Technical Disclosure Commons

Defensive Publications Series

July 31, 2017

Sub-type checking at runtime with dynamic class loading

Igor Murashkin

Mingyao Yang

Sub-type checking at runtime with dynamic class loading

ABSTRACT

Subtype checking is common in computer programs to determine whether an object or an array is a subtype of a class. This check is typically performed at runtime. Traditional techniques for subtype checking are slow since such techniques often need to traverse an entire class hierarchy to return a result. The techniques of this disclosure offer a faster alternative. Each class in the hierarchy is encoded. Leading bits or prefix in the encoding are used to determine, during dynamic class loading, whether a subclass relationship exists between a source class and a target class.

KEYWORDS

● Subtype checking
● Dynamic class loading
● Fast path check
● slow path check

BACKGROUND

Computer programs can have many subtype checks, which in languages such as Java, are in the form of either an instanceof test or a checkcast. For example, in Java, subtype checking is used heavily in collections, downcasting from object/interfaces, implementing Object.equals(), serialization, reflection, etc. A checkcast may be hidden in the source code as shown below.

```
ArrayList al; MyObject myobject = al.get(0);
```

The front-end compiler inserts a hidden `checkcast(MyObject)al.get()`. A checkcast is almost always successful in practice, otherwise an exception will be thrown. Instanceof test, on the other hand, may be either positive or negative in practice, although it is likely to be positive.

A subtype check may be for checking if an object/array is a subtype of a class or an interface.

```
boolean instanceof(Class source, Class target) {
do {
if (source == target) {
return true;
}
source = source.getSuperclass();
} while (source != null);
return false;
}
```

Traditional subtype checking (example shown above) is not efficient since it requires traversing the superclass chain. Some common techniques to speed up subtype checks are described below:

1. **Binary matrix:** A binary matrix M[x,y] which holds a 1 if x <: y and 0 otherwise. The downside of this approach is the space needed for encoding the matrix, which is usually computed statically.

2. **Schubert's Numbering** (also called Relative Numbering) involves a depth-first traversal of the class hierarchy that numbers each class c with a pair of pre-order id (l(c)) and max-pre-order id of the subtree(r(c)). Then, `x <: y if and only if: l(y) = l(x) and r(x) = r(y)`. The downside of this approach is that full class hierarchy is used to compute the pair of ids statically.

3. **Display based subtype checking** works well for dynamic class loading since display can be populated when a class is linked. Consider the following class hierarchy,

```
class A {}
class B extends A {}
class C extends B {}
```

The display for class C of this class hierarchy is [Object.class, A.class, B.class, C.class]. A display can be populated in the class object during class linking and a depth field can be added to keep the distance of a class from Object in its superclass chain (Object is at depth 0).

```
boolean instanceof(Class source, Class target) {
if (target.depth > source.depth) {
return false;
}
return source.display[target.depth] == target;
}
```

This method is essentially a range check coupled with loading a display element and comparing. To avoid the range check, the display can be padded with nulls to a fixed length of DISPLAY_LENGTH. When target.depth is less than DISPLAY_LENGTH, the method uses the below check.

```
boolean instanceof(Class source, Class target) {
return source.display[target.depth] == target;
}
```

By choosing a suitable value for DISPLAY_LENGTH, this method can cover most of the subtype checks. The downside is the memory requirement as each class needs to carry a display of DISPLAY_LENGTH classes.

4. **Bit-Vector Encoding of Tree Hierarchies:** Encoding class hierarchy with bit vectors can be used to minimize required bits. The downside is that the full class hierarchy is still used to compute the bit vectors, which does not work in the presence of dynamic class loading.

DESCRIPTION

This disclosure describes techniques for checking subtype of a class efficiently and overcomes some limitations of traditional subtype checks described in the background.

Bit-String Path Encoding of Dynamic Tree Hierarchies

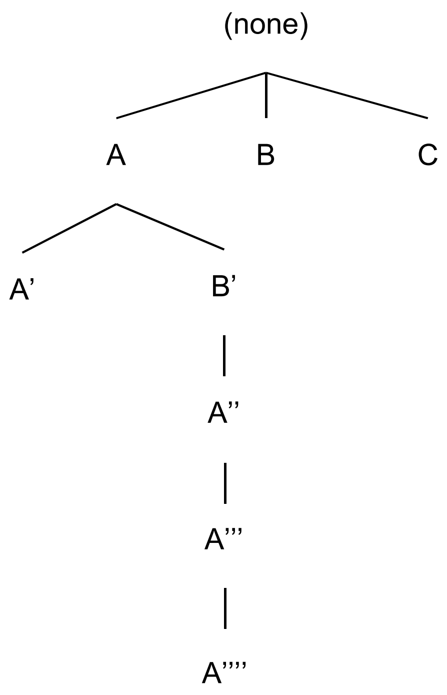

Fig. 1

Any node in a tree can obtain its path (from the root to the node) by concatenating the path of the parent with that of the current node. Each node can be annotated with a "sibling-label," which is some value unique amongst all of the node's siblings. As a special case, the root is empty. Fig. 1 is an illustration of a tree with the root being empty as the special case.

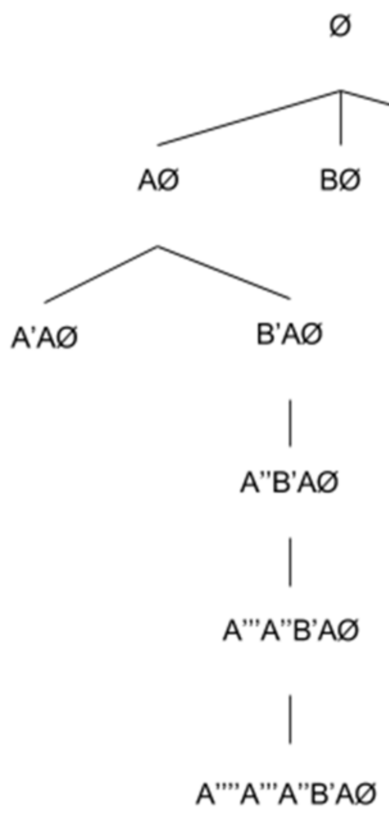

Fig. 2

Given the sibling-labels, the path can be encoded from any node to the root by starting at the node and going up all the way to the root, marking each node with this "path-label." The special character Ø denotes "end of path." Fig. 2 shows a sample tree with path-labels encoded.

Also, based on the above "path-label," an O(1) expression can be used to denote if any two nodes are an offspring of the other as given below.

```
x <: y if and only if: suffix(x,y) == y
```

In the above example, suffix(x,y) means the suffix of x that is as long as y (right-padded with Øs if x is shorter than y).

```
suffix(x,y) := x(x.length - y.length .. 0] + repeat(Ø,
max(y.length - x.length, 0))
```

A few generalities apply:

- There are at most D levels in the tree.
- Each level L has an alphabet A and maximum number of nodes is defined by |A|.
- The alphabet A can be a subset, superset, equal, or unique with respect to other alphabets without loss of generality. In practice, the alphabet at a certain level is likely a subset of the previous level's alphabet with the assumption that most classes have less children than the classes at the next higher level.
- The "sibling-label" doesn't need to be stored as an explicit value but can be a temporary value that is determined when visiting every immediate child of a node. Only the "path-label" needs to be actually stored for every node.

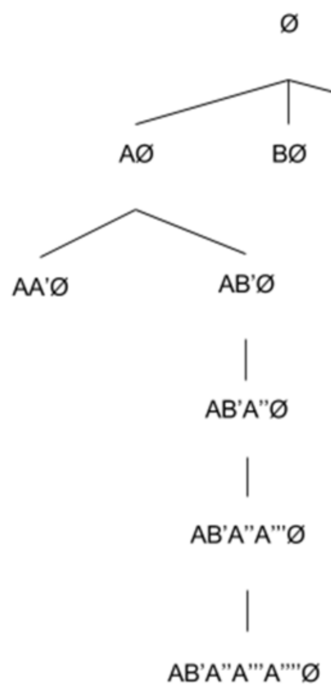

Fig. 3

The path can also be reversed and can use a prefix instead of a suffix to define the parent-child relationship. Fig. 3 displays a tree with path labels identified from root to node.

Also, based on the above “path-label,” an O(1) expression can be used to denote if any nodes is an offspring of another node as given below.

```
x <: y if and only if: prefix(x, y) == y
```

In the above example, prefix(x,y) means the prefix of x that is as long as y (left-padded with Øs if x is shorter than y).

```
prefix(x,y) :=  x[0 .. y.length) + repeat(Ø,
max(y.length - x.length, 0))
```

In a dynamic tree, new nodes can be inserted at any time. This means that if a minimal alphabet is selected to contain the initial tree hierarchy, later node insertions will be illegal because of a lack of room to encode the path. Below is one such example.

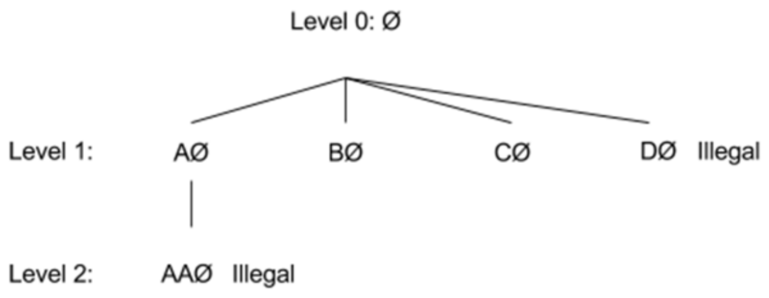

Fig. 4

In this simple example with alphabet {A, B, C} and a maximum of one level, attempting to insert sibling “D” at Level 1 will be illegal because the alphabet {A, B, C} does not contain D and inserting an extra node with the label A, B or C would mean the “sibling-label” is no longer unique. Attempting to insert “AAØ” is illegal because level 2 is greater than maximum level 1. A possible solution is to revisit the entire graph, select a larger alphabet so that every “sibling-label” is unique, choose a larger maximum level count and store the updated “path-label” accordingly. A more common approach rather would be to select a set of alphabet and maximum level values statically with large enough sizes. For example, below alphabet and maximum value configuration allows up to 4 levels with each successive level having one less sibling.

```
Alphabets = {{A,B,C,D}, {A,B,C}, {A,B}, {A}}
Max Levels = |Alphabets|
```

Attempting to insert a new node which does not fit into that level's alphabet can be represented by re-using the path-label of the parent. Such a path-label is considered truncated (because it can only have a prefix of the full path from the root to the node).

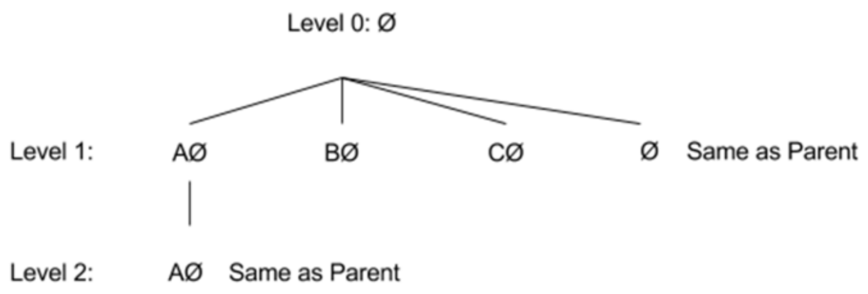

Fig. 5

The updated relationship for an offspring in Figure 5 is given below.

```
x <: y if and only if:
if !truncated_path(y): return prefix(x, y) == y
else: return slow_check_is_offspring(x, y)
```

In this case, as long as there is no truncated path, the current O(1) method is followed. Alternatively, a non-O(1) method is implemented. Any semantically equivalent way to check that a "sibling-label" is not unique can also be used to implement the truncated_path() function. Below is an example of such a function.

```
truncated_path(y) := return y == parent(y)
```

Similarly, any non-O(1) method to check whether x is an offspring of y can be used. Below is the traversing superclass hierarchy solution that can be used as the slower method to check the relationship. In addition, it doesn't matter if the "x" from the below equation is a unique sibling or not - the relation will still be correct.

```
slow_check_is_offspring(x,y) := if not x: return
false
return x == y || recursive_is_offspring(parent(x),y)
```

Consider an example that can be applied to most modern hardware today. This is a binary alphabet 01,10,11 and Ø == 0. The disclosed techniques work for a class hierarchy that can update dynamically, perform subtype check in constant time in most cases, and fall back on a traditional slow path for uncommon cases. Moreover, the techniques can be implemented for managed programming language runtime with even one or two 32-bit values for each class object.

The disclosed techniques assign each class an id with a number of bits termed label-bits. The label-bits are a physical representation of the path-label, with Øs being padded to fill up all the bits in an id. The assignment of label bits meets the following requirements:

- Label-bits of a subclass have the superclass's label-bits as the leading bits.
- Sibling classes have different label-bits unless they both have their parent class's label-bits.
- Bits other than label-bits required for the class level of that depth are padded with 0's.

For the alphabet set {01, 10, 11} with a maximum of three levels, if there are a total of 6 bits for label-bits with two bits for each additional class hierarchy level, one way to assign label-bits is:

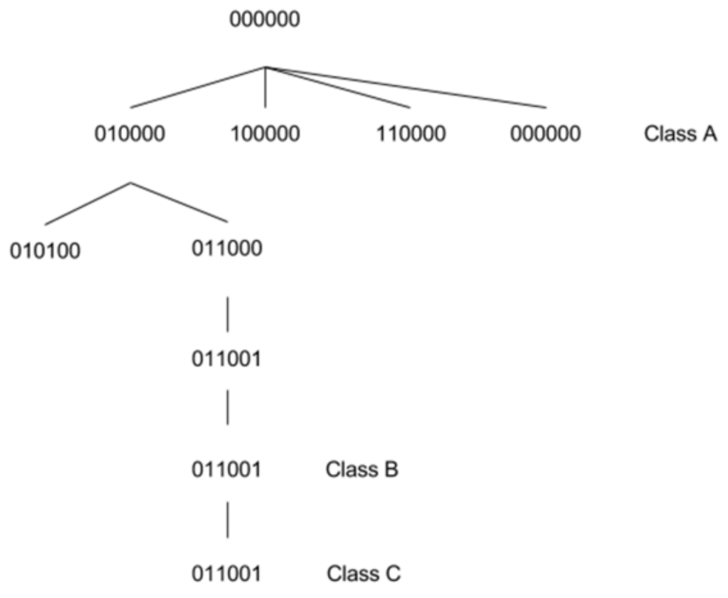

Fig. 6

If the label-bits of a class are the same as those of its parent class, then there are not enough bits to assign to the subclass - either there are too many siblings or the depth of the class is too much. If a class cannot get its own label-bits, all the label-bits in the subtree starting at that class will get the same label-bits. In the above figure, A, B and C cannot be assigned unique label-bits. However, the label bits of A/B/C can still be used to check against a target class with unique label-bits.

The techniques refine the definition of prefix(x, y) with an optimization:

```
prefix(x,y) := label-bits(x) & mask(y),
```

where mask(y) consists of all 1's corresponding to the positions of y's assigned label-bits.

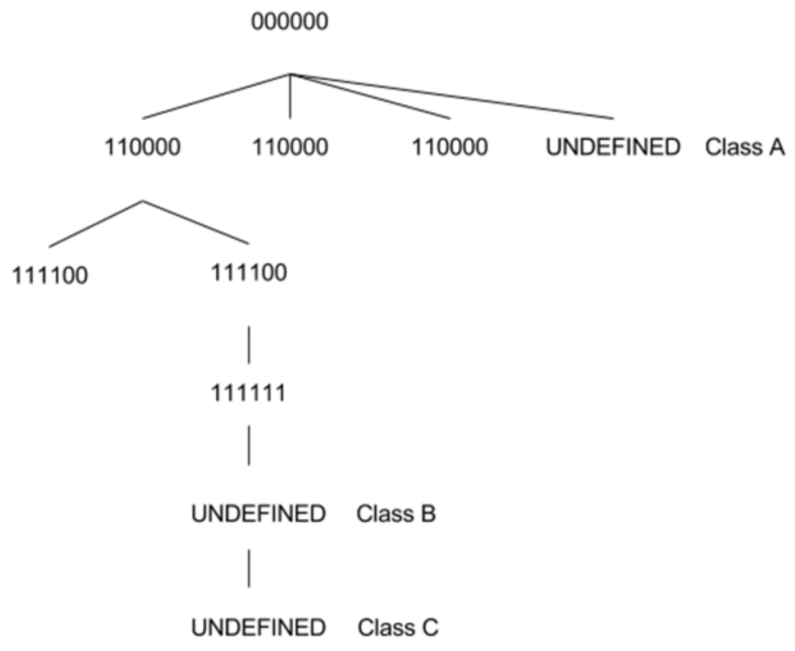

Fig. 7

An example of the structure of the mask for each node of the tree in Fig. 6 is shown in Fig. 7.

As the prefix function is only defined when the label-bits of node y are not truncated, mask(y) is only defined for untruncated paths. Intuitively, a class x is a subclass of class y if x's label-bits extend y's label-bits. This is also implied by general definition of offspring relationship below:

```
  x <: y if and only if:
     if truncated_path(y): return prefix(x, y) == y // O(1)
     else: return slow_check_is_offspring(x, y) // worse than O(1)
```

Efficient Generated Code at Compile Time

The check truncated_path(y) is often performed at compile time (the class is already inserted into the tree hierarchy and it already has a unique “sibling-label”). The final compile-time definition is (only if y is known not to be a truncated path):

```
        x <: y := return label-bits(x) & mask(y) == label-
    bits(y)
```

This can be implemented very efficiently in the compiled code since mask(y) and label-bits(y) are usually both compile-time constants if class y is resolved at compile time. Also, there is no need to even fetch class y in the compiled code. Display-based approach, on the other hand, needs y for comparison with the display entry.

As long as the target class has its own label-bits, the compiler generates a fast path check. When a class fails to get its own label-bits and is also the target of a subtype check, the compiler does not follow the label-bits check but instead implements superclass chain traversal slow path check instead. A class without unique label-bits is not mistakenly treated as being a subtype of another class with the same label-bits as the slow path check returns the correct result.

Bitstring Length Selection

When 64-bits are available for encoding label-bits, 15 bits are assigned to classes that extend java.lang.Object. This allows for encoding of label-bits for 32k - 1 classes that directly extend java.lang.Object. For successive class depth levels, 9, 8, 8, 8, 8, 4, 4 bits can be assigned. As a reference, the OpenJDK display length is set to 8, which is greater than the required length for all of the classes in rt.jar. If the target class is within three levels of the java.lang.Object, loading the 32-bit value of label-bits (source), the mask and comparing them with some constants is sufficient. Alternatively, for a target class beyond three levels, one more 32-bit value is loaded for the remaining label-bits. The additional memory load (of 32 bits) is inexpensive, e.g., if it is in the same cache line and if the label-bits are placed together. For 64-bit systems, the full 64-bit value of label-bits can be loaded in one shot.

Often, in existing implementations of a managed programming language runtime, the class object representation includes some fields that are partially occupied but still need 32-bits due to alignment requirement. In such cases, the remaining unused bits can be used for label-bits. For example, if only 4 bits are used for encoding the class status, then the remaining 28 bits can be used to encode label-bits. By adding a 32-bit word for label-bits, 60 bits become available.

During class loading, label-bits are assigned to the newly loaded class if available. An increment of 1 to the label-bits value of the last sibling is assigned as the label-bits value of the new class. Alternatively, 00..001 (number of 0's depends on the label-bits length of the depth level of the class) is appended to the label-bits of the parent if it is the first sibling that is assigned label-bits. The increment/appending cannot overflow, otherwise the class is assigned the same label-bits as its parent resulting in a truncated label-bits case.

Efficient Insertion Optimization

In order to track the assigned label-bits of the last sibling, either a data field is reserved in the parent class that points to the last sibling, or some bits are reserved in the parent to remember label-bits value of the last sibling. The latter is economical, since for the first few class depth levels, there are enough label-bits unused for prefix bits. For example, as only the first 32-bit value is needed for label-bits of the first few depth levels, part of the second 32-bit value can be used to encode the assigned prefix bits of last sibling. As the number of prefix bits assigned to successive levels of a class hierarchy usually decrease progressively, towards the end, the overhead to store the assigned prefix bits of the last sibling in the label-bits is no more than a couple of bits.

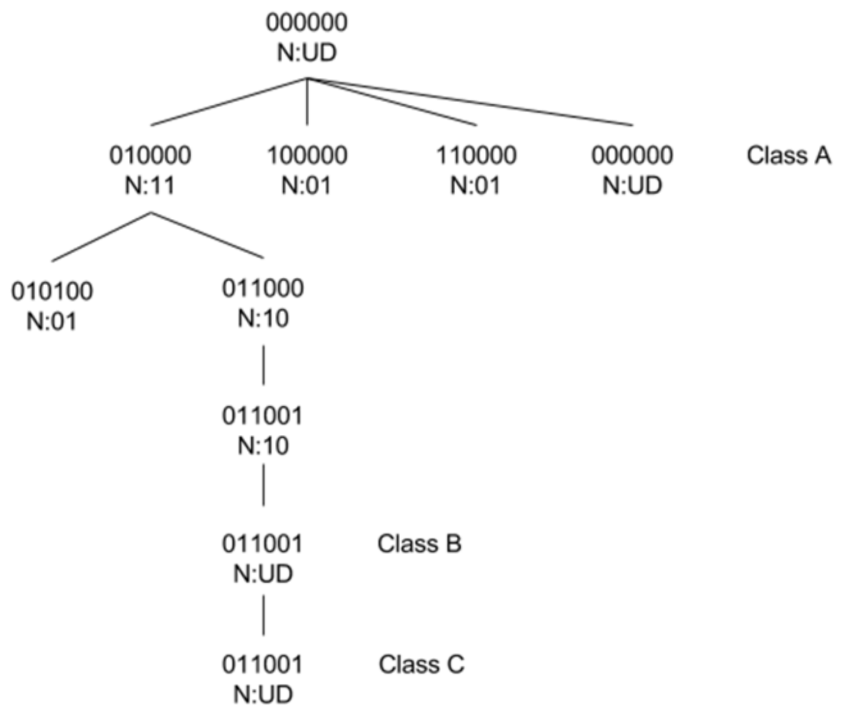

Fig. 8

In the above example, when inserting a new child for a node K, the “next value” (N:##) is used as the “sibling-label.” Note that the “next value” is undefined (N:UD) when the “path-label” is truncated (as in the examples for classes A, B, C).

Minimizing Truncated Encodings Optimization

The assignment of label-bits for each depth level of the class hierarchy can be tuned by experimenting with applications and libraries. For embedded systems, applications generally have a small number of classes and a well-tuned label-bits assignment can generate fast paths for most of the subtype checking. Also, a more complicated encoding, which allows variable length label-bits encoding for a class is possible. In this case, instead of using fixed length label-bits for a class, the label-bits length is calculated for each level depending on its depth. Even if there are many classes, profile-guided compilation can be used to load popular target classes used for subtype checking early during runtime to ensure assignment of valid label-bits.

CONCLUSION

This disclosure describes techniques to check the subtype of a dynamically loaded class during runtime. Specifically, the techniques encode the class hierarchy with data that is usable to determine whether a target class is a subclass of a source class. For example, this data includes information about the superclasses of each class to help decide the existence of a subtype relationship. In uncommon situations, the traditional superchain traversal method is still used as a fallback. The techniques provide optimizations that improve the effectiveness and execution of the subtype checking.