.
- assign(x, expr) returns the state obtained after assigning expr to the variable represented by object x.
- setParameter(x, expr) returns the state obtained after assigning expr to parameter x. Usually, its semantics is the same of assign. The only difference is that this method is invoked at the beginning of the analysis to set the arguments of the analyzed method, while assign is invoked while the method is analyzed, and in particular to abstract the semantics of assignments.
- assume(expr) returns the state obtained after assuming that expression expr holds.

- createVariable(x, t) returns the state obtained after creating a new variable x of type t.
- createVariableForParameter(x, t) returns the state obtained after creating a new parameter x of type t. Usually, its semantics is the same of createVariable. The only difference is that this method is invoked at the beginning of the analysis to create the arguments of the analyzed method, while createVariable is invoked while the method is analyzed, and in particular to abstract the semantics of variable declaration.
- removeVariable(x) removes variable x from the state of the domain.
- access(f) returns the state obtained after accessing field f of the object contained in the current expression. Usually, this action is not supposed to change the abstract state.
- backwardAccess(f) returns the state obtained before accessing field f of the object contained in the current expression. It is the backward counterpart of access.
- backwardAssign(x, expr) returns the state obtained before assigning expr to variable x. It is the backward counterpart of assign.

The trait SimplifiedSemanticDomain provides a simplified version of SemanticDomain. It does not support the backward operators, while access does not modify the state, setParameter(variable, expr) corresponds to assign(variable, expr), and createVariableForParameter(variable, typ) corresponds to createVariable(variable, typ). Then the methods that have to be implemented using SimplifiedSemanticDomain are the following ones:

package ch.ethz.inf.pm.AbstractInterpreter.AbstractDomain

```
trait SimplifiedSemanticDomain[T <: SimplifiedSemanticDomain[T]] extends SemanticDomain[T] {
  def getStringOfId(id : Identifier) : String;
  def setToTop(variable : Identifier) : T;
  def assign(variable : Identifier, expr : Expression) : T;
  def assume(expr : Expression) : T;
  def createVariable(variable : Identifier, typ : Type) : T;
  def removeVariable(variable : Identifier) : T;
}
```

3 NativeMethodSemantics

package ch.ethz.inf.pm.AbstractInterpreter.OORepresentation

```
trait NativeMethodSemantics {
    def applyForwardNativeSemantics[S <: State[S]](
            thisExpr : SymbolicAbstractValue[S],
```

```
            operator : String,
            parameters : List[SymbolicAbstractValue[S]],
            typeparameters : List[Type],
            returnedtype : Type,
            state : S
        ) : Option[S] ;

        def applyBackwardNativeSemantics[S <: State[S]](
            thisExpr : SymbolicAbstractValue[S],
            operator : String,
            parameters : List[SymbolicAbstractValue[S]],
            typeparameters : List[Type],
            returnedtype : Type,
            state : S
        ) : Option[S] ;
}
```

Trait NativeMethodSemantics provides the forward (applyForwardNativeSemantics) and backward (applyBackwardNativeSemantics) semantics of some methods particularly interesting for the analysis we want to implement. These semantics return None if the semantics of the analyzed method is not defined, or Some(state) otherwise (where state is the state obtained after the evaluation of the given method call). The meaning of the arguments of these methods are the following ones:

- thisExpr contains the expression representing the object on which the method is invoked.
- operator contains the string representing the name of the invoked method.
- parameters contains a list of expressions representing the arguments passed to the method.
- typeparameters contains a list of types representing the type arguments passed to the method.
- returnedtype contains the type returned by the invoked method.
- state contains the abstract state before the invocation of the method.

Sometimes after the invocation of the method we want to put an expression on the state. For instance, after the invocation of x.+(y) we want to put on the stack an arithmetic expression representing x+y. In order to create such expressions, class SymbolicAbstractValue provides a method createAbstractOperator(thisExpr, parameters, typeParameters, method, state, returnedType) where method is the value of an AbstractOperatorIdentifier. Once the expression has been created using this method, it can be put on the state invoking method setExpression.

Note that once the expression has been created it can be used to be assigned to a variable or to evaluate a boolean condition. This expression is represented as an instance of the class AbstractOperator, e.g., AbstractOperator(left, args, typepars, AbstractOperatorIdentifiers .==, typ) represents a comparison between left and args.

4 How to run the analysis

```
SemanticAnalysis.analyze(
        "<ClassName>",
        "<MethodName>",
        "<FilePath>",
        <SemanticAnalysis>
);
```

In order to run the analysis in Scala, the user has to call the method analyze on the static object NonRelationalNumericalDomainAndHeapAnalysis. The arguments are the following ones:

1. "<ClassName>": the string containing the name of the class to be analyzed
2. "<MethodName>": the string containing the name of the method to be analyzed
3. "<FilePath>": the complete path of the file to be analyzed
4. <SemanticAnalysis>: an instance of the analysis to be performed

5 Standard domains

Several standard domains are already implemented in Sample, and they can be used to implement new analyses. These implement all the method of Lattice trait.

5.1 FunctionalDomain

This class represents functional domains that relates elements in a domain K to abstract values in V, where V is a lattice. Formally, FunctionalDomain: $K \to V$. The lattice operators are defined as follows (we denote op_V the corresponding lattice operator on the codomain V):

- $\bar{f}_1 \sqcup \bar{f}_2 = \{[x \mapsto \bar{v}] : x \in dom(\bar{f}_1) \cup dom(\bar{f}_2) \wedge \bar{v} = \begin{cases} \bar{f}_1(x) \sqcup_V \bar{f}_2(x) & \text{if } x \in dom(\bar{f}_1) \cap dom(\bar{f}_2) \\ \bar{f}_1(x) & \text{if } x \in dom(\bar{f}_1) \setminus dom(\bar{f}_2) \\ \bar{f}_2(x) & \text{if } x \in dom(\bar{f}_2) \setminus dom(\bar{f}_1) \end{cases} \}$
- $\bar{f}_1 \nabla \bar{f}_2 = \{[x \mapsto \bar{v}] : x \in dom(\bar{f}_1) \cup dom(\bar{f}_2) \wedge \bar{v} = \begin{cases} \bar{f}_1(x) \nabla_V \bar{f}_2(x) & \text{if } x \in dom(\bar{f}_1) \cap dom(\bar{f}_2) \\ \bar{f}_1(x) & \text{if } x \in dom(\bar{f}_1) \setminus dom(\bar{f}_2) \\ \bar{f}_2(x) & \text{if } x \in dom(\bar{f}_2) \setminus dom(\bar{f}_1) \end{cases} \}$
- $\bar{f}_1 \sqcap \bar{f}_2 = \{[x \mapsto \bar{v}] : x \in dom(\bar{f}_1) \cup dom(\bar{f}_2) \wedge \bar{v} = \begin{cases} \bar{f}_1(x) \sqcap_V \bar{f}_2(x) & \text{if } x \in dom(\bar{f}_1) \cap dom(\bar{f}_2) \\ \bot_V & \text{otherwise} \end{cases} \}$
- $\bar{f}_1 \leq \bar{f}_2 \Leftrightarrow dom(\bar{f}_1) \subseteq dom(\bar{f}_2) \wedge \forall x \in dom(\bar{f}_1) : \bar{f}_1(x) \leq \bar{f}_2(x)$

- $\top = \lambda x.\top_V$
- $\bot = \emptyset$

5.2 BoxedDomain

BoxedDomain is a particular case of FunctionalDomain when the domain is the set of identifiers (e.g., variables).

5.3 SetDomain

This class represents domains that contain a set of elements. The lattice operators rely on the set operators. Formally, SetDomain$= \wp(V)$. The lattice operators are defined as follows:

- $\bar{s}_1 \sqcup \bar{s}_2 = \bar{s}_1 \cup \bar{s}_2$
- $\bar{s}_1 \nabla \bar{s}_2 = \bar{s}_1 \cup \bar{s}_2$ (here we suppose that the domain contains a finite number of elements, otherwise the set union would not guarantee the convergence of the analysis)
- $\bar{s}_1 \sqcap \bar{s}_2 = \bar{s}_1 \cap \bar{s}_2$
- $\bar{s}_1 \leq \bar{s}_2 \Leftrightarrow \bar{s}_1 \subseteq \bar{s}_2$
- $\top = V$
- $\bot = \emptyset$

5.4 InverseSetDomain

This class represents domains that contain a set of elements. The lattice operators are the inverse of the ones of SetDomain. Formally, InverseSetDomain$= \wp(V)$. The lattice operators are defined as follows:

- $\bar{s}_1 \sqcup \bar{s}_2 = \bar{s}_1 \cap \bar{s}_2$
- $\bar{s}_1 \nabla \bar{s}_2 = \bar{s}_1 \cap \bar{s}_2$
- $\bar{s}_1 \sqcap \bar{s}_2 = \bar{s}_1 \cup \bar{s}_2$
- $\bar{s}_1 \leq \bar{s}_2 \Leftrightarrow \bar{s}_1 \supseteq \bar{s}_2$
- $\top = \emptyset$
- $\bot = V$

5.5 CartesianProductDomain

This class represents a combination of two domains that have to be lattices. Formally, CartesianProductDomain= $A \times B$. The lattice operators are defined as the pointwise application of the operators on A and B (where op_A and op_B represent the lattice operator op respectively on A and B):

- $(\overline{\mathsf{a}}_1, \overline{\mathsf{b}}_1) \sqcup (\overline{\mathsf{a}}_2, \overline{\mathsf{b}}_2) = (\overline{\mathsf{a}}_1 \sqcup_A \overline{\mathsf{a}}_2, \overline{\mathsf{b}}_1 \sqcup_B \overline{\mathsf{b}}_2)$
- $(\overline{\mathsf{a}}_1, \overline{\mathsf{b}}_1) \nabla (\overline{\mathsf{a}}_2, \overline{\mathsf{b}}_2) = (\overline{\mathsf{a}}_1 \nabla_A \overline{\mathsf{a}}_2, \overline{\mathsf{b}}_1 \nabla_B \overline{\mathsf{b}}_2)$
- $(\overline{\mathsf{a}}_1, \overline{\mathsf{b}}_1) \sqcap (\overline{\mathsf{a}}_2, \overline{\mathsf{b}}_2) = (\overline{\mathsf{a}}_1 \sqcap_A \overline{\mathsf{a}}_2, \overline{\mathsf{b}}_1 \sqcap_B \overline{\mathsf{b}}_2)$
- $(\overline{\mathsf{a}}_1, \overline{\mathsf{b}}_1) \leq (\overline{\mathsf{a}}_2, \overline{\mathsf{b}}_2) \Leftrightarrow \overline{\mathsf{a}}_1 \leq_A \overline{\mathsf{a}}_2 \wedge \overline{\mathsf{b}}_1 \leq_B \overline{\mathsf{b}}_2$
- $\top = (\top_A, \top_B)$
- $\bot = (\bot_A, \bot_B)$

5.6 ReducedProductDomain

The ReducedProductDomain is the CartesianProductDomain on which a reduce() method has to be defined. This method aims at mutually refining the information contained in the two domains.

5.7 SemanticCartesianProductDomain and SemanticReducedProductDomain

SemanticCartesianProductDomain and SemanticReducedProductDomain are extensions of the two previous domains that extends also the SemanticDomain trait. The two domains contained by the product have to extend SemanticDomain too. SemanticCartesianProductDomain simply apply the methods of SemanticDomain on the two domains, while SemanticReducedProductDomain calls also method reduce() after performing the semantic action.

6 Output of the analysis