Simple and Efficient Subclass Tests

Draft for ECOOP-02 – v24 – 30NOV01

Jonathan Bachrach
Artificial Intelligence Laboratory
Massachussetts Institute of Technology
Cambridge, MA 02139
`jrb@ai.mit.edu`

ABSTRACT

Fast subclass tests are crucial for the performance of object-oriented languages, especially those with dynamic typing. Unfortunately, fast constant time subclass encodings to date present a frustrating tradeoff: either choose a simple encoding with `O(n^2)` space requirements (where `n` is the number of classes) or a more complicated and slower to construct encoding with better space properties. In this paper, we present a new subclass test encoding, called the Packed Vector Encoding (PVE), that is fast, simple and requires average case `O(n log n)` space.

1. INTRODUCTION

Subclass tests determine whether one class is a subclass of another class. They are used throughout the implementation of an object-oriented language including in the compiler and often in the runtime. For example, they can be used for typechecks, downcasts, typecases, and method selection. Optimizing them can greatly improve the overall performance of object-oriented systems.

When designing a subclass test algorithm (*subclass?*), there are four important measures to consider:

1. The speed of the subclass? predicate,
2. the speed of the construction of subclass? data,
3. the space of necessary subclass? data, and
4. the support for, and cost of incremental changes (i.e., class (re)definitions).

Note that if the speed of construction is high enough then a full reconstruction could serve as an acceptable mechanism for supporting incremental changes.

This paper introduces a new subclass test encoding, called the Packed Vector Encoding (PVE). We start by formalizing the problem in Section 2. Then we discuss previous work in Section 3. From there, we introduce the PVE in Section 4. Next, in Section 5, we present PVE results and compare it against relavent algorithms. Finally, we present a summary and some recommendations in Section 6 and some remaining work in Section 7.

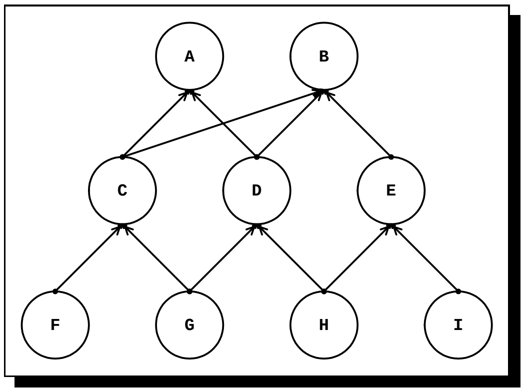


Figure 1: An example class hierarchy.

2. DEFINITIONS AND EXAMPLE

A class hierarchy is a partial order of classes. Figure 1 shows a class hierarchy (taken from [16]) drawn as a directed acyclic graph (DAG). This hierarchy will be used as a running example throughout the remainder of this paper. The conventions used in these class hierarchy figures are that parents are drawn above their children and that children point to their parents above.

We introduce the important properties of class hierarchies by defining them in Dylan [12] as shown in Figure 2. We chose to present algorithms and definitions in Dylan because it is high-level, object-oriented, and executable. We assume that regardless of the actual implementation language, that the properties defined in Figure 2 will be available to the subclass? test algorithm (e.g., level, ancestors, and descendents). We use the class name `<klass>` to avoid name conflicts with Dylan's `<class>` class.

We assume objects as shown in Figure 3 having direct access

```
define constant <buf>    = <stretchy-object-vector>;
define constant <vec>    = <simple-object-vector>;
define constant <int>    = <integer>;
define constant <mat>    = <array>;
define constant <bit>    = limited(<int>, min: 0, max: 1);
define constant <intvec> = limited(<vec>, of: <int>);
define constant <bitset> = limited(<set>, of: <bit>);

define constant $no-id = -1;

define class <klass> (<object>)
  slot parents :: <buf>;
  slot children :: <buf> = make(<buf>, 0);
  slot level :: <int>    = 0;
  slot id :: <int>       = $no-id;
end class;

define method initialize
    (k :: <klass>, #key parents, #all-keys)
  next-method();
  k.parents := as(<buf>, parents);
  for (parent in parents)
    parent.children := add(parent.children, k);
  end for;
end method;

define method assign-level (k :: <klass>)
  for(child in c.children)
    child.level := max(child.level, k.level + 1);
  end for;
  for (child in c.children)
    compute-level(child);
  end for;
end method;

define generic subclass?
  (x :: <klass>, y :: <klass>) => (well?);

define method ancestors (k :: <klass>) => (res :: <buf>)
  remove-duplicates
    (add(reduce(cat, #(), map(ancestors, k.parents)), k))
end method;

define method descendents (k :: <klass>) => (res :: <buf>)
  remove-duplicates
    (add(reduce(cat, #(), map(descendents, k.children)), k))
end method;

define method multiple-inheritance? (k :: <klass>) => (well?)
  size(k.parents) > 1 | any?(multiple-inheritance?, k.children)
end method;

define method single-inheritance? (k :: <klass>) => (well?)
  ~multiple-inheritance?(k)
end method;

define method depth (k :: <klass>) => (well?)
  reduce(max, k.level, map(level, k.children))
end method;
```

Figure 2: The basic klass structure, methods, and properties.

```
define class <objekt> (<object>)
  slot object-klass :: <klass>,
    required-init-keyword: klass:;
  repeated slot object-slots,
    size-getter: slot-size, size-init-keyword: size:;
end class;

define function isa? (x :: <objekt>, k :: <klass>) => (well?)
  subclass?(object-klass(x), k)
end function;
```

Figure 3: The basic object class and instance of test.

```
define method subclass? (x :: <klass>, y :: <klass>) => (well?)
  member?(y, x.ancestors)
end method;
```

Figure 4: The linear lookup subclass? algorithm.

to their class object. Typechecks can be performed with the `isa?` function. Again, we use the class name `<objekt>` to avoid name conflicts with Dylan's `<object>` class.

3. PREVIOUS WORK

The problem of efficient subclass? tests has been a long-standing challenge problem. Consult the surveys of Vitek et al. [13] and Zibin and Gil [16] for further information.

3.1 The Linear Lookup Algorithm

The simplest subclass? algorithm is to simply search a class' cached list of ancestors as shown in Figure 4. Unfortunately, this algorithm is $O(n)$, but because of its simplicity and modest space requirements it is often used in language implementations (e.g., [15]). Unfortunately, the linear lookup subclass? test is too large to inline into a call-site, and by its very nature does not yield predictable performance.

3.2 The Bitmatrix Algorithm

The *bitmatrix* algorithm is the simplest constant time algorithm. The n classes are numbered, and their numbers are used as indices into an $n \times n$ bitmatrix. The bitmatrix subclass? test, necessary additional klass slots, and construction algorithm are shown in Figure 6. We assume that the bitmatrix is represented by storing a bitvector row per klass. Note that the bit twiddling complexity is hidden by implementing the operations in terms of bitvectors.

As an example consider the example class hierarchy from Figure 1 with nine classes. The classes are labeled using consecutive nonnegative integer birth dates starting with zero as shown in Figure 7. A 9x9 bitmatrix is then constructed and populated as in Figure 6. The resulting bitmatrix is shown in Figure 5.

The advantages of the bitmatrix algorithm are that its subclass? test is constant time, it is fast to construct, works for multiple inheritance, and is generally very simple. These advantages make it appealing enough that many implementors use it as their mechanism of choice ([5] and [9]). The

	x	A	B	C	D	E	F	G	H	I
y	*id*	0	1	2	3	4	5	6	7	8
A	0	1	0	1	1	0	1	1	1	0
B	1	0	1	1	1	1	1	1	1	1
C	2	0	0	1	0	0	1	1	0	0
D	3	0	0	0	1	0	0	1	1	0
E	4	0	0	0	0	1	0	0	1	1
F	5	0	0	0	0	0	1	0	0	0
G	6	0	0	0	0	0	0	1	0	0
H	7	0	0	0	0	0	0	0	1	0
I	8	0	0	0	0	0	0	0	0	1

Figure 5: Bitmatrix for class hierarchy from Figure 7.

```
define class <klass> (<object>)
  ...
  slot row :: <bitvec>;
end class;

define method subclass? (x :: <klass>, y :: <klass>) => (well?)
  y.row[x.id] = 1
end method;

define method fab-subclass-encoding (klasses :: <vec>)
  let n = size(klasses);
  for (y in klasses)
    y.row := make(<bitvec>, n);
    for (child in y.children)
      y.row[child.id] := 1;
    end for;
  end for;
end method;
```

Figure 6: The bitmatrix subclass? predicate.

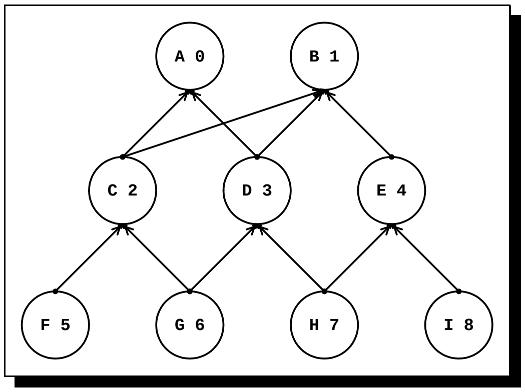


Figure 7: The example class hierarchy labeled with birth date ids.

```
define class <klass> (<object>)
  ...
  slot row :: <intvec>;
end class;

define method subclass? (x :: <klass>, y :: <klass>) => (well?)
  x.level < y.level & x.row[y.level] = y.id
end method;
```

Figure 8: The Cohen subclass? algorithm.

```
define class <klass> (<object>)
  ...
  slot bucket :: <int>;
  slot row :: <intvec>;
end class;

define method subclass? (x :: <klass>, y :: <klass>) => (well?)
  x.row[y.bucket] = y.id
end method;
```

Figure 9: The PE subclass? algorithm.

big disadvantage is that the size of the bitmatrix grows as $O(n^2)$.

3.3 Cohen's Algorithm

The first nontrivial constant time subclass? algorithm was proposed by Cohen ([4]). In this algorithm, classes are assigned unique integer ids and allocated a row containing its ancestor ids. A class `x` can then determine whether it is a subclass of another class `y` by looking to see whether `y`'s `id` is found in its `row` at the index corresponding to `y`'s `level`. Figure 8 shows the Cohen subclass? algorithm and its necessary additional klass support structure. Zibin and Gil ([16]) suggested that the range check can be removed by packing these rows into a single vector.

The advantages of Cohen's algorithm are that it is simple, fast, constant time, and requires modest space. The main disadvantage is that it only works for single-inheritance hierarchies.

3.4 The (B)PE Algorithm

Vitek et al. [13] generalized Cohen's algorithm to multiple inheritance and called it the Packed Encoding (PE). The general insight is that columns of the bitmatrix can be shared for unrelated classes. These shared columns are called buckets and a bucket assignment algorithm determines to which bucket a class belongs and the number of buckets required for a given class hierarchy. The general principle is that classes in the same bucket can not have common subclasses and must have distinct ids. Figure 9 shows the PE specific additional klass slots and PE's subclass? predicate.

The PE subclass? algorithm gives good compression over the bitmatrix algorithm, has a very fast constant time subclass? test, and is reasonably quick to construct. The big disadvantage is that, compared to the bitmatrix algorithm, it is

```
define class <klass> (<object>)
  ...
  slot min-id :: <int>;
  slot max-id :: <int>;
end class;

define method subclass? (x :: <klass>, y :: <klass>) => (well?)
  y.min-id <= x.id & x.id <= y.max-id
end method;

define function assign-ids (root :: <klass>, base :: <int>)
  local method descend (klass :: <klass>, this-id :: <int>)
          if (klass.id == $no-id)
            klass.id   := this-id;
            let min-id* = klass.id;
            let max-id* = klass.id;
            let next-id = this-id + 1;
            for (sub in klass.children)
              next-id := descend(sub, next-id);
              min-id* := min(min-id*, sub.min-id);
              max-id* := max(max-id*, sub.max-id);
            end for;
            klass.min-id := min-id*;
            klass.max-id := max-id*;
            next-id;
          else
            this-id
          end if;
        end method;
  descend(root);
end function;
```

Figure 10: The relative numbering subclass? algorithm.

still relatively complicated to understand and to implement efficiently.

3.5 Relative Numbering

One very efficient encoding of a class hierarchy is based on storing two additional values per class, a `min-id` and `max-id` value, which represent the min and max ids of a class and it's descendents. Figure 10 shows the necessary additional class slots, and how, in one preorder pass, classes are numbered and their min/max's are calculated. In a single inheritance class hierarchy, the relative numbering subclass? test works out to be a simple range test as shown in Figure 10.

The relative numbering algorithm is incredibly simple, has fast constant time algorithm, and exhibits $O(n)$ space requirements. As with Cohen's algorithm, the big problem is that it only works with single inheritance class hierarchies.

3.6 The PQ Encoding

Zibin and Gil [16] have generalized the relative numbering algorithm for multiple inheritance class hierarchies. Their encoding, called PQE, combines the ideas of relative numbering with the bucket idea of Vitek et al. [13] as shown in Figure 11. Buckets are assigned and rows initialized using PQ-trees [1].

The algorithm produces the best space results for any constant time multiple-inheritance capable algorithm. The PQE subclass? test is also very fast and compares well to the best constant time subclass? instruction sequence. Unfortunately, the PQE construction algorithm is complex and is slower to run than both the bitmatrix and PE construction algorithms.

```
define class <klass> (<object>)
  ...
  slot min-id :: <int>;
  slot max-id :: <int>;
  slot bucket :: <int>;
  slot row :: <intvec>;
end class;

define method subclass? (x :: <klass>, y :: <klass>) => (well?)
  let id = row[x.bucket];
  y.min-id <= id & id <= y.max-id
end method;
```

Figure 11: The PQE subclass? algorithm.

```
define class <klass> (<object>)
  ...
  slot code :: <bitset>;
end class;

define method subclass? (x :: <klass>, y :: <klass>) => (well?)
  subset?(x.code, y.code)
end method;
```

Figure 12: The Hierarchical Encoding subclass? algorithm.

3.7 Hierarchical Encodings

Hierarchical encodings perform subclass? tests by representing classes as sets of integers, typically a bitset. A class `x` is a subclass of `y` if `x`'s bitset is a subset of `y`'s. Figure 12 shows the necessary additional class slots and the subclass? test. Simple bitsets that obey this constraint are the rows of the bitmatrix representation. Of course the problem is that this representation still has $O(n^2)$ space requirements. Constructing an optimal bitset is NP-hard, but several researchers([10] and [2]) have devised heuristics for building bitset representations that have more modest space needs. Unfortunately, their construction algorithms are complicated. Furthermore, their subclass? tests require varying number of instructions.

3.8 Other Related Work

Eric Kidd [8] presents a related scheme for compressing method dispatch matrices. The idea is that each row can be better compressed by removing the initial and trailing zero's by recording the offset of the first and last nonzero entries.

Several researchers ([3], [11]) have utilized class numbering for method dispatch. The usual scheme is to number classes such that ids for classes and their children form a dense interval. Lists of relavent ranges are then constructed and are searched using code generated decision trees.

4. THE PACKED VECTOR ENCODING

This paper introduces the Packed Vector Encoding (PVE), a fast and simple subclass? encoding. The basic idea is to

```
define function assign-ids
    (root :: <klass>, base :: <int>) => (res :: <int>)
  local method descend (klass :: <klass>, this-id :: <int>)
          if (klass.id == $no-id)
            klass.id    := this-id;
            let next-id  = this-id + 1;
            for (sub in klass.children)
              next-id := descend(sub, next-id);
            end for;
            next-id;
          else
            this-id;
          end if;
        end method;
  descend(root, base);
end function;
```

Figure 13: A simple preorder class numbering scheme.

```
define function assign-min/max-ids
    (root :: <klass>, klasses :: <vec>)
  local method descend (klass :: <klass>)
          when (klass.min-id == $no-id)
            let min-id*        = klass.id;
            let max-id*        = klass.id;
            klasses[klass.id] := klass;
            for (sub in klass.children)
              descend(sub);
              min-id* := min(min-id*, sub.min-id);
              max-id* := max(max-id*, sub.max-id);
            end for;
            klass.min-id := min-id*;
            klass.max-id := max-id*;
          end when;
        end method;
  descend(root);
end function;
```

Figure 14: A min/max id assignment numbering procedure.

compress the sparse bitmatrix information into a vector using simple class numbering and range compression. The tradeoff is simplicity for slightly increased but manageable space characteristics.

4.1 Preorder Class Numbering

The strategy for class numbering is to choose class ids such that for each row the "ones" are moved into a single dense clump. The simplest class numbering algorithm with this property is a preorder depth first scheme shown in Figure 13, which tends to force the ids for a given class and its children into the densest possible interval. Figure 14 shows the algorithm for computing the minimum and maximum ids for a class and its children and collecting each class into a vector indexed by class id. Figure 15 shows the resulting labeling after running this simple class numbering algorithm on the class hierarchy from Figure 1. Figure 16 shows the clumps in the resulting bitmatrix. Note that this class numbering results in a far "clumpier" result than the birth date based class numbering scheme used to create the bitmatrix in Figure 5.

4.2 The Packed Vector Encoding

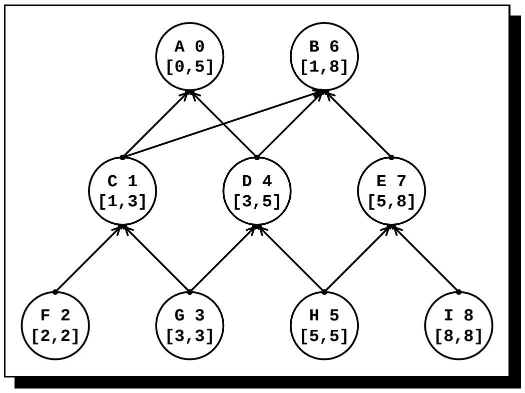
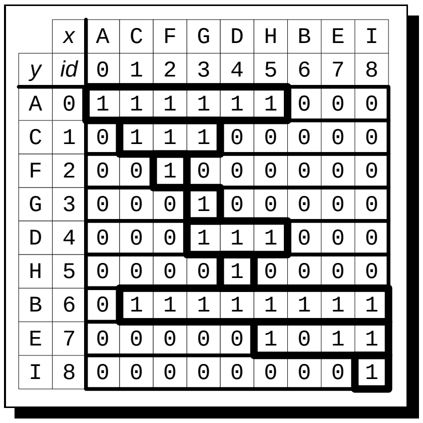


Figure 15: Results of simple preorder class numbering.



	x	A	C	F	G	D	H	B	E	I
y	*id*	0	1	2	3	4	5	6	7	8
A	0	1	1	1	1	1	1	0	0	0
C	1	0	1	1	1	0	0	0	0	0
F	2	0	0	1	0	0	0	0	0	0
G	3	0	0	0	1	0	0	0	0	0
D	4	0	0	0	1	1	1	0	0	0
H	5	0	0	0	0	1	0	0	0	0
B	6	0	1	1	1	1	1	1	1	1
E	7	0	0	0	0	0	1	0	1	1
I	8	0	0	0	0	0	0	0	0	1

Figure 16: The bitmatrix constructed using a preorder assignment of class ids.

```
define function assign-offsets
    (klasses :: <vec>) => (res :: <int>)
  let off = 0;
  for (klass in klasses)
    let range   = klass.max-id - klass.min-id + 1;
    klass.base := off;
    off        := off + range;
  end for;
  off
end function;
```

Figure 17: A procedure for assigning base offsets into the packed subclass vector.

```
define function fill-pv
    (klasses :: <vec>, pv-len :: <int>) => (res :: <intvec>)
  let pv = fill(make(<intvec>, size: pv-len), $no-id);
  for (klass in klasses)
    for (super in klass.ancestors)
      let offset = super.base - super.min-id;
      pv[klass.id + offset] := 1;
    end for;
  end for;
  pv
end function;
```

Figure 18: A procedure for populating a packed bit vector.

This bitmatrix can now be usefully compressed by packing the clumps into a single vector. Each class can easily record its base offset into this vector by summing the ranges of all lower numbered classes as shown in Figure 17, and the ones can be stored in the appropriately sized PVE as shown in Figure 18. The resulting packed bitmatrix vector is shown in Figure 19, and the corresponding subclass? predicate is shown in Figure 20. Unfortunately, this results in many more instructions than the bitmatrix subclass? algorithm of Figure 6.

4.3 Avoiding the Range Checks

Fortunately, we can improve this subclass? predicate by avoiding the range checks using a row unique "one" value, knowing that out of range accesses will be guaranteed to innocuously read invalid ids. The simplest choice is to just use a given row's class id as its unique key. Making this modification produces the subclass matrix shown in Figure 21, and the PVE shown in Figure 22. The new subclass? predicate is shown in Figure 23.

We can further improve this predicate by combining the `min-id` and `base` constants to form a single constant, called `offset`, which can be precomputed during `assign-offsets`.

y:	A						C			F	G	D			H	B								E				I
x:	A	C	F	G	D	H	C	F	G	F	G	G	D	H	H	C	F	G	D	H	B	E	I	D	H	B	E	I
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1

Figure 19: A packed bit subclass vector.

```
define class <klass> (<object>)
  ...
  slot min-id :: <int>;
  slot max-id :: <int>;
  slot base :: <int>;
end class;

define variable *pv* :: <intvec> = make(<intvec>, size: 0);

define function subclass?
    (x :: <klass>, y :: <klass>) => (well?)
  y.min-id <= x.id & x.id <= y.max-id
    & *pv*[y.base - y.min-id + x.id] = 1
end function;
```

Figure 20: The naive PVE subclass predicate with range checks.

	x	A	C	F	G	D	H	B	E	I
y	*id*	0	1	2	3	4	5	6	7	8
A	0	0	0	0	0	0	0	-	-	-
C	1	-	1	1	1	-	-	-	-	-
F	2	-	-	2	-	-	-	-	-	-
G	3	-	-	-	3	-	-	-	-	-
D	4	-	-	-	4	4	4	-	-	-
H	5	-	-	-	-	5	-	-	-	-
B	6	-	6	6	6	6	6	6	6	6
E	7	-	-	-	-	-	7	-	7	7
I	8	-	-	-	-	-	-	-	-	8

Figure 21: A subclass matrix using class ids as unique keys.

y:	A						C			F	G	D			H	B								E				I
x:	A	C	F	G	D	H	C	F	G	F	G	G	D	H	H	C	F	G	D	H	B	E	I	D	H	B	E	I
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
	0	0	0	0	0	0	1	1	1	2	3	4	4	4	5	6	6	6	6	6	6	6	6	7	-	7	7	8

Figure 22: A packed vector encoding using class ids as unique keys.

```
define function subclass?
    (x :: <klass>, y :: <klass>) => (well?)
  *pv*[y.base - y.min-id + x.id] = y.id
end function;
```

Figure 23: A more sophisticated PVE subclass predicate without range checks.

```
define class <klass> (<object>)
  ...
  slot min-id :: <int>;
  slot max-id :: <int>;
  slot offset :: <int>;
end class;

define function subclass?
    (x :: <klass>, y :: <klass>) => (well?)
  *pv*[y.offset + x.id] = y.id
end function;
```

Figure 24: The final optimized PVE subclass predicate.

```
define function assign-offsets
    (klasses :: <vec>) => (res :: <int>)
  let off = 0;
  for (klass in klasses)
    let range     = klass.max-id - klass.min-id + 1;
    klass.offset := off - klass.min-id;
    off          := off + range;
  end for;
  off
end function;
```

Figure 25: A procedure for assigning offsets into the PVE.

The final optimized subclass? predicate shown in Figure 24 is now competitive with the bitmatrix subclass? predicate of Figure 6. For completeness, the updated `assign-offsets` code is shown in Figure 25, the optimized PVE populating procedure is shown in Figure 26, and the top-level PVE construction procedure is shown in Figure 27.

4.4 Limited Size Keys – PVE$_k$

As currently described, PVE's require $\log_2 n$ bits per key compared to the single bit per key required by the bitmatrix algorithm. Even better compression could be achieved by using keys limited to $k < \log_2 n$ number of bits in the packed vector. In order to make this work, we must ensure that each clump's key is unique across the entire range of class ids. This constraint is easily achieved during the offset assignment process by adding a gap of negatives to the packed vector whenever the current key collides with that key's last max index, that is, whenever for a given `klass`, the max index of the last recorded `klass.id` is greater than `klass.base` minus `klass.min-id`, or `klass.offset`. The

```
define function fill-pv
    (klasses :: <vec>, pv-len :: <int>) => (res :: <intvec>)
  let pv = fill(make(<intvec>, size: pv-len), $no-id);
  for (klass in klasses)
    for (child in klass.children)
      pv[klass.offset + child.id] := klass.id;
    end for;
  end for;
  pv
end function;
```

Figure 26: The PVE populating procedure.

```
define method fab-subclass-encoding
    (roots :: <vec>) => (res :: <intvec>)
  let n = 0;
  for (root in roots)
    n := assign-ids(root, n);
  end for;
  let klasses = make(<vec>, size: n);
  for (root in roots)
    assign-min/max-ids(root, klasses);
  end for;
  let len = assign-offsets(klasses);
  fill-pv(klasses, len);
end method;
```

Figure 27: The PVE construction procedure.

`assign-lim-offsets` algorithm shown in Figure 28 shows the new PVE$_k$ construction procedure [1], subclass? test, and necessary additional klass slots. The PVE$_k$ offset assignment procedure (`assign-lim-offsets`) maintains a vector of last indices with an entry for each key adding a gap whenever necessary to enforce this constraint.

As an example, calling the `assign-lim-offsets` function with `k` equal to 2 produces the packed vector shown in Figure 29. One pad had to be inserted at index 11 to avoid a collision. The previous best PVE from Figure 22 required 4 bits per key times 28 entries resulting in a total of 112 bits. The new vector requires only 2 bits per key times 29 entries resulting in a total of 58 bits, which is 52% as small. As the number of bits decreases, the padding cost increases until `k` equals 1, when the algorithm becomes exactly the same as the bitmatrix algorithm.

Unfortunately, allowing arbitrary key sizes makes the subclass? predicate more complicated because it must now extract bit fields. Therefore, the best speed/space trade off would force the keys to sizes that are supported by the hardware, such as 8, 16, and 32 bit keys.

4.5 Multiple Inheritance

Unfortunately, the simple preorder class numbering scheme of Figure 13 is suboptimal for class hierarchy's involving multiple inheritance. In particular, mixins do not necessarily have nearby class numbers to their children. As a result, mixin ids will get assigned somewhat randomly, resulting in large and sparse clumps. A simple class numbering variation shown in Figure 30, aggressively walks mixins, increasing the likelihood that a mixin's id will be nearby to its children's ids.

Figure 31 shows the resulting class ids and clump ranges after running the multiple inheritance version of `assign-ids`. The resulting PVE is shown in Figure 32. Notice that the algorithm was able to squeeze out the single gap in clump 7 of Figure 22.

4.6 Space Analysis

[1]Note that a couple trivial changes would be required to top-level construction procedure as well.

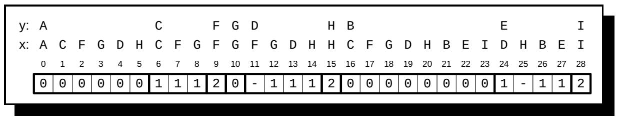
```
define class <klass> (<object>)
  ...
  slot min-id :: <int>;
  slot max-id :: <int>;
  slot offset :: <int>;
  slot key :: <int>;
end class;

define function subclass?
    (x :: <klass>, y :: <klass>) => (well?)
  *pv*[y.offset + x.id] = y.key
end function;

define function fill-pv
    (klasses :: <vec>, pv-len :: <int>) => (res :: <intvec>)
  let pv = fill(make(<intvec>, size: pv-len), $no-id);
  for (klass in klasses)
    for (child in klass.children)
      pv[klass.offset + child.id] := klass.key;
    end for;
  end for;
  pv
end function;

define function assign-lim-offsets
    (klasses :: <vec>, k :: <int>) => (res :: <int>)
  let nkeys      = 2 ^ k - 1;
  let max-bases  = fill(make(<intvec>, size: nkeys), 0);
  let nbase      = 0;
  for (klass in klasses, key = 0 then modulo(key + 1, nkeys))
    let min-id     = klass.min-id;
    let base       = max(ibase, max-bases[id] + min-id);
    let range      = klass.max-id - min-id + 1;
    klass.key     := key;
    klass.offset  := base - min-id;
    nbase         := base + range;
    max-bases[id] := nbase - 1;
  end for;
  nbase
end function;
```

Figure 28: A procedure for assigning PVE offsets given limited keys.



y:	A						C			F	G	D				H	B								E				I
x:	A	C	F	G	D	H	C	F	G	F	G	F	G	D	H	H	C	F	G	D	H	B	E	I	D	H	B	E	I
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
	0	0	0	0	0	0	1	1	1	2	0	-	1	1	1	2	0	0	0	0	0	0	0	0	1	-	1	1	2

Figure 29: A packed vector encoding using keys limited to two bits.

```
define function assign-ids
    (root :: <klass>, base :: <int>) => (res :: <int>)
  local method descend (klass :: <klass>, this-id :: <int>)
          if (klass.id == $no-id)
            klass.id    := this-id;
            let next-id  = this-id + 1;
            for (sub in klass.children)
              next-id := descend(sub, next-id);
            end for;
            for (sup in klass.parents)
              next-id := descend(sup, next-id);
            end for;
            next-id;
          else
            this-id;
          end if;
        end method;
  descend(root, base);
end function;
```

Figure 30: Mixin aware preorder class numbering function.

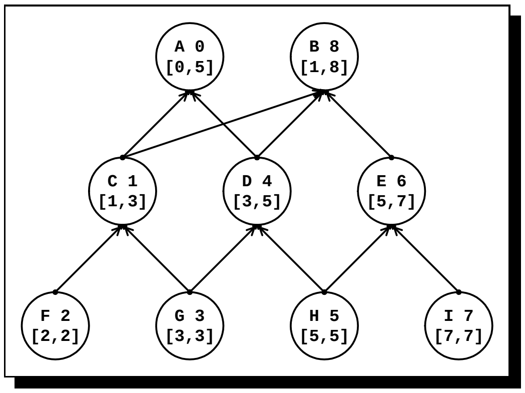
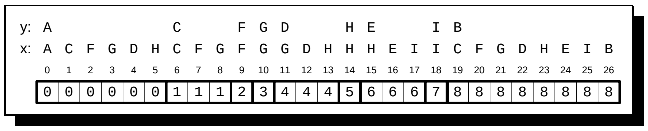


Figure 31: Mixin aware class numbering.



y:	A						C			F	G	D			H	E			I	B							
x:	A	C	F	G	D	H	C	F	G	F	G	G	D	H	H	H	E	I	I	C	F	G	D	H	E	I	B
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
	0	0	0	0	0	0	1	1	1	2	3	4	4	4	5	6	6	6	7	8	8	8	8	8	8	8	8

Figure 32: A packed vector encoding using a mixin aware preorder class numbering procedure.

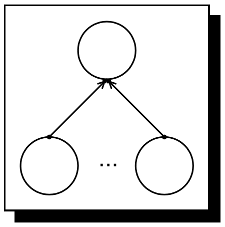


Figure 33: Best case Hierarchy.

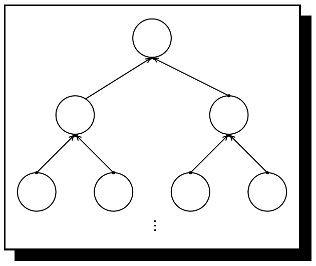


Figure 34: Worst Case Single Inheritance Hierarchy.

In this section we consider the best and worst case space properties of the PVE in order to better understand its space consumption. PVE has best case space usage of $O(n)$, worst case of $O(n^2)$, and worst case single inheritance of $O(n \log n)$. The best case arises from depth one single inheritance hierarchies of the form shown in Figure 33. The worst single inheritance case arises from binary trees as shown in Figure 34. Finally, the worst case arises from depth one hierarchies with an equal number of roots to children as shown in Figure 35. The actual space usage depends on the kind and amount of multiple inheritance.

5. RESULTS

Twenty real world data sets ranging in size from 66 to 5515 classes were used to test the various aspects of the PVE. The bulk of these class hierarchies were taken from ([6], [13], and [16]). We also added two new large hierarchies from Dylan [7] and Flavors [14]. Table 1 shows the properties of the various data sets. The class hierarchies are listed in order of number of classes. The amount of multiple inheritance is reported in the parents and ancestors columns. LOV and GEO represent an extreme use of multiple inheritance and are not representative of hierarchies created by humans.

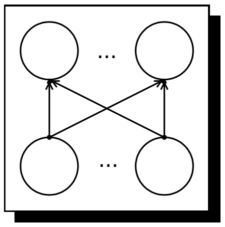


Figure 35: Worst Case Hierarchy.

Table 2 shows the space results with PVE_8 compared against the bitmatrix, PE, and PQE algorithms [2] with space numbers reported in kbytes. PQE results are reported only for hierarchies exhibiting multiple inheritance. PVE's compression over the bitmatrix algorithm gets better with increasing numbers of classes. For small n, the factor of eight win of the bitmatrix algorithm by using one bit keys dominates the $O(n^2)$ consumption, but as n grows, PVE's $\log n$ number of bits needed for keys combined with its approximate $O(n \log n)$ number of entries starts to kick in. PQE is the clear winner of compression with an average of 5x space advantage over PVE_8 [3]. Finally, PVE_8 consistently runs neck and neck with PE across the various class hierarchies, but in the end, PE compresses about 43% better than PVE_8.

Table 3 compares PVE sizes against the n^2, $n \log n$ and optimal baselines. The n^2 baseline is the number of entries in a bitmatrix, the $n \log n$ baseline is the worst case for single inheritance hierarchies and represents acceptable space growth, and the optimal baseline is the number of ones in the bitmatrix needed to represent a class hierarchy and the smallest PVE possible. First, we can see that the bitmatrix is often quite sparse with as low as 1.3% occupancy in the JAV30 hierarchy. Second, these results suggest that PVE's space usage is pretty close to $n \log n$ and that the optimal baseline is always smaller than $n \log n$ for the example hierarchies. Finally, based on the optimal results, there appears to be approximately a factor of up to about 12 room for improvement for PVE.

Table 4 compares sizes with various values of k in the PVE_k algorithm. Note that PVE's key sizes are automatically chosen to be either 8 or 16 depending on the number of classes in the class hierarchy. It is clear that smaller keys can improve the compression and that too small a value of k can worsen the compression. A reasonable choice of k for the sample hierarchies is 8.

Table 5 shows the construction time results of PVE_8 compared against PE and bitmatrix with time reported in milliseconds. The timings were conducted on a 700Mhz Pentium III notebook computer. All algorithms except for PQE were coded in Dylan with full optimizations, adequate type declarations, and range checks turned off. The PE algorithm was implemented in a rather straightforward fashion using bitsets for the sets [4]. The PQE algorithm was written in C++ and was supplied by the PQE authors [5].

First, we can see that PVE_8's construction algorithm is on average 4x faster than bitmatrix's. Second, PVE_8's con-

[2]Consult either [13] or [16] for space results for and comparisons against the NHE algorithm.

[3]Unfortunately, this advantage might not be so dramatic in the dynamic case (e.g., where y is unknown statically, perhaps by choice), where their heterogenous representations might compromise subclass? test performance.

[4]A more simple-minded PE implementation using stretchy vectors instead of bitsets resulted in times approximately 10x slower.

[5]The PQE times do not include required hierarchy transformations and calculations that preliminary timings indicate could add an additional 30% overhead.

Name	n	Max Depth	Max/Avg # Parents	Max/Avg # Ancestors	Language	Description
IDL	66	6	2/1.1	8/3.8	IDL	IDL
JAV	225	6	3/1.2	7/3.2	Java	JDK 1.1
LAU	293	11	3/1.1	16/8.2	Laure	Base system
NST	309	6	1/1.0	7/3.0	Objective-C	NextStep
ED	434	10	7/1.7	23/8.0	Laure	Editor
LOV	436	9	10/1.8	24/8.5	LOV	Base system
UNI	613	8	2/1.0	9/3.0	C++	Unigraph
CEC	932	12	3/1.2	23/6.5	Cecil	System
CLO	1070	12	9/1.4	51/8.0	CLOS	Symbolics
GEO	1318	13	16/2.1	50/14.0	LOV	Machine-generated
DT3	1355	12	1/1.0	13/5.4	Smalltalk	Digitalk3
JAV18	1704	9	11/1.1	16/4.3	Java	JDK 1.18
SLF	1801	16	9/1.0	40/29.9	Self	System
VW2	1956	14	1/1.0	15/6.4	Smalltalk	VisualWorks 2
EIF	1999	17	10/1.3	39/8.8	Eiffel	Eiffel4
VA2	3239	13	2/1.0	14/6.4	Smalltalk	VisualAge 2
JAV22	4339	9	14/1.2	17/4.4	Java	Java 1.22
DYL	4931	14	9/1.2	40/7.3	Dylan	Functional Developer
JAV30	5438	9	14/1.2	19/4.4	Java	JDK 1.30
FLA	5515	12	53/5.3	54/7.1	Flavors	Symbolics
Avg.	1898.6	10.9	9.0/1.4	24.2/7.5		

Table 1: Properties of class hierarchy data sets.

Name	BM	PE	PE % BM	PQE	PQE % BM	PVE_8	PVE_8 % BM	PVE_8 % PE	PVE_8 % PQE
IDL	0.5	0.4	72.8	0.1	20.0	0.3	49.3	67.7	300.0
JAV	6.3	1.6	24.9	0.2	3.2	0.7	11.8	47.3	350.0
LAU	10.7	4.7	43.7	0.5	4.7	5.9	54.6	125.0	1180.0
NST	11.9	0.6	5.2			0.9	7.6	147.6	
ED	23.5	11.7	49.8			18.9	80.2	161.2	
LOV	23.8	12.2	51.4	2.7	11.3	23.4	98.7	192.1	866.7
UNI	47.0	4.9	10.4	0.8	1.7	2.0	4.3	41.1	250.0
CEC	108.6	20.5	18.9	3.7	3.4	18.4	16.9	89.7	497.3
CLO	143.1	54.6	38.1	10.8	7.5	18.5	12.9	33.9	171.3
GEO	217.1	68.5	31.6	14.5	6.7	153.8	70.9	224.6	1060.7
DT3	229.5	2.7	1.2			8.7	3.8	319.2	
JAV18	363.0	27.3	7.5	7.0	1.9	25.7	7.1	94.4	367.1
SLF	405.4	68.4	16.9	10.6	2.6	59.4	14.7	86.8	560.4
VW2	478.2	3.9	0.8			15.9	3.3	406.8	
EIF	499.5	78.0	15.6	18.2	3.6	216.5	43.4	277.7	1189.6
VA2	1311.4	29.2	2.2			34.7	2.6	118.9	
JAV22	2353.4	78.1	3.3	24.4	1.0	109.2	4.6	139.8	447.5
DYL	3039.3	207.1	6.8	50.5	1.7	120.0	4.0	58.0	237.6
JAV30	3696.5	103.3	2.8	31.9	0.9	154.5	4.2	149.5	484.3
FLA	3801.9	297.7	7.8	102.0	2.7	286.0	7.5	96.0	280.4
Avg.			20.6		4.9		25.1	143.8	545.5

Table 2: Space results of the bitmatrix, PE, PQE, and PVE_8 algorithms given in kbytes.

NAME	n^2	$n \log n$	opt.	opt.	opt. %	PVE	PVE %	PVE %
				% n^2	$n \log n$		$n \log n$	opt.
IDL	0.5	0.5	0.3	46.5	54.8	0.3	58.0	105.9
JAV	6.3	1.8	0.7	11.3	39.7	0.7	41.4	104.3
LAU	10.7	2.6	2.4	44.6	90.7	5.9	222.3	245.0
NST	11.9	2.8	0.9	15.3	32.8	0.9	32.8	100.0
ED	23.5	3.9	3.5	29.4	88.7	18.9	483.5	544.9
LOV	23.8	3.9	3.7	31.2	94.5	23.4	597.5	632.5
UNI	47.0	6.1	1.9	7.9	30.2	1.9	30.5	100.9
CEC	108.6	9.3	6.0	11.1	64.6	18.2	194.8	301.0
CLO	143.1	11.8	8.6	12.0	72.8	18.2	154.5	212.1
GEO	217.1	14.5	18.4	17.0	127.2	153.8	1061.8	834.7
DT3	229.5	14.9	7.3	6.4	49.1	7.3	49.1	100.0
JAV18	363.0	18.7	7.4	4.1	39.5	23.2	123.6	312.7
SLF	405.4	19.8	53.8	26.6	271.7	57.7	291.2	107.2
VW2	478.2	21.5	12.5	5.2	58.2	12.5	58.2	100.0
EIF	499.5	22.0	17.6	7.0	79.8	214.9	977.2	1223.7
VA2	1311.4	38.9	20.7	3.2	53.1	20.7	53.3	100.3
JAV22	2353.4	56.4	18.9	1.6	33.6	85.0	150.6	448.6
DYL	3039.3	64.1	36.0	2.4	56.2	90.6	141.3	251.3
JAV30	3696.5	70.7	23.8	1.3	33.6	111.8	158.2	470.4
FLA	3801.9	71.7	39.4	2.1	55.0	259.0	361.2	657.3
Avg.				14.3	71.3		262.1	347.6

Table 3: Comparison of PVE against n^2, $n \log n$, and optimal baselines reporting number of K entries.

Name	BM	PE	PQE	PVE_8	PVE_8	PVE_8	PVE_8
					% BM	% PE	% PQE
IDL	0.3	3.1	3.0	1.0	42.7	3.5	33.3
JAV	0.8	12.7	3.0	0.3	35.8	2.3	10.0
LAU	2.5	18.5	4.0	0.6	24.6	3.4	15.0
NST	1.1	14.2		0.4	37.4	2.8	
ED	3.7	41.4		1.6	43.5	3.9	
LOV	4.4	42.1	58.1	1.4	30.8	3.2	2.4
UNI	2.4	31.8	3.0	0.8	34.8	2.6	26.7
CEC	7.6	86.5	46.0	2.1	27.7	2.4	4.6
CLO	40.9	211.4	118.2	3.1	7.5	1.4	2.6
GEO	23.6	204.3	461.6	8.3	35.1	4.0	1.8
DT3	63.1	71.3		3.1	4.8	4.3	
JAV18	12.1	158.0	22.1	4.4	36.6	2.8	19.9
SLF	176.3	214.6	89.1	8.4	4.8	3.9	9.4
VW2	54.6	127.2		12.3	22.6	9.7	
EIF	26.9	285.6	235.3	11.6	42.9	4.0	4.9
VA2	38.8	210.0		10.0	25.7	4.8	
JAV22	50.8	786.7	125.2	14.9	29.3	1.9	11.9
DYL	76.7	891.9	333.5	24.3	31.7	2.7	7.3
JAV30	73.3	1238.7	179.3	19.9	27.1	1.6	11.1
FLA	251.1	1884.8	2991.3	34.4	13.7	1.8	1.2
Avg.					27.9	3.4	10.8

Table 5: Construction time results of the bitmatrix, PE, and PVE_8 algorithms given in milliseconds.

Name	PVE	PVE_4	PVE_4 % PVE	PVE_8	PVE_8 % PVE
IDL	0.3	0.2	61.9	0.3	100.0
JAV	0.7	1.0	137.2	0.7	100.0
LAU	11.7	3.9	33.7	5.9	50.0
NST	1.8	1.8	100.6	0.9	50.0
ED	37.8	10.9	29.0	18.9	50.0
LOV	46.9	13.1	28.0	23.4	50.0
UNI	3.7	6.7	178.4	2.0	53.8
CEC	36.3	21.5	59.1	18.4	50.7
CLO	36.4	23.5	64.6	18.5	50.9
GEO	307.9	96.3	31.3	153.8	50.0
DT3	14.6	32.7	223.6	8.7	59.1
JAV18	46.3	56.7	122.4	25.7	55.5
SLF	115.4	78.9	68.3	59.4	51.5
VW2	25.0	67.4	269.2	15.9	63.5
EIF	429.8	155.7	36.2	216.5	50.4
VA2	41.4	180.8	436.3	34.7	83.6
JAV22	169.9	344.0	202.4	109.2	64.3
DYL	181.2	431.4	238.1	120.0	66.3
JAV30	223.6	535.0	239.2	154.5	69.1
FLA	517.9	589.0	113.7	286.0	55.2
Avg.			133.7		61.2

Table 4: Effect of k on space in kbytes.

struction times are much lower than PE and PQE across the board. In fact, PVE_8 is up to 100x faster to construct than PE or PQE. There is more that could be done to further optimize the PE construction algorithm, but a significant effort was made for the purposes of this paper. The construction times for PE reported in ([13] and [16]) suggest that PE's can be constructed faster but clearly at the cost of implementation complexity. Finally, PVE_8's construction time is low enough to consider it as a mechanism for incremental update. Even for the largest hierarchy (FLA) with 5515 classes, the full reconstruction takes under 35 milliseconds. It is likely that these PVE_8 construction times are conservative given their implementation in such a high-level language as Dylan.

6. CONCLUSIONS

The PVE is a very simple and efficient subclass? test algorithm. It is fast enough to be used in incremental settings for dynamic class (re)definition. It is easy to understand and can be implemented in a page of code. In comparison, the two previous best encodings, PE and PQE, require at least a 10x and 100x implementation effort respectively [6]. While PVE's are not as small as those produced by PQE, it does produce encodings comparable to PE, the previous best encoding.

7. FUTURE

The comparison between PVE and optimal PVE from Table 3 shows that there is still much room for improvement in the numbering / packing algorithm. We will be investigating other simple algorithms that further increase compression.

[6]In the very least, implementors would need to include and potentially tune this relative number of lines of code.

8. CREDITS

This paper benefitted from helpful discussions with Craig Chambers, James Knight, Greg Sullivan, and Jan Vitek. Zibin and Gil [16] supplied their PQE software making possible PQE comparisons. We thank Gary Palter, Kalman Reti, Jan Vitek, and Zoav Zibin for supplying the various class hierarchies. We are especially grateful to Eric Kidd for many fruitful conversations and for general inspiration. Finally, Greg Sullivan and Eric Kidd read and commented on many drafts of this paper leading to great improvements.

9. REFERENCES

[1] K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using P-Q tree algorithms. *J. of Comp. and Syst. Sci.*, 13:335–379, 1976.

[2] Yves Caseau. Efficient Handling of Multiple Inheritance Hierarchies. In *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 271–287, October 1993. Published as Proceedings OOPSLA '93, ACM SIGPLAN Notices, volume 28, number10.

[3] Craig Chambers and Weimin Chen. Efficient multiple and predicate dispatching. In Loren Meissner, editor, *Proceeings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34.10 of *ACM Sigplan Notices*, pages 238–255, N. Y., November 1–5 1999. ACM Press.

[4] N. H. Cohen. Type-extension tests can be performed in constant time. In *ACM Transactions on Programming Languages and Systems*, volume 13, pages 626–629, 1991.

[5] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. Vortex: An optimizing compiler for object-oriented languages. In *Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '96)*, pages 83–100, 1996.

[6] Natalie Eckel and Joseph Gil. Empirical study of object-layout strategies and optimization techniques. In Elisa Bertino, editor, *ECOOP '00 — Object-Oriented Programming 14th European Conference*, volume 1850 of *Lecture Notes in Computer Science*, pages 394–421. Springer-Verlag, New York, NY, June 2000.

[7] Functional Objects Inc. Functional developer. `www.fun-o.com`, 2001.

[8] Eric Kidd. Efficient Compression of Generic Function Dispatch Tables. Technical Report TR2001-404, Dartmouth College, Computer Science, Hanover, NH, June 2001.

[9] Andreas Krall and Reinhard Grafl. CACAO — A 64-bit JavaVM just-in-time compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, November 1997. Special Issue: Java for computational science and engineering — simulation and modeling II.

[10] Andreas Krall, Jan Vitek, and R. Nigel Horspool. Near optimal hierarchical encoding of types. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 128–145. Springer-Verlag, New York, NY, June 1997.

[11] Christian Queinnec. Fast and compact dispatching for dynamic object-oriented languages. Submitted to IPL.

[12] A. Shalit. *The Dylan Reference Manual*. Addison Wesley, 1996.

[13] Jan Vitek, Nigel Horspool, and Andreas Krall. Efficient type inclusion tests. In Toby Bloom, editor, *Conference on Object Oriented Programming Systems, Languages & Applications (OOPSLA'97)*, pages 142–157, Atlanta, October 1997. ACM.

[14] D. Weinreb and David Moon. *The Lisp Machine Manual*. Symbolics Inc., 1981.

[15] N. Wirth. Type extensions. *ACM Transactions on Programming Languages and Systems*, 10(2):204–214, April 1988.

[16] Yoav Zibin and Joseph (Yossi) Gil. Efficient subtyping tests with pq-encoding. In Toby Bloom, editor, *Conference on Object Oriented Programming Systems, Languages & Applications (OOPSLA'01)*, pages 142–157, Tampa, October 2001. ACM.