Extensible Encoding of Type Hierarchies

Hamed Seiied Alavi
IC, EPFL, Lausanne, Switzerland
hamed.alavi@epfl.ch

Seth Gilbert
IC, EPFL, Lausanne, Switzerland
seth.gilbert@epfl.ch

Rachid Guerraoui
IC, EPFL, Lausanne, Switzerland
rachid.guerraoui@epfl.ch

Abstract

The *subtyping test* consists of checking whether a type t is a descendant of a type r (Agrawal et al. 1989). We study how to perform such a test efficiently, assuming a *dynamic* hierarchy when new types are inserted at run-time. The goal is to achieve time and space efficiency, even as new types are inserted. We propose an *extensible* scheme, named *ESE*, that ensures (1) efficient insertion of new types, (2) efficient subtyping tests, and (3) small space usage. On the one hand ESE provides comparable test times to the most efficient existing *static* schemes (e.g., Zibin et al. (2001)). On the other hand, ESE has comparable insertion times to the most efficient existing *dynamic* scheme (Baehni et al. 2007), while ESE outperforms it by a factor of 2-3 times in terms of space usage.

Categories and Subject Descriptors D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features

General Terms Languages, Algorithms

Keywords Subtyping Test, Dynamic Loading

1. Introduction

One of the most frequent operations in object-oriented programs is the *subtyping test*:

- Given an object *O* of a type *t*, is *O* also an instance of a type *r* i.e is t a subtype of r?

Such tests, also known as *type inclusion tests*, are usually invoked by explicit requests of programmers when they use linguistic constructs like `instanceof` (Java) (Gosling et al. 2005), `is` (C#) (Hejlsberg et al. 2001) or `iskindof` (SmallTalk) (Goldberg et al. 1983). They are also sometimes inserted by the compiler in the contexts of type casting, array casts and exception handling (Gosling et al. 2005; Hejlsberg et al. 2001). Since the cost of subtyping tests has a significant effect on the overall performance of systems, considerable attention has been paid to optimizing the test. Typically, the main challenge has been to preprocess the type hierarchy and producing a data structure that encodes the subtyping relationships in an efficient manner. The efficiency here is typically defined in terms of test time and required memory space.

The most obvious encoding scheme is a *binary matrix* (BM) in which $M[i, j] = 1$ iff i is a subtype of j; this results in a constant $O(1)$ test time, but requires $O(n^2)$ space, which is considerably big in large type hierarchies (e.g. JDK 1.3 with around 5500 types takes 3.8MB (Zibin et al. 2001)). If the type hierarchy is simply encoded as a tree, requiring $O(n)$ space, the worst-case time for a test may be $O(n)$. More elegant schemes have improved this trade-off between time and space complexity. *Relative numbering* (Schubert et al. 1983) guarantees both constant test time and minimal encoding size of $\log(n)$ bits per type for single-inheritance type hierarchies. In the case of multiple-inheritance type hierarchies, PQE (Zibin et al. 2001) provides constant test time and, in comparison to other algorithms, provides the smallest encoding size (to the best of our knowledge).

Maybe surprisingly, relatively little attention has been paid to the handling of *dynamic type loading*, i.e. the ability to add new types at run-time in the hierarchy. (See Baehni et al. (2007) for the only exception we know of, and which we discuss of length in Section 6.) Dynamic type loading is a key requirement of modern systems (Ajmani et al. 2006) and is promoted in Java (Gosling et al. 2005) and .Net (Microsoft 2005). In fact, even more traditional languages like C++ and SmallTalk support dynamic (or incremental) linking (Stroustrup 2004; Goldberg et al. 1983). One way to perform subtyping tests is to use traditional static solutions and re-encode the hierarchy for each new type insertion. Clearly, this leads to slow insertion time. In this paper we consider the cost of type insertions as an important metric in devising the subtyping test.

In short, this paper proposes a new encoding algorithm that performs insertions efficiently while preserving the performance of the best known static schemes in terms of test time and encoding size. We focus on designing an encoding algorithm that works well for real-world type hierarchies (e.g. JDK), as validated by experimental evidences. (See Section 6.)

One reason why inserting new types is expensive is that most algorithms need to update the encoding of old types whenever a new type is inserted (Zibin et al. 2001, 2002; Cohen 1991; Vitek et al. 1997). This modification takes considerable time and thus slows down the overall speed of the system significantly. For example PQE (Zibin et al. 2001) which outperforms all the other subtyping algorithms in terms of encoding size and query time, requires the reconstruction of almost the entire encoding for each newly added type. PQE partitions the type hierarchy into a set of types called *slices* using a sophisticated algorithm. Since this algorithm requires the whole type hierarchy as input, it recomputes the slicing for each newly inserted type, which is quite time consuming for large type hierarchies (e.g. JDK 1.3: 113 msec). This is so in a centralized setting. In distributed settings the problem becomes even more drastic since any modification to the encoding has to be broadcast over the network to maintain the consistency, i.e. to ensure a unique encoding throughout the distributed system. Such mechanism is extremely costly as it involves reaching agreement in a distributed setting (Lynch 1996).

POPL'08, January 7–12, 2008, San Francisco, California, USA.

The reason why traditional encoding schemes require reconfiguration upon the addition of new types is actually easy to see. In these schemes, the encoding of a type depends on its subtypes (Zibin et al. 2001, 2002; Cohen 1991; Vitek et al. 1997) and this requires modifying the old encoding upon insertion of new types. The solution proposed by Baehni et al. (2007), called *DST*, avoids this reconfiguration by encoding each type based on its supertypes rather than its subtypes. Like PQE, DST partitions the hierarchy into a set of slices. The number of slices is the major factor influencing the required memory space, i.e. the number of slices used directly impacts the size of the memory required. The problem with DST is in (1) its partitioning algorithm, which does not provide the optimal slicing and (2) its insertion algorithm which generates too many new slices upon insertion of new types. Our ESE solution improves DST in terms of space usage. More precisely, ESE partitions a hierarchy into fewer slices than DST, requiring less space. On the other hand, upon addition of new types, ESE's insertion algorithm creates new slices less often than DST. This fact further underlines the difference between ESE and DST in terms of space usage.

Time and space efficiency. Our ESE solution provides comparable performance with the most efficient static subtyping methods we know of, with respect to the test time and encoding size. This is so even after the insertion of new types (in contrast to DST (Baehni et al. 2007)).

Extensibility. ESE does not require any modification in the existing encoding when new types are inserted.

Our performance analysis on standard Java type hierarchies (e.g. JDK 1.5, java.lang) shows that:

- ESE performs subtyping tests in constant time, which is comparable to the most efficient approach we know of (Zibin et al. 2001).
- ESE has a smaller encoding size than DST (2-3 times), which is the most efficient extensible encoding algorithm we know about. Moreover, after a sequence of insertions, the difference in size between ESE's and DST's encoding becomes even more pronounced. More precisely, the encoding size in DST grows 3 times faster than ESE with the number of insertions at run-time.
- ESE is an extensible algorithm that ensures very fast insertion in centralized settings and minimum bandwidth utilization in distributed settings, compared to non-extensible algorithms like PQE.

Outline. The reminder of this paper is organized as follows. Section 2 presents some basic definitions related to type hierarchies and Section 3 introduces and formally defines the problem of *extensible encoding*. Section 4 describes our encoding algorithm. Section 5 shows how we implement our algorithm efficiently. Section 6 evaluates the performance of our scheme. Section 7 summarizes the related works and finally Section 8 draws some conclusions.

2. Model

This section (1) reviews the basic subtyping model that has been traditionally used in the literature (Sections 2.1, 2.2) as well as (2) introduces some new concepts which we will use in describing our extensible encoding scheme (Section 2.3).

2.1 Type hierarchy

A type hierarchy $H = \langle T, R\rangle$ consists of two components: a set T of types, and a relation R on pairs of types. The relation R is referred to as the *subtyping relation* and is reflexive, transitive and anti-symmetric. We often use the notion $\prec$ as follows: $t_i \prec t_j$ iff $R(t_i, t_j)$; this implies that t_i is subtype of t_j.

2.2 Relations

In a type hierarchy $H = \langle T, R\rangle$:

- **Descendant.** A type t_j is a *descendant* of a type t_i if $R(t_j, t_i)$ (i.e. t_j is a subtype of t_i). $D(t_i)$ denotes the set of all the descendants of t_i in H. It is formally defined as:

$$D(t_i) = \{t_j \in T | t_j \prec t_i\}$$

- **Ancestor.** A type t_j is an *ancestor* of a type t_i if $R(t_i, t_j)$ (i.e. t_j is a super-type of t_i). $A(t_i)$ denotes all the ancestors of t_i in H. It is formally defined as:

$$A(t_i) = \{t_j \in T | t_i \prec t_j\}$$

Note that each type t is parent and ancestor of itself.

- **Child.** A type t_j is a *child* of a type t_i if (1) $R(t_j, t_i)$ (2) $i \neq j$ (3) for any $k \neq i, j$, either $t_k \notin A(t_j)$ or $t_k \notin D(t_i)$.
- **Parent.** A type t_j is a *parent* of a type t_i if t_j has t_i as a child.
- **Root.** A type t_i is a *root* of H if $A(t_i) = t_i$.
- **Leaf.** A type t_i is a *leaf* of H if $D(t_i) = t_j$.

2.3 Slicing

In this section we recall the notion of *straight slicing* (Baehni et al. 2007).

- **Slice.** A *slice* s of a type hierarchy $\langle T, R\rangle$ is an ordered subset of types in T like $[t_i; t_j; \ldots; t_k]$ such that $\{t_i, t_j, \ldots, t_k\} \subseteq T$. For example, [B;A;C] is a slice of the type hierarchy of Figure 1. Notice that, we could change the ordering (e.g. [A;B;C]) and have another slice.
- **Slicing.** A *slicing* S of a type hierarchy $\langle T, R\rangle$ is a partition of types in T in which each set in the partition is ordered. For example, considering the type hierarchy of Figure 1, $\{[B; A; C], [E; F; J; G; k], [H; I; D]\}$ is a slicing. Notice that $\{[A; B; D], [E]\}$ and $\{[A; B; C; E; F; G; H; I], [J; K; I; A]\}$ are not slicing because the first one does not cover all the types and in the second one A occurs two times.
- **Straight Slice.** A *straight slice* s of a type hierarchy $H = \langle T, R\rangle$ is a slice of H such that for any type $t_i \in T$, all the supertypes of t within s are consecutive in s. Formally, a slice $s = [t_1; t_2; ...; t_n]$ is straight if:

$$\forall t \in T, 1 \leq a < b < c \leq n : t_a, t_c \in A(t) \Rightarrow t_b \in A(t)$$

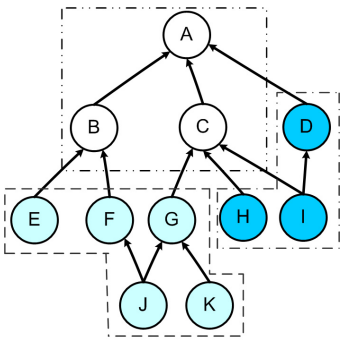

Figure 1. A straight slicing of a type hierarchy

Which is equivalent to:

$$\forall 1 \leq a < b < c \leq n : D(t_a) \cap D(t_c) \subseteq D(t_b)$$

Accordingly, a slicing is *straight* if all its slices are straight. A type hierarchy might have more than one straight slicing. Figure 1 depicts a type hierarchy and one of its straight slicings. Arrows are directed from a child to its parent. In this figure, slice [B;A;C] for example is straight. Indeed, for every type $\in T$, all of its ancestors in this slice are consecutive. For instance the ancestors of C (C, A) and the ancestors of J within this slice (A, B, C) are consecutive. Considering the type hierarchy of Figure 1 a slice like [D;B;A;C] is not straight because the ancestors of I in this slice (A, C, D) are not consecutive.

3. Subtype Encoding

In this section we define the problem we address in this paper: providing an extensible subtype encoding.

- **Encoding algorithm.** Given a type hierarchy $H = \langle T, R \rangle$, an *encoding algorithm* $E = \langle V, f, R^* \rangle$ consists of three components: (1) a value domain V (often referred to as the "encoding"), (2) a function $f : T \rightarrow V$ that maps types to values, and (3) a relation R^* between pairs of values, such that for $v_i = f(t_i)$ and $v_j = f(t_j)$, $R^*(v_i, v_j)$ iff $R(t_i, t_j)$.

Considering the above defined model, *encoding size* is $log|V|$ i.e. the size of V, and the *subtyping test time* corresponds to the time needed to evaluate $R^*(v_i, v_j)$. A trivial encoding algorithm maps each type $t \in T$ to an identifier and the list of its super-types. Calculating $R^*(v_i, v_j)$ consists in searching in the list of super-types of t_i, which costs $O(n)$ where $n = |T|$. We are basically interested in encoding algorithms where (1) the relation R^* can be evaluated efficiently, as in constant-time, and (2) provides a minimal encoding size.

- **Extension.** Roughly speaking, a type hierarchy $H^+ = \langle T^+, R^+ \rangle$ is an *extension* of $H = \langle T, R \rangle$ if H^+ is the outcome of inserting some new types into H. Formally, H^+ is an extension of H if:
  - $T \subseteq T^+$
  - $\forall t_i, t_j \in T : R(t_i, t_j) \Leftrightarrow R^+(t_i, t_j)$
  - $\forall t_i \in T, t_j \in T^+ : R^+(t_i, t_j) \Rightarrow t_j \in T$

  The third condition captures the fact that newly inserted types cannot be super-types of existing ones.

Notice that the notions of encoding algorithm and extension have been informally used in the literature. Using these notions, we introduce and formally define the concept of *extensible encoding algorithm* in the following.

- **Extensible encoding algorithm.** We say that an encoding algorithm is *extensible* if the value mapped to any type t remains unchanged after the insertion of new types. More precisely, given a type hierarchy $H = \langle T, R \rangle$, its encoding algorithm $E = \langle V, f, R^* \rangle$ is extensible if for any extension of H like $H^+ = \langle T^+, R^+ \rangle$, there exists an encoding algorithm $E^+ = \langle V^+, f^+, R^{*+} \rangle$ such that:
  - $\forall t \in T : f(t) = f^+(t)$

We focus on providing an extensible encoding algorithm which provides a test time and encoding size comparable to the most efficient non-extensible algorithms, even after the insertion of new types at run-time.

Describing an encoding algorithm consists of describing how to implement the following data structure:

1. $Query(t_i, t_j)$
   - Input: $t_i, t_j \in T$.
   - Function: check whether t_i is subtype of t_j i.e. evaluate $R(t_i, t_j) = R^*(f(t_i), f(t_j))$.
2. $Insert(t_i, T_1)$
   - Input: $t_i \in T, T_1 \subseteq T$.
   - Function: insert t_i in H as child of all types like t_j such that $t_j \in T_1$ i.e. calculate E^+ when new type t and its corresponding relations are inserted.

The first operation is well-known as the subtyping test, whereas the second captures the ability to include new types at run-time.

4. Extensible Subtyping Encoding: ESE

In the following, first we introduce our ESE encoding scheme (which defines the value domain V) in Section 4.1. Section 4.2 shows how to perform the subtyping test (i.e. how to calculate R^*) using this scheme. Then in Section 4.3 we explain how to construct the initial encoding (which completes the definition of function f). The insertion algorithm is given in Section 4.4.

The basic idea is the following. We first provide a straight slicing of the type hierarchy. Then we assign an identifier to each type corresponding to its position in a slice, and to its position with respect to its ancestors in each slice. Testing if t is subtype of r consists of comparing the identifier of r with identifiers of ancestors of t (which define an interval). We insert new types in such a way that we maintain the straightness of the slicing whenever possible. It is worth mentioning that, in cases where the ancestors of the new type t are not consecutive in a slice s_i, by definition, s_i is not straight with respect to t. Since our insertion algorithm does not change the encoding of the old types, we might end up in these cases with slices that are not straight. We evaluate the effect of such cases on the average test time in the Section 6.

4.1 Encoding

We explain here how we use the notion of straight slicing to encode a type hierarchy. Specifically, we define the value domain V (the "encoding") and how to encode H, given a slicing S.

Considering a type hierarchy $H = \langle T, R \rangle$, and a slicing S of H, we encode each type $t \in T$ with the following parameters:

1. $SID(t)$. The identifier of the slice to which t belongs.
2. $ID(t)$. An integer indicating the position of t within its slice.
3. $I_i(t)$. For each slice $s_i \in S$, $I_i(t)$ represents the set of intervals of ancestors of t in s_i. To be more accurate, we specify an interval I with two integers $I.beginning$ and $I.end$ for the beginning and the end of I respectively.

For example in Figure 1, according to the given straight slicing ($[B; A; C][E; F; J; G; K][H; I; D]$), the type J is encoded by $SID(J) = 1$, $ID(J) = 2$, $I_0(J) = [0, 2]$, $I_1(J) = [1, 3]$ and $I_2(J) = [-]$, and the type C is encoded by $SID(C) = 0$, $ID(C) = 2$, $I_0(C) = [1, 2], I_1(C) = [-]$ and $I_2(C) = [-]$. Since the given slicing is straight, the ancestors of each type (like J) are consecutive in each slice (like s_0) and so they are represented by only one interval like ($[0, 2]$). In general, as long as the slicing is straight, $I_i(t)$ (i.e. set of intervals of ancestors of t in slice s_i) includes only one interval. In the case of single inheritance type hierarchies, we insert new types such that the slicing always remains straight. This is not the case for multiple inheritance, as the

insertion algorithm in some special cases violates the straightness of slices, resulting in certain types having more than one interval as part of their encoding.

4.2 Subtyping test

In order to test whether a given type *t* is a subtype of another type *r*, we simply check the ancestors of t in r's slice, which are defined by one or more intervals. That is, if $SID(r) = i$, the subtyping test consists in checking whether or not $ID(r)$ is within one of the intervals $I_i(t)$.

Figure 2 shows how we implement the subtyping test more precisely.

```
1: function QUERY(t,r)                          {Check if t is subtype of r}
2:    i ← SID(r)
3:    for all I ∈ I_i(t) do
4:       if I.beginning ≤ ID(r) ≤ I.end then
5:          return true
6:       end if
7:    end for
8:    return false
9: end
```

Figure 2. Subtyping test.

For example in the type hierarchy given by Figure 1, it is observable that J is subtype of C because $SID(C) = 0$ and $I_0(J).beginning = 0 \leq ID(C) = 2 \leq I_0(J).end = 2$.

4.3 Preprocessing (encoding initialization)

We now describe how to preprocess the type hierarchy in order to construct a straight slicing that results in an efficient encoding.

Consider the trivial straight slicing in which each slice contains exactly one type. Notice that this slicing effectively corresponds to the trivial encoding algorithm, and results in $n = |T|$ slices. Our goal is to devise a straight slicing with a minimal number of slices.

4.3.1 Constructing the straight slicing

Our encoding initialization algorithm goes as follows (Figure 3):

1. Sort all the types by their number of descendants, such that the type with the maximum number of descendants is the head of the list.
2. Put the head of the list in the first slice.
3. Iterate over the list, and insert each type within the first position such that it maintains a straight slice. If such a slice does not exist, simply create a new one and put that type within the new slice. (An efficient method to check whether or not inserting a type in a slice violates the straightness of that slice is given in Section 5.)

If we consider the type hierarchy of Figure 1, for instance, after ordering types by their number of descendants we end up with the ordered list $L = \{A, C, B, G, D, F, E, H, I, J, K\}$. Initially, we put A in a slice leading to the slicing $S = \{[A]\}$. Putting C right before A does not violate the straightness of its slice and thus $S = \{[C; A]\}$. B and A are ancestors of B and C is not ancestor of B, therefore $[B; C; A]$ is not straight; for the same reason $[C; B; A]$ is not straight; so we put B right after A yielding $S = \{[C; A; B]\}$. G is subtype of C and A, and can be inserted right before C. The next type in the list is D and putting D anywhere in this slice violates its straightness. So according to the algorithm we create a new slice and put D within that ($S = \{[G; C; A; B][D]\}$). Continuing this process we will end up with a straight slicing $S = \{[K; G; C; A; B; F][J; H; E; D; I]\}$ which has only two slices.

```
1: procedure STRAIGHTSLICING(H)     {Generates a straight slicing of H}
2:    L ← sort(H)                   {Sort types by number of descendants}
3:    Put(L[0], Slices[0], 0)       {Put the 1st type within the 1st slice}
4:    numberOfSlices ← 1
5:    for i ← 1 to sizeOf(L) do
6:       for j ← 0 to numberOfSlices − 1 do
7:          for k ← 0 to sizeOf(Slices[j]) do
8:             if STRAIGHT(Slices[j], L[i], k) then
9:                Put(L[i], Slices[j], k)
10:               Goto 5                {Goto the next type in the list}
11:            end if
12:         end for
13:      end for
14:      Put(L[i], Slices[numberOfSlices], 0)      {Put in new slice}
15:      numberOfSlices ← numberOfSlices + 1
16:   end for
17: end
```

Figure 3. Initialization (creating a straight slicing).

Precisely because the algorithm ensures the straightness of all the slices before adding a new type, one can immediately conclude that the generated slicing is always straight. On the other hand, the order in which types are inserted into the slicing is the key point that limits the number of generated slices in this algorithm. Roughly speaking, the only constraint that might force our algorithm to create a new slice is a subtyping relation between two types. For instance, considering the example given in Figure 1, the reason why the algorithm has to create a new slice for D is that B, C, and D are all subtypes of A. Thus, after serving the types which have lots of descendants, it would be less likely for the algorithm to need new slices.

4.3.2 Identifier reservation

ESE reserves some places (identifiers) in each slice for run-time insertions. This list of reserved identifiers (called *reservedIDs*) is initialized after the straight slicing construction and will be used by our insertion algorithm. Section 5 illustrates how we provide such a list. Roughly speaking, the idea is to reserve some places in each slice, for which the insertion of new types does not violate the straightness of slicing. According to our reservation algorithm (given in Section 5), each reserved place is surrounded by two existing types (i.e. reserved places are not consecutive), the number of reserved places cannot be more than the number of existing types.

4.3.3 Identifier assignment

Using the straight slicing constructed with one of the algorithms proposed in Sections 4.3.1 and 4.3.2, we assign to each type t two identifiers ($ID(t)$, $SID(t)$), as well as its interval of ancestors for each slice s ($I_s(t)$). Each slice s is assigned a unique integer i as the identifier. If type t is placed within s_i, $SID(t) = i$. $ID(t)$ specifies the position of t within s_i. The identifier of the types of a slice might not be consecutive as ESE reserves some identifiers in each slice for run-time insertions. The identifiers of ancestors of t in each slice s_i specify the $I_i(t)$.

4.4 Insertion

In this section we illustrate how to insert a new type in the hierarchy at run-time.

Considering the fact that a new type t added at run-time cannot be ancestor of any existing types, inserting a new type in a slice between two ancestors of an existing type would violate the straightness of the slicing. Thus we either (1) use the reserved list for new types (reservedIDs) (2) append it at the head or tail of a slice, or (3) create a new slice. On the other hand, in order to place ancestors of

t within a minimal number of intervals, it would be preferable to put t adjacent to one of its parents. Our insertion algorithm goes as following:

1. If there exists a reserved place adjacent to one of the parents of t, insert t there, and remove that place from the list of reserved places. (Notice that each slice has its own list of reserved places.)
2. Otherwise, if a parent of t is at the head (tail) of a slice s_i, put t within s_i right before (after) its parent.
3. Otherwise, create a new slice and put t within it.
4. Set the parameters of t accordingly (e.g. $ID(t)$, $SID(t)$, etc.).

Figure 4 illustrates the insertion of a new type N as subtype of H and I into the type hierarchy of Figure 1. Since H (one of the parents of N) is the head of its slice according to the insertion algorithm we put N right before H. It is easily observable that not only the encoding of all the old types remains unchanged but also, in this case, the slicing is still straight with respect to the newly added type (i.e. all the ancestors of the new type are consecutive in all the slices).

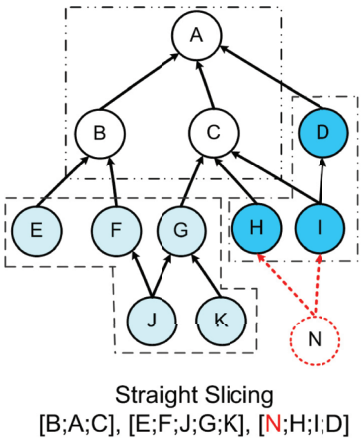

Figure 4. Inserting a new type N in a type hierarchy

Figure 5 depicts the protocol for inserting a new type in a given type hierarchy.

Notice that, in specific cases, after the insertion of a new type, the resulting slicing might be non-straight. This is especially true when the newly added type is subtype of two types of a slice s_i whose ancestors are not consecutive in s_i. For example, in Figure 6, inserting N as subtype of E, K violates the straightness of the slice [N;E;F;J;G;K], because ancestors of N in this slice (N, E, G, K) are not consecutive.

5. Detecting a Straight Slicing

This section illustrates (1) how we test if the straightness of a slice is maintained after the insertion of a type and (2) how we provide the list of reserved identifiers for run-time insertions. The first one is needed to ensure the straightness of slicing during the initialization process at compile-time (Figure 3) and the second one ensures the same property upon insertion of new types at run-time (Figure 5).

5.1 Straightness maintenance

As shown in Section 4.3.1, our initialization algorithm iteratively picks a type t and puts it in an existing or a new slice as we

```
1: procedure INSERT(t,T′) {Insert t in a given type hierarchy as child of all
   the types in T′}
2:   for all s_i ∈ S do                    {S is the slicing of the type hierarchy}
3:     for all r ∈ T′ do
4:       I_i(t) ← I_i(t) ∪ I_i(r)
5:     end for
6:   end for
7:   createNewSlice ← true
8:   for all r ∈ T′ do
9:     if isHead(r) or reservedIDs(SID(r), ID(r) − 1) then
10:      createNewSlice ← false
11:      SID(t) ← SID(r)
12:      ID(t) ← ID(r) − 1
13:      i ← SID(t)
14:      I_i(t) ← I_i(t) ∪ [ID(t), ID(r)]
15:      break
16:    end if
17:    if isTail(r) or reservedIDs(SID(r), ID(r) + 1) then
18:      createNewSlice ← false
19:      SID(t) ← SID(r)
20:      ID(t) ← ID(r) + 1
21:      i ← SID(t)
22:      I_i(t) ← I_i(t) ∪ [ID(r), ID(t)]
23:      break
24:    end if
25:  end for
26:  if createNewSlice then
27:    SID(t) ← newSliceID
28:    ID(t) ← 0
29:    i ← SID(t)
30:    I_i(t) ← [0, 1]
31:  end if
32:  if ID(t) ∈ reservedID(SID(t)) then
33:    reservedID(SID(t)) ← reservedID(SID(t)) − ID(t)
34:  end if
35: end
```

Figure 5. Inserting a new type in a type hierarchy.

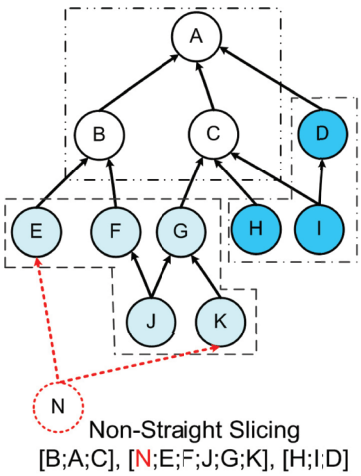

Figure 6. Inserting a new type N in a type hierarchy

discuss below. It puts t in an existing straight slice s if this insertion maintains the straightness of s. This section presents an efficient algorithm to check this constraint. More precisely, for a type t and a straight slice s, our algorithm finds the first place in s such that insertion of t at that place maintains the straightness of s. This algorithm performs the following operation:

- $Search(r, s)$
  - Input: a type $r \in T$, a straight slice $s = [t_1; t_2; ...; t_n]$.
  - Output: the first position i such that $s\prime = [t_1; t_2; ...; t_i; r; t_{i+1}; ...; t_n]$ is straight, if any.

Assuming that $s = [t_1; t_2; ...; t_n]$ is a straight slice by definition we have:

$\forall a, b, c \in \{1..n\}\ , a < c < b : D(t_a) \cap D(t_b) \subseteq D(t_c)$

In order to check if $s\prime = [t_1; t_2; ...; t_i; r; t_{i+1}; ...; t_n]$ is straight we have to ensure three conditions:

(1) $\forall a, b \in \{1..n\}\ , i < a < b : D(r) \cap D(t_b) \subseteq D(t_a)$

(2) $\forall a, b \in \{1..n\}\ , b < a \leq i : D(r) \cap D(t_b) \subseteq D(t_a)$

(3) $\forall a, b \in \{1..n\}\ , a \leq i < b : D(t_a) \cap D(t_b) \subseteq D(r)$

A trivial algorithm which checks all the above conditions takes $O(n^2)$ subset tests (i.e. considering two sets A, B to test if $A \subseteq B$), for each position i, leading to an overall $O(n^3)$ number of tests to search the entire slice s. In the following we present a method to perform the search in $O(n^2)$. In fact, the idea is to remove the redundant tests.

Observe first that, independently of the position i, the following set of conditions includes those in (1) and (2).

(4) $\forall a, b \in \{1..n\} : D(r) \cap D(t_b) \subseteq D(t_a)$

Clearly, (1) and (2) can be checked in $O(n^2)$ number of tests for all the positions i (i.e $\forall i \in \{1..n\}$).

On the other hand it is easy to see that condition (3) is reducible to:

(5) $D(t_i) \cap D(t_{i+1}) \subseteq D(r)$

This is induced by the straightness of s since:

$\forall a, b \in \{1..n\}\ , a \leq i < b : D(t_a) \cap D(t_b) \subseteq D(t_i)$, and

$\forall a, b \in \{1..n\}\ , a < i + 1 \leq b : D(t_a) \cap D(t_b) \subseteq D(t_{i+1})$

which yields:

$\forall a, b \in \{1..n\}\ , a \leq i < b : D(t_a) \cap D(t_b) \subseteq D(t_i) \cap D(t_{i+1})$

And thus knowing (5) we have:

$\forall a, b \in \{1..n\}\ , a \leq i < b : D(t_a) \cap D(t_b) \subseteq D(r)$ (i.e. (3))

Checking condition (5) rather than (3) takes $O(1)$ tests.

Using the above mentioned facts, our algorithm performs search(r, $s = [t_1; ...; t_n]$) as follows:

1. Perform all the tests in (4), and generate a two dimensional array $A[n][n]$ such that $A[a, b] = 1$ iff $D(r) \cap D(t_b) \subseteq D(t_a)$.
2. List all the positions like i satisfying (1) and (2). Notice that the answer to all the tests in (1) and (2) are available in A.
3. Return the first i in the list that satisfies (5).

5.2 Reserved identifiers

After constructing the straight slicing we put a place P of a slice $s = [t_1; t_2; ...; t_n]$ in the reserved list if:

- $D(t_{P-1}) \cap D(t_P) = \emptyset$

If P is a reserved place, then $ID(t_P) = ID(t_{P-1} + 2)$. As illustrated in Section 4.4, we insert a new type r in P if r has t_{P-1} or t_P as a parent. Considering the facts that r is an ancestor of itself and we insert r beside one of its parents in s, if the ancestors of r are consecutive in s before inserting r, these ancestors remain consecutive after this insertion (i.e. s remains straight with respect to r). In the following we show that this is also the case for existing types. More precisely, we show that if the ancestors of an existing type u are consecutive in s, the insertion of r in P does not separate them (i.e. s remains straight with respect to the existing types).

Suppose by contradiction that the insertion of r in P separates some consecutive ancestors of u. Since r is placed between t_{p-1} and t_p, u has both t_{p-1} and t_p as its ancestors which contradicts with our assumption that $D(t_{P-1}) \cap D(t_P) = \emptyset$.

6. Performance Analysis

In this section we analyze the performance of our algorithm in static and dynamic settings. In static settings, we consider the traditional metrics of subtyping test time and encoding size. In dynamic settings, we also consider the cost of insertions. We examine the performance of our algorithm in the context of 13 commonly used type-hierarchies, and provide experimental data to support our efficiency claims.

We compare our algorithm with two commonly used schemes: PQE, the fastest (to our knowledge) in static hierarchies and DST, one of the only schemes designed for dynamic type hierarchies.

The conclusions we draw are as follows:

- All three protocols, PQE, DST, and ESE have roughly the same subtyping test time(see Table 2).
- Among extensible schemes, ESE has the smallest encoding size (see Table 3). Moreover, after a sequence of insertions, the difference is size between ESE's and DST's encoding become even more pronounced (see Figure 9). (By contrast, non-extensible PQE has a smaller encoding size than ESE.)
- The running time for an insertion for ESE is only slightly slower than for DST, and much faster than for (non-extensible) PQE.

We thus conclude that ESE provides a better trade-off between the encoding size and insertion time, while maintaining a near-optimal subtype test time.

All experiments we refer to were obtained from Java implementations of ESE, PQE and DST[1] using an Intel Pentium 4 2.4GHz with 1GB RAM on a Fedora 2.6 machine. We consider the type hierarchies of standard Java packages including java.io, java.lang, Java EE 5, JDK 1.1.8, JDK 1.2.2, JDK 1.3.1, JDK 1.4.2, and JDK 1.5.0. All the presented values are averaged over 10000 measurements. We generate subtyping tests randomly (in a uniform manner over the set of types). The types to be inserted are generated randomly but with respect to the statistical analysis made by Zibin et al. (2001), as recalled in Table 1. This table shows some topological properties of 13 type hierarchies, including the number of types, the average number of parents and the average number of ancestors for each hierarchy.

6.1 Subtyping test time

Worst case. In a static setting, each test takes $O(1)$ steps. Since all the slices are straight, all the ancestors of a type t are consecutive. Then a subtyping test consists in checking one interval. In a dynamic setting, checking whether a type t is subtype of a type r

[1] For this algorithm we use its open source code at: `http://lpd.epfl.ch/baehni/dst.tgz`

Hierarchy	n	$\|P\|/n$	$\|A\|/n$
IDL	66	0.98	3.83
JDK 1.1	225	1.04	3.17
Laure	295	1.07	8.13
ED	434	1.66	7.99
LOV	436	1.71	8.50
Unidraw	613	0.78	3.02
Cecil	932	1.21	6.47
Geode	1,318	1.89	13.99
JDK 1.18	1,704	1.10	4.35
Self	1,801	1.02	29.89
Eiffel4	1,999	1.28	8.78
JDK 1.22	4,339	1.19	4.37
JDK 1.30	5,438	1.17	4.37

Table 1. Topological properties of some hierarchies. n: number of types, $|P|/n$: average number of parents, $|A|/n$: average number of ancestors (Zibin et al. 2001).

takes $O(|parents(t)|)$ steps, since our insertion algorithm is such that, in some cases, the ancestors of a newly inserted type t are separated into $|parents(t)|$ intervals.

Experiments. The results given in Table 2 compare the test time of ESE with PQE and DST (1) in a static setting and (2) after the insertion of some randomly chosen new types (we add to each hierarchy 10% of its size). In static settings, ESE performs a subtyping test slower than PQE but faster than DST. For the dynamic case, as Table 2 conveys, ESE like PQE and DST maintains its efficiency after the addition of new types.

Type hierarchy	Static			Dynamic		
	ESE	PQE	DST	ESE	PQE	DST
java.lang	12	10	13	12	10	13
java.io	12	10	13	12	10	13
java EE 5	12	10	13	13	10	13
JDK 1.1.8	13	11	13	13	11	14
JDK 1.2.2	13	11	14	14	11	14
JDK 1.3.1	13	11	13	13	11	14
JDK 1.4.2	13	11	14	14	11	14
JDK 1.5.2	13	11	14	14	12	14

Table 2. Subtyping test time in micro second.

According to Table 1, and the analysis of Zibin et al. (2001), the average number of parents is no more than 1.89; also the number of types inserted at run-time is usually small compared to the entire hierarchy, and so the situation in which the resulting slicing is not straight happens rarely. For example, after the insertion of 500 new types into the type hierarchy of JDK 1.5, our experiments show that, on the average, the number of intervals that have to be checked is less than 1.052. More precisely, in order to perform the test $Query(t, r)$ (i.e if t is subtype of r) we check only one interval if t has existed from compile-time. If t is added at run-time, the number of intervals that have to be checked depends on the number of parents it has and their position in the type hierarchy. Figure 7 shows how the query time is influenced by the average number of parents. In this experiment we consider queries like $Query(t, r)$ in which t is inserted at run-time, and we measure the average test time in JDK 1.5, after the addition of new types with different average numbers of parents: 2, 3, and 14 (the maximum number of ancestors according to Table 1).

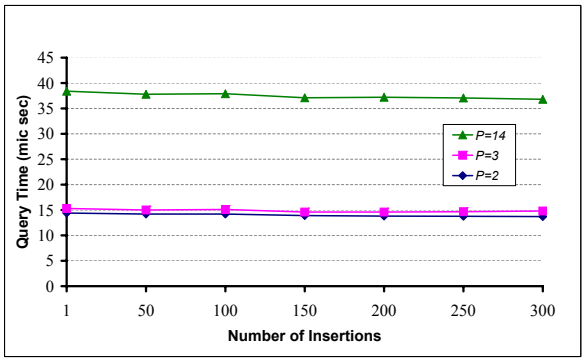

Figure 7. Test time after insertions

6.2 Encoding size

Worst case. If a slicing has k slices, then the encoding size of ESE is $O(k)$. We assign to each type two identifiers (ID, SID) and the interval of its ancestors for each slice (see Section 2.1) which leads to the encoding length of $2k + 2$ for each type. (The encoding size of DST and PQE is in the order of $2k + 2$ and $k + 2$ respectively.)

Experiments. Table 3 compares ESE with DST and PQE in terms of encoding size which is influenced mainly by the number of slices. Here, we consider the algorithms excluding their optimizations related to the compression of integer indices (ids) into a minimal bit representation. (Such optimizations are common to all three schemes.) According to this table, the amount of space that ESE uses is between 2 to 3 times less than DST, but it is larger than PQE.

Type hierarchy	Encoding Size			of Slices		
	ESE	DST	PQE	ESE	DST	PQE
java.lang	8	26	-	3	12	-
java.io	8	22	-	3	10	-
Java EE 5	10	32	-	4	15	-
JDK 1.1.8	16	36	8	8	17	6
JDK 1.2.2	18	38	10	8	18	8
JDK 1.3.1	18	36	10	8	17	8
JDK 1.4.2	18	38	-	8	18	-
JDK 1.5.0	18	38	-	8	18	-

Table 3. Encoding size per type (number of integers).

For the case of single-inheritance type hierarchies, our experimental results show that ESE gives a near-optimal number of slices. Figure 8 gives an example showing that the approximation-rate[2] of DST is not less than $n/6$ in terms of encoding size in comparison to ESE. In this example the optimal slicing has 2 slices ([A;D;G;J][B;C;E;F;H;I]), while DST gives 4 ([B;A;C][E;D;F][H;G;I][J]). If we continue the pattern of Figure 8 (a) (like Figure 8 (b)), for each additional 3 types, DST creates a new slice. Since, in such hierarchies the optimal slicing has always 2 slices, the approximation-rate of DST is $(n/3)/2 = n/6$.

Figure 9 evaluates ESE against DST in terms of average encoding length, in a dynamic setting, after the addition of new types at

[2] The approximation-rate is defined as *result of the algorithm / optimal result*

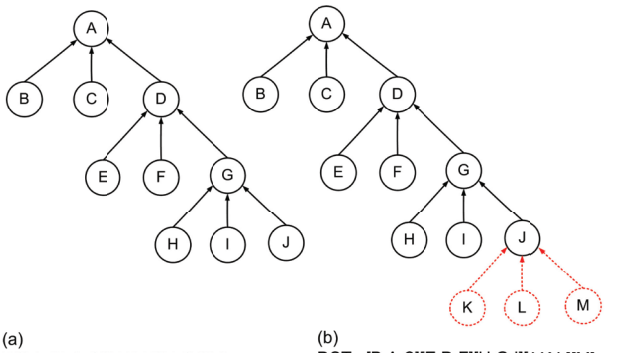

Figure 8. Approximation rate of DST's encoding size

run-time. In this experiment, we only measure the encoding size of the new types, because the encoding size of the old types remains unchanged. We consider java.lang and JDK 1.5 as samples of relatively small and large type hierarchies. We generate the new types randomly with the average number of parents of 2 which is more than the average number of parents of all the hierarchies shown in Table 2. As can be observed from Figure 9, compared to DST, our algorithm is much more efficient under insertion of new types. Inserting 20 types, the average encoding size in ESE grows 3-30 times less than PQE. This fact could be explained by the method we use to reserve identifiers for newly inserted types, given in Section 4.3.3. Apparently, this method is effective until the reserved identifiers are exhausted. Figure 10 illustrates the effectiveness of our method after the addition of several types (up to 300) in JDK 1.5 type hierarchy. Let us observe that the growth of the encoding length with the number of insertions becomes slower when we increase the average number of parents. This may be explained by our insertion algorithm which tries to insert new types beside one of their parents. (Otherwise, it creates a new slice.) The worst cases happen when all the parents of the new type are among newly added types. According to our insertion algorithm, no reserved identifier can be used for such cases. Figure 11 depicts the fast growth of encoding length with the number of insertions in one of these cases.

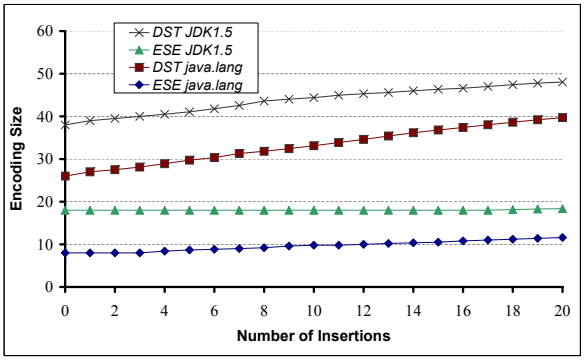

Figure 9. Growth of encoding size with insertions.

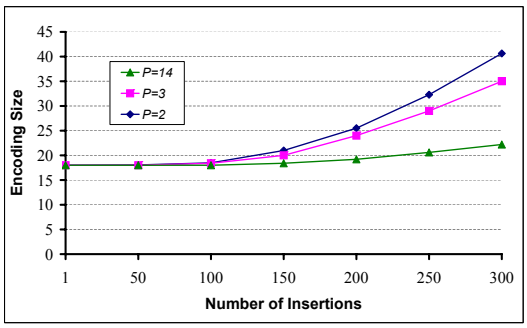

Figure 10. Encoding length after addition of new types

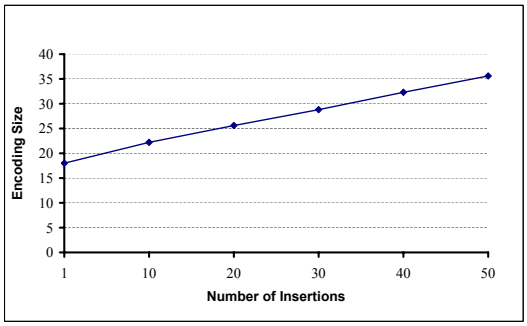

Figure 11. Growth of encoding length in the worst case

6.3 Insertion time

As noticed before, the insertion method proposed in Section 4.4 does not impact the encoding of the old types. In centralized settings, this fact normally speeds up the subtyping process, whereas in distributed systems it avoids the propagation of encoding changes throughout the network. With respect to this property, we compare our algorithm with PQE and DST which are the most efficient (non-extensible and extensible respectively) approaches we know of.

Worst case. While ESE and DST takes $O(|parents(t)|)$ to add a new type t, PQE inserts new types in $O(|A|)$ in which $|A|$ is the number of ancestors in the type hierarchy.

From Table 2, the average number of parents is always less than 2. On the other hand, the number of ancestors in a type hierarchy can be 29.89 times larger than the number of types. These facts illustrate the significant difference between the insertion time in ESE and PQE.

To better appreciate the advantage of extensible algorithms in dynamic distributed settings, consider a system with two components C_1, C_2, and a type t a priori known by both components. Upon receiving an object O, C_2 which was expecting an object of type t, has to check whether or not the object it has received is of that type. To do that, it is necessary for C_1 to send the identifier of type t along with O (Microsoft 2005; Sun Microsystems 2005; OMG

2001; Dedecker et al. 2006). On the other hand, upon the addition of a new type r at C_1, PQE changes the encoding of the old types including t, and hence the new identifier of t at C_1 is not valid for C_2 anymore. Apparently, any algorithm aimed to fix this problem has to perform a global agreement on the encoding between all the components of the system for each newly added type, which is extremely costly in terms of time and bandwidth utilization. We recall that, in extensible schemes (ESE and DST), after the addition of r at C_1, the encoding of the types at C_2 are still valid and thus, there is no need for further information exchanges.

Experiments. Table 4 compares the insertion time of ESE with DST and PQE. While ESE and PQE usually provides comparable speed, PQE as the most efficient non-extensible solution takes much more time.

Hierarchy	ESE	DST	PQE
java.lang @ JDK 1.5.0	16	12	37
java.io @ JDK 1.5.0	16	12	36
Java EE 5	16	11	385
JDK 1.1.8	17	13	757
JDK 1.2.2	17	15	4263
JDK 1.3.1	19	15	5381
JDK 1.4.2	21	15	6995
JDK 1.5.0	21	17	8905

Table 4. Insertion time in micro second.

7. Related Work

Given that the subtyping test has a significant impact on the overall performance of systems, several type encoding schemes have been proposed in the literature (Zibin et al. 2001, 2002; Cohen 1991; Vitek et al. 1997; Sprugnoli 1977; Krall et al. 1997; Dietz 1982, 1987; Denielou et al. 2006; Palacz et al. 2003). However, almost all previous works were basically designed for static type systems and do not efficiently provide support for newly inserted types. We overview in the following these algorithms as well as the dynamic algorithms we know of (Baehni et al. 2007; Vitek et al. 1997).

(C)PQE: (Coalesced) PQ-encoding (Zibin et al. 2001). Based on *PQ-trees* (Gosling et al. 2005), a technique for searching an ordering satisfying prescribed constraints, PQ-encoding (*PQE*) splits the type hierarchy into a minimum number of groups of types called *slices*, such that each slice satisfies *local consecutiveness*. A slice i is locally consecutive if there is an ordering of i in which for each type t, all the descendants of t are consecutive in i. PQE encodes a type t with (1) an integer s_t which is the number of slice to which t belongs (2) a pseudoarray id_t, such that $id_t@i$ is the id of type t with respect to slice i (3) for each slice i, the interval of descendants of t in i. In order to check if a type t is a subtype of another type r, PQE checks if the identifier of t is part of the interval of the subtypes of r in the slice containing t. While PQE improves the encoding length of all previous results, it can not handle newly added types at runtime without reconstructing the entire encoding. CPQE is an optimization of PQE which reduces the space consumption of PQE even further. CPQE does not simply support the addition of new types at run-time.

TS: Type Slicing (Zibin et al. 2002). TS uses the idea of PQE-encoding without requiring global reconfiguration for each newly added type. It maintains an ordered list for all types in a slice and substitutes integers and arithmetical comparisons by list positions and list order maintenance. Upon addition of a new type, TS inserts this type into the first ordered list in which this insertion does not violate the local consecutiveness. If such a slice does not exist, it creates a new slice. It is worth mentioning that, when a new type is added, the encoding of the parents of the new type must be modified. On the other hand, TS is around 10 times less efficient than CPQE in terms of encoding size.

DST: Distributed Subtyping Test (Baehni et al. 2007). Basically, DST is designed for distributed environments while retaining the efficiency of a centralized scheme in terms of encoding size and test time. The idea of DST and ESE are similar in the sense that they use the same slicing technique, which can be viewed as a modification of PQE's slicing scheme with a fundamental difference: ancestors are ordered instead of descendants. A major difference between DST and ESE is in the construction of the initial slicing and the algorithm they use for inserting new types at run-time.

Range compression encoding (Agrawal et al. 1989). Like PQE, range compression encoding splits the type hierarchy into subsets of types which are in the form of trees. The algorithm ensures that all the subtypes of a type t are in one interval in a tree (the types are ordered using a post-order traversal algorithm). A type t is identified by its identifier which is its position in the tree it belongs to, as well as the set of its subtypes intervals. Upon addition of a new type, the old encoding is useless and the algorithm has to reconstruct the trees.

Bit vector encoding (Krall et al. 1997). This algorithm encodes a type t with a vector of k bits called vec_t such that, if a type t is a subtype of a type r, then $vec_t \wedge vec_r = vec_r$. Obviously, whenever a new type is added at run-time, the entire type hierarchy encoding has to be re-computed.

(B)PE: (Bit) Packed Encoding (Vitek et al. 1997). With packet encoding, the type hierarchy is divided into subset of types called *bucket*, such that two super-types of a type can not be in the same bucket. The encoding of a type t consists of the identifier of the bucket in which t is contained, the position of t in its bucket, and a set of pairs $\langle bucket, super - type \rangle$ indicating the super-type of t in each bucket. Bit packet encoding is an optimization which permits two or more buckets to be encoded in a single byte. According to both algorithms, the number of buckets and hence the average encoding length grows with the number of ancestors in the type hierarchy. Both algorithms avoid global reconfiguration upon addition of new types at runtime.

Perfect Hashing (Fredman et al. 1984; Sprugnoli 1977). This algorithm simply hashes each ancestor of a type with a unique hash key (Ducournau 2006). The subtyping test then is just a search in a hash table. While perfect hashing does not need global reconfiguration for each newly added type at runtime, the size of the hash table and hence the encoding length grows with the number of ancestors in the type hierarchy.

Two-Hop Cover (Schenkel et al. 2005). For each pair of nodes t, r, if there is a path from t to r, it chooses a node w on a path from t to r and adds w to the set of ancestors of r ($L_{in}(r)$) and set of descendants of t ($L_{out}(t)$). It tests if r is reachable from t by checking whether $L_{in}(r) \cap L_{out}(t) = \emptyset$. The connection from t to r is given by (1) the hop from t to w and (2) the hop from w to r, hence the name of the method. Upon insertion of a new type u, this method adds u to $L_{out}(a)$ for all ancestors a of u (i.e. It changes the encoding of the old types, leading to a non-extensible encoding algorithm).

Dietz's Algorithm (Dietz 1982, 1987). The main idea of Deitz's algorithm is to maintain the pre- and post-order of the tree in an ordered list. Type t_i is subtype of t_j iff t_i occurs before t_j in the post order and t_j occurs before t_i in the pre-order. It provides constant

time for insertion and query operations and linear encoding size. The problem of having a dynamic Deitz's algorithm turns into the problem of *order maintenance* (Gosling et al. 2005). This restricts the scope of the algorithm to single inheritance only.

Cohen's Algorithm (Cohen 1991). This algorithm is also restricted to single inheritance hierarchies. It defines for each type t a level denoted by l_t which is its number of ancestors (i.e. its distance to the root of the tree). Then it associates with t a unique identifier and an array A of length l_t which stores the identifier of ancestors of t like $t\prime$ in $L[l_{t\prime}]$. While the algorithm is fully incremental, in the worst case the encoding length is $O(n^2)$ when the type hierarchy is a chain.

8. Conclusion and Future Research

This paper introduces ESE, an extensible subtyping test algorithm that ensures (1) efficient insertion of new types, (2) efficient subtyping tests, and (3) small space usage. More precisely, ESE provides subtyping test time and encoding size that are comparable to the most efficient static subtyping algorithms we know of (PQE). ESE, on the other hand, uses less memory space (2-3 times) than the most efficient dynamic subtyping algorithms published before (DST). In addition, unlike DST, ESE remains efficient even after new types are inserted at run-time.

References

R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient Management of Transitive Relationships in Large Data and Knowledge Bases. In *Proceedings of the ACM International Conference on Management of Data*, pages 253–262, 1989.

S. Ajmani, B. Liskov, and L. Shrira. Modular Software Upgrades for Distributed Systems. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 452–476, 2006.

S. Baehni, J. Bareto, P. Eugster, and R. Guerraoui. Efficient Distributed Subtyping Tests. In *Proceedings of the ACM International Conference on Distributed Event-based Systems*, pages 214 – 225, 2007.

N. H. Cohen. Type-Extension Tests can be Performed in Constant Time. *ACM Transactions on Programming Languages and Systems*, 13:626–629, 1991.

J. Dedecker, T. V. Cutsem, S. Mostinckx, T. D'Hondt, and W. D. Meuter. Ambient-Oriented Programming in AmbientTalk. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 230–254, 2006.

P. F. Dietz. Maintaining order in a linked list. In *Proceedings of the 14th Ann. ACM Symposium on Theory of Computing*, pages 122–127, 1982.

P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Ann. ACM Symposium on Theory of Computing*, pages 365–372, 1987.

P.-M. Denielou and J. Leifer. Abstraction Preservation and Subtyping in Distributed Languages. In *Proceedings of the 11th ACM International Conference on Functional Programming (ICFP '06)*, pages 286–297, 2006.

R. Ducournau. Le hachage parfait fait-il un parfait test de sous-typage? In *Proceedings of the Conférence des Languages et Modèles à Objets*, pages 71–86, 2006.

M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a Sparse Table with $O(1)$ Worst Case Access Time. *Journal of the ACM*, 31(3):58–544, 1984.

A. Goldberg and A. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java 1.5 Language Specification*, third edition. `http://java.sun.com/j2se/1.5.0/docs/index.html`, 2005.

A. Hejlsberg and S. Wiltamuth. *C# Language Specification*. Microsoft Press, 2001.

A. Krall, J. Vitek, and R. N. Horspool. Near Optimal Hierarchical Encoding of Types. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 128–145, 1997.

N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1996.

Microsoft. *Microsoft Message Queuing*, 2005.

Microsoft. *.NET Framework Reference Documentation*. `http://www.microsoft.com/net/`, 2005.

Sun Microsystems Inc. *Java Message Service - Specification, version 1.1*. `http://java.sun.com/products/jms/docs.html`, 2005.

OMG. *The Common Object Request Broker: Architecture and Specification*, 2001.

K. Palacz and J. Vitek. Java Subtype Tests in Real-Time. In *Proceedings of the 17th European Conference on Object-Oriented Programming*, pages 378–404, 2003.

L. K. Schubert, M. A. Papalaskaris, and J. Taugher. Determining Type, Part, Colour, and Time Relationships. *IEEE Computer*, 16:53–60, 1983.

R. Sprugnoli. Perfect Hashing Functions: A Single Probe Retrieving Method for Static Sets. *Communications of the ACM*, 20(11):841–850, 1977.

B. Stroustrup. The C++ Programming Language. Addison-Wesley, 2004.

J. Vitek, R. N. Horspool, and A. Krall. Efficient Type Inclusion Tests. In *Proceedings of the 12th ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 142–157, 1997.

Y. Zibin and J. Gil. Efficient Subtyping Tests with PQ-Encoding. In *Proceedings of the 16th ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 819–856, 2001.

Y. Zibin and J. Gil. Fast Algorithm for Creating Space Efficient Dispatching Tables with Application to Multi-Dispatching. In *Proceedings of the 17th ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 142–160, 2002.

R. Schenkel, A. Theobald, and G. Weikum. Efficient Creation and Incremental Maintenance of the HOPI Index for Complex XML Document Collections. In *Proceedings of the 21st International Conference on Data Engineering*, pages 360–371, 2005.