FACULTAD DE INFORMATICA

UNIVERSIDAD POLITECNICA DE MADRID

TRABAJO FIN DE CARRERA

MASTER UNIVERSITARIO
EN SOFTWARE Y SISTEMAS

VALUATION: DEVELOPER SUPPORT FOR BY-REFERENCE TO BY-VALUE TYPE CONVERSION

AUTOR: Shiva Shabaninejad

ADVISOR: Manuel Carro Linares

CO-ADVISOR: Mark Marron

July 2012

Contents

Chapter 1

Abstract

1.0.1 Abstracto

Los lenguajes modernos orientados a objetos, como C# y Java, permiten que los programadores desarrollen aplicaciones complejas con ms rpidez. Esos lenguajes estn basados en la definicin de estructuras de datos mediante el uso de objetos en el heap que se pasan por referencia. Esto simplifica la programacin al tener reserva y liberacin automtica de memoria, a la vez que recogida automtica de basura.

Esta simplificacin de la programacin tiene un coste en eficiencia. El uso de objetos pasados por referencia en lugar de los ms livianos pasados por valor puede tener un efecto negativo en la memoria en algunos casos. Ese coste puede ser crtico cuando los programas se ejecutan en entornos con recursos limitados, como dispositivos mviles y systemas de ”cloud computing”.

Esta tesis explora el problema mediante el uso de un modelo de memoria simple y uniforme para mejorar la eficiencia. En el presente trabajo atacamos este problema proporcionando un anlisis esttico automtico y correcto que identifica si un tipo ”por referencia” puede ser convertido en un tipo ”por valor” donde la conversin puede traer mejoras de eficiencia. Nos centramos en programas C# y nos basamos en una combinacin de comprobaciones sintcticas y semnticas para identificar clases cuya conversin es segura.

La eficacia de este trabajo se evala con la identificacin de los tipos convertibles y el impacto de la transformacin propuesta. El resultado muestra que la transformacin de tipos por referencia a tipos por valor puede tener un efecto sustancial en la eficiencia. En nuestros casos de estudio, optimizamos la eficiencia de una simulacin de Barnes-Hut, consiguiendo una reduccin del 89% en la reduccin total de la memoria reservada y una reduccin del tiempo de ejecucin de 8%.

1.0.2 Abstract

Modern object oriented languages like C# and JAVA enable developers to build complex application in less time. These languages are based on selecting heap allocated pass-by-reference

objects for user defined data structures. This simplifies programming by automatically managing memory allocation and deallocation in conjunction with automated garbage collection. This simplification of programming comes at the cost of performance. Using pass-by-reference objects instead of lighter weight pass-by value structs can have memory impact in some cases. These costs can be critical when these application runs on limited resource environments such as mobile devices and cloud computing systems. We explore the problem by using the simple and uniform memory model to improve the performance. In this work we address this problem by providing an automated and sounds static conversion analysis which identifies if a by reference type can be safely converted to a by value type where the conversion may result in performance improvements. This works focus on C# programs. Our approach is based on a combination of syntactic and semantic checks to identify classes that are safe to convert. We evaluate the effectiveness of our work in identifying convertible types and impact of this transformation. The result shows that the transformation of reference type to value type can have substantial performance impact in practice. In our case studies we optimize the performance in Barnes-Hut program which shows total memory allocation decreased by 93% and execution time also reduced by 15%.

Chapter 2

Introduction

Modern object-oriented languages, such as C# and Java, provide a host of features that enable developers to build complex applications in less time and with fewer errors than is possible with languages such as C/C++. One of the most important of these features is the emphasis on a simplified and uniform memory model that is exposed to the developer. Modern object-oriented languages are based on selecting heap allocated pass-by-reference objects as the primary (or only) form of user defined aggregate concepts. This uniformity greatly simplifies the conceptual model for the behavior of a program that the developer must consider. In conjunction with automated garbage collection, this simplification eliminates entire classes of possible program errors while allowing the developer (architect) to focus on the desired behavior of the program instead of the details of storage location issues, lifetime management, and assignment semantics in the program.

This simplification of the semantics of the memory model, and associated simplification of the development process, comes at the potential cost of program performance. In some cases the use of pass-by-reference *objects* instead of lighter weight pass-by-value stack *structs* can have a non-trivial performance impact. The performance impacts are usually seen in as increased runtime, increases in the total amount of memory allocated, and increases in the maximum live memory space needed by the program. In general these types of issues have been seen as an acceptable (if unfortunate) cost to pay for improved reliability and programmer productivity when developing desktop and server applications where memory, computational resources, and power are all abundant resources. In these scenarios the extra memory used, along with computational and power cost of manipulating/managing this memory, is negligible in comparison to the value gained by eliminating large classes of common program errors. However, these costs can be critical when developing applications for mobile devices or in cloud computing scenarios. In the case of mobile computing (smart phones, tablets, etc.) there are hard limits on the amount of memory available, processor speeds, and energy which can be consumed. For cloud computing power is a major concern and the cost of hosting the

application is proportional to the processor and memory resources required.

In this work we explore the problem of how to maintain the improvements in developer productivity and program quality that are obtained by using the simple and uniform memory model while minimizing the performance impacts that can result. Our approach is designed around the desire to avoid premature optimization "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil" [15]. Thus, we want to enable the developer to do the majority of the implementation using a simple memory model, focusing on translating the application domain into a meaningful set of classes, and then when needed to optimize the specific memory representations that are actually performance critical. In practice this approach is problematic as identifying all the locations where the behavior of a pass-by-reference class may differ from the behavior of a pass-by-value struct is time consuming and error prone. In this work we address this problem by providing an automated and *sound* static conversion analysis which identifies if a by-reference type can be *safely* converted to a by-value type without altering the semantics of the program. Further, in the case where a type cannot be safely converted the analysis can flag the specific features/locations in the program where the semantics before/after the conversion may differ and recommends possible resolutions for these problems to the developer.

The approach presented in this work is based on a combination of *syntactic* and *semantic* checks. The syntactic checks identify any possible violations of any language restrictions on by-value types that may be violated by the conversion. The syntactic checks then identify possible uses of an instance of a by-reference type which would have visibly different semantics after conversion to a by-value type. A central issue in determining if a conversion may alter the observable semantics of the program is in the modification of object fields. A simple approach is to only allow the conversion of immutable objects – since if an object is immutable then trivially creating multiple copies of it by converting to by-value semantics will not alter the observable semantics of the program. However, this is a substantial restriction and greatly limits the opportunities for conversion. Thus, we introduce the concept of *single-observer* writes where at the time of the modification we know that there is a *unique* base location that can observe the effect of the write.

The work in this paper focuses on the conversion in C# programs, which provide language level support for by-value structs. Thus, in many cases the conversion can be as simple as changing the `class` keyword to `struct`[1]. We demonstrate the effectiveness of the analysis on a suite of C# applications. The experimental results show that the approach is useful in automatically identifying by-reference class types which can be safely converted to by-value struct types and, in the cases where conversion may not be safe, provides the developer with a detailed

[1]The same effect can be achieved, although in a less satisfactory manner, by manually inlining definitions and passing multiple parameters to methods in Java programs.

summary of possible problems. Using a profiler we identify high allocation types and utilize the conversion reports provided to refactor high allocation rate class types into structs. The resulting programs show up to a 15% reduction in runtime and 93% reduction in total memory allocation.

This paper makes the following contributions:

Problem Formalization Despite the intuitive nature of the struct conversion problem, and the frequent manual application by developers, this work provides a novel formalization of conditions under which such a transformation can be considered safe. This formalization includes both syntactic and semantic constraints which are sufficient to ensure that the transformation preserves the behavior of the original program.

Algorithms This paper presents two algorithms for performing the safety analysis for conversion to structs. The simple algorithm, based on immutability, can be implemented in existing compilers using only minor extensions to standard points-to analyses. We also present a more powerful algorithm which requires the use of a more powerful heap analysis algorithm chapter 5 but which can identify substantially more conversions as safe.

Warnings The safety analysis algorithms defined in this paper are also capable of diagnosing why a conversion may be unsafe and reporting a set of possible actions a developer could take to make the desired conversion safe.

Evaluation We evaluate the effectiveness of both the safety analysis algorithm in identifying types that can be converted and the impact of these transformations in the overall memory use and performance of programs in our benchmark suite. These results show that the transformation of by-reference types to by-value types can have a substantial performance impact in practice (in some program a 93% memory and 15% time reduction). The results also show that the transformation safety analysis algorithm is able to automatically identify many types as safe to convert in practice and, given types identified as *important* via profiling, can quickly aid a developer in performing any manual code modifications to safely perform the desired conversion to a value type.

Chapter 3

Formal Problem and Motivating Example

In this section we formalize the problem of conversion by-reference objects into by-value structs in a C# program and introduce a running example that we use to motivate and illustrate various concepts throughout the paper.

3.0.1 Formal Problem

The state of a concrete program is modeled in a standard way where there is an environment, mapping variables to addresses, and a store, mapping addresses to values. We define the state of a program in the usual way based on an environment together with a store and a set of objects as a *concrete program state*. Given a program that defines a set of types, Types, and a set of fields (and array indices), Labels, a concrete program state is a tuple $(\mathsf{Env}, \sigma, \mathsf{Ob})$ where:

$$
\begin{aligned}
\mathsf{Env} \in \text{Environment} &= \mathsf{Vars} \rightharpoonup \mathsf{Addresses} \\
\sigma \in \text{Store} &= \mathsf{Addresses} \to \mathsf{Objects} \cup \{\mathsf{null}\} \cup \mathsf{Values} \\
\mathsf{Ob} &\in 2^{\mathrm{Objects}} \\
\mathsf{Values} &= \mathbb{N} \cup \mathrm{Structs} \\
\mathrm{Structs} &= \mathsf{Types} \times (\mathsf{Labels} \rightharpoonup \mathsf{Addresses}) \\
\mathrm{Objects} &= \mathsf{OID} \times \mathsf{Types} \times (\mathsf{Labels} \rightharpoonup \mathsf{Addresses}) \\
&\qquad \text{where the object identifier set } \mathsf{OID} = \mathbb{N}
\end{aligned}
$$

In order to focus on the core issues we assume that the set of Values in the program is limited to integers and user declared by value structs. The Structs are tuples of a type and a map from field labels to concrete addresses for the fields. Each object o in the set Ob is a tuple consisting of a unique identifier for the object, the type of the object, and a map from field labels to concrete addresses for the fields defined in the object. We use the notation $\mathsf{Ty}(o)$ to refer to the type of an object. The notation $o.l$ refers to the value of the field (or array index) l in the

object. It is also useful to refer to a *non-null pointer* as a specific structure in a number of definitions. Therefore, we define a *non-null pointer p* associated with an object o and a label as l in a specific concrete heap, $(\mathsf{Env}, \sigma, \mathsf{Ob})$, as $p = (o, l, \sigma(o.l))$ where $\sigma(o.l) \neq \mathsf{null}$. We define a helper function $\mathsf{Fld} : \mathsf{Types} \mapsto 2^{\mathsf{Labels}}$ to get the set of all fields (or array indices) that are defined for a given type.

As the goal of this work is to transform the uses of a given by-reference type (object) to uses of a by-value type (struct) without changing the observed behavior of the program we now formalize what it means for two programs to have the same observational behavior. This behavior can be formalized by looking at the observable state of the program which we define based on the value of all primitive paths in the program state.

Definition 1 (Primitive Path)**.** *Given a program state* $s = (\mathsf{Env}, \sigma, \mathsf{Ob})$ *an access path,* A_p*, in this program state is a initial variable and sequence of fields* $(v, \langle f_1, \ldots, f_k \rangle)$ *where each* $f_i \in \mathsf{Labels}$*. The value,* $V(A_p, s)$*, of a path is given by the repeated lookup of addresses in the store* $\sigma((\ldots\sigma(\sigma(\mathsf{Env}(v)).f_1)\ldots).f_k)$

- *An access path* invalid *if any step in this lookup is undefined or applied on a non-address value (integer, null, etc.).*
- *An access path is* primitive *if the resulting value,* $V(A_p, s)$*, is an integer.*

Definition 2 (Equivalent Program State)**.** *Given two program states* $s_1 = (\mathsf{Env}_1, \sigma_1, \mathsf{Ob}_1)$ *and* $s_2 = (\mathsf{Env}_2, \sigma_2, \mathsf{Ob}_2)$ *we say these program states are* indistinguishable *if for all* primitive *access paths* $A_p = (v, \langle f_1, \ldots, f_k \rangle)$ *in* s_1 *then both* A_p *is* primitive *in* s_2 *and* $V(A_p, s_1) = V(A_p, s_2)$

Given these definitions we can check that, despite any differences in the organization of memory, any pure expression will have the same valuation in both states. If we have a set of methods that are observable, *e.g.* in C# this methods in the `System.IO` namespace which interact with the file system would be observable, then as long as at every call to any of these methods the program state before and after our value type transformations are *equivalent* then the two programs are observationally equivalent as well. While this definition allows for a large amount of flexibility in how we restructure the program this condition provides more flexibility than needed here. Thus we opt for the simpler, although more restrictive condition, and consider a program before transformation to value types *observationally equivalent* to the program after the transformation when the program states are equivalent at every basic block entry in the control-flow graph of the program.

Definition 3 (Observational Program Equivalence)**.** *Given a program* P_r *which uses by-reference types* $\{r_1, \ldots, r_k\}$ *and* P_v *which is identical to* P_r *except for the replacement of the by-reference types with by-value types* $\{v_1, \ldots, v_k\}$*. We define* P_r *as observationally equivalent to* P_v *if when*

```
public class MathVector
{
    public static readonly int NDIM = 3;
    private double[] data;

}
```

Figure 3.1: Original version of MathVector class

```
public struct MathVector
{
    private double data0;
    private double data1;
    private double data2;

}
```

Figure 3.2: Here we changed the MathVector and its members to Value type

executing them in lockstep then during the execution at every basic block entry the program state of P_r is equivalent to the program state of P_v.

3.0.2 Motivating Example – Barnes-Hut N-Body Simulation

BH is a program that performs an n-body simulation. This program implements the BarnesHut simulation algorithm which is a well-known method for reducing the cost of computing the force interaction of n bodies from order $O(n^2)$ to $O(n\ log\ n)$.

In a three dimensional space the bodies are divided to form of tree. Each node of tree represents a part of space and contains sub group of bodies. The root of the tree represents the entire space. Moving down the tree splits nodes into smaller sub-spaces until the leaf-nodes which each contain 0 or1 body. Our C implementation of this program contains a `MathVector` class which represents a three dimensional vector. This class implements standard operations, addition, scalar multiplication, dot product, etc., and uses array of double represents the vectors data. The performance of this program can be improved by converting reference types to value types. As Figure 3.2 shows we changed the class `MathVector` in two steps: First we changed the type of class `MathVector` to a `struct`. Second, because we knew that the data array represents a three dimensional space we replaced an array of double with three members of double type (`data0`, `data2`, and `data2`).

Using our analyzer, we can create a diagram which represents a snapshot of the abstract heap chapter 5 at the point of maximum memory consumption during program execution. In this diagram each node represents a reference type (object) and each edge represents a set

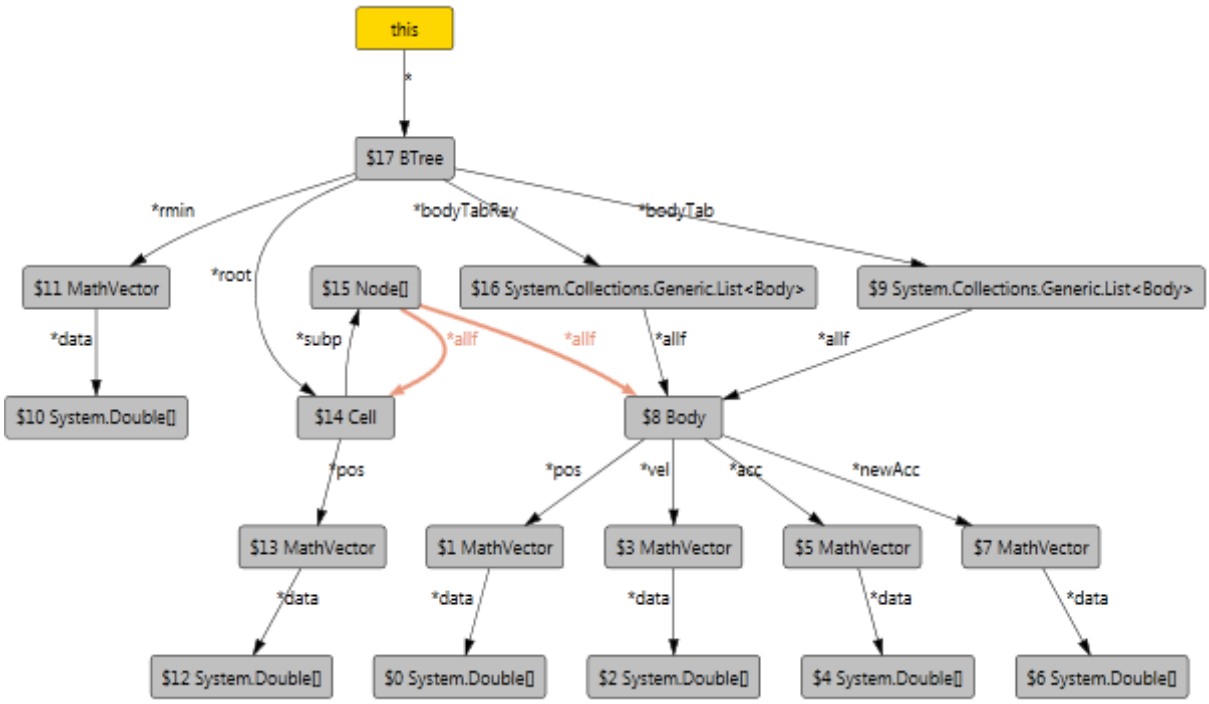


Figure 3.3: Abstract heap diagram of Barnes-Hut

of pointers. Each node can have multiple edges coming out of the node which represent the pointers stored in the member fields of that class type. In the BH program the point of maximum live objects is inside the method `stepSystem` in the class `BTree`. Figure 3.3 shows local heap when `MathVector` is defined as reference type (`class`) while Figure 3.4 shows local heap of the same state of the program when we define `MathVector` as a value type (`struct`).

Looking at the `Body` node we see that it points to four nodes of type `MathVector`. Figure 3.5 shows the definition of class `Body` which has three members of type `MathVector` (the fourth is defined in the parent `Node` class). If we convert the `MathVector` type to a struct then the shape of heap would be the same as if we explicitly defined all members of `MathVector` inside the definition of the `Body` class. This can be seen in Figure 3.6. In addition to reducing the memory overhead of representing the `MathVector` objects and the `double[]` this transformation reduces the number of memory dereferences needed to access the `Mathvectors` data by two.

We evaluate the performance impact of the transformation of this program as one of our case studies and the result showed 93% reduction in total memory usage and 15% reduction in execution time.

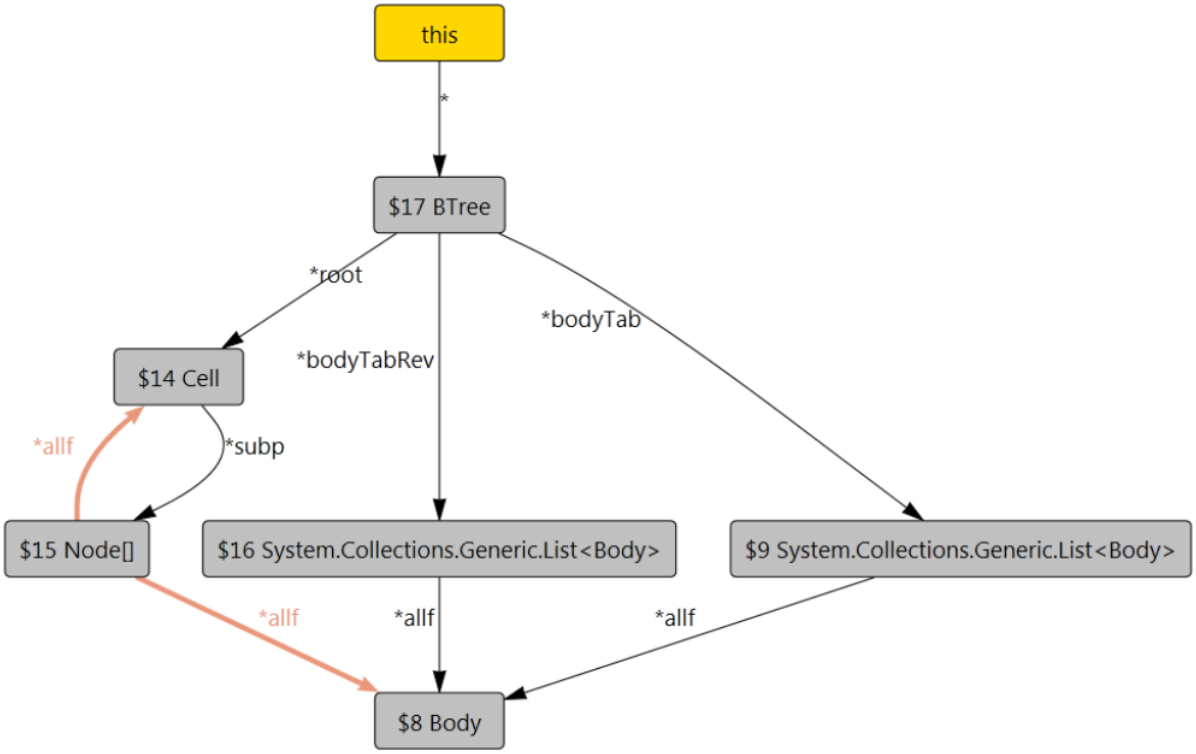


Figure 3.4: Abstract heap diagram of Barnes-Hut which has been converted to value type.

```
public class Body : Node
{
    public MathVector vel;
    public MathVector acc;
    public MathVector newAcc;
    public double phi;

}
```

Figure 3.5: Original definition of Body class

```
public class Body : Node
{
    public double vel_X;
    public double vel_Y;
    public double vel_Z;

    public double acc_X;
    public double acc_Y;
    public double acc_Z;

    public double newAcc_X;
    public double newAcc_Y;
    public double newAcc_Z;

    public double phi;

}
```

Figure 3.6: All members of the Body class defined explicitly inline.

Chapter 4

Conversion Safety Conditions

We begin by formalizing the conditions that need to be checked to determine if a by-reference type can be safely converted into a by-value type without altering the observable behavior of the program. In order to precisely describe these conditions we will examine this problem in the context of the C# programming language. However, we note that our approach can be applied to many other object-oriented programming languages with only minor changes. We split the conditions that need to be checked into two groups. The first group involves checking for a range of syntactic language properties that involve restrictions that the C# language places on the definitions (*e.g.* value types cannot be inherited from) and basic usage restrictions (*e.g.* value types cannot be assigned the null literal). The second set of conditions involve checking how the impact of a conversion to by-value semantics may impact the operational semantics of the program and thus the observable behavior of the execution (*e.g.* during object mutation or changing reference equality comparisons to value equality comparisons).

4.0.1 Syntactic Conditions

There are a number of restrictions the C# language places on what can be done with structs relative to objects. In this section we highlight some of the more important restrictions and how they can be checked. All of these conditions can be checked via simple analysis of program source code (and do not require any semantic analysis of program behavior).

Inheritance. In C# structs (value types) are not allowed to inherit from other classes/structs. However, structs are permitted to implement interfaces. Thus, Given a class c_τ the analysis must inspect the type system for the program to check that (1) the definition for c_τ does not declare any inheritance relations and (2) that there does not exist a c'_τ that inherits from c_τ.

Default Constructions and Field Values. The C# language allows class member fields to be given a default value during allocation and similarly objects can have parameterless constructors. However, to ensure that the by-value copy and default initialization routines can be implemented efficiently the C# language definition does not permit structs to declare either default values for member fields or a constructor with no parameters.

Constructor Completeness. In order to ensure that all of the fields in a struct are well defined the C# language requires that all constructors declared for a struct contain definite assignments to all member fields declared for the struct.

Type Decorators. In C# classes may be declared `static` (indicating that there are no member methods) or `sealed` (indicating that the class may not be further inherited from). However, as structs are always implicitly non-static and can never be inherited from these qualifiers are not meaningful when applied to structs and thus using them is classified by the C# language as a error.

Explicit Null Assignment and Compare. All structs have a well defined default value. In particular each field is assigned the default type for the given field with the C# language defining the default values for primitive types as:

$$\texttt{default(T)} = \begin{cases} \text{false} & \text{if T is bool} \\ 0 & \text{if T is int} \\ 0.0 & \text{if T is float} \\ \text{null} & \text{if T is pointer} \end{cases}$$

However, in contrast to pointers where `null` always represents *not-valid*, it is not clear if this default struct value is a *valid* value or, like for pointers, it represents something that is *not-valid*. Thus, there is an ambiguity in the meaning of explicit `null` assignments or comparisons. The C# language defines the special type `Nullable<T>` which can be used with struct types to add a special *not-valid* value to the domain of the struct type which functions like the `null` pointer value. In fact the C# language allows explicit assignment of the `null` pointer to `Nullable<T>` types, which can later be queried via the `HasValue` property to determine if it contains a value.

Runtime and Performance Issues. There are a number of situations in which conversion from by-reference semantics to by-value semantics may be safe wrt. no altering the behavior of the program but which may result in performance degradation instead of improvement. The

most two most obvious ways this can happen are if the class which is converted is large, resulting in expensive copy operations in the transformed program, and if the type is used heavily as a method parameter or return value, resulting in large numbers of copy operations being required. A more subtle issue is that often runtime systems are optimized for the common case of by-reference semantics instead of the less frequent by-value semantics. This can lead to some operations having unexpected costs. An example of such an operation is the `List<T>.Remove` operation in Microsoft's .Net 4.0 runtime which, when operating on a list of struct types, allocates a large number of `System.Reflection.FieldInfo[]`. All of these conditions can be checked for via a simple examination of the code in the source program.

4.0.2 Semantic Conditions

The next set of conditions we examine involve how the replacement of a by-reference type with a by-value type may impact the operational semantics in the program and thus alter the observable behavior of the program.

Immutability. The most obvious issue is that mutation of an object may be observed through multiple memory locations via pointer aliasing. The most obvious condition we can place on the program to eliminate this problem is to require that any classes we convert are *immutable*. We consider an object to be immutable if after construction no fields are modified. We present a formal algorithm for checking this in Section 5.0.6.

However, as seen in the experimental results (chapter 6) this is too strict and prevents to conversion of many classes that actually can be safely converted to structs. Thus, we present a more sophisticated criteria below, *Unique Write Observation*, which we use to determine if a write may be on an aliased location.

Equality and Equals. In C# equality (== and `!=`) have default definitions in the language. These default definitions are different for by-reference types, where equality is defined based on the addresses of the pointers, and by-value equality, which is defined based on the bitwise equality of the values. Thus, it is possible to have two by-reference types which have different target addresses but which point to memory locations that contain bitwise equal values (*i.e.* they are not equal by-reference but are equal by-value). In this case naively converting from by-reference to by-value type would alter the behavior of the program. Similarly the default implementation of the `Equals` member method differs for by-reference and by-value types.In Figure 4.1 two objects of type Complex has been compared and since this class doesn't override equality, the comparison would be based on pointer's addresses. So c1 and c2 objects are not equal and the assertion is correct.

```
public class Complex
{
    float real, imag;
    ...
}
...
public void Main()
{
    Complex c1 = new Complex(0.0f, 0.0f);
    Complex c2 = new Complex(0.0f, 0.0f);

    Assert(c1 != c2);
}
```

Figure 4.1: Complex Class with Default Equality – Not Convertible

However, the C# language permits the developer to override the equality operators (==, !=) and the `Equals` method. In many cases the user defined equality operators/methods are defined based on the values in the objects instead of the addresses of the objects (as in the default by-reference operations). If the operators have been overridden in this way then converting the type from a by-reference class to a by-value struct does not impact the behavior of the program. As shown in Figure 4.2 class Complex overrides equality operators so in method Main the comparison between c1 and c2 is based on their field's values and assertion is not correct.

Thus, in order for the transformation of a class into a struct to be safe, either (1) the program must not use the equality operations or (2) they must be overridden and the new implementation must be based only on the values of the fields in the objects. This check is complicated by the fact that the `Equals` method can be implicitly invoked by many operations in the standard library and virtual calls to the `Equals` method must be resolved. We present a method for checking these properties in Section 5.0.6.

4.0.3 Unique Write Observation

As there are many by-reference class types which can be safely converted to structs but which are not *immutable* this section outlines a novel property *unique write observation*. This property is sufficient to ensure that the observational equivalence of a program using a by-reference class and this program after replacing the class with a by-value struct. In this section we informally describe this condition and, after introducing our formal heap model, we present a formal definition and method for checking it in Section 5.0.6.

Figure 4.3 shows a program fragment which contains a write that is observed via multiple locations. In particular the write occurs through the variable `c1` but, due to aliasing, the write

```
public class Complex
{
    float real, imag;
    ...
    public static bool operator ==(Complex c1, Complex c2)
    { return (c1.real == c2.real) & (c1.imag == c2.imag); }

    public static bool operator !=(Complex c1, Complex c2)
    { return (c1.real != c2.real) | (c1.imag != c2.imag); }
}
...
public void Main()
{
    Complex c1 = new Complex(0.0f, 0.0f);
    Complex c2 = new Complex(0.0f, 0.0f);

    Assert(c1 != c2);
}
```

Figure 4.2: Complex Class with User Defined Equality – Convertible

```
public void Main()
{
    Complex c1 = new Complex(0.0f, 0.0f);
    Complex c2 = c1;

    c1.real = 1.0f;
    Assert(c2.real == 1.0f);
}
```

Figure 4.3: Mutation With Multiple Observers

is also observed when reading values through the location `c2`. In this example converting the `Complex` class to a struct would alter the semantics of the assignment `Complex c2 = c1` from by-reference assignment, which creates an alias, to by-value assignment, which simply copies the values in `c1` into `c2`. The result of this copy is that the later write through `c1` is not observed by (does not effect) `c2` which changes the behavior of the assertion. Thus, we know that writes to aliased locations are potentially unsafe.

Figure 4.4 shows a similar code fragment but, critically, in this fragment the variables `c1` and `c2` are not aliased. Thus, the write to `c1.real` is not observed when later reading values through `c2`. Thus, in this program converting the `Complex` class into a struct does not alter the behavior of the program. This leads to the following definition of *unique write observation*.

Definition 4 (Unique Write Observation Variable)**.** *Given an instruction* `v.f = x;` *and the concrete program state* $(\mathsf{Env}_1, \sigma_1, \mathsf{Ob}_1)$ *the write is uniquely observed through the variable* `v`

```
public void Main()
{
    Complex c1 = new Complex(0.0f, 0.0f);
    Complex c2 = new Complex(0.0f, 0.0f);

    c1.real = 1.0f;
    Assert(c2.real == 0.0f);
}
```

Figure 4.4: Mutation That is Uniquely Observed

if:

- $\forall v' \in \mathsf{Vars}, v' = v \vee \mathsf{Env}(v') \neq \mathsf{Env}(v)$
- $\forall o \in \mathsf{Ob}, f \in \mathsf{Labels}, o.f \neq \mathsf{Env}(v)$

The definition of unique write observation through variables is a straight forward formalization of the statement that the only location which holds the address of the write target is the variable `v`. This is checked by first asserting that any variable which is not `v` has a different address and all addresses stored in heap objects differ from the address stored in `v`.

Definition 5 (Unique Write Observation Heap Location)**.** *Given an instruction* `v.f.g = x;` *and the concrete program state* $(\mathsf{Env}_1, \sigma_1, \mathsf{Ob}_1)$*, where* $\sigma(\mathsf{Env}(v)) = o$*, the write is uniquely observed through the field* `f` *if:*

- $\forall v' \in \mathsf{Vars}, \mathsf{Env}(v') \neq o.g$
- $\forall o' \in \mathsf{Ob}, f \in \mathsf{Labels}, (o' = o \wedge f = g) \vee (o.f \neq o.g)$

The definition of unique write observation is the dual of the definition for variables. It is again a straight forward formalization of the statement that all variables hold different addresses than the field location `g` in the object o. Similarly each object and field are checked to ensure that (aside from $o.g$) all of them contain a different address. While the definition given here only contains a single field dereference it can be generalized in the natural way to account for any number of intermediate dereferencing steps.

We note that the unique observed write condition (and immutability condition) directly ensure that after any write the program state of the original program and the by-value transformed programs are always equivalent. Thus, it is immediate that these conditions are sufficient to ensure that the original program and resulting programs are *observationally equivalent*.

Abb	Condition	Detail
IA	Inheritance: Abstract	The class is an abstract class or contains abstract methods
IP	Inheritance: Parent	The class is a parent of some other classes
IE	Inheritance: Extends	The class extends other classes
ST	Static	The class is static
NA	Null: Assignment	The `null` value is explicitly assigned to an object of this class
NC	Null: Compare	An object of this class is explicitly compared to the `null` value
OC	Object Comparison	Comparison of two objects of this class may occur.
RP	Runtime and performance	We may not see improvement from converting this class to struct
IM	Immutability	An instance of the class has been mutated,while there was more than one object points to this instance
EQ	Equality	This type exist in comparison while it did not overrides the equal method
CM	Constructor method	The definition of constructor method is incomplete.
MM	Main method	The class contains main method.

Table 4.1: Reference table for generated analysis reports.

4.0.4 Analyzer Reference Table

For ease of reference we summarize each of the conditions in Table 4.1. Each condition that makes a class unconvertible is listed along with an abbreviation for each item.

Chapter 5

Heap Analysis Domain

We begin this section by introducing two properties of the concrete program heap (introduced in chapter 3) that are used to define the abstract domain that is used to analyze the original program which uses by-reference types. As the intent of the analysis is to understand the behavior of the by-reference classes in the original program and to simplify the descriptions in this section, without loss of generality, we assume that the input program only uses by-reference classes. Thus, the simplified abstract domain in this section omits discussion of primitive types and by-value structs.

5.0.1 Concrete Heap Properties

In the context of a specific concrete heap, $(\mathsf{Env}, \sigma, \mathsf{Ob})$, a *region* of memory is a subset of concrete heap objects $C \subseteq \mathsf{Ob}$. It is useful to define the set $P(C_1, C_2, \sigma)$ of all non-null pointers crossing from region C_1 to region C_2 as:

$$P(C_1, C_2, \sigma) = \{(o_s, l, \sigma(o_s.l)) \mid \exists o_s \in C_1, l \in \mathsf{Fld}(\mathsf{Ty}(o_s)) \,.\, \sigma(o_s.l) \in C_2\}$$

Injectivity. Given two regions C_1 and C_2 in the heap, $(\mathsf{Env}, \sigma, \mathsf{Ob})$, the non-null pointers with the label l from C_1 to C_2 are *injective*, written $\mathsf{inj}(C_1, C_2, l, \sigma)$, if for all pairs of non-null pointers (o_s, l, o_t) and (o'_s, l, o'_t) drawn from $P(C_1, C_2, \sigma)$, $o_s \neq o'_s \Rightarrow o_t \neq o'_t$. As a special case when we have an array object, we say the non-null pointer set $P(C_1, C_2, \sigma)$ is *array injective*, written, $\mathsf{inj}_{[]}(C_1, C_2, \sigma)$, if for all pairs of non-null pointers (o'_s, i, o_t) and (o_s, j, o'_t) drawn from $P(C_1, C_2, \sigma)$ and i, j valid array indices, $i \neq j \Rightarrow o_t \neq o'_t$.

5.0.2 Abstract Heaps

An abstract heap is an instance of a storage shape graph [6]. More precisely, an abstract heap graph is a tuple: $(\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}})$ where:

$$\begin{aligned}
\widehat{\mathsf{Env}} \in \text{Environments} &= \mathsf{Vars} \rightharpoonup \widehat{\mathsf{Addresses}} \\
\widehat{\sigma} \in \text{Stores} &= \widehat{\mathsf{Addresses}} \to \mathsf{Inj} \times 2^{\text{Nodes}} \\
&\quad \text{where the injectivity values } \mathsf{Inj} = \{\mathit{true}, \mathit{false}\} \\
\widehat{\mathsf{Ob}} \in \text{Heaps} &= 2^{\text{Nodes}} \\
\text{Nodes} = &\mathsf{NID} \times 2^{\mathsf{Types}} \times \times (\widehat{\mathsf{Labels}} \rightharpoonup \widehat{\mathsf{Addresses}}) \\
&\quad \text{where the node identifier set } \mathsf{NID} = \mathbb{N}
\end{aligned}$$

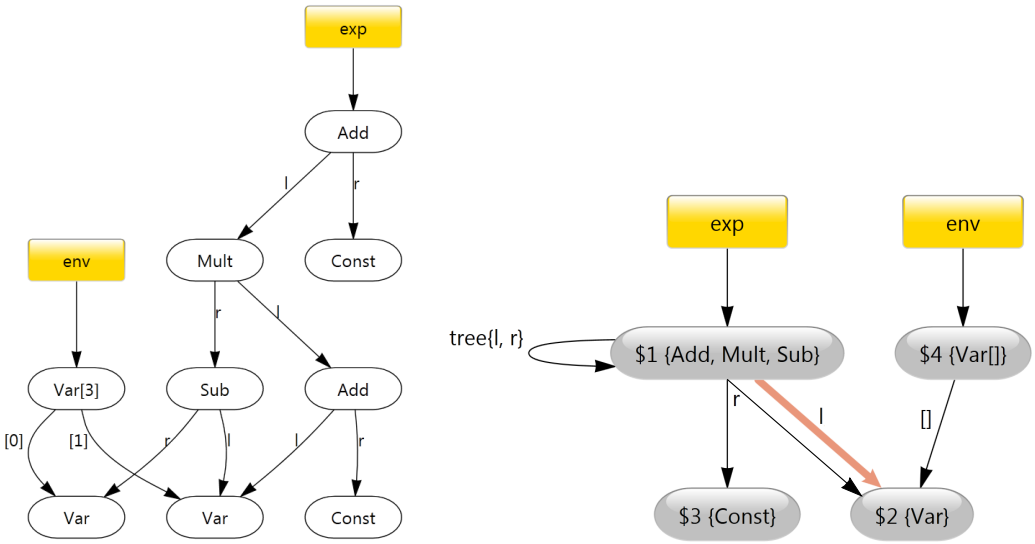
The abstract store ($\widehat{\sigma}$) maps from abstract addresses to tuples consisting of the injectivity associated with the abstract address and a set of target nodes. Each node n in the set $\widehat{\mathsf{Ob}}$ is a tuple consisting of a unique identifier for the node, a set of types, a shape tag, and a map from abstract labels to abstract addresses. The use of an infinite set of node identity tags, NID, allows for an unbounded number of nodes associated with a given type/allocation context allowing the local analysis to precisely represent freshly allocated objects for as long as they appear to be of special interest in the program [18]. The abstract labels ($\widehat{\mathsf{Labels}}$) are the field labels and the special label []. The special label [] abstracts the indices of all array elements (i.e., array smashing). Otherwise an abstract label $\widehat{l}$ represents the object field with the given name.

As with the concrete objects we introduce the notation $\widehat{\mathsf{Ty}}(n)$ to refer to the type set associated with a node. The notation $\widehat{l}$ is used to refer to the abstract value associated with the label $\widehat{l}$. Since the abstract store ($\widehat{\sigma}$) maps to tuples of *injectivity* and node target information we use the notation $\widehat{\mathsf{Inj}}(\widehat{\sigma}(\widehat{a}))$ to refer to the *injectivity* and $\widehat{\mathsf{Trgts}}(\widehat{\sigma}(\widehat{a}))$ to refer to the set of possible abstract node targets associated with the abstract address. We define the helper function $\widehat{\mathsf{Fld}} : 2^{\mathsf{Types}} \to 2^{\widehat{\mathsf{Labels}}}$ to refer to the set of all abstract labels that are defined for the types in a given set (including [] if the set contains an array type).

5.0.3 Abstraction Relation

We are now ready to formally relate the abstract heap graph to its concrete counterparts by specifying which heaps are in the concretization (γ) of an abstract heap:

$$\begin{aligned}
(\mathsf{Env}, \sigma, \mathsf{Ob}) \in \gamma((\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}})) \Leftrightarrow \exists \text{ an embedding } \mu \text{ where} \\
\mathsf{Injective}(\mu, \mathsf{Env}, \sigma, \mathsf{Ob}, \widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}})
\end{aligned}$$



(a) A Concrete Heap. (b) Corresponding Abstract Heap.

Figure 5.1: Concrete and Abstract Heap

A concrete heap is an instance of an abstract heap, if there exists an embedding function $\mu : \mathsf{Ob} \to \widehat{\mathsf{Ob}}$ which respects the structure and labels of the concrete heap and also satisfies the injectivity and shape relations between the structures.

$$\mathsf{Injective}(\mu, \mathsf{Env}, \sigma, \mathsf{Ob}, \widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}}) = \forall n_s, n_t \in \widehat{\mathsf{Ob}}, \widehat{l} \in \widehat{\mathsf{Fld}}(\widehat{\mathsf{Ty}}(n_s)) \,.\, \widehat{\mathsf{Inj}}(\widehat{\sigma}(n_s.\widehat{l})) \Rightarrow$$
$$(\widehat{l} \neq [] \Rightarrow \mathsf{inj}(\mu^{-1}(n_s), \mu^{-1}(n_t), l, \sigma)) \wedge (\widehat{l} = [] \Rightarrow \mathsf{inj}_{[]}(\mu^{-1}(n_s), \mu^{-1}(n_t), \sigma))$$

The injectivity relation guarantees that every pointer set marked as injective corresponds to injective (and array injective as needed) pointers between the concrete source and target regions of the heap.

5.0.4 Example Heap

Figure 5.1(a) shows a snapshot of the concrete heap from a simple program that manipulates expression trees. An expression tree consists of binary nodes for `Add`, `Sub`, and `Mult` expressions, and leaf nodes for `Constants` and `Variables`. The local variable `exp` (rectangular box) points to an expression tree consisting of 4 interior binary expression objects, 2 `Var`, and 2 `Const` objects. The local variable `env` points to an array representing an environment of `Var` objects that are shared with the expression tree.

Figure 5.1(b) shows the corresponding abstract heap for this concrete heap. To ease discussion each node in a graph is labeled with a unique node id. The abstraction summarizes the concrete objects into three regions represented by the nodes in the abstract heap graph: (1) a node representing all interior recursive objects in the expression tree (`Add`, `Mult`, `Sub`), (2) a

node representing the two `Var` objects, and (3) a node representing the two `Const` objects. The edges represent possible sets of non-null cross region pointers associated with the given abstract labels. Details about the order and branching structure of expression nodes are absent but other more general properties are still present. The label `tree{l,r}` on the self-edge expresses that pointers stored in the `l` and `r` fields of the objects represented by node 1 form a tree.

The abstract graph also captures the fact that no `Const` object is referenced from multiple expression objects but that several expression objects might point to the same `Var` object. The abstract graph shows this possible non-injectivity using wide orange colored edges (if color is available), whereas normal edges indicate injective pointers. Similarly the edge from node 4 (the `env` array) to the set of `Var` objects represented by node 2 is injective, not shaded and wide. This implies that there is no aliasing between the pointers stored in the array (a fact which could not be obtained via points-to analysis).

5.0.5 Normal Form

Given the definitions for the abstract heap it is clear that the domain is infinite. Thus, we define a normal form that ensures the number of distinct normal form graphs is finite and use this set during the fixpoint computation (see [18] for additional information).

Definition 6 (Normal Form)**.** *We say that the abstract heap is in normal form iff:*

1. *All nodes are reachable from a variable or static field.*
2. *All recursive structures are summarized (Definition 7).*
3. *All equivalent successors are summarized (Definition 9).*
4. *All variable/global equivalent targets are summarized (Definition 10).*

This normal form definition possesses three key properties that ensure finiteness: (1) the resulting abstract heap graph has a bounded depth, (2) each node has a bounded out degree, and (3) for each node the possible targets of the abstract addresses associated with it are unique wrt. the label and the types in the target nodes.

As each of the properties (*recursive structures*, *ambiguous successors*, and *ambiguous targets*) are defined in terms of, congruence between abstract nodes the transformation of an abstract heap into the corresponding normal form is fundamentally the computation of a congruence closure over the nodes in the abstract heap followed by merging the resulting equivalence sets. Thus, we build a map from the abstract nodes to equivalence sets (partitions) using a Tarjan union-find structure. Formally $\Pi : \widehat{\mathsf{Ob}} \to \{\pi_1, \ldots, \pi_k\}$ where $\pi_i \in 2^{\widehat{\mathsf{Ob}}}$ and $\{\pi_1, \ldots, \pi_k\}$ are a *partition* of $\widehat{\mathsf{Ob}}$.

Recursive Structures. The first step in computing the normal form is to identify any nodes that may be parts of unbounded depth structures. This is accomplished by examining the type system for the program that is under analysis and identifying all the types, τ_1 and τ_2, that have mutually recursive type definitions denoted: $\tau_1 \sim \tau_2$.

Definition 7 (Recursive Structure)**.** *Given two partitions* π_1 *and* π_2 *we define the* recursive structure *congruence relation as*[1]*:*

$$\pi_1 \equiv_r^{\Pi} \pi_2 \Leftrightarrow \exists \tau_1 \in \bigcup_{n_1 \in \pi_1} \widehat{\mathsf{Ty}}(n_1), \tau_2 \in \bigcup_{n_2 \in \pi_2} \widehat{\mathsf{Ty}}(n_2) \,.\, \tau_1 \sim \tau_2$$
$$\wedge \exists n \in \pi_1, \widehat{l} \in \widehat{\mathsf{Fld}}(\widehat{\mathsf{Ty}}(n)) \,.\, \widehat{\mathsf{Trgts}}(\widehat{\sigma}(n.\widehat{l})) \cap \pi_2 \neq \emptyset$$

Equivalent Successors and Targets. The other part of the normal form computation is to identify any partitions that have *equivalent successors* and variables that have *equivalent targets*. Both of these operations depend on the notion of a successor partition which is based on the underlying structure of the abstract heap graph and a general notion of node compatibility: π_1 a successor of π_2 and $\widehat{l} \Leftrightarrow \exists n_2 \in \pi_2 \,.\, \widehat{\mathsf{Trgts}}(\widehat{\sigma}(n_2.\widehat{l})) \cap \pi_1 \neq \emptyset$.

Definition 8 (Partition Compatibility)**.** *We define the relation Compatible(*π_1*,* π_2*) as:* $\text{Compatible}(\pi_1, \pi_2) \Leftrightarrow \bigcup_{n' \in \pi_1} \widehat{\mathsf{Ty}}(n') \cap \bigcup_{n' \in \pi_2} \widehat{\mathsf{Ty}}(n') \neq \emptyset$.

Definition 9 (Equivalent Successors)**.** *Given* π_1*,* π_2 *which are successors of* π *on labels* $\widehat{l_1}$*,* $\widehat{l_2}$ *we define the relation* π_1*,* π_2 equivalent successors *as:* $\pi_1 \equiv_s^{\Pi} \pi_2 \Leftrightarrow \widehat{l_1} = \widehat{l_2} \wedge \text{Compatible}(\pi_1, \pi_2)$.

Definition 10 (Equivalent on Targets)**.** *Given a root r (a variable or a static field) and two target partitions* π_1*,* π_2 *we define the* equivalent targets *relation as:* $\pi_1 \equiv_t^{\Pi} \pi_2 \Leftrightarrow \text{Compatible}(\pi_1, \pi_2) \wedge$ (*r is a static field* $\vee$ π_1, π_2 *only have local var predecessors*).

Using the *recursive structure* relation and the *equivalent successor (target)* relations we can efficiently compute the congruence closure over an abstract heap producing the corresponding normal form abstract heap (Definition 7). This computation can be done via a standard worklist algorithm that merges partitions that contain equivalent nodes and can be done in $O((N+E) * log(N))$ time where N is the number of abstract nodes in the initial abstract heap, and E is the number of abstract addresses in the heap.

5.0.6 Condition Checking

Using this domain the semantic conditions for conversion from chapter 4 can be checked in a straight forward manner. The abstract heap analysis is run on the program to compute a call-

[1]The definition is symmetric on the properties of the nodes, the $\tau_1 \sim \tau_2$ equivalence relation on types, but is not symmetric on the structure of the underlying graph.

graph and an over-approximation of the heap state at every point in the program. We then iterate over all the instructions in the program and inspect the abstract heap model at the operations (assignment, comparison, and method calls).

Immutability. The immutability condition is checked by enumerating the bytecodes in every method and for each assignment noting the type of the node that the write operation is mutating. The main complication is for writes in constructors where we need to ignore assignments to fields through the `this` object *unless* other pointers to it have been created. The model enables these items to be quickly checked by ensuring that (1) the target abstract node of the write has a single incoming edge and (2) that this edge comes from the `this` variable.

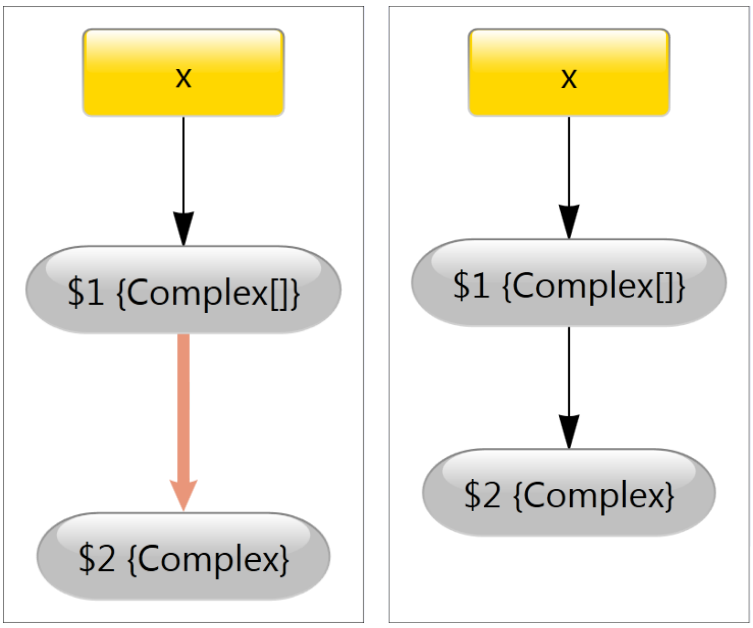
Equality and Equals. Checking for use of the equality or equals methods is again a straight forward enumeration and checking of the relevant bytecodes and method invocations. However, as C# programs can make heavy use of interfaces (which structs can implement) we make use of the call graph to determine if a call to `Equals` on an interface type may result in a call to the `Equals` method of a type we are interested in.

Unique Write Observation. In order to check the unique write observation condition at a member field assignment we consider two cases: writes through a variable and writes through a pointer stored in the heap.

The first case is a simple write through a variable. In this case we simply check that in the abstract heap model the only edge which points to the target abstract node is the edge associated with the variable that the write occurs through.

The more complex case is if the write is being done through a pointer stored in a heap location. In this case we first check, as in the write through variable cases, that there is a single incoming edge to the abstract location that is being modified by the write. However, since edges between abstract heap locations may represent multiple pointers simply ensuring there is a single incoming edge is *not* sufficient to ensure that there is only a single location which contains a pointer to the object which is being modified by the write (as needed by the definition of *unique write observation*). Thus, we also check that this single incoming edge is also *injective*, *i.e.* none of the pointers it represents are aliased, which implies that there is only a single location which contains a pointer to the object being written.

Figure 5.2(b) shows an example with two abstract heaps. On the left is an abstract heap where the write `x[3].g = 1` may not be uniquely observed as the edge associated with the pointers stored in the array (the *[]* labeled edge) are not injective. However, the abstract heap on the right illustrates an abstract heap where the write `x[3].g = 1` is guaranteed to be uniquely observed as the node representing the objects that are pointed to by the array has (1) a single



(a) It is not safe to convert to value type.

(b) It is safe to convert to value type.

Figure 5.2: Abstract heap shows if an object is injective to not.

incoming edge and (2) this edge is *injective*. Thus, this example illustrates how the notion of uniquely observed writes enables the conversion to structs of both objects that are modified through variables but also when they are stored in heap based data structures. In addition this example illustrates the utility of the heap domain, and injectivity information, described in this section.

Chapter 6

Implementation and Experimental Evaluation

In order to help developer quickly identify the convertible classes we implemented a plug-in in Microsoft Visual Studio using the Roslyn Framework [2] and the Jackalope heap analyzer [1]. This application analyzes C# programs and generates a detailed report about the convertibility (or issues identified) for each class in the program. The add-in also supports a automatic conversion of classes that are identified as convertible.

In order to check the effect of the conversion, we selected several programs to use as case studies. These programs represent code from high-performance computing, simple databases, and image rendering. We used our software to find out which classes are convertible to structs and evaluate the following questions:

1. How many classes the converter is able to identify as trivially convertible, i.e. by changing the `class` keyword to `struct`.

2. Of the classes which are not trivially convertible, which conditions do they fail and how much manual refactoring effort is required to convert the most profitable (based on profiling information) of these.

3. What is the impact of these conversions on total memory allocated.

4. What is the impact of these conversions on Total Execution Time.

We measure the runtimes using a command-line batch file to run the application K times and report the average of all runs. Memory allocation is measured using the .Net Analyzer in Visual Studio 2010 Ultimate.

Class	Is Convertible?	Reason
`Driver`	No	MM
`Raytracer`	Yes	
`Surfaces`	No	ST
`Vector`	Yes	
`Color`	Yes	
`Ray`	Yes	
`Isect`	No	NC $\times$ 1 , NA $\times$ 1
`Surface`	Yes	
`Camera`	Yes	
`Light`	Yes	
`ScenceObject`	No	IP, IA
`Sphere`	No	IE
`Plane`	No	IE
`Scene`	Yes	

Table 6.1: Convertibility report for Raytracer. The safety of converting each type is reported in the *Is Convertible?* column. For types that are not convertible the detected issues and multiplicities of these issues are reported in the *Reason* column refer to Table 4.1.

6.0.1 Case study:Raytracer

The first case study is a basic Raytracer from Microsoft Research and was initially used as a demonstration benchmark for the Thread Parallel Libraries (TPL) [3]. This program generates an image by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects (in our benchmark 2 spheres on a flat surface).

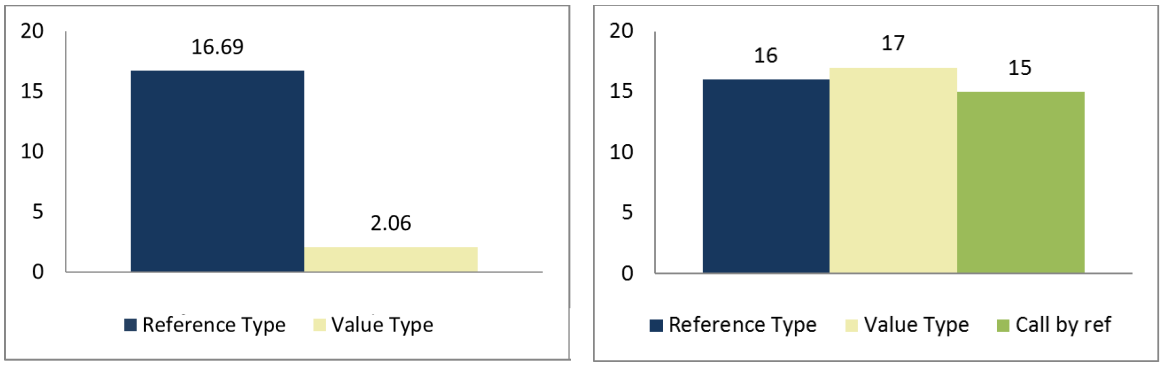
Table 6.1 shows an overview of the analysis report generated by our tool. For each user defined class in the program we list the convertibility result in the *IsConvertable?* column and if it is not convertible we list the reasons as reported by the tool in the *Reason* column. In the table the reasons for non-convertibility are given via their code from Table 4.1 and we also show the multiplicity of the locations where the given issue was detected in the source. In actual use the tool provides a more extensive listing, including line numbers, for these issues so that programmers can quickly and easily resolve them to enable safe conversion to using struct types.

The result shows that more than 50% of the classes are automatically convertible. However, we want to focus converting classes that have a large number of instances as these converting them will have the largest impact on memory usage and performance. Table 6.2 shows the top three class types, as reported by the Visual Studio Profiler, that were allocated during program execution.

Figure 6.1 shows the performance of the original program using by-reference classes and the program that results from automatically converting the top three types, as reported by the

Class	Pct. Allocations	Class Fields	Total Size
`Vector`	57.72%	`double` × 3	32 B
`Color`	27.54%	`double` × 3	32 B
`Ray`	9.34%	`Vector` × 2	16 B

Table 6.2: Highest allocation rate classes. The percentage of allocations for each of the 3 classes with the highest allocation is shown in column *Pct. Allocations*. The *Class Fields* column shows the member fields for the type and the *Total Size* column reports the number of bytes required by a single object of the type.



(a) Total Memory Allocation In Megabyte.

(b) Total Execution Time.

Figure 6.1: Raytracer performance comparison: baseline by-reference implementation (with classes) and converted by-value implementation (using structs).

profiler in Table 6.2, to use by-value structs. The total memory allocated by each program is shown in Figure 6.1(a) and the Total Execution Time for the programs is shown in Figure 6.1(b). The results show that converting to value types reduce the total memory allocation by 87%, compared to reference type implementation.

These results show that a large reduction in memory allocation can be obtained by converting only a few types in the program. Although the converted classes are relatively small (between 16-32 bytes each) they are used extensively throughout the program and thus the number that are allocated is very large during the program execution.

The results also show (Value Type bar in Figure 6.1) that the conversion to the value types results in a slight slowdown in execution. This is due to the frequent passing of `Vector` types as parameters. As these types are 32B each this has a non-trivial performance impact. However, C# allows by-reference passing of value types using explicit `ref` parameters. Converting the needed parameters results in an implementation which reduces execution time by 6% (the Call by ref bar).

Class	Is Convertible?	Reason
BH	No	MM
BTree	No	IM $\times$ 4
Node	No	IA ,IP
Cell	No	NC $\times$ 1,NA $\times$ 1,IM $\times$ 2,IE
HG	Yes	
MathVector	No	NA $\times$ 3
Body	No	IM $\times$ 7 ,OC $\times$ 1,IE

Table 6.3: Convertibility report for BH. The safety of converting each type is reported in the *Is Convertible?* column. For types that are not convertible the detected issues and multiplicities of these issues are reported in the *Reason* column.

Class	Pct. Allocations	Class Fields
Double[]	71.06%	
MathVector	23.69%	double[] with length 3
HG	1.28%	MathVector ,Body(3 $\times$ MathVector, double) ,double

Table 6.4: Highest allocation rate classes. The percentage of allocations for each of the 3 classes with the highest allocation is shown in column *Pct. Allocations*. The *Class Fields* column shows the member fields for the type and the *Total Size* column reports the number of bytes required by a single object of the type.

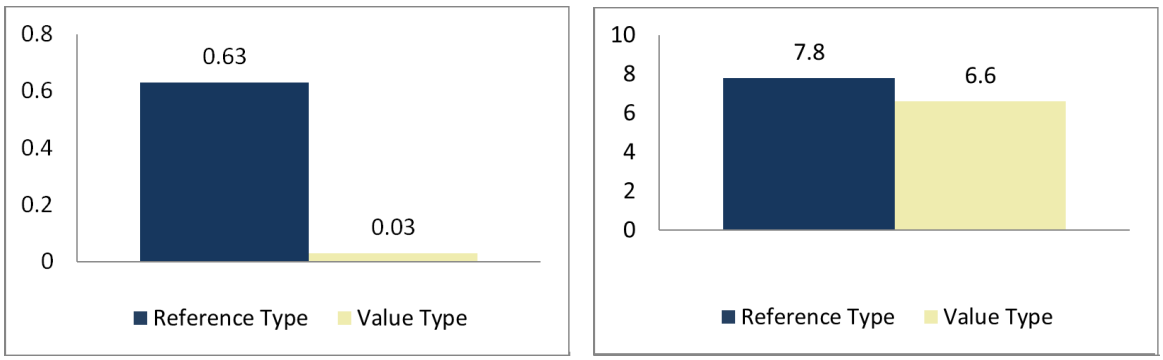
6.0.2 Case study:Barnes-Hut

Our second case study is an implementation of our motivating example, the Barnes-Hut algorithm. This program performs a gravitational interaction simulation on a set of bodies (the Body objects) using a fast-multiple technique with a space decomposition tree. The implementation includes seven classes. Class `MathVector` in this program represents a vector in three dimensional space. Since the nature of this program creates a lot of instances of type `MathVector` converting this class to value type would make a huge memory impact.The analyzer reports this class as not automatically convertible, as shown in Table 6.3. However, the report shows this is soley due to two explicit null assignments. Changing the code slightly to remove the null assignment is sufficient to make the `Vector` type convertible to struct. We also noticed that by converting the only member of the class, an array of double with fixed size of three, to three double primitive type can further improve program performance. According to Table 6.4 arrays of type double has more than 71% of the total number of allocated objects. The second highest is the `MathVector` type.

In table Table 6.3 BTree has been reported unconvertible because it is been mutated in four different parts of the program while it has more than one object pointing to it. Figure 6.2 shows on such part of the code. The method `expandBox` receives an instance of the `Btree`

```
public void expandBox(BTree tree, int nsteps)
{

 bool inbox = icTest(tree);

 tree.rsize = 2.0 * rsize;

 tree.root = newt;

}
```

Figure 6.2: Mutation happened at line 6 and 8 in method expandBox in Barnes-Hut program



(a) Total Memory Allocation In Megabyte.

(b) Total Execution Time.

Figure 6.3: BH performance comparison: baseline by-reference implementation (with classes) and converted by-value implementation (using structs).

class as a parameter. Inside the method this object has been is mutated. At the time the `BTree` object is mutated there are other live references pointing to the instance. It is obvious that programmer expect to see these changes in caller method after returning from the callee method (`expandBox`). However, converting the `BTree` class to a by-value type will result in copies being created for the callee and the loss of the updates in caller method.

To see how effective was the conversion of the `MathVector` and `HG` classes is we run .Net profiler and as Figure 6.3(a) shows, the BH application shows a 93% saving in memory allocation. Moreover these transformations result in a 15% of improvement in performance.

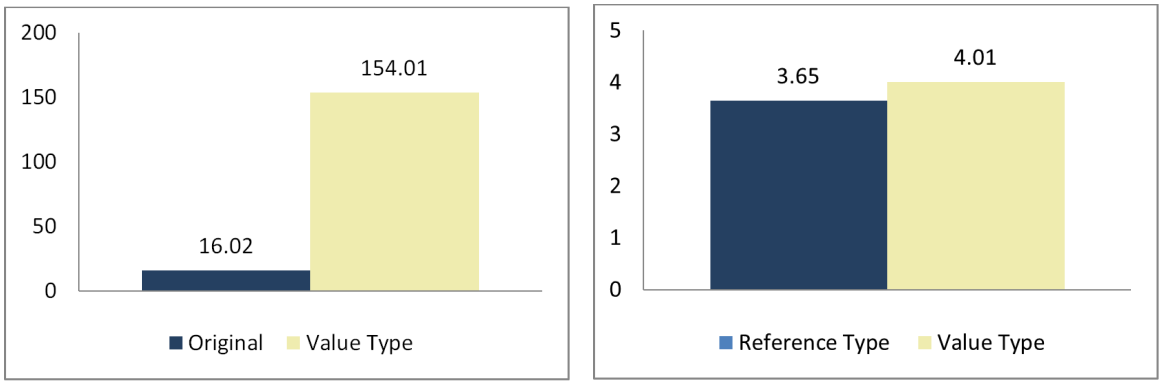
6.0.3 Case study:DB

The final case study is the Database application from the SPEC JVM98 benchmark suite (which we converted to C#). This program reads records from a 1MB data file and execute multiple database operations including add, delete, search and sort.

This program contains three classes of which the `Entry` class has the highest rate of alloca-

Class	Is Convertible?	Reason
Entry	Yes	
Database	No	IM × 6
MainCL	No	MM

Table 6.5: Convertibility report for DB. The safety of converting each type is reported in the *Is Convertible?* column. For types that are not convertible the detected issues and multiplicities of these issues are reported in the *Reason* column.



(a) Total Memory Allocation In Megabyte.

(b) Total Execution Time.

Figure 6.4: DB performance comparison: baseline by-reference implementation (with classes) and converted by-value implementation (using structs).

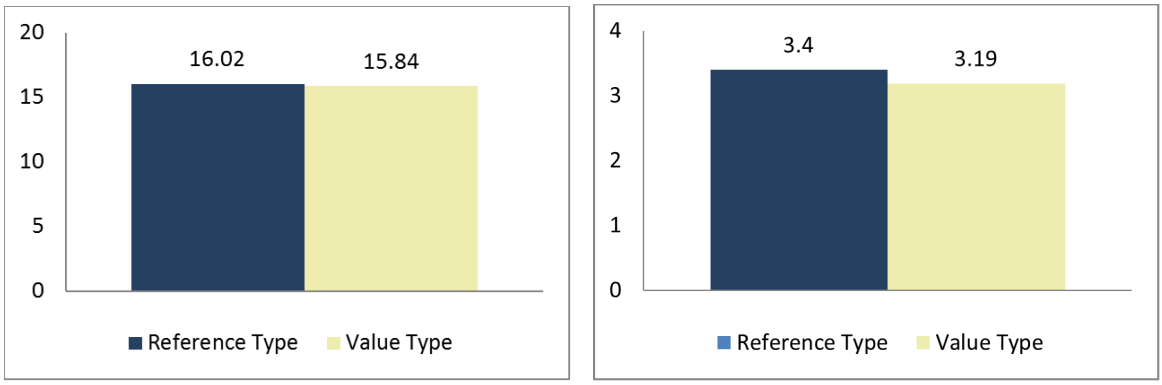
tion. So we expect converting it will be an effective way to decrease memory allocation.

Table 6.5 shows the analysis report generated by our tool. The result shows that the class `Entry` is automatically convertible. Converting this class is our best option. Since there is only one instance of class `Database` during runtime and this class is not automatically convertible we did not try to convert it.

Figure 6.4 shows the performance of the original program using by-reference classes and the program after we convert class `Entry` to a by-value type. Figure 6.4(a) shows the total memory allocated for each program and the Total Execution Time is shown in Figure 6.4(b). It can be seen that, despite our expectations, memory usage *increased* by 89%. And moreover the execution time grows up by 8%.

As can be seen we didn't achieve any performance improvement! So we checked the compiled code of the program and we discovered that this is because of the way .Net manipulates the `List` of struct type. Our investigation showed that the internal runtime was using reflection during copies involving value types, allocating large quantities of `System.Reflection.FieldInfo[]` objects.

We reported this finding as a potential performance bug in the .Net runtime to our collaborators at Microsoft. To evaluate the performance impact of our transformations in the absence



(a) Total memory usage in megabyte.

(b) Total Execution Time.

Figure 6.5: Performance comparison for the secon version of DB: in this version we eliminated the *remove* operation , baseline by-reference implementation (with classes) and converted by-value implementation (using structs).

of this problem we disabled the *remove* operation in the database and re-ran the programs. The new times are shown in Figure 6.5 the total memory allocation dropped by 140 MB in comparison with the case with the *remove* operation and reached the point that is slightly less than original by-reference type. Figure 6.5(b) shows that after we eliminated the *remove* operation the execution time also decreased by 6%.

Chapter 7

Related Work

The idea of altering the semantics of a type declaration from by-reference to by-value is a well known technique for improving the memory use and runtime of a program. Despite the ubiquity of this transformation and the large benefits it can provide, there is a relatively small amount of work on how to automate the process (either fully or partially). A major reason for the lack of work on this problem is the need to have precise heap sharing information, such as provided by the analysis used in this work, in order to identify opportunities outside of the simple case when the objects of interest are only stored in variables.

The largest body of related work is on the topic of compiler directed stack-allocation and escape analyses in object-oriented languages such as Java [5, 7, 10, 23]. These techniques focus on the identification of objects that do not *escape* from a given call scope and can thus be allocated on the call stack like a local variable which is then reclaimed automatically when the call returns. This approach provides several of the same benefits as the conversion of a by-reference type to a by-value type – reduced memory allocation/deallocation. However, the full object still needs to be allocated (including space for object headers) and it does not reduce the number of pointer loads that are taken during an access path traversal in the program. Thus, in contrast to a full conversion to by-value semantics, stack-allocation does not reduce the heap footprint and does not provide some performance benefits that arise from reduced memory loads.

The most directly related work is on the topic of object-colocation [9, 11, 17], conversion to immutable types [14], and pool allocation or region-based allocation [7, 12, 16, 19]. These approaches use various static analysis techniques to identify sets of objects that are in related structures and allocate them in the same memory region. Often they are also able to infer points in the program when all objects in a given region are known to be dead and can then free the entire region (instead of individually reclaiming each object). The work by Guyer and McKinley uses static analysis (along with runtime information) to allocated objects close to the object (or objects) that point-to them. The objective is to make memory prefetching

more effective by placing the objects nearby in memory. However, like the work on stack allocation, the full objects (including headers) must still be allocated and the pointers explicitly dereferenced when accessing the object fields. The work in [9, 17] examines how two object definitions can be merged (*i.e.* inlining) using the condition of *one-to-one fields* or *unique-store fields*. These definitions focus on when there is a unique points-to relation between parent and child objects either always the same child or allowing for multiple values at different times in the program. The definitions used in this work are orthogonal in the sense that they focus on mutation instead of ownership to drive the conversion. However, the check for *unique write observation* uses the *unique-store* field property when a write occurs. Thus, the safety definitions in this paper can be seen as a further refinement of the work in [9, 17].

Finally, we note that despite the large body of work on points-to analysis [4, 13, 21, 22] none of these techniques can satisfactorily resolve the needed unique ownership properties (*one-to-one*, *unique-store*, or *unique observer*) that are used in this paper and previous work. Conversely, work on shape analysis has resulted in a number of techniques that can resolve these properties but are too computationally expensive [20] or limited in the classes of programs they can accept [8, 24]. Thus, this work utilizes the Jackalope heap analysis toolkit [18] which is able to provide the precise information needed on sharing in order to resolve the unique observer property while being efficient enough and sufficiently robust to run on real-world programs.

Chapter 8

Conclusion

In this work we examined how high memory use in object-oriented programs can be addressed through the conversion of by-reference `class` types into by-value `struct` types. The approach taken was to provide an automated analysis tool which could (1) identify types that were safe to convert without any further program modifications and (2) when a type is not safe to convert to provide a summary of needed changes that can be made by the developer. The effectiveness of this approach was demonstrated on a suite of C# applications. The results show that the approach is useful in automatically identifying by-reference class types which can be safely converted to by-value struct types and, in the cases where conversion may not be safe, provides the developer with a useful summary of possible problems. Using a profiler we identified high allocation types and utilized the conversion reports to re-factor these class types into structs. The resulting programs show up to a 15% reduction in runtime and 93% reduction in total memory allocation. These positive preliminary results provide guidance for future work on the topic. Two topics of particular interest are the further automation of the conversion process and exploring how to perform conversion only for specific subcomponents of the program. In the first area of future work it is clear that many of the changes could automated and included as part of a programmer directed refactoring tool. The second area of work is motivated by the following observation: in many cases a class is often used as a by-value type (usually as an immutable type) in some computationally expressive parts of the program but there are other places in the code where converting the class to a struct is not feasible. Thus, research into identifying these *important* parts of the program and automatically generating the needed boxing/unboxing for the relevant classes is of interest.

References

[1] Jackalope heap analysis toolkit. http://jackalope.codeplex.com/.

[2] Microsoft Roslyn. http://msdn.microsoft.com/en-gb/roslyn.

[3] Thread Parallel libraries. http://code.msdn.microsoft.com/windowsdesktop/Samples-for-Parallel-b4b76364.

[4] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *PLDI*, 2003.

[5] B. Blanchet. Escape analysis for object-oriented languages: Application to Java. In *OOPSLA*, 1999.

[6] D. Chase, M. Wegman, and K. Zadeck. Analysis of pointers and structures. In *PLDI*, 1990.

[7] S. Cherem and R. Rugina. Region analysis and transformation for Java programs. In *ISMM*, 2004.

[8] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *PLDI*, 2011.

[9] J. Dolby and A. Chien. An automatic object inlining optimization and its evaluation. In *PLDI*, 2000.

[10] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *CC*, 2000.

[11] S. Guyer and K. McKinley. Finding your cronies: Static analysis for dynamic object colocation. In *OOPSLA*, 2004.

[12] N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. In *PLDI*, 2002.

[13] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *ISSTA*, 2001.

[14] F. Kjolstad, D. Dig, G. Acevedo, and M. Snir. Transformation for class immutability. In *ICSE*, 2011.

[15] Knuth. Structured programming with go to statements. ACM Journal Computing Surveys, 1974.

[16] C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *PLDI*, 2005.

[17] O. Lhoták and L. Hendren. Run-time evaluation of opportunities for object inlining in Java. In *ACM-ISCOPE*, 2002.

[18] M. Marron. Structural analysis: Combining shape analysis information with points-to analysis computation. arXiv:1201.1277v1 [cs.PL], 2012.

[19] F. Qian and L. J. Hendren. An adaptive, region-based allocator for Java. In *ISMM*, 2002.

[20] S. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, 1999.

[21] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. In *POPL*, 2011.

[22] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, 1996.

[23] J. Whaley and M. C. Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA*, 1999.

[24] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *CAV*, 2008.