Uber-lazy Symbolic Execution

Neha Rungta
NASA Ames

Eric Mercer and Benjamin Hillery
Brigham Young University

```
1: procedure UBER-LAZY($a, b$)                ▷ The g.c.d. of a and b
2:     $r \leftarrow a \bmod b$
3:     while $r \neq 0$ do                    ▷ We have the answer if r is 0
4:         $a \leftarrow b$
5:         $b \leftarrow r$
6:         $r \leftarrow a \bmod b$
7:     end while
8:     return $b$                             ▷ The gcd is b
9: end procedure
```

Fig. 1. Euclids algorithm

***Abstract*—The abstract goes here.**

I. Introduction

Text goes here.

II. Bytecodes

PUTFIELD needs to remove path constraints from PC that enforcing equality between references.

Assume: all symbolic locations are concertized lazily. Although the algorithm is not specific to any particular initialization strategy, the presentation assumes a lazy initialization. Extending to lazier initialization may be non-trivial even though this reduction is orthogonal to the lazier reduction (i.e., this reduction should further improve the performance of the lazier algorithm).

Assume: all variables, symbolic or otherwise, are non-primitive (i.e. objects). *Must relax this assumption because you need to do some interesting things with primitives as they relate to getfield.*

This papers uses subsumption, which is expressed as a subtyping relation $\leq$ over types T. For classes C and D, $C \leq D$ iff either $C = D$ or the class declaration for C is `class C extends B` $\{\ldots\}$ for some $B \leq D$. For example, in $A \leq B \leq C \leq D$, D is the supertype, and if you have something that is an instance of A but currently viewed as B, then you can move it toward D in a typecast (up the hierarchy).

Heap locations range over positive natural numbers $H \subseteq \mathbb{N}_{\geq 0}$. Every heap location has a special variable T_h used in constraints over the type of the object stored in the heap location. The varable SH is the set of heap locations created when concretizing symbolic variables. The set of constraints over the type stored in the heap location is given by the function $\mathtt{C}(h)$. Constraints are of the form $T_h \sim T$ or $T \sim T$ where $\sim \in \{\leq, =, \neq\}$. The initial type hierarchy for the program is expressed as a set of constraints C_{init}. The set contains all relationships needed to describe the entire hierarchy.

For a set of constraints C, the function $\mathtt{SAT}(C) \mapsto \{0, 1\}$ returns 1 if the constraints are satisfiable and 0 otherwise. The usual Boolean connectives are used as expected. The function $\mathtt{Type}(h)$ returns the actual type of the object in the heap location and the function $\mathtt{Obj}(h)$ returns the object at the location.

Each variable v is associated with a set of heap locations $H(v) = \{h_0, h_1, \ldots, h_n\}$ that represents an equivalence class (i.e., each heap location yields the same execution path and behavior up to the current point of execution). The representative object for a given variable (i.e., the one that is currently being used by the variable) is given by the function $I(v) \mapsto H(v)$. The set of heap locations and the representative location are part of the meta-deta for the variable. This meta-data follows the variable through the program execution and is appropriately copied on assignment to other variables such that each variable has its own copy of the meta-data that is separate from other copies.

Finally, there is a global variable PC that represents the path constraint along the current path of exploration. This path constraint is used to track relationships between symbolic variables such as equality. Properties of symbolic variables are represented in the path constraint by creating special variables for the representative and the set of represented objects. For a variable v, the special variable I_v is the representative heap opbject and the special variable H_v is the set of associated heap locations. It is assumed the v is alpha-renamed to be unique in the path constraint. Finally, we use the special value `SYM` to denote a symbolic variable that is yet to be initialized.

A. Reference

`GETFIELD`: the bytecode behavior depends on the field operand: concrete, concrete though initialized from symbolic, and symbolic. Each case is enumerated:

1) Referencing a concrete field: the bytecode has default behavior returning the field.
2) Referencing an initialized field from a symbolic variable (i.e., the base type for the field is initialized from a symbolic object): the bytecode may have multiple outcomes; it partitions the equiavalence class to group heap locations with objects that have common values for the field.
3) Referencing a symbolic variable that has yet to be initialized: the bytecode has two outcomes: one that returns `null` and another that builds the potential equivalence class, chooses a representative location, and returns that location.

Consider the code

```
// The declared type of f is F
T t = b.f;
```

For the case where `b.f` is an already initialized symbolic variable, $h = \mathtt{I}(\mathtt{b})$, $\mathtt{H}(\mathtt{b})$ is partitioned into disjoint sets, $S_0, P_1, \ldots, P_n$, with $n+1$ partitions, or choices. The first set S_0 is a special set that includes h, the representative object for b, and any members of the equivalence class that either have the same field value for `b.f` or the field value is a symbolic variable that has yet to be initialized:

$$\begin{aligned} S_0 \;\; = \;\; & \{h_i \mid h_i \in \mathtt{H}(\mathtt{b}) \;\wedge \\ & (\mathtt{Obj}(h).\mathtt{f} = \mathtt{Obj}(h_i).\mathtt{f} \;\vee\; \mathtt{Obj}(h_i).\mathtt{f} = \mathtt{SYM})\} \end{aligned}$$

For this special case of S_0, $\mathtt{H}_0(\mathtt{b}) = S_0$, and $I_0(\mathtt{b}) = h$ where the subscript indicates the choice number in the choice generator (i.e., the partition size may change but not the representative heap location). The other partitions group common values of the field such that

- $\forall h_i, h_j \in P_i,\ \mathtt{Obj}(h_i).f = \mathtt{Obj}(h_j).f$
- $\mathtt{H}_i(\mathtt{b}) = P_i$
- $\exists h \in P_i,\ \mathtt{I}_i(\mathtt{b}) = h$

The partitions are maximal and represent a unique value that has been created thus far in the program execution. The non-initialized symbolic members of the partition all belong to S_0 as the original representative heap location h captures that these other aliases were intended to have the same field value for field f before the split (i.e., the value is assigned programatically but the change was only reflected in the representative heap location). Once the choice generator is created over the different partitions, the bytecode returns the requested field value of the representative as expected.

Returning to the third behavior of the bytecode, the case in which the accessed field has yet to be initialized, then the bytecode follows lazy initialization creating a `null` instance, a new instance that is the representative, and the alias set. When creating the alias set, the new instance should be included in the set, as well as any prior object created in concretization of symbolic variables that is type compatible with the new instance. Recall that SH is set of locations in the symbolic heap and C_{init} is the set of constraints describing the type hierarchy, assuming h is the heap location of the new instance of the type, then

- $\mathtt{I}(\mathtt{b.f}) = h$
- $\forall h_i \in SH, \mathtt{SAT}(C_{\text{init}} \cup \{T_h \leq \mathtt{Type}(h_i)\}) \rightarrow h_i \in \mathtt{H}(\mathtt{b.f})$
- $\mathtt{C}(\mathtt{b.f}) = C_{\text{init}} \cup \{T_h \leq \mathtt{Type}(h)\}$

The C_{init} set constains relationships in the class hierarchy with the correct sub-types and super-types as they relate to the delcared type of the object.

GETSTATIC: the bytecode is handled similarly to `GETFIELD`.

ALOAD: the bytecode is handled similarly to `GETFIELD`.

B. Comparison

IF_ACMPEQ: the bytecode may return both the `true` and `false` values, and it must possibly refine the set of represented concretizations and mutate the heap location of the object involved in the bytecode according to the returned outcome. Consider the code

```
if (a == b) {
   // code...
}
```

There are two cases that need to be considered to determine the outcome of the bytecode:

1) $\mathtt{H}(a) \cap \mathtt{H}(b) = \emptyset$: the bytecode returns `false` and nothing further is requred.
2) $\mathtt{H}(a) \cap \mathtt{H}(b) \neq \emptyset \wedge \mathtt{SAT}(PC \cup \{I_a = I_b, H_a = H_b\})$: the bytecode may return either `true` or `false` and a choice generator needs to be created.

The choice generator for the compare bytecode is more complex than for other bytecodes because it must create representative sets without enumerating all possible outcomes using the path constraint. For the case `true` outcome

- $PC = PC \cup \{\mathtt{I}(a) = \mathtt{I}(b), \mathtt{H}(a) = \mathtt{H}(b)\}$
- $\mathtt{H}(a) = \mathtt{H}(b) = \mathtt{H}(a) \cap \mathtt{H}(b)$
- $\mathtt{I}(a) \in \mathtt{H}(a) \cap \mathtt{H}(b) \rightarrow \mathtt{I}(b) = \mathtt{I}(a) \vee \exists h \in \mathtt{H}(a) \cap \mathtt{H}(b)\ .\ \mathtt{I}(b) = \mathtt{I}(a) = h$

In essence, in the case where two variable reference the same object, the path constraint and sets are modified to represent the new restriction. The `false` outcome is handled similarly with a few notable exceptions on the path constraint and the represented set.

- $PC = PC \cup \{\mathtt{I}(a) \neq \mathtt{I}(b)\}$
- $\mathtt{I}(a) = \mathtt{I}(b) \rightarrow \exists h \in \mathtt{H}(b)\ .\ h \neq \mathtt{I}(a) \wedge \mathtt{I}(b) = h$

IF_ACMPNE: the bytecode is handled similarly to `IF_ACMPEQ`.

C. Invocation

INVOKEVIRTUAL

When we come to an invoke virtual you have to look for all the specialized implementations of the method, creating choices with symbolic locations of various "actual types". The number of choices will be equal to the number of specialized implementations of the method. When you create a choice on a specialization, you need to update the "actual type" field in the symbolic location. The "current cast" does not need to change. The number of types that the symbolic location cannot be will also be updated according to the "actual type" field. The number of types that the symbolic location cannot be will be updated with the types of the other specializations since invoking a specialization associated with a type implies that the object cannot be the types containing the other specializations.

D. Checking Types and Casting

INSTANCEOF: the bytecode may return both the `true` and `false` values when dealing with initialized symbolic variables, and it must possibly refine the equivalence class for the represented object referenced by the variable and mutate the contents of the heap location of the object involved in the bytecode according to the returned outcome. The bytecode implements the default bahvior when the operand is concrete and not an initialized symbolic variable. For the rest of the discussion, assume the operand is an initialized symbolic variable. Consider the code

```
if (a instanceof C) {
    // code...
}
```

There are two cases that need to be considered to determine the outcome of the bytecode where $h = \mathtt{I}(a)$ is the representative object of the equivalence class:

1) $\mathtt{Type}(h) = C$: the bytecode returns `true` and nothing further is required as the type stored in the heap location is C.
2) $\neg\mathtt{SAT}(\mathtt{C}(h) \cup \{T_h \leq C\})$: the bytecode returns `false` and nothing further is requred as the current constraints on what is in the heap location restrict it from being of type C.
3) $\mathtt{SAT}(\mathtt{C}(h) \cup \{T_h \leq C\})$: the bytecode can return either `true` or `false` requiring a choice generator.

The `true` outcome for the choice generator in clause (3) is

- $\mathtt{C}(h) = \mathtt{C}(h) \cup \{T_h \leq C\}$
- $\mathtt{Type}(h) = C$
- $\mathtt{H}(a) = H'$ where $H' = \{h_i \mid h_i \in \mathtt{H}(a) \wedge \mathtt{SAT}(\mathtt{C}(h_i) \cup \{T_{h_i} \leq C\})\}$

The second statement indicates that the actual type in the heap location h needs to change. As such, the object is mutated to be an instance of C. This mutation retains all fields and values from the previous object and only adds new fields for type C. The last statement refines the equivalence class to exclude any heap locations that cannot be considered something of type C.

For the `false` outcome of the generator, $\mathtt{C}(h) = \mathtt{C}(h) \cup \{C \leq T_h, T_h \neq C\}$. Unlike the `true` outcome, the `false` outcome retains the entire equivalence class and does not need to mutate any heap entries.

CHECKCAST: the bytecode is syntactic sugar for

```
if (! (obj == null  ||
obj instanceof <class>)) {
     throw new ClassCastException();
}
// if this point is reached, then object
// is either null, or an instance of <class>
// or one of its superclasses.
```

Please see the `IFNULL` and `INSTANCEOF` bytecodes for details. If the exception is thrown, then JPF will catch the unhandled exception as per its normal behavior.

E. Programs to consider

- `TestGetfieldSplit.java`: checks alias equivalence classes when assigning to initialized values.

III. Conclusion

The conclusion goes here.

Acknowledgment

The authors would like to thank...

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.