An Approach to improve Locality using Sandwich Types

Daniela Genius Martin Trapp Wolf Zimmermann

Institut für Programmstrukturen
und Datenorganisation
University of Karlsruhe
76128 Karlsruhe
Germany

Abstract

We show how to increase locality of object-oriented programs using several heaps. We introduce the notion of sandwich types which allow a coarser view on objects. Our idea for increasing locality is to use one heap per object of sandwich types. We demonstrate by performance measurements the effect of this strategy on running time, and show how to derive sandwich types from classes. Thus, it is possible to control the allocation of the different heaps using compile-time information.

1 Introduction

In object-oriented programs, the notion of an object is rather fine-grained. The objects are usually allocated on a heap and the size of these objects is small. Thus, a single heap may destroy locality. Improving locality may improve execution time due to caching and paging effects. Often, a coarser view on objects is possible. For example, a list may be considered as a collection of small objects linked in an adequate way, but it may also be considered as one object. We introduce the notion of *sandwich types* in order to characterize this situation. Our goal is to maintain objects of sandwich types (called *sandwich objects*) in one single heap (i.e. consecutive fragment of memory) in order to increase locality. We maintain these heaps for a sandwich object by the doubling strategy well-known from the theory of algorithms and data structures (see e.g. [CLR91]).

This work was initiated by observations during experiments where lists, trees, sets etc. were implemented with flexible arrays using the doubling strategy. These implementations improved the performance considerably compared to linked implemementations.

Increasing locality of reference by partitioning the heap is a well known technique. The language Euclid [Hor78] introduced special *collections* which can be viewed as independent heaps. Dynamically allocated data structures could be assigned to a single collection. The same idea is exploited by the Gnu obstack structure [Fre]. This package gives the programmer control over an arbitrary number of heaps that require stack discipline for allocation and deallocation.

However, the responsibility for mapping objects to heaps remains totally with the programmer, both in using Euclid as well as obstack.

Approaches in automatically finding such mappings have been developed in the context of Smalltalk. The Object-Oriented Zoned Environment (OOZE) [Ing78] locates all instances of a type in one contiguous interval of virtual addresses. This increases locality of reference for objects of the same type, but is not able to consider structures built from objects of various types.

Stamos [Sta84] presents also additional algorithms for grouping related objects with the intention of increasing locality of reference. They require complete knowledge of the dynamic object graph and are used for restructuring the memory image during garbage collection. I.e. the mapping can not be found at compile time. To the authors' knowledge, there is no work on automatic a priori mapping of dynamically allocated objects to multiple heaps for sequential object-oriented programms.

Section 2 introduces the notion of sandwich types and gives some examples. Section 3 shows the performance improvements obtained by object heaps. Finally, Section 4 shows a conservative analysis for identifying sandwich types. Section 5 concludes our paper.

2 Sandwich Types

The notion of *sandwich types* is a generalization of balloon types [Alm97]. All objects in a balloon can be accesses only via a distinguished balloon object. However, this excludes container types such as e.g. lists or sets. These types usually contain methods to return their elements. Thus, these elements can be accessed from outside. However, there is often an internal structure which cannot be accessed from outside. This is the reason for the term *sandwich object*: its internal structure can be accessed only via the sandwich object, but parts of its structure are known externally, but it is the sandwich object which decides what is external. Figure 1 visualizes this idea.

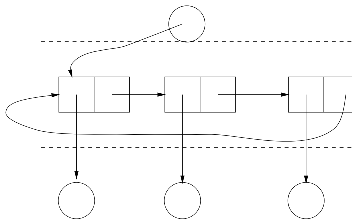

Figure 1: A Sandwich

We now define these notions more precisely. In this paper, a *class* A consists of a set of attributes (variables) and methods (procedures or functions). A class with attributes x_i of class T_i, $i = 1, \dots, k$ and procedures m_j whose input is of type $T_1^{(j)} \times \cdots \times T_{r_j}^{(j)}$, $j = 1, \dots, l$ is denoted by

class A **is**
  attr $x_1 : T_1$
    $\vdots$
  attr $x_k : T_k$
  method $m_1(T_1^{(1)}, \dots, T_{r_1}^{(1)})$
    $\vdots$
  method $m_l(T_1^{(l)}, \dots, T_{r_l}^{(l)})$
end

Functions are denoted by return types, i.e. a function $f : T_1 \times \cdots \times T_k \to T$ is denoted by $f(T_1, \dots, T_k) : T$. $\#A$ denotes the creation of an object of class A.

Remark: We do not consider inheritance or method implementations since this plays no role in the discussion. We also do not consider generic classes. Instead, we assume that in a program the parameters of every generic class are instantiated. ■

Attributes of a class A may be private, i.e. for any object x of class A these attributes can only be accessed by x. The notion of sandwich objects is based on reachability. A state of a program is a triple $(OBJ, REF, STACK)$ where OBJ is a set of objects, each element of the stack contains a set $S \subseteq \{OBJ\}$, and $STATE = (OBJ, REF)$ is directed graph where $(o_1, o_2) \in REF$ iff there is an attribute of object o_1 that refers to object o_2. $\overset{*}{\to}$ denotes the reachability relation in states. $type(x)$ denotes the type of an object x. An object o is *reachable* iff there is an object $o' \in STACK$ such that $o' \overset{*}{\to} o$. Otherwise it is unreachable. [SZ94a, SZ94b] defines an operational semantics based on this definition of states. Suppose that there is a state transition such that objects become unreachable in a state. Then, it will be unreachable forever. Therefore, we can assume w.l.o.g. that the unreachable objects are removed, i.e. no state contains unreachable objects. The paper does not require any further knowledge of state transitions.

Definition 1 (Sandwich Type) *Let A be a class whose attributes are all private. Let x be an object of class A and $s = (OBJ, REF, STACK)$ be a state. The set $INTERNAL_x^{(s)}$ of objects* internal *to x is the smallest set satisfying*[1]

$$INTERNAL_x^{(s)} = \{y : x \overset{+}{\to} y \wedge \forall z \in PRED_y : z \in \{x\} \cup INTERNAL_x^{(s)}\}$$

x is a sandwich object *or* upper slice *iff $INTERNAL_x^{(s)} \neq \emptyset$ or $outdeg_x = 0$. All objects $z \in OBJ \setminus (\{x\} \cup INTERNAL_x^{(s)})$ are external. If $outdeg_x \neq 0$, the* lower slice *of x is the set of all external objects that have a predecessor $y \in INTERNAL_x^{(s)}$. A* sandwich *is a sandwich object x together with the set of its internal objects. A is a* sandwich type *iff for all states s every object x of type A in s is a sandwich object.*

Remark: If there is no lower slice, a sandwich object is also a balloon object according to [Alm97]. Observe that the attributes of internal objects need not be private. ■

Example: Consider the following implementation of a doubly-linked list (*next* and *previous* are used to navigate in the list):

```
class LIST(T) is
    private attr head : LIST_CELL(T)
    private attr end : LIST_CELL(T)
    private attr current : LIST_CELL(T)
    method previous
    method next
    method insert(T)
    method delete
    method elem : T
    method is_empty : BOOL
    method at_head : BOOL
    method at_end : BOOL
end
```

[1] $\overset{+}{\to}$ denotes the proper reachability relation on states.

class $LIST\_CELL(T)$ **is**
  attr $elem : T$
  attr $previous : SAME$
  attr $next : SAME$
end

For every type T, $LIST(T)$ is a sandwich type. Let $x : LIST(T)$. Then x is a sandwich object. Its head and its end are internal, for example. The elements of the list are the lower slice of the sandwich. ∎

Container types are typical examples of sandwich types.

Internal objects can be reached only via the upper slice, i.e.

Lemma 1 *Let $s = (OBJ, REF, STACK)$ be a state and $x \in OBJ$ a sandwich object of type A. Then for every $y \in OBJ$, $y \in INTERNAL_x^{(s)}$, iff for every $z \in OBJ$ such that $z \xrightarrow{*} y$ one of the following conditions hold:*

(i) $z \in INTERNAL_x^{(s)}$.

(ii) Each path from z to y contains x.

Proof: "⇒": Suppose this would not be the case, i.e. there is a $z \notin INTERNAL_x^{(s)}$ and a path π from z to y not containing x. Since $y \in INTERNAL_x^{(s)}$ there must be $u, v \in \pi$ such that $u \notin INTERNAL_x^{(s)}$, $v \in INTERNAL_x^{(s)}$, and $(u, v) \in REF$. The definition of $INTERNAL_x^{(s)}$ implies that $u = x$ contradicting our assumption that π does not contain x.

"⇐": Suppose (i) and (ii) holds, but $y \notin INTERNAL_x^{(s)}$. Then either, $x \overset{+}{\not\to} y$ or there is a $z \in PRED_y$ and $z \notin INTERNAL_x^{(s)} \cup \{x\}$. The latter contradicts (ii). Thus $x \overset{+}{\not\to} y$. Consider a $z \in OBJ$ such that $z \xrightarrow{+} y$. Then (i) must hold since (ii) is excluded by $x \overset{+}{\not\to} y$. (i) implies $x \xrightarrow{+} z$, contradicting $x \overset{+}{\not\to} y$. Thus, there is no path to y, i.e. y cannot be reached. This is what we excluded from the definition of states. ∎

The next Lemma states that sandwiches are either disjoint or nested:

Lemma 2 *Let $s = (OBJ, REF, STACK)$ be a state, $x \in OBJ$ be any sandwich object of type A and $y \in OBJ$ be a sandwich object of type B. If y is internal to x, then every object of the lower slice of x is external to y.*

Proof: Suppose there is a state s such that $y \in INTERNAL_x^{(s)}$ and there is an object $u \in INTERNAL_y^{(s)}$ of the lower slice of x. Figure 2 visualizes this situation. Since u is in the lower slice of x, it is external to x, i.e., there is a path from an object w external to x and not containing x. Since $u \in INTERNAL_y^{(s)}$, by Lemma 1, every path from an object external to y must contain u. Thus, w must be internal to y.

Let v be an object external to x. Since y is internal in x, every path from v to y must contain x (by Lemma 1). Since w is internal to y, every path from x to w must contain y (by Lemma 1). Hence, every path from v to w must contain x. This contradicts the fact that w is external to x. ∎

Example: Consider the class $HASHTAB(T)$ with collision resolution by chain lists:

class $HASHTAB(T)$ **is**
  private attr $tab : ARR[n](LIST(T))$
  method $insert(x : T)$
  method $delete$
  method $member(T)$ **end**

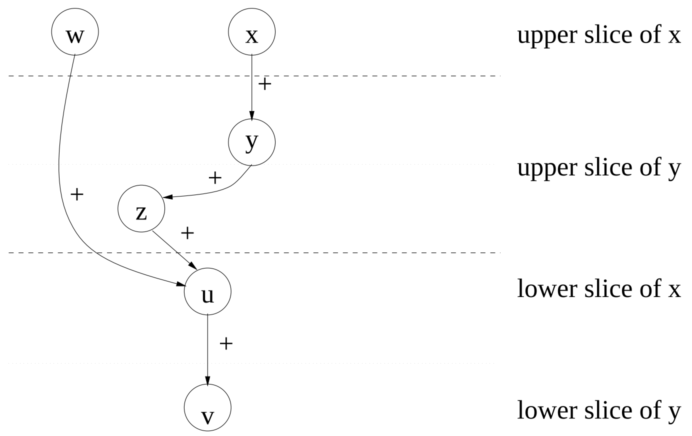

Figure 2: Contrary of Lemma 2

Objects of class $HASHTAB(T)$ are sandwich objects. The collision lists in the array *tab* are also sandwich objects. Each lower slice of a collision list is contained in the lower slice of hash tables. ■

3 Performance Improvement Using Object Heaps

In general, the number of objects that will be allocated in a sandwich's heap is unknown at compile time. Thus, the heap must be able to grow (and shrink) at runtime. To achieve locality of reference for the objects in a heap, the latter must extend over a minimal number of physical memory pages. I.e. all but at most one page used must be used completely or not at all for object allocation. We guarantee this by allocating a contiguous[2] area of virtual memory for a heap and doubling the size whenever the heap would overflow. Since it is not possible to grow a heap in place, we copy its data to a new memory area. This works in amortized constant time, see [CLR91].

Note that all references pointing at objects in a sandwich's heap come from inside that heap or from its sandwich object. We exploit this fact by using a compacting copy garbage collector [Jon96] to move the objects from the old memory area to the new one.

The root set for the collector is the singleton set containing just the sandwich object. The important advantage is that the heap of a sandwich can be copy collected independently of all external objects. If after garbage collection less than a quarter of the heap is occupied by allocated objects, we halve its size. Whenever a sandwich object is garbage collected, its heap can be deleted at once.

In our measurements we take up the small list example. In our test program two empty list are created. We alternately insert single elements in both lists until they hold MAX elements, each. Afterwards, the test program iterates through each list separately ITER times. Figure 3 shows the runtime of this iterated list traversals depending on the length of the lists. We have chosen the values of of MAX and ITER so that their product is constant (10^7). In the first part of the plot, runtime decreases with the length of the list because the iteration overhead becomes less significant.

Both axes of the plot are logarithmically scaled. The x-axis shows the number of elements in the list, while the y-axis shows the total runtime in seconds. The values are measured on a 200

[2]If it is guaranteed that the sizes of all internal objects evenly divide the physical page size, it would not matter whether the pages are contiguous in virtual memory.

Mhz i586 linux system. Each program is run 10 times and the smallest elapsed time is shown in Figure 3.

The curve labelled single shows run times for the usual implementation with one heap for all objects created by the program. The multi curve denotes the result of our method: As noted above, LIST(T) is a sandwich type. Both list objects are sandwich objects by themselves. Thus, they have their own heaps for their internal LIST_CELL(T) objects. There is a third heap for all other objects created by the program.

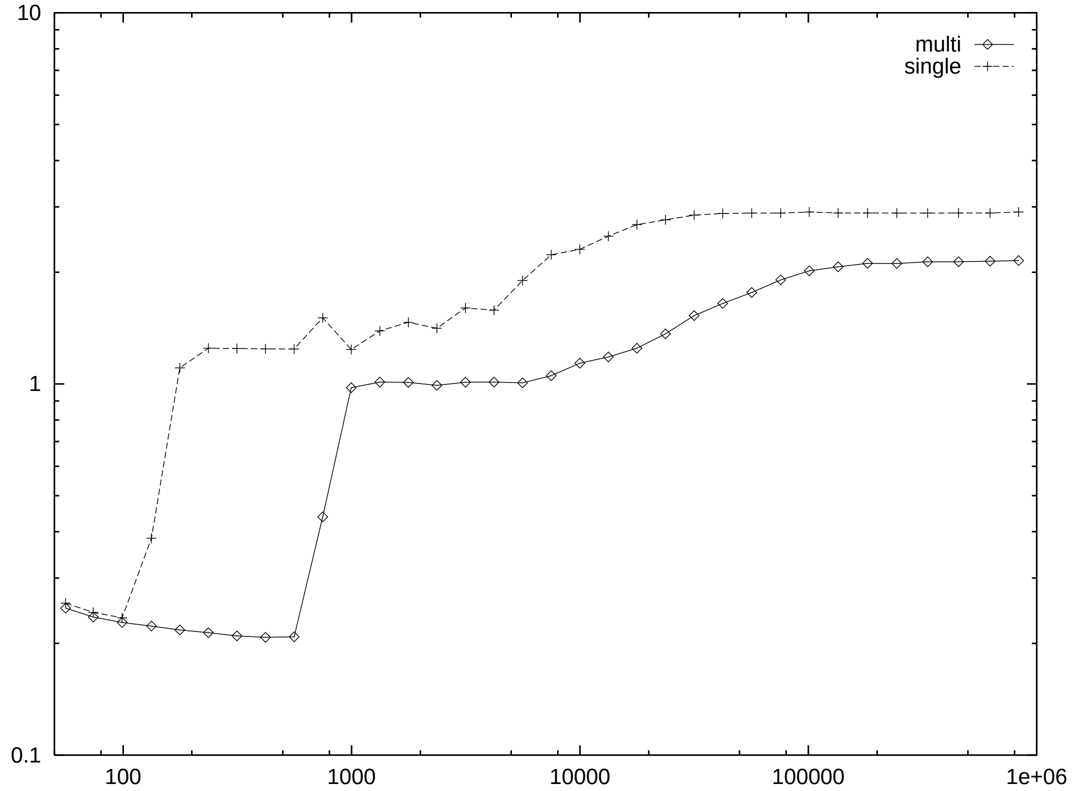

Figure 3: Measurements

Both variants start with run times below 0.3 seconds as long as all list elements fit into the data cache. Cache misses start at about 100 elements for the single heap implementation. With the multi heap implementation this effect is postponed to lists of about 500 elements because of the better locality. Also after the sharp increase in run time due to cache misses and page faults the multi heap variant clearly outperforms the single heap version. Less memory pages have to be accessed to traverse the lists. For very long lists results for classical allocation stay below 3 seconds, while using multiple heaps never greatly exceeds 2 seconds. An overall improvement by our method of about 25 per cent emerges in this case. In general the improvement is even larger: For lists of around 10000 elements the multi heap variant is twice as fast.

4 Recognition of Sandwich Types

This section shows how sandwich types can be recognized in a program. We first derive a sufficient condition for sandwich types. This sufficient condition abstracts from the state by considering the types of the objects. In particular, we consider the *type graph* of an object-oriented program, i.e. a graph $TG = (CLASSES, USE)$ where $CLASSES$ are the classes of a program and $(A, B) \in USE$ iff A has an attribute of type B or has a method with parameter type B or return type B. $\stackrel{*}{\Rightarrow}$ denotes the reachability relation in type graphs.A class A is *recursive* iff $A \stackrel{+}{\Rightarrow} A$. We lift now parts of Definition 1 to types.

Definition 2 *Let $TG = (CLASSES, USE)$ and $A \in CLASSES$ be a non-recursive class whose attributes are private. Classes which are parameter types or return types of methods of A are called the* accessible types *of A. $ACCESSIBLE_A$ denotes the set of accessible types of class A. The set $INTERNAL_A$ of classes that are* internal to A *is the smallest set satisfying*

$$INTERNAL_A = \{B : A \stackrel{+}{\Rightarrow} B \wedge B \notin ACCESSIBLE_A \cup \{A\} \wedge \forall C \in PRED_B : C \in INTERNAL_A \cup \{A\}\}$$

All classes $B \in CLASSES \setminus \{A\} \cup INTERNAL_A$ are external to A.

The following lemma relates type graphs and states.

Lemma 3 *Let $TG = (CLASSES, USE)$ and $s = (OBJ, REF)$ an arbitrary state. Then $x \stackrel{*}{\rightarrow} y$ implies $type(x) \stackrel{*}{\Rightarrow} type(y)$ for every objects $x, y \in OBJ$.*

Proof: It not hard to see that $(x, y) \in REF$ implies $(type(x), type(y)) \in USE$. The claim follows by induction. ■

A class B internal to a class A can be reached only via A:

Lemma 4 *Let $TG = (CLASSES, USE)$ and A be a non-recursive class whose attributes are private, and $INTERNAL_A \neq \emptyset$. Then, for any class $C \in INTERNAL_A$ and every class B such that $B \stackrel{*}{\Rightarrow} C$, one of the following properties hold*

(i) $B \in INTERNAL_A$

(ii) *B is external to A and every path from B to C contains A.*

Proof: Analogous to Lemma 1. ■

Every object y whose type is internal to a class A is either reachable from an object z on the stack whose type is internal to A or an object internal to a sandwich object of class A:

Lemma 5 *Let $TG = (CLASSES, USE)$ and A be a non-recursive class whose attributes are private and $INTERNAL_A \neq \emptyset$. Then, for any state $s = (OBJ, REF, STACK)$ and for every object y with $type(y) \in INTERNAL_A$ there is an object x of type A such that $y \in INTERNAL_x^{(s)}$ or an object $z \in STACK$ of type internal to A such that $z \stackrel{*}{\rightarrow} y$.*

Proof: Suppose there is a $C \in INTERNAL_A$ and an object y of type C such that it is not internal to an object x of type A. Suppose there is an object z with a path π from z to y where $type(z) \notin INTERNAL(A)$ and the type of every $u \in \pi$ is different from A. By Lemma 3 there is path π' from $type(z)$ to $type(y)$ not containing A. This contradicts Lemma 4(ii). Thus, $type(z) \in INTERNAL_A$ for all objects z such that $z \stackrel{*}{\rightarrow} y$. Since $A \notin INTERNAL_A$, by Lemma 3 there is no object of type which is not internal to A that can reach z or x. Since s does not contain unreachable objects, there must be an object $w \in STACK$ of type internal to A such that $w \stackrel{*}{\rightarrow} y$.
■

There is a nesting property analogous to nesting of sandwiches:

Lemma 6 *Let $TG = (CLASSES, USE)$, A be a non-recursive class whose attributes are private, and $INTERNAL_A \neq \emptyset$, and B be a non-recursive class whose attributes are private and $INTERNAL_B \neq \emptyset$. If there is a $C \in ACCESSIBLE_B \cap INTERNAL_A$, then $ACCESSIBLE_A \cap INTERNAL_B = \emptyset$.*

Proof: Analogous to Lemma 2. ■

If a class B is recursive and reachable from a class A, all classes in the strongly connected component of TG are either external or internal to A.

Lemma 7 *Let $TG = (CLASSES, USE)$ be the type graph of a program, A be a non-recursive class where all attributes are private and $INTERNAL_A \neq \emptyset$, and B be a type reachable from A, i.e. $A \stackrel{+}{\Rightarrow} B$. If B is recursive, then either all $C \in \mathfrak{B}$ are external to A or all $C \in \mathfrak{B}$ are internal to A where $\mathfrak{B}$ is the strongly connected component of TG containing B.*

Proof: Suppose there is a class $C \in \mathfrak{B}$ internal to A and a class $C' \in \mathfrak{B}$ external to A. Since $\mathfrak{B}$ is a strongly connected component and $\mathfrak{A}$ is non-recursive, there is a path from $\mathfrak{C}'$ to $\mathfrak{C}$ in TG not containing A. Then, by Lemma 4, A cannot be a non-recursive class satisfying $INTERNAL_A \neq \emptyset$ whose attributes are private. ■

Finally, we show that all objects of classes which have internal classes are sandwich objects:

Theorem 1 *Let $TG = (CLASSES, USE)$ and A be a non-recursive class whose attributes are private, the types of these attributes are not accessible, and $INTERNAL_A \neq \emptyset$. Then, for every state $s = (OBJ, REF, STACK)$, all objects $x \in OBJ$ of type A are either sandwich objects or none of the attributes of x refer to objects.*

Proof: Suppose that there is a state $s = (OBJ, REF, STACK)$ which contains an $x \in OBJ$ of type A that is not a sandwich object and one of the attributes refer to an object. By definition this attribute refers to an object y with $type(y) \in INTERNAL(A)$. Since x is not a sandwich type, there must be an object w with a path π from w to y not containing x. By Lemma 5 there is a sandwich object z of type A in path π such that y is internal to z. But then there cannot be a reference to y from x contradicting our assumption. ■

The algorithm identifies the types satisfying the sufficient condition of Theorem 1. The algorithm computes these types by maintaining a set of candidates. The invariant is that all classes which are not candidates do not satisfy the sufficient condition of Theorem 1. After the last step, every candidate satisfies the sufficient condition of Theorem 1, i.e. they are sandwich types. Furthermore, for every sandwich type A, the set of its internal types is computed. Objects of these types are allocated in the heap associated with sandwich objects of type A.

The algorithm *sandwich_types* performs the following steps:

1. Compute the the type graph $TG = (CLASSES, USE)$ of π. Define *Candidates* to be the set of all classes that have only private attributes.

2. Compute the strongly connected components SCC of TG and the reduced graph, i.e. $RUSE = (SCC, RE)$ where $(\mathfrak{A}, \mathfrak{B}) \in RE$ iff there are $A \in \mathfrak{A}$ and $B \in \mathfrak{B}$ such that $(A, B) \in USE$.

3. Let $Candidates := \{A \in Candidates : A \in \mathfrak{A} \text{for an } \mathfrak{A} \in SCC \text{ with } |\mathfrak{A}| = 1 \wedge (A, A) \notin USE\}$.

4. For every $A \in Candidates$ compute its accessible types $ACCESSIBLE_A$.

5. For every $A \in Candidates$ define and perform the following step (starting with $\mathbb{I}_A = \emptyset$) until I_A does not change:

$$\mathbb{I}_A := \mathbb{I}_A \cup \{B \in CLASSES : B \notin ACCESSIBLE_A \wedge \forall C \in PRED_B : B \in \mathbb{I}_A \cup \{A\}\}.$$

6. Remove every A from *Candidates* where $\mathbb{I}_A = \emptyset$,

7. Declare every $A \in$ *Candidates* to be a sandwich type and define $INTERNAL_A = \mathbb{I}_A$.

Remark: Algorithm *sandwich_types* recognizes all examples of Section 2. ■

The following lemmas explain algorithm *sandwich_types*:

Lemma 8 (Step 1) *After Step 1, all classes $A \in OBJ \setminus Candidates$ violate the assumption of Theorem 1.*

Proof: The assumption of Theorem 1 requires that all attributes of a class are private. ■

Lemma 9 (Step 3) *After Step 3, all classes $A \in OBJ \setminus Candidates$ violate the assumption of Theorem 1.*

Proof: Suppose $(A, A) \in USE$ or there is a strongly connected component $\mathfrak{A} \in SCC$ such that $A \in \mathfrak{A}$. In both cases $A \overset{+}{\Rightarrow} A$, i.e. A is recursive. Hence, the assumptions of Theorem 1 are violated. ■

Lemma 10 (Step 5) *After Step 5, for all classes $A \in Candidates$, $\mathbb{I}_A$ contains the set of all classes internal to A.*

Proof: Step 5 is a closure algorithm in the lattice of sets (ordered by the subset relation) starting with the smallest element. Each step increases the set. Thus, by the fixpoint theorem of Tarski, the smallest set satisfying

$$\mathbb{I}_A = \mathbb{I}_A \cup \{B \in CLASSES : B \notin ACCESSIBLE_A \wedge \forall C \in PRED_B : B \in \mathbb{I}_A \cup \{A\}\}$$

is computed. It is not hard to see that $\mathbb{I}_A$ is also the smallest set satisfying

$$\mathbb{I}_A = \{B \in CLASSES : A \overset{+}{\Rightarrow} B \wedge B \notin ACCESSIBLE_A \wedge \forall C \in PRED_B : B \in \mathbb{I}_A \cup \{A\}\}.$$

Thus, the claim follows by Definition 2. ■

Lemma 11 (Step 6) *After Step 6, for every class $A \in CLASSES$: $A \in Candidates$ iff A is non-recursive, contains only private attributes, and which contains types internal to A.*

Proof: Before Step 6, $A \in$ *Candidates* iff A is non-recursive and contains only private attributes (Lemmas 8 and 9). Thus, after Step 6, $A \in$ *Candidates* iff A is non-recursive, contains only private attributes, and $\mathbb{I}_A$. The claim follows from Lemma 10 since it implies $INTERNAL_A = \mathbb{I}_A$. ■

Theorem 2 (Correctness of Algorithm *sandwich_type***)** *Let π be a program. Every class A declared by Algorithm sandwich_type to be a sandwich type is a sandwich type and $INTERNAL_A$ is the set of its internal classes.*

Proof: Follows directly from Step 7, Lemmas 10 and 11, and Theorem 1 ■

It remains to prove the time complexity of Algorithm *sandwich_type*:

Theorem 3 *Algorithm sandwich_type terminates for every program π in time $O(m \cdot n)$ where n is the size of program π (i.e. number of nodes in the abstract syntax tree of π) and m is the number of classes in the program.*

Proof: Obviously, the type graph TG can be constructed in time $O(n)$ by a traversal through the abstract syntax tree of π. In particular, this implies $|USE| = O(n)$. Step 2 can be performed in $O(|CLASSES| + |USE|) = O(m + n)$ (see e.g. [AHU83, Section 6.7]). While computing the strongly connected components of TG it is possible to mark the classes A such that $(A, A) \notin USE$ and $\{A\} \in SCC$. Thus, Step 3 can be executed in time $O(|Candidates|) = O(m)$. The accessible types of a class A can be computed by a traversal to the abstract syntax tree of class A. Thus, Step 4 can be executed in time $O(n)$. If the sets $\mathbb{I}_A$ are implemented by Bit vectors over the classes, it is sufficient to set in one iteration the Bit for the classes B to true iff $B \notin ACCESSIBLE_A \wedge \forall C \in PRED_B : C \in \mathbb{I}_A$. The initialization costs time $O(m)$ and the test costs times $O(|USE|)$ amortized over all classes. The maximum number of iterations is $O(m)$. Hence, Step 5 can be executed in time $O(m \cdot n)$. The implementation of Step 5 can be extended with additional $O(m)$ execution time such that every class A with $\mathbb{I}_A = \emptyset$ is marked. Thus, the execution time of Step 5 remains $O(m \cdot n)$ and the execution time of Step 6 is $O(m)$. It is not hard to see that the execution time of Step 7 is $O(m^2) = O(m \cdot n)$, because $m \leq n$. ∎

5 Conclusions

We introduced the notion of sandwich types and sandwich object and showed that using object heaps (i.e. one heap per sandwich object) can improve the execution time of object-oriented programs. Furthermore, we showed how to recognize sandwich types in object-oriented programs. Hence, upon creation of a sandwich object, its heap can be created and maintained.

Theorem 1 gives a sufficient condition for sandwich types. It would be useful to extend the assumptions. In particular, the requirement that all attributes are private may be relaxed by defining the type of non-private attributes to be accessible. It would also be useful to relax the definition of internal classes. The problem is that Lemma 4 implies that every class can be internal to at most one class except a nesting property is satisfied (Lemma 6). This excludes, e.g. that the type *LIST_CELL* in our example is used for more than one sandwich type. Hence, the next step in this work is to relax Definition 2 allowing for some classes to be internal for more than one class without nesting. If it would be allowed in general is internal to more than one class, $INTERNAL_A \neq \emptyset$ is not sufficient to imply that A is a sandwich type, because this does not exclude that an object is internal to two different sandwich objects (cf. Figure 4). The key question is: What is the restriction such that $INTERNAL_A \neq \emptyset$ implies that A is a sandwich type?

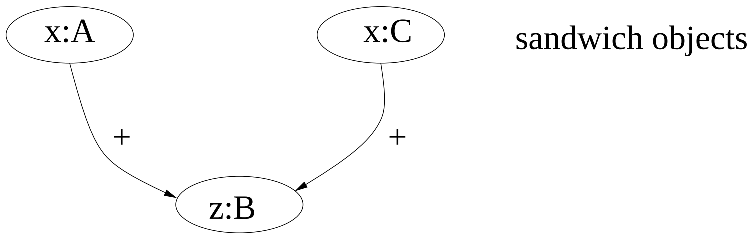

Figure 4: Situation if a class B is internal to different classes

An alias and pointer analysis may lead to additional improvements for the implementation of object heaps. For example, stacks, heaps, and lists may be implemented even more efficiently than sketched in this paper. For example, there is no necessity to have garbage collection on stacks, queues, and double-ended queues, since the elements are inserted and deleted at their ends. When copying a heap of a list object, it can be linearized. This leads to an additional improvement of locality.

Another issue that should be investigated the influence of sandwich types in object-oriented design. It seems natural to use sandwich types when designing object-oriented programs, because it is a way of information hiding. Furthermore, aliasing and sharing of objects can be controlled.

References

[AHU83] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[Alm97] P. S. Almeida. Balloon types: Controlling sharing of state in data types. *Lecture Notes in Computer Science*, 1241:32–59, 1997.

[CLR91] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* McGraw Hill, 1991.

[Fre] Free Software Foundation. The GNU C library.
URL: http://www.gnu.ai.mit.edu/software/libc/libc.html.

[Hor78] J. J. Horning. A case study in language design: Euclid. In F. L. Bauer and M. Broy, editors, *Proceedings of the International Summer School on Program Construction*, volume 69 of *LNCS*, pages 125–132, Marktoberdorf, FRG, July-August 1978. Springer.

[Ing78] Daniel H. H. Ingalls. The smalltalk-76 programming system design and implementation. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona*, pages 9–16. ACM, January 1978.

[Jon96] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Wiley, July 1996. With a chapter on Distributed Garbage Collection by R. Lins. Reprinted February 1997.

[Sta84] James W. Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Transactions on Computer Systems*, 2(2):155–180, May 1984.

[SZ94a] H. W. Schmidt and W. Zimmermann. A complexity calculus for object-oriented programs. *Journal of Object-Oriented Systems*, 1(2):117–147, 1994.

[SZ94b] H. W. Schmidt and W. Zimmermann. Reasoning about complexity of object-oriented programs. In E.-R. Olderog, editor, *Programming Concepts, Methods and Calculi*, volume A–56 of *IFIP Transactions*, pages 553–572, 1994.