Analysis for Object Inlining in Java

Peeter Laud*

FR Informatik
Universität des Saarlandes
laud@cs.uni-sb.de

Abstract

Object-oriented languages allow to encapsulate the data; the unit of the encapsulation is an object. Programmers are advised to reflect the structure of the data while designing the layout of objects. This means that a lot of objects may be created. The semantics of Java dictates that such objects are stored in the heap and are accessed through references. Direct implementation of this would suffer from the overhead of memory management and pointer dereferencing. Object Inlining is an analysis to detect, whether some child objects could be stored together with their parent—the reference from the parent object to the child object would be replaced with the actual data of the child object. We present our analysis for Object Inlining, whose improvement over the previous ones is, that it can also detect, when a child object is replaced with a new one.

1 Introduction and Related Work

Java has a simple, uniform object model—all objects are accessed via references. This simplifies programming, as the programmer is faced with only one possible behaviour of objects. On the other hand, implementing such a model literally—each object is represented as a pointer to the heap, where the object's data resides—generates several kinds of overhead. The most explicit of them is the necessity to dereference a pointer each time an object is accessed. Also, the memory manager has to handle each object separately—the information that a group of related objects is always allocated and deallocated together is not available to it. Handling related objects together may also increase cache performance, if a profitable representation can be chosen for the *group*.

The object-oriented programming paradigm advises programmers to write the code in a way that increases the overhead described above. They are advised to distribute the data across several classes to reflect its structure.

In other languages, most notably in C++, the programmer can manually tune the representation of data in memory by declaring the fields of objects to be either objects or pointers to objects. In this way the program can be made

*Supported by the Esprit LTR project #28198 "JAVA and CoSy Technology for Embedded Systems"

free of the overhead described before, but from the software design point of view, such non-uniform object model is of course inferior to the uniform one.

Our approach is to look for automatic possibilities of *object inlining*—determining, when the type of a field of an object can be changed from "pointer to object" to "object". The best known work in this area is probably that of Dolby and Chien, see [5] and the references it contains. Their approach is to find pairs of objects (r, e) together with a field f of r, where the only operation of storing a reference (i.e. an assignment of the form `x.f = y`), where e stands on the right-hand side, is $r.f = e$; the same operation of storing a reference also has to be the only one where $r.f$ stands on the left-hand side. This constraint ensures that the following program transformation does not change the aliasing information (the sharing patterns). We generalise their analysis by also considering the possibility that several objects $(e_1, \ldots, e_n)$ are inlined into the field f of an object r in succession. Still, the same requirement remains.

In a transformed program it is generally not statically known whether the field `f` in an expression `x.f` is a reference field or an inlined one. Also, the layout of the object pointed to by `x` is not known, hence the offset of the field `f` in this object is not known, either. Thus, if nothing is done about it, all field accesses must be done via a dynamic call of an accessor method. The analysis of Dolby and Chien makes sure that if the class (incl. layout) of the object pointed to by `x` is not known statically, then the field `f` is not inlined. Their cloning framework [7] appears to be well-suited for this. This ensures that dynamically dispatched accessor methods are not needed. Our analysis also handles the same issues (possibly more conservatively, as the cloning framework that we use is less powerful). Our approach differs from Dolby's and Chien's by clearly separating the issues of preserving sharing patterns (Sec. 3) and statically knowing, how to access fields (Sec. 6).

Our analysis proceeds as follows. We start with parts of the analyses that are described in Sec. 3, namely with those that do not require the results of type inference. They are described in the beginning of Sec. 3, in enumeration items 1 and 2. Cloning (Sec. 4), based on the results of these analyses, follows. Next are the type inference [4] and the rest of Sec. 3. This is followed by the data flow analysis described in Sec. 6 and another cloning, based on the results of analyses after the first cloning, except type inference, which is not a bit-vector analysis (and thus cannot be handled by our cloning framework). We finish by creating a constraint system over the values of the predicate "field `f` is inlined at objects created at the `new`-statement s" and the predicates described in Sec. 5 and 6. We feed this system to a generic constraint solver, together with the objective to inline as many fields as possible.

Our contributions are the semantic model of the heap and the analysis based on it, which are presented in Sec. 2 and 3. Also, we believe that our way of coinductively stating when the field accesses do not need dynamic dispatch, can be superior to the methods of Dolby and Chien, when one considers complex recursive data structures.

2 Semantics of the Memory

The memory at a certain program point consists of a set of variables **Var**, containing both global variables and local variables of all methods in the execution stack, and a heap $\mathcal{H}$. The heap is a set (with no further structure) of objects. There is also a set of fields **Fld**; each object is assumed to have all fields in it. There is a function

$$\mathbf{Ref} : \big(\mathbf{Var} \uplus (\mathcal{H} \times \mathbf{Fld})\big) \to \mathcal{H}_\perp,$$

where $\uplus$ denotes disjoint union and $\mathcal{H}_\perp := \mathcal{H} \uplus \{\perp\}$. This function describes, to which objects the variables and object fields point. *NULL* is denoted by $\perp$. We assume that the values of all fields and variables are references; we can overlook the atomic values for our purposes.

There is a predicate inl over $\mathcal{H} \times \mathbf{Fld}$, which says whether a particular field in a particular object is inlined or not. We require that if a field f of an object o is inlined (i.e. $\mathsf{inl}(o.f)$ holds) and $\mathbf{Ref}(o.f) \neq \perp$, then there exists no other field f' of an object o', such that $\mathbf{Ref}(o'.f') = \mathbf{Ref}(o.f)$ AND $\mathsf{inl}(o'.f')$ HOLDS.

We are going to define the *sharing pattern* of the memory, which must be preserved by program transformations. A pointer chain pc is either a variable in **Var** or a pointer chain followed by a field. Define a partial function $\mathcal{E}$ from pointer chains to objects in $\mathcal{H}$ as follows:

$$\begin{aligned} \mathcal{E}(v) &= \mathbf{Ref}(v), && \text{if } v \in \mathbf{Var} \\ \mathcal{E}(\mathsf{pc} \cdot f) &= \mathbf{Ref}(\mathcal{E}(\mathsf{pc}).f), && \text{otherwise.} \end{aligned}$$

We call the relation Ker $\mathcal{E}$ the sharing pattern of the memory $(\mathbf{Var}, \mathcal{H}, \mathbf{Ref})$ (this relation is actually the same as the *alias relation* of Deutsch [3]).

We assume to have the following statements in our programming language: `x = new(f1,...,fk)`, `x = y`, `x = y.f` (we call these statements "field loads"), `x.f = y` (we call these statements "field stores"), `x.f := y` (we call these statements "deep copies") and statements for controlling control flow. The arguments of a `new`-statement are fields that will be inlined in the newly generated object. The semantics of `new`-statements and simple variable assignments is fully intuitive. The statement `x.f = y` denotes making the reference field `x.f` point to the object pointed to by `y`; it semantics is intuitively clear, too. The statement `x.f := y` denotes making the inlined field `x.f` reference the object pointed to by `y`. It works as follows: If the value of the field `f` of the object pointed to by `x` (i.e. the quantity $\mathbf{Ref}(\mathbf{Ref}(\mathtt{x}).\mathtt{f})$) was *NULL* before the assignment, then a new object is created (with the same inlined fields as $\mathbf{Ref}(\mathtt{y})$) and `x.f` is set to reference it. Then, in any case, `x.f := y` makes a *deep copy* of the object pointed to by `y`—the value of `y.g` will be assigned to `x.f.g` for all $\mathtt{g} \in \mathbf{Fld}$. This is recursive with respect to the inlined fields of `y` (which must be the same as the inlined fields of `x.f`).

3 Heap Analysis

We are given a program P that works with objects that have no inlined fields. We want to have a transformation that makes as many fields inlined as possible. More precisely, we are looking for a transformation that

- only changes the `new`-statements in the program (by marking some fields inlined) and turns some field stores to deep copies;

- is correct in a sense, that the resulting program P' has the same sharing patterns as P at each program point, if we only consider live pointer chains.

Assume w.l.o.g. that the program P is such, that after each statement `x.f = y`, the variable `y` is dead. This can easily be ensured by following this statement by `y = x.f`.

The main constraint for inlined fields was, that no other field may point to the object that this field points to. The crucial point for the analysis is therefore finding out, the references of which objects may be stored in the fields of other objects several times. Also, for each field store we must know whether we change it to a deep copy or keep it as is. The analysis is as follows:

1. For each field store `x.f = y` find out, whether `y` may have flown from a field load. If this is the case, then this field store may not be turned to a deep copy. Also, the field `f` of objects that `x` may point at (and which are found by type inference [4]), cannot be inlined. This is done by a simple forward data flow analysis, which assigns to each variable at each program point either "not from a field load" or "may be from a field load".

2. For each field store `x.f = y` find out, whether `y` may have flown from a `new`-statement, and if this is the case, then find out whether `y` may have live aliases at this program point. If this is the case, then this field store cannot be turned to a deep copy, as this could change the fact, whether `x.f` and the alias of `y` point to the same object or not. This is done by a simple may-alias analysis that considers only variables, and not all pointer chains, see [6, Sec 4.2.3]. The results of this analysis have to be combined with the liveness analysis. Again, the field `f` of objects potentially pointed to by `x`, cannot be inlined.

3. For each field store `x.f = y` find out, whether the field `f` of all objects that `x` can point to, may be inlined. If this is not the case, then this field store cannot be turned to deep copy and the field `f` of none of the objects that `x` can point to, may be inlined. We have to iterate this marking of field stores that cannot be deep copies, and fields that cannot be inlined, until a fixpoint has been reached.

The analyses considered so far ensure that at a deep copy `x.f := y`, the right hand side has flown directly from a `new`-statement and that after this statement, the only way to access the object pointed to by `y`, is through `x.f`. We also have to ensure that the object $\mathbf{Ref}(\mathbf{Ref}(\mathtt{x}).\mathtt{f})$ is dead before a field store `x.f = y`, if we want to turn it into a deep copy. This amounts to determining, whether for each field load `w = z.f`

- `x` and `z` may be referring to the same object;
- `w` or variables, where it has flown, are alive at the statement `x.f = y`.

This is done by a forward data flow analysis that assigns to each variable at each program point a set of pairs $\langle$`new`-statement, field$\rangle$. For `w` at the statement `w = z.f`, the first components of the pairs would be the possible creation points of the object that is pointed to by `z`, and the second component of the pairs would be `f`.

We have thus found out, which objects cannot be put into inlined fields and which field stores can be deep copies and which ones cannot. From this information we can easily deduce, which fields of the objects at which creation points can be inlined fields and which cannot. The resulting transformation satisfies the conditions presented at the beginning of this section.

There exists a simple extension of the analysis and transformation, that does not preserve sharing patterns, but is obviously allowed. It concerns "constant objects"—objects, that are modified only at the beginning of their lifetime, i.e. that are initialised first and afterwards only read. The "beginning of lifetime" runs from the creation of the object until the object has been stored in a field of some other object. Obviously, one can create several copies of the same object later, because these copies cannot get "out of sync", as they are not modified at all. This increases inlining possibilities. Extra care has to be taken in transforming the comparisons of references that may point to constant objects, though. As they may point to different copies of the same object afterwards, an extra "identity" field should be added to such objects.

4 Cloning

Interprocedural data flow analyses use interprocedural control flow graph which, in itself, does not reflect that after the end of each method the control returns to the point where this method was called from. In general, this causes propagating the analysis information from some call-edge coming to the method, to a return-edge returning to some other call-site. This can cause the analysis to lose precision. Code reuse, promoted by the object-oriented programming paradigm, magnifies this problem.

A technique to avoid the propagation of information over invalid paths is *effect calculation* a.k.a. the functional approach to the interproc. DFA by Sharir and Pnueli. If the data flow analysis $\mathcal{A}$ assigns an element $\mathcal{A}(N)$ of an upper semilattice $\mathcal{L}$ to each program point N, then we transform the analysis in a way, that for each possible call context C of some method, the analysis assigns a value $\mathcal{A}_C(N)$ to the program point N inside that method. Additionally, we just let the call contexts to be all possible analysis results for the start node of the method, i.e. we let the set of analysis contexts be $\mathcal{L}$. In principle, this amounts to replacing $\mathcal{L}$ with $\mathcal{L} \to \mathcal{L}$.

Generally, effect calculation has a high price tag. It may greatly increase the complexity of the analysis or even make it nonterminating. For bit-vector analyses, the effect calculation can be made quite cheaply, however. In the case of a bit-vector analysis, the semilattice $\mathcal{L}$ has a quite small set of generators $\mathcal{B}$—the set of all vectors where a single bit is set. Also, the result of effect calculation $\mathcal{A} : \mathcal{L} \to \mathcal{L}$ is an upper semilattice homomorphism. This means that its values are uniquely determined by its values on some set of generators. Thus we only have to save the analysis results for all possible call contexts from the set $\mathcal{B}$ for doing the effect calculation. All analyses described in the previous section, except type inference, are bit-vector analyses.

Suppose that we use the analysis $\mathcal{A}$ for determining whether a particular transformation is valid. Also suppose that we have determined, that it is valid for some calling contexts $\mathcal{B}_{\text{good}}$ and invalid for contexts $\mathcal{B}_{\text{bad}}$. We can still do the transformation if we first create a clone of the current method and then redirect all calls to the method, that create bad calling contexts, to this clone.

In this way only calling contexts from $\mathcal{B}_{\text{good}}$ remain and the transformation can be done. If a call site generates both good and bad contexts, then we may attempt to clone the method that contains this call site.

5 Representing Objects in Memory

An object is represented in the memory of a computer as a sequence of fields. Thus, at `new`-statements, the layout of the object has to be fixed. For each field we have to fix its offset. If a field is inlined, then we also need to know how much room to leave for the objects that can be copied into it (the size of a reference field is fixed—it is the length of an address). For objects `o` and fields `f` we have an equation system to determine their sizes:

$$size(\mathtt{f}) = \begin{cases} \text{predefined}, & \mathtt{f} \text{ is reference (and also if it is atomic)} \\ \max\limits_{\mathtt{o} \in \mathit{Creations}(\mathtt{f})} size(\mathtt{o}), & \mathtt{f} \text{ is inlined} \end{cases}$$

$$size(\mathtt{o}) = \sum_{\mathtt{f} \in \mathsf{fields}(\mathtt{o})} size(\mathtt{f}) + \langle\text{administrative overhead}\rangle,$$

where $\mathsf{fields}(\mathtt{o}) \subseteq \mathbf{Fld}$ is the set of fields that are actually used in `o`; $\mathit{Creations}(\mathtt{f})$ is the set of objects (actually the set of creation points of objects), that `f` may point to; administrative overhead may contain information about object's runtime type, may include necessary attributes for memory management etc.

The constraint that follows directly from this system of equations is: No object may be inlined into its own field (not even indirectly). I.e. if $c_1, \ldots, c_n = c_0$ are creation points and $f_0, \ldots, f_{n-1}$ are fields and $c_{i+1} \in \mathit{Creations}(c_i.f_i)$ for all i between 0 and $(n-1)$, then at least one of the values $\mathsf{inl}(c_i.f_i)$ must be false.

Another constraint is caused by the type system of Java—each field of each class must have a fixed type. Thus, if the objects created at several different creation points may end up inlined into the same field, then these objects must be compatible enough to fit into the type system. The predicate $\mathsf{coinlinable}(c_1, c_2)$ describes whether objects created at the creation points c_1 and c_2 may be inlined into the same field.

$$\begin{aligned} \mathsf{coinlinable}(c_1, c_2) :\Leftrightarrow\ & \mathit{Class}(c_1) = \mathit{Class}(c_2) \wedge \\ & \forall \mathtt{f} \in \mathsf{fields}(c_1) : \Big(\big(\mathsf{inl}(c_1.\mathtt{f}) \Leftrightarrow \mathsf{inl}(c_2.\mathtt{f})\big) \wedge \big(\mathsf{inl}(c_1.\mathtt{f}) \Rightarrow \\ & \forall c_1' \in \mathit{Creations}(c_1.\mathtt{f})\, \forall c_2' \in \mathit{Creations}(c_2.\mathtt{f}) : \mathsf{coinlinable}(c_1', c_2')\big)\Big), \end{aligned} \tag{1}$$

where $\mathit{Class}(c)$ is the actual class that is named in the `new`-statement at the program point c. We see that $\mathsf{coinlinable}$ is defined recursively. To make this definition a correct one, we state that $\mathsf{coinlinable}$ is defined *coinductively*—it is the biggest relation satisfying (1) (for the given predicate inl).

We require that for all creation points c, for all fields $\mathtt{f} \in \mathsf{fields}(c)$ and all creation points $c_1, c_2 \in \mathit{Creations}(c.\mathtt{f})$ the formula $\mathsf{inl}(c.\mathtt{f}) \Rightarrow \mathsf{coinlinable}(c_1, c_2)$ holds.

statement	transfer function
`x = new`$_c$ `C`	$F_\bullet = F_\circ[\mathtt{x} \mapsto \{c\}]$
`x = y`	$F_\bullet = F_\circ[\mathtt{x} \mapsto F_\circ(\mathtt{y})]$
`x = y.f`	$F_\bullet = F_\circ[\mathtt{x} \mapsto \bigcup_{t \in F_\circ(\mathtt{y})} \mathit{MakeTag}(t, \mathtt{f})]$
`x.f = y`	$F_\bullet = F_\circ$

$$\mathit{MakeTag}(\langle c_1 \mathtt{f}_1 \cdots c_n \rangle, \mathtt{f}_n) = \{\langle c_1 \mathtt{f}_1 \cdots c_n \mathtt{f}_n c_{n+1} \rangle \mid c_{n+1} \in \mathit{Creations}(c_n.\mathtt{f}_n)\}$$

Table 1: Analysis of accesses of inlined fields

6 Accessing the Fields of Objects Uniformly

We already mentioned in the introduction, that we do not want to use dynamically dispatched accessor methods to access the fields of objects. This requires the following:

- All objects that a reference variable may point to at a certain program point, must be compatible enough.
- If a reference variable at a certain program point *may* have been defined by a field load `x = y.f`, where `f` was an inlined field, then it *must* have been defined by such a field load. Our program transformation will be such, that a statement `x = y.f`, where `f` is an inlined field, will be changed to `x = y` and it will be statically remembered, that `x` actually points to the inlined field `f`. Such static remembering is impossible, if it could have been some other inlined field or no inlined field at all.

The required compatibility is the following: $\mathsf{copointable}(c_1, c_2, C)$ describes, whether objects created at the creation points c_1 and c_2 can be accessed through the same pointer with type "reference to an object of class C".

$$\begin{aligned}\mathsf{copointable}(c_1, c_2, C) :\Leftrightarrow\ & \mathit{Class}(c_1), \mathit{Class}(c_2) \leq C \wedge \\ & \forall \mathtt{f} \in \mathsf{fields}(C) : \Big(\big(\mathsf{inl}(c_1, \mathtt{f}) \Leftrightarrow \mathsf{inl}(c_2, \mathtt{f})\big) \wedge \\ \forall c_1' \in \mathit{Creations}(c_1.\mathtt{f})\, \forall c_2' \in \mathit{Creations}(c_2.\mathtt{f}) &: \mathsf{copointable}(c_1', c_2', \mathit{Class}(C.\mathtt{f})\Big),\end{aligned}$$

which is again defined coinductively. For all creation points c, fields $\mathtt{f} \in \mathsf{fields}(c)$ and creation points $c_1, c_2 \in \mathit{Creations}(c.\mathtt{f})$ we require that the statement $\mathsf{copointable}(c_1, c_2, \mathit{Class}(c.\mathtt{f}))$ holds.

For the second issue we record, from which fields of which objects the reference variables have been loaded. This part is similar to [5]. Let **Tag** be the set of strings $\langle c_1 \mathtt{f}_1 \cdots c_n \mathtt{f}_n c_{n+1} \rangle$, where $n \geq 0$ and for each i between 1 and n we have $c_{i+1} \in \mathit{Creations}(c_i.\mathtt{f}_i)$. The analysis associates each reference variable at each program point with a set of tags. Let $F_\circ$ and $F_\bullet$ denote the analysis result before and after some statement, respectively. $F_\circ$ and $F_\bullet$ are functions mapping variables to sets of tags. The transfer functions are depicted at Table 1.

The elements of **Tag** can be of any length, thus the described analysis may be nonterminating. The set **Tag** can be made finite, if we take into account that we are only interested in such tags $\langle c_1 \mathtt{f}_1 \cdots c_n \mathtt{f}_n c_{n+1} \rangle$, where $\mathsf{inl}(c_i.\mathtt{f}_i)$ may hold for all i between 1 and n. Thus, if we already know from the analysis presented in Sec. 3, that some $\mathsf{inl}(c_i.\mathtt{f}_i)$ does not hold, then we may identify this tag with $\langle c_{i+1}\mathtt{f}_{i+1}\cdots c_n\mathtt{f}_n c_{n+1}\rangle$. Also, we had the requirement that an object may not be inlined into a field of itself. Hence, if $c_i = c_{n+1}$, then this tag may be identified with $\langle c_{i+1}\mathtt{f}_{i+1}\cdots c_n\mathtt{f}_n c_{n+1}\rangle$. The last identification makes the set **Tag** effectively finite and the analysis terminating.

We require that all tags for some variable at some program point have the same length and contain the same fields in the same order. If this is not the case, then we have to not inline some fields.

We have to pick the "best" value for the predicate inl, that satisfies all the requirements presented above. Thus we have got an optimisation problem. The simplest definition of the objective function of the problem (i.e. the definition of "best") would just be the number of the fields that can be inlined. With help of some profiling information, one may be able to derive other, more advanced, objective functions. Still, we do not believe that the different maximal values of inl that satisfy all presented requirements, are very different, thus such simplistic objective function may be good enough.

Acknowledgements

We are thankful to Bruno Blanchet for helpful discussions. The described analysis has been implemented in the CoSy compiler construction framework [1]. Several of its component analyses have been implemented with the program analyser generator PAG [2].

References

[1] M. Alt, U. Aßmann, H. van Someren. Cosy Compiler Phase Embedding with the CoSy Compiler Model. In *5th Int. Conf. on Compiler Construction*, 1994.

[2] M. Alt, F. Martin. Generation of Efficient Interprocedural Analyzers with PAG. In *Static Analysis Symposium*, 1995.

[3] A. Deutsch. A Storeless Model of Aliasing and its Abstractions using Finite Representations of Right-Regular Equivalence Relations. In *IEEE 1992 Int. Conf. on Computer Languages*, San Francisco, April 1992.

[4] H. Dewes, C. Probst. Static Method Call in Java. Submitted to *JOSES: Java Optimization Strategies for Embedded Systems*, 2001.

[5] J. Dolby, A. Chien. An Automatic Object Inlining Optimization and its Evaluation. In *Programming Language Design and Implementation*, 2000.

[6] W. Landi. Interprocedural Aliasing in the Presence of Pointers. PhD thesis, Rutgers University, 1992.

[7] J. Plevyak. Optimization of Object-Oriented and Concurrent Programs. PhD thesis, University of Illinois at Urbana-Champaign, 1996.