URL: http://www.elsevier.nl/locate/entcs/volume20.html 27 pages

Type-checking Balloon Types

Paulo Sérgio Almeida

Departamento de Informática
Universidade do Minho
Braga, Portugal

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.

Balloon types, which we have introduced in [2], are a general extension to programming languages. They make the ability to share state a first class property of a data type. The balloon invariant expresses a strong form of encapsulation: no state reachable (directly or transitively) by a balloon object is referenced by any external object.

In this paper we describe the checking mechanism for balloon types. It relies on a non-trivial static analysis, described as an abstract interpretation. Here we focus in particular on the design of the abstract domain which allows the checking mechanism to work under realistic assumptions regarding possible object aliasing.

1 Introduction

Modern imperative languages (object-oriented languages in particular) have benefited from advances such as structured control-flow, data abstraction [9,11], subtype polymorphism [7,4], and bounded parametric polymorphism [6]. In spite of all these advances, programming in object-oriented languages remains an error prone activity, and reasoning (either people or static analysis tools) remains difficult. We consider that one of the reasons for this is a flaw in current data abstraction mechanisms: they do not provide an appropriate mechanism to organise the state (i. e. the graph of objects) manipulated by the program. The direct cause of the problem is the trivialisation of the use of references (pointers), with no appropriate mechanism to control the proliferation of inter-object references.

Not much has been done to address the problem; one of the few attempts has been the *Islands* [13] proposal. *Balloon Types*, which we have introduced in [2], is a general language/type-system mechanism which aims to resolve the

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

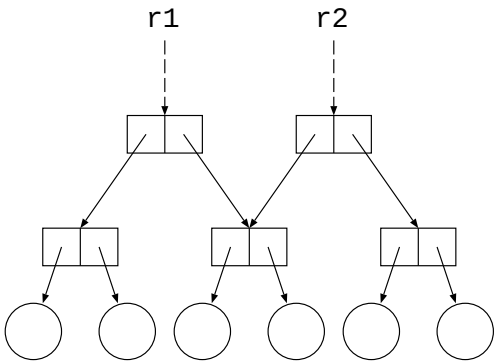


Fig. 1. Two Rectangles Sharing State

problem by making the ability to share state a first-class property of a data type. A detailed presentation of balloon types and the checking mechanism, can be found in [3].

In Section 2 we discuss briefly the problems caused by accidental state sharing and why current data abstraction mechanisms do not provide appropriate support. We review the basic idea of balloon types in Section 3 and we present the *balloon invariant* in Section 4.

Section 5 is the central part of the paper; it presents the checking mechanism for balloon types. The mechanism is based on an abstract interpretation [8] which a candidate program undergoes so as to check that the balloon invariant will definitely hold at run-time. The abstract interpretation is somewhat involved and, due to space constraints, we will restrict to presenting the more central part, the base abstract domain. We do, however, take some care in discussing its design.

2 Sharing of State

In many object-oriented languages (eg. Smalltalk [10], Java [5]) 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` in Figure 1.

After the assignment both rectangles share a common point object. (As in Simula [9] we have used the '`:-`' notation for reference assignment.) Consider the operation `rotate` which updates the point objects that constitute a rectangle; there would be interference between the two rotate operations, 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

```
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);
```

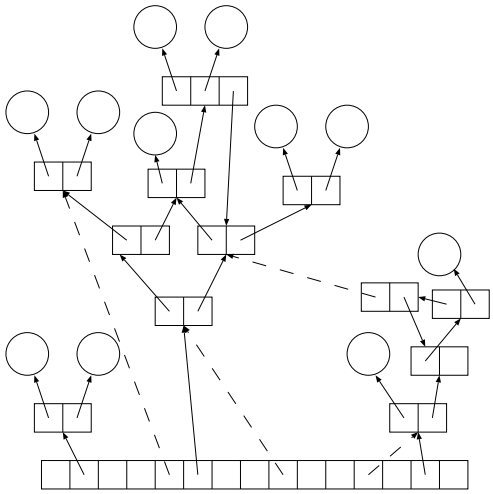


Fig. 2. An array of shapes

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.

Consider now 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 2.

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.

3 Balloon Types

According to [15], 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.

However, even if technically possible, a data type should not always enforce encapsulation of state (as we see it). 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 not even have 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 to share 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 reasoning about programs, and it is used in compiler optimisations.

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

- Balloon types are used to prevent unwanted sharing of state, guaranteeing a strong form of encapsulation. 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 and can be used to represent linked structures with possible substructure sharing.

Considering a snapshot of the run-time object structures at a given instant, we can look at it as an *object graph*: a finite directed graph whose nodes correspond to objects and whose edges correspond to inter-object references (pointers); the nodes can be labelled as balloon or non-balloon, corresponding to the classification of the corresponding object types.

```
Shape = balloon { 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);
```

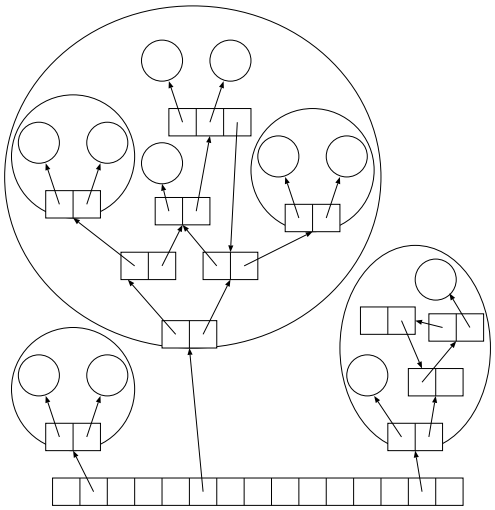


Fig. 3. An array of balloon shapes

Balloon types provide an invariant regarding the structure of the object graph; essentially:

- Objects of a balloon type are unshareable by state variables of objects.
- All the state reachable by a balloon object 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 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).

In the example shown in Figure 3 the programmer has chosen `Shape` to be a balloon type to obtain 'nice' semantics in its use in programs. It prevents accidental sharing even if each shape is a complex structure with internal sharing and even cycles.

In the loop presented, the balloon invariant makes clear to both programmer and compiler the absence of interference between iterations: performing a rotate on a shape `a[i]` does not affect a shape `a[j]` (when `i` and `j` are different). This makes reasoning about the program easier and the compiler can perform loop transformations such as parallelisation. This is accomplished with an almost negligible syntactic cost; if we compare Figure 2 with Figure 3, there is only one extra keyword in the new program.

This figure also illustrates that in spite of the binary classification, both balloon and non-balloon objects can be used as part of the state of each other. This results in a hierarchical organisation of the object graph, important for

the scalability of the mechanism.

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; 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. No checking of non-balloon client code is needed.

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.

4 The Balloon Invariant

We now describe more precisely the run-time invariant which is enforced by balloon types. Every object is an instance of either a balloon or a non-balloon type, and thus the terms balloon and non-balloon object. First we present some definitions.

Definition 4.1 [Cluster] Let G be the subgraph of the object graph obtained by removing all edges corresponding to references to balloon objects. A cluster is the set of objects in a connected subgraph of G that is not contained in a larger connected subgraph.

The set of all clusters is thus a partition of the set of all objects.

Definition 4.2 [Internal] An object O is said to be *internal* to a balloon object B iff :

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

Definition 4.3 [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.

Definition 4.4 [Balloon Invariant] If B is an object of a balloon type then:

I_1 There is at most one reference to B in the set of all objects.

I_2 This reference (if it exists) is from an object external to B.

I_3 No object internal to B is referenced by any object external to B.

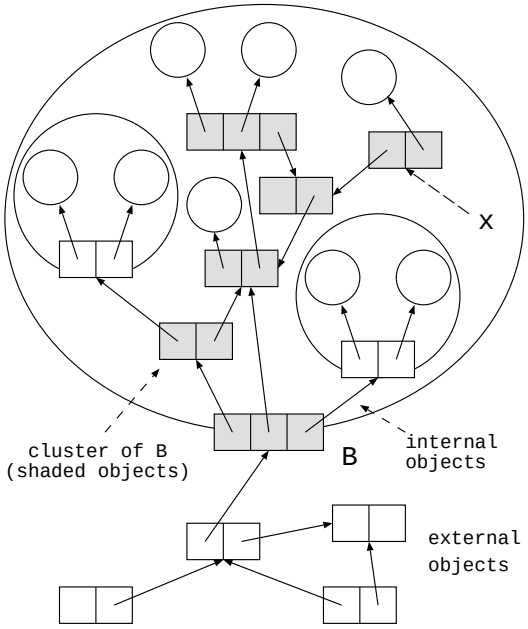


Fig. 4. A balloon B and its internal and external objects

Figure 4 clarifies these concepts. We should stress that the invariant is concerned with the organisation of the object graph (objects and inter-object references); it ignores references in variables from the chain of procedure calls (i.e. temporary local variables). In other words, the invariant is concerned with static aliasing, ignoring dynamic aliasing [12].

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 for this is as follows.

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 scenario did not actually happen, the mere possibility of it happening would 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. They are allowed to be created and manipulated by a procedure of a balloon type, but will be prevented from being returned to client code, as we will discuss later. The state reachable by a balloon object

is a subset of the internal objects; it is encapsulated in the sense that it is prevented from being referenced by external objects.

5 Type-checking Balloon Types

In this section we present the essence of the checking mechanism for balloon types: a static analysis, presented as an abstract interpretation, to verify that the balloon invariant holds for all object graphs which may occur during the execution of a given program. We use an approach to abstract interpretation based on logical relations, inspired by Abramsky's [1], which we describe in [3]. We define a simple language (RISO), and present a standard denotational semantics, followed by the abstract semantics.

The mechanism as presented here involves a global program analysis, something which is not realistic nor intended. We cannot afford to discuss here modularity, nor other relevant issues for incorporating balloon types in 'real' languages, but they are addressed in [3].

Due to space limitations, we will not address functions in RISO as well as in the concrete and abstract semantics; we will concentrate on the base domains and the semantics of simple statements; also, all proofs have been omitted.

5.1 An Equivalent Invariant

It will be useful to express the invariant in another form. For this we define I_4:

Definition 5.1 [I_4] From all objects that make up a cluster, at most one is a balloon object.

It can be shown that the balloon invariant is equivalent to $I_1 \wedge I_2 \wedge I_4$. The balloon type-checking mechanism enforces this last expression. I_1 is enforced by a simple rule concerning the reference assignment, which we present next, while I_2 and I_4 are enforced by means of the abstract interpretation.

5.2 Reference and Copy Assignment

Programs with balloon types will be subject to the following rule:

Definition 5.2 [The Simple Rule] A reference to a (pre-existing) balloon cannot be stored in any state variable of any object (by the reference assignment).

This 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 I_1, 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, to illustrate the usefulness of balloon references despite this restriction, consider a dictionary containing elements that 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. 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.

The simple rule implies that state variables of balloon type can only be made to reference newly created balloons. This can be done with a general copy mechanism with the semantics of deep-copy (as in e.g. [14]), which creates a copy of a balloon and all its reachable state, while preserving internal sharing. It can be provided as a *copy assignment*:

```
x.shape := s;
```

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'.

5.3 RISO

RISO is an imperative language with recursive definition of functions and shareable objects. RISO models accurately both the possibility of several variables referring to the same object (dynamic aliasing) and the sharing of

$$
\begin{aligned}
o : \mathit{Op} \quad &::= + \mid - \mid * \mid / \mid = \mid < \mid > \\
e : \mathit{Exp} \quad &::= n \mid x\ o\ y \mid \texttt{isnull}\ x \\
a : \mathit{Asgn} &::= x :\!- y \mid x :\!- y.z \mid x :\!- \texttt{null} \mid x :\!- \texttt{new} \mid \\
&\qquad x.y :\!- z \mid x.y :\!- \texttt{null} \mid x.y :\!- \texttt{new} \mid \\
&\qquad x <\!- e
\end{aligned}
$$

Fig. 5. Abstract syntax of RISO

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.

Integers do not receive a special treatment: integer variables are also references to possibly shared integer objects. It can be argued that the use of integers in RISO does not correspond to realistic languages. We note, however, that RISO should be regarded more as a target language that not only allows translating some restrictive ways in which integers are treated in a particular language, but which also allows different possibilities of both dynamic and static aliasing to be expressed in a orthogonal way for all data-types. The idea is to make no exception so that all data-types are treated alike, and to provide full freedom of sharing so that restrictions can be later expressed. Integers are only included to give the language a traditional form and make it naturally expressive without resorting to artificial encodings; integers also play the role of boolean values (0 plays the role of true and any other number the role of false).

The abstract syntax is given in Figure 5. We use $n \in N$ for numbers and x, y and z—ranging over a set of identifiers I—for identifiers of both variables and (object) state variables. An expression with integers is restricted to being a number, an operation between integer variables and the test for the null reference.

The reference assignment is denoted by '$:\!-$'. We use the traditional dot notation to access state variables; `null` for the null reference; `isnull` for the test for a null reference; and `new` for the creation of objects (including integer objects which are initialised to zero). We also have the operator '$<\!-$' for performing updates on integer objects (changing the associated integer value), because the '$:\!-$' assignment does not modify the integer object but makes the variable reference some other object.

To avoid considering both type annotations and type-checking in the classic sense (something which in this first-order non-polymorphic language is trivial but also irrelevant and distracting), and to concentrate on the balloon aspect, we assume a simple type checking of a type annotated version of RISO is performed, producing:

- a set of object types T, with $\mathrm{Int} \in T$,
- the set I of variable identifiers in the program,
- a mapping $\mathrm{typeof} : I \to T$ (we assume, without loss of generality, that a given identifier cannot be used for different object types in different parts of a program),
- a predicate $\mathrm{balloon} : I \to \{\mathrm{true}, \mathrm{false}\}$, corresponding to the annotation which will be the subject of the checking, with $\mathrm{balloon}\, x = \mathrm{balloon}\, y$ if $\mathrm{typeof}\, x = \mathrm{typeof}\, y$, and with $\mathrm{balloon}\, x = \mathrm{true}$ if $\mathrm{typeof}\, x = \mathrm{Int}$,
- 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 for identifiers x used in expressions (Exp) $\mathrm{typeof}\, x = \mathrm{Int}$.

5.4 Standard Denotational Semantics of RISO

We now present a denotational semantics for RISO. It is the concrete semantics to which the abstract semantics will be related. This semantics models accurately both 'heap allocated' objects and recursive definitions of functions, being suitable to be adapted to real imperative languages. 'The state' has two components:

- one is a mapping from variables to addresses;
- 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. (We have chosen the term *address* without implying that it corresponds to physical addresses in some implementation. Others may prefer the term *object identifier*.)

The semantic domains are given in Figure 6. In the representation of object graphs addresses not in use are mapped to the undefined object ($\perp_O$). In the representation of a record the identifiers which are not part of the record remain mapped to the null address ($\perp_A$). This enables us to work with total mappings. Note also the discreteness of S, instead of having the standard coordinatewise order; otherwise most functions would not be monotone, and hence continuous.

Figure 7 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.

The definition of the semantic functions is given in Figure 8.

The semantic functions reflect what we have informally described, and we will only make a few remarks. Accessing a state variable of an object $x.y$ causes program abortion if x is the `null` reference, to what corresponds $\perp$; these cases are expressed using the let $- \Leftarrow -. -$ construct.

N	discrete cpo of numbers
I	finite discrete cpo of identifiers
A	finite flat cpo of addresses
	($\bot_A$ denotes the null address)
$V = [I \to A]$	pointed cpo of variable mappings
	$\bot_V$ denotes the null mapping
$O = (N + V)_\bot$	pointed cpo of object values
	($\bot_O$ denotes the undefined object)
$G = [A \to O]$	pointed cpo of graphs of objects
	$\bot_G$ denotes the null graph
$S = \overline{G \times V}$	discrete cpo, 'the state'

Fig. 6. Semantic domains

$$\mathcal{O} : Op \to (N_\bot \times N_\bot) \to N_\bot$$
$$\mathcal{E} : Exp \to S \to N_\bot$$
$$\mathcal{A} : Asgn \to S \to S_\bot$$

Fig. 7. Semantic functions

The semantic functions make use of a function $\mathrm{alloc} : G \to A$, which we do not need to specify fully; we only assume this function satisfies the following properties, typical of a memory allocation function:

$$\forall g \in G.((\forall a \in A \setminus \bot_A . ga \neq \bot_O) \Rightarrow \mathrm{alloc}\, g = \bot_A)$$
$$\wedge\, ((\exists a \neq \bot_A . ga = \bot_O) \Rightarrow \mathrm{alloc}\, g \neq \bot_A \wedge g(\mathrm{alloc}\, g) = \bot_O).$$

That is, it returns the null address ($\bot_A$) if there are no free memory addresses, and returns a free memory address otherwise.

5.5 Base States in the Abstract Interpretation

Here we describe the way we abstract the relevant properties about concrete states. To best understand the structure of the resulting domain, the presentation is divided in two parts.

- First we describe a domain which represents the information about clusters:

$$
\begin{aligned}
\mathcal{E}[\![n]\!] &= \lambda(g,v).\lfloor n \rfloor \\
\mathcal{E}[\![x\ o\ y]\!] &= \lambda(g,v).\mathcal{O}[\![o]\!](g(vx), g(vy)) \\
\mathcal{E}[\![\texttt{isnull}\ x]\!] &= \lambda(g,v).\begin{cases} \lfloor 0 \rfloor & \text{if } vx = \bot_A, \\ \lfloor 1 \rfloor & \text{otherwise.} \end{cases} \\
\mathcal{A}[\![x :- y]\!] &= \lambda(g,v).\lfloor g, v[x \mapsto vy] \rfloor \\
\mathcal{A}[\![x :- y.z]\!] &= \lambda(g,v).\,\text{let}\, a \Leftarrow vy.\lfloor g, v[x \mapsto g\lfloor a \rfloor z] \rfloor \\
\mathcal{A}[\![x :- \texttt{null}]\!] &= \lambda(g,v).\lfloor g, v[x \mapsto \bot_A] \rfloor \\
\mathcal{A}[\![x :- \texttt{new}]\!] &= \lambda(g,v).\,\text{let}\, a \Leftarrow \text{alloc}\, g.\lfloor g[\lfloor a \rfloor \mapsto o], v[x \mapsto \lfloor a \rfloor] \rfloor \\
&\qquad \text{where } o = \begin{cases} \lfloor 0 \rfloor & \text{if typeof}\, x = \text{Int}, \\ \lfloor \bot_V \rfloor & \text{otherwise.} \end{cases} \\
\mathcal{A}[\![x.y :- z]\!] &= \lambda(g,v).\,\text{let}\, a \Leftarrow vx.\lfloor g[\lfloor a \rfloor \mapsto g\lfloor a \rfloor[y \mapsto vz]], v \rfloor \\
\mathcal{A}[\![x.y :- \texttt{null}]\!] &= \lambda(g,v).\,\text{let}\, a \Leftarrow vx.\lfloor g[\lfloor a \rfloor \mapsto g\lfloor a \rfloor[y \mapsto \bot_A]], v \rfloor \\
\mathcal{A}[\![x.y :- \texttt{new}]\!] &= \lambda(g,v).\,\text{let}\, a_x \Leftarrow vx.\,\text{let}\, a_n \Leftarrow \text{alloc}\, g. \\
&\qquad \lfloor g[\lfloor a_n \rfloor \mapsto o][\lfloor a_x \rfloor \mapsto g\lfloor a_x \rfloor[y \mapsto \lfloor a_n \rfloor]], v \rfloor \\
&\qquad \text{where } o = \begin{cases} \lfloor 0 \rfloor & \text{if typeof}\, y = \text{Int}, \\ \lfloor \bot_V \rfloor & \text{otherwise.} \end{cases} \\
\mathcal{A}[\![x <- e]\!] &= \lambda(g,v).\,\text{let}\, a \Leftarrow vx.\,\text{let}\, i \Leftarrow \mathcal{E}[\![e]\!](g,v).\lfloor g[\lfloor a \rfloor \mapsto \lfloor i \rfloor], v \rfloor
\end{aligned}
$$

Fig. 8. Semantic function definitions

whether different variables may reference objects in the same cluster and whether the cluster is free (does not contain any balloon object) or captured. This would be the base domain used if only invariant I_4 needed to be enforced.

- Then we present the base domain used in the actual abstract interpretation (which must also take invariant I_2 into account). This domain is obtained by refining each original state into several states, by adding information about cluster relationships: whether a free cluster may 'reach' a captured cluster.

From the set of concrete states S, we will concentrate essentially on the set of *valid* states S_b—the set of states in which the invariant holds. This set will be abstracted to a set C.

All *invalid* concrete states are represented by one more abstract state. We do not need to further discriminate them because if an invalid state results

at some point, the analysis terminates and the outcome is 'invalid program'. Thus, we only need to discriminate relevant information about valid concrete states.

5.5.1 Representing Clusters

A concrete state (a variable mapping and an object graph) is a complex structure. We do not, however, need to manipulate it directly. The relevant (for now) information about concrete states can be summarised by two functions (which only serve presentation purposes and are not used in the actual static analysis):

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

Abstract States and the Abstraction Function

Each element of S_b is abstracted into an element of a finite set C, which constitutes a direct representation of what is expressed by the above functions. Elements of C have the form $(p, b) \in \mathcal{P}(I \times I) \times (I \to \{0, 1\})$ such that:

- p is an equivalence relation on the set of identifiers I; defining a partition according to what variables 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 $x\ p\ y \Rightarrow bx = by$.

We use a function to $\{0, 1\}$ because in a valid state a cluster can have at most one balloon (as expressed by invariant I_4). The abstraction function $\alpha\colon S_b \to C$ is defined directly in terms of P and B:

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

The way C is defined, for each element of C there will be at least one concrete state abstracted into it; that is, α is surjective.

The Concretisation Function

While a concrete state in S_b is abstracted to a single state in C, each element of C represents a set of concrete states which is larger than the set of those elements abstracted to it. For a given abstract state (p, b):

- If x and y are not both mapped to 1 by b, and are not in the same equivalence class in 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* reference objects in the same

cluster.

- If $bx = 0$ it means that there is *definitely* 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 the following point.)
- 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 $\gamma : C \to \mathcal{P}(S_b)$:

$$\gamma = \lambda(p, b).\{s \in S_b \mid \forall x, y \in I.\ ((bx = 0 \vee by = 0) \wedge x \not\equiv y \Rightarrow (x, y) \notin Ps) \\ \wedge B(s, p\{x\}) \leq bx\}.$$

This choice of what an abstract state represents follows from the assumptions that must be made and the purpose of the analysis, as we now explain:

- Here we aim to check that invariant I_4 is not broken; towards this, we want to forbid two clusters from being merged when each cluster may have one balloon object. From this it follows that it matters to know that either *definitely* there are no balloons in a cluster or there *may* exist one. (It is irrelevant to know that there is *definitely* one.)
- In order to decide whether an operation that may merge clusters is to be allowed, the states have been designed so that a variable x that references a cluster with no balloons ($bx = 0$), *definitely* references a different cluster than other variable in a different equivalence class. In this case x can be used in such operation, with the result of merging the equivalence classes in the abstract semantics.

  However, due to loss of information in the analysis, we may not be sure whether clusters have been *actually* merged. Therefore, when two variables are in the same equivalence class they *may* reference different clusters.
- When two variables x and y reference clusters which possibly have one balloon ($bx = by = 1$), they may be in different equivalence classes and reference objects in the same cluster. The reason is that if both variables point to possibly captured clusters, they must be forbidden to appear in some operation which may merge the clusters when more than one cluster *may* be involved. In this case there is no point in trying to establish that the variables involved *definitely* point to different clusters.

  More than unnecessary, it is indeed important that no assumption is made. The reason is that, when a procedure of some balloon type has several parameters of that type, the checking must assume that they may point to different balloons (under a modular checking no assumptions can be made regarding client code), and the corresponding variables must be in different equivalence classes. However, it may be the case that dynamic aliasing exists, and several variables point to the same cluster. The mechanism must work under this possibility of dynamic aliasing.

  Even if preventing dynamic aliasing may be desirable, such can only be

accomplished either by an overly conservative static mechanism, or by dynamic checking. Therefore, absence of dynamic aliasing is something which we do not assume/enforce in the balloon mechanism.

- The final remark we make is about variables in the same equivalence class mapped to 1 by b. They may point to different clusters, but they must be allowed to be used together in operations that possibly merge the clusters (if they had not already been merged). Thus, we must assume that at most one balloon is present even if more than one cluster is involved.

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. That is, for all $s \in S_b$:

$$\{s\} \subseteq \gamma(\alpha s).$$

We now present some examples of abstract states and corresponding concrete states. We use a graphic notation to refer to elements of C; this is not only much more compact but also more ilustrative than using tuples and plain set notation. An element (p, b) represented as $\boxed{\text{xy}}\,\underline{\overline{\text{z}}}$ 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. Concrete states are also exemplified graphically, in this case by drawing a graph of objects.

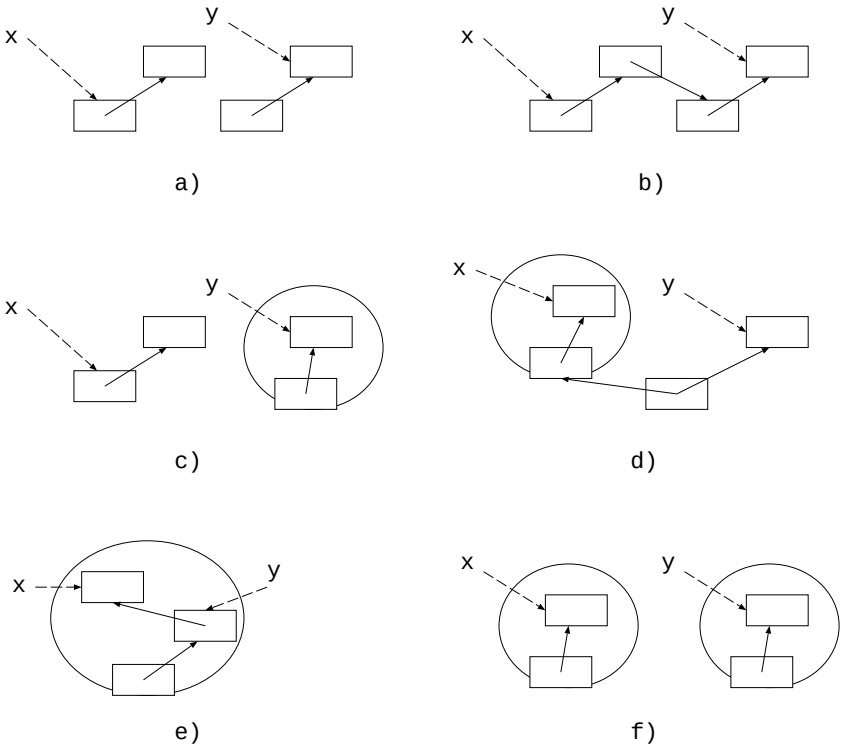
Figure 9 contains some examples of concrete and abstract states when two variables (x and y) are involved. For two variables there are six possible abstract states. The figure shows six representative concrete states (labeled a through f), each one abstracted into one of the possible states, as shown in the table. In the table are also shown (ticked), for each abstract state, which are the corresponding concrete states. For example, a and c belong to $\gamma(\underline{\overline{\text{x}}}\,\boxed{\text{y}})$, and all concrete states belong to $\gamma(\boxed{\text{x}}\,\boxed{\text{y}})$. All cells in the diagonal are ticked, as required by $\{s\} \subseteq \gamma(\alpha s)$ for all concrete states s. We remark that in state d, variables x and y point to different clusters, as a reference from a non-balloon to a balloon does not make the objects belong to the same cluster; therefore $\alpha d = \boxed{\text{x}}\,\underline{\overline{\text{y}}}$.

Order in the Abstract States

The function γ induces a partial order on C such that γ becomes an order-embedding of C into $\mathcal{P}(S_b)$; that is, such that for all $c_1, c_2 \in C$ we have $c_1 \sqsubseteq c_2 \Leftrightarrow \gamma c_1 \subseteq \gamma c_2$. This happens if we define the order as:

Definition 5.3

$$\begin{aligned}(p_1, b_1) \sqsubseteq (p_2, b_2) \Leftrightarrow (b_1 \sqsubseteq b_2) \wedge \forall x, y \in I.\ & \\ (x\ p_1\ y \wedge x \not\equiv_2 y \Rightarrow b_2 x = b_2 y = 1) & \\ \wedge\ (x \not\equiv_1 y \wedge x\ p_2\ y \Rightarrow b_1 x + b_1 y \leq b_2 x).&\end{aligned}$$



s	αs	$\underline{\overline{x}}\,\underline{\overline{y}}$	$\underline{\overline{xy}}$	$\underline{\overline{x}}\,\underline{\boxed{y}}$	$\underline{\boxed{x}}\,\underline{\overline{y}}$	$\underline{\boxed{xy}}$	$\underline{\boxed{x}}\,\underline{\boxed{y}}$
a	$\underline{\overline{x}}\,\underline{\overline{y}}$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
b	$\underline{\overline{xy}}$		$\surd$			$\surd$	$\surd$
c	$\underline{\overline{x}}\,\underline{\boxed{y}}$			$\surd$		$\surd$	$\surd$
d	$\underline{\boxed{x}}\,\underline{\overline{y}}$				$\surd$	$\surd$	$\surd$
e	$\underline{\boxed{xy}}$					$\surd$	$\surd$
f	$\underline{\boxed{x}}\,\underline{\boxed{y}}$						$\surd$

Fig. 9. Examples of concrete and abstract states

(Here $b_1 \sqsubseteq b_2$ means the usual pointwise ordering: $\forall x \in I.\, b_1 x \leq b_2 x$.) Figure 10 shows C, which is a pointed cpo, in the case when $I = \{x, y, z\}$. With the above order we have for example: $\underline{\boxed{x}}\,\underline{\overline{y}}\,\underline{\overline{z}} \sqsubseteq \underline{\boxed{xy}}\,\underline{\overline{z}}$ and $\underline{\boxed{xy}}\,\underline{\overline{z}} \sqsubseteq \underline{\boxed{x}}\,\underline{\boxed{y}}\,\underline{\overline{z}}$.

Lemma 5.4 $c_1 \sqsubseteq c_2 \Rightarrow \gamma c_1 \subseteq \gamma c_2$.

Lemma 5.5 $s \in \gamma c \Rightarrow \alpha s \sqsubseteq c$.

Proposition 5.6 $\alpha s \sqsubseteq c \Leftrightarrow s \in \gamma c$.

Proposition 5.7 $c_1 \sqsubseteq c_2 \Leftrightarrow \gamma c_1 \subseteq \gamma c_2$.

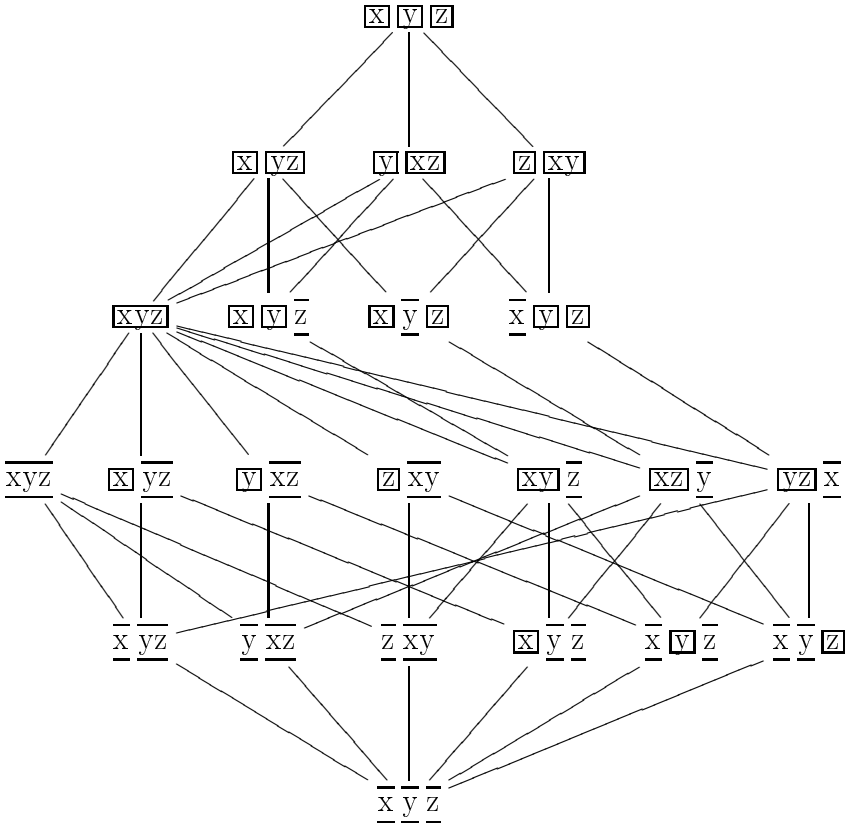


Fig. 10. The $C_{\{x,y,z\}}$ cpo

5.5.2 Extending the Representation to Cluster Relationships

We begin by making an informal presentation. By the simple rule, a reference to a balloon can be stored in some object only through the copy assignment, which gives a newly created balloon; therefore, the reference is stored in an external object. To enforce I_2—the only reference to a balloon B is from an external object—we must prevent the external object which references a balloon from becoming internal. To do this, we must prevent any balloon B from capturing non-balloons in a free cluster which references either B or some balloon which contains B. (Only free clusters need surveillance, as non-balloons in captured clusters no longer can be captured; this is assured by the mechanism which enforces I_4.)

The abstract states as presented above partition variables according to clusters, but do not contain information about relationships between different clusters. As an example, the abstract state $\overline{\text{X}}\,\boxed{\text{Y}}\,\boxed{\text{Z}}$ can correspond to any of the three cases in Figure 11. While in the first case (on the left) it would we acceptable to perform an instruction like '`y.a :- x`', in the other two cases performing this instruction would break I_2.

To enforce I_2, the previously described abstract states are refined in order to distinguish these situations: to a free cluster is now associated a set of which captured clusters *may be* 'reachable' by the free cluster. If a captured

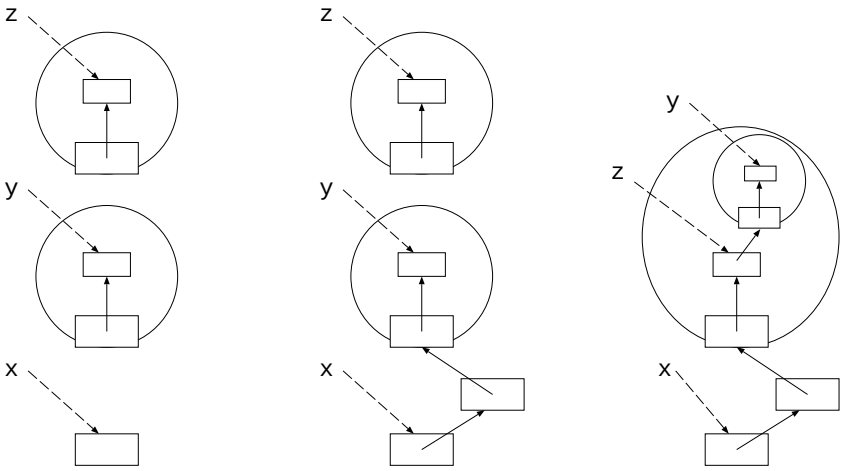


Fig. 11. Three cases abstracted into the same state

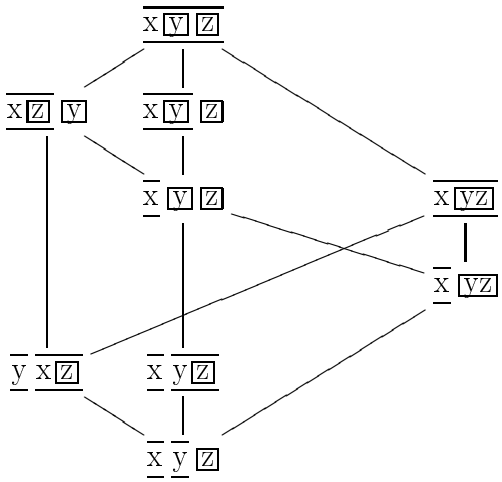


Fig. 12. Fragment of the extended $C_{\{x,y,z\}}$ cpo

cluster is not in this set, then it is *definitely* not 'reachable' by the free cluster. (Every captured cluster is associated with at most one free cluster.)

The original state $\underline{\overline{x}}\,\boxed{y}\,\boxed{z}$ is now refined into four states: $\underline{\overline{x}}\,\boxed{y}\,\boxed{z}$, $\underline{\overline{x\boxed{y}}}\,\boxed{z}$, $\underline{\overline{x\boxed{z}}}\,\boxed{y}$, and $\underline{\overline{x\boxed{y}\,\boxed{z}}}$. The graphic notation for abstract states is similar to the previous one, with the added possibility of a free cluster 'reaching' a set of captured clusters. The three cases in Figure 11 are now abstracted as $\underline{\overline{x}}\,\boxed{y}\,\boxed{z}$, $\underline{\overline{x\boxed{y}}}\,\boxed{z}$, and $\underline{\overline{x\boxed{y}\,\boxed{z}}}$ respectively. (Note how the nesting in the third case is not relevant; y and z could even refer to the same cluster, as before.) Figure 12 shows a fragment of the extended $C_{\{x,y,z\}}$ cpo, corresponding to refining the following three states: $\underline{\overline{x}}\,\underline{\overline{y}}\,\boxed{z} \sqsubseteq \underline{\overline{x}}\,\underline{\overline{yz}} \sqsubseteq \underline{\overline{x}}\,\boxed{y}\,\boxed{z}$.

Abstract States and The Abstraction Function

We now formalise what we have just described. The set of base abstract states C is extended to become a set of triples (p, b, r). For each state, the components p and b are as before, while the component r describes 'cluster

reachability': it is a binary relation on the set of identifiers I, subject to:

$$x\ r\ y \wedge y\ p\ z \Rightarrow x\ r\ z,$$

$$x\ r\ y \wedge x\ p\ z \Rightarrow z\ r\ y,$$

$$x\ r\ y \Rightarrow bx = 0 \wedge by = 1, \text{ and}$$

$$x\ r\ z \wedge y\ r\ z \Rightarrow x\ p\ y.$$

The first two conditions state that r defines a relation from clusters to clusters, the third states that it is from free to captured clusters, and the fourth states that no more than one (free) cluster is related to any given (captured) cluster.

To define the abstraction function, once again we use an auxiliary predicate which gives the relevant information:

- $R : S_b \times I \times I \to bool$, is a predicate such that $R(s, x, y)$ is true if and only if in the state s, variable x references an object in a free cluster F, there exists a balloon object B referenced by an object in F, and variable y references either B or an object internal to B.

The abstraction function $\alpha : S_b \to C$ becomes:

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

where the first two components of the resulting abstract state are as before. The abstraction function remains surjective.

The Concretisation Function

The new concretisation function is also based on the previous one:

$$\gamma = \lambda(p, b, r).\{s \in S_b \mid \forall x, y \in I.$$
$$((bx = 0 \vee by = 0) \wedge x \not p\ y \Rightarrow (x, y) \notin Ps) \wedge B(s, p\{x\}) \leq bx$$
$$\wedge\ ((bx = 0 \wedge by = 1 \wedge x \not r\ y) \vee x\ p\ y \Rightarrow \neg R(s, x, y))\}.$$

For the new pair of abstraction and concretisation functions it remains true that, for all $s \in S_b$:

$$\{s\} \subseteq \gamma(\alpha s).$$

Order in the Abstract States

As before, γ induces a partial order on C; the new order becomes:

Definition 5.8

$$(p_1, b_1, r_1) \sqsubseteq (p_2, b_2, r_2) \Leftrightarrow (b_1 \sqsubseteq b_2) \wedge \forall x, y \in I.$$
$$(x\ p_1\ y \wedge x \not p_2\ y \Rightarrow b_2 x = b_2 y = 1)$$
$$\wedge\ (x \not p_1\ y \wedge x\ p_2\ y \Rightarrow b_1 x + b_1 y \leq b_2 x)$$
$$\wedge\ (x\ r_1\ y \wedge x \not r_2\ y \Rightarrow x \not p_2\ y \wedge b_2 x = 1).$$

We notice that C has a least element $\bot_C$, given by:

$$\bot_C = (\{(x, x) \mid x \in I\}, \{(x, 0) \mid x \in I\}, \emptyset).$$

Figure 13 shows the C cpo in the case when $I = \{x, y, z\}$.

Lemma 5.9 $c_1 \sqsubseteq c_2 \Rightarrow \gamma c_1 \subseteq \gamma c_2$.

Lemma 5.10 $s \in \gamma c \Rightarrow \alpha s \sqsubseteq c$.

Proposition 5.11 $\alpha s \sqsubseteq c \Leftrightarrow s \in \gamma c$.

Proposition 5.12 $c_1 \sqsubseteq c_2 \Leftrightarrow \gamma c_1 \subseteq \gamma c_2$.

We have thus that, not only γ is an order-embedding (which means we have obtained the largest order appropriate to our concretisation function), but also the safety relation on the base domain derived from α (i.e. $s \; R \; c \Leftrightarrow \alpha s \sqsubseteq c$) coincides with what is expressed by γ:

$$s \; R \; c \Leftrightarrow \alpha s \sqsubseteq c \Leftrightarrow s \in \gamma c.$$

(We use the the same letter R for the logical relation and for the above auxiliary predicate; they are used in different contexts and no confusion should arise.)

5.5.3 Invalid States and Non-termination

Having defined C, invalid states in S are considered by adding a $\top$ to C, which represents all states in S—both valid and invalid states. The abstraction and concretisation functions are extended to $\alpha : S \to C_\top$ and $\gamma : C_\top \to \mathcal{P}(S)$ by making $\alpha(s) = \top$ if $s \notin S_b$, and $\gamma(\top) = S$. The order $\sqsubseteq$ becomes

$$c_1 \sqsubseteq c_2 \Leftrightarrow c_2 = \top \vee c_1 = (p_1, b_1, r_1) \sqsubseteq (p_2, b_2, r_2) = c_2$$

The relation R is also extended to become $R : S \to C_\top$; it is as before $s \; R \; c \Leftrightarrow \alpha s \sqsubseteq c \Leftrightarrow s \in \gamma c$, with the extended α, γ, and $\sqsubseteq$.

The standard semantics also uses a $\bot$ to represent non-termination or abortion (when an invalid operation occurs, such as dereferencing a null reference). Although it can be useful on its own to determine that a program aborts or does not terminate, this is not needed for our balloon checking purpose, and we do not use any dedicated abstract state to represent it. Instead we make $\bot \; R \; c$ for all $c \in C_\top$: for any possible abstract state, non-termination or abortion is a possible corresponding outcome in the standard semantics. This way, in establishing the correspondence between semantics, we can simply ignore the $\bot$ outcome in the standard semantics, namely in the 'let $- \Leftarrow -. -$' construct.

We can use the fact that C has a least element to be able to maintain $s \; R \; c \Leftrightarrow \alpha s \sqsubseteq c \Leftrightarrow s \in \gamma c$, by extending α and γ with $\alpha\bot = \bot_C$ and making $\bot \in \gamma c$ for all $c \in C_\top$.

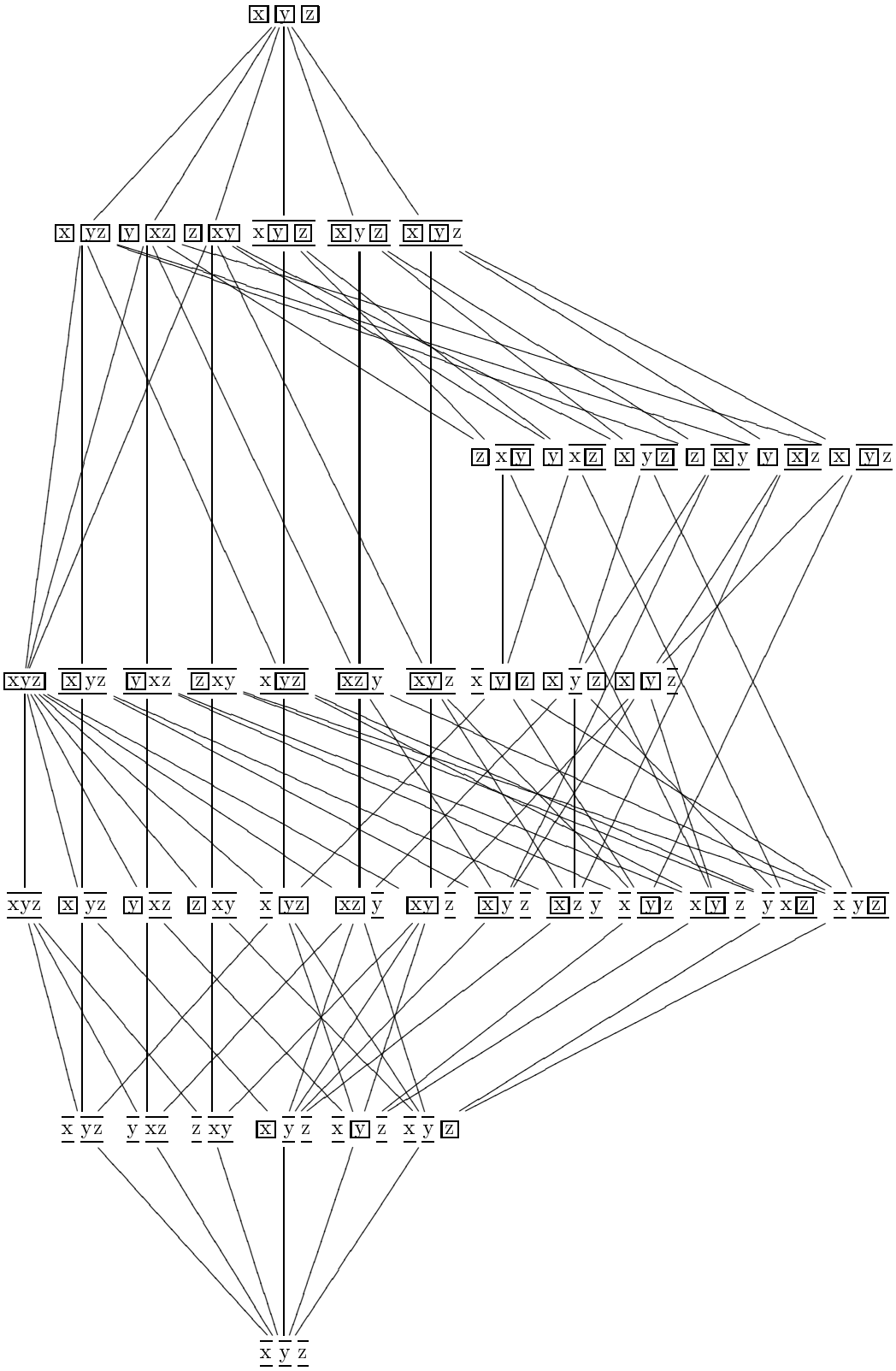


Fig. 13. The extended $C_{\{x,y,z\}}$ cpo

5.6 Atomic Commands

5.6.1 Operations on Base Abstract States

We now define some operations on abstract states which will be used in defining abstract semantics for the atomic commands (the assignments). These operations, not only serve to factorise similar cases, but are meaningful in themselves. We will use a notation resembling 'function update' (i.e. $f[x \mapsto y]$).

Definition 5.13 $(p, b, r)[x\triangleright] = (p', b', r')$, where

$$p' = p \overline{\restriction} \{x\} \cup \{(x, x)\},$$
$$b' = \begin{cases} b[x \mapsto 1] & \text{if balloon } x, \\ b[x \mapsto 0] & \text{otherwise.} \end{cases}$$
$$r' = r \overline{\restriction} \{x\}.$$

This operation detaches an identifier from the equivalence class where it was, and makes it become an equivalence class on its own. Moreover, this new equivalence class will be 'captured' or 'free' according to whether the identifier is from a balloon or a non-balloon type. For example, if balloon x then

$$\boxed{\mathrm{xy}}\,\underline{\overline{\mathrm{z}}}[x\triangleright] = \boxed{\mathrm{x}}\,\boxed{\mathrm{y}}\,\underline{\overline{\mathrm{z}}},$$
$$\underline{\overline{\boxed{\mathrm{xy}}\mathrm{z}}}[x\triangleright] = \boxed{\mathrm{x}}\,\underline{\overline{\boxed{\mathrm{y}}\mathrm{z}}},$$

and if $\neg$ balloon x then

$$\underline{\overline{\mathrm{xy}}}\,\underline{\overline{\mathrm{z}}}[x\triangleright] = \underline{\overline{\mathrm{x}}}\,\underline{\overline{\mathrm{y}}}\,\underline{\overline{\mathrm{z}}},$$
$$\boxed{\mathrm{xy}}\,\underline{\overline{\mathrm{z}}}[x\triangleright] = \underline{\overline{\mathrm{x}}}\,\boxed{\mathrm{y}}\,\underline{\overline{\mathrm{z}}},$$
$$\underline{\overline{\boxed{\mathrm{xy}}\mathrm{z}}}[x\triangleright] = \underline{\overline{\mathrm{x}}}\,\underline{\overline{\boxed{\mathrm{y}}\mathrm{z}}}.$$

Definition 5.14 $(p, b, r)[x \triangleright y] = (p', b', r')$, where

$$p' = \begin{cases} p & \text{if } x = y, \\ (p \overline{\restriction} \{x\} \cup \{(x, y), (y, x)\})^+ & \text{otherwise.} \end{cases}$$
$$b' = b[x \mapsto by],$$
$$r' = r \overline{\restriction} \{x\} \cup \{(z, x) \mid z\ r\ y\} \cup \{(x, z) \mid y\ r\ z\}.$$

This operation moves an identifier from an equivalence class to another. Some examples are:

$$\underline{\overline{\mathrm{wx}}}\,\underline{\overline{\mathrm{yz}}}[x \triangleright y] = \underline{\overline{\mathrm{w}}}\,\underline{\overline{\mathrm{xyz}}},$$
$$\boxed{\mathrm{wx}}\,\underline{\overline{\mathrm{yz}}}[x \triangleright y] = \boxed{\mathrm{w}}\,\underline{\overline{\mathrm{xyz}}},$$
$$\underline{\overline{\mathrm{wx}}}\,\boxed{\mathrm{yz}}[x \triangleright y] = \underline{\overline{\mathrm{w}}}\,\boxed{\mathrm{xyz}},$$
$$\underline{\overline{\mathrm{wx}\boxed{\mathrm{yz}}}}[x \triangleright y] = \underline{\overline{\mathrm{w}\boxed{\mathrm{xyz}}}},$$
$$\underline{\overline{\mathrm{w}\boxed{\mathrm{x}}}}\,\underline{\overline{\mathrm{y}\boxed{\mathrm{z}}}}[x \triangleright y] = \underline{\overline{\mathrm{w}}}\,\underline{\overline{\mathrm{xy}\boxed{\mathrm{z}}}}.$$

Definition 5.15 $(p,b,r)[x \rhd\lhd y] = \begin{cases} \top & \text{if } x \not\, p\, y \wedge bx = by = 1, \\ \top & \text{if } x\ r\ y \vee y\ r\ x, \\ (p',b',r') & \text{otherwise, where} \end{cases}$

$p' = (p \cup \{(x,y),(y,x)\})^+,$
$b' = b[z \mapsto bx \sqcup by \mid z\ p\ x \vee z\ p\ y],$

$$r' = \begin{cases} r \cup \{(z,w) \mid (z\ p\ x \vee z\ p\ y) \wedge (x\ r\ w \vee y\ r\ w)\} & \text{if } bx = by = 0, \\ r \setminus \{(w,z) \mid w\ p\ y \wedge w\ r\ z\} & \\ \quad \cup \{(w,z) \mid w\ r\ x \wedge (y\ p\ z \vee y\ r\ z)\} & \text{if } bx = 1 \wedge by = 0, \\ r \setminus \{(w,z) \mid w\ p\ x \wedge w\ r\ z\} & \\ \quad \cup \{(w,z) \mid w\ r\ y \wedge (x\ p\ z \vee x\ r\ z)\} & \text{if } bx = 0 \wedge by = 1, \\ r & \text{if } bx = by = 1. \end{cases}$$

This operation merges equivalence classes; it defines the effect on abstract states corresponging to the merging of clusters in valid concrete states. It is defined for all elements in C; for some of them the corresponding merging of clusters leads to an invalid state; therefore this operation has $\top$ as a possible outcome. The possible cases are:

- If $x \not\, p\, y$ and $bx = by = 1$, there exists one corresponding concrete state with x and y pointing to two different clusters both containing a balloon object. Merging the clusters breaks invariant I_4; therefore we must have $\top$ as the corresponding abstract state. One example is
$$\boxed{\mathrm{x}}\,\boxed{\mathrm{y}}[x \rhd\lhd y] = \top.$$
- If x and y are related by r, there exists a corresponding concrete state for which merging the clusters breaks invariant I_2; therefore we must have $\top$ as the corresponding abstract state. One example is
$$\overline{\mathrm{x}\,\boxed{\mathrm{y}}}\,[x \rhd\lhd y] = \top.$$
- In the remaining cases, merging clusters in any corresponding concrete state does not lead to breaking the invariant; there will result a valid state, with a corresponding abstract state in C; some examples are:
$$\begin{aligned} \overline{\mathrm{x}}\ \overline{\mathrm{yz}}\,[x \rhd\lhd y] &= \overline{\mathrm{xyz}}, \\ \overline{\mathrm{x}\,\boxed{\mathrm{w}}}\ \overline{\mathrm{y}\,\boxed{\mathrm{z}}}\,[x \rhd\lhd y] &= \overline{\mathrm{xy}\,\boxed{\mathrm{w}}\,\boxed{\mathrm{z}}}, \\ \boxed{\mathrm{x}}\ \overline{\mathrm{y}\,\boxed{\mathrm{z}}}\,[x \rhd\lhd y] &= \boxed{\mathrm{xy}}\,\boxed{\mathrm{z}}, \\ \overline{\mathrm{w}\,\boxed{\mathrm{x}}}\ \overline{\mathrm{y}\,\boxed{\mathrm{z}}}\,[x \rhd\lhd y] &= \overline{\mathrm{w}\,\boxed{\mathrm{xy}}\,\boxed{\mathrm{z}}}, \\ \boxed{\mathrm{xy}}\ \overline{\mathrm{z}}\,[x \rhd\lhd y] &= \boxed{\mathrm{xy}}\ \overline{\mathrm{z}}. \end{aligned}$$

Definition 5.16 $(p,b,r)[x \xrightarrow{+} y] = (p,b,r')$, where

$r' = r \setminus \{(w,z) \mid w\ r\ z \wedge y\ p\ z\} \cup \{(w,z) \mid x\ p\ w \wedge y\ p\ z\}.$

$$
\begin{aligned}
\mathcal{A}^a[\![x :- y]\!] &= \lambda c.\, c[x \rhd y] \\
\mathcal{A}^a[\![x :- y.\, z]\!] &= \lambda(p,b,r).\begin{cases} (p,b,r)[x \rhd y] & \text{if } \neg\, \text{balloon}\, x, \\ (p,b,r)[x\rhd][y \xrightarrow{+} x] & \text{if balloon}\, x \wedge by = 0, \\ (p,b,r)[x\rhd][w \xrightarrow{+} x] & \text{if balloon}\, x \wedge \exists w.\, w\ r\ y, \\ (p,b,r)[x\rhd] & \text{otherwise.} \end{cases} \\
\mathcal{A}^a[\![x :- \texttt{null}]\!] &= \lambda c.\, c[x\rhd] \\
\mathcal{A}^a[\![x :- \texttt{new}]\!] &= \lambda c.\, c[x\rhd] \\
\mathcal{A}^a[\![x.\, y :- z]\!] &= \lambda c.\begin{cases} \top & \text{if balloon}\, z, \\ c[x \rhd\!\lhd z] & \text{otherwise.} \end{cases} \\
\mathcal{A}^a[\![x.\, y :- \texttt{null}]\!] &= \lambda c.\, c \\
\mathcal{A}^a[\![x.\, y :- \texttt{new}]\!] &= \lambda c.\, c \\
\mathcal{A}^a[\![x <- e]\!] &= \lambda c.\, c
\end{aligned}
$$

Fig. 14. Abstract semantics for assignments

This operation, which is only defined if $bx = 0$ and $by = 1$, adds the equivalence class where y is to the set of equivalence classes related (by r) to the one where x is. Some examples are:

$$\underline{\overline{\mathrm{x}}}\ \boxed{\mathrm{y}}\ \boxed{\mathrm{z}}[x \xrightarrow{+} y] = \underline{\overline{\mathrm{x}\boxed{\mathrm{y}}}}\ \boxed{\mathrm{z}},$$

$$\underline{\overline{\mathrm{x}\boxed{\mathrm{z}}}}\ \boxed{\mathrm{y}}[x \xrightarrow{+} y] = \underline{\overline{\mathrm{x}\boxed{\mathrm{y}}\boxed{\mathrm{z}}}}.$$

5.6.2 Abstract Semantics for Assignments

We now define the abstract semantic function for assignments in terms of the above operations. The semantic function is

$$\mathcal{A}^a : Asgn \to C \to C_\top$$

with the definition shown in Figure 14.

Proposition 5.17 *For all $a \in Asgn$, $\mathcal{A}[\![a]\!]\ R\ \mathcal{A}^a[\![a]\!]$.*

6 Conclusion

The pervading possibility of sharing objects due to the uncontrolled proliferation of inter-object references is a source of errors and makes reasoning about programs difficult. Balloon types make the ability of sharing a first class property of a data type and provide an invariant that expresses a strong form of encapsulation of state. Given the benefits that static type-checking is known to provide, we have developped a static checking mechanism for balloon types.

The mechanism relies on a static analysis of the candidate program in order to check that the balloon invariant holds at run-time. Given the complexity of the balloon invariant (and after having tried ad-hoc methods and having found errors in the mechanism), we have resorted to developing the static analysis as an abstract interpretation.

While it is common that in abstract interpretations for functional languages base domains are simple and it is the higher-order features that deserve attention, here the base domain is considerably complex, and we have devoted our attention to presenting it.

Balloon types and its checking mechanism exemplify how we can improve type systems of programming languages by: designing minimal concepts which provide important invariants; using the knowledge in the invariants inductively in the checking mechanism itself; designing the checking mechanism using semantics-based methods like abstract interpretation. This way we can obtain languages with higher level concepts which improve the ability to reason about programs.

References

[1] Samson Abramsky. Abstract interpretation, logical relations, and Kan extensions. *Journal of Logic and Computation*, 1(1):5–40, 1990.

[2] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In *Proceedings ECOOP'97*, volume 1241 of *LNCS*, pages 32–59. Springer-Verlag, June 1997.

[3] P. S. Almeida. *Control of Object Sharing in Programming Languages*. PhD thesis, University of London, Imperial College, Department of Computing, June 1998.

[4] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.

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

[6] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[7] Luca Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types*, volume 173 of *LNCS*, pages 51–67. Springer-Verlag, 1984. Full version in *Information and Computation* 76(2/3):138–164, 1988.

[8] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

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

[10] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

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

[12] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. 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.

[13] John Hogg. Islands: Aliasing protection in object-oriented languages. *Proceedings OOPSLA'91. SIGPLAN Notices*, 26(11):271–285, November 1991.

[14] S. Khoshafian and G. Copeland. Object identity. *Proceedings OOPSLA'86. SIGPLAN Notices*, 21(11):406–416, November 1986.

[15] Peter Wegner. Dimensions of object-based language design. *Proceedings OOPSLA'87. SIGPLAN Notices*, 22(12):168–182, December 1987.