Balloon Types: Controlling Sharing of State in Data Types

Paulo Sérgio Almeida *
Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ

Abstract

Current data abstraction mechanisms are not adequate to control sharing of state in the general case involving objects in linked structures. The pervading possibility of sharing is a source of errors and an obstacle to language implementation techniques.

We present a general extension to programming languages which makes the ability to share state a first class property of a data type, resolving a long-standing flaw in existing data abstraction mechanisms.

Balloon types enforce a strong form of encapsulation: no state reachable (directly or transitively) by a balloon object is referenced by any external object. Syntactic simplicity is achieved by relying on a non-trivial static analysis as the checking mechanism.

Balloon types are applicable in a wide range of areas such as program transformation, memory management and distributed systems. They are the key to obtaining self-contained composite objects, truly opaque data abstractions and value types—important concepts for the development of large scale, provably correct programs.

1 Introduction

Data abstraction mechanisms were introduced [6, 10] long ago and are used by most modern languages. While these mechanisms have proved successful when value semantics and named objects were the norm, this is not quite true in the case of dynamic linked data structures composed of unnamed objects and when assignment and parameter passing have reference semantics—something now common in object-oriented languages.

The use of dynamic linked structures is a source of errors due to the pervading possibility of sharing objects. The phenomenon of interference, for which solutions have been devised for the case of named objects [24], becomes considerably more complex, making it difficult to reason about programs or to perform program transformations such as parallelisation.

The contribution of this paper is a general language mechanism to control the complexity which arises from sharing unnamed mutable objects in dynamic linked structures, by making the ability to share state a first class property of a data type.

In section 2 we explain the problem in detail. In particular we explain why the current data abstraction mechanisms do not provide appropriate support for managing the sharing of objects and discuss the fundamental flaw in the form of encapsulation provided.

In section 3 we describe balloon types: the basic idea, the invariant it enforces, and give an overview of the checking mechanism to enforce the invariant. This includes a 'simple rule' and a non-trivial static analysis which we developed formally as an abstract interpretation. We also briefly introduce two important specialisations of balloon types—opaque balloon types and value types—and summarise the taxonomy of state sharing which results from balloon types.

*On leave from Departamento de Informática, Universidade do Minho, Portugal. Partially supported by PRAXIS XXI, grant BD-3390-94.

In section 4 we present the related work and finally the conclusions. The technical details of the abstract interpretation are somewhat involved and we do not describe them here, but we present in the appendix the most essential part.

2 State, sharing and encapsulation

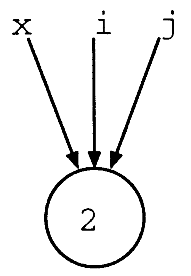
2.1 Values, objects and aliasing

Language semantics for toy imperative languages associate a variable with a value. If we are to consider realistic procedural or object-oriented languages we must make the distinction between variables, objects and values. The concept of object is needed to reason about sharing, as it is meaningless to talk about the sharing of values [21].

This is true even in the case of primitive types like integer: integer variables denote objects that contain (representations of) values. We use this model to accurately describe the behaviour of a program when *aliasing* is present. In the case of integers, aliasing can be created when using call-by-reference. For example:

```
PROCEDURE f(VAR i:INTEGER, VAR j:INTEGER)
BEGIN
  WRITE i;
  j := j+1;
  WRITE i
END

VAR x:INTEGER;
BEGIN
  x := 1;
  f(x, x)
END
```



During the invocation of procedure f variables i and j denote the same integer object: they are said to be aliases for the same object. The increment of j changes the object which is also denoted by i. As such, the two write statements will result in '1' and '2'.

As in [12, 13] we will use the terms:

- *dynamic aliasing* when stack based *variables* (including parameters) are involved (as above),
- *static aliasing* when *state variables*[1] of objects are involved (discussed next section).

Dynamic aliasing has a lifetime which depends on the execution of functions, whereas static aliasing reflects the structure of the object graph.

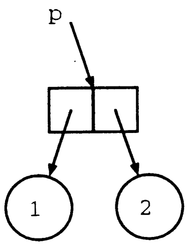
Aliasing has been long ago recognised as a source of problems [11] and some attempts have been made to prevent it. The previous example would be invalid in Pascal [28], due to using x twice as an argument, but in general dynamic aliasing cannot be prevented at compile time given parameter passing with reference semantics. An example is the call `f(A[i], A[j])`, when the indexes cannot be determined at compile time.

One nice property of integer objects is that they cannot be *statically aliased*: they cannot be shared by several state variables of objects.[2] This is a result of the assignment having value semantics: it copies the integer value. Consider the type `Point`, with state variables x and y of type `Int`:

[1] We will use the term *state variables* for what in different places is referred to as instance variables or fields.

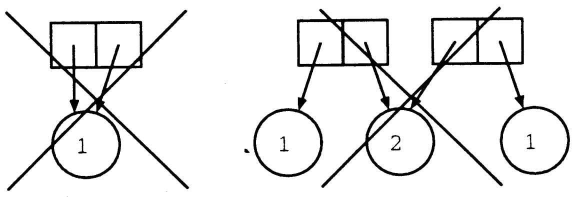
[2] This is assuming that languages impose some 'reasonable' constraints, like forbidding the explicit use of pointers and dereferencing. Such constraints tend to be incorporated in modern languages like Eiffel or Java. In languages like C++ [26] 'anything is possible', including storing in heap based objects pointers to stack based objects.

```
Point = { x,y:Int }
p:Point;
...
p.x := 1;
p.y := p.x;
increment(p.y);
write p.x, p.y;
```



Programmers expect that the increment operation applied to $p.y$ does not affect $p.x$ and that the outcome of the program is '1 2'.

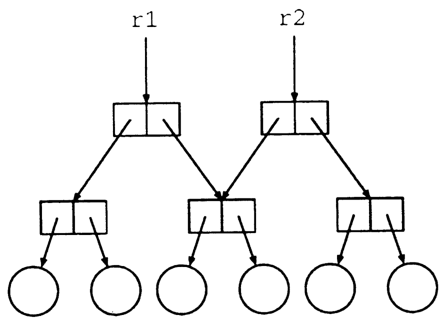
To summarise: in primitive types regardless of whether dynamic aliasing occurs (due to call-by-reference), static aliasing does not occur as the assignment has value semantics. The following cannot occur:



2.2 Sharing of state

In many object-oriented languages (eg. Smalltalk [8], Java [2]) variables of user-defined types are references to objects, and the assignment has reference semantics (copies just the reference). This makes sharing of objects by other objects (static aliasing) possible. As an example, consider the type **Rectangle**:

```
Rectangle = { p1,p2:Point;
              rotate(Int)
            }
r1,r2:Rectangle;
...
r2.p1 :- r1.p2;
r1.rotate(90);
r2.rotate(45);
```



As in Simula [6] we have used the ':-' notation for reference assignment. After the assignment both rectangles share a common point object. Consider the operation **rotate** which updates the point objects that constitute a rectangle; there would be interference between the two rotate instructions, as the first would modify a point that is accessed by the second.

Although sharing can be useful and may be desired in some cases, this is probably not what the users of rectangle objects would desire. They would expect that each rectangle is a self-contained object, and that operations on different rectangles do not interfere.

Programmers can obtain a copy of a point instead of copying a reference to it, but they can copy the reference accidentally. This can easily happen if the available assignment operator copies just the reference.

Expanded types were introduced in Eiffel [22], whereas originally user-defined types were always *reference types*. If a type is declared as expanded, variables will hold the object itself and not a reference to the object; also parameter passing and assignment copies the object.

Expanded types solve the problem in this particular case: the programmer just has to declare types **Point** and **Rectangle** to be expanded. This however makes it impossible to pass references to rectangles to a function so that the objects can be updated in-place. Moreover, it is not possible to have subtype polymorphism in expanded types.

```
Shape = { rotate(Int) }

Rectangle <: Shape
    = { p1,p2:Point }
Circle <: Shape
    = { c:Point; r:Real }
Polygon <: Shape
    = { List[Point] }
Graph <: Shape
    = { ... }

a:Array[Shape];
for i = 1 to N
  a[i].rotate(45);
```

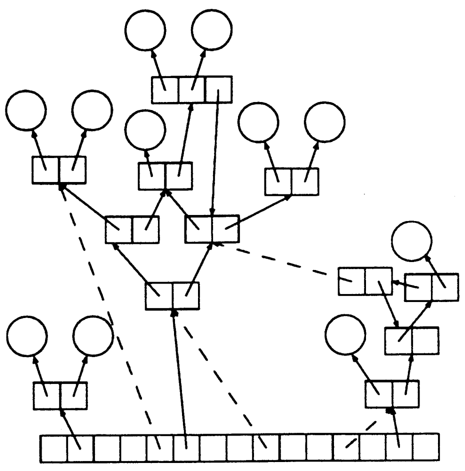


Figure 1

2.3 Sharing and unbounded linked structures

Expanded types provide insufficient support to prevent unwanted sharing. In the case of unbounded linked structures such as linked lists the recursive nature of these types prevents them from being declared as expanded. This means that even if one type is declared as expanded, objects of that type may need to reference a list, which cannot be an expanded type and may itself be shared. Expanded types thus fail since they are not able to prevent sharing of linked substructures.

Consider a **Shape** type with several subtypes such as **Rectangle** and **Polygon**. Some of these types may require pointer structures such as a linked list of points in the case of polygon. Some of the structures may even contain cycles. Suppose we have an array of shapes and a loop which rotates each of the shapes in the array, as illustrated in Figure 1.

It could happen contrary to the programmer's intent that two shapes share the whole or part of the objects of their states, as illustrated by the dashed arrows. This would imply that performing a rotate on one shape would interfere with other rotate operations, contrary to the expectations of the programmer.

This pervading possibility of sharing state is what makes it difficult to reason about programs in procedural or object-oriented languages. Contrast the shapes example with plain integers:

```
a:Array[Int];
for i = 1 to N
  increment(a[i]);
```

Although trivial for integers, it can be extremely difficult for the compiler to determine in the case of shapes if the different iterations of the loop interfere. Providing an unchecked directive so that a programmer who 'knows' that they do not interfere gives that 'knowledge' to the compiler is dangerous. It can be the case that the programmer is wrong and they do indeed interfere, due to a bug in the implementation of some operation that causes the unwanted sharing.

2.4 Data abstraction and encapsulation of state

According to [27], a data abstraction is "an object whose state ~~is~~ accessible only through its operations". It may be thought that current data abstraction mechanisms are appropriate enough for controlling sharing of state. The problem is that currently they just control the access to the state variables and not to the whole reachable state; they consider it 'other objects'. However, to reason about program behaviour it matters precisely whether these 'other objects' are shared.

Only by thinking of the state associated with an object as the state directly or transitively reachable by the state variables is it possible to argue about whether the state is encapsulated (and not referenced by external objects), or is shared (and part of it is also referenced by external objects). This is how we see state and encapsulation of state.

The same view of state is expressed in [13], and a similar attitude towards encapsulation can be perceived in [3]. Also [12] has this interesting remark: 'the big lie of object-oriented programming is that objects provide encapsulation'.

A widespread misconception is that if encapsulation (as we see it) is wanted it is enough not to have functions of the type returning references to the state; this is definitely false:

- There can be interaction between the state and the parameters received by some function of the data type. This interaction can involve invocations of operations which may cause some object from the state to become referenced by an object reachable by a parameter or vice-versa, breaking the encapsulation of the state.
- The implementation itself, while manipulating several instances of the data type, may cause sharing of their states.

These situations may happen accidentally, contrary to the expectations of the implementor of the data type, with no warning or prevention by the compiler.

3 Balloon types

Even if technically possible, a data type should not always enforce encapsulation of state. Although encapsulation may be wanted for some types, for others sharing may be needed. Designers of data types must be able to choose.

The point we make is that current languages do not provide a suitable mechanism for making this choice. One source of problems is precisely because this choice is not apparent (it may have not even been considered), and users of a data type may have wrong expectations about the behaviour in terms of sharing.

The basic idea of balloon types is precisely to make the ability of sharing state a first class property of data types, as important as the operations provided and their signatures. Among other things: it becomes part of a type definition, it is considered in type checking, it affects what code programmers are allowed to write, it is considered in the formal reasoning about programs, and it is used in compiler optimisations.

We propose a binary classification of data types with respect to sharing properties. Any type[3] is classified as either a *balloon* type or a *non-balloon* type.

- Balloon types are useful to prevent unwanted sharing of state, guaranteeing encapsulation of state. They result in cleaner semantics, being a means to prevent unexpected interference.
- Non-balloon types correspond to what current languages offer regarding user-defined types. They allow full freedom of sharing, being useful to represent linked structures with possible substructure sharing.

Balloon types have the properties that:

- Objects of a balloon type are unsharable by state variables of objects.
- All the state reachable by a balloon is encapsulated, in the sense that no part of it can be referenced by state variables of any 'external' object.

Some examples of balloon types are primitive types such as integer, real and boolean. People expect that they may be at most (and preferably not) dynamically aliased, but not statically aliased (not shared by different objects). There is no more than one object owner of an integer

[3] We will use frequently 'type' as a short for data type (including primitive types), as we will not mention function types or higher-order types.

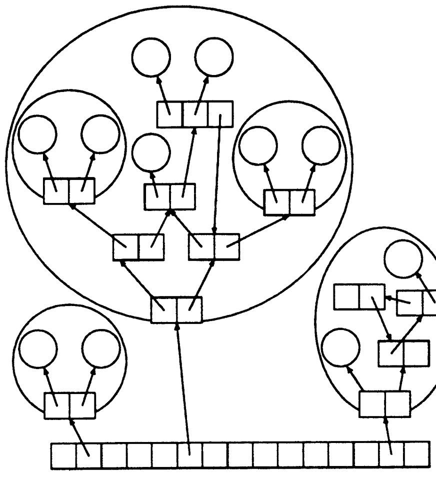


Figure 2

object, and there are no objects which can have a reference to part of the state of an integer object (a reference to some bit).

Programmers would probably choose 'shape' to be a balloon type to obtain 'nice' semantics in its use in programs. It would prevent accidental sharing even if each shape is a complex structure with internal sharing and even cycles, as illustrated in Figure 2.

In the loop example the balloon invariant would make clear to both programmer and compiler the absence of interference between iterations:

```
a:Array[Shape];
for i = 1 to N
  a[i].rotate(45);
```

performing a rotate on a shape `a[i]` would not affect a shape `a[j]` ($j \neq i$). This would make reasoning about the program easier and the compiler could perform loop transformations such as parallelisation.

This figure also illustrates that in spite of the binary classification, both balloons and non-balloons can be used as part of the state of each other. This results in a hierarchical organisation of the graph of objects and in scalability of the control of sharing.

We consider static type checking as the useful thing to do regarding balloon types:

- Whether some type is a balloon type is declared by one keyword (such as `balloon`) in the definition of the type, and no syntactic cost is imposed on client code.
- A candidate implementation of the type undergoes a non-trivial compile-time checking which enforces the run-time invariant for objects of the type. The implementation may be accepted or rejected.

The emphasis is on extreme syntactic simplicity, placing the burden on the compiler. We consider this important for the success of the integration of balloon types in languages and the acceptance by programmers.

3.1 The balloon invariant

We now describe more precisely the run-time invariant which is enforced by balloon types. First some definitions:

Cluster: Let G be the undirected graph with all objects as nodes and all references between non-balloon objects and from balloon to non-balloon objects as edges.

A cluster is a connected subgraph of G which is not contained in a larger connected subgraph.

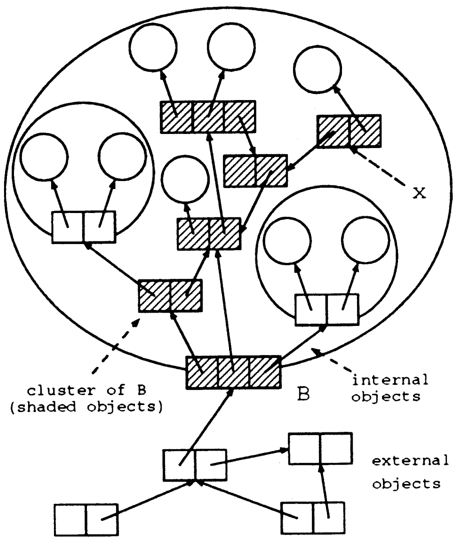


Figure 3

Internal: An object O is said to be *internal* to a balloon object B iff :

1. O is a non-balloon in the same cluster as B or
2. O is a balloon referenced by B or by some non-balloon in the same cluster as B or
3. there exists a balloon B' internal to B and O is internal to B'.

External: An object is said to be *external* to a balloon object B iff it is neither B nor internal to B.

Now we can state the invariant:

Balloon Invariant: If B is an object of a balloon type then:

$I1$ There is at most one reference to B in the set of all objects.

$I2$ This reference (if it exists) is from an object external to B.

$I3$ No object internal to B is referenced by any object external to B.

Figure 3 clarifies these concepts. We should stress that the invariant is concerned with the organisation of the graph of objects; it ignores references in variables from the chain of procedure calls (i.e. stack based).

The invariant deserves some explanation, in particular why internal objects were not simply defined as the objects in the state of the balloon (that is, reachable by the transitive closure of the references relation). With such definition we would have the 'naive invariant'. However it would not be as useful or feasible of being enforced as the chosen invariant.

The reason is that during the execution of some operation of a balloon type several objects may be created. Some of them may be temporary, only referenced by local variables (or other similar objects), and not incorporated into the state of any 'external' object, being subject to garbage collection when the function terminates. The figure shows an object only referenced by a local variable (x).

While they exist these objects may store references to the state of a balloon. This violates the naive invariant as these objects are not part of the state of the balloon but have references to the state. Even if such did not actually happen, the mere possibility of it happening could lead to conservative rejection of code by a checking mechanism. For both these reasons, the naive invariant would make the set of valid programs unnecessarily restricted.

In the balloon invariant such temporary non-balloons are allowed and are classified as internal objects. The state reachable by a balloon object is a subset of the internal objects.

3.2 The Simple Rule

One consequence that can be derived from the invariant is:

$C1$: In the set of objects that make up a cluster there is one balloon object at most.

Furthermore, it can be shown that $(I1 \wedge I2 \wedge C1 \Rightarrow I3)$, which means that the invariant is equivalent to $(I1 \wedge I2 \wedge C1)$. The balloon type checking mechanism enforces this last expression.

$I1$ and $I2$ are enforced by a simple rule, while $C1$ is enforced by means of a static analysis of the candidate program. The simple rule is:

The Simple Rule: A reference to a balloon cannot be stored in any state variable of any object.

Which means that no statement like

```
x.v :- b;
```

is allowed when b is of balloon type. It is important to note that the rule only mentions state variables of objects. This means that stack based variables and state variables of objects are treated differently by the type system.

The rule emphasises the difference between 'temporarily' *using* a reference to an object and *storing* the reference in some state variable of an object. This last case is what creates sharing of objects by other objects, and it is forbidden for balloon types.

The simple rule is enough to enforce $I1$ and $I2$, while allowing great freedom in the use of balloons: a reference to a balloon can be stored in variables, passed as argument to functions and returned from functions. The only case prevented is storing the reference in a state variable of some object. In particular, a function of a balloon type can safely return a reference to an internal balloon and client code can use the reference to invoke operations on it.

As an example consider a dictionary containing elements which can be searched using a key. Here the elements are shapes and the keys are strings. We define a function which invokes a search to locate a shape and then rotates and moves the shape:

```
DictShape = balloon Dictionary[Elem = Shape, Key = String];

rotate_and_move(ds:DictShape, name:String)
{
  s:Shape;
  s :- ds.search(name);
  s.rotate(45);
  s.move(10,15);
}
```

Here both shape and dictionary of shape are balloon types.[4] The simple rule allows the search to safely return a reference to an internal shape of the dictionary, as it will be forbidden to be stored in any object.

Balloon types can be important towards obtaining provably correct programs. The uncontrolled possibility of state sharing in current languages result in unexpected modifications to the state manipulated by the implementation of a data type. Quoting [19]: "If modifications can occur elsewhere, then we cannot establish the correctness of the implementation just by examining its code; for example, we cannot guarantee locally that the representation invariant holds". In balloon types—as opposed to current languages—the data type has complete control: the balloon invariant ensures that the only way a client can gain access to the state of a balloon is by a reference being returned by a function of the data type, as the search function above.

[4] Dictionary here is a generic type. The treatment of bounded polymorphism (as well as subtyping) is beyond the scope of this paper. Informally we can say that it will be more useful that the ballooness is not a property of a generic type itself but of its instantiations. An implementation of a generic type can be type checked to access the correctness of its instantiation as balloon/non-balloon for each possibility regarding the type parameters being or not balloon types.

A function from a balloon type may decide to return a reference to an internal balloon specially if the (composite) value associated with this balloon does not matter for the representation invariant of the data type. Clearly, the contents of shapes are irrelevant to the implementation of the dictionary data type and returning a reference to an internal shape does not prevent reasoning about the correctness of its implementation.

3.3 Copy assignment

The simple rule implies that state variables of balloon type can only be made to reference newly created balloons. There are two cases which should be provided for:

- The creation of an object, using some constructor mechanism. Say:

  ```
  x.shape := Rectangle(10,20,100,130);
  ```

- A general copy mechanism with the semantics of deep-copy (as mentioned in e.g. [17]), which creates a copy of a balloon and all its reachable state (while preserving the internal sharing). It can be provided as a *copy assignment*.

This copy assignment—which we denote by ':=' as opposed to ':-' for reference assignment—is the natural generalisation of the assignment for primitive types; it copies the (composite) value associated with the object. It emphasises 'obtain new object' as opposed to 'reference existing object'. To stress this we have used the ':=' notation in the constructor example above where no physical copy is involved.

We put the emphasis on observable behaviour and reasoning about the program rather than on an implementation directed definition. The copy does not have to happen physically, being subject to optimisation. This contrasts with current languages where it is close to impossible to optimise some built-in deep-copy mechanism. As a result, programmers rarely use it and suffer from unexpected interference or sometimes use it when it is not physically necessary.

Several possibilities for avoiding the deep-copy and performing only a pointer copy include:

- If the balloon which is the source of the assignment is not used subsequently until being 'released'.

- The balloon invariant does not have to hold physically: an implementation can share physically a balloon if that does not affect the outcome of the program. If the balloon remains immutable sharing becomes irrelevant.

- Using a 'copy on update': physically sharing a balloon and only copying it if some operation which causes updates is performed.

Although this kind of optimisations are the norm in functional languages, in current imperative languages the pervading possibility of mutable substructure sharing makes such optimisations unrealistic. The balloon invariant makes these optimisations more realistic. They will be the subject of further research.

3.4 Type checking balloon types

We now give an informal overview of the balloon type checking mechanism. The first point worth emphasising is that the non-trivial mechanism applies to checking the implementations (*classes* in object-oriented terminology) of balloon types:

> In non-balloon classes there are no restrictions other than 'the simple rule' in the use of balloons and no restrictions in the use of non-balloons; non-balloon classes are not subject to static analysis as in the case of current languages.[5]

[5] Interestingly, the simple rule is implicit in current languages, as for the only possible 'balloons'—the primitive types—the assignment has value semantics.

```
balloon Int { operator + (Int): Int; ... }

balloon BigInt
{
private:
  Node // nested declaration of a non-balloon type
  { public: val:Int; nxt:Node; }
  lst:Node; // reference to the head of linked list
public:
  operator + (other:BigInt): BigInt
  {
    p,q,r:Node;
    carry:Int;
    num:BigInt;
    num :- new BigInt;
    num.lst :- new Node;
    r :- num.lst;  p :- self.lst;  q :- other.lst;
    r.val := p.val + q.val;
    if ( ... )
      then carry := 0;
      else carry := 1;
    while (p.nxt and q.nxt) do
      r.nxt :- new Node;
      r :- r.nxt;  p :- p.nxt;  q :- q.nxt;
      r.val := p.val + q.val + carry;
      ... // calc next carry
    ... // traverse the remainder of the largest BigInt
    return num;
  }
  ... // other operations on BigInt
}
```

Figure 4

Each balloon class[6] is subject to a static analysis which will decide its acceptance as the implementation of a balloon type. The essence of the checking mechanism is performing the inductive step of using the assumption that some types are balloons to check the implementation of some balloon type which uses them. Methods of a balloon class can manipulate the state of several instances of that class. Essentially the checking makes sure that, for any possible execution, the several balloons involved will remain balloons; or in other words, the invariant is preserved.

As an example, suppose a data type for arbitrarily large integers is required (**BigInt**). These can be represented by a linked list of plain integers whose size depends on how large the number is. **BigInt** being balloon guarantees that there will not be accidental sharing of parts of linked lists corresponding to different **BigInts**. A fragment of a possible implementation is given in Figure 4.

The implementation manipulates the state of typically three[7] balloon **BigInts**, referenced by **self**, **other** and **num**. Three variables are used to traverse the linked list of **Node** (a non-balloon type). The analysis determines that during all possible executions these variables point to the above mentioned balloons, and that no statement creates sharing of states from any two different balloons. A statement like

[6]Some construct for module of balloon type checking can be introduced (e.g. a set of classes). It can be used to accept more code, being useful if a group of classes cooperate tightly, typically some of them being auxiliary non-balloon classes. The mechanism was developed considering this general case.

[7]It is possible that **self** and **other** refer to the same balloon. The analysis considers this possibility of dynamic aliasing.

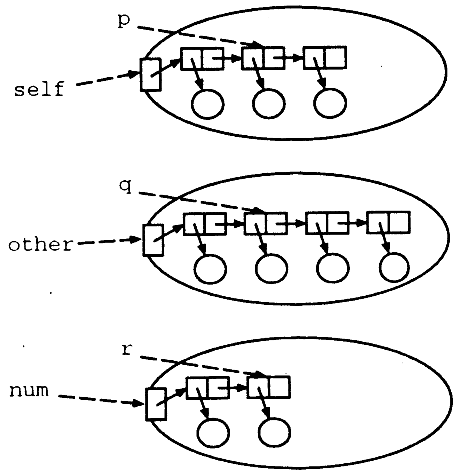


Figure 5

```
r.nxt :- p;
```

would be rejected by the checking mechanism: the analysis would assess that r and p could point to non-balloons 'belonging' to different balloons, and that sharing would be created, breaking the balloon invariant. Note that here the simple rule does not apply because **Node** is a non-balloon type.

Although this is a simple example it serves to illustrate some points. Non-balloons play the main role in the checking mechanism. The essential task of the analysis is to make sure that for accepted programs the consequence $C1$ holds (there is at most 1 balloon object in a cluster). The analysis essentially takes care that:

- non-balloons in clusters containing no balloon (*free* clusters) are prevented from being *captured* by more than one balloon, and
- non-balloons from different clusters which may already have a balloon are prevented from becoming linked (*merging* the clusters).

Although there may exist an arbitrarily large number of objects, in the body of a method there is just a small number of variables (parameters, self and local variables) from which to reach the graph of objects. The key to the analysis is to keep track, for these variables, of the possibility of:

- different variables pointing to objects in the same cluster and
- the cluster to which a variable points containing a balloon.

Variables of a balloon type other than the one being checked can be ignored. In this example **Int** variables are ignored because **Int** is a balloon type. This means that primitive types or other user-defined balloon types can be ignored. This is important because the size of the domains in the analysis grows exponentially with the number of variables in a method (see appendix).

Regarding the **BigInt** example, Figure 5 shows one possible fragment of the object graph at some point in the execution of the '+' function. This scenario would be summarised by the static analysis as (using the notation described in the appendix, and s, o, n as a short for self, other and num):

$$\boxed{sp}\ \boxed{oq}\ \boxed{nr}$$

Which essentially states that:

- if the non-balloon to which p points belongs to a cluster with one balloon, that balloon is the one pointed to by s (and the analogous for (o, q) and (n, r)), and
- s, o and n do not necessarily point to the same balloon.

A detailed presentation can be found in the appendix.

Although that was not so in the **BigInt** example, the analysis considers the general case in which functions of a balloon type can have non-balloon parameters or result. Two restrictions are added to functions which are part of the interface of a balloon class B (public functions); they are forbidden to:

- return a reference to a non-balloon which is internal to some balloon of class B.
- make a balloon of class B capture non-balloons which come as parameters.

A special case of these restrictions is that in a balloon class there can be no public non-balloon state variables—as they are equivalent to a pair of get and set functions. The first restriction is necessary because a reference to a non-balloon can be freely stored in some object by non-balloon code. The second restriction applies because non-balloons that come as parameters may already 'belong' to some balloon (or they may be captured later). The static analysis that a balloon class undergoes enforces these restrictions.

In keeping track of the relevant information the analysis must consider all possible executions of the methods. To achieve this, each balloon class is subjected to a static analysis which includes fixed-point calculations due to loops and mutual recursion in functions of the class. However it uses approximations of functions which are external to the class.

The less restrictive way to enforce the balloon invariant would be by considering information about the implementation of other types while checking some class. We note however that, although such mechanisms may be considered for optimisation purposes, they are unacceptable for type checking:

- Programmers expect to be able to construct modules, have them type checked and use them in other parts of the program. Furthermore, they expect to be able to change the implementation of a module without it affecting the type correctness of the rest of the program—as long as there is no change in the interface of the module.
- Libraries are an essential component and may be provided without source code or even be coded in different languages.

For these reasons, from the moment the idea was conceived (May 1994) the decision was for the checking mechanism to be modular, considering only the interface of other types. The simple rule was essential towards this, by allowing a reference to a balloon to be freely propagated without requiring any form of static analysis.

An advantage of not considering the implementation of other types is that the mechanism can be used in the presence of subtype polymorphism, which is essential in object-oriented languages. This because it is not relevant to know what code will be executed (which due to dynamic dispatch depends on the class of the object); only the type information regarding external functions is used.

The same applies to interfacing with code written in other languages. One example is the primitive types: the implementation of integers may be in assembly, without undergoing the static analysis we developed, but it is enough that integers are declared as balloon types to be considered in type checking user-defined code (as above), as long as the implementor of integers is trusted. This emphasises the importance of the knowledge obtained by making the ability to share state a property of a type, independently of whether the implementation is checked by the mechanism we have developed.

The static analysis was developed first as a global analysis of a program (which is not what intended) on which conservative approximations are made so that it becomes modular. It was developed using abstract interpretation [5], a semantics-based model for static analysis of programs. In the appendix we describe the static analysis. A reader who is not familiar with abstract interpretation can find a modern introduction and further references in [15].

3.5 Opaque balloon types

The main balloon mechanism focuses exclusively on the control of static aliasing—this is deliberate. The idea is that further constraints can be added on top of 'plain' balloons to focus on the control of dynamic aliasing, involving variables from the chain of procedure calls. An important specialisation is the concept of *opaque balloon types.*

Informally, an opaque balloon type has the added invariant that objects do not expose to clients any references to their internal state, even to be used temporarily by variables from some procedure: they represent *truly opaque data abstractions*, which guarantee that all the internal state remains unchanged between invocations of operations from the data type.

As an example, if the dictionary of shapes was an opaque balloon type, client code could not obtain a reference to a shape in the dictionary in order to update it in-place. It would only be possible to obtain a copy of a shape, which could be stored or operated upon with no effect on the shape in the dictionary.

Typical examples of opaque balloon types are the primitive types like integers. They are not simply balloon types, but opaque balloon types: there cannot be any reference to internal state of an integer (some bit) neither in external objects nor in variables from client code.

In terms of language use the situation is analogous to plain balloons: opaque balloons are declared using a keyword (such as **opaque**), and a candidate implementation undergoes a static check (on top of the plain balloon checking) in order to be accepted or rejected.[8]

3.6 Value types

Typical primitive types like integer or boolean, are something more than just opaque balloon types; they can be said to be *value types*:[9] an integer variable is associated with an integer value. The important property for reasoning about programs using integer variables is that the value associated with a variable cannot change as a result of operations on other variables.

In object-oriented languages this property holds for primitive types because a variable contains an atomic representation of the value (as opposed to a reference to a possibly shared object) and parameter passing copies the value. However, if we are to be able to define value types in general (eg. for stacks or sets), the value will have to be represented by a group of objects and it is not possible for a variable to 'contain the value'. An example of this is the failure of expanded types in Eiffel to provide appropriate support for value types: by not being able to prevent sharing of linked substructures (as discussed above) they can lead to subtle interference, unlike in primitive types.

What is important is, not that the variable physically contains the object, which can be seen 'only' as an implementation issue, but that the group of objects that represent the value can only be accessed by the variable. This happens if:

- we have an opaque balloon type, and
- the variable holds the only reference to the opaque balloon object.

This means we can have the concept of value types as a (slight) specialisation of opaque balloon types. Value types will be declared by some keyword (such as **value**), and on top of the opaque balloon type checking, for value types the use of assignment or invocation with reference semantics will be forbidden in client code, being only allowed (deep) copy operations.

It is important to note that these copy operations are conceptual; they do not necessarily have to be performed physically. In fact, being only allowed assignment and parameter passing with copy semantics, the group of objects which constitute the representation of a value will remain immutable after being computed and returned by some function in the implementation of the data type. This means that they can be physically shared and all conceptual deep copies in assignment or parameter passing in client code will be implemented as simple pointer copies.

[8] The exact definition of opaque balloon types and the respective checking mechanism are more involved than may be thought at first, and are beyond the scope of this paper.

[9] [16] also uses this term and discusses problems with expanded types in Eiffel.

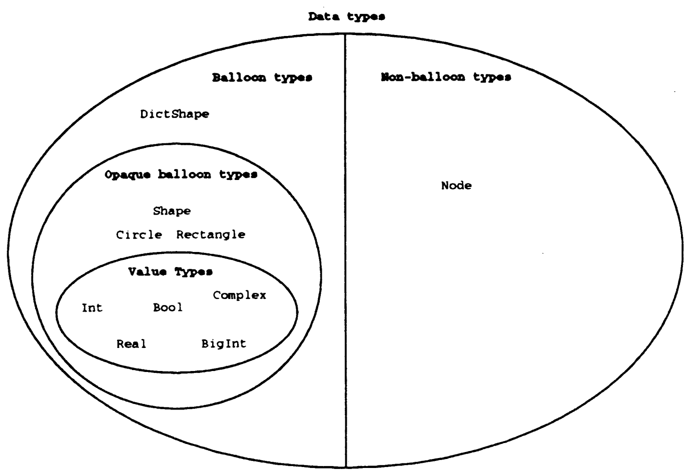


Figure 6: Taxonomy regarding state sharing in data types

In what concerns the implementation of a value type, if a function does not modify an object passed as parameter, the corresponding conceptual copy can also be optimised away. In the **BigInt** example no deep copies would exist at all. In other cases, even if some copy is needed, internal objects of value type can be physically shared, which means the copy does not need to be fully deep.

These situations are analogous to what happens in the implementation of functional languages. Indeed, user-defined value types can contribute towards bridging the gap between imperative and functional languages, concerning both programming styles and language implementation techniques.

Currently we have a gap between primitive types and user-defined types. With value types, the traditional behaviour concerning interference exhibited by traditional primitive types can be obtained in user-defined types if desired. This means primitive types can truly cease to be 'special'.

We finish by presenting in Figure 6 the taxonomy of data types with respect to state sharing: the main binary classification in balloon/non-balloon and the two specialisations of balloon types. The figure also shows examples of each case.

4 Related work

In [13] a taxonomy regarding the treatment of aliasing is described. There has been a lot of research on *alias detection*, and the interest has been progressively shifting towards dynamic data structures, two recent examples being [7, 25].

Much less has been done on *alias prevention*. A classic example which includes alias prevention for named objects is Reynolds' Syntactic Control of Interference [24].

A recent example regarding unnamed objects is [23]: it mentions unshareable objects (there can only be one pointer to them in the system) and unique pointers (from which assignment copies the pointer and nullifies the variable, i.e. moves the pointer). However, this proposal does not address substructure sharing: even if there is only one pointer to an u-object, it does not prevent sharing of objects reachable by the u-object. This means the usual subtle possibilities of interference can occur.

The principle of *design for analysability* [9] is interesting, and our work can also be seen as an example of this principle. We present now the research on alias prevention for dynamic data structures we found more related to balloon types.

Euclid's Collections

The problem of aliasing was considered in the design of the language Euclid [18]. Pointer variables are declared as being pointers to a collection. A dynamically allocated object must be an element of a collection and it is enforced that pointers to different collections point to different objects.

The serious limitation is that collections are named entities like local variables of some procedure: it is not possible to have a collection as a member of a dynamically allocated object. This means that neither the number of collections can be related to the size of the object graph nor it is possible to obtain a hierarchical structure. As a result, collections are not appropriate as a means of organising the object graph.

FX's Regions

[20] describes an "effect system" with three base kinds: types, effects and regions. Effects describe the possible side-effects of an expression and regions describe the area of the store where those side-effects may occur. The effect system computes statically a conservative approximation of the actual side-effects that an expression may have. Regions are similar to collections and although more powerful they suffer from the same problem: they are named entities.

Islands

John Hogg's *islands* [12] proposal is more closely related to balloons. It represents an advance because, unlike collections or regions, islands impose a structure on the object graph without using named entities as units. This enables hierarchical structures to be described.

Similar to a balloon, an island defines a cluster of objects to which there are no references stored in external objects. There are however not only some differences in the invariants enforced, but major differences in the mechanisms:

- Islands have a considerable syntactic cost by requiring *access modes* to variables to be spread both in the signatures of all the methods in a class and in client code.

  In balloons, one keyword in the definition of the type summarises the concept they represent, and no syntactic cost is imposed on client code.

- The detection by the compiler of whether a class is a *bridge* (entry to an island) is purely syntactic according to the access modes.

  The use of a static analysis in balloons enabled minimising restrictions while maintaining syntactic simplicity.

- The knowledge that class A is a bridge is not used to help in assessing whether a class B (which uses A) is also a bridge. The knowledge that a class is a bridge is something that does not affect what code can be written (more than the individual signatures of the operations provided), and can be determined *a posteriori*, after the program is written.

  Balloons are a full part of the type system: whether type T is balloon or not affects what code can be written. Although we rely on static analysis, the mechanism is not an *a posteriori* alias analysis of a program. Using the knowledge that type T is balloon is essential in checking the implementation of a balloon type which uses T. The use of induction is the essence of the balloon checking mechanism.

5 Conclusions

Although there has been much research on aliasing, most has been devoted to analysis techniques towards some optimisation: aliasing tends to be considered for implementation purposes and not as a first class entity in a type system.

We have presented a mechanism which makes the ability to share state a first class property of a data type. Balloon types is an extension to programming languages which permits expressing and enforcing that all state reachable by an object is encapsulated: no external objects are allowed to have references to any object that is part of that state.

It allows designers of the data type to truly control what state is made available to be operated upon by the client, users of the data type to have confidence that the more pervasive form of interference does not occur and compilers to explore this knowledge to perform automatic program transformations.

The syntactic simplicity reflects the essence of the concept: a classification of data types according to the ability to share state. By the use of a static analysis as the checking mechanism it was possible to achieve this syntactic simplicity while avoiding imposing restrictions, otherwise unavoidable if the verification was purely syntactic.

The knowledge given by the balloon invariant can be seen to have wide usefulness. Some examples are program transformation for parallelisation or execution reordering and garbage collection. Being self-contained composite objects, balloons are suitable as units of placement/copy/migration; this is specially important in distributed systems.

Balloon types also constitute a valuable support for the development of large scale provably correct programs. Two specialisations of balloon types can be particularly important. Opaque balloon types represent truly opaque data abstractions. They are also the key to obtaining value types. These bridge the current gap between primitive and user-defined types: primitive types can truly cease to be 'special'.

Acknowledgements

I thank Paul Kelly, Sophia Drossopoulou, and Sue Eisenbach for their comments to this paper.

References

[1] G. E. Andrews. *The Theory of Partitions. Encyclopedia of Mathematics and its Applications (Vol 2).* Addison-Wesley, 1976.

[2] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[3] Franco Civello. Roles for composite objects in object-oriented analysis and design. In *Proceedings OOPSLA '93*, pages 376–393, October 1993. SIGPLAN Notices, volume 28, number 10.

[4] Chris Clack and Simon Peyton Jones. Strictness analysis—a practical approach. In *FPCA '85*, volume 201 of *LNCS*, pages 35–49. Springer-Verlag, September 1985.

[5] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages (POPL'77)*, pages 238-252, January 1977.

[6] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. The Simula 67 common base language. Publication S-22, Norwegian Computing Center, Oslo, 1970.

[7] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Conference Record of the 23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 1–15, January 21–24, 1996.

[8] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[9] L. J. Hendren and G. R. Gao. Designing programming languages for analyzability: A fresh look at pointer data structures. In *Proceedings of the 4th IEEE International Conference on Computer Languages*, pages 242–251, April 1992.

[10] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[11] C. A. R. Hoare. Hints on programming language design. Technical Report STAN//CS-TR-73-403, Stanford University, Department of Computer Science, December 1973. Based on a keynote address presented at the ACM Symposium on Principles of Programming Languages.

[12] John Hogg. Islands: Aliasing protection in object-oriented languages. In *Proceedings OOPSLA '91*, pages 271–285, November 1991. SIGPLAN Notices, volume 26, number 11.

[13] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The Geneva convention on the treatment of object aliasing. Followup report on ECOOP '91 workshop W3: Object-oriented formal methods. *OOPS Messenger*, 3(2):11–16, April 1992.

[14] Sebastian Hunt. Frontiers and open sets in abstract interpretation. In *FPCA'89*, pages 1–11. ACM Press, September 1989.

[15] Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of Logic in Computer Science (Vol 4, Semantic Modelling)*, pages 527–636. Oxford University Press, 1995.

[16] S. Kent and J. Howse. Value types in eiffel. In *Proceedings of TOOLS Europe 96 (TOOLS 19)*. Prentice Hall, 1996.

[17] S. Khoshafian and G. Copeland. Object identity. In *Proceedings OOPSLA '86*, November 1986. SIGPLAN Notices, volume 21, number 11.

[18] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek. Report on the programming language EUCLID. *SIGPLAN Notices*, 12(2), 1977.

[19] Barbara H. Liskov and John Guttag. *Abstraction and Specification in Program Development*. The MIT Press, 1986.

[20] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pages 47–57, January 13-15 1988.

[21] B. J. MacLennan. Values and objects in programming languages. *SIGPLAN Notices*, 17(12):70–79, December 1982.

[22] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[23] Naftaly Minsky. Towards alias-free pointers. In *Proceedings ECOOP '96*, LNCS 1098, pages 189–209. Springer-Verlag, July 1996.

[24] J. C. Reynolds. Syntactic control of interference. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 39–46, January 1978.

[25] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Conference Record of the 23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 16–31, January 21–24, 1996.

[26] B. Stroustrup. *The C++ programming language*. Addison-Wesley Publishing Company, 1986.

[27] Peter Wegner. Dimensions of object-based language design. In *Proceedings OOPSLA '87*, pages 168–182, December 1987. SIGPLAN Notices, volume 22, number 12.

[28] N. Wirth. The programming language Pascal. *Acta Informatica*, 1:35–63, 1971.

$$
\begin{aligned}
op &::= + \mid - \mid * \mid / \mid = \mid < \mid > \\
e &::= n \mid x\ op\ y \mid \mathtt{isnull}(x) \\
asgn &::= x\ :-\ y \mid x\ :-\ y.z \mid x\ :-\ \mathtt{null} \mid x.y\ :-\ z \mid x.y\ :-\ \mathtt{null} \mid x\ :-\ \mathtt{new} \mid \\
&\quad\ x.y\ :-\ \mathtt{new} \mid x\ :=\ e \\
c &::= asgn \mid c_0\ ;\ c_1 \mid \mathtt{if}\ e\ \mathtt{then}\ c_0\ \mathtt{else}\ c_1 \mid \mathtt{while}\ e\ \mathtt{do}\ c \mid x\ :-\ f_i(x_1,\ \ldots,\ x_{a_i}) \\
fdecl &::= f_1(x_{11},\ \ldots,\ x_{1a_1})\ \mathtt{do}\ c_1\ \mathtt{return}\ y_1 \\
&\qquad\vdots \\
&\quad\ f_k(x_{k1},\ \ldots,\ x_{ka_k})\ \mathtt{do}\ c_k\ \mathtt{return}\ y_k \\
prg &::= fdecl\ \mathtt{do}\ c
\end{aligned}
$$

Figure 7: Abstract syntax of RISO

A Static Analysis

We now present the static analysis on which the balloon checking mechanism is based. We define a small language, a denotational semantics and finally present the abstract interpretation.

It is presented as a global analysis from which a modular one can be obtained by using a conservative approximation in the function environment in what concerns functions from other modules than the one being analysed. The constraints in the public functions of a balloon type can be counted upon to obtain an approximation.

A.1 RISO—an imperative language with recursive definition of functions and shared objects

We now define a simple language that models accurately both the possibility of several variables referring to the same object (dynamic aliasing) and the sharing of objects by state variables of other objects (static aliasing). This is accomplished by making every variable or state variable a reference to a possibly shared object. Integer variables do not receive a special treatment, being also references to possibly shared integer objects.

The abstract syntax is given in Figure 7. We use x, y and z—ranging over a set of identifiers—for identifiers of both variables and (object) state variables; the traditional dot notation to access state variables; `null` for the null reference; `isnull` for the test for a null reference; `new` for the creation of objects (including integer objects which are initialised to zero) and '`:-`' for the reference assignment.

We also have the operator '`:=`' for performing updates on integer objects. Integer variables are references to an integer and '`:-`' does not modify the integer object but just makes the variable reference some integer object.

Commands can be an assignment, sequence, conditional, loop and function invocation. In an invocation both paramenter passing and the return of the result have the semantics of a reference assignment ('`:-`'). This is the interesting—and problem causing—case. A program is a (possibly mutually recursive) declaration of functions followed by the 'main' command.

A.2 Denotational semantics of RISO

We developed a denotational semantics that corresponds to what was described informally in the previous section. The denotational style was chosen because it is described in a compositional way by structural induction—like the static analysis we have developed—making it appropriate to serve as the standard semantics on which to base the abstract interpretation.

N	flat pre-cpo of numbers
I	finite flat pre-cpo of identifiers
A	finite flat cpo of addresses
	($\bot_A$ denotes the null address)
$V = [I \rightarrow A]$	cpo of variable mappings
	$\bot_V$ denotes the null mapping
$O = (N + V)_\bot$	cpo of object values
	($\bot_O$ denotes the undefined object)
$G = [A \rightarrow_s O]$	cpo of graphs of objects
	($\forall g \in G : g(\bot_A) = \bot_O$)
	$\bot_G$ denotes the null graph
$S = \overline{G \times V}$	flat pre-cpo, 'the state'
$F = F_1 \times \cdots \times F_k$	cpo of function environments,
	$F_i = [\overline{G \times A^{a_i}} \rightarrow (\overline{G \times A})_\bot]$

Figure 8: Semantic domains

This semantics is quite interesting in itself as models accurately both 'heap allocated' objects and recursive definitions of functions, being suitable to be adapted to real imperative languages. The more important aspect is that 'the state' has two components:

- one is a mapping from variables to addresses,[10]
- the other is a mapping from addresses to object values, defining the object graph. An object value can be an integer or a record, the latter represented by a mapping from (state) variables to addresses.

To simplify the description of both denotational semantics and abstract interpretation, we assume a simple type checking of a type annotated version of RISO has been performed which produces:

- a set of object types T, corresponding these to either Int or record types,
- I: the set of variable identifiers in the program,
- a mapping typeof: $I \rightarrow T$,[11]
- a predicate balloon: $T \rightarrow \{\text{true}, \text{false}\}$, which corresponds to the (still unchecked) annotation stated by the programmer, and such that balloon Int = true),
- a program free of annotations, with the above described abstract syntax and which is type correct in the simple sense that: types are compatible in assignments and functions invocation, and for identifiers x used in expressions typeof x = Int.

The semantic domains are given in Figure 8. In the representation of object graphs addresses not in use are mapped to the undefined object ($\bot_O$). In the representation of a record the identifiers which are not part of the record remain mapped to the null address ($\bot_A$). This enables us to work with total mappings.

A subtle point is the flatness of S and of the domain and codomain in each F_i, instead of having the standard coordinatewise order. Otherwise most functions would not be monotone, and hence continuous.

Figure 9 lists the semantic functions. There is a function for each corresponding syntactic set in the abstract syntax. This factors similar cases, which helps in keeping down the size of the function definitions.

[10]We have chosen the term *address* without implying that it corresponds to physical addresses in some implementation. Others may prefer the term *object ID*.

[11]To simplify the description, and without loss of generality, we have assumed that a given identifier cannot be used for different object types in different parts of a program.

$$\begin{aligned}
&\mathcal{O}: Op \to (N_\perp \times N_\perp) \to N_\perp\\
&\mathcal{E}: Exp \to S \to N_\perp\\
&\mathcal{A}: Asgn \to S \to S_\perp\\
&\mathcal{C}: Com \to F \to S \to S_\perp\\
&\mathcal{F}: Fdecl \to F\\
&\mathcal{P}: Prg \to S_\perp
\end{aligned}$$

Figure 9: Semantic functions

We use the functions $\mathcal{A}$ for the atomic commands (the assignments) and $\mathcal{C}$ for general commands (assignment, sequence, conditional, loop and function invocation).

Using also $\mathcal{A}$ instead of just $\mathcal{C}$ gives emphasis to the actual atomic actions (as opposed to combinations of actions) and avoids the need for a function environment in describing the semantics of the atomic actions. Such function environment is necessary in the case of general commands which may involve function invocations.

$\mathcal{F}$ is a function from a declaration of functions to a function environment. Finally $\mathcal{P}$ maps a program to its denotation: the resulting state of executing the given command starting from the null graph of objects and null variable mapping, with the function environment as given by the declaration.

The definition of the semantic functions is given in Figure 10.

A.3 Abstract interpretation

From the set of concrete states S, we will use S_b for the set of states in which the balloon invariant holds (valid states). Each element of S_b is abstracted into an element of a finite set C. Elements of C have the form $(p, b) \in \wp(I \times I) \times (I \to \{0, 1\})$ such that:

- p is an equivalence relation on the set of identifiers I; defining a partition according to what identifiers reference objects in the same cluster.
- b is a function from equivalence classes to $\{0, 1\}$; representing the number of balloons in the cluster corresponding to the given equivalence class. b is presented as a function with domain I and the invariant $(x, y) \in p \Rightarrow bx = by$.

All the characterisation of concrete states that is relevant to the presentation can be given by two functions:[12]

- $P: S_b \to \wp(I \times I)$ maps a valid state to an equivalence relation on I. $(x, y) \in Ps$ iff in the state s, x and y reference objects in the same cluster.
- $B: S_b \times \wp(I) \to N$ gives the number of balloons in all clusters referenced by the given set of identifiers in the given state.

The abstraction[13] function $\alpha: S_b \to C$ is defined in a straightforward manner in terms of P and B:

$$\alpha = \lambda s.(Ps, \lambda x.B(s, \{x\}))$$

While a concrete state in S_b is abstracted to a single abstract state in C, an element of C represents a set of concrete states. For a given abstract state (p, b):

[12] These two functions serve just presentation purposes and are not used in the actual static analysis.

[13] The pair of abstraction and concretisation functions is not presented as an adjoined pair; however they play a similar role.

$$
\begin{array}{lcl}
\mathcal{E}\llbracket n \rrbracket & = & \lambda(g,v).\lfloor n \rfloor \\
\mathcal{E}\llbracket x \; op \; y \rrbracket & = & \lambda(g,v).\mathcal{O}\llbracket op \rrbracket(g(vx), g(vy)) \\
\mathcal{E}\llbracket \texttt{isnull}(x) \rrbracket & = & \lambda(g,v).\begin{cases} \lfloor 0 \rfloor & \text{if } vx = \bot_A, \\ \lfloor 1 \rfloor & \text{otherwise.} \end{cases} \\
\mathcal{A}\llbracket x :- y \rrbracket & = & \lambda(g,v).\lfloor g, v[x \mapsto vy] \rfloor \\
\mathcal{A}\llbracket x :- y.z \rrbracket & = & \lambda(g,v).\,\text{let}\, a \Leftarrow vy.\lfloor g, v[x \mapsto g\lfloor a \rfloor z] \rfloor \\
\mathcal{A}\llbracket x :- \texttt{null} \rrbracket & = & \lambda(g,v).\lfloor g, v[x \mapsto \bot_A] \rfloor \\
\mathcal{A}\llbracket x.y :- z \rrbracket & = & \lambda(g,v).\,\text{let}\, a \Leftarrow vx.\lfloor g[\lfloor a \rfloor \mapsto g\lfloor a \rfloor[y \mapsto vz]], v \rfloor \\
\mathcal{A}\llbracket x.y :- \texttt{null} \rrbracket & = & \lambda(g,v).\,\text{let}\, a \Leftarrow vx.\lfloor g[\lfloor a \rfloor \mapsto g\lfloor a \rfloor[y \mapsto \bot_A]], v \rfloor \\
\mathcal{A}\llbracket x :- \texttt{new} \rrbracket & = & \lambda(g,v).\,\text{let}\, a \Leftarrow \text{alloc}\, g.\lfloor g[\lfloor a \rfloor \mapsto o], v[x \mapsto \lfloor a \rfloor] \rfloor \\
 & & \quad \text{where } o = \begin{cases} \lfloor 0 \rfloor & \text{if typeof } x = \text{Int}, \\ \lfloor \bot_V \rfloor & \text{otherwise.} \end{cases} \\
\mathcal{A}\llbracket x.y :- \texttt{new} \rrbracket & = & \lambda(g,v).\,\text{let}\, a_x \Leftarrow vx.\,\text{let}\, a_n \Leftarrow \text{alloc}\, g.\lfloor g[\lfloor a_n \rfloor \mapsto o][\lfloor a_x \rfloor \mapsto g\lfloor a_x \rfloor[y \mapsto \lfloor a_n \rfloor]], v \rfloor \\
 & & \quad \text{where } o = \begin{cases} \lfloor 0 \rfloor & \text{if typeof } y = \text{Int}, \\ \lfloor \bot_V \rfloor & \text{otherwise.} \end{cases} \\
\mathcal{A}\llbracket x := e \rrbracket & = & \lambda(g,v).\,\text{let}\, a \Leftarrow vx.\,\text{let}\, i \Leftarrow \mathcal{E}\llbracket e \rrbracket(g,v).\lfloor g[\lfloor a \rfloor \mapsto \lfloor i \rfloor], v \rfloor \\
\mathcal{C}\llbracket asgn \rrbracket & = & \lambda\varphi.\lambda(g,v).\mathcal{A}\llbracket asgn \rrbracket(g,v) \\
\mathcal{C}\llbracket c_0; c_1 \rrbracket & = & \lambda\varphi.\lambda(g,v).\,\text{let}(g',v') \Leftarrow \mathcal{C}\llbracket c_0 \rrbracket\varphi(g,v).\mathcal{C}\llbracket c_1 \rrbracket\varphi(g',v') \\
\mathcal{C}\llbracket \texttt{if}\, e\, \texttt{then}\, c_0\, \texttt{else}\, c_1 \rrbracket & = & \lambda\varphi.\lambda(g,v).\,\text{let}\, i \Leftarrow \mathcal{E}\llbracket e \rrbracket(g,v).\begin{cases} \mathcal{C}\llbracket c_0 \rrbracket\varphi(g,v) & \text{if } i = 0, \\ \mathcal{C}\llbracket c_1 \rrbracket\varphi(g,v) & \text{otherwise.} \end{cases} \\
\mathcal{C}\llbracket \texttt{while}\, e\, \texttt{do}\, c \rrbracket & = & \lambda\varphi.\,\text{fix}\,\lambda h.\lambda(g,v).\,\text{let}\, i \Leftarrow \mathcal{E}\llbracket e \rrbracket(g,v).\begin{cases} \text{let}(g',v') \Leftarrow \mathcal{C}\llbracket c \rrbracket\varphi(g,v).h(g',v') & \text{if } i = 0, \\ \lfloor g, v \rfloor & \text{otherwise.} \end{cases} \\
\mathcal{C}\llbracket x :- f_i(x_1, \ldots, x_{a_i}) \rrbracket & = & \lambda\varphi.\lambda(g,v).\,\text{let}(g',r) \Leftarrow \varphi_i(g, vx_1, \ldots, vx_{a_i}).\lfloor g', v[x \mapsto r] \rfloor \\
\mathcal{F}\llbracket fdecl \rrbracket & = & \text{fix}\,\lambda\varphi.(\varphi'_1, \ldots, \varphi'_k) \text{ where} \\
 & & \varphi'_i = \lambda(g, r_1, \ldots, r_{a_i}).\,\text{let}(g',v) \Leftarrow \mathcal{C}\llbracket c_i \rrbracket\varphi(g, \bot_v[x_{i1} \mapsto r_1, \ldots, x_{ia_i} \mapsto r_{a_i}]).\lfloor g', vy_i \rfloor \\
 & & (fdecl \text{ as given by the abstract syntax}) \\
\mathcal{P}\llbracket fdecl\, \texttt{do}\, c \rrbracket & = & \mathcal{C}\llbracket c \rrbracket\mathcal{F}\llbracket fdecl \rrbracket(\bot_G, \bot_V) \\
 & & \\
\text{Where} & & \text{alloc}\colon G \to A \text{ is a function such that:} \\
 & & \forall g \in G.((\forall a \in A \setminus \bot_A.ga \neq \bot_O) \Rightarrow \text{alloc}\, g = \bot_A) \\
 & & \qquad \wedge((\exists a \neq \bot_A.ga = \bot_O) \Rightarrow \text{alloc}\, g \neq \bot_A \wedge g(\text{alloc}\, g) = \bot_O)
\end{array}
$$

Figure 10: Semantic function definitions

- If x and y are not both mapped to 1 by b, and are not in the same equivalence class (as defined by p), then in the corresponding concrete states x and y *definitely* do not reference objects in the same cluster.

  If they are both mapped to 1 by b or belong to the same equivalence class, then nothing can be assumed: they *may or may not* reference objects in the same cluster.

- If $bx = 0$ it means that *definitely* there is no balloon in the cluster referenced by x. If $bx = 1$ it means that there *may* exist one balloon in the cluster referenced by x; this is included in:

- There is at most one balloon in the union of all clusters referenced by the set of identifiers in an equivalence class which is mapped to 1 by b.

This is given by the concretisation function $\gamma\colon C \to \wp(S_b)$:

$$
\begin{array}{l}
\gamma = \lambda(p,b).\{s \in S_b \mid \forall x, y \in I. \\
\qquad ((bx = 0 \vee by = 0) \wedge (x,y) \notin p \Rightarrow (x,y) \notin Ps) \\
\qquad\quad \wedge\, bx \geq B(s, px)\}
\end{array}
$$

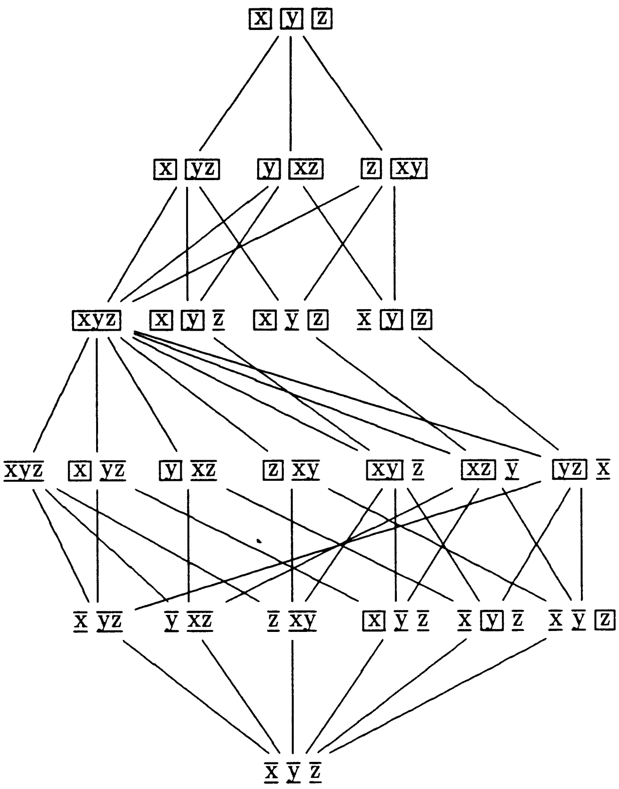


Figure 11: The $C_{\{x,y,z\}}$ cpo

where we use the notation px for $\{y \mid (x,y) \in p\}$.

States in S in which the invariant does not hold (invalid states) are considered by adding a $\top$ to C, representing all states in S—both valid and invalid states.

The abstraction and concretisation functions are extended to $\alpha: S \rightarrow C_\top$ and $\gamma: C_\top \rightarrow \wp(S)$ by making $\alpha(s) = \top$ if $s \notin S_b$, and $\gamma(\top) = S$.

The pair of functions α and γ satisfy the expected property that the abstraction of a given concrete state s represents a set of concrete states which include s:

$$\forall s \in S . s \in \gamma(\alpha s)$$

γ induces a partial order on C such that γ becomes an order-embedding of C into $\wp(S_b)$:

$$\forall c_1, c_2 \in C.(c_1 \leq c_2 \iff \gamma c_1 \subseteq \gamma c_2)$$

This order is:

$$\begin{aligned}(p_1, b_1) \leq (p_2, b_2) \iff & (b_1 \leq b_2) \wedge \forall x, y \in I. \\ & ((x,y) \in p_1 \wedge (x,y) \notin p_2 \Rightarrow b_2 x = 1 \wedge b_2 y = 1) \\ & \wedge (b_2 x = 1 \wedge (x,y) \in p_2 \Rightarrow (b_1 x = 1 \wedge b_1 y = 1 \Rightarrow (x,y) \in p_1))\end{aligned}$$

Figure 11 shows the C cpo in the case when $I = \{x, y, z\}$. We use a graphic notation to refer to elements of C. An element (p, b) represented as $\boxed{xy}\ \underline{z}$ means that there are two equivalence classes defined by p—$\{x, y\}$ and $\{z\}$—and that b maps $\{x, y\}$ to 1, and $\{z\}$ to 0. With the given order we have for example: $\boxed{x}\ \underline{y}\ \underline{z} \leq \boxed{xy}\ \underline{z}$ and $\boxed{xy}\ \underline{z} \leq \boxed{x}\ \boxed{y}\ \underline{z}$.

The first observation we make is that in general C is not a lattice (although that is the case when I has just one or two elements). Nevertheless we can use C as the set of abstract states in which to describe the effects of the simple actions (the assignments).

$$\begin{aligned}
&\mathcal{A}^a\colon Asgn \to C \to C_\top \\
&\mathcal{C}^a\colon Com \to D^k \to D \to D \\
&\mathcal{F}^a\colon Fdecl \to D^k \\
&\mathcal{P}^a\colon Prg \to D \\
&D = \mathcal{O}(C_\top) \setminus \{\emptyset\}
\end{aligned}$$

Figure 12: Abstract semantic functions

For general commands we must have a complete lattice as there is the need for joins and fixed-points. For this we use a completion of C: the lattice of order ideals (down-sets) $\mathcal{O}(C_\top) \setminus \{\emptyset\}$. Although we use down-sets to simplify the presentation, an implementation needs only manipulate their maximal elements, similarly to the frontier representation of a function [4, 14]. Figure 12 lists the abstract semantic functions, while their definition is given in Figure 13.

It is straightforward to see that for any program the analysis terminates in finite time. That is so because the fixed point calculations which occur for the loop command and the function environment are performed on finite lattices.

The interpretation can be proven to be safe in the sense of:

$$\forall p \in \mathrm{Prg}.\mathcal{P}[\![p]\!] = \bot_S \vee \mathcal{P}[\![p]\!] \in \bigcup\{\gamma(c) | c \in \mathrm{Max}\,\mathcal{P}^a[\![p]\!]\}$$

That is, a program may fail to terminate—something which the abstract interpretation cannot determine (halting problem)—but if it terminates in a given state, it will be one of the states represented by the result of the abstract interpretation of the program.

The safety is straightforward to prove in all cases except what concerns function invocations and the way a function environment is abstracted. This because a concrete function is abstracted—not to a function on abstract states—but simply to an element of D.

It can however be proven by fixed-point induction (together with an induction on the structure of a function, essentially on commands) that the way we abstract a function gives the same result as if we used the naive function space on abstract states.

It is easy to understand why this is so. We abstract a function to the response of the body of the function to a special input—what we call the unit state (**1**); this is the input which (from the possible inputs to the function) contains more information. We use the property that from the response of the function to this input it is possible to compute what would be the response to any other input.

This property is due to the following: an invocation does not change the variables which are arguments[14] but the objects referenced. The possible effects of an invocation on the abstract calling environment are to merge or to capture equivalence classes; they just 'add' sharing. As such, from the effect of the function to the more precise input (**1**)—where all identifiers are in separate equivalence classes, and the non-balloon identifiers are free—it is possible to compute the exact[15] response to any other input. The property does not hold for commands in general, as reference assignments to variables make the corresponding identifier 'move' to a different equivalence class.

This property, which makes it possible to avoid the use of a function space, is very important; otherwise the whole analysis would be intractable as the size of the C_I grows exponentially with the size of I; more precisely:

$$|C_I| = \sum_{k=0}^{n} \binom{n}{k} B_k B_{n-k}$$

[14] Due to this and to simplify the presentation we assume formal parameters are constant references—they do not suffer reference assignments in the function body. This is no real restriction as it is always possible to copy the reference to a local variable and use the variable instead of the paramenter.

[15] Here we mean exact when compared to substituting the function body for the invocation and performing the analysis; that is, no further conservative approximation is added.

$$
\begin{array}{lcl}
\mathcal{A}^a\llbracket x :\!- y\rrbracket & = & \lambda c.c[x \rhd y] \\
\mathcal{A}^a\llbracket x :\!- y.z\rrbracket & = & \lambda c.\begin{cases} c[x \rhd] & \text{if balloon(typeof } x\text{)}, \\ c[x \rhd y] & \text{otherwise.} \end{cases} \\
\mathcal{A}^a\llbracket x :\!- \mathtt{null}\rrbracket & = & \lambda c.c[x \rhd] \\
\mathcal{A}^a\llbracket x.y :\!- z\rrbracket & = & \lambda(p,b).\begin{cases} \top & \text{if balloon(typeof } z\text{)}, \\ \top & \text{if } (x,z) \notin p \wedge bx = 1 \wedge bz = 1, \\ (p,b)[x \rhd\!\lhd z] & \text{otherwise.} \end{cases} \\
\mathcal{A}^a\llbracket x.y :\!- \mathtt{null}\rrbracket & = & \lambda c.c \\
\mathcal{A}^a\llbracket x :\!- \mathtt{new}\rrbracket & = & \lambda c.c[x \rhd] \\
\mathcal{A}^a\llbracket x.y :\!- \mathtt{new}\rrbracket & = & \lambda c.c \\
\mathcal{A}^a\llbracket x := e\rrbracket & = & \lambda c.c \\
\mathcal{C}^a\llbracket asgn\rrbracket & = & \lambda\varphi.\lambda o.\begin{cases} \top & \text{if } o = \top, \\ \bigsqcup_{c \in \mathrm{Max}\, o} \downarrow\mathcal{A}^a\llbracket asgn\rrbracket c & \text{otherwise.} \end{cases} \\
\mathcal{C}^a\llbracket c_0; c_1\rrbracket & = & \lambda\varphi.\lambda o.\mathcal{C}^a\llbracket c_1\rrbracket\varphi(\mathcal{C}^a\llbracket c_0\rrbracket\varphi o) \\
\mathcal{C}^a\llbracket \mathtt{if}\, e\, \mathtt{then}\, c_0\, \mathtt{else}\, c_1\rrbracket & = & \lambda\varphi.\lambda o.\mathcal{C}^a\llbracket c_0\rrbracket\varphi o \sqcup \mathcal{C}^a\llbracket c_1\rrbracket\varphi o \\
\mathcal{C}^a\llbracket \mathtt{while}\, e\, \mathbf{do}\, c\rrbracket & = & \lambda\varphi.\,\mathrm{fix}\,\lambda h.\lambda o.h(\mathcal{C}^a\llbracket c\rrbracket\varphi o) \sqcup o \\
\mathcal{C}^a\llbracket x :\!- f_i(x_1,\ldots,x_{a_i})\rrbracket & = & \lambda\varphi.\lambda o.\begin{cases} \top & \text{if } o = \top \vee \varphi_i = \top, \\ \bigsqcup_{c_a \in \mathrm{Max}\, o}\ \bigsqcup_{c_f \in \mathrm{Max}\, o_f} \downarrow \mathrm{apply}(c_f, c_a[x \rhd]) & \text{otherwise.} \end{cases} \\
 & & \quad \text{where } o_f = \varphi_i[x, x_1, \ldots, x_{a_i}/y_i, x_{i1}, \ldots, x_{ia_i}] \\
\mathcal{F}^a\llbracket fdecl\rrbracket & = & \mathrm{fix}\,\lambda\varphi.(\varphi'_1, \ldots, \varphi'_k) \\
 & & \quad \text{where } \varphi'_i = (\mathcal{C}^a\llbracket c_i\rrbracket\varphi\downarrow\mathbf{1})_{|\{y_i, x_{i1}, \ldots, x_{ia_i}\}} \\
 & & \quad (fdecl \text{ as given by the abstract syntax}) \\
\mathcal{P}^a\llbracket fdecl\, \mathbf{do}\, c\rrbracket & = & \mathcal{C}^a\llbracket c\rrbracket\mathcal{F}^a\llbracket fdecl\rrbracket\downarrow\mathbf{1}
\end{array}
$$

Where:

$$
\begin{array}{lcl}
\mathbf{1} & = & (\{(x,x) | x \in I\}, \{x \mapsto \begin{cases} 1 & \text{if balloon(typeof } x\text{)}, \\ 0 & \text{otherwise.} \end{cases} | x \in I\}) \\
(p,b)[x \rhd y] & = & ((p \setminus \bigcup_{z \in px} \{(x,z),(z,x)\} \cup \{(x,x),(x,y),(y,x)\})^+, b[x \mapsto by]) \\
(p,b)[x \rhd] & = & (p \setminus \bigcup_{z \in px} \{(x,z),(z,x)\} \cup \{(x,x)\}, b[x \mapsto \begin{cases} 1 & \text{if balloon(typeof } x\text{)}, \\ 0 & \text{otherwise.} \end{cases}]) \\
(p,b)[x \rhd\!\lhd y] & = & ((p \cup \{(x,y),(y,x)\})^+, b[z \mapsto bx \sqcup by | z \in px \cup py]) \\
\mathrm{apply}\colon C \times C \to C_\top & = & \lambda((p_f,b_f),(p_a,b_a)).\begin{cases} \top & \text{if } \exists (x,y) \in p.b_a x = 1 \wedge b_a y = 1 \wedge (x,y) \notin p_a, \\ \top & \text{if } \exists x \in I.b_f x = 1 \wedge (\forall y \in p_f x.\neg\, \text{balloon(typeof } y\text{)}) \\ & \quad \wedge \exists y \in px.b_a y = 1 \vee (y \notin p_f x \wedge b_f y = 1), \\ (p,b) & \text{otherwise.} \end{cases} \\
 & & \quad \text{where } p = (p_f \cup p_a)^+ \\
 & & \quad \text{and } b = \{x \mapsto \bigsqcup_{y \in px} b_a y \sqcup b_f y | x \in I\}
\end{array}
$$

Figure 13: Abstract semantic function definitions

where $n = |I|$ and B_i is the i^{th} Bell number, the number of partitions of a set of i elements, which can be calculated by (see eg. [1]):

$$\begin{array}{lcl} B_0 & = & 1 \\ B_{n+1} & = & \sum_{k=0}^{n} \binom{n}{k} B_k \end{array}$$

As all the abstract states except $\top$ represent sets of concrete states where the balloon invariant holds, it follows that if $\mathcal{P}^a[\![p]\!] \neq \top_D$ then the balloon invariant holds in the final state of p.

Moreover the invariant will hold at any intermediate state during the execution of p, as desired. This follows from the functions being $\top$-strict (they propagate $\top$). This way if at any intermediate point in the program the balloon invariant is broken, the corresponding abstract state which is $\top$ is propagated to the final result.

The (conservative) procedure for checking a program p is thus: if $\mathcal{P}^a[\![p]\!] \neq \top_D$ the program is accepted, otherwise the program is rejected.