University of London
Imperial College of Science, Technology and Medicine
Department of Computing

# Control of Object Sharing in Programming Languages

Paulo Sérgio Soares de Almeida

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Engineering of the University of London and
the Diploma of Imperial College, June 1998

# Abstract

Current data abstraction mechanisms are not adequate to control sharing of state in the general case involving objects in linked structures. They only prevent the direct access to the state variables of single objects, as opposed to considering the state reachable by an object and the inter-object references, neglecting the fact that an object is not, in general, self-contained. The pervading possibility of sharing is a source of errors and an obstacle both to reasoning about programs and to language implementation techniques.

This thesis presents *balloon types*, 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 provide the *balloon invariant*, which expresses a strong form of encapsulation of state: it is guaranteed that no state reachable (directly or transitively) by an object of a balloon type is referenced by any 'external' object.

The mechanism is syntactically very simple, relying on a non-trivial static analysis to perform checking. The static analysis is presented as an abstract interpretation based on a denotational semantics of a simple imperative first-order language with constructs for creating and manipulating objects.

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.

## Acknowledgements

I wish to thank those most involved in this thesis: my supervisor Paul Kelly for the valuable guidance and insightful comments he has provided, and for the fun discussions about crazy ideas in language design; Sophia Drossopoulou for her comments and discussions about object-oriented type systems; Chris Hankin for the comments about abstract interpretation. I also thank Bacelar for reading the appendix.

I thank the people in the ALA section, the folks at 313 in special, for providing a nice atmosphere; Marco, Francesca B, Francesca C, Uli and Tobi, my house-mates at Oxford Gardens, who made my final year in London a happy one; Helena and Geraldina for the weekends in Canterbury; Arménio and Dalila for the nice moments together and for sheltering me when I needed; the many friends back in Portugal who kept in touch with me; the members of the Distributed Systems Group at the University of Minho for the leave of absence these three years and for making the thought of going back a very pleasant one.

This work was supported by University of Minho and by JNICT—PRAXIS XXI grant BD-3390-94.

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Motivation

The imperative model of computation involves performing operations on a mutable state which evolves with time, either to converge to some result, or to interact with an external dynamic system. Some operations change a part of the state, which is then used by other operations; there are side-effects, which result in dependencies. While some of these dependencies reflect the tasks to be performed, there can also exist accidental and undesired *interference* between instructions.

Some unexpected interference may be the manifestation of an error in the program. Such bugs can be very hard to detect, and reasoning about the program is also difficult. Another kind of interference may reflect a poor design in which common state (such as auxiliary variables) is used by tasks that are not genuinely dependent. This last case can lead to an unnecessary bottleneck: the serialisation of tasks which could in fact be performed in parallel. These are some of the reasons why, in imperative languages, the issue of how to organise the state becomes important.

The more widely used languages in the 'real world' follow the imperative model of computation: either classic procedural languages like Pascal [77] or C [48], or object-oriented languages like Smalltalk [35], C++ [71], Eiffel [62], and Java [8]. Modern imperative languages (object-oriented languages in particular) have benefited from advances such as structured control-flow, data abstraction [27, 39], subtype polymorphism [20, 6], and bounded parametric polymorphism [19]. 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 argue 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.

If we look back, references were considered a dangerous feature [40, 78], like the *goto* statement. Quoting [40]: 'References are like jumps, leading wildly from one part of a data structure to another. Their introduction into high-level languages has been a step backward from which we may never recover.'

While the goto statement was successfully abolished in favour of structured control-flow mechanisms, the same did not happen to references; they are important to represent unbounded linked structures with possible mutable substructure sharing. In modern object-oriented languages (e.g. Java) references have become in fact the norm and not the exception.

Unfortunately, the widespread use of references was not accompanied by appropriate controlling mechanisms. Data abstraction mechanisms provided by current languages provide insufficient encapsulation: they just prevent the *direct* access to the state variables of *single* objects, as opposed to considering the state *reachable* by an object and the inter-object references, neglecting the fact that an object is not in general self-contained.

## 1.2 Contribution

The contribution of this thesis is *balloon types*, a general language/type-system mechanism for imperative languages:

- Balloons allow a cluster of objects to be treated as a self-contained composite object.
- Balloons make the ability to share state a first class property of a data-type.
- Balloons are a support for data abstraction, by providing a strong form of encapsulation of state.
- Balloons prevent the more insidious form of interference and facilitates reasoning about program correctness.

- Balloons give user-defined types the same status as primitive types, which need no longer be considered ‘special’.
- Balloons are a result in *language design for analysability*: a language mechanism that provides information/invariants which can be counted upon to enhance the outcome of static analysis of programs, such as dependency analysis towards parallelisation.
- Balloons are syntactically and conceptually minimal, hiding the complexity from programmers by relying on a non-trivial static checking developed using abstract interpretation.

The thesis characterises a long-standing problem in imperative languages; proposes, as solution, a novel mechanism that, at minimal syntactical cost, makes explicit in data-types an important invariant for reasoning about programs; formulates the invariant considering usefulness and enforceability; presents a static analysis to check the invariant; presents an extension of the mechanism that provides a stronger invariant; and discusses practical concerns for integrating the mechanism into real object-oriented languages. We have presented an introductory paper about this work in [5].

## 1.3 Thesis Structure

The problem we address deserves a detailed introduction, which we present in Chapter 2. We start by introducing some terminology; then, we argue why current data abstraction mechanisms fail in supporting effective encapsulation in the general case involving linked structures of objects; finally, we discuss previous approaches to the problem.

Chapter 3 introduces *balloon types*, the core contribution of the thesis. There we define the *balloon invariant* and present an overview of the checking mechanism for balloon types.

Chapter 4 presents, in a problem independent setting, our approach to abstract interpretation—the tool for program analysis used in developing the mechanism.

In Chapter 5 we describe the checking mechanism we have developed for balloon types. For this we define a simple language (RISO), a denotational semantics, and present an abstract interpretation which is the basis of the checking mechanism.

In Chapter 6 we describe an efficient way to handle functions, avoiding the use of naive function spaces; this is important due to the complexity of the base domains involved.

Chapter 7 presents *opaque balloon types*, an extension of the concept. It motivates the concept, describes the *opaque balloon invariant*, presents the *nesting interpretation*, and outlines two approaches to a checking mechanism, based on the nesting interpretation.

In Chapter 8 we focus on several issues related to incorporating balloon types into real languages, such as modularity, subtype and parametric polymorphism, and the existence of global variables.

We conclude in Chapter 9 by summarising what has been achieved and pointing to research directions opened up by balloon types.

# Chapter 2

# State, Sharing, and Language Design

In the first part of this chapter we establish some terminology, distinguish two forms of aliasing, and identify problems which result from object sharing. In particular, we explain why current data abstraction mechanisms do not provide appropriate support for managing object sharing and discuss the fundamental flaw in the form of encapsulation provided. In the second part of the chapter we do a survey of language mechanisms which attempt to solve some part of the problem.

## 2.1 Sharing and Encapsulation

### 2.1.1 Values, Objects, and Variables

The traditional notion of *variable* in an imperative language is a container for a value at some point in the execution of a program. We say that the integer variable `i` holds the value 4 at some point. If the assignment '`i := i + 1`' is then performed, it makes `i` hold the value 5.

A *value* is an abstract mathematical entity, and a variable holds in fact a *representation* of a value; it is, however, usual to say for short that 'a variable contains a value'.

A variable is also traditionally thought of as an independent assignable element. If we have variables `i` and `j` with values 4 and 7, and then perform the assignment '`i := i + 1`', the resulting values will be 5 and 7 respectively: the assignment to `i` does not affect `j`. This is what students are told to expect when a 'toy' imperative language is introduced.

If we are to consider realistic imperative languages, either procedural or object-oriented, the concept of object becomes important. An *object* contains state, typically through a set of *state variables* (we use the term state variable for what in different places is referred to as instance variable or field). An object has also some form of *identity*: we can speak of two different objects having the same state; identity is what distinguishes them [49, 10]. Identity gives objects the property of being *shareable*. This can be done by allowing variables to be *references* to objects; in this case the value of a variable is some kind of *object identifier*. It is possible that two different variables have the same object identifier as their value; in this case we say that they reference the same object.

Unlike values such as integers, an object identifier is not useful in itself, but only as a means to access an object. For this reason, although every variable contains a value, we distinguish the special case of references and say that a variable either contains a value or is a reference to an object.

State variables of objects can also be references, making it possible to build linked data structures, including recursive structures like lists or trees. The set of objects together with the references in the state variables can be seen as a graph, with objects as nodes and references as edges. In general it is possible to have sharing of nodes and even cycles.

While the notion of value is not generally argued about, the notions of variable and object are more debatable and sometimes blurred. We define them in the way we consider more consistent with the widespread intuition for the concepts.

The notion of variable is, like in mathematics, the association of a value to a name, but with an imperative flavour in that the associated value may change with time. The term 'object' comes from a physical or modelling sense, in that it is an entity containing state and with an identity which can be 'known' by other objects.

Our usage is quite general, stressing the duality of value versus object, on the lines of [58]; it suits both procedural and object-oriented languages. The difference between these two variants of imperative languages is essentially the way programs are organised. In procedural languages a program is organised as a set of procedures which can have direct access to the state of different objects. In object-oriented languages the program is organised around what classes of objects there are and what operations can be invoked for each different class of objects. State variables of each object are only, in principle, manipulated by the implementation of these operations, which are, in turn, used by client code.

Also, while there are some similarities between variable and object in the sense that both are updateable, we stress some important differences. A variable is accessed through a name and is assignable. An object is not normally assigned in itself, the state is instead updated by assignments to its state variables. Also an object is, in general, unnamed, even if it may be common that its lifetime is related to some variable.

Objects are shareable, possibly being referenced by different variables. On the contrary, a variable is not shareable in itself (it is not in itself the content of other variables); variables are, at most, part of an object which may itself be shared. In any case, the name of the variable must be used for access.

To emphasise these differences, consider a simple imperative language with a loop construct but without recursion. If there are no variables which are references, we have a fixed number of variables to consider, referred to by names such as `x`, `y`, and `z`. On the other hand, if there are references and objects are created, the execution of the program can create a linked data structure, and we cannot establish a bound for the size of this data structure: the number of objects is unrelated to the number of identifiers in the program. These differences have important consequences for reasoning about programs; they will play a central role throughout this thesis.

### 2.1.2 Aliasing

Even in the case of primitive types like integer, objects become relevant to describe the behaviour of a program when *aliasing* is present. As a general model, we can say that integer variables denote objects that contain (representations of) values. In the case of integers aliasing can be created when using call-by-reference, as illustrated in Figure 2.1.

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’.

It can be remarked that in this case `i` and `j` are not really variables but are constant references, as they refer always a given object. A similar remark applies to `x`; in this case, as the lifetime of the object is connected to the ‘variable’, it is common to say that ‘the variable contains the object’. It is possible, however, to have ‘true’ variables which can reference different integer objects at different

```
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
```

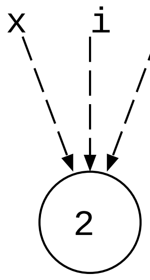


Figure 2.1: Aliasing in integer objects due to call-by-reference

times; such is the case in Algol 68 [72].

Aliasing has been long ago recognised as a source of problems [40, 78] and some attempts have been made to prevent it. The previous example would be invalid in Pascal [77], due to using `x` twice as an argument, but in general aliasing cannot be prevented at compile time (without being overly restrictive) 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.

Another example, involving `Point` objects is presented in Figure 2.2. `Point` objects have state variables `x` and `y` of type `Int`, and a `move` method. Here one point object is created and the reference assigned to `p`. A function is invoked using this reference in both arguments.

We represent objects either by rectangles divided in cells (when we want to consider their state variables) or by circles (when the representation of their state is irrelevant and self-contained, as for integer objects). Variables are simply represented by their name, without any kind of box. References in state variables are represented with full lines, while those in stack-based variables with dashed lines. As in Simula [27] we have used the '`:-`' notation for reference assignment.

During the execution of the function both formal parameters refer to the same object. The two move instructions would operate on the same object, and could lead to surprising behaviour.

While the first example refers to a typical built-in type (integer) and the second to a user-defined type (Point), these are similar cases. We have two stack-based

```
Point = { x,y:Int;
          move(Int,Int)
        }
p : Point;
p :- new Point;
p.x := 1;
p.y := 2;
f(p, p);

f(p1, p2 : Point)
{
  p1.move(10,20);
  p2.move(10,20);
}
```

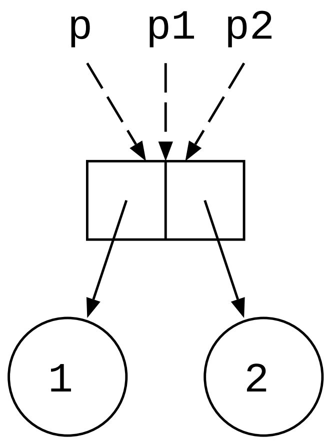


Figure 2.2: Aliasing in `Point` objects

variables, in this case formal parameters, which refer to the same object. This kind of aliasing we will call *dynamic aliasing.*

As in [42, 41] we will use the terms:

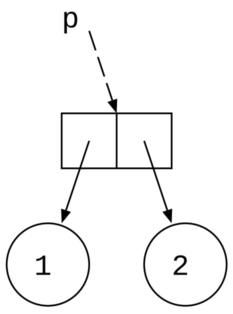
- *dynamic aliasing* when stack based *variables* (including parameters) are involved (as above),
- *static aliasing* when only *state variables* of objects are involved.

Dynamic aliasing has a lifetime which depends on the execution of functions. It will disappear when the corresponding variables cease to exist when some function exits; thus the use of the word dynamic. On the contrary, static aliasing reflects the structure of the object graph: how the set of existent objects are interlinked, regardless of the variables in the stack of invocation records. It ignores the (dynamic) control flow state of the program, and reflects the more persistent state of the graph of objects.

Static aliasing is more pervasive as it can be created during the execution of a function, survive the end of the function and, at an unrelated point in the program, result in interference between two instructions which do not use common variables.

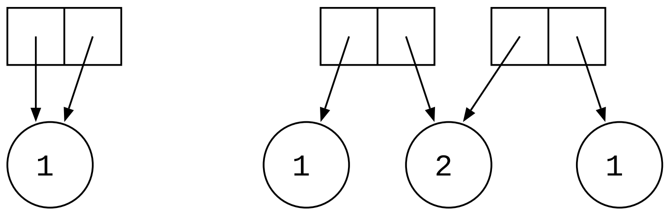
One nice property which holds for built-in types like integer in many programming languages is that objects cannot be statically aliased: they cannot be shared by several state variables of objects. This is a result of the assignment having value semantics: it copies the state of the object and not a reference to the object. Consider the following example involving `Point`:

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

This nice property can be summarised as follows: 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:

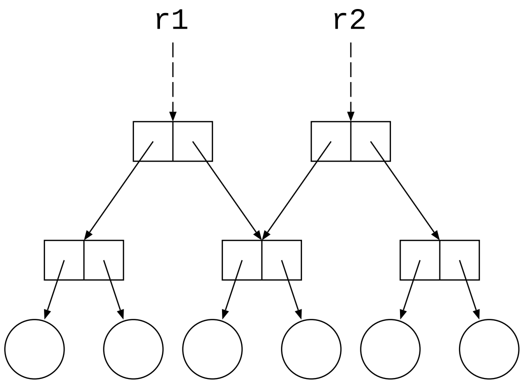


This nice property is true if languages impose some ‘reasonable’ constraints, like banning the explicit use of pointers, dereferencing and the ‘address-of’ operator. Such constraints tend to be incorporated in modern languages like Eiffel or Java. In more permissive languages like C++ [71] ‘anything is possible’, including storing in heap based objects pointers to stack based objects, with the danger of creating *dangling references*: references to something that has ceased to exist.

### 2.1.3 Sharing of state

In many object-oriented languages (eg. Smalltalk [35], Java [8]) 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);
```



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 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 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 [62], 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.

However, accidental substructure sharing is still possible, because the mechanism does not prevent an object of an expanded type from referring to objects of non-expanded types, which can be shared themselves. In this case we could have `Rectangle` expanded but `Point` declared accidentally as non-expanded.

### 2.1.4 Sharing and Unbounded Linked Structures

While in the previous example it is at least possible to prevent sharing by declaring both `Point` and `Rectangle` as expanded, not always expanded types provide sufficient support to prevent unwanted sharing. In the case of unbounded linked structures such as linked lists the recursive nature of these types prevents them

```
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);
```

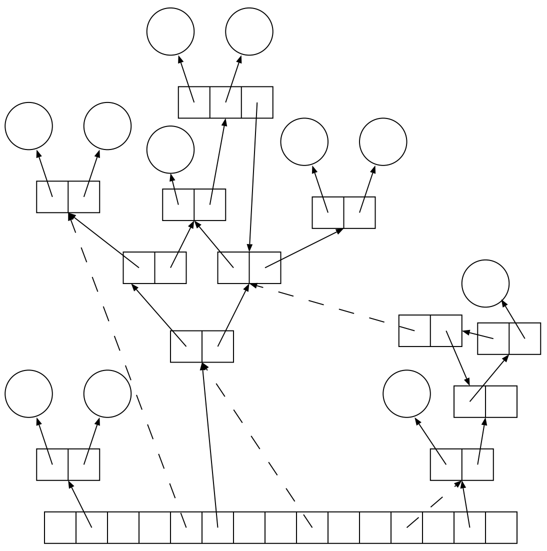


Figure 2.3: An array of shapes

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 2.3.

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, for example due to a bug in the implementation of some operation that causes the unwanted sharing.

### 2.1.5 Data Abstraction and Encapsulation of State

According to [75], 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 [41], and a similar attitude towards encapsulation can be perceived in [22]. Also [42] 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. An example is presented in Figure 2.4.

- The implementation of a *binary method* [14], while manipulating several instances of the data type, may cause sharing of their states. This is illustrated in Figure 2.5.

```
Node = { val:Int; nxt:Node;
         link(p:Node)
         {
           self.nxt :- p;
         }
       }
T = {
      list:Node;
      break(par:Node)
      {
        par.link(self.list)
      }
    }
t:T; n:Node;
...
t.break(n);
```

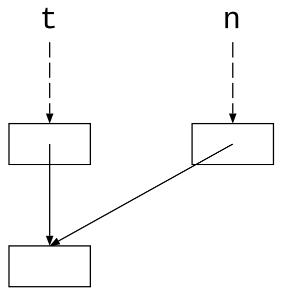


Figure 2.4: An object from the state captured by an external object

```
T =
{
  list:Node;
  break(t:T)
  {
    self.list.link(t.list);
  }
}
t1,t2:T;
...
t1.break(t2);
```

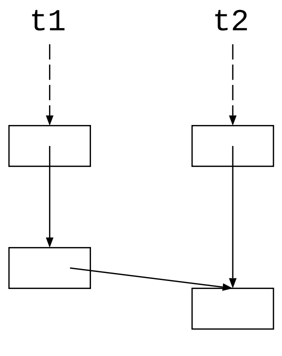


Figure 2.5: Creating substructure sharing between two instances

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

## 2.2 Approaches to Control of Sharing

### 2.2.1 Syntactic Control of Interference

Even before the widespread use of pointers, it was recognised that the use of call-by-reference is a source of problems, due to the possibility of introducing aliasing between parameters (and also global variables). This aliasing, together with the manipulation of global variables, gives rise to interfering side-effects that Reynolds denotes by *interference*.

In [69] he proposes that syntactic restrictions are introduced so that there is no interference between identifiers (eg. of variables or procedures). This means that for example, a procedure cannot assign to a global variable used by another, or that in an invocation no two arguments can be the same variable or a global variable used by the procedure. To enable constructive interference, it is proposed that interfering identifiers must be grouped into a collection named by a single identifier. Also, to avoid being unnecessarily restrictive, the concept of *passive* phrases is introduced: if no assignments to free variables are made an expression is passive, and two such passive expressions do not interfere even if they share free variables.

Although being a classic reference to some of the concerns that should be taken more seriously in language design, the scope of this proposal is somewhat limited. It does not consider pointers and it fails to address problems caused by arrays, like the use of call-by-reference together with array indexing. An invocation like '`f(a[i],a[j])`' can cause the two formal parameters to refer to the same array element, something which should not be allowed. However, as the subscripts can be the result of some arithmetic expression, this cannot be prevented at compile time without being overly restrictive.

In Euclid [55, 67, 24] there were concerns about aliasing which resulted in restrictions similar to Reynolds, preventing aliasing between formal parameters or parameters and global variables. Moreover, arrays were also considered and, recognising that static checking would be overly restrictive in the case of subscript calculations, it was proposed that the potential for aliasing is detected statically and that run-time checks are made: the so called *legality assertions* [79].

With the recognition of the pitfalls of global variables [80] on one hand, and with the introduction of dynamic data structures via references/pointers on the other, the problem shifts from identifiers to unnamed objects and references. These approaches that focus on identifiers as the sole channels of interference become powerless on their own: they can serve at most as a complement. The main problem becomes the interactions due to unnamed objects linked via references; our approach addresses precisely this problem.

### 2.2.2 Banning References to Variables

A basic issue is what a reference can refer to. When references/pointers were introduced in many high-level languages, as in Algol 68 or C, it was made possible that they refer, not only to whole dynamically allocated objects, but also to local variables of some procedure or state variables of an object. The address of a variable could be obtained, either implicitly, or through an 'address-of' operator like '`&`' in C.

While this can be thought of allowing great generality of expression, it causes problems. It becomes possible that the value of a variable changes with no assignment to the variable being made, which makes reasoning about the program more difficult. In our terminology, it confuses two concepts which should be kept separate: variable and object. Moreover, in languages like C it is possible that a reference to a stack-based variable is returned from a function and is used after the variable ceases to exist—the so called dangling references.

The obvious solution to the problem is to allow references only to dynamically allocated (unnamed) objects, and not to variables themselves. This clearly distinguishes the concepts of object and variable, as we argue: even if part of an object, it is the object and not the variable which is (directly) shared; a variable can only be changed through assignment.

Although these problems have been recognised and the obvious solution proposed as early as in [78], languages in wide use like C++ still offer the full freedom, with all its problems. Common practice in modern language design tends, however, to correct the problem, by banning the possibility of 'references to variables'; examples are Eiffel and Java.

### 2.2.3 Part Objects

A rectangle is an example of a *composite* object. It is more appropriate to say that a rectangle is composed of two *part* objects (two points), than to say that it is associated with the points. The *is-part-of* relationship, which is one of the basic data modelling concepts, cannot be represented by having just plain references to objects, as discussed in [50]. The issue is whether a given point is part of a rectangle or it is independent and referenced by several objects.

Many languages allow these two alternatives to be expressed. For rectangles we may wish that the state variables contain, not references to points, but points themselves. This last case is indeed the norm in classic procedural languages like Pascal or C; these languages offer records which can be made up of other records. Proposals for part objects in object-oriented languages can be found in [13, 70]; part objects are possible in languages like C++ and Beta [53, 60].

This possibility of choice has been neglected in some object-oriented languages in favour of only providing the general 'variables are references to objects'. Notable cases are Smalltalk and Java. In Java built-in types are 'special': we can have 'a point is made up of two integers', but not 'a rectangle is made up of two points'; we can only say 'a rectangle is made up of two references to points'.

Unfortunately, even if a language provides 'a variable contains an object', although it makes possible to mimic the conceptual modelling of objects, it does not solve the problems we have described. The reason is that the fundamental issue (whether an object may be shared) does not depend on the object being or not a physical part of another, but on whether it is possible to obtain (and store elsewhere) references to an object. Languages which offer part objects (like C++ and Beta) typically allow references (or pointers) to be obtained so that operations may be invoked on these objects. These references to a part object may be stored by the invoked function in some object, creating sharing.

Furthermore, a drawback with physical part objects is the inability to exploit subtype polymorphism, as the part object must have a fixed size and cannot be of different classes at different points in time.

A proposal [50] for object-oriented databases uses the concept of *composite link* as a substitute to physical containment. However, it is concerned with relative lifetimes of objects, and does not prevent part objects from being referenced by third parties, nor the existence of dangling references.

### 2.2.4 Object Ownership

Another proposal towards reasoning about linked structures and modeling composite objects is described in [47], where the concept of *object ownership* is introduced. Every object manipulated through references has an owner (an object), constant throughout its lifetime, each object being existence-dependent upon its owner (i.e. it ceases to exist when the owner does so).

Keywords `private` and `protected` are introduced (which should not be confused with the same keywords in C++), as well as the concept of *proprietorship*: every variable (of reference type) has a proprietor object; the proprietor of a `private` variable is 'the object in which it is declared' (by which the authors must mean 'the recipient object of the invocation', which in the case of parameters turns out to be quite artificial); the proprietor of a `protected` variable is 'the owner of the object in which it is declared'; the proprietor of a `public` (the default) variable is `Void`.

A rule is stated that the owner of an object referenced by a variable is the proprietor of that variable. However, no language mechanism which could enforce the rule statically is presented. The authors simply propose that a run-time check is made; essentially, the reference assignment is redefined to cause a run-time exception if the proprietors of the assigned-to and assigned-from variables are different. This requires run-time support and corresponding overhead, something that, together with the possibility of having unexpected exceptions, makes the concept unsuitable for production code (althougt it may serve during the test phase).

Another weakness of the proposal concerns the failure to make a distinction between being owner of an object and temporarily using an object received as a parameter; this is not realistic in what concerns parameter passing, as briefly hinted at in the conclusions of the paper.

A final remark we make is that, via public state variables, an object can reference objects that are owned by `Void` and that can be referenced by any other public variables. This means that `public` state variables of an object stored in a `private` state variable can escape the encapsulation provided by the mechanism. Unexpected substructure sharing and corresponding problems may, therefore, happen.

### 2.2.5 Block Structure and Nested Types

Languages like Simula and Beta support block structure, and in particular nested classes; this provides a form of locality with interesting properties as discussed in [59, 17, 68].

As an example (from [59]) a class `token` declared within class `grammar` becomes local; we can then declare instances of `grammar` '`g1`' and '`g2`', and we have that '`g1.token`' and '`g2.token`' are different types. This avoids accidental mixing of tokens from different grammars. Moreover, unlike C++, where nested classes are only a way to structure the namespace, in Beta a nested class can access variables in the containing class. In the example, a set $T_1$ of token objects can access a state variable of a corresponding grammar object $g_1$, and the same for a set of tokens $T_2$ and grammar object $g_2$; a token object can be seen as 'belonging' to a given grammar object.

However, this separation between tokens of different grammars cannot be obtained in general. One example is if the instances of `grammar` are accessed through references: this implies the use of a common '`grammar.token`' type in accessing the tokens; if tokens are to be linked between themselves, there can be some accidental cross-linking between tokens from different grammars.

Also, nesting does not prevent substructure sharing: a nested type can have a component which is from a 'global' type, and which can be shared. Although a useful mechanism on its own, nesting does not solve the problem of sharing prevention.

### 2.2.6 Expanded Types

Primitive types, like integer, commonly receive a special treatment—integer variables hold an object, which cannot be referenced from elsewhere. This is assured by making both assignment and parameter passing have value semantics: they copy the value associated with the object instead of a reference to the object. In the resulting complete absence of aliasing we can simplify the description and say that a variable holds a (representation of a) value. When this happens we can say that we have a *value type*. This is the case of primitive types in Java.

It is natural to extend this to non-primitive types. One such attempt has been, as we have mentioned, Eiffel's expanded types [62]. In Eiffel, any user-defined type can be either a reference type or an expanded type. Primitive types like integer and boolean are provided as expanded types.

Expanded types in Eiffel are an example of the introduction of a property, applying to all data types, which reduces the special treatment that some of them—typically the primitive types—are subject to. (This is something which we advocate, as will become clear later on.)

Expanded types can be used to build composite objects. Moreover, unlike when using reference types in part objects, we are sure that the part objects which are of some expanded type will not be shared with any other objects.

A drawback of expanded types is that we cannot pass objects to a function to perform some update in-place, because parameter passing copies the object. In the rectangle example, expanded types makes it impossible to pass references to rectangles to a function which performs some in-place update like enlarging them. Also, as for part objects and for the same reason, it is not possible to have subtype polymorphism for expanded types.

The real problem of expanded types is, as we have mentioned, that an object of an expanded type may contain references to other objects which may themselves be shared. This means that objects from a user-defined expanded type are not in general self-contained objects, in contrast with primitive types such as integer.

We have then, a difference between primitive types and general user-defined expanded types, namely the possibility of substructure sharing. This possibility can be, as we have discussed, a cause of unwanted interference. For this reason expanded types fail in fulfilling the requisites for value types. In a very concrete sense primitive types remain 'special'.

### 2.2.7 Linear Types

Girard's *linear logic* [34] was a source of inspiration for experimental type-systems, and the origin of the term *linear types*. A variable holds the single reference to a linear object and can only be used exactly once; duplication and deallocation must be explicit.

These type-systems were originally targeted to functional languages as a way to allow update in-place and avoid the need for garbage collection. The issue was therefore not a semantic one—whether values are shared is meaningless in semantic terms—but an implementation one. Possible benefits are, however, obtained at the cost of forcing the allocation and deallocation of (the representation of) linear values to be explicit in the program text. We are interested in imperative languages, with mutable objects, where sharing becomes relevant for the semantics;

here linear types can possibly make a difference by preventing some accidental sharing from occurring.

Lafont [54] presents a linear functional language where there is exactly one reference to each value. While this has the advantage that no garbage collection is needed, it suffers from the drawback that results from computations cannot be shared. A shared value can be represented by a pointer to a closure—which forces its reevaluation every time the value is used—or by copying the value each time a reference to it is copied. Either way is less efficient than the usual practice in functional languages; the proposal is unsuitable for realistic use.

Wadler [74] presents a linear type-system for functional languages where both linear and non-linear types are allowed to coexist. A restriction is imposed that nonlinear data structures cannot have linear components. This causes an asymmetry and some separation between linear and nonlinear types in terms of building a new type from existing ones: linear types can always be built using any existing type, but once defined they cannot be used as a component for a new nonlinear type. This is a weakness from the software engineering point of view. (In our approach there is also a binary classification of data types but no such restriction is imposed.) The motive for this restriction comes from the 'use-once' nature of linear types; if a linear value were allowed as a component of a nonlinear data structure, as this structure can be duplicated, the linear value could be used several times.

Baker presents [9] *Linear Lisp*, a language with assignment but no sharing and no need for garbage collection. The use of linear types is also discussed in [11], which advocates the use of a graphical representation of a program as a data-flow model to circumvent the difficulty to read programs in linear languages due to the typical use of multiple returned values.

Even though linear types may be a relevant concept for language implementation, as in the case of functional languages, both 'use-once' and 'one-reference' properties make them an unsuitable mechanism for imperative languages:

- A procedure may perform some operation on a mutable data structure, as opposed to a function returning a result; the data structure will typically be used afterwards by other procedures or functions, contradicting the use-once rule.

- Also, the one-reference-only property is not appropriate: a function or procedure operating on a given data structure will frequently use temporary reference variables to traverse the structure; therefore there will exist two references to some object: one from other objects in the structure and another from the temporary variable.

The essential reason why linear types are not adequate for imperative languages is the fact that they do not make the distinction between static and dynamic aliasing, something not surprising since they were conceived for functional languages.

Lastly, but a significant weak point in linear types, is the fact that they do not address the important issue of substructure sharing: two linear objects can share a given non-linear object.

### 2.2.8 Unshareable Objects and Unique Pointers

A proposal related to linear types is presented in [64]. It introduces unshareable objects (*u-objects*)—there can only be one pointer to them in the system: a unique pointer (*u-pointer*). It is proposed, as a mechanism to achieve this, that assignment from a u-pointer copies the pointer and nullifies the source variable, i.e. *moves* the pointer.

However, the claim that there is only one pointer to a u-object in the system is not true. Parameter passing is not treated as reference assignment, and does not nullify a u-argument (the contrary would be unrealistic in an imperative language). We can have, therefore, two pointers to a given object: the variable used as argument and the formal parameter. The paper sees parameter passing in common object oriented languages (mentioning the case of Eiffel) as not duplicating pointers; i.e. that a parameter refers to the variable used as argument and is not 'another' reference. We find this quite artificial: a parameter is no less a reference than a local variable, and takes part in the aliasing problems that u-pointers are supposed to address.

Even if the intention was to have just one pointer to an u-object in a given procedure as opposed to the whole system (something not discussed in the paper), this does not hold anyway. Although some simple cases can be prevented, such as passing the same u-variable in two different arguments, in general it cannot be prevented due to the unrestricted aliasing which can exist for normal objects containing u-variables. As an example, a reference to a u-object, residing in a state variable of a normal object, can be passed as argument to a procedure in which this object can also be reached; it is then possible, without being detected, to invoke another procedure with two different arguments referring to the same u-object.

This proposal, which attempts to achieve the one-reference-only property of linear types is, thus, fundamentally flawed. It constitutes an example of how unrealistic it is to statically prevent dynamic aliasing in imperative languages (without dealing with sharing in the object graph in an appropriate manner).

Again, as for linear types, the important issue of substructure sharing is not addressed by this proposal: an u-object does not prevent objects it refers from being shared by other objects.

### 2.2.9 Inductive Data Structures

It is proposed in [37] that programmers should be able to classify linked data structures as either *inductive* or *non-inductive*; an inductive structure has no cycles and each node has at most one parent. Inductive structures have benefits in terms of program analyzability: the component substructures do not share storage, breaking a link yields two independent structures, and a traversal following links does not visit the same node more than once.

It is interesting that this proposal (unlike linear types and u-objects) distinguishes the data structure itself from temporary variables used to access it. There is at most one pointer to a node of an inductive structure from other nodes, but there can be more pointers from local variables, unlike in unshareable objects. The proposal distinguishes between dynamic and static aliasing and recognises the practical need in imperative languages for the use of temporary variables in building or traversing a data structure, even though there is no sharing in the data structure itself. For this reason this proposal is more realistic than linear types or u-objects in terms of imperative languages.

Although just one of a range of mechanisms devoted to describing the ‘shape’ of a data structure, as considered below, it is a mechanism which emphasises the importance of substructure disjointness and which is minimal in terms of choices offered to the programmer (a binary classification); these characteristics are present in our own approach.

The proposal is, however, that the declaration is not a full part of the type-system but more to be treated as a directive: either as an unchecked promise from the programmer, or as an aid to the compiler in choosing an appropriate form of aliasing analysis. Moreover, it is only contemplated declarations of recursive types whose fields are either the type itself or scalar types; the possibility of fields being references to other (possibly non-inductive) types is not considered.

### 2.2.10 Euclid's Collections and FX's Regions

One concern in the design of Euclid [55, 67] was the possibility of aliasing in accessing dynamically allocated objects. Pointer variables in Euclid are declared as being pointers to a *collection*; there may exist several collections of objects of the same type. A collection is a declared entity (with a name), in a sense like a variable. A dynamically allocated object must be an element of a collection and it is enforced that a pointer to a collection points only to objects within that collection. Different collections contain non-overlapping sets of objects.

A serious limitation of collections is that they 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: although they contain dynamically allocated objects, collections cannot be themselves dynamically allocated. The number of collections and respective lifetimes is determined by the activation records in existence. The above means that neither the number of collections can be related to the size of the object graph nor is it possible to use collections to obtain a hierarchical structure of objects. As a result, collections are a weak means of organising the object graph.

In the language Turing [43], it is further imposed that collections cannot be passed as parameters or declared in subprograms, making the possible uses of collections in Turing even more restricted.

As described in [57], the language FX has 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 part of a more powerful effect system, they suffer from the same essential problem: regions are named entities that cannot be part of a dynamically created data structure.

### 2.2.11 Islands

Hogg recognised the problems posed by the uncontrolled possibility of static aliasing in object structures. Towards a solution he introduces the concept of *Is-*

*lands* [42]. An island is the transitive closure of the set of objects accessible from a *bridge* object. The bridge is the only entry point to the island: no other object that is not a member of the island has a reference to an object in the island.

Like collections and regions, islands permit obtaining groups of disjoint objects. Islands represent, however, an advance when compared to collections or regions. Islands unify the unit of grouping with the main unit of abstraction—instances of classes; an island is represented by a bridge object which can be subject to the usual manipulations. On the contrary, a collection has no use in itself; no operations can be applied to collections. Islands also overcome the main problem with collections: an island is not a named entity. Islands are dynamically created and can be used to obtain a hierarchical organisation of the object graph. In these aspects islands are similar to our own mechanism; we will make a comparison between them later on.

### 2.2.12 Shape Description Approaches

Some recent proposals have been made towards enhancing the description of recursive structures by describing their possible 'shape'. The description is intended to be relatively precise, and the program is subject to some static checking mechanism towards validation. These proposals are of a more experimental nature, and have not been put to the widespread use as many of the above described mechanisms.

#### ADDS

In [38] ADDS (Abstract Description of Data Structures) is presented. It allows a recursive data type to be augmented with a description about *dimensions* (different paths which can be traversed), and the *direction* (along a given dimension) that each field traverses. It also allows specifying whether different dimensions are independent (disjoint) or dependent (potentially leading to a common node); whether traversing in a given direction from different nodes leads always to different nodes (traversing *uniquely*); and whether the linked structure is *circular*.

The authors claim that the ADDS declarations are then validated by a static analysis that determines at which points in the program the data structures conform to the declarations. The lack of conformance is not an error, and the mechanism allows for the recovery of the property after a temporary violation; the idea is that the property is only exploited by some program transformation at the points it is valid.

### Graph Types

*Graph types* [51] allows the description of data structures with sharing. A graph type consists of a backbone, which is a spanning tree, augmented with extra edges that are functionally determined by the backbone. The extra edges are specified by a language of regular *routing expressions*: regular expressions over a language of directives like 'move up/down' or 'verify if this is root/leaf'.

Although many common data structures can be described, even for simple cases like a doubly-linked circular list the routing expressions become cumbersome. A reason for this, which is one of the problems with this approach, is the distinction between edges in the backbone and extra edges; this division is artificial for some data structures.

More importantly, as the extra edges depend functionally on the backbone, they contain no information in themselves: two instances of a given graph type with identical backbone will have identical extra edges, so two instances can be compared by looking at the backbone only. Structures where the pattern of sharing depends on the value itself, or provides information in itself cannot be represented. Therefore, the sole purpose of the extra edges is to support efficient access to the data structure. However, traditional pointer manipulation is not permitted. Extra edges can only be read, but cannot be directly assigned; they are reevaluated automatically when an operation on the underlying backbone is performed. This appeals to a sophisticated mechanism for optimising the amount of reevaluation needed.

In [52] the authors remove the restrictions that extra edges are functionally determined by the backbone. Extra edges are specified by *edge constraints* that allow different extra edge configurations for a given backbone.

### Shape Types

Related to the previous approach, *shape types* [32] allow the 'shape' of a data structure to be described; this is done using a formalism based on context-free graph grammars [73]. Like the previous proposal, it is suitable for structures in which the pattern of sharing is fixed; although possible in theory, it is quite unrealistic to use the mechanism for structures where the pattern of sharing depends on some value or contains information itself.

The manipulation of shape types is through *shape transformers* (essentially single step rewritings); an algorithm for static checking that a transformer preserves a shape invariant is presented. A notation for integrating shape types and transformers in C is described, together with a translation into C code.

The manipulation of shapes can only be made through the transformers, no direct pointer operations are allowed. Although the goal of banning the traditional error-prone pointer manipulation in favour of more high level manipulation primitives is desirable, it is difficult to achieve without incurring problems. In this case in particular, as a shape is an independent entity, and no programmer visible pointers are allowed into internal nodes, no traditional algorithms can be written to traverse a shape. While traditionally recursive data structures are often manipulated by recursive algorithms, here the programmer is limited to applying transformers. Moreover, as a transformer is to be seen as an atomic operation which preserves the shape invariant, transformers cannot be nested. Also, there are several limitations on the class of transformers that can be used in the mentioned extension to C. We have, therefore, serious doubts about this proposal being suitable for realistic use.

These mechanisms are described for individual recursive types; the situation where nodes of several types are linked is not considered. The proposals mention that nodes have some 'value', typically some scalar like integer, which is not considered to cause problems. This does not consider the common situation where the 'value' field is in fact a reference to an object of some user-defined type, and the consequent possibility of substructure sharing.

## 2.3 Discussion

We have described how accidental state sharing is a source of unexpected program behaviour that can be difficult to correct, and how the possibility of it happening (even if that is not the case) is an obstacle to reasoning about the program or performing program transformations. We have argued that current language mechanisms do not provide appropriate support to prevent accidental sharing, essentially by disregarding the fact that an object is not normally self-contained, and by neglecting substructure sharing.

While the emphasis of research was first on (named) variables and a limited form of dynamic aliasing, the widespread use of references and linked structures shifted the focus towards (unnamed) objects and to include static aliasing.

In spite of the trend towards a 'civilised' use of references by banning explicit dereferencing and the *address-of* construct, the proliferation of their use, culminating in the 'all variables of user-defined types are references' in Java, makes the possibility of accidental object sharing more present than ever.

Some concepts like expanded types or part objects have some use, but are not sufficient in themselves: they do not prevent substructure sharing, with the disadvantage of not allowing subtype polymorphism. Physical containment is not the solution to preventing unwanted sharing.

We consider linear types unsuitable to be adapted to imperative languages. They fail to distinguish between static and dynamic aliasing; in practical terms, they fail to distinguish between a data structure itself and the temporary variables used in some traversal. Moreover, substructure sharing is, again, not considered.

Some attempts made to organise the object graph, like collections and regions, use named entities; this is not appropriate to obtain hierarchical structures of unnamed objects. Islands overcome this problem, being the proposal most related to our own balloon types.

Mechanisms have been proposed to describe the shape of recursively defined data structures. We have, however, some doubts about whether they are realistic and would be accepted by programmers.

Language mechanisms to control sharing are examples of (using the expression from [37]) *language design for analyzability*; it is not enough that they provide useful information/invariants to reason about the program: if they are not simple from the programmer's point of view they will not be successful.

Something which recurs in the description of several proposals is the mention of *values* or *scalars* of *primitive* types: it is assumed that nodes of the linked structure being described contain (apart from references to other nodes) some value which is self-contained (the examples typically use the integer type). This means that, regardless of the virtues the mechanism may have, it will not handle the problems which exist if the 'value' is of some user-defined type, which may not be self-contained.

Primitive types are, thus, treated as 'special'; they possess some properties which cannot be counted upon in user-defined types. This is something we consider very undesirable for the scalability of any mechanism; the more complex situations are bound to appear in large programs, where the percentage of user-defined types is large.

To change this situation, more information should be considered in the description of a data type, namely concerning sharing, so that language mechanisms or reasoning about a language ignores whether a data type is primitive or user-defined, and only looks at its definition. This is one of the essential points in our mechanism.

We conclude by compiling a list of the relevant points that we have discussed, which should be taken into account by language mechanisms:

- Single objects are not self-contained: reachable state and substructure sharing must be addressed.
- Physical containment or similar implementation-oriented mechanisms do not provide a solution.
- Duplication of references is inevitable, because imperative languages use temporary variables to traverse already existing linked structures.
- Use of a reference by a temporary local variable must be distinguished from its incorporation into a linked structure of undetermined lifetime; the different nature of dynamic and static aliasing must be recognised and addressed.
- Variables are named, objects are unnamed. It is not appropriate to use named entities to organise a graph of unnamed objects.
- A mechanism should be, not only useful, but also syntactically simple and conceptually relevant to be accepted by programmers.
- It should be irrelevant whether some data type is primitive or user-defined; this is not accomplished by current abstraction mechanisms.

# Chapter 3

# Balloon Types

Here we introduce *balloon types*, the concept which is the core contribution of this thesis. This chapter gives a general overview and describes the *balloon invariant*; more specific details, in particular the balloon type-checking mechanism, will be addressed in subsequent chapters.

## 3.1 The Idea

We have discussed some problems caused by substructure sharing, have argued that current data abstraction mechanisms provide an insufficient form of encapsulation, and have advocated that, in defining encapsulation of state, the transitively reachable state should be considered.

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.

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 3.1 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 2.3 with Figure 3.1, 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

```
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);
```

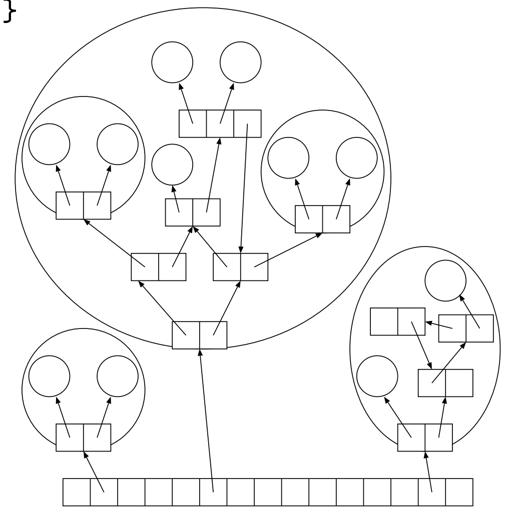


Figure 3.1: An array of balloon shapes

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.

## 3.2 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 3.1 (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 3.2 (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 3.3 (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 3.4 (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 $B$.

Figure 3.2 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.

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

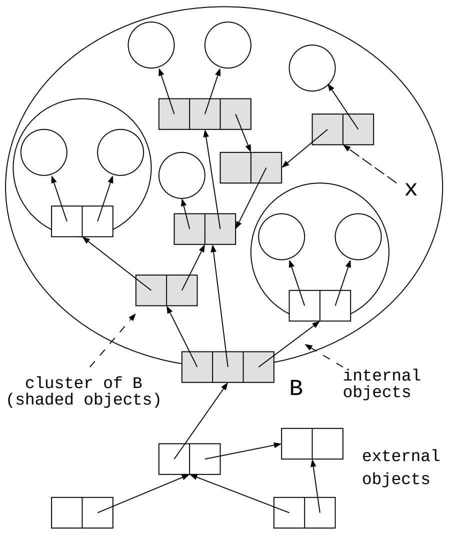


Figure 3.2: A balloon B and its internal and external objects

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. We now present the three components of the invariant in detail.

### 3.2.1 $I_1$: Balloons Have a Single Owner

The first and simplest requirement for the usefulness of the concept is that no two references stored in objects denote the same balloon, i.e. that a balloon has a single owner. This can be motivated by the ‘array of shapes’ example in

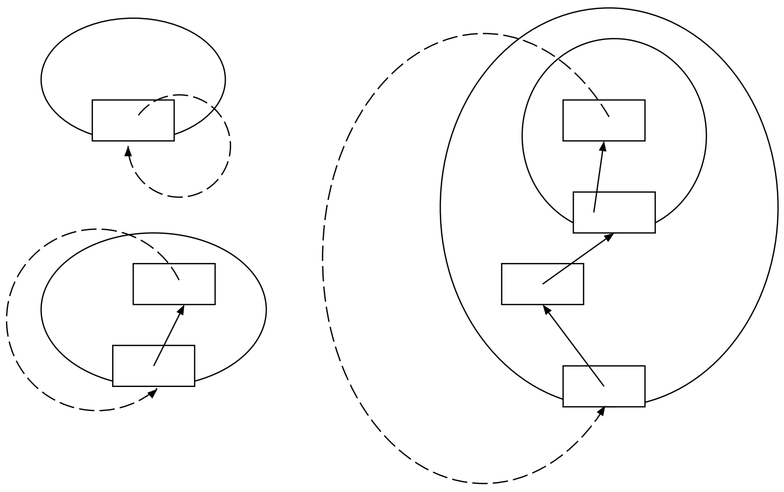


Figure 3.3: Examples of situations which $I_2$ prevents

Figure 3.1. If several entries of the array could denote the same balloon, there would be interference between different iterations of the loop, even if the other components of the invariant ($I_2$ and $I_3$) were verified.

The *islands* proposal has similarities with balloons; however, several references to a bridge object are, in general, allowed from other objects. This would result in possible interference in the array of shapes example, which would prevent transformations such as parallelisation or loop reordering.

The component $I_1$ would be all that is required if objects were atomic and did not reference other objects. In the general case reachable state becomes relevant; thus the other two components of the invariant.

### 3.2.2 $I_2$: Balloons are Well-founded

The $I_2$ component expresses that a balloon object is not internal to itself, or in other words, that there are no cycles in terms of clusters and inter-cluster references (references to balloon objects). Without $I_2$, the situations in Figure 3.3 would be possible.

$I_2$ can be seen as equivalent to a reasonable assumption normally made in set theory. Suppose we draw a parallel between balloons and sets, and associate a balloon $B$ with a set whose elements are the non-balloon objects in the cluster of $B$ and the sets associated with the balloons referenced by $B$ or by any non-balloon in the cluster of $B$. An example is shown in Figure 3.4.

Then, $I_2$ parallels the axiom of foundation in set theory (see e.g. [33]), which essentially says that sets cannot 'contain themselves' (directly or transitively). If

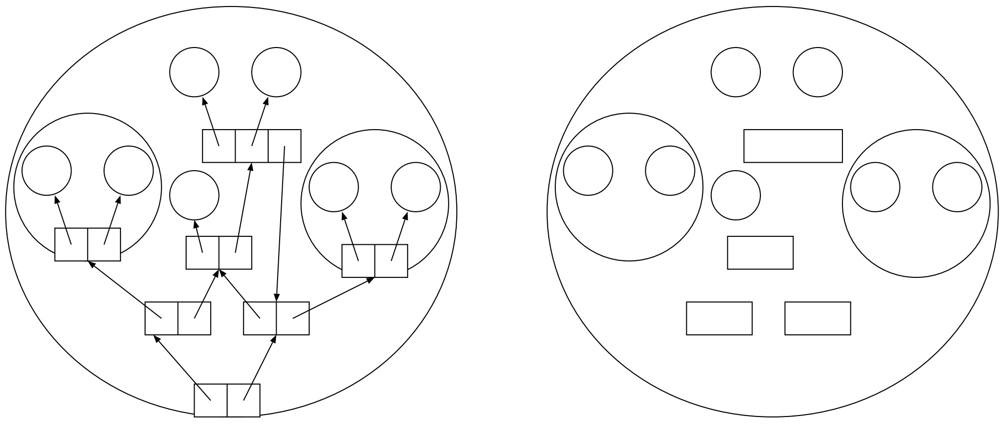


Figure 3.4: Balloons and (well-founded) sets

we start from a balloon and we follow the references to contained balloons, the process must stop in a finite number of steps; we cannot have a cycle of references which results in an infinite loop.

From a practical perspective, we want to have the assurance that, if we start from a balloon $B$ and access a state variable, we 'enter' a nested object from where we cannot 'escape' and reach back to $B$. Consider the following program:

```
T = balloon { b1,b2:T ... }

f(b:T)
{
  do_something(b.b1);
  do_something(b.b2);
}
```

We want to be able to assume that there is no interference between the two instructions; that they operate on disjoint objects. Such would not be the case if $I_2$ is not enforced and cycles are allowed.

### 3.2.3 $I_3$: Internal State is Encapsulated

The third component of the invariant expresses a strong form of encapsulation. By stating that external objects cannot reference internal objects, it draws a boundary, and makes 'internal' and 'external' as used here correspond to an intuitive meaning of these words.

Without $I_3$, even if the other two components of the invariant were verified, the so called internal objects of a balloon $B$ could be shared with external balloon objects. In Figure 3.5 we show an example where this happens: internal objects

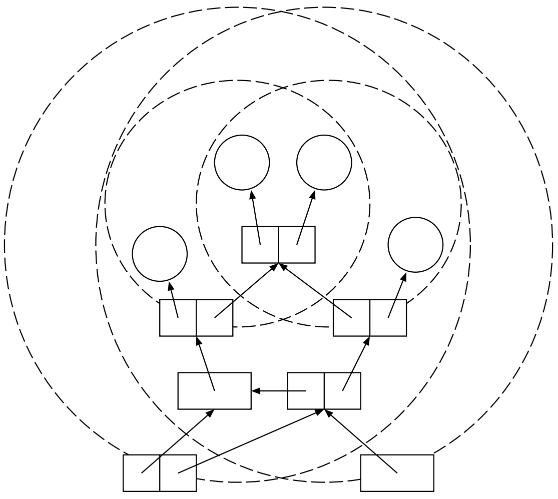


Figure 3.5: A situation which $I_3$ prevents

are inside dashed elipses, and different dashed lines overlap. This is a kind of sharing pattern which balloons are intended to prevent; such is expressed by $I_3$.

### 3.2.4 An Equivalent Invariant

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

**Definition 3.5** ($I_4$) From all objects that make up a cluster, at most one is a balloon object.

We will show that we obtain an equivalent invariant if we substitute $I_4$ for $I_3$ in the balloon invariant. Towards that we first derive some lemmas.

**Lemma 3.6** *Given two balloon objects, $B_1$ internal to $B_2$, if the balloon invariant holds then $B_2$ is external to $B_1$.*

**Proof** Suppose, by way of contradiction, that $B_2$ is internal to $B_1$. This can only happen if $B_2$ is referenced by $B_1$, if $B_2$ is referenced by a non-balloon in the cluster of $B_1$, or if $B_2$ is internal to a balloon $B_3$ which is internal to $B_1$. In all three cases $B_2$ is referenced by an object which is internal to $B_2$, contradicting $I_2$. □

**Lemma 3.7** $I_1 \wedge I_2 \wedge I_3 \Rightarrow I_4$.

**Proof** Suppose that the balloon invariants $I_1$, $I_2$, $I_3$ hold and that, by way of contradiction, a cluster $C$ contains more than one balloon object; in particular suppose that it contains two balloons $B_1$ and $B_2$. Then $B_1$ and $B_2$ reference non-balloons in $C$ (possibly the same non-balloon). There are two possible situations:

- $B_2$ is internal to $B_1$. From Lemma 3.6 it follows that $B_1$ must be external to $B_2$. This means that we have $B_1$, an object external to $B_2$, with a reference to an object internal to $B_2$ (a non-balloon in $C$), which contradicts $I_3$.
- $B_2$ is external to $B_1$. Again, $I_3$ is contradicted for the same reason (swapping the roles of $B_1$ and $B_2$).

□

**Lemma 3.8** $I_1 \wedge I_2 \wedge I_4 \Rightarrow I_3$.

**Proof** Suppose, by way of contradiction, that $I_1$, $I_2$, and $I_4$ hold, but $I_3$ does not hold: there is some object $O$, internal to a balloon $B$, which is referenced by an object $E$ external to $B$. Object $O$ is either balloon or non-balloon:

- If $O$ is balloon, it is referenced by either $B$ or an object internal to $B$. Either way, no other reference can exist to $O$, namely from $E$, as this contradicts $I_1$.
- If $O$ is non-balloon, it belongs to a cluster $C$ which contains a balloon: either $B$ or a balloon internal to $B$. Moreover, object $E$ which references (the non-balloon) $O$, belongs to cluster $C$. $E$ cannot be a non-balloon, as in this case it would be internal to $B$. But it cannot either be a balloon, because $C$ would have more than one balloon, contradicting $I_4$.

□

**Proposition 3.9** $I_1 \wedge I_2 \wedge I_3 \Leftrightarrow I_1 \wedge I_2 \wedge I_4$.

**Proof** Combine the two previous lemmas. □

We have obtained an equivalent expression for the balloon invariant: $I_1 \wedge I_2 \wedge I_4$. 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 a static analysis of the candidate program, which will be presented in Chapter 5.

## 3.3 Reference Assignment—the Simple Rule

**Definition 3.10 (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.

By the simple rule, if only a reference assignment were provided by a language, no objects at all could have a reference to a balloon object. Such is made possible by providing the language with a *copy* assignment, as discussed below.

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.

**Example** 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.

Dictionary here is a generic type. Parametric polymorphism, as well as subtyping, will be addressed in Chapter 8. For now we say informally that it will be more useful that 'balloonness' is not a property of a generic type itself but of its instantiations. An implementation of a generic type can be type-checked to assess the correctness of its instantiation as balloon/non-balloon for each possibility regarding the type parameters being or not balloon types.

Balloon types can be an important contribution 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 [56]: '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. 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.

In the example above, the contents of a shape are, in principle, irrelevant to the implementation of the dictionary data type; returning a reference to an internal shape does not prevent reasoning about the correctness of the dictionary implementation. This is to be contrasted with the situation where references to some linked structure used to hold shapes in the dictionary escape accidentally to client code without being returned.

## 3.4 Copy Assignment

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

- The creation of an object using some constructor mechanism:

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

- A general copy mechanism with the semantics of deep-copy (as in e.g. [49]), 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'. 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 possible 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 can make these optimisations more realistic; they will be the subject of further research.

## 3.5 Type-checking Balloon Types—An Introduction

We now give an informal overview of the balloon type-checking mechanism. The essence of the 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 3.6.

The implementation manipulates the state of typically three balloon `BigInts`, referenced by `self`, `other` and `num`. (It is possible that `self` and `other` refer to the same balloon; the analysis works under this possibility of dynamic aliasing.) 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

```
    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 task of the analysis is to make sure that for accepted programs both $I_2$ and $I_4$ hold. 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; or in other words, different clusters which may already have a balloon—*captured clusters*—are prevented from becoming linked (*merging* the clusters), and

```
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 3.6: The `BigInt` data type

- non-balloons in any free cluster $A$ are prevented from being captured by some balloon 'reachable' by $A$ (some balloon $B$ referenced by an object in $A$ or some balloon internal to that balloon $B$).

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,
- the cluster to which a variable points containing a balloon, and
- a free cluster 'reaching' captured clusters.

Regarding the `BigInt` example, Figure 3.7 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 Chapter 5, 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.

With this information the checker would determine that the mentioned assignment '`r.nxt :- p`' breaks the balloon invariant, and would reject the program.

## 3.6 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*.

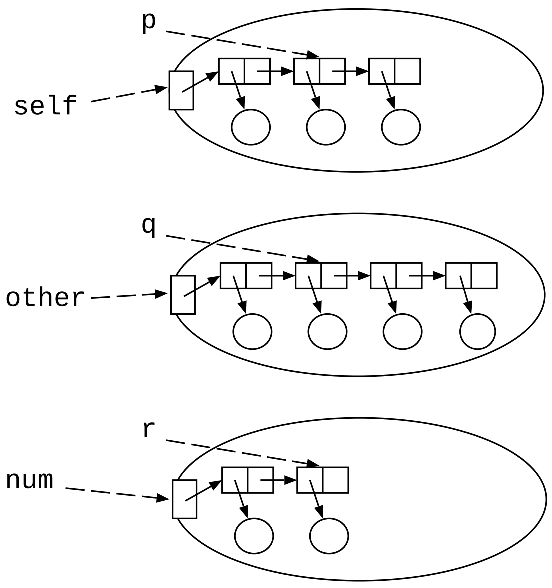


Figure 3.7: A scenario with `BigInt` objects

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. The exact definition of opaque balloon types and respective checking mechanism are described in Chapter 7.

## 3.7 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*: an integer variable is associated with an integer value. (This term is also used in [46], which discusses problems with expanded types in Eiffel.) 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 we have already discussed) they can lead to subtle interference, unlike in primitive types.

It is important, not that the variable physically contains the object, which can be seen 'only' as an implementation issue, but that the group of objects which represents 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 represent 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.

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.

## 3.8 Taxonomy of Data Types According to State Sharing

Currently we have a gap between primitive types and user-defined types. Primitive types have nice properties which, even when they hold in some user-defined types, cannot be counted upon to reason about programs or to perform compiler optimisations. With value types, the traditional behaviour concerning interference exhibited by traditional primitive types can be obtained in user-defined types if desired, and can be counted upon. This means primitive types can truly cease to be ‘special’.

By making the ability to share state an explicit property of a data type, providing the concepts of balloon, opaque balloon, and value types, we obtain a taxonomy of data types with respect to state sharing. The position of a data type in this taxonomy is what matters; whether a data type is primitive or user-defined becomes irrelevant. We present this taxonomy in Figure 3.8: the main binary classification in balloon/non-balloon and the two specialisations of balloon types.

## 3.9 Summary

We have introduced the essential idea of balloon types: to make the ability to share state a first class property of a data type. Balloon types provide a strong form of encapsulation of state by considering, not only the state variables of an object, but the reachable state. This is expressed by the balloon invariant, which we took some care to define, namely the concept of an object internal to a balloon. The invariant concerns the structure of the object graph, ignoring local variables.

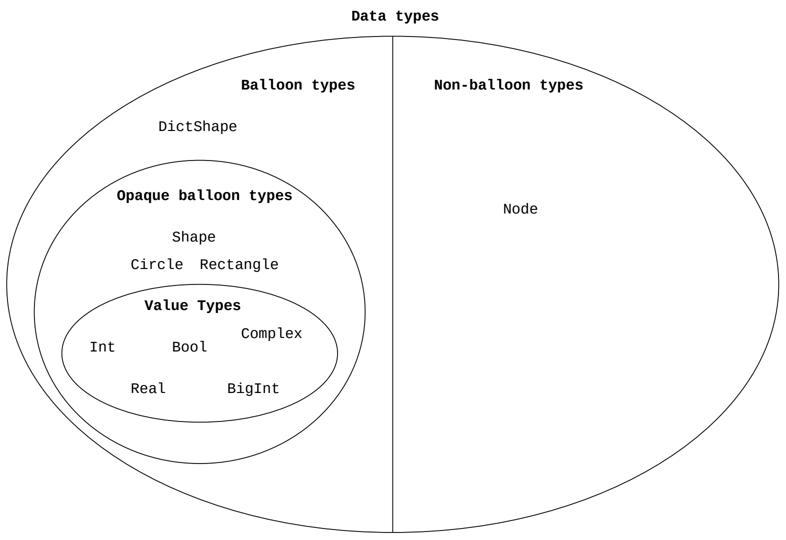


Figure 3.8: Taxonomy regarding state sharing in data types

It essentially states that there is a single entry point to a balloon, there are no cycles involving balloons, and internal objects are encapsulated.

We have presented 'the simple rule' concerning reference assignment. This rule forbids storing references to balloons in state variables of objects, but allows assignments to local variables; this is essential for parameter passing and, in general, for being able to 'use a balloon temporarily'. This reflects the crucial distinction between dynamic and static aliasing. We advocate the use of a copy assignment, which copies a balloon and its reachable state, as the way to store balloons in state variables.

We have introduced briefly the checking mechanism for balloon types and discussed two specialisations of the concept: opaque balloon types constitute truly opaque data abstractions which guarantee that no state reachable by an instance can be manipulated by clients; value types have value semantics, being the generalisation to user-defined types of what happens in primitive types such as integer. We have also presented the taxonomy of data types according to state sharing which results given balloon types and the two specialisations presented.

# Chapter 4

# A Framework for Abstract Interpretation

Although this thesis is not devoted to abstract interpretation in itself, some lessons were learnt in the process of applying it to the problem at hand. Here we present our approach to abstract interpretation in a problem independent setting.

We base our approach on the framework by Abramsky [2], which we take as the starting point. We do not attempt to develop a general framework, such as [26], but simply address some specific issues. Our approach is essentially concerned with the design of base abstract domains that are not lattices and the separation of concerns between summarising information about concrete states and merging control paths.

In the presentation, and without loss of generality, we avoid notational clutter by considering just one base domain, and by focusing on a *safety* relation. The remarks can be generalised to several base domains, and if a *liveness* relation is required the dual remarks apply. We also omit the type $\sigma$ from each relation $R_\sigma$ of a family $\{R_\sigma\}$ (as it is understood from context) and overload a symbol $R$ for the logical relation and each relation in the family.

## 4.1 The Base Domains

It is generally assumed that domains of abstract states are lattices. A reason for this is the need for a join or meet operation, as in defining the semantics of conditionals and loops in the abstract interpretation. Having lattices in the base domains also makes it possible to induce best abstract values for constants of higher-types, as described in [2].

In our experience of developing an abstract interpretation the first concern was obtaining an abstract domain which represents the relevant information to the static analysis in question. There was no concern about obtaining a lattice, and the result turned out in fact not to be a lattice, but simply a cpo which is not even bounded-complete (and therefore is not a Scott domain). The cpo obtained was however the more natural way of summarising the relevant information and should become the base domain for the abstract interpretation.

The particular problem we address (balloon checking) constitutes thus an example which demonstrates that it is not always realistic to assume that the base abstract domain is a lattice. This should serve to rethink some assumptions commonly made in existing frameworks. (Even the assumption sometimes made in semantics of taking a domain to be a Scott domain—which cannot be made in our particular abstract semantics—may come simply from 'tradition' and not as a conscious choice.)

For merging the information regarding different control paths a completion of the base cpo can be used. Different completions can be used depending on the compromise between information loss and requirements in terms of space/time of the static analysis.

Our experience points to the clear separation between two aspects which are commonly merged:

- How to summarise the relevant information about concrete states.
- How to merge the representations from different control paths.

However, not having a lattice in a base abstract domain causes problems. In terms of finding best representatives for higher-type constants, not only is it not possible to use the results from [2], but in general a best abstract value may not exist. We will discuss some sufficient conditions for the existence of such a best representative.

Even for constants of base types this problem arises; a way to have a best abstract representative is if we have an abstraction function $\alpha$, an order on abstract states $\sqsubseteq$, and a relation $R$ defined (like in [2]) as:

$$s \; R \; a \Leftrightarrow \alpha s \sqsubseteq a.$$

There is, however, a problem of practical nature. This assumes an *a priori* order on abstract states. In our experience, for a complex abstract domain the order is not derived as a first step but only after we have characterised what an abstract state represents. This can be done by a concretisation function $\gamma : S^a \to \mathcal{P}(S)$; however, if we have an arbitrary concretisation function, and define the relation through this function ($s \; R \; a \Leftrightarrow s \in \gamma a$), there is no guarantee that there will exist a best representative for each concrete state. Our approach also takes these issues into account.

### 4.1.1 Abstract States and the Abstraction Function

We start by defining the set of abstract states and an abstraction function. Contrary to the classic framework of the Cousot's [25], where the abstraction function is from sets of concrete states to abstract states:

$$\alpha : \mathcal{P}(S) \to S^a,$$

we use, like [2], an abstraction function from a single state to an abstract state:

$$\alpha : S \to S^a,$$

which gives the abstract state that will best represent the given concrete state when an order is imposed on the abstract domain. It will also be used to induce a logical relation [66], as in the framework by Abramsky [2].

We use a surjective abstraction function, meaning that any abstract state is intended to represent directly some concrete state. No abstract states exist just for the sake of the abstract domain being a lattice (and in general that will not be the case). The abstraction function just partitions the set of concrete states into equivalence classes.

### 4.1.2 The Concretisation Function

To each abstract state we make correspond more concrete states, in addition to the ones directly abstracted to it by $\alpha$. This is accomplished by a concretisation function

$$\gamma : S^a \to \mathcal{P}(S),$$

subject to the condition that for all $s \in S$:

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

### 4.1.3 Order in the Abstract States

The concretisation function induces a partial order on the set of abstract states according to set inclusion in the corresponding concrete states. For $a_1, a_2 \in S^a$:

$$a_1 \sqsubseteq a_2 \Leftrightarrow \gamma a_1 \subseteq \gamma a_2.$$

This way $\gamma$ becomes an order-embedding of $S^a$ into $\mathcal{P}(S)$, and we have defined the largest order such that $\gamma$ is monotone:

$$a_1 \sqsubseteq a_2 \Rightarrow \gamma a_1 \subseteq \gamma a_2.$$

**Remark** While in terms of optimality it is desirable to define the largest order, for correctness purposes only monotonicity of $\gamma$ is required. As long as the defined order makes $\gamma$ monotone ($\Rightarrow$) and we are intuitively satisfied with it, there is no real need to prove the reverse implication ($\Leftarrow$). Moreover, we will show below an indirect way to check if we have an order-embedding which avoids the direct proof of the reverse implication.

### 4.1.4 The Logical Relation

We then use the logical relation induced from the safety relation on the base domains derived from $\alpha$:

$$s\ R\ a \Leftrightarrow \alpha s \sqsubseteq a.$$

A question arises: given that the concretisation function defines a relation between $S$ and $S^a$, how does that compare with the derived relation? Intuitively we can hope that the derived relation coincides with what the concretisation function expresses, that is:

$$s\ R\ a \Leftrightarrow s \in \gamma a.$$

This is not, however, necessarily the case even if $\gamma$ is an order-embedding. As a counter-example suppose:

$$\begin{array}{ll} S = \{A, B, C, D\} & S^a = \{a, b, c, d\} \\ \alpha A = a & \gamma a = \{A\} \\ \alpha B = b & \gamma b = \{B\} \\ \alpha C = c & \gamma c = \{C, A\} \\ \alpha D = d & \gamma d = \{D, B, C\} \end{array}$$

If we define $a_1 \sqsubseteq a_2$ as $\gamma a_1 \subseteq \gamma a_2$, we have $C \in \gamma d$ but $C \not R\, d$, because $\alpha C = c$ but $c \not\sqsubseteq d$.

Although possible this is an artificial example. For realistic concretisation functions we may expect the relations to coincide. To check whether such is the case we can use the fact that:

**Lemma 4.1**

$$s\ R\ a \Rightarrow s \in \gamma a,$$

**Proof** If $s\ R\ a$ holds, then $\alpha s \sqsubseteq a$, therefore $\gamma(\alpha s) \subseteq \gamma a$; due to the constraint $\{s\} \subseteq \gamma(\alpha s)$ imposed on $\gamma$, we obtain $s \in \gamma a$. □

As a result it is only necessary to check that the reverse implication $s \in \gamma a \Rightarrow s\ R\ a$ holds to check whether the relations coincide. This is summarised in:

**Proposition 4.2** *Suppose $\alpha, \gamma, \sqsubseteq$, as above, $\gamma$ monotone, and satisfying $\{s\} \subseteq \gamma(\alpha s)$. If we have $s \in \gamma a \Rightarrow \alpha s \sqsubseteq a$, then it follows that $\alpha s \sqsubseteq a \Leftrightarrow s \in \gamma a$.*

Moreover, they will only coincide if $\gamma$ is an order-embedding of $S^a$ into $\mathcal{P}(S)$, that is:

**Proposition 4.3** *If $s\ R\ a \Leftrightarrow s \in \gamma a$ then it must be the case that $a_1 \sqsubseteq a_2 \Leftrightarrow \gamma a_1 \subseteq \gamma a_2$.*

**Proof** Suppose that the relations coincide, and that $\gamma a_1 \subseteq \gamma a_2$. Let $s_1$ be such that $\alpha s_1 = a_1$ (always possible as $\alpha$ is surjective). Then we have: $\gamma(\alpha s_1) \subseteq \gamma a_2 \Rightarrow s_1 \in \gamma a_2 \Leftrightarrow s_1\ R\ a_2 \Leftrightarrow \alpha s_1 \sqsubseteq a_2 \Leftrightarrow a_1 \sqsubseteq a_2$. The reverse holds because $\sqsubseteq$ is defined in a way such that $\gamma$ is monotone. □

This gives an indirect way to check whether we have defined the largest order on abstract states that makes $\gamma$ monotone.

## 4.2 First-order Constants

Not having a lattice in a base abstract domain causes problems. In terms of finding best representatives for higher-type constants, not only is it not possible to use the results from [2], but in general a best abstract value does not exist. Such a best value will, however, exist in some cases; we now discuss some sufficient conditions for the existence of a best representative.

### 4.2.1 A Pre-order on the Concrete States

The base concrete domain is frequently a flat domain. It will be useful for presentation purposes to define a pre-order on concrete states according to the order of their correspondents in the abstract domain:

$$s_1 \sqsubseteq\!\!\!\!\sim s_2 \Leftrightarrow \alpha s_1 \sqsubseteq \alpha s_2.$$

This means smaller concrete elements have corresponding abstract elements which are more precise (represent smaller sets) then larger elements. It is a pre-order and not a partial order because two different concrete states may correspond to the same abstract state.

### 4.2.2 A First Condition: Monotonicity

Given a constant $f : S \to S$ in the concrete domain, for which we want to obtain a corresponding $g$, suppose $f$ is monotone with respect to the above pre-order:

$$s_1 \sqsubseteq\!\!\!\!\sim s_2 \Rightarrow f s_1 \sqsubseteq\!\!\!\!\sim f s_2.$$

**Proposition 4.4** *For any $a$ there exists an $a^*$ such that, for all $s$:*

$$\alpha s = a \Rightarrow \alpha(f s) = a^*.$$

**Proof** Given any $s_1, s_2$ such that $\alpha s_1 = \alpha s_2 = a$, we have $s_1 \sqsubseteq\!\!\!\!\sim s_2$, and so $f s_1 \sqsubseteq\!\!\!\!\sim f s_2$, which is $\alpha(f s_1) \sqsubseteq \alpha(f s_2)$. But also $s_2 \sqsubseteq\!\!\!\!\sim s_1$ and so $\alpha(f s_2) \sqsubseteq \alpha(f s_1)$; this means $\alpha(f s_1) = \alpha(f s_2) = a^*$. $\square$

This allows us to define a function in the abstract domain: let $g^*$ be defined by $g^* a = a^*$, for any given $a$.

**Proposition 4.5** *$f \; R \; g^*$.*

**Proof** We have $f \; R \; g^*$ if $s \; R \; a \Rightarrow f s \; R \; g^* a$. Suppose $s \; R \; a$, that is $\alpha s \sqsubseteq a$, and let $s_1$ be such that $\alpha s_1 = a$. Then we have: $\alpha s \sqsubseteq \alpha s_1 \Leftrightarrow s \sqsubseteq\!\!\!\!\sim s_1 \Rightarrow f s \sqsubseteq\!\!\!\!\sim f s_1 \Leftrightarrow \alpha(f s) \sqsubseteq \alpha(f s_1) \Leftrightarrow \alpha(f s) \sqsubseteq g^* a \Leftrightarrow f s \; R \; g^* a$. $\square$

**Proposition 4.6** *For any $g$ such that $f \; R \; g$, we have $g^* \sqsubseteq g$.*

**Proof** Suppose $f\ R\ g$; then, whenever $s\ R\ a$ we have $fs\ R\ ga$, that is $\alpha(fs) \sqsubseteq ga$. For an $s_1$ such that $\alpha s_1 = a$, we have $s_1\ R\ a$, and so $\alpha(fs_1) \sqsubseteq ga$. But as $g^*a = \alpha(fs_1)$ we have $g^*a \sqsubseteq ga$. As this holds for any given $a$, we have $g^* \sqsubseteq g$. $\Box$

Although the described condition is sufficient for the existence of a best representative in the abstract domain, it is a strong condition which may not hold for realistic functions; this has happened in our experience.

This condition means that any two concrete states $s_1$ and $s_2$, such that $\alpha s_1 = \alpha s_2$, will be transformed to states which also have the same abstraction: $\alpha(fs_1) = \alpha(fs_2)$. Essentially it means that any pair of states which are undistinguishable from the abstract point of view are transformed into a pair of again undistinguishable states.

### 4.2.3 A Less Demanding Condition

Given a constant $f : S \to S$ in the concrete domain, suppose that for any $a : S^a$ there exists an $s^a$, with $s^a\ R\ a$, such that for any other $s$:

$$s\ R\ a \Rightarrow fs \precsim fs^a.$$

Then, we have that $s\ R\ a$ implies $fs\ R\ \alpha(fs^a)$. Defining a function $g^*$ by $g^*a = \alpha(fs^a)$, we have that $f\ R\ g^*$.

On the other hand, for any $g$ such that $f\ R\ g$, as $s^a\ R\ a$, we must have $fs^a\ R\ ga$, that is $\alpha(fs^a) \sqsubseteq ga$, or $g^*a \sqsubseteq ga$. As this holds for any $a$, then $g^*$ is the best function related to $f$.

This condition for the existence of a best representative is weaker than the one previously described. It just requires that for each abstract state there is a representative concrete state that is mapped by the function to a state that is larger than the mapping of any other corresponding concrete state.

Essentially the condition says that for any given abstract state $a$, the set of states:

$$\{\alpha(fs) \mid s\ R\ a\}$$

has a greatest element which we can take to be $g^*a$, obtaining a best function related to $f$:

$$g^*a = \alpha(fs^a) = \bigsqcup\{\alpha(fs) \mid s\ R\ a\},$$

where we are taking the join of a directed set.

## 4.3 Control Flow

It is usual for imperative languages to have control flow commands for conditionals and loops. In defining the abstract semantics corresponding to these commands, it is normally unrealistic to try to consider the condition which determines the path that is taken. It is common practice to simply ignore the condition and assume that any branch can be taken.

Given that an abstract state represents a set of possible concrete states, the natural way to express the abstract semantics of a conditional is to use a join of the result of each branch. This motivates the use of a lattice, as happens traditionally, to ensure the existence of the join for every pair of abstract states.

In our approach we have base domains which are not necessarily lattices. We use the base domain for representing information about the state as manipulated by *atomic commands*, and use a completion of the base domain for *composite commands* such as conditionals and loops.

A point to note is that even if the base domain happens to be a lattice, some information is lost by performing a join—as the result, in general, represents a set of states larger than the union of the sets represented by the operands to the join. This information loss adds to the (almost inevitable) loss due to ignoring the boolean condition that defines the branch taken.

In our approach we can avoid suffering this extra loss by using for the completion the set of down-sets (order ideals) of the base domain, as long as we can afford the size of the resulting domain. This way we maintain the information about the outcome of the different branches, incurring only the loss due to ignoring the condition. Using down-sets amounts in practice to using sets of incomparable elements (antichains), as they are isomorphic.

On the other hand, if the size of the abstract domains is a problem, smaller completions can be used. The extreme case is using the smallest completion of the base domain—the Dedekind-MacNeille completion (also known as *completion by cuts* or the *normal completion*), given by (see eg. [28]):

$$\mathrm{DM}(D) = \{A \subseteq D \mid A^{ul} = A\}.$$

Either way, the choice of the completion to use is a separate issue from the definition of the base domain. We consider, however, the completion by down-sets the first choice to consider, as it avoids extra information loss. For this reason and for the sake of concreteness, we will just consider using the completion by down-sets in what follows.

We use $S^d$ for the lattice of down-sets of the base abstract domain:

$$S^d = \mathcal{O}(S^a).$$

We then define $R$ between concrete states $s \in S$ and elements $d \in S^d$ as

$$s \; R \; d \Leftrightarrow \exists a \in d.\, s \; R \; a.$$

This means that a down-set $d$ represent the union of the sets of concrete states corresponding to each element in $d$. As we have a safety relation $s \; R \; a \Leftrightarrow \alpha s \sqsubseteq a$, the above expression is equivalent to

$$s \; R \; d \Leftrightarrow \exists a \in \operatorname{Max} d.\, s \; R \; a.$$

This confirms the intuition that an antichain consisting of the maximal elements represents the same concrete states as a whole down-set. Although we may present an abstract interpretation using down-sets, a practical implementation of the static analysis will only need to consider these frontier elements and not whole down-sets.

The join of elements in $S^d$, which amounts to set union, becomes appropriate to describe the abstract semantics of conditional-like commands on the lines of:

$$\mathcal{C}^a[\![\texttt{if}\, e \,\texttt{then}\, c_0 \,\texttt{else}\, c_1]\!] = \lambda d.\, \mathcal{C}^a[\![c_0]\!]d \sqcup \mathcal{C}^a[\![c_1]\!]d,$$

the semantic function for composite commands being

$$\mathcal{C}^a : \mathit{Com} \to S^d \to S^d.$$

These semantic functions do not manipulate base states directly, but perform operations on downsets, like joins (the conditional) and fixed point calculations (the loop). One of the possible composite commands can be an atomic command; for this case $\mathcal{C}^a$ makes use of the semantic function for atomic commands $\mathcal{A}^a$, applying it to each element of the down-set and joining the principal ideals corresponding to the resulting states:

$$\mathcal{C}^a[\![a]\!]d = \bigsqcup_{e \in d} {\downarrow}\mathcal{A}^a[\![a]\!]e.$$

In fact, it is only necessary to consider each representative element of the down-set (each maximal element):

$$\mathcal{C}^a[\![a]\!]d = \bigsqcup_{m \in \operatorname{Max} d} {\downarrow}\mathcal{A}^a[\![a]\!]m.$$

This property is inherited by arbitrary composite commands and functions (defined in terms of commands), and we have that elements $f$ of a function space $S^d \to S^d$ which occur in the abstract interpretation are not arbitrary (merely continuous), but restricted in the sense that:

$$f d = \bigsqcup_{m \in \mathrm{Max}\, d} f{\downarrow}m.$$

Which means that to represent an element of a function space, we need only keep the output for each principal ideal, as opposed to the whole graph of the function.

It also means that function application distributes with respect to the join operation:

$$f(d_1 \sqcup d_2) = f d_1 \sqcup f d_2.$$

## 4.4 Avoiding Function Spaces

Representing functions and finding the fixed points due to recursive definitions in a efficient way is one of the main problems in abstract interpretation. This has been the target of research, an example being the frontier representation of a function [23, 61, 44].

Here we do not describe a general method but simply the approach we use in our specific problem, which has led to a very efficient representation of functions. Unfortunately, it is not generally applicable, as it relies on a 'special' property which may not hold. We do, however, draw attention to it as something to be checked for in each particular problem; if it turns out to be applicable, it will solve the intractability problems associated with the representation of functions. To simplify the presentation we will not consider explicitly the parameters of a function.

The possible effects of commands to a base abstract state $a$ may be very diverse. In our particular problem, however, the possible effects of a function invocation on a state in the calling context are only to 'move' it to a larger state: for every possible function $f$ and calling state $a$ we have

$$f a \sqsupseteq a;$$

that is, the functions are *inflations.*

Moreover, a state can only be inflated through a restricted number of operations. Also, we have a smallest (most precise) abstract state ($\perp$). It turns out that, looking at the effect of a function when applied to the most precise abstract state allows us to determine what possible operations the function may perform with effects on the calling state. This in turn allows us to extrapolate what will be the effect of applying the function to any other state. In other words, for any possible function $f$ and calling state $a$,

$$f a = g(a, f \bot),$$

where $g$ is a single computation which works for every function $f$ which can be defined in the language in question.

This way, having determined computation $g$, a function $f : S^d \to S^d$ can be represented by a single element of $S^d$ (the response of the function to $\bot$) instead of its graph. Considering that a program may define a set of $k$ functions, with possibly mutually recursive definitions, this allows us to compute a global fixed point

$$\mathrm{fix}\, \lambda\varphi.\ \ldots \varphi \ldots,$$

using, as the representation of the functions, a tuple

$$\varphi = (f_1 : S^d, \ldots, f_k : S^d),$$

instead of

$$\varphi = (f_1 : S^d \to S^d, \ldots, f_k : S^d \to S^d).$$

This will have a profound effect on the efficiency of the static analysis, and can be the difference between an intractable and a practical one.

As a remark, this kind of regularity is nothing very strange or unique and can be seen to exist in other areas. As an example, an analogy can be drawn to linear systems in control theory, where we have that, given a system $F$ and signals $u(t)$ and $v(t)$:

$$F(u + v) = F(u) + F(v),$$

similar to the distributivity with respect to joins. Moreover, those systems can be characterised through their response to a single signal containing all frequencies, like the step function or the unit impulse, from which the response to any other signal can be inferred by a unique computation (convolution):

$$F(u) = u \otimes F(\delta).$$

In our case the functions are inflations and are distributive; these are not, however, sufficient conditions for the 'special property' to hold, and they may not be necessary conditions. It would be interesting to have some general characterisation which could tell whether the property holds and if that is the case, which provided an automated way to synthesize $g$, without considering specific details and intuition about the problem. This is, however, beyond our aim, and can eventually be considered as a topic of future research.

## 4.5 Summary

We have presented the approach to abstract interpretation that we will use in the checking mechanism for balloon types. The central point concerned designing base abstract domains that are not lattices and separating the issues of how to summarise the relevant information about concrete states from how to merge the representations from different control paths.

We have discussed practical issues in deriving an order on the abstract domain and a logical relation, presenting a series of steps which start from the definition of the abstraction and concretisation functions. In this, we have discussed the existence of a best representative for a given concrete state, and whether there is a match between the logical relation and what the concretisation function expresses.

We have discussed the issue of obtaining abstract domains for composite commands, dealing with control flow, through completions of the base domain. We have focused, in particular, on the completion by down-sets.

Lastly, we have addressed the representation of functions in the abstract domain. Although the approach we use in our specific problem will not work in general, it leads to a very efficient representation; therefore, it is always worth to consider whether it is applicable to a particular problem.

# Chapter 5

# Type-checking Balloon Types

In this chapter we present the essence of the checking mechanism for balloon types: a static analysis which is presented as an abstract interpretation. Towards this, we start by defining a simple language (RISO), and presenting 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. Modularity, as well as other relevant issues for incorporating balloon types in 'real' languages, will be addressed in chapter 8.

## 5.1 RISO

We now define RISO, 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 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

$$
\begin{array}{lll}
o : \mathit{Op} & ::= & + \mid - \mid * \mid / \mid = \mid < \mid > \\
e : \mathit{Exp} & ::= & n \mid x\ o\ y \mid \texttt{isnull}\ x \\
a : \mathit{Asgn} & ::= & x :- y \mid x :- y.z \mid x :- \texttt{null} \mid x :- \texttt{new} \mid \\
 & & x.y :- z \mid x.y :- \texttt{null} \mid x.y :- \texttt{new} \mid \\
 & & x <- e \\
c : \mathit{Com} & ::= & a \mid c_0 ; c_1 \mid \\
 & & \texttt{if}\ e\ \texttt{then}\ c_0\ \texttt{else}\ c_1 \mid \\
 & & \texttt{while}\ e\ \texttt{do}\ c \mid \\
 & & x :- f_i(x_1, \ldots, x_{a_i}) \\
d : \mathit{Decl} & ::= & f_1(x_{11}, \ldots, x_{1a_1})\ \texttt{do}\ c_1\ \texttt{return}\ y_1 \\
 & & \qquad\qquad\vdots \\
 & & f_k(x_{k1}, \ldots, x_{ka_k})\ \texttt{do}\ c_k\ \texttt{return}\ y_k \\
p : \mathit{Prg} & ::= & d\ \texttt{do}\ c
\end{array}
$$

Figure 5.1: Abstract syntax of RISO

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 5.1. 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.

We do not allow general expressions because side-effects are not only possible but are a central part of what is being modelled. Allowing, for example, function invocation in an expression would force us to consider issues such as evaluation order. Such would complicate the semantics and be a distraction from what is essential without changing the computational power of the language.

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.

Commands are assignment, sequence, conditional, loop and function invocation. Function variables are represented by $f_i \in \mathit{Fvar}$; being assumed that function $f_i$ has arity $a_i$. In an invocation both parameter passing and the return of the result have the semantics of a reference assignment $(:-)$; this is the interesting case. A program consists of declarations of functions (which may be recursive or mutually recursive), followed by the 'main' command.

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 functions invocation, and for identifiers $x$ used in expressions ($\mathit{Exp}$) $\mathrm{typeof}\, x = \mathrm{Int}$.

We also assume that in RISO programs we have that, for each function $f_i$:

- formal parameters $x_{i1}, \ldots, x_{ia_i}$ are part of the variables in its body $c_i$;
- no assignments $(:-)$ are made to any formal parameter $x_{ij}$ in the body $c_i$.

This last assumption does not impair expressive power, as it is always possible to copy the references in the parameters to local variables and use them instead. (The intended semantics of RISO parameter passing is like in user-defined types in object-oriented languages like Java; even if a reference assignment to a parameter were allowed in RISO, it would have no effect on the variable passed as actual parameter.) The assumption helps simplify the description, because this way a formal parameter always refers to the object referenced by the actual parameter, and there is no need to introduce extra variables in describing the semantics. In particular, it makes possible to use a 'inlining' semantics for function invocation, which turns out to be a quite elegant way of describing the effects of a function (with side-effects) on the calling context.

We also make the assumption that in invocations:

$$x :- f_i(x_1, \ldots, x_{a_i})$$

all actual parameters $x_1, \ldots, x_{a_i}$ are different identifiers. This will allow avoiding notational clutter and does not impair expressive power, as temporary variables can always be used; i.e. instead of some invocation:

```
y :- f(x,x);
```

we can use:

```
t1 :- x;
t2 :- x;
y  :- f(t1,t2);
```

to obtain the same effect.

## 5.2 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. An important aspect is that '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

| | |
|---|---|
| $N$ | discrete cpo of numbers |
| $I$ | finite discrete cpo of identifiers |
| $A$ | finite flat cpo of addresses |
| | ($\perp_A$ denotes the null address) |
| $V = [I \to A]$ | pointed cpo of variable mappings |
| | $\perp_V$ denotes the null mapping |
| $O = (N + V)_\perp$ | pointed cpo of object values |
| | ($\perp_O$ denotes the undefined object) |
| $G = [A \to O]$ | pointed cpo of graphs of objects |
| | $\perp_G$ denotes the null graph |
| $S = \overline{G \times V}$ | discrete cpo, 'the state' |
| $F = F_1 \times \cdots \times F_k$ | pointed cpo of function environments, where |
| | $F_i = [I^{a_i+1} \to S \to S_\perp]$ |

Figure 5.2: Semantic domains

implying that it corresponds to physical addresses in some implementation. Others may prefer the term *object identifier.*)

The semantic domains are given in Figure 5.2. 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 5.3 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.

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 both $\mathcal{A}$ and $\mathcal{C}$ instead of having a single $\mathcal{C}$ 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. A function environment is necessary in the case of general commands as they may involve function invocations.

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

Figure 5.3: Semantic functions

Semantic function $\mathcal{D}$ maps 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. A program is a closed unit; the meaning of a program is not expressed in terms of some environment; this only happens for sub-parts of a program. The definition of the semantic functions is given in Figure 5.4.

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.

The semantics for declaration of functions involves a global fixed point calculation due to possible mutually recursive definitions. It is defined to have the same result as 'inlining' the body of the function on the calling context, after renaming formal to actual parameters and 'hiding' local variables under new names.

Making the inlined body act on the calling context state while avoiding conflicts with local variables requires performing a domain extension ($\overset{+}{|}$). In the extended state, identifiers corresponding to local variables are initialised to the `null` reference, while the object graph remains the same:

$$(g, v) \overset{+}{|} X = (g, v[x \mapsto \perp_A \mid x \in X]).$$

A domain restriction is performed 'after' applying the renamed body, to return back to the original set of identifiers in the calling context.

We use the symbol 'var' overloaded to both a function which gives the identifiers for variables in a piece of syntax (with an obvious definition which we omit), and to a function which gives the variables in a state $s : S$ $(\mathrm{var}(g, v) = \mathrm{dom}\, v)$.

$$
\begin{array}{lcl}
\mathcal{E}\llbracket n \rrbracket & = & \lambda(g,v).\, \lfloor n \rfloor \\
\mathcal{E}\llbracket x\ o\ y \rrbracket & = & \lambda(g,v).\, \mathcal{O}\llbracket o \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).\, \mathrm{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 :- \texttt{new} \rrbracket & = & \lambda(g,v).\, \mathrm{let}\, a \Leftarrow \mathrm{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 } \mathrm{typeof}\, x = \mathrm{Int}, \\ \lfloor \bot_V \rfloor & \text{otherwise.} \end{cases} \\
\mathcal{A}\llbracket x.\, y :- z \rrbracket & = & \lambda(g,v).\, \mathrm{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).\, \mathrm{let}\, a \Leftarrow vx.\, \lfloor g[\lfloor a \rfloor \mapsto g\lfloor a \rfloor [y \mapsto \bot_A]], v \rfloor \\
\mathcal{A}\llbracket x.\, y :- \texttt{new} \rrbracket & = & \lambda(g,v).\, \mathrm{let}\, a_x \Leftarrow vx.\, \mathrm{let}\, a_n \Leftarrow \mathrm{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 \\
 & & \quad \text{where } o = \begin{cases} \lfloor 0 \rfloor & \text{if } \mathrm{typeof}\, y = \mathrm{Int}, \\ \lfloor \bot_V \rfloor & \text{otherwise.} \end{cases} \\
\mathcal{A}\llbracket x <- e \rrbracket & = & \lambda(g,v).\, \mathrm{let}\, a \Leftarrow vx.\, \mathrm{let}\, i \Leftarrow \mathcal{E}\llbracket e \rrbracket (g,v).\, \lfloor g[\lfloor a \rfloor \mapsto \lfloor i \rfloor], v \rfloor \\
\mathcal{C}\llbracket a \rrbracket & = & \lambda\varphi.\, \lambda s.\, \mathcal{A}\llbracket a \rrbracket s \\
\mathcal{C}\llbracket c_0 ; c_1 \rrbracket & = & \lambda\varphi.\, \lambda s.\, \mathrm{let}\, s' \Leftarrow \mathcal{C}\llbracket c_0 \rrbracket \varphi s.\, \mathcal{C}\llbracket c_1 \rrbracket \varphi s' \\
\mathcal{C}\llbracket \texttt{if}\, e\, \texttt{then}\, c_0\, \texttt{else}\, c_1 \rrbracket & = & \lambda\varphi.\, \lambda s.\, \mathrm{let}\, i \Leftarrow \mathcal{E}\llbracket e \rrbracket s. \begin{cases} \mathcal{C}\llbracket c_0 \rrbracket \varphi s & \text{if } i = 0, \\ \mathcal{C}\llbracket c_1 \rrbracket \varphi s & \text{otherwise.} \end{cases} \\
\mathcal{C}\llbracket \texttt{while}\, e\, \texttt{do}\, c \rrbracket & = & \lambda\varphi.\, \mathrm{fix}\, \lambda h.\, \lambda s.\, \mathrm{let}\, i \Leftarrow \mathcal{E}\llbracket e \rrbracket s. \\
 & & \qquad \begin{cases} \mathrm{let}\, s' \Leftarrow \mathcal{C}\llbracket c \rrbracket \varphi s.\, hs' & \text{if } i = 0, \\ \lfloor s \rfloor & \text{otherwise.} \end{cases} \\
\mathcal{C}\llbracket x :- f_i(x_1, \ldots, x_{a_i}) \rrbracket & = & \lambda\varphi.\, \lambda s.\, \varphi_i(x, x_1, \ldots, x_{a_i}) s \\
\mathcal{D}\llbracket d \rrbracket & = & \mathrm{fix}\, \lambda\varphi.\, (\overrightarrow{\varphi'_k}),\ \text{where}
\end{array}
$$

$$
\begin{array}{l}
\varphi'_i = \lambda(x, \overrightarrow{x_{a_i}}).\, \lambda s. \\
\qquad \mathcal{C}\llbracket c_i[\mathrm{ren}\, z/z \mid z \in \mathrm{var}\, c_i][\overrightarrow{x_{a_i}}/\overrightarrow{\mathrm{ren}\, x_{ia_i}}]; x :- \mathrm{ren}\, y_i \rrbracket \varphi(s \overset{+}{|} X) \mid \mathrm{var}\, s \\
\text{with } X \text{ a set of new identifiers, } |X| = |\mathrm{var}\, c_i|, \\
\text{and } \mathrm{ren} : \mathrm{var}\, c_i \to X, \text{ bijective;} \\
d \text{ as given by the abstract syntax in Figure 5.1.}
\end{array}
$$

$$
\mathcal{P}\llbracket d\, \texttt{do}\, c \rrbracket \quad = \quad \mathcal{C}\llbracket c \rrbracket \mathcal{D}\llbracket d \rrbracket (\bot_G, \bot_V)
$$

Figure 5.4: Semantic function definitions

The semantic functions make use of a function 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:

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

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

## 5.3 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: 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.

### 5.3.1 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.

We now state properties of these functions that will be useful later. These properties follow directly from the above definitions.

**Lemma 5.1** *For all $s \in S_b$, $x, y \in I$, and $X, Y \subseteq I$:*

$$\begin{aligned}&B(s, Ps\{x\}) = B(s, \{x\}),\\ &X \subseteq Y \Rightarrow B(s, X) \leq B(s, Y), \text{ and}\\ &(\forall x \in X, y \in Y.\, (x, y) \notin Ps) \Rightarrow B(s, X \cup Y) = B(s, X) + B(s, Y).\end{aligned}$$

From these properties it immediately follows that:

**Lemma 5.2** *For all $s \in S_b$, $x, y \in I$, and $X \subseteq I$:*

$$\begin{aligned}&B(s, X) \leq \textstyle\sum_{x \in X} B(s, \{x\}), \text{ and}\\ &(\forall x, y \in X.\, B(s, \{x, y\}) \leq 1) \Rightarrow B(s, X) \leq 1.\end{aligned}$$

These lemmas formalise the intuitive expectations about $P$ and $B$.

## 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, $\alpha$ 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 $\gamma\colon C \to \mathcal{P}(S_b)$:

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

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 $\alpha$ and $\gamma$ 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 $s \in S_b$:

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

**Proof** From the definitions it follows that, given an arbitrary $s_1 \in S_b$, we have $\alpha s_1 = (Ps_1, \lambda x.\, B(s_1, \{x\}))$, and $s \in \gamma(\alpha s_1)$ if and only if

$$\begin{aligned} \forall x, y \in I.((B(s_1,\{x\}) = 0 \vee B(s_1,\{y\}) = 0) \wedge (x,y) \notin Ps_1 \Rightarrow (x,y) \notin Ps) \\ \wedge\ B(s, Ps_1\{x\}) \leq B(s_1,\{x\}). \end{aligned}$$

For $s = s_1$ the first component of the condition holds as it takes the form $X \wedge Y \Rightarrow Y$, while the second component holds because from Lemma 5.1 we have that $B(s_1, Ps_1\{x\}) = B(s_1,\{x\})$. Therefore we have $s_1 \in \gamma(\alpha s_1)$. $\square$

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 $\boxed{\mathrm{xy}}\ \overline{\underline{\mathrm{z}}}$ 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. It is indeed relevant to remark that throughout the presentation of the abstract interpretation we make no direct manipulation of the components $g$ and $v$ of a concrete state $s = (g, v)$. We use a state $s$ as an atomic entity; relating concrete and abstract states as well as reasoning in general is made through the predicates $P$ and $B$ we have defined and their corresponding properties as stated in Lemmas 5.1 and 5.2 (which simply describe trivial facts about graphs, connectedness and clusters as defined). This means that the concrete semantics could have been written using another representation for states; we have chosen one which makes the semantics naturally 'concrete' and realistic.

Figure 5.5 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 $\gamma(\overline{\underline{\mathrm{x}}}\ \boxed{\mathrm{y}})$, and all concrete states belong to $\gamma(\boxed{\mathrm{x}}\ \boxed{\mathrm{y}})$. 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 $\alpha d = \boxed{\mathrm{x}}\,\underline{\overline{\mathrm{y}}}$.

### Order in the Abstract States

The function $\gamma$ induces a partial order on $C$ such that $\gamma$ 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**

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

(Here $b_1 \sqsubseteq b_2$ means the usual pointwise ordering: $\forall x \in I.\, b_1 x \le b_2 x$.) Figure 5.6 shows $C$, which is a pointed cpo, in the case when $I = \{x, y, z\}$. With the above order we have for example: $\boxed{\mathrm{x}}\,\underline{\overline{\mathrm{y}}}\,\underline{\overline{\mathrm{z}}} \sqsubseteq \boxed{\mathrm{xy}}\,\underline{\overline{\mathrm{z}}}$ and $\boxed{\mathrm{xy}}\,\underline{\overline{\mathrm{z}}} \sqsubseteq \boxed{\mathrm{x}}\,\boxed{\mathrm{y}}\,\underline{\overline{\mathrm{z}}}$.

To show that the defined order makes $\gamma$ an order-embedding we will show first that $c_1 \sqsubseteq c_2 \Rightarrow \gamma c_1 \subseteq \gamma c_2$, and then that $s \in \gamma c \Rightarrow \alpha s \sqsubseteq c$ (so that we can use Propositions 4.2 and 4.3). We will also make use of the following lemma:

**Lemma 5.4** *Let $F$ stand for $\forall x \in I.\, B(s, p\{x\}) \le bx$ and $G$ stand for $\forall x, y \in I.\, x\ p\ y \Rightarrow B(s, \{x, y\}) \le bx$. We have $F \Leftrightarrow G$.*

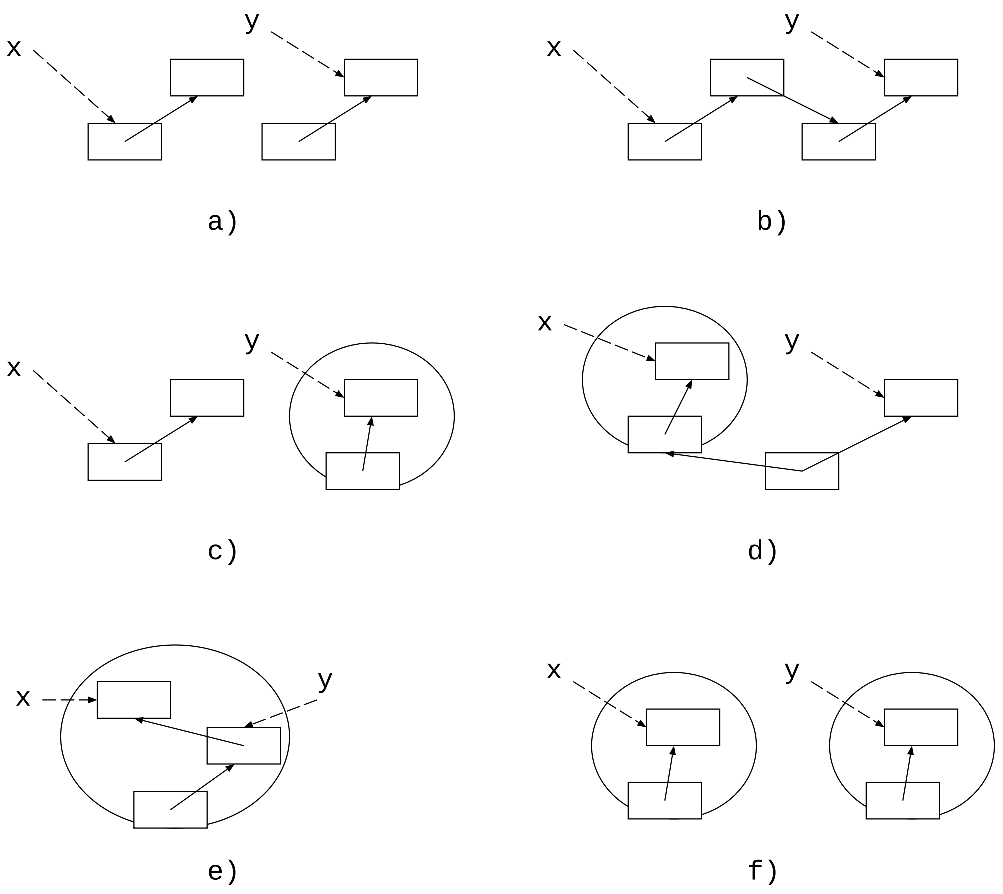
**Proof** Suppose $F$ holds; if $x \not p\ y$ the implication in $G$ holds trivially; if $x\ p\ y$, we have $\{x, y\} \subseteq p\{x\}$, and so $B(s, \{x, y\}) \le B(s, p\{x\})$, therefore we have $B(s, \{x, y\}) \le bx$.

Suppose $G$ holds; either $bx = 0$, in which case we have for all $z \in p\{x\}$ that $bz = 0$ and $B(s, \{z, z\}) \le 0$, and therefore $B(s, p\{x\}) = 0$; or we have $bx = 1$, in which case we have for all $y \in p\{x\}$ that $B(s, \{x, y\}) \le 1$, and therefore $B(s, p\{x\}) \le 1$. □

**Lemma 5.5** *$c_1 \sqsubseteq c_2 \Rightarrow \gamma c_1 \subseteq \gamma c_2$.*

**Proof** We have $\gamma$ in the form $\gamma = \lambda c.\, \{s \mid \bigwedge_i F_i(s, c)\}$. We show that $c_1 \sqsubseteq c_2$ as defined above implies that $\forall s \in \gamma c_1.\, \forall i.\, F_i(s, c_1) \Rightarrow F_i(s, c_2)$, from which follows immediately that $c_1 \sqsubseteq c_2 \Rightarrow \gamma c_1 \subseteq \gamma c_2$.

Some $F_i$ are $(bx = 0 \vee by = 0) \wedge x \not p\ y \Rightarrow (x, y) \notin Ps$; they have the form $Gc \Rightarrow Hs$. The expression $\forall s.\, (Gc_1 \Rightarrow Hs) \Rightarrow (Gc_2 \Rightarrow Hs)$ is equivalent to $Gc_2 \Rightarrow$



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

Figure 5.5: Examples of concrete and abstract states

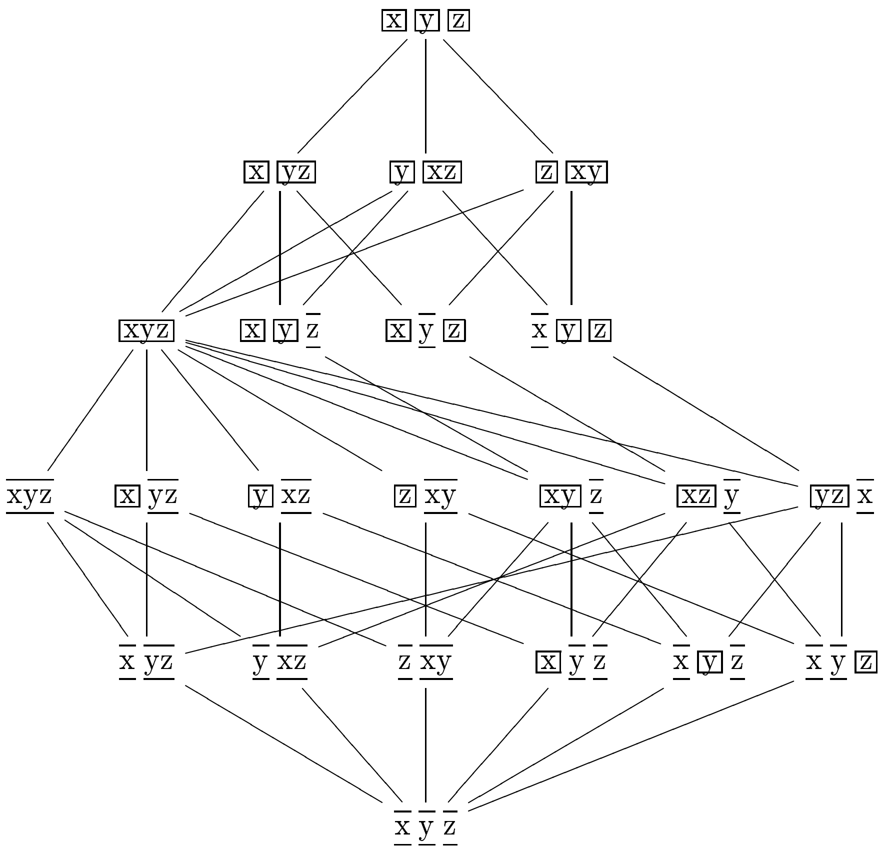


Figure 5.6: The $C_{\{x,y,z\}}$ cpo

$Gc_1$, in this case to $(b_2x = 0 \vee b_2y = 0) \wedge x \not p_2\ y \Rightarrow (b_1x = 0 \vee b_1y = 0) \wedge x \not p_1\ y$. This expression, anded with $b_1 \sqsubseteq b_2$, simplifies to $(b_2x = 0 \vee b_2y = 0) \wedge x \not p_2\ y \Rightarrow x \not p_1\ y$, which is equivalent to $x\ p_1\ y \wedge x \not p_2\ y \Rightarrow b_2x = b_2y = 1$.

The remainder of the definition of $\sqsubseteq$ is just a simplification of the predicate $x\ p_2\ y \Rightarrow (x\ p_1\ y \Rightarrow b_1x \leq b_2x) \wedge (x \not p_1\ y \Rightarrow b_1x + b_1y \leq b_2x)$. (We have always $x\ p\ x$ and so $(x\ p_2\ y \Rightarrow (x\ p_1\ y \Rightarrow b_1x \leq b_2x)$ is equivalent to $b_1x \leq b_2x$.) We now show that the above predicate implies that $\forall s \in \gamma c_1 .\ F_i(s, c_1) \Rightarrow F_i(s, c_2)$ with $F_i(s, c) = (x\ p\ y \Rightarrow B(s, \{x, y\}) \leq bx)$. (The previous lemma allows us to use this expression intead of $B(s, p\{x\}) \leq bx$ as in the definition of $\gamma$.) We do this by a case analysis. Suppose the above predicate holds. If $x \not p_2\ y$ then $F_i(s, c_2)$ and so trivially $F_i(s, c_1) \Rightarrow F_i(s, c_2)$. If $x\ p_2\ y$ there are two cases to consider:

- If $x\ p_1\ y$ then we must have $b_1x \leq b_2x$; if $F_i(s, c_1)$ it means that $B(s, \{x, y\}) \leq b_1x$, and so $B(s, \{x, y\}) \leq b_2x$; therefore $F_i(s, c_2)$;

- if $x \not p_1\ y$ (in which case $F_i(s, c_1)$ is true) then we must have $b_1x + b_1y \leq b_2x$. It is true that for all $s \in \gamma c_1$ we have $B(s, \{x\}) \leq b_1x$, $B(s, \{y\}) \leq b_1y$,

$B(s,\{x,y\}) \leq B(s,\{x\}) + B(s,\{y\}) \leq b_1 x + b_1 y$. Therefore $B(s,\{x,y\}) \leq b_2 x$ and so $F_i(s, c_2)$ is true.

□

**Lemma 5.6** $s \in \gamma c \Rightarrow \alpha s \sqsubseteq c$.

**Proof** We must show that if $s \in \gamma(p,b)$, then $(Ps, \lambda x.\, B(s,\{x\}) \sqsubseteq (p,b)$, that is:

$$\begin{aligned}\forall x, y \in I.\ & B(s,\{x\}) \leq bx \\ & \wedge\ ((x,y) \in Ps \wedge x \not p\, y \Rightarrow bx = by = 1) \\ & \wedge\ ((x,y) \notin Ps \wedge x\ p\ y \Rightarrow B(s,\{x\}) + B(s,\{y\}) \leq bx)\end{aligned}$$

Suppose that $s \in \gamma(p,b)$; then for all $x, y \in I$:

- $B(s,p\{x\}) \leq bx$; it is always true that $B(s,\{x\}) \leq B(s,p\{x\})$, therefore it follows that $B(s,\{x\}) \leq bx$.
- If $(x,y) \in Ps$ we have $\neg((bx = 0 \vee by = 0) \wedge x \not p\, y)$; if also $x \not p\, y$ then it follows that $bx = 1 \wedge by = 1$.
- If $(x,y) \notin Ps$ then $B(s,\{x,y\}) = B(s,\{x\}) + B(s,\{y\})$; if also $x\ p\ y$ then $B(s,\{x,y\}) \leq B(s,p\{x\})$; as we have $B(s,p\{x\}) \leq bx$, then $B(s,\{x\}) + B(s,\{y\}) \leq bx$.

□

**Proposition 5.7** $\alpha s \sqsubseteq c \Leftrightarrow s \in \gamma c$.

**Proof** Combine the two previous lemmas and Proposition 4.2. □

**Proposition 5.8** $c_1 \sqsubseteq c_2 \Leftrightarrow \gamma c_1 \subseteq \gamma c_2$.

**Proof** Combine the previous proposition and Proposition 4.3. □

## The size of $C$

The size of $C_I$ grows exponentially with the number of identifiers in $I$; more precisely:

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

where $n = |I|$ and $B_i$ is the $i^{\text{th}}$ Bell number, the number of partitions of a set of $i$ elements. There is no simple closed formula for $B_i$, but it can be calculated recursively by (see eg. [7]):

$$\begin{aligned} B_0 &= 1, \\ B_{n+1} &= \sum_{k=0}^{n} \binom{n}{k} B_k. \end{aligned}$$

## 5.3.2 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 $\overline{\underline{\mathtt{x}}}\,\boxed{\mathtt{y}}\,\boxed{\mathtt{z}}$ can correspond to any of the three cases in Figure 5.7. 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 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 $\overline{\underline{\mathtt{x}}}\,\boxed{\mathtt{y}}\,\boxed{\mathtt{z}}$ is now refined into four states: $\overline{\underline{\mathtt{x}}}\,\boxed{\mathtt{y}}\,\boxed{\mathtt{z}}$, $\overline{\underline{\mathtt{x}\boxed{\mathtt{y}}}}\,\boxed{\mathtt{z}}$, $\overline{\underline{\mathtt{x}\boxed{\mathtt{z}}}}\,\boxed{\mathtt{y}}$, and $\overline{\underline{\mathtt{x}\boxed{\mathtt{y}}\,\boxed{\mathtt{z}}}}$. 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 5.7 are now abstracted as $\overline{\underline{\mathtt{x}}}\,\boxed{\mathtt{y}}\,\boxed{\mathtt{z}}$, $\overline{\underline{\mathtt{x}\boxed{\mathtt{y}}}}\,\boxed{\mathtt{z}}$, and $\overline{\underline{\mathtt{x}\boxed{\mathtt{y}}\,\boxed{\mathtt{z}}}}$ respectively. (Note how the nesting in the third case is not relevant; $y$ and $z$

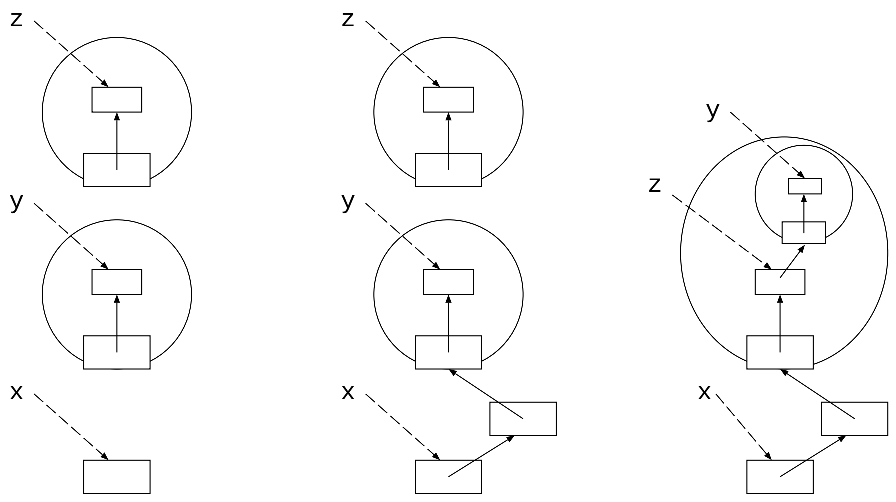


Figure 5.7: Three cases abstracted into the same state

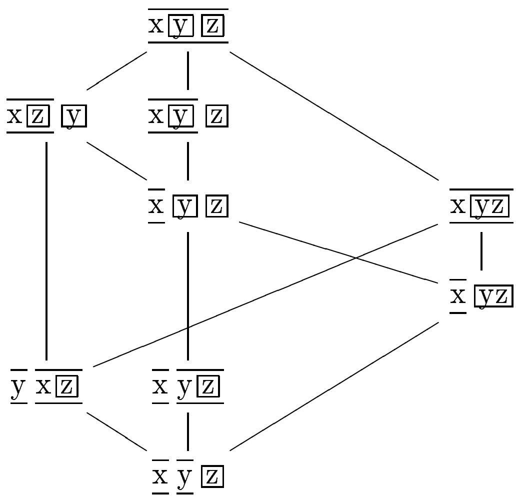

$\underline{\overline{\mathrm{x}\boxed{\mathrm{y}}\,\boxed{\mathrm{z}}}}$
$\underline{\overline{\mathrm{x}\boxed{\mathrm{z}}}}\ \boxed{\mathrm{y}}$
$\underline{\overline{\mathrm{x}\boxed{\mathrm{y}}}}\ \boxed{\mathrm{z}}$
$\underline{\overline{\mathrm{x}}}\ \boxed{\mathrm{y}}\ \boxed{\mathrm{z}}$
$\underline{\overline{\mathrm{x}\boxed{\mathrm{yz}}}}$
$\underline{\overline{\mathrm{x}}}\ \boxed{\mathrm{yz}}$
$\underline{\overline{\mathrm{y}}}\ \underline{\overline{\mathrm{x}\boxed{\mathrm{z}}}}$
$\underline{\overline{\mathrm{x}}}\ \underline{\overline{\mathrm{y}\boxed{\mathrm{z}}}}$
$\underline{\overline{\mathrm{x}}}\ \underline{\overline{\mathrm{y}}}\ \boxed{\mathrm{z}}$

Figure 5.8: Fragment of the extended $C_{\{x,y,z\}}$ cpo

could even refer to the same cluster, as before.) Figure 5.8 shows a fragment of the extended $C_{\{x,y,z\}}$ cpo, corresponding to refining the following three states: $\underline{\overline{\mathrm{x}}}\ \underline{\overline{\mathrm{y}}}\ \boxed{\mathrm{z}} \sqsubseteq \underline{\overline{\mathrm{x}}}\ \boxed{\mathrm{yz}} \sqsubseteq \underline{\overline{\mathrm{x}}}\ \boxed{\mathrm{y}}\ \boxed{\mathrm{z}}$.

### 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:

$$\begin{aligned}
&x\ r\ y \wedge y\ p\ z \Rightarrow x\ r\ z,\\
&x\ r\ y \wedge x\ p\ z \Rightarrow z\ r\ y,\\
&x\ r\ y \Rightarrow bx = 0 \wedge by = 1, \text{ and}\\
&x\ r\ z \wedge y\ r\ z \Rightarrow x\ p\ y.
\end{aligned}$$

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:

- $R : S_b \times I \times I \to \mathit{bool}$, 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:

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

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

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

**Proof** For an arbitrary $s_1 \in S_b$ we have $\alpha s_1 = (p_1, b_1, r_1)$, with $r_1 = \{(x, y) \mid R(s_1, x, y)\}$. It remains to prove that for $s = s_1$ we have $(b_1 x = 0 \wedge b_1 y = 1 \wedge x \not{r}_1\, y) \vee x\ p_1\ y \Rightarrow \neg R(s, x, y)$; this becomes $(b_1 x = 0 \wedge b_1 y = 1 \wedge \neg R(s_1, x, y)) \vee x\ p_1\ y \Rightarrow \neg R(s_1, x, y)$. Either $x \not{p}_1\, y$ and the above is true, or $x\ p_1\ y$, which means that $(x, y) \in Ps_1$, $R(s_1, x, y)$ must be false, and the above holds as well. Therefore we have $s_1 \in \gamma(\alpha s_1)$. $\square$

## Order in the Abstract States

As before, $\gamma$ induces a partial order on $C$; the new order becomes:

**Definition 5.9**

$$\begin{aligned}(p_1, b_1, r_1) \sqsubseteq (p_2, b_2, r_2) \Leftrightarrow (b_1 \sqsubseteq b_2) \wedge \forall x, y \in I. & \\ (x \; p_1 \; y \wedge x \not{p}_2 \; y \Rightarrow b_2 x = b_2 y = 1) & \\ \wedge \; (x \not{p}_1 \; y \wedge x \; p_2 \; y \Rightarrow b_1 x + b_1 y \leq b_2 x) & \\ \wedge \; (x \; r_1 \; y \wedge x \not{r}_2 \; y \Rightarrow x \not{p}_2 \; y \wedge b_2 x = 1). &\end{aligned}$$

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

**Lemma 5.10** $c_1 \sqsubseteq c_2 \Rightarrow \gamma c_1 \subseteq \gamma c_2$.

**Proof** Suppose $c_1 \sqsubseteq c_2$; it remains to show that if $s \in \gamma c_1$ then $F(s, c_2)$ holds, with $F(s, (p, b, r)) = ((bx = 0 \wedge by = 1 \wedge x \not{r} y) \vee x \; p \; y \Rightarrow \neg R(s, x, y))$. Suppose $s \in \gamma c_1$; then we have $F(s, c_1)$. Either $\neg R(s, x, y)$ in which case $F(s, c_2)$ holds trivially, or $R(s, x, y)$. In this last case we have $\neg(b_1 x = 0 \wedge b_1 y = 1 \wedge x \not{r}_1 \; y) \wedge x \not{p}_1 \; y$, and also $b_1 y = 1$ (as $y$ references an object in a captured cluster); therefore $(b_1 x = 1 \vee x \; r_1 \; y) \wedge x \not{p}_1 \; y$. If $b_1 x = 1$, as $c_1 \sqsubseteq c_2$, we must have $b_2 x = 1$ and $x \not{p}_2 \; y$; therefore $F(s, c_2)$ holds. If $x \; r_1 \; y$ we must have $x \; r_2 \; y \vee (x \not{p}_2 \; y \wedge b_2 x = 1)$. Both if $x \not{p}_2 \; y \wedge b_2 x = 1$ or if $x \; r_2 \; y$ (in which case $x \not{p}_2 \; y$) we have that $F(s, c_2)$ holds. □

**Lemma 5.11** $s \in \gamma c \Rightarrow \alpha s \sqsubseteq c$.

**Proof** It remains to show that if $s \in \gamma c$ then $F(s, c)$ holds, with $F(s, (p, b, r)) = (R(s, x, y) \wedge x \not{r} \; y \Rightarrow x \not{p} \; y \wedge bx = 1)$. Suppose that $s \in \gamma(p, b, r)$; then we have $(bx = 0 \wedge by = 1 \wedge x \not{r} y) \vee x \; p \; y \Rightarrow \neg R(s, x, y)$. Either $\neg R(s, x, y)$ in which case $F(s, c)$ holds; or $R(s, x, y)$ and we have $(bx = 1 \vee by = 0 \vee x \; r \; y) \wedge x \not{p} y$. In this case (being $R(s, x, y)$ and $x \not{p} \; y$) we need to prove $(x \not{r} \; y \Rightarrow bx = 1)$; this holds whether $bx = 1$ or $x \; r \; y$; we cannot have $by = 0$ in this case where $R(s, x, y)$ (as $y$ points to a captured cluster in $s$). □

**Proposition 5.12** $\alpha s \sqsubseteq c \Leftrightarrow s \in \gamma c$.

**Proof** Combine the two previous lemmas and Proposition 4.2. □

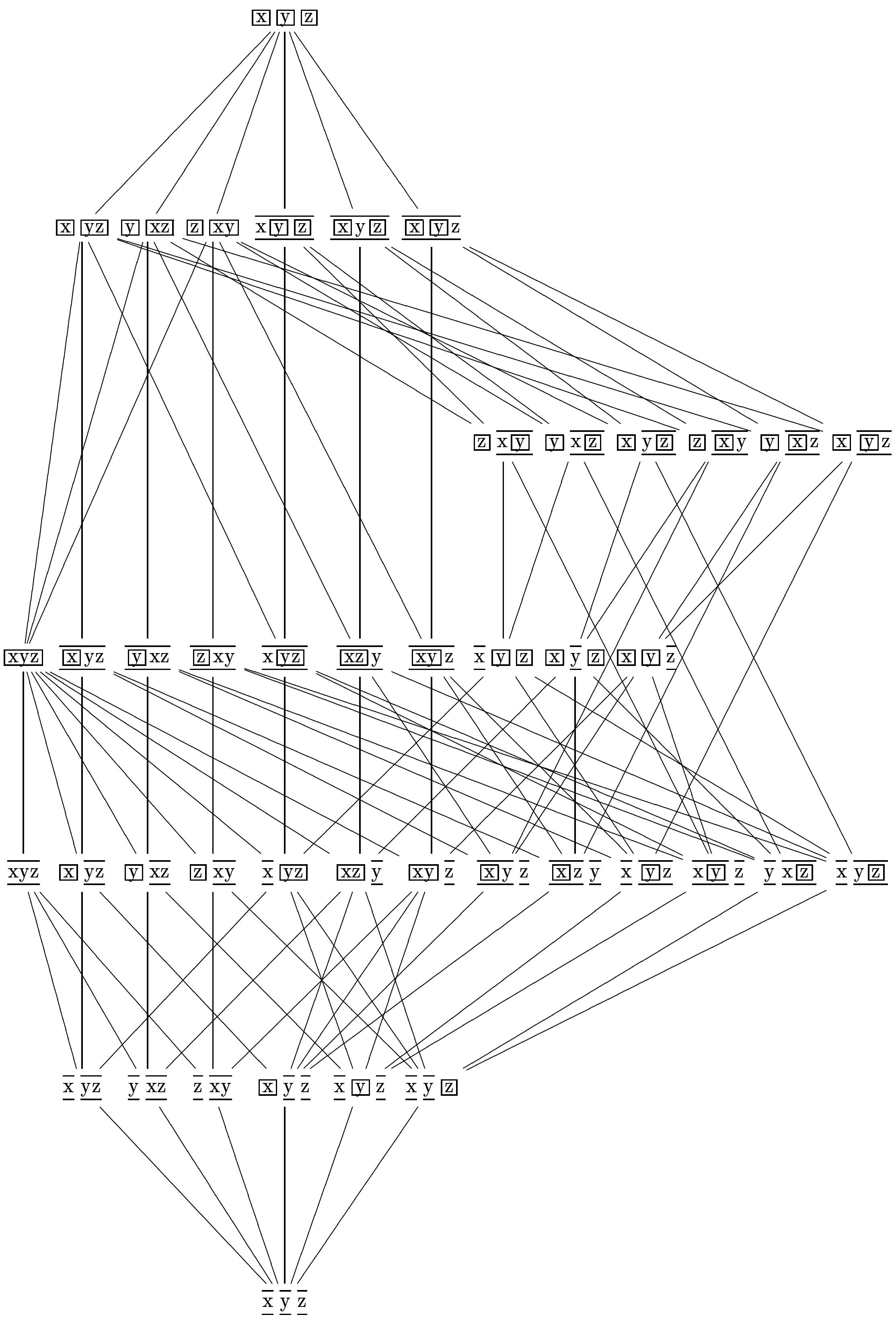


Figure 5.9: The extended $C_{\{x,y,z\}}$ cpo

**Proposition 5.13** $c_1 \sqsubseteq c_2 \Leftrightarrow \gamma c_1 \subseteq \gamma c_2$.

**Proof** Combine the previous proposition and Proposition 4.3. $\square$

We have thus that, not only $\gamma$ 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 $\alpha$ (i.e. $s \; R \; c \Leftrightarrow \alpha s \sqsubseteq c$) coincides with what is expressed by $\gamma$:

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

(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.)

### 5.3.3 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 $\alpha$, $\gamma$, 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 $\alpha$ and $\gamma$ with $\alpha\bot = \bot_C$ and making $\bot \in \gamma c$ for all $c \in C_\top$.

### 5.3.4 The Null Reference

An important remark is that no attempt is made to represent the special case of the null reference in the abstract domain. This is so because as no information is kept about the structure of objects it is unrealistic to try to determine that a variable *definitely* contains the null reference. Moreover, a null reference typically occurs as an end-marker of a linked structure; variables which traverse these structures contain a non-null reference most of the time.

With no general representation devoted to the null reference, we decided also not to explore the possibility (in the domain as it is) of allowing a variable $x$ of balloon type to be in an equivalence class mapped to zero (i.e. $\overline{\underline{\mathrm{x}}}$), representing concrete states where variable $x$ contains the null reference. It would be a source of irregularity and complexity in the mechanism (semantic functions would have to be made more complex to explore the information provided) with dubious benefits.

We have therefore decided that if a variable $x$ is of balloon type, the only states considered by the analysis are those where $x$ is in equivalence classes mapped to one (i.e. $\boxed{\mathrm{x}}$). As an example, if there are three variables, $x$ and $z$ of non-balloon type and $y$ of balloon type, the analysis only considers states in $C_{\{x,y,z\}}$ greater or equal than $\overline{\underline{\mathrm{x}}}\,\boxed{\mathrm{y}}\,\overline{\underline{\mathrm{z}}}$. The domain will remain a pointed cpo with the previous state as bottom, instead of $\overline{\underline{\mathrm{x}}}\,\overline{\underline{\mathrm{y}}}\,\overline{\underline{\mathrm{z}}}$. To emphasize that the bottom of the effectively used domain is in general different from $\overline{\underline{\mathrm{x}}}\,\overline{\underline{\mathrm{y}}}\,\overline{\underline{\mathrm{z}}}$, and due to its role (as we will describe) we will use the symbol $\mathbf{1}$ to denote it:

$$\mathbf{1} = (\{(x,x) \mid x \in I\}, \{x \mapsto \begin{cases} 1 & \text{if balloon}\, x, \\ 0 & \text{otherwise.} \end{cases} \mid x \in I\}, \emptyset).$$

## 5.4 Atomic Commands

### 5.4.1 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 5.14** $(p, b, r)[x\triangleright] = (p', b', r')$, where

$$p' \;=\; p \,\overline{\mid}\, \{x\} \cup \{(x,x)\},$$

$$\begin{aligned} b' &= \begin{cases} b[x \mapsto 1] & \text{if } \mathrm{balloon}\, x, \\ b[x \mapsto 0] & \text{otherwise.} \end{cases} \\ r' &= r \mathbin{\overline{|}} \{x\}. \end{aligned}$$

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

$$\begin{aligned} \boxed{\mathrm{xy}}\ \underline{\overline{\mathrm{z}}}\,[x\rhd] &= \boxed{\mathrm{x}}\ \boxed{\mathrm{y}}\ \underline{\overline{\mathrm{z}}}, \\ \underline{\overline{\boxed{\mathrm{xy}}\,\mathrm{z}}}\,[x\rhd] &= \boxed{\mathrm{x}}\ \underline{\overline{\boxed{\mathrm{y}}\,\mathrm{z}}}, \end{aligned}$$

and if $\neg\, \mathrm{balloon}\, x$ then

$$\begin{aligned} \underline{\overline{\mathrm{xy}}}\ \underline{\overline{\mathrm{z}}}\,[x\rhd] &= \underline{\overline{\mathrm{x}}}\ \underline{\overline{\mathrm{y}}}\ \underline{\overline{\mathrm{z}}}, \\ \boxed{\mathrm{xy}}\ \underline{\overline{\mathrm{z}}}\,[x\rhd] &= \underline{\overline{\mathrm{x}}}\ \boxed{\mathrm{y}}\ \underline{\overline{\mathrm{z}}}, \\ \underline{\overline{\boxed{\mathrm{xy}}\,\mathrm{z}}}\,[x\rhd] &= \underline{\overline{\mathrm{x}}}\ \underline{\overline{\boxed{\mathrm{y}}\,\mathrm{z}}}. \end{aligned}$$

**Definition 5.15** $(p, b, r)[x \rhd y] = (p', b', r')$, where

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

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

$$\begin{aligned} \underline{\overline{\mathrm{wx}}}\ \underline{\overline{\mathrm{yz}}}\,[x \rhd y] &= \underline{\overline{\mathrm{w}}}\ \underline{\overline{\mathrm{xyz}}}, \\ \boxed{\mathrm{wx}}\ \underline{\overline{\mathrm{yz}}}\,[x \rhd y] &= \boxed{\mathrm{w}}\ \underline{\overline{\mathrm{xyz}}}, \\ \underline{\overline{\mathrm{wx}}}\ \boxed{\mathrm{yz}}\,[x \rhd y] &= \underline{\overline{\mathrm{w}}}\ \boxed{\mathrm{xyz}}, \\ \underline{\overline{\mathrm{wx}\boxed{\mathrm{yz}}}}\,[x \rhd y] &= \underline{\overline{\mathrm{w}\boxed{\mathrm{xyz}}}}, \\ \underline{\overline{\mathrm{w}\boxed{\mathrm{x}}}}\ \underline{\overline{\mathrm{y}\boxed{\mathrm{z}}}}\,[x \rhd y] &= \underline{\overline{\mathrm{w}}}\ \underline{\overline{\mathrm{xy}\boxed{\mathrm{z}}}}. \end{aligned}$$

**Definition 5.16** $(p, b, r)[x \rhd\!\lhd y] = \begin{cases} \top & \text{if } x \not p\ y \wedge bx = by = 1, \\ \top & \text{if } x\ r\ y \vee y\ r\ x, \\ (p', b', r') & \text{otherwise, where} \end{cases}$

$$p' = (p \cup \{(x, y), (y, x)\})^+,$$

$$b' = b[z \mapsto bx \sqcup by \mid z\ p\ x \vee z\ p\ y],$$

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

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

$$\underline{\overline{\mathrm{x}\boxed{\mathrm{y}}}}\,[x \rhd\lhd y] = \top.$$

- 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:

$$\begin{aligned} \underline{\overline{\mathrm{x}}}\ \underline{\overline{\mathrm{yz}}}\,[x \rhd\lhd y] &= \underline{\overline{\mathrm{xyz}}}, \\ \underline{\overline{\mathrm{x}\boxed{\mathrm{w}}}}\ \underline{\overline{\mathrm{y}\boxed{\mathrm{z}}}}\,[x \rhd\lhd y] &= \underline{\overline{\mathrm{xy}\boxed{\mathrm{w}}\ \boxed{\mathrm{z}}}}, \\ \boxed{\mathrm{x}}\ \underline{\overline{\mathrm{y}\boxed{\mathrm{z}}}}\,[x \rhd\lhd y] &= \boxed{\mathrm{xy}}\ \boxed{\mathrm{z}}, \\ \underline{\overline{\mathrm{w}\boxed{\mathrm{x}}}}\ \underline{\overline{\mathrm{y}\boxed{\mathrm{z}}}}\,[x \rhd\lhd y] &= \underline{\overline{\mathrm{w}\boxed{\mathrm{xy}}\ \boxed{\mathrm{z}}}}, \\ \boxed{\mathrm{xy}}\ \underline{\overline{\mathrm{z}}}\,[x \rhd\lhd y] &= \boxed{\mathrm{xy}}\ \underline{\overline{\mathrm{z}}}. \end{aligned}$$

**Definition 5.17** $(p,b,r)[x \xrightarrow{+} y] = (p,b,r')$, where

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

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

Figure 5.10: 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:

$$\begin{array}{rcl}
\overline{\underline{\mathrm{x}}}\;\boxed{\mathrm{y}}\;\boxed{\mathrm{z}}[x \overset{+}{\rightarrow} y] & = & \overline{\underline{\mathrm{x}\boxed{\mathrm{y}}}}\;\boxed{\mathrm{z}}, \\
\overline{\underline{\mathrm{x}\boxed{\mathrm{z}}}}\;\boxed{\mathrm{y}}[x \overset{+}{\rightarrow} y] & = & \overline{\underline{\mathrm{x}\boxed{\mathrm{y}}\;\boxed{\mathrm{z}}}}.
\end{array}$$

## 5.4.2 Abstract Semantics for Assignments

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

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

with the definition shown in Figure 5.10.

**Proposition 5.18** *For all $a \in \mathit{Asgn}$, $\mathcal{A}[\![a]\!] \; R \; \mathcal{A}^a[\![a]\!]$.*

**Proof** (Sketch) We must show that, for all $a \in \mathit{Asgn}$, $s \in S$, and $c \in C$, whenever $s \; R \; c$ we have $\mathcal{A}[\![a]\!]s \; R \; \mathcal{A}^a[\![a]\!]c$. Suppose we have $s \; R \; c$, that is $(g, v) \; R \; (p, b, r)$. We must show that the states which result after performing an assignment are also related; the possible cases are:

- $\mathcal{A}[\![x :- y]\!]s \; R \; \mathcal{A}^a[\![x :- y]\!]c$.

  The object graph $g$ of $s$ remains the same, and $x$ takes the value of $y$ in $v$. This operation is accurately represented in the abstract domain by removing $x$ from the equivalence class where it was and adding it to the equivalence class where $y$ is, and making corresponding adjustments to $b$ and $r$.

- $\mathcal{A}[\![x :- y.\, z]\!]s \; R \; \mathcal{A}^a[\![x :- y.\, z]\!]c$.

  The object graph remains the same, and $x$ takes the value of a state variable in the object referenced by $y$. If $x$ (and $z$) is non-balloon, $y.\, z$ refers to a non-balloon object in the same cluster as the object referenced by $y$. Therefore, the corresponding operation in the abstract domain is the same as for $x :- y$.

  Otherwise, $x$ is balloon, and $y.\, z$ refers to a balloon object in a cluster different from the one referenced by $y$. No representation is kept in the abstract domain for the structure of an object and therefore there will be some loss of information in this case: we must put $x$ on its own in a new equivalence class (which will be mapped by $b$ to 1). (This is one of the reasons why the concretisation function was defined so that $\gamma(\boxed{\text{w}}\,\boxed{\text{x}})$ represents states in which $w$ and $x$ may or may not point to the same cluster.) In what concerns inter-cluster relationships, the assignment makes $x$ refer to an object in a cluster reachable from $y$; in the abstract domain there are three cases for the corresponding operation:

  – if $by = 0$ we must say that $y$ 'reaches' the cluster referenced by $x$, i.e. add the new equivalence class of $x$ to those related to the equivalence class of $y$;

  – otherwise (being $by = 1$), if there exists some variable $w$ which relates to $y$, we must add the new equivalence class of $x$ to those related to the equivalence class of $w$;

  – otherwise, no identifier will relate to $x$.

- $\mathcal{A}[\![x :- \texttt{null}]\!]s \; R \; \mathcal{A}^a[\![x :- \texttt{null}]\!]c$.

  The object graph remains the same, and $x$ becomes the null reference. Therefore, $x$ does not reference any object in any cluster referenced by other variable. In the abstract domain this corresponds to removing $x$ from the equivalence class where it was and making $x$ on its own a new equivalence class, with corresponding adjustments to $b$ and $r$.

- $\mathcal{A}[\![x :- \texttt{new}]\!]s \; R \; \mathcal{A}^a[\![x :- \texttt{new}]\!]c$.

  A new cluster consisting of only one object is added to the object graph, and $x$ is made to reference it. In the abstract domain this corresponds to making $x$ into a new equivalence class, as in the previous case.

- $\mathcal{A}[\![x.\,y :- z]\!]s \; R \; \mathcal{A}^a[\![x.\,y :- z]\!]c$.

  This assigment changes the object graph by updating a state variable of an object. If $z$ is balloon, the assignment stores a reference to a balloon in a state variable of an object, something which may violate $I_1$ (if there exists already one reference to the object); therefore the resulting state in the abstract domain must be $\top$. (This corresponds to the simple rule.) If $z$ is non-balloon, the assignment merges clusters (if $x$ and $z$ do not already point to objects in the same cluster). The corresponding operation in the abstract domain is what we denote by $[\triangleright\triangleleft]$; the possible cases were described in detail when it was defined.

  This statement can also result in the division of a cluster in two, which would correspond to a smaller abstract state. however, not having information in the abstract domain about the structure of objects, we must incur an information loss and do not represent the corresponding division of equivalence classes.

- $\mathcal{A}[\![x.\,y :- \texttt{null}]\!]s \; R \; \mathcal{A}^a[\![x.\,y :- \texttt{null}]\!]c$.

  A state variable of an object is made to be the null reference. Either it results in a state which has the same corresponding abstract state as before; or it may cause the division of a cluster in two or make a cluster cease to be 'reached' by another, with corresponding abstract state which is smaller than the original. We must, as in the previous case, be conservative and use the larger of the two possible outcomes in the abstract domain; in this case we must maintain the abstract state before the operation.

- $\mathcal{A}[\![x.\,y :- \texttt{new}]\!]s \; R \; \mathcal{A}^a[\![x.\,y :- \texttt{new}]\!]c$.

  A state variable of an object is made to reference a newly created object. In the abstract domain this has exactly the same effect as the previous case.

- $\mathcal{A}[\![x <- e]\!]s \; R \; \mathcal{A}^a[\![x <- e]\!]c$.

  The value of an integer object is changed. This has no effect on the abstract representation.

□

## 5.5 Composite Commands, Functions and the Program

Composite commands include conditionals; a join operation in the abstract domain becomes the natural choice in defining their semantics and our approach is, as we have discussed, to use a completion of the base abstract domain—the set of down-sets—in defining the abstract semantics for commands. We define thus the lattice:

$$D = \mathcal{O}(C_\top),$$

and define $R$ between $s \in S$ and $o \in D$ as:

$$s\ R\ o \Leftrightarrow \exists c \in o.\ s\ R\ c.$$

A composite command may include function invocations; this leads to the use of an environment for functions in describing the semantics of composite commands (as we have already done in the standard semantics). The abstract semantic function for commands is thus

$$\mathcal{C}^a : \mathit{Com} \to F^a \to D \to D,$$

with

$$F^a = F^a_1 \times \cdots \times F^a_k,$$

where

$$F^a_i = [I^{a_i+1} \to D \to D].$$

The definition of $\mathcal{C}^a$ is given in Figure 5.11.

**Proposition 5.19** *For all $c \in \mathit{Com}$, $\mathcal{C}[\![c]\!]\ R\ \mathcal{C}^a[\![c]\!]$.*

**Proof** We must show that, for all $c \in \mathit{Com}$, $\varphi \in F$, $\varphi^a \in F^a$, $s \in S$, and $o \in D$, whenever $\varphi\ R\ \varphi^a$ and $s\ R\ o$ we have $\mathcal{C}[\![c]\!]\varphi s\ R\ \mathcal{C}^a[\![c]\!]\varphi^a o$. The proof is by induction on the structure of commands; for an assignment $a$, the abstract function either propagates $\top$ or returns the join $j$ of the principal ideals generated by the elements which result from applying the semantic function for assignments to each maximal element in $o$. As $s\ R\ o$, for at least one such maximal element $m$ we must have

$$\begin{array}{lcl}
\mathcal{C}^a[\![a]\!] & = & \lambda\varphi^a.\,\lambda o.\begin{cases}\top & \text{if } o = \top,\\ \bigsqcup\limits_{c\in\mathrm{Max}\,o} \downarrow\mathcal{A}[\![a]\!]c & \text{otherwise.}\end{cases}\\
\mathcal{C}^a[\![c_0; c_1]\!] & = & \lambda\varphi^a.\,\lambda o.\,\mathcal{C}^a[\![c_1]\!]\varphi^a(\mathcal{C}^a[\![c_0]\!]\varphi^a o)\\
\mathcal{C}^a[\![\texttt{if}\ e\ \texttt{then}\ c_0\ \texttt{else}\ c_1]\!] & = & \lambda\varphi^a.\,\lambda o.\,\mathcal{C}^a[\![c_0]\!]\varphi^a o \sqcup \mathcal{C}^a[\![c_1]\!]\varphi^a o\\
\mathcal{C}^a[\![\texttt{while}\ e\ \texttt{do}\ c]\!] & = & \lambda\varphi^a.\,\mathrm{fix}\,\lambda h.\,\lambda o.\,h(\mathcal{C}^a[\![c]\!]\varphi^a o) \sqcup o\\
\mathcal{C}^a[\![x :-\ f_i(x_1, \ldots, x_{a_i})]\!] & = & \lambda\varphi^a.\,\lambda o.\,\varphi^a_i(x, x_1, \ldots, x_{a_i})o
\end{array}$$

Figure 5.11: Abstract semantics for composite commands

$$\mathcal{D}^a[\![d]\!] \quad = \quad \mathrm{fix}\,\lambda\varphi^a.\,(\overrightarrow{\varphi'_k}),$$

where

$$\varphi'_i = \lambda(x, \overrightarrow{x_{a_i}}).\,\lambda o.$$

$$\mathcal{C}^a[\![c_i[\mathrm{ren}\,z/z \mid z \in \mathrm{var}\,c_i][\overrightarrow{x_{a_i}}/\overrightarrow{\mathrm{ren}\,x_{ia_i}}]; x :-\ \mathrm{ren}\,y_i]\!]\varphi^a(o \overset{+}{\mid} X) \mid \mathrm{var}\,o$$

with $X$ a set of new identifiers, $|X| = |\mathrm{var}\,c_i|$,

and $\mathrm{ren} : \mathrm{var}\,c_i \to X$, bijective;

$d$ as given by the abstract syntax in Figure 5.1.

$$\mathcal{P}^a[\![d\ \texttt{do}\ c]\!] \quad = \quad \mathcal{C}^a[\![c]\!]\mathcal{D}^a[\![d]\!]\!\downarrow\!\mathbf{1}$$

Figure 5.12: Abstract semantics for functions and the program

$s\ R\ m$, and so $\mathcal{A}[\![a]\!]s\ R\ \mathcal{A}^a[\![a]\!]m$, and also $\mathcal{A}[\![a]\!]s\ R\ j$; therefore $\mathcal{C}[\![a]\!]\varphi s\ R\ \mathcal{C}^a[\![a]\!]\varphi^a o$. All remaining cases are trivial (compare the definitions in Figures 5.4 and 5.11). □

Note how the integer expression in the conditional and loop is ignored, and the conservative join is used; this is why there is no abstract semantic function for integer expressions. Also, the domain for function environment matches perfectly a function invocation. The remaining abstract semantic functins, for declaration of functions ($\mathcal{F}^a$ : *Decl* $\to F^a$) and the program ($\mathcal{P}^a$ : *Prg* $\to D$), are given in Figure 5.12.

Again, there is a perfect match between the concrete and abstract counterparts (compare with Figure 5.4). Domain extension for abstract states is defined to reflect the initialisation to `null` references in the concrete domain:

$$(p,b,r) \overset{+}{|} X = (p \cup \{(x,x) \mid x \in X\}, b[x \mapsto \begin{cases} 1 & \text{if balloon}\, x, \\ 0 & \text{otherwise.} \end{cases} \mid x \in X\}, r).$$

This is naturally lifted to down-sets as:

$$o \overset{+}{|} X = \bigsqcup_{m \in \mathrm{Max}\, o} \downarrow(m \overset{+}{|} X).$$

By the definition of $\overset{+}{|}$ we have that if $s\ R\ c$ then $(s \overset{+}{|} X)\ R\ (c \overset{+}{|} X)$, and therefore if $s\ R\ o$ we have $(s \overset{+}{|} X)\ R\ (o \overset{+}{|} X)$. Given that the concrete and abstract interpretations are related for all possible commands (previous proposition), it follows trivially that:

**Proposition 5.20** *For all $d \in$ Decl, $\mathcal{D}[\![d]\!]\ R\ \mathcal{D}^a[\![d]\!]$.*

And finally, given that $(\bot_G, \bot_V)\ R\ \mathbf{1}$, we have that:

**Proposition 5.21** *For all $p \in$ Prg, $\mathcal{P}[\![p]\!]\ R\ \mathcal{P}^a[\![p]\!]$.*

As all the abstract states except $\top$ represent sets of concrete states where the balloon invariant holds, it follows that for any given program $p$, if $\mathcal{P}^a[\![p]\!] \neq \top_D$ then the balloon invariant holds in the final state resulting from running $p$ (if the program does not abort or diverge).

Moreover, the invariant will also hold at any intermediate state during the execution of $p$, as desired. This follows from the fact that semantic functions for commands 'memorise' $\top$: once $\top$ results at any intermediate state, it will be propagated to the final result. This way, if at any intermediate point in the program the balloon invariant is broken, the corresponding abstract state ($\top$) is propagated to the final result (even if the balloon invariant happens to hold in the final state).

Balloon type-checking a program $p$ can thus be as follows: if $\mathcal{P}^a[\![p]\!] \neq \top_D$ the program is accepted, otherwise the program is rejected.

## 5.6 Summary

We have presented the basis for the checking mechanism for balloon types: an abstract interpretation whose outcome serves to decide on the acceptance of the program. For this we defined a small but illustrative imperative language (RISO), together with a corresponding concrete denotational semantics. To concentrate on the essentials, we have only assumed a flat set of declarations of functions (no declarations of data types), having postponed modularity to a separate chapter. Here we have focused on having variables which are references to possibly shared objects, and having classic control flow structures and first-order functions (possibly mutually recursive).

A central point in the abstract interpretation is the base domain itself, in contrast with typical abstract interpretations for functional languages where the base domains are simple and the emphasis is on the higher-order features. As a remark, it was indeed the complexity we faced in representing information for the static analysis which led us to use a formal approach and to abandon an error-prone ad-hoc approach.

The base domain was carefully designed, tailored to the problem at hand. We have presented it in two stages. First information about clusters: whether variables may reference the same cluster and whether it may contain a balloon object; then, information about cluster relationships.

Then, we have presented the abstract semantics for assignments and, finally, the abstract semantics for composite commands and functions. The representation of functions here was a naive one, not the one we intend to use in some realistic implementation, which we have left to be addressed in the next chapter.

# Chapter 6

# Avoiding Function Spaces

There is an aspect in the described abstract interpretation which makes it unsuitable as a static analysis: the naive way the recursive declaration of functions is treated. While being an elegant description of the interprocedural component of the semantics and allowing the concrete and abstract interpretations to be trivially related, the naive representation of a function by a function leads to computational complexity problems.

In our case, even though we have a first-order language, we have a quite complex base domain ($C$) with size growing exponentially with the number of identifiers, and use the completion by down-sets ($D$) for the state in the case of commands. The representation of a function is by an element of $F_i^a = [I^{a_i+1} \to D \to D]$, we have a tuple of such elements, and we need to calculate a fixed point of a function from tuples to tuples. This is most unrealistic from an implementation point of view. To make matters worse, the 'inlining' semantics, with the associated domain extension (adding the local variables to the caller environment), means that we have no bound on the number of identifiers, as opposed to having a small number of identifiers to consider separately in each function. The above means that we need something more than just some general purpose technique in abstract interpretation (like frontiers [23, 44]) in order to obtain a feasible static analysis.

During the development of the abstract domain and semantic functions we have observed a property to hold in our particular interpretation, which can be exploited to obtain a compact representation of functions and be the solution to a realistic static analysis. We now describe it, starting with the intuition behind it.

## 6.1 Intuition

Two relevant points can be made regarding functions. The first is that the semantics of function declaration and invocation are defined so that the effect of an invocation is the same as if the body of the function is inlined in the calling context (after renaming of formal to actual parameters and renaming of local variables to avoid conflicts). The second is that in the body of a function there are no assignments to formal parameters.

From these two points it follows that the possible effects of a function on the calling state are the ones due to an arbitrary composite command, which does not contain function invocations, and does not make assignments to variables from the calling state. (Although may make assignments to some extra variables that correspond to the local variables in the function.)

Moreover, to see what are the possible effects on a base state we can ignore conditionals and loops; they have only the effect of merging different antichains of base states and do not modify a base state itself. We can, therefore, concentrate on the possible effects of sequences of assignments.

Looking at the effect of a sequence of assignments which contains no assignment to a set of variables $A$, when considering the domain restricted to $A$ (i.e. ignoring local variables of the function, which may be assigned), we can observe that the state can only become larger. The state moves up via a few possible operations which can either cause merging, capture, or reaching. It is not possible for these operations to be 'undone' or for different input states to be moved to some 'fixed' state. This means that the effect of a function on the calling context has an incremental nature.

The above points in the direction that: knowing the response of the function to the **1** state (the more precise state, which allows for more merging, captures and reaching to be made) characterises the 'increment', and makes it possible to extrapolate the response of the function to any other input state. Actually, for the property to hold, we need to instrument the **1** with 'shadow' variables for balloon parameters, as we discuss later.

As an example, suppose a calling context with non-balloon variables $x, y$, and an invocation $f(x, y)$ (we ignore for now the returned variable). If $f$ is

```
f(x,y) do
  x.a :- y
```

the response to the **1** state is $\overline{\underline{\mathrm{xy}}}$, expressing that the function merges $x$ and $y$.

(We have used the same identifiers in actual and formal variables to make it easier to compare states.)

The response of the function to input $\overline{\underline{\mathtt{x}}}\,\boxed{\mathtt{y}}$ will be $\boxed{\mathtt{xy}}$, and the response to both $\overline{\underline{\mathtt{x}\boxed{\mathtt{y}}}}$ and $\boxed{\mathtt{x}}\,\boxed{\mathtt{y}}$ will be $\top$. In all cases the output states can be calculated by 'applying' the merging expressed by $\overline{\underline{\mathtt{xy}}}$ to the input states, via a simple algorithm.

It should be obvious that the response to **1**, $\overline{\underline{\mathtt{xy}}}$, does not define the body of $f$ univocally (as there can be many different commands with the same response to **1**). However, it characterises $f$ because any other command with the same response to **1** will have the same effect on the calling state. Some examples are

```
g(x,y) do
  y.a :- x
```

and

```
h(x,y) do
  l :- new;
  x.a :- l;
  l.a :- y
```

where $l$ is a local variable of non-balloon type. For these functions, both the response to **1** and to the above mentioned states is the same as for $f$.

We can have, therefore, a compact representation of each function $f_i$: an element of $D$ (the response to **1**) instead of an element of $F_i^a = [I^{a_i+1} \to D \to D]$. Towards this we need to substitute the direct function application in the semantics of function invocation by a computation which combines the input state and the response to **1** to obtain the output state. In particular, we need to define the 'simple algorithm' which performs this computation.

Although both the input state and the representation of a function will be elements of $D$, each maximal element can be considered individually (as in the semantic function $\mathcal{C}^a[\![a]\!]$), and we can concentrate on elements of $C$. We need, therefore, to define a function which combines $i : C$ (one of the maximal elements of the input state) and $f : C$ (one of the maximal elements of the response to **1**) and returns $o : C_\top$ (one of the possibilities to be joined in the output). We will call this function *apply*, as it 'applies' the representation of a function to an input to obtain an output: $o = \text{apply}(f, i)$.

## 6.2 The Apply Function

This function takes two elements, $i = (p_i, b_i, r_i)$ and $f = (p_f, b_f, r_f)$, and returns either $\top$ or $(p, b, r)$. There are two main aspects to consider in the function: one is determining whether the output is or not $\top$; another is computing the components $(p, b, r)$ in the case when the output is not $\top$.

It turns out that, to decide whether the output is $\top$, it is useful to determine tentative $p$, $b$, and $r$: the components that would result if the output is not $\top$. We consider now each of these components in turn. The output can be $\top$ only due to an operation $[x \rhd\lhd y]$ being performed at some point when either:

- $x$ and $y$ are in different equivalence classes mapped to 1, i.e. $\boxed{\text{x}}\,\boxed{\text{y}}$ (which we address while considering component $b$); or
- $x$ and $y$ are 'related', i.e. $\overline{\underline{\text{x}\boxed{\text{y}}}}$ or $\overline{\underline{\text{y}\boxed{\text{x}}}}$ (which we address while considering component $r$.

### 6.2.1 The $p$ Component

If the output is not $\top$, the resulting $p$ will depend only on the components $p_i$ and $p_f$. A set of identifiers $\{x, y, z\}$ in a given equivalence class in $p_f$, means that some operations were performed, like $[x \rhd\lhd y]; [x \rhd\lhd z]$ or $[y \rhd\lhd z]; [z \rhd\lhd x]$, or other combinations possibly involving local variables. The effect will be that $x$, $y$ and $z$ will be in the same equivalence class in $p$, together with whatever other identifiers were in each of the equivalence classes of $x$, $y$ and $z$ in $p_i$. As an example, given

$$\begin{aligned} i &= \overline{\underline{\text{xy}}}\,\overline{\underline{\text{zw}}}, \\ f &= \overline{\underline{\text{x}}}\,\overline{\underline{\text{yz}}}\,\overline{\underline{\text{w}}}, \end{aligned}$$

the output should be

$$o = \overline{\underline{\text{xyzw}}}.$$

Another example is

$$\begin{aligned} i &= \overline{\underline{\text{ab}}}\,\overline{\underline{\text{cd}}}\,\overline{\underline{\text{ef}}}, \\ f &= \overline{\underline{\text{a}}}\,\overline{\underline{\text{bc}}}\,\overline{\underline{\text{de}}}\,\overline{\underline{\text{f}}}, \\ o &= \overline{\underline{\text{abcdef}}}. \end{aligned}$$

This is accomplished by making the union of $p_i$ and $p_f$ and taking the transitive closure:

$$p = (p_i \cup p_f)^+.$$

### 6.2.2 The $b$ Component

If the output is not $\top$, an equivalence class $X$ in $p$ will be mapped to 1 if at least one of the equivalence classes in $p_f$ or $p_i$ which are merged to make up $X$ is mapped to 1; otherwise it will be mapped to 0. As examples, we have

$$\begin{array}{rcl} i & = & \overline{\underline{\mathrm{x}}}\,\boxed{\mathrm{y}}, \\ f & = & \overline{\underline{\mathrm{xy}}}, \\ o & = & \boxed{\mathrm{xy}}; \end{array}$$

and

$$\begin{array}{rcl} i & = & \overline{\underline{\mathrm{x}}}\,\overline{\underline{\mathrm{yx}}}, \\ f & = & \overline{\underline{\mathrm{xy}}}\,\boxed{\mathrm{z}}, \\ o & = & \boxed{\mathrm{xyz}}. \end{array}$$

This is accomplished by making:

$$\begin{aligned} b = \ & \{x \mapsto \bigsqcup\{b_i y \mid y \in \operatorname{dom} b_i \wedge x\ p\ y\} \\ & \qquad \sqcup \bigsqcup\{b_f y \mid y \in \operatorname{dom} b_f \wedge x\ p\ y\} \mid x \in \operatorname{dom}(b_i \cup b_f)\}. \end{aligned}$$

Now we show how to detect whether the result is $\top$ due to an operation like $\boxed{\mathrm{x}}\,\boxed{\mathrm{y}}[x \rhd\!\lhd y]$ being performed, merging two different captured equivalence classes. There are three cases to consider:

- The two captured equivalence classes exist in the input state. An example is

  $$\begin{array}{rcl} i & = & \boxed{\mathrm{x}}\,\overline{\underline{\mathrm{yz}}}\,\boxed{\mathrm{w}}, \\ f & = & \overline{\underline{\mathrm{xy}}}\,\overline{\underline{\mathrm{zw}}}, \\ o & = & \top. \end{array}$$

  This situation is detected by the predicate

  $$\exists (x, y) \in p.\, b_i x = 1 \wedge b_i y = 1 \wedge x \not{p}_i\, y.$$

- The two equivalence classes become captured during the execution of the function. To detect if a function creates some capture (as opposed to manipulating already captured equivalence classes), it is not enough to test whether equivalence classes in $p_f$ are mapped to 1 by $b_f$ and to use a predicate analogous to the previous one:

$$\exists (x, y) \in p.\, b_f x = 1 \wedge b_f y = 1 \wedge x \not p_f\, y.$$

It is necessary to consider whether variables are of balloon or non-balloon type. The apply function operates on the response to **1**, but this is not a unique state common to all functions, but depends on whether parameters are of balloon type. As an example, if we have a function with parameters $x$, $y$, and $z$, with balloon $x$, balloon $y$, and $\neg$ balloon $z$, the corresponding **1** is $\boxed{\mathrm{x}}\,\boxed{\mathrm{y}}\,\overline{\underline{\mathrm{z}}}$. If the function does not perform any action at all (and therefore does not cause any capture) the response to **1** will be **1** itself:

$$f = \boxed{\mathrm{x}}\,\boxed{\mathrm{y}}\,\overline{\underline{\mathrm{z}}}.$$

This function when applied to an input

$$i = \boxed{\mathrm{xy}}\,\boxed{\mathrm{z}}$$

should produce an identical output

$$o = \boxed{\mathrm{xy}}\,\boxed{\mathrm{z}}.$$

However, the above predicate would be true resulting (incorrectly) in $\top$. The function creates capture only if all the identifiers in a captured equivalence class in the response to **1** are not already captured in **1**; i.e. are not of balloon type. This can be detected by:

$$\exists x \in I.\, b_f x = 1 \wedge (\forall y \in p_f\{x\}.\, \neg\, \text{balloon}\, y).$$

To detect that we have two such equivalence classes captured by the function and merged, resulting in $\top$, we can use:

$$\begin{aligned}\exists (x, y) \in p.\, x \not p_f\, y \wedge b_f x = 1 \wedge (\forall z \in p_f\{x\}.\, \neg\, \text{balloon}\, z)\\ \wedge\, b_f y = 1 \wedge (\forall z \in p_f\{y\}.\, \neg\, \text{balloon}\, z)\end{aligned}$$

An example (where all variables are of non-balloon type) is:

$$\begin{aligned} i &= \overline{\underline{\mathrm{xy}}}\,\overline{\underline{\mathrm{zw}}},\\ f &= \boxed{\mathrm{x}}\,\overline{\underline{\mathrm{yz}}}\,\boxed{\mathrm{w}},\\ o &= \top. \end{aligned}$$

- One equivalence class is already captured in the input state, while the other becomes captured during the execution of the function. An example is

  $$\begin{aligned} i &= \boxed{\mathrm{x}}\,\overline{\underline{\mathrm{yz}}}, \\ f &= \overline{\underline{\mathrm{xy}}}\,\boxed{\mathrm{z}}, \\ o &= \top. \end{aligned}$$

  This can be detected by a predicate which combines the two previous cases:

  $$\exists (x, y) \in p.\, b_i x = 1 \wedge b_f y = 1 \wedge (\forall z \in p_f\{y\}.\, \neg\, \mathrm{balloon}\, z).$$

The second and third cases can be factorised into a single predicate, which simplifies to:

$$\begin{aligned} \exists (x, y) \in p.\ b_f x = 1 \wedge (\forall z \in p_f\{x\}.\, \neg\, \mathrm{balloon}\, z) \\ \wedge\, (b_i y = 1 \vee (x \not{p}_f\, y \wedge b_f y = 1)). \end{aligned}$$

### 6.2.3 The $r$ Component

As for the previous components, we will first show how to calculate $r$ supposing the output is of the form $(p, b, r)$; then we show how to decide if the output is $\top$.

First some intuition should be given regarding the appearance of $\overline{\underline{\mathrm{x}\boxed{\mathrm{y}}}}$ as part of $f$. As a function cannot assign to parameters, the only way a parameter can become 'reached' is by being captured into a reached cluster. (This means that if the argument is already captured it cannot become reached due to the function, as it would result in $\top$.) An example is the function

```
f(x,y,z,w) do
  b :- y.b;
  b.a :- z
```

where parameters are of non-balloon type and $b$ is of balloon type. The response to **1** of this function would be

$$f = \overline{\underline{\mathrm{x}}}\,\overline{\underline{\mathrm{y}\boxed{\mathrm{z}}}}\,\overline{\underline{\mathrm{w}}}.$$

For this function we have the following examples of input/output:

$$\begin{aligned} i &= \overline{\underline{\mathrm{xy}}}\,\overline{\underline{\mathrm{zw}}}, \\ o &= \overline{\underline{\mathrm{xy}\boxed{\mathrm{zw}}}}, \end{aligned}$$

and

$$\begin{aligned} i &= \underline{\overline{\mathrm{x}\boxed{\mathrm{y}}}}\ \underline{\overline{\mathrm{z}\boxed{\mathrm{w}}}}\,, \\ o &= \underline{\overline{\mathrm{x}\boxed{\mathrm{y}}\ \boxed{\mathrm{z}}\ \boxed{\mathrm{w}}}}\,. \end{aligned}$$

Other simple examples of input/output for different functions are:

$$\begin{aligned} i &= \underline{\overline{\mathrm{x}\boxed{\mathrm{y}}}}\ \underline{\overline{\mathrm{z}\boxed{\mathrm{w}}}}\,, \\ f &= \underline{\overline{\mathrm{x}}}\ \underline{\overline{\mathrm{yz}}}\ \underline{\overline{\mathrm{w}}}\,, \\ o &= \underline{\overline{\mathrm{x}\boxed{\mathrm{yz}}\ \boxed{\mathrm{w}}}}\,; \end{aligned}$$

and

$$\begin{aligned} i &= \underline{\overline{\mathrm{x}}}\ \underline{\overline{\mathrm{yz}}}\ \underline{\overline{\mathrm{w}}}\,, \\ f &= \underline{\overline{\mathrm{x}\boxed{\mathrm{y}}}}\ \underline{\overline{\mathrm{z}\boxed{\mathrm{w}}}}\,, \\ o &= \underline{\overline{\mathrm{x}\boxed{\mathrm{yz}}\ \boxed{\mathrm{w}}}}\,. \end{aligned}$$

Other examples with more variables, which show several aspects combined are:

$$\begin{aligned} i &= \underline{\overline{\mathrm{a}\boxed{\mathrm{b}}}}\ \underline{\overline{\mathrm{c}\boxed{\mathrm{d}}}}\ \underline{\overline{\mathrm{e}\boxed{\mathrm{f}}}}\,, \\ f &= \underline{\overline{\mathrm{a}}}\ \underline{\overline{\mathrm{bc}}}\ \underline{\overline{\mathrm{d}\boxed{\mathrm{e}}}}\ \underline{\overline{\mathrm{f}}}\,, \\ o &= \underline{\overline{\mathrm{a}\boxed{\mathrm{bc}}\ \boxed{\mathrm{d}}\ \boxed{\mathrm{e}}\ \boxed{\mathrm{f}}}}\,; \end{aligned}$$

and

$$\begin{aligned} i &= \underline{\overline{\mathrm{a}\boxed{\mathrm{b}}}}\ \underline{\overline{\mathrm{cd}}}\ \underline{\overline{\mathrm{e}\boxed{\mathrm{f}}}}\,, \\ f &= \underline{\overline{\mathrm{a}}}\ \underline{\overline{\mathrm{bc}}}\ \underline{\overline{\mathrm{de}}}\ \underline{\overline{\mathrm{f}}}\,, \\ o &= \underline{\overline{\mathrm{a}\boxed{\mathrm{bcde}}\ \boxed{\mathrm{f}}}}\,. \end{aligned}$$

These examples show that, in calculating $r$, there is the need to:

- Merge the reaching in the input with that added by the function: this can be accomplished by making the union of the relations, extending it to other identifiers in the resulting equivalence classes, and taking the transitive closure.

- Make an adjustment to remove $x\ r\ y$ when $bx = 1$, according to the rules for well-formedness of $(p, b, r)$.

Component $r$ can, thus, be computed by the following:

$$\begin{aligned}
r_1 &= r_i \cup r_f, \\
r_2 &= \{(x,w) \mid \exists y, z.\, x\ p\ y \wedge y\ r_1\ z \wedge z\ p\ w\}, \\
r_3 &= r_2^+, \\
r &= r_3 \setminus \{(x,y) \mid x\ r_3\ y \wedge bx = 1\}.
\end{aligned}$$

Now we show how to detect whether the result is $\top$ due to an operation like $\overline{\underline{\mathrm{x}\boxed{\mathrm{y}}}}\,[x \rhd\!\lhd y]$ being performed. Two simple examples when $\top$ occurs are:

$$\begin{aligned}
i &= \overline{\underline{\mathrm{x}\boxed{\mathrm{y}}}}, \\
f &= \overline{\underline{\mathrm{xy}}}, \\
o &= \top;
\end{aligned}$$

and

$$\begin{aligned}
i &= \overline{\underline{\mathrm{xy}}}, \\
f &= \overline{\underline{\mathrm{x}\boxed{\mathrm{y}}}}, \\
o &= \top.
\end{aligned}$$

These situations could be detected by a predicate like:

$$\exists x, y.\, (x\ r_i\ y \vee x\ r_f\ y) \wedge x\ p\ y.$$

The above predicate would not, however, detect $\top$ in the following case:

$$\begin{aligned}
i &= \overline{\underline{\mathrm{x}\boxed{\mathrm{y}}}}\ \overline{\underline{\mathrm{z}\boxed{\mathrm{w}}}}, \\
f &= \overline{\underline{\mathrm{xw}}}\ \overline{\underline{\mathrm{yz}}}, \\
o &= \top.
\end{aligned}$$

Here $\top$ results due to sequences of operations like:

$$\overline{\underline{\mathrm{x}\boxed{\mathrm{y}}}}\ \overline{\underline{\mathrm{z}\boxed{\mathrm{w}}}} \xrightarrow{[y \rhd\!\lhd z]} \overline{\underline{\mathrm{x}\boxed{\mathrm{yz}}\ \boxed{\mathrm{w}}}} \xrightarrow{[x \rhd\!\lhd w]} \top,$$

or

$$\overline{\underline{\mathrm{x}\boxed{\mathrm{y}}}}\ \overline{\underline{\mathrm{z}\boxed{\mathrm{w}}}} \xrightarrow{[x \rhd\!\lhd w]} \overline{\underline{\mathrm{z}\boxed{\mathrm{xw}}\ \boxed{\mathrm{y}}}} \xrightarrow{[y \rhd\!\lhd z]} \top.$$

We must use a predicate which detects reaching created during the execution of the function. For this we cannot, however, use the final $r$ as calculated above, because some reaching has already been removed so that $x\ r\ y$ does not hold when

$bx = 1$, as one of the requirements to have $(p, b, r)$ well formed. In this example, $(p, b, r)$ calculated by the above formulae would be:

$$(p, b, r) = \boxed{\text{xw}}\,\boxed{\text{yz}},$$

where we have $r = \emptyset$.

We can, however, detect all reaching which results from the invocation by using $r_3$, where no removal has yet been performed, and test whether related identifiers have become merged in $p$. This way we can have the following predicate to test for the occurrence of $\top$:

$$\exists (x, y) \in p.\, x \; r_3 \; y.$$

We have by now predicates which test the possible cases when $\top$ can occur, and formulae for calculating $(p, b, r)$ otherwise; combining them we have the function apply : $C \times C \to C_\top$:

**Definition 6.1**

$$\begin{array}{rcl}
\text{apply} & = & \lambda((p_f, b_f, r_f), (p_i, b_i, r_i)).\\
& & \left\{\begin{array}{ll}
\top & \text{if } \exists (x, y) \in p.\, b_i x = 1 \wedge b_i y = 1 \wedge x \not{p}_i \; y,\\
\top & \text{if } \exists (x, y) \in p.\; b_f x = 1 \wedge (\forall z \in p_f\{x\}.\, \neg\, \text{balloon}\, z)\\
& \qquad\qquad \wedge\, (b_i y = 1 \vee (x \not{p}_f \; y \wedge b_f y = 1)),\\
\top & \text{if } \exists (x, y) \in p.\, x \; r_3 \; y,\\
(p, b, r) & \text{otherwise,}
\end{array}\right.
\end{array}$$

where

$$\begin{array}{rcl}
p & = & (p_i \cup p_f)^+,\\
b & = & \{x \mapsto \bigsqcup\{b_i y \mid y \in \text{dom}\, b_i \wedge x \; p \; y\}\\
& & \qquad \sqcup \bigsqcup\{b_f y \mid y \in \text{dom}\, b_f \wedge x \; p \; y\} \mid x \in \text{dom}(b_i \cup b_f)\},\\
r_1 & = & r_i \cup r_f,\\
r_2 & = & \{(x, w) \mid \exists y, z.\, x \; p \; y \wedge y \; r_1 \; z \wedge z \; p \; w\},\\
r_3 & = & r_2^+,\\
r & = & r_3 \setminus \{(x, y) \mid x \; r_3 \; y \wedge bx = 1\}.
\end{array}$$

## 6.3 Equivalent Semantics for Functions

We can now reformulate the abstract semantics to an equivalent one, making use of the apply function, and avoiding function spaces in the representation of functions.

The only changes are to the representation of functions, to the semantics for the declaration of functions, and to the semantics of the function invocation.

The representation of a function is now the 'response to **1**'. The new environment for functions becomes:

$$F^a = F_1^a \times \cdots \times F_k^a,$$

with

$$F_i^a = D.$$

The semantics for declaration of functions becomes:

$$\begin{aligned}
\mathcal{D}^a[\![d]\!] \quad &= \quad \mathrm{fix}\,\lambda\varphi^a.\,(\overrightarrow{\varphi'_k}),\\
&\text{where}\\
&\varphi'_i = (\mathcal{C}^a[\![c_i]\!]\varphi^a{\downarrow}\mathbf{1}') \mid (\{y_i, \overrightarrow{x_{ia_i}}\} \cup \mathrm{shadow}\,A),\\
&A = \{x_{ij} \mid j \in \{1, \ldots, a_i\} \wedge \mathrm{balloon}\,x_{ij}\},\\
&\mathbf{1}' = (\mathbf{1} \overset{+}{\mid} \mathrm{shadow}\,A)[\mathrm{shadow}\,x \overset{+}{\rightarrow} x \mid x \in A],\\
&d \text{ as given by the abstract syntax in Figure 5.1.}
\end{aligned}$$

As before, a fixed point is calculated, but now on a much smaller domain, resulting in a tuple of the 'response to **1**' of each body for the functions involved. For our approach to work, we need to solve a technical problem, which we do through what we call *shadow* variables, as we now discuss.

Under the base domain and operations that we have defined, reaching is only tracked when starting from a free variable, because non-balloons in captured clusters no longer can be captured. While this domain and operations preserve the relevant information if we are dealing with a fixed set of identifiers, the same cannot be said if we work with a given domain and then extend it to include more identifiers. Unfortunately, this is exactly what we do in our approach to dealing with function invocations.

The problem lies with formal parameters of balloon type. As they start mapped to 1 by $b$, no reaching starting from them is tracked in calculating the response to **1**. However, when calculating the effect of the function on the calling environment, there may be some variable which reaches one of such arguments of balloon type. This means that using a simple 'response to **1**' does not give the correct result; it discards essential information.

To solve this problem, we use a shadow variable for each formal parameter of balloon type. Each shadow behaves as a non-balloon variable, starts reaching

$$\mathcal{C}^a[\![x :- f_i(x_1,\ldots,x_{a_i})]\!] = \lambda\varphi^a.\,\lambda o.$$
$$\begin{cases} \top & \text{if } o = \top \vee \varphi_i^a = \top, \\ \displaystyle\bigsqcup_{(p_i,b_i,r_i)\in \mathrm{Max}\, o}\ \bigsqcup_{c_f \in \mathrm{Max}\,\varphi_i^a} \downarrow M & \text{otherwise.} \end{cases}$$
$$\text{where } M = \mathrm{apply}(c'_f, (p_i,b_i,r_i)) \overline{\mid} \{x\}[x/z],$$
$$\text{with } c'_f = c_f[z, x_1,\ldots,x_{a_i}/y_i, x_{i1},\ldots,x_{ia_i}]$$
$$[\bigsqcup\{u \mid u\ r_i\ x_j\}/\,\text{shadow}\, x_{ij} \mid \text{balloon}\, x_{ij} \wedge \exists u.\, u\ r_i\ x_j]$$
$$\overline{\mid}\{\text{shadow}\, x_{ij} \mid \text{balloon}\, x_{ij} \wedge \neg\exists u.\, u\ r_i\ x_j\},$$
and $z$ a new identifier.

Figure 6.1: Abstract semantics of function invocation

the corresponding parameter, and keeps track of reaching which may be needed to be propagated to the calling environment. In calculating the representation of a function, we use a **1** extended with the shadow variables. In the semantics we have used a function 'shadow' that maps a variable to an unused identifier (and we also use this function lifted to sets).

Note that the domain of the representation of a function is restricted to the formal parameters, shadow variables and the returned variable, forgetting all other local variables. This is important to keep down the size of the domains involved.

For the semantics of function invocation, instead of the previous direct (and naive):

$$\mathcal{C}^a[\![x :- f_i(x_1,\ldots,x_{a_i})]\!] \;=\; \lambda\varphi^a.\,\lambda o.\,\varphi_i(x,x_1,\ldots,x_{a_i})o$$

we now have the semantics as in Figure 6.1.

If either the input or the function representation is $\top$, the output will be also $\top$. Otherwise we make use of the apply function, performing the join over both the maximal elements of the input state and the maximal elements of the function representation. In doing this, several points are addressed:

- A renaming of formal to actual parameters in the function representation is made to place the computation in the domain of the calling environment.

- Each shadow variable is either renamed to a variable in the calling environment which reaches the corresponding actual parameter, or is removed from the domain. In choosing a variable to which rename a shadow variable, when several are possible, we resort to assuming a total order on $I$ and taking the greatest candidate.

- The returned variable $y_i$ which must be assigned to $x$ is considered as follows: while calculating the function representation, $y_i$ is kept in the domain; when making use of the apply function, $y_i$ is renamed to an unused identifier $z$ (as opposed to $x$); this allows both computing the effect of $x$ to the resulting state (as the original $x$ is used) and obtaining the 'value' of the returned variable; the identifier $z$ is used so that the function apply does not 'add' the effect of the return variable to the input state as it does for parameters; only as a final step is $x$ discarded from the domain and $z$ renamed to $x$, to consider the assignment.

## 6.4 Summary

We have presented a compact and efficient way to represent and compute functions in the abstract semantics, which replaces a naive function space by a single element. This makes use of the incremental nature of the effect of functions on the calling environment in our particular abstract semantics, which makes it possible to extrapolate the response of a function to any state knowing only the response to the more precise state. This extrapolation is made by an 'apply' function which we have derived and which is used in the abstract semantics for functions.

# Chapter 7

# Opaque Balloon Types

Plain balloon types are the first step in a hierarchy of mechanisms providing invariants that are useful to reason about programs. The balloon invariant is a basic support: it concerns static aliasing. By organising the object graph, it removes a first big obstacle in analysing programs with linked data structures.

Dynamic aliasing is an important aspect in establishing the possibility of interference between instructions, and in reasoning about programs. It has been left out from the concerns of the plain balloon mechanism, to be considered by the next layer in the hierarchy.

*Opaque balloon types* are an extension to plain balloon types; they constitute this next layer, devoted to dynamic aliasing. Here we motivate the concept, define the *opaque balloon invariant* and present the essentials of the corresponding checking mechanism.

The checking mechanism for opaque balloon types builds on what we call *nesting interpretation*, an abstract interpretation obtained by making a refinement in the representation of cluster relationships and extending it with what we call *nesting level* information. We describe the nesting interpretation for the intraprocedural case.

We conclude by outlining two approaches to obtaining a checking mechanism for opaque balloon types, making use of the nesting interpretation.

## 7.1 Introduction

In the preview of opaque balloons given in Chapter 3 we focused on the issue of returning references to client code: 'objects do not expose to clients any references to their internal state, even to be used temporarily by variables'. While not exposing internal state is the central issue, in the general case where mutually recursive data structures and functions are present, some care must be taken in defining what should be the properties of opaque balloon types and the corresponding checking mechanism.

The informal description does not cover the situation where some code external to an opaque balloon type $T$ is invoked from $T$ itself and is granted access to both an object $O : T$ and an object internal to $O$. For this reason, we will arrive at the precise definition of opaque balloon types from the perspective of the creation of dynamic aliasing.

The intuition about dynamic aliasing is that it cannot be prevented by a static mechanism without being overly conservative. This is true if we are trying to prevent *direct* aliases, i.e. to prevent `x` and `y` from referencing the same object. It is enough to recall the classic array subscript problem:

```
i := f(...);
j := g(...);
x :- a[i];
y :- a[j];
```

where it cannot be statically prevented that `x` and `y` are aliases without rejecting useful code, when the subscripts cannot be determined at compile time.

Even though it may be unrealistic to prevent aliasing as above, we can try to prevent what could be called *indirect* aliasing: the possibility of aliases being obtained using `x` and `y` as starting references, when `x` and `y` are not aliases themselves. (If `p` and `q` are of different types they cannot be aliases—unless one is a subtype of the other.)

As an example, if we have '`p: Point`' and '`r: Rectangle`', `p` and `r` will reference different objects. However, `p` may reference one of the points in the state of the rectangle, as in Figure 7.1; this may be the case even if both `Point` and `Rectangle` are balloon types: the balloon invariant does not concern dynamic references. This situation can lead to interference between instructions. Suppose both types have a move operation, which we apply to both `p` and `r`:

```
p.move(3,2);
r.move(2,4);
```

These two instructions would interfere, both updating the coordinates of a common point object. This may be an undesired accidental situation, and is an obstacle to reasoning about programs.

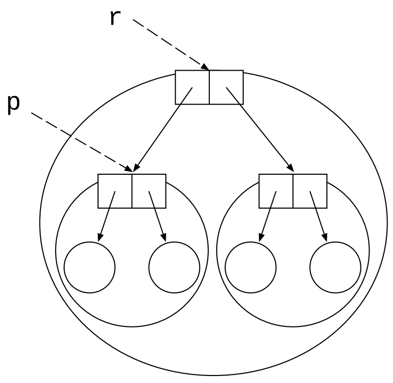


Figure 7.1: A dynamic reference to an internal object

The motivation behind opaque balloon types is to prevent the above situation from happening. As a first approximation we may say that, if `Rectangle` is an opaque balloon type, given a fragment of code with a `Rectangle` variable:

```
...
r: Rectangle
...
```

at run-time no variable in client code can reference an object internal to the rectangle referenced by `r`. (That is, in addition to state variables of external objects, as stated by the balloon invariant, also local variables in client code are prevented from referencing internal objects of a rectangle.)

Opaque balloon types enhance the characterisation of user-defined types given by plain balloons in what already happens for primitive types like integer:

- no variable can reference internal state of an integer object;
- although dynamic aliasing between variables `x` and `y` may exist (due to call by reference), a simple reference equality test is enough to see whether `x` and `y` may reach common state.

## 7.2 Opaque Balloon Invariant

From what we have discussed, the invariant should allow some arbitrary function to have references to objects internal to an opaque balloon $O$, as long as it is not able to reach $O$ itself. An example is a function invoked from the code in the implementation of an opaque balloon type. The class `Rectangle` could be something like:

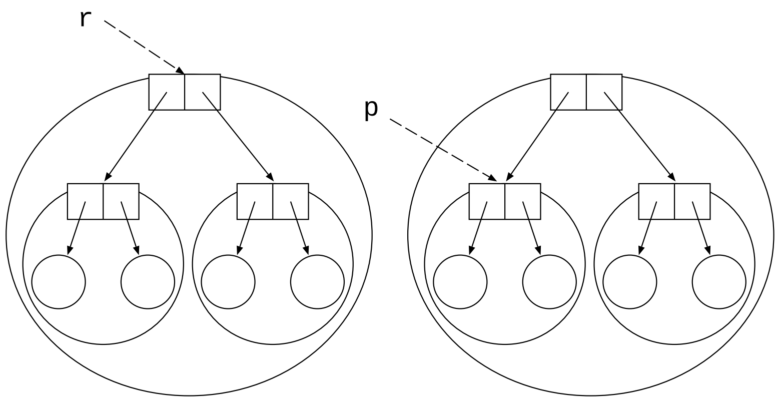


Figure 7.2: A reference to a point of an unrelated rectangle

```
class Rectangle
{
  p1,p2: Point;
  move(dx:Int, dy:Int)
  {
    p1.move(dx,dy);
    p2.move(dx,dy);
  }
  ...
}
```

The move method for rectangles invokes the corresponding move for each point in the rectangle. These methods from `Point` have access to objects internal to a rectangle, but not to the rectangle object itself.

In particular, the invariant should allow arbitrary code to have access both to an object of an opaque balloon class $C$ and to objects internal to a different instance of $C$; an example would be performing an invocation '`f(r,p)`' with `r` and `p` as in Figure 7.2. (In this case, as `p` refers to an object internal to a rectangle, the chain of invocations must have started from the rectangle class.)

Before presenting the invariant itself, we first define a basic notion on which it relies; this notion characterises the scenario that should not happen if the invariant is to hold.

**Definition 7.1 (Pierce)** In a given state, the pair of variables $(x, y)$ *pierces* balloon object $O$ if $O$ is reachable by $x$ and $y$ refers to an object internal to $O$.

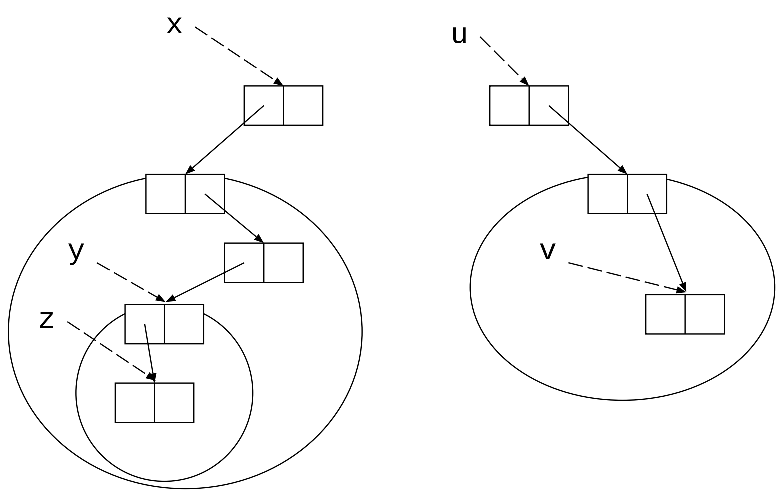


Figure 7.3: Piercing balloons

**Definition 7.2 (Pierce)** In a given state, the pair of variables $(x, y)$ *pierces* balloon class $C$ if $(x, y)$ pierces an object $O$ of class $C$.

Figure 7.3 illustrates this notion; supposing that all balloons present are from the same opaque balloon class $C$, the following pairs of variables pierce class $C$: $(x, y)$, $(x, z)$, $(y, z)$, and $(u, v)$.

We emphasise that for a pair $(x, y)$ to pierce class $C$, when $y$ references an object internal to $O$, the issue is not whether $x$ refers to an object non-internal to $O$: $x$ must reach $O$. As examples from the same figure, the following pairs do not pierce $C$: $(x, v)$, $(y, v)$, $(z, v)$, $(u, y)$, $(v, y)$, $(u, z)$, and $(v, z)$. The rationale for the definition is that, if in a given context $O$ cannot be reached, it is as if $O$ does not exist, and having a reference to an internal object should not be considered as piercing $O$.

The opaque balloon invariant will be defined in terms of the lack of piercing. It is, however, easy to see that what we have described informally as the property of opaque balloons, and which we will define formally via piercing, cannot apply to the code in the class which implements the opaque balloon type itself. The code for the rectangle type will be able to have rectangle variables and access their internal objects. This will happen at least for the `self` variable and for parameters of binary methods.

In defining the opaque balloon invariant it is essential, therefore, that a program is organised as a set of data types, and to distinguish the class implementing the opaque balloon type from the rest of the program. (This is in contrast with the plain balloon invariant, which is formulated only in terms of the object graph, regardless of whether the program is organised around data types or it is a flat set of functions, as in RISO.) We can now state the opaque balloon invariant.

**Definition 7.3 (Opaque Balloon Invariant)** If $C$ is a class implementing an opaque balloon type, in the execution of a method $m$ of any class other than $C$ no pair of variables in $m$ pierces class $C$.

## 7.3 Nesting Interpretation

### 7.3.1 Revising Cluster Relationships

In plain balloon checking, information is kept about whether some free cluster 'reaches' a given captured cluster; there is no need to track relationships between captured clusters, as non-balloons in such clusters can no longer be captured. It is enough to know that in a state $\boxed{x}\,\boxed{y}$, $x$ and $y$ *may* point to different captured clusters; there is no attempt to express whether they *definitely* point to different clusters, or to clusters unreachable from each other.

In the opaque balloon checking there is the need to establish the possibility of piercing involving pairs of variables. This requires information about cluster relationships even when only captured clusters are involved.

For example, in Figure 7.3, there is no piercing involving $(y, v)$, as they point to separated clusters. In some cases where free clusters are involved, this separation can be inferred from an abstract state. This case is abstracted as $\overline{\underline{x\boxed{yz}}}\;\overline{\underline{u\boxed{v}}}$, from which we know that $y$ and $v$ cannot point to the same cluster, and that the balloons of the corresponding clusters cannot be one internal to the other.

However, no such information exists when no free clusters are present. In the same example, if variables $x$ and $u$ are not considered, the corresponding abstract state will be $\boxed{yz}\,\boxed{v}$, where no relationship between captured clusters is represented. Therefore, the absence of piercing involving $(y, v)$ cannot be inferred.

In the checking mechanism for opaque balloon types we make a modification in the component $r$ of a state $(p, b, r)$, in order to obtain a refined set of states representing the extra relevant information. The basic idea is to allow captured clusters to be related; if they are not related it means they are definitely separated and no piercing occurs. Towards this we use modified versions of predicate $R$, abstraction and concretisation functions, and order from Section 5.3.2. Predicate $R$ becomes:

- $R : S_b \times I \times I \rightarrow \mathit{bool}$, 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 cluster $X$, there exists a balloon object $B$ referenced by an object in $X$ and variable $y$ references either $B$ or an object internal to $B$.

This modified predicate describes 'reaching' regardless of whether it is from a free or a captured cluster. Given the above definition, if we have $\neg R(s, x, y)$ then, in state $s$, the pair $(x, y)$ does not pierce any object; therefore, $x$ and $y$ can be used simultaneously as arguments to an arbitrary external function invoked from a method in an opaque balloon type.

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

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

Under the modified $R$, the component $r$ of an abstract state $(p, b, r)$ leads to a larger set of states. The conditions of well-formedness of $r$ are now:

$$\begin{aligned} &x\ p\ y \Rightarrow x\ r\ y, \\ &x\ r\ y \wedge by = 0 \Rightarrow x\ p\ y, \\ &x\ r\ y \wedge y\ r\ z \Rightarrow x\ r\ z, \end{aligned}$$

As before, it is a relation from clusters to clusters, but now captured clusters may also be related, which introduces more possibilities. Relation $r$ is transitively closed to mimic $R$. As a matter of convenience, we have made clusters related to themselves. Relation $r$ has now the form of a pre-order.

The new concretisation function is based on the previous one, becoming:

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

The last part in the concretisation function, $x \not{r}\, y \Rightarrow (x, y) \notin Ps$, states that if $x$ and $y$ are not related by $r$ (in which case we have also $x \not{p}\, y$), then they do not reference objects in the same cluster. (This is not covered by $p$ when $bx = by = 1$.)

The partial order on $C$ becomes:

$$\begin{aligned} (p_1, b_1, r_1) \sqsubseteq (p_2, b_2, r_2) \Leftrightarrow\ &(b_1 \sqsubseteq b_2) \wedge (r_1 \sqsubseteq r_2) \wedge \forall x, y \in I. \\ &(x\ p_1\ y \wedge x \not{p}_2\ y \Rightarrow b_2 x = b_2 y = 1) \\ &\wedge (x \not{p}_1\ y \wedge x\ p_2\ y \Rightarrow b_1 x + b_1 y \leq b_2 x). \end{aligned}$$

(Here $r_1 \sqsubseteq r_2$ is the usual $r_1 \subseteq r_2$.) Under the modified $r$ and $R$, and concretisation function and order, the properties $\{s\} \subseteq \gamma(\alpha s)$, $c_1 \sqsubseteq c_2 \Rightarrow \gamma c_1 \subseteq \gamma c_2$, and $s \in \gamma c \Rightarrow \alpha s \sqsubseteq c$ hold as before.

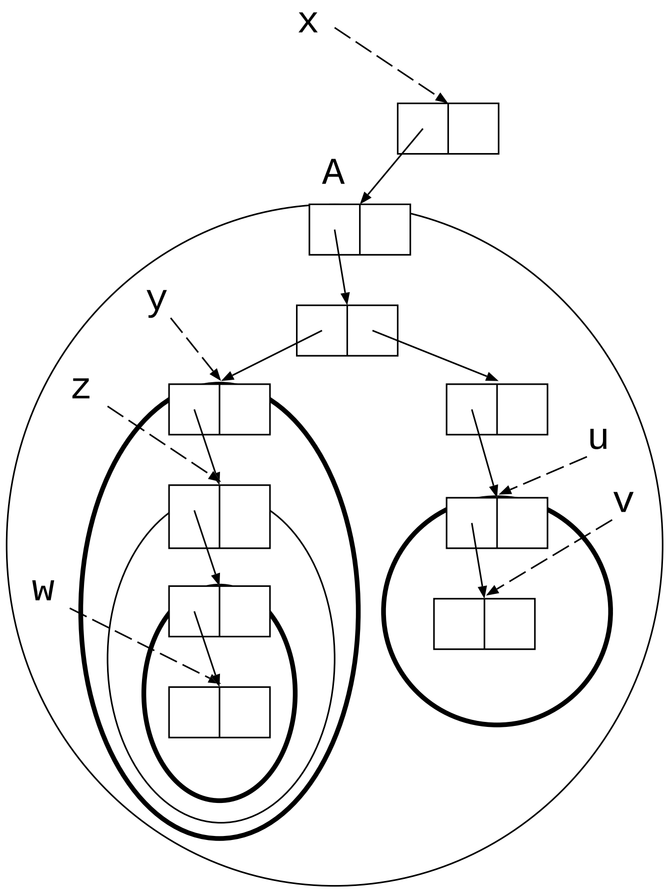


Figure 7.4: Levels in opaque balloons

## 7.3.2 Nesting Levels

With $r$ as defined above, given an abstract state with $x \not r\ y$, it is possible to perform an invocation of an external function '`f(x,y)`' from an opaque balloon class $C$; this because $(x, y)$ does not pierce any class, in particular $C$.

If $x\ r\ y$ we may have $R(s, x, y)$, but there are situations where $(x, y)$ does not pierce class $C$ and it is still valid to perform '`f(x,y)`'. Figure 7.4, presents a state $s$ during the execution of a method of opaque balloon class $C$, whose objects are represented with a thicker 'membrane'.

In the state shown, although we have $R(s, x, y)$ and that $(x, y)$ pierces object $A$, $(x, y)$ does not pierce class $C$. In other words, although a balloon membrane lies between $x$ and $y$, this membrane is not of an object of class $C$. This means that we could perform an invocation '`f(x,y)`' of a function external to $C$. On the contrary, a $C$ membrane lies between $x$ and $z$, the pair $(x, z)$ pierces class $C$, and it is not possible to perform '`f(x,z)`', as it would break the opaque balloon invariant.

Comparing these two cases, we can say that $x$ and $y$ are at the same *nesting level* in terms of class $C$, and that $z$ is one level inner. Nesting level, or simply *level*, turns out to be the relevant information in deciding whether or not to allow $x$ and $y$ to be used together as arguments invoking external code, when we have $x\ r\ y$.

This analysis suggests that we extend an abstract state with level information $l : I \to N$, associating each variable with a level (an integer). When abstracting a concrete state levels start from 0 in variables referencing free clusters, and increase by 1 for each entry into an opaque balloon of class $C$. Like the opaque balloon invariant, the checking mechanism considers each class individually, with no inter-class analysis (and interprocedural analysis at most within the class). This allows the level information to be relative to the class being checked, here denoted 'class $C$'; there is no need to tag a level with a class. In terms of the level information, the state in the same figure would be abstracted to:

$$l = \{x \mapsto 0, y \mapsto 0, z \mapsto 1, w \mapsto 2, u \mapsto 0, v \mapsto 1\}.$$

In each equivalence class in $p$, all variables not of class $C$ have the same level, which is one more than the level of variables of class $C$. This constrains $l$, making it essentially a function from equivalence classes to levels.

Level information is also useful when $\neg R(s, x, y)$ and $\neg R(s, y, x)$ but, due to a conservative calculation of $r$, we may have $x\ r\ y$ or $y\ r\ x$. If no level information existed, $x$ and $y$ would have to be prevented from being used together as arguments to external functions. The level information makes such invocations acceptable if $x$ and $y$ are at the same level. This could be the case checking '`f(y,u)`' (in the same figure) and $y\ r\ u$ due to a conservative calculation of $r$. (Although the calculation of levels may be itself conservative, as we discuss below.)

By adding levels to abstract states we obtain tuples $(p, b, r, l)$. While before there were 'few' possible states and we managed to use a compact graphical notation, that is no longer possible. We can, however, use a graphical notation which, although less compact, is still useful to represent states in an intuitive fashion. A state is represented as follows: $r$ being a pre-order, we represent it by drawing the corresponding diagram; each node in the diagram corresponds to an equivalence class in $p$; $b$ is represented using the old notation; $l$ is represented by attaching a number to each node (which is the smallest of the two levels in case the node contains both variables of the self class and of other types). Figure 7.5 presents

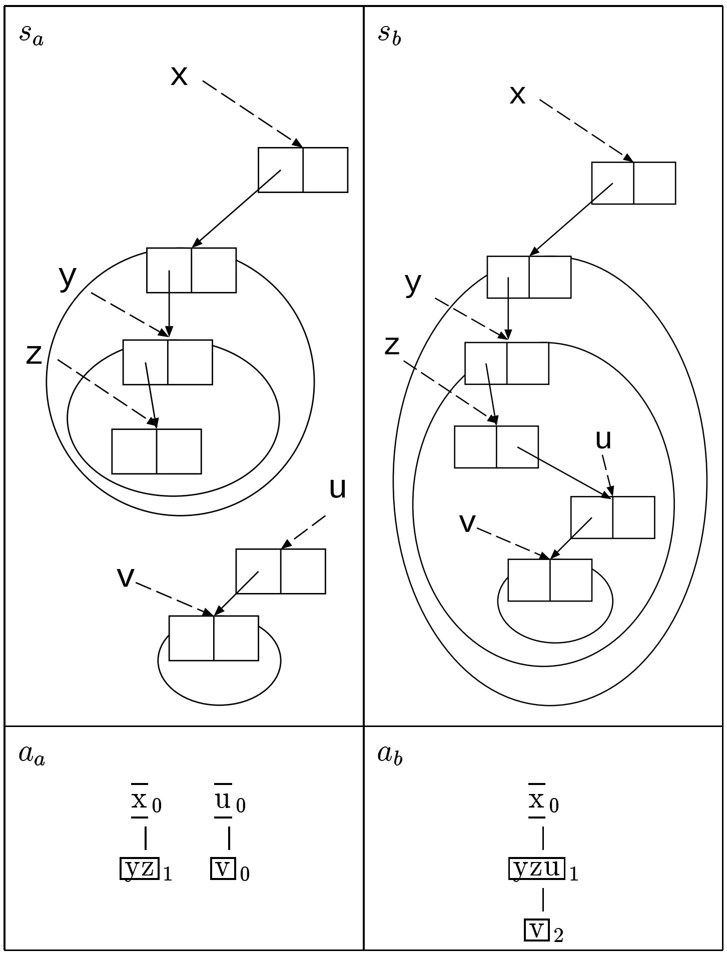


Figure 7.5: Two concrete states and corresponding abstract states

two concrete states and corresponding states into which they are abstracted.

The order on the abstract states is now:

$$\begin{aligned}(p_1, b_1, r_1, l_1) \sqsubseteq (p_2, b_2, r_2, l_2) &\Leftrightarrow (p_1, b_1, r_1) \sqsubseteq (p_2, b_2, r_2)\\ &\wedge \forall x, y.\ x\ r_1\ y \Rightarrow l_1 x - l_1 y = l_2 x - l_2 y.\end{aligned}$$

This order takes into account both the $(p, b, r)$ and the level information. Levels of two given identifiers only matter for concrete states where those identifiers are related via $R$. In an abstract state where $x \not r\ y$, we have $\neg R(s, x, y)$ in the corresponding concrete states, and the relative level between $x$ and $y$ in the abstract state is not relevant; this is what happens for $z$ and $u$ in state $a_a$ in Figure 7.5.

Under this order we have, correctly, $a_a \sqsubseteq a_b$ for the abstract states in the same figure. Indeed, $a_b$ also represents the concrete state $s_a$; in $a_b$ we have $z$ and $u$

in the same equivalence class in $p$, which means that they are either in the same cluster (as in $s_b$) or in different clusters not related via $R$ (as in $s_a$).

As only the relative level between identifiers related through $r$ matters, we need to impose a well-formedness condition to abstract states in order to prevent different abstract states from representing the same information. If the following two states were allowed:

$$a_1 = \begin{tikzcd} \boxed{a}_2 \arrow[d, dash] & \boxed{c}_3 \arrow[d, dash] & \boxed{e}_1 \arrow[d, dash] \\ \boxed{b}_3 & \boxed{d}_4 \arrow[ur, dash] & \boxed{f}_2 \end{tikzcd} \qquad a_2 = \begin{tikzcd} \boxed{a}_1 \arrow[d, dash] & \boxed{c}_4 \arrow[d, dash] & \boxed{e}_2 \arrow[d, dash] \\ \boxed{b}_2 & \boxed{d}_5 \arrow[ur, dash] & \boxed{f}_3 \end{tikzcd}$$

we would have $a_1 \sqsubseteq a_2$ and $a_2 \sqsubseteq a_1$; they would represent the same set of concrete states. This means that we would have, not a partial order, but a pre-order of abstract states. To avoid the above situation we need to normalise the level information in abstract states. This is done by having the condition that in each connected diagram in $r$ there is at least one identifier with level zero. That is, given an abstract state $(p, b, r, l)$, we have:

$$\forall x \in I.\, \exists y \in (r \cup r^{op})^{+}\{x\}.\;\; ly = 0.$$

The valid abstract state corresponding to $a_1$ and $a_2$ is:

$$\begin{tikzcd} \boxed{a}_0 \arrow[d, dash] & \boxed{c}_2 \arrow[d, dash] & \boxed{e}_0 \arrow[d, dash] \\ \boxed{b}_1 & \boxed{d}_3 \arrow[ur, dash] & \boxed{f}_1 \end{tikzcd}$$

### 7.3.3 Equivalence Classes of Levels

As happens in general when abstracting information, levels cannot be represented exactly; abstract states must represent a set of possibilities. A variable may present different levels at a given point, due to branches in the control path. A typical example is a loop where a variable descends an arbitrary number of steps into a recursive data structure (of opaque balloons of the class being checked) as in:

```
while (...)
  x :- x.a;
  ...
```

As there may be no bound on the number of iterations, after the loop a state must represent, not a single level, but a range of levels for variable $x$.

When the level of a variable $x$ varies as above, independently from some other variable $y$, it is not essential to quantify the possible variation; it is enough to express that the levels of $x$ and $y$ fluctuate independently. In this case, if variables $x$ and $y$ are related by $r$, they must be prevented from being used together as arguments of external functions, regardless of the amount of variation.

On the other hand, the level of several variables may vary in a related fashion. As an example, consider variables $x$, $y$, and $z$, with known levels, $x$ and $y$ traversing together a given data structure in a loop like:

```
while (...)
  x :- x.a;
  y :- y.a;
  ...
f(x,y);
```

Even if the number of iterations is unknown, after the loop the relative levels of $x$ and $y$ remain as before (although we cannot compare the levels of $x$ or $y$ with $z$). If prior to the loop the levels of $x$ and $y$ are the same, they will remain the same during and after the loop; this allows an invocation with these variables as arguments to be accepted. Knowing the relative levels of $x$ and $y$ is, precisely, what matters in deciding whether to accept an invocation '`f(x,y)`' of an external function, when the variables are related by $r$. (The kind of analysis where one aims to track pointer following in 'lockstep' has been addressed in [29, 30]. In our case, we have an analysis tailored to our needs and assumptions, where we follow 'opaque balloon entering' as opposed to individual pointer derreferencing.)

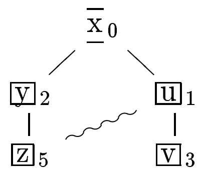
From this discussion, the level information is represented by elements $(l, s)$ of a domain $L$, where:

- $l : I \to N$ is a function mapping each variable to a level, as before;
- $s$: $\mathcal{P}(I \times I)$, is a relation defining equivalence classes of variables whose levels may vary simultaneously starting from the value given by $l$, or expressing that a variable has a fixed level.

An abstract state becomes a tuple $(p, b, r, l, s) \in C \times L$. Variables with a 'fixed' level do not appear in $s$, while a set of variables whose levels vary simultaneously forms an equivalence class: $x\ s\ y$ means that the possible levels for $x$ and $y$ are $lx + n$ and $ly + n$ respectively, for $n \in \{0, 1, \ldots, \omega\}$. With the exception of the variables with fixed level, $s$ is an equivalence relation.

Each equivalence class $i$ in $s$ gives rise to a $n_i$, which represents an independent variation. A state $(p, b, r, l, s)$ with $k$ equivalence classes in $s$ generates a k-dimensional space of abstract states $(p, b, r, l)$; it represents the union of the sets of concrete states associated with the points in the space defined.

The graphical notation for abstract states is extended by connecting nodes in each equivalence class in $s$ by a curly line. As an example, a state $(p, b, r, l, s)$ with component $(p, b) = \underline{\overline{\mathrm{x}}}\,\boxed{\mathrm{y}}\,\boxed{\mathrm{z}}\,\boxed{\mathrm{u}}\,\boxed{\mathrm{v}}$, component $r = \{(x, y), (y, z), (x, u), (u, v)\}^*$, component $l = \{x \mapsto 0, y \mapsto 2, z \mapsto 5, u \mapsto 1, v \mapsto 3\}$, and $s = \{(u, z), (z, u)\}^*$ can be represented as follows:



In addition to the normalisation of levels in each connected diagram, as we have discussed, for abstract states $(p, b, r, l, s)$ we need to impose another condition of well-formedness to prevent several abstract states from representing the same information; the reason is the same as before. Now, if all identifiers in a $r$ connected diagram are in the same $s$ equivalence class, this represents the same situation as if all of them are 'fixed'; this is because all generated points represent the same normalised $(p, b, r, l)$. In this case, the state is made well-formed by making all of them fixed. This new condition of well-formedness is:

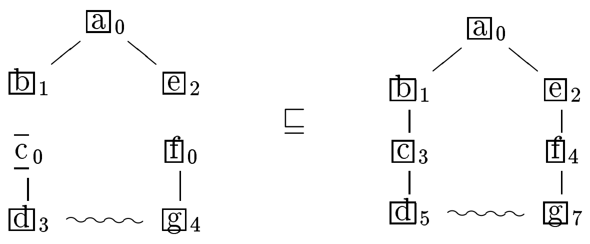
$$\forall x \in I.\, \exists y, z \in (r \cup r^{op})^{+}\{x\}.\ y \not{s}\, z.$$

As an abstract state represents a set of the 'old' (4-tuple) states, we have $a_1 \sqsubseteq a_2$ when, in addition to being ordered in the $(p, b, r)$ component, they respect:

- if the level of an identifier varies in $a_1$, then it varies in $a_2$;
- if the level of an identifier varies in $a_2$, then it must start from a value less or equal than in $a_1$;
- if the levels of two identifiers vary together in $a_2$, then either they vary together or they are fixed in $a_1$;
- if the levels of two identifiers vary together or are both fixed in $a_2$, then their relative level in $a_2$ is the same as in $a_1$.

Levels in $a_1$ and $a_2$ cannot be compared directly; we must offset one of them. This is because, as we have discussed, levels only matter for identifiers related through $r$; each connected diagram $D$ in $a_1$, which is normalised so that levels start at 0 in some identifier $x$, may be part of a larger connected diagram in $a_2$ where $l_2 x > 0$. To compare levels between $a_1$ and $a_2$, we offset the level of identifiers in $D$ by $l_2 x$.

An example of a pair of comparable abstract states is:



The order on abstract states which we have described can be written as:

**Definition 7.4**

$$\begin{aligned}
&(p_1, b_1, r_1, l_1, s_1) \sqsubseteq (p_2, b_2, r_2, l_2, s_2) \Leftrightarrow \\
&\quad (p_1, b_1, r_1) \sqsubseteq (p_2, b_2, r_2) \wedge \forall x, y.\ (x\ s_1\ x \Rightarrow x\ s_2\ x) \\
&\quad \wedge (x\ s_2\ x \Rightarrow l_2 x \leq l_1 x + l_2 \underset{r_1,l_1}{\triangle} x) \wedge (x\ s_2\ y \Rightarrow x\ s_1\ y \vee x \not s_1\ x) \\
&\quad \wedge (x\ s_2\ y \vee (x \not s_2\ x \wedge y \not s_2\ y) \Rightarrow (l_1 x + l_2 \underset{r_1,l_1}{\triangle} x) - (l_1 y + l_2 \underset{r_1,l_1}{\triangle} y) = l_2 x - l_2 y),
\end{aligned}$$

where

$$\underset{r,l}{\triangle} z = \bigsqcup\{w \mid (w, z) \in (r \cup r^{op})^+ \wedge lw = 0\}.$$

Here, $\underset{r,l}{\triangle} z$ gives an identifier with level 0 (in $l$) in the connected diagram (defined by $r$) where $z$ lies. The join is simply a way to choose one identifier if several candidates exist; in this we assume a total order on the set of identifiers.

## 7.3.4 Abstract Semantics

### Atomic Commands

We now define the semantic function for assignments. For a state $(p, b, r, l, s)$ we need only consider components $r$, $l$, and $s$; components $p$ and $b$ of a resulting state remain as before (presented in Figure 5.10). The semantic functions are presented in Figure 7.6 (assignments to variables) and Figure 7.7 (assignments to state variables and integer objects). Function norm performs both normalisations that we have discussed towards making a state well-formed.

While most cases are relatively straightforward, some deserve special attention, in particular $\mathcal{A}^a[\![x :-\ y.\,z]\!]$ and $\mathcal{A}^a[\![x.\,z :-\ y]\!]$.

$$
\begin{aligned}
\mathcal{A}^a[\![x :- y]\!] \quad &= \quad \lambda(p,b,r,l,s).\,\mathrm{norm}(p',b',r',l',s'),\ \text{ where}\\
r' &= \begin{cases} r & \text{if } x = y,\\ (r \mathbin{\overline{|}} \{x\} \cup \{(x,y),(y,x)\})^+ & \text{otherwise.} \end{cases}\\
l' &= l[x \mapsto ly]\\
s' &= \begin{cases} s & \text{if } x = y,\\ s \mathbin{\overline{|}} \{x\} & \text{if } x \neq y \wedge y \mathrel{\not{s}} y,\\ (s \mathbin{\overline{|}} \{x\} \cup \{(x,y),(y,x)\})^+ & \text{otherwise.} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{A}^a[\![x :- y.\,z]\!] \quad &= \quad \lambda(p,b,r,l,s).\,\mathrm{norm}(p',b',r',l',s'),\ \text{ where}\\
r' &= \begin{cases} r & \text{if } \neg\,\mathrm{balloon}\,x \wedge x = y,\\ (r \mathbin{\overline{|}} \{x\} \cup \{(x,y),(y,x)\})^+ & \text{if } \neg\,\mathrm{balloon}\,x \wedge x \neq y,\\ r \mathbin{\overline{|}} \{x\} \cup \{(x,x)\} \cup \{(x,w) \mid y\ r\ w & \\ \quad \wedge \neg(\mathrm{self}\,y \wedge (w \mathrel{\not{s}} w \vee w\ s\ y) \wedge lw \leq ly)\} & \text{otherwise.}\\ \cup \{(w,x) \mid w\ r\ y \vee (y\ r\ w \wedge ((y\ s\ y \wedge y \mathrel{\not{s}} w) & \\ \quad \vee (\mathrm{self}\,y \wedge lw \leq ly{+}1) \vee (\neg\,\mathrm{self}\,y \wedge lw \leq ly)))\} & \end{cases}\\
l' &= \begin{cases} l[x \mapsto ly + 1] & \text{if } \mathrm{self}\,y,\\ l[x \mapsto ly] & \text{otherwise.} \end{cases}\\
s' &= \begin{cases} s & \text{if } x = y,\\ s \mathbin{\overline{|}} \{x\} & \text{if } x \neq y \wedge y \mathrel{\not{s}} y,\\ (s \mathbin{\overline{|}} \{x\} \cup \{(x,y),(y,x)\})^+ & \text{otherwise.} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{A}^a[\![x :- \texttt{null}]\!] \quad &= \quad \lambda(p,b,r,l,s).\,\mathrm{norm}(p',b',r',l',s'),\ \text{ where}\\
r' &= r \mathbin{\overline{|}} \{x\} \cup \{(x,x)\}\\
l' &= l[x \mapsto 0]\\
s' &= s \mathbin{\overline{|}} \{x\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{A}^a[\![x :- \texttt{new}]\!] \quad &= \quad \lambda(p,b,r,l,s).\,\mathrm{norm}(p',b',r',l',s'),\ \text{ where}\\
r' &= r \mathbin{\overline{|}} \{x\} \cup \{(x,x)\}\\
l' &= l[x \mapsto 0]\\
s' &= s \mathbin{\overline{|}} \{x\}
\end{aligned}
$$

Figure 7.6: Abstract semantics for assignments

$$
\begin{array}{lcl}
\mathcal{A}^a[\![x.\,z :- y]\!] & = & \lambda(p,b,r,l,s).\begin{cases} \top & \text{if balloon}\, y, \\ \top & \text{if } x \not{p}\, y \wedge bx = by = 1, \\ \top & \text{if } x \not{p}\, y \wedge (x\ r\ y \vee y\ r\ x), \\ \mathrm{norm}(p',b',r',l',s') & \text{otherwise, where} \end{cases}
\end{array}
$$

$$
r' = (r \cup \{(x,y),(y,x)\})^+
$$

$$
l' = \begin{cases} l & \text{if } bx = by, \\ l[w \mapsto lw + lx + 1 \mid y\ r\ w] & \text{if } bx = 1 \wedge by = 0 \wedge \text{self}\, x, \\ l[w \mapsto lw + lx \mid y\ r\ w] & \text{if } bx = 1 \wedge by = 0 \wedge \neg\,\text{self}\, x, \\ l[w \mapsto lw + ly \mid x\ r\ w] & \text{otherwise.} \end{cases}
$$

$$
s' = \begin{cases} s & \text{if } bx = by, \\ s & \text{if } bx = 1 \wedge by = 0 \wedge x \not{s}\, x, \\ (s \setminus \{(u,v),(v,u) \mid y\ r\ u \wedge y \not{r}\, v\} & \\ \quad \cup \{(x,u),(u,x) \mid y\ r\ u \wedge u \not{s}\, u\})^+ & \text{if } bx = 1 \wedge by = 0 \wedge x\ s\ x, \\ s & \text{if } by = 1 \wedge bx = 0 \wedge y \not{s}\, y, \\ (s \setminus \{(u,v),(v,u) \mid x\ r\ u \wedge x \not{r}\, v\} & \\ \quad \cup \{(y,u),(u,y) \mid x\ r\ u \wedge u \not{s}\, u\})^+ & \text{if } by = 1 \wedge bx = 0 \wedge y\ s\ y, \end{cases}
$$

$$
\begin{array}{lcl}
\mathcal{A}^a[\![x.\,y :- \texttt{null}]\!] & = & \lambda c.\, c \\
\mathcal{A}^a[\![x.\,y :- \texttt{new}]\!] & = & \lambda c.\, c \\
\mathcal{A}^a[\![x <- e]\!] & = & \lambda c.\, c
\end{array}
$$

Figure 7.7: Abstract semantics for assignments (cont.)

- In $\mathcal{A}^a[\![x :- y.\,z]\!]$, when $x$ is of balloon type we recall that $x$ moves to an equivalence class of its own in $p$. To calculate which identifiers $x$ becomes related to (via $r$) in the more precise way under the information available we use, not only $r$ itself, but also the level information. In particular, when $y$ is of the opaque balloon type being checked, $x$ increases one level, and $x$ need not be related to identifiers $w$ in levels up to $y$, as $R(s,x,w)$ will be false.

  Identifiers related to $y$ become related to $x$, as well as those identifiers $w$ to which $y$ relates and, according to the level information, it may be the case that $R(s,w,x)$ for one of the possible concrete resulting states.

- In $\mathcal{A}^a[\![x.\,z :- y]\!]$, when $x$ is in a captured cluster and $y$ in a free cluster (or vice-versa), as the clusters are merged, we need to offset the levels of the group of identifiers to which $y$ relates to make them start at the level of $x$ (or one above).

Also, if in this case $x$ has a fluctuating level, we must update the equivalence classes of levels for the group of identifiers to which $y$ relates, so that the ones which were fixed become fluctuating together with $x$, and the ones which were fluctuating become disconnected from any other identifier outside that group.

### Composite Commands

The poset of base abstract states with elements $(p, b, r, l, s)$ and order as in Definition 7.4 now has infinite width (there is an infinite number of incomparable elements). This means that we cannot use for composite commands the completion by down-sets as we have done for plain balloon types, because it would result in a lattice of infinite height.

However, we have designed the level representation so that the join exists for elements that are comparable in the $(p, b, r)$ component. Given two elements, $(p_1, b_1, r_1, l_1, s_1)$ and $(p_2, b_2, r_2, l_2, s_2)$, with $(p_1, b_1, r_1) \sqsubseteq (p_2, b_2, r_2)$, their join exists and is given by $(p_1, b_1, r_1, l_1, s_1) \sqcup (p_2, b_2, r_2, l_2, s_2) = (p_2, b_2, r_2, l, s)$, where:

$$\begin{array}{rcl}
l & = & \{x \mapsto (l_1 x + l_2 \underset{r_1,l_1}{\triangle} x) \sqcap l_2 x \mid x \in I\}, \\
s & = & \{(x, y) \mid (l_1 x + l_2 \underset{r_1,l_1}{\triangle} x) - (l_1 y + l_2 \underset{r_1,l_1}{\triangle} y) = l_2 x - l_2 y \\
 & & \quad \wedge\ (x\ s_1\ y \vee (x \not s_1\ x \wedge y \not s_1\ y)) \wedge (x\ s_2\ y \vee (x \not s_2\ x \wedge y \not s_2\ y)) \\
 & & \quad\quad \wedge\ (x \not s_1\ x \wedge x \not s_2\ x \Rightarrow l_1 x + l_2 \underset{r_1,l_1}{\triangle} x \neq l_2 x)\}.
\end{array}$$

The join of two such elements results in the greatest $(p, b, r)$ component and a $(l, s)$ component that covers both states:

- $l$ maps each identifier to the minimum of the two corresponding levels in the operands (with $l_1$ adjusted);
- two identifiers are put in the same equivalence class in $s$ when (after adjusting $l_1$): they have the same relative levels in both operands; they are, for each operand, either fixed or varying simultaneously; and each has a different level according to the operand when they are fixed in both operands.

To obtain states for composite commands we use sets of states $(p, b, r, l, s)$ that are incomparable even considering only the $(p, b, r)$ component; this is possible by making use of the available joins for states that are comparable in $(p, b, r)$. The domain for composite commands is now:

$$\begin{aligned} D = \{X \subseteq C \times L \mid\ & (p_1, b_1, r_1, l_1, s_1), (p_2, b_2, r_2, l_2, s_2) \in X \\ & \Rightarrow (p_1, b_1, r_1) \not\sqsubseteq (p_2, b_2, r_2)\}_\top, \end{aligned}$$

This way, an abstract state is a set with at most as many elements as the width of $C$. The order on $D$ is:

$$\begin{aligned} X \sqsubseteq Y \Leftrightarrow Y = \top \vee ((p, b, r, l, s) \in X \Rightarrow\ & \exists (p', b', r', l', s') \in Y. \\ & (p, b, r, l, s) \sqsubseteq (p', b', r', l', s')), \end{aligned}$$

It is desirable that $D$ is a join-semilattice, as the semantics for conditionals and loops is expressed using joins. Unfortunately, joins in $D$ do not exist in general. This can be illustrated by the following example. Given $X = \{u, v\}$ and $Y = \{w\}$, with $(p_w, b_w, r_w) \sqsubseteq (p_u, b_u, r_u)$, $(p_w, b_w, r_w) \sqsubseteq (p_v, b_v, r_v)$, $w \not\sqsubseteq u$, $w \not\sqsubseteq v$, we can have two minimal upper bounds of $\{X, Y\}$, $U_1 = \{u \sqcup w, v\}$ and $U_2 = \{u, v \sqcup w\}$, but no least upper bound. Also, there is no reason to choose one over the other, if some upper bound—if not the least—is to be kept.

We can, however, define an operation $(\check{\sqcup})$ to 'join' two states $X, Y$, obtaining an upper bound which, although being more conservative, does not require making an arbitrary choice as in the example above. $X \check{\sqcup} Y$ results in $\top$ if either operand is $\top$; or forms the union $X \cup Y$ of the operands and approximates it by an element of $D$ through a function $\lceil - \rceil$:

$$X \check{\sqcup} Y \quad = \quad \begin{cases} \top & \text{if } X = \top \vee Y = \top, \\ \lceil X \cup Y \rceil & \text{otherwise.} \end{cases}$$

$\lceil X \rceil$ results in a set whose elements are, for each maximal element in $X$ considering only components $(p, b, r)$, the join over the corresponding principal ideal:

$$\begin{aligned} \lceil X \rceil \quad = \quad & \{\bigsqcup\{(p, b, r, l, s) \in X \mid (p, b, r) \sqsubseteq (p_m, b_m, r_m)\} \\ & \qquad \mid (p_m, b_m, r_m) \in \mathrm{Max}\{(p, b, r) \mid (p, b, r, l, s) \in X\}\}, \end{aligned}$$

The abstract semantics for composite commands are almost independent of the particular base states, and remain as before in most cases (as in Figure 5.11), with $\check{\sqcup}$ being used instead of $\sqcup$ for conditionals and loops. In what concerns the assignment (as a composite command), the semantics is now defined making use of $\lceil - \rceil$:

$$
\begin{array}{lcl}
\mathcal{C}^a[\![a]\!] & = & \lambda\varphi.\,\lambda o. \begin{cases} \top & \text{if } \top \in A, \\ \lceil A \rceil & \text{otherwise.} \end{cases} \\
\text{where} & & \\
A & = & \begin{cases} \{\top\} & \text{if } o = \top, \\ \{\mathcal{A}^a[\![a]\!]c \mid c \in o\} & \text{otherwise.} \end{cases}
\end{array}
$$

## 7.4 Type-checking Approaches

The nesting interpretation forms the basis of opaque balloon checking. We have not, however, described any mechanism to actually decide on the acceptance of code. For plain balloon types we have presented an interpretation which decides on the acceptance of a whole program, having postponed modularity issues to Chapter 8. In the case of opaque balloon types, not only is it essential for the invariant itself to assume that a program is organised as a set of data types, but we assume from the start that the checking mechanism will consider each data type individually. A global program analysis, which has been considered (at least in theory) for plain balloon types, is here discarded from the start.

The opaque balloon invariant states that, while executing code external to an opaque balloon class $C$, no pair of variables pierces class $C$. A checking mechanism to enforce the invariant must verify that:

- In the body of methods of each opaque balloon class $C$, when an invocation of external code is made, no piercing relative to $C$ involving arguments exists. If at the point where the invocation occurs we have an abstract state $X$ as described in the nesting interpretation, we can check that no piercing involving arguments exists by verifying that, for each $(p, b, r, l, s)$ in $X$, each pair of arguments to the invocation is:
  - either unrelated via $r$, or
  - at the same level in $l$ and either in the same equivalence class in $s$ or fixed.

It is important to mention that the absence of piercing can be counted upon in some situations with no need for the nesting interpretation. This is the case when only references to opaque balloon objects of other classes are involved. The reason is that the class being checked will be in this case 'external code' with respect to these classes, which means that we can count on the invariant itself to know that none of these balloons can be internal to each other. This situation is bound to occur frequently, namely when using primitive types such as integer or real.

- Methods of each opaque balloon class $C$ do not return references to an object internal to any object of class $C$ manipulated. Again, this can be done checking that no references are returned in identifiers that have a level greater than 0 or in some equivalence class in $s$.

A general solution to checking opaque balloon types would be to extend the nesting interpretation to consider function invocation. The extended interpretation would make the verifications above, resulting in $\top$ in the cases leading to the invariant being broken, or computing the effect of the invocation on the calling abstract state otherwise. However, the base domain in the nesting interpretation is already quite complex; we have doubts whether it would be realistic to extend the nesting interpretation to function invocation (with the corresponding problems of representing functions in an efficient way, and having to calculate fixed points due to recursive functions) and to base the checking mechanism on it. Therefore, we leave it to be considered as a possibility for future research.

A simple alternative is to use a more conservative way of computing the effect of an invocation:

- making use of the representation of functions in the plain balloon checking mechanism in order to calculate the components $p$, $b$ and $r$;

- using the knowledge that, in public methods or external functions, parameters and result cannot be related at different levels; in particular, the returned reference must have the same level as a parameter from which it is reached; assuming, otherwise, a worse case approximation to calculate the level component.

We now outline two approaches to check opaque balloon types; in both of them each opaque balloon class is considered in isolation. The first approach, more conservative, considers each method individually; the second approach checks the class as a whole.

## 7.4.1 Extending Absence of Piercing to Every Invocation

When a function in (the interface of) an opaque balloon type is invoked from external code, assuming the invariant holds, we can count on the absence of piercing involving parameters to the function. This means that, given parameters $x$, $y$, and $z$, we can represent the situation on entry by the state

$$\boxed{X}_0 - \boxed{Y}_0 - \boxed{Z}_0$$

and analyse the body, making the verifications described above when an invocation of an external function or the end of the body is encountered.

There is, however, one problem: this absence of piercing between parameters will not necessarily hold in methods of class $C$ when they are invoked from within $C$. This is what happens, in particular, when the class has private methods; these are not in the interface of the opaque balloon type, being only invoked from within the class. No assumption can be made while checking their bodies unless some constraint is imposed when they are invoked.

Being conservative, we can devise a way to do checking, considering each method in isolation. The idea is to extend the absence of piercing to every method invocation, using it to perform an inductive step of assuring that it will hold for an invocation performed within a method $m$, assuming that it held on entry to $m$. More precisely:

- a method is checked under the assumption that no piercing involving parameters exists upon entry, represented by the abstract state $\boxed{X}_0 - \boxed{Y}_0 - \boxed{Z}_0$ for parameters $x$, $y$, and $z$, as we have mentioned;
- when analysing the body, the absence of piercing between arguments to an invocation is checked for, not only for external functions, but for every invocation (including the own methods, public or private, of the class being checked).

Although this may seem arbitrarily conservative, it is a natural extension of what already happens in external code. Under this constraint, although piercing may exist in opaque balloon code between local variables or between these and parameters, its absence on entry to a method provides support to reason about each method in isolation, leading to more disciplined interactions between the methods that implement an opaque balloon type. If experimental evidence does not show it to be overly conservative, this can be an interesting possibility for opaque balloon checking.

### 7.4.2 Nesting Constraint Propagation

A less conservative checking mechanism can be obtained, without relying on a full interprocedural nesting interpretation within the class, using only the simple approximation described above in calculating the effect of an invocation.

Contrary to the approach in the previous section, here we do not impose the absence of piercing between parameters. This implies, however, that we cannot count on it, which means that the mechanism must be more complex than the previous one.

There are situations where piercing between parameters of a method of the opaque balloon class being checked will be irrelevant and can be allowed; this is the case if:

- the method invokes external code with all arguments being references to opaque balloon objects of other classes, or

- the method invokes other methods in the class, for which the absence of piercing between parameters is irrelevant (for these two reasons).

An example is shown in figure 7.8, where all the methods of class $O$ may be invoked with no restrictions regarding piercing between parameters.

On the other hand, a method that contains invocations of external code involving arguments of non-opaque types may be unacceptable, regardless of how it is invoked, or acceptable but only if invoked with parameters complying with some *constraints* on their relative level, when they are related.

As an example, in Figure 7.9 the private method `aux` invokes an external function with two arguments. An invocation of `aux` where the parameters (`self` and `other`) are related (through $r$) is only valid if the level of `other` is one greater than the level of `self`. This constraint can be seen to be satisfied by the invocation in the body of method `m`; this is because `m` is only invoked from external code and, therefore, we can count on its parameters being either unrelated or at the same level.

```
opaque balloon O
{
private:
  val:Int;
  nxt:O;
  sum(other:O):Int
  {
    Int n := self.val + other.val;
    return n;
  }
  sqr_sum(other:O):Int
  {
    Int n := self.sum(other);
    return n * n;
  }
public:
  m(other:O)
  {
    Int x,y,z;
    x := self.sum(other);
    y := self.sqr_sum(other.nxt);
    z := self.sqr_sum(self.nxt);
  }
}
```

Figure 7.8: Example where piercing between parameters is irrelevant

In the same figure, method `err` invokes the external function using both `other` and `other.nxt` as arguments, causing piercing between them in external code, regardless of how it is invoked. This means that it must be flagged as invalid by the checking mechanism.

To illustrate the way the checking mechanism can generate constraints on the invocation of a method or flag it as invalid, we now consider the simple case of a method `met`, containing an invocation of external code:

```
met(x,y)
{
  ...
  ext_fun(u,v);
  ...
}
```

A nesting interpretation is performed, with the parameters of `met` unrelated in the

```
opaque balloon O
{
private:
  val:Int;
  nxt:O;
  aux(other:O)
  {
    ext_fun(self.nxt, other);
  }
  err(other:O)
  {
    ext_fun(other.nxt, other);
  }
public:
  m(other:O)
  {
    o1,o2:O;
    o1 :- self;
    o2 :- other.nxt;
    while (...) do
      o1.aux(o2);
      o1 :- o1.nxt;
      o2 :- o2.nxt;
      ...
  }
}
```

Figure 7.9: Constraints in the relative levels of parameters

initial state; i.e.

$$\boxed{x}_0 \quad \boxed{y}_0$$

The state at the invocation point is then looked at, and one of the following may happen:

- `met` is valid regardless of how it is invoked, if in the elements in the state at the invocation point, `u` and `v` are related to by at most one of the parameters, and are unrelated or at the same relative level; examples are:

$$\begin{tikzcd}\boxed{x}_0 \arrow[d, dash] & \boxed{y}_0 & \boxed{v}_0 \\ \boxed{u}_2 & & \end{tikzcd} \quad \text{and} \quad \begin{tikzcd}\boxed{x}_0 & \boxed{y}_0 \arrow[d, dash] \arrow[dr, dash] & \\ & \boxed{u}_2 \arrow[r, dash] & \boxed{v}_2\end{tikzcd}$$

- `met` is invalid regardless of how it is invoked, if in some element in the state at the invocation point, `u` and `v` are related to by at most one of the parameters, and are related at different levels; an example is:

$$\begin{tikzcd}[row sep=small] \boxed{\mathtt{x}}_0 & \boxed{\mathtt{y}}_0 \arrow[d, dash] \\ & \boxed{\mathtt{u}}_2 \arrow[d, dash] \\ & \boxed{\mathtt{v}}_3 \end{tikzcd}$$

- `met` is acceptable subject to constraints on its invocation, if in some element in the state at the invocation point, `u` and `v` are related to by different parameters, and are both fixed or in the same equivalence class in $s$; an example is:

  $$\begin{tikzcd}[row sep=small] \boxed{\mathtt{x}}_0 \arrow[d, dash] & \boxed{\mathtt{y}}_0 \arrow[d, dash] \\ \boxed{\mathtt{u}}_2 \arrow[r, squiggly, dash] & \boxed{\mathtt{v}}_3 \end{tikzcd}$$

  This state would generate the constraint regarding parameters of `met`:

  $$x\ r\ y \vee y\ r\ x \Rightarrow lx = ly + 1 \wedge (x\ s\ y \vee (x \not s\ x \wedge y \not s\ y))$$

  This constraint means that it is acceptable to perform an invocation '`met(a,b)`' with calling states such as:

  $$\boxed{\mathtt{a}}_0 \quad \boxed{\mathtt{b}}_0 \qquad \text{or} \qquad \begin{tikzcd}[row sep=small] \boxed{\mathtt{b}}_0 \arrow[d, dash] \\ \boxed{\mathtt{a}}_1 \end{tikzcd}$$

  but it is unacceptable to perform the same invocation with calling states such as:

  $$\boxed{\mathtt{a}}_0 - \boxed{\mathtt{b}}_0 \qquad \text{or} \qquad \begin{tikzcd}[row sep=small] \boxed{\mathtt{b}}_0 \arrow[d, dash] \\ \boxed{\mathtt{a}}_2 \end{tikzcd} \qquad \text{or} \qquad \begin{tikzcd}[row sep=small] \boxed{\mathtt{a}}_0 \arrow[d, dash] \\ \boxed{\mathtt{b}}_1 \end{tikzcd}$$

In the general case where a method may invoke other methods for which constraints may exist, or if the methods are mutually recursive, the same approach to generating constraints applies:

- When an invocation of a method with constraints is encountered, the procedure for generating constraints is the same as for an invocation of external code. An external invocation is just a special case of a constrained method having the constraint 'if the parameters are related they must be at the same level'.

- In the case of mutual recursion we have cyclic dependencies. We start by assuming no constraints in methods; then we do an iterative process where in each iteration we analyse the methods under the current constraints, generating new constraints; we iterate until either a fixed point is reached or inconsistent constraints are generated leading to 'invalid'.

After the constraints are generated for all methods in the class, if the 'invalid' outcome did not occur, public methods are checked to comply with their constraints, considering the state 'parameters related at the same level' which corresponds to the possible invocations by client code.

## 7.5 Summary

We have presented opaque balloon types, an extension to balloon types, which constitute the next layer in a hierarchy of mechanisms devoted to controlling object sharing.

Opaque balloon types are devoted to dynamic aliasing—not to direct aliases to a given object—but to the more rewarding issue, which can only be addressed after introducing balloon types, of preventing aliasing involving any part of the state reachable by an object and any dynamic reference in a given context.

Client code manipulating an opaque balloon object $O$ can be assured that no variable references any state reachable by $O$. Opaque balloon types offer, thus, a strong form of encapsulation of state, present in primitive types such as integer, but not offered for user-defined types by any current data abstraction mechanism.

The opaque balloon invariant turned out to demand a careful definition, based on a concept that we have called piercing. As the basis for a checking mechanism to enforce the invariant, we have developed the nesting interpretation. This we did by refining the information about cluster relationships and adding what we call nesting levels.

Finally, we have sketched two alternatives for a checking mechanism for opaque balloon types, based on the nesting interpretation. These can form the starting point to obtain an actual checking mechanism, tuned to the particular assumptions that can be made about the language in question.

# Chapter 8

# Language Issues

The balloon checking mechanism was presented in the context of the simple RISO language. There are several issues related to incorporating balloon types in real languages which were left to be discussed now. We will address modular checking, global variables, subtype and parametric polymorphism, and (briefly) concurrency and distributed systems. In some cases we do not intend to present 'the solution', but to consider several possibilities in the design space.

## 8.1 Modularity

The checking mechanism for plain balloon types as previously described is a global program analysis: the whole program undergoes a static analysis, and is accepted or rejected according to the outcome. This is quite unrealistic; although global analyses may be considered for optimisation purposes, they are unacceptable for type checking:

- programmers expect to be able to construct modules, have them individually type-checked and use them in other parts of the program;
- 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 may even be written in different languages.

For these reasons, from the moment the idea was conceived modularity was taken into account. For presentation purposes only did we describe the checking mechanism as a global interprocedural analysis of RISO programs containing a 'flat' set of function declarations. A modular checking mechanism can be based on the static analysis as described, including performing fixed-point calculations due to loops and mutual recursion in functions within the module, but ignoring external code and using conservative approximations of functions external to the module.

Another point which we consider important for the integration of balloon-types in languages is that non-balloon types, like current user-defined types, should be able to be freely coded, with no restrictions and no need for a static analysis as the checking mechanism. Therefore, 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 (unless they are part of a module containing balloon classes).

The simple rule was essential towards this, by allowing a reference to a balloon to be freely propagated in client code without requiring any form of static analysis. Interestingly, this is what happens in current languages such as Java: the simple rule is implicit because for the only possible 'balloons' (the primitive types) the assignment has value semantics, and no special restrictions exist for the 'non-balloons' (the user-defined types).

### 8.1.1 Constraints and Conservative Approximations

Not only is it necessary to use approximations of functions, but we must also impose constraints on the functions which belong to the interface of a balloon type, i.e. on the *public* methods of a *class* which implements a balloon type. Constraints are necessary because these functions may be invoked by non-balloon client code which is not subject to static analysis. A public method in balloon class `B` is forbidden to:

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

The first constraint applies because non-balloons that come as parameters may already 'belong' to some balloon or if captured could make the balloon 'internal to itself', violating $I_2$. The second constraint is necessary because a reference to an internal non-balloon could be stored in some external object by non-balloon client code. A special case of these constraints 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. With these constraints each balloon class `C` 'takes care of itself', without making assumptions about how it is used.

To verify that these constraints are met by a class `B`, the analysis checks that:

- non-balloon parameters to a public method do not become part of the same equivalence class as a variable of class `B`.

- a public method in class `B` does not return a variable of non-balloon type that belongs to the same equivalence class as a variable of class `B`;

The above constraints are also helpful in the approximation of functions: they can be counted upon to hold in functions from any balloon type, which allows an approximation better than a 'blind' worse case. In checking client code which invokes a method of `B`, '`b.f(x,y,z)`' where `x`, `y`, and `z` are of non-balloon type, although it is assumed that parameters can be merged, the first constraint ensures that they are not 'captured' (a state $\boxed{\mathrm{b}}\,\overline{\underline{\mathrm{x}}}\,\overline{\underline{\mathrm{y}}}\,\overline{\underline{\mathrm{z}}}$ cannot be transformed into another like $\boxed{\mathrm{bx}}\,\overline{\underline{\mathrm{y}}}\,\overline{\underline{\mathrm{z}}}$ due to the invocation). Therefore, this function is approximated by using $\boxed{\mathrm{b}}\,\overline{\underline{\mathrm{xyz}}}$ as the response to **1** ($\boxed{\mathrm{b}}\,\overline{\underline{\mathrm{x}}}\,\overline{\underline{\mathrm{y}}}\,\overline{\underline{\mathrm{z}}}$).

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

In the case of opaque balloon types being used in a plain balloon type, not only does what we say above apply, but we can also make use of the opaque balloon invariant to simplify checking. In the analysis of a function in a balloon class `B`, as no internal objects of an opaque balloon can be accessed, the relevant set of identifiers `I`—which defines the cpo $C_I$—need not include those corresponding to opaque balloon types. This means that variables of opaque balloon types (including all primitive types such as integer or real) can be ignored. This is important because the size of $C_I$ has an exponential growth with the number of identifiers in $I$.

The points made in the two previous paragraphs also apply when 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 (opaque) balloon types to be considered in type-checking user-defined code, 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.

### 8.1.2 Units of Modularity

The more simple option for a modular checking mechanism is to make the unit of modularity coincide with the *class*. Each class is checked individually and, apart from counting on the constraints that exist in functions of balloon types, blind approximations of external functions are used.

Looking at each class individually means that some programs may be rejected when such would not be the case were a global analysis performed. It is possible to increase the range of accepted programs by simultaneously analysing a group of classes, if not the whole program. This is specially important when a small group of classes cooperate tightly. It may happen that a balloon class uses some auxiliary non-balloon classes for some internal data structures, and it may be useful to have a more precise knowledge of the effect of functions from these non-balloon classes.

Constructs can be provided so that the checking mechanism considers groups of classes simultaneously; we now discuss some of the more obvious (and not mutually exclusive) options.

#### A Module as a Set of Classes

Some construct for module of balloon type checking as a set of classes could be introduced. It could look something like:

```
module
{
  class A { ... }
  class B { ... }
  ...
}
```

All classes within a module would be analysed together. A weak point in this construct is that a class would belong to one module only; different modules do not intersect. This does not allow the internals of an 'auxiliary' class `A` to be considered by several classes, say B and C, unless these latter classes are also in the same module. It could be the case that these classes are unrelated and would be artificial to put them in the same module. Moreover, it could be that B uses a D and C uses an E, which would also have to be in the same module if their internals were to be considered. Soon the size of a module could become unmanageable.

### Expressing Dependence

The previously described situation is dealt with by expressing that a class depends on the internals of other classes. This can be done like:

```
class A { ... }
class B depends on A,D { ... }
class C depends on A,E { ... }
class D { ... }
class E { ... }
```

This allows a class A to be considered by different unrelated classes during their checking. From a programmer's perspective it expresses in a natural way that some class B is checked relying on the behaviour of some class A (and not just its interface), and that internal changes to A may invalidate B.

### Inheritance

A situation which should be treated like the previous case is the use of *inheritance* (a mechanism to reuse code, which should not be confused with subtyping, even if they frequently coincide). Consider the following example where class B inherits from A and overrides method g:

```
class A
{
  f() {  self.g() ... }
  g() { ... }
}
class B inherits A
{
  override g() { ... }
}
```

Class A is checked considering the interaction between f and g. Namely, the constraints which f must obey if it is a public method are checked considering the effects of the original g.

In class `B` method `f` is inherited, but the invocation of `g` in its body will now refer to the new `g` defined in `B`. This means that in checking `B` it is not correct to assume that the constraints which held in `f` are still valid in the context of class `B`. This implies that, while checking `B`, the definitions of the inherited parts must be considered.

In terms of modularity the **`inherits`** should, therefore, be considered analogous to the **`depends on`**: the checking of class `B` considers the internals of class `A`, and so internal changes to `A` affect the acceptance of `B`.

While this lack of modular checking is not a pleasant property, interestingly it parallels the tight coupling between super and subclasses when reusing code: programmers are often required to look at the inherited code in order to write and understand the subclass.

This inability of a class to be self-contained (concerning modular checking) in the presence of inheritance can be overcome by a more conservative checking mechanism with no interprocedural analysis even in the intra-class case. This could, however, lead to fewer programs being accepted. Whether or not to use a more conservative checking will be a language design issue.

## Nested Classes

If a language has a mechanism of nested classes, it can also be used to provide a unit of modularity for balloon checking. A given balloon class would be checked by considering simultaneously its nested classes. In the following program class `A` would be analysed together with `B` and `C`.

```
class A
{
  class B { ... }
  class C { ... }
  ...
}
```

Nested classes enable the following design option: as nested classes are not used outside, the constraints we have described need not be imposed in the case of balloon classes nested in a balloon class; this is possible because the containing class is subject to checking.

In languages like Beta a nested class is allowed to manipulate state variables in a containing class, as discussed in [59]. In this case, the points we make below concerning global variables apply.

### 8.1.3 Increasing Interface Information

We designed the balloon mechanism as syntactically minimal: nothing more than a keyword in the definition of the data type is necessary. It can, however, be useful under modular checking to have more information explicit (syntactically flagged), that can be counted upon by the mechanism. Extra information can be provided by *optional* keywords in the interface of the data type. This is analogous to providing keywords such as `const` in C++.

For a modular balloon checking mechanism in particular, it can be useful to have extra information regarding formal parameters of non-balloon type. As it is, the mechanism assumes that an invocation of some function from another module '`f(x,y,z)`' may cause `x`, `y` and `z` to be 'merged' if they are of non-balloon type: for example, a state $\underline{\overline{\mathrm{x}}}\,\underline{\overline{\mathrm{y}}}\,\underline{\overline{\mathrm{z}}}$ goes to $\underline{\overline{\mathrm{xyz}}}$, and $\underline{\overline{\mathrm{x}}}\,\underline{\overline{\mathrm{y}}}\,\boxed{\mathrm{z}}$ goes to $\boxed{\mathrm{xyz}}$.

Information about non-balloon variables can be provided by a keyword like `free`, meaning that the parameter will not be 'merged' with other parameters.

```
non-balloon type T { ... }
f(free a:T, b:T, c:T) { ... }
```

Given the above function, some code containing an invocation '`f(x,y,z)`' will be checked assuming that `x` will not be 'merged' with either `y` or `z`: for example, a state $\underline{\overline{\mathrm{x}}}\,\underline{\overline{\mathrm{y}}}\,\underline{\overline{\mathrm{z}}}$ goes to $\underline{\overline{\mathrm{x}}}\,\underline{\overline{\mathrm{yz}}}$, and $\boxed{\mathrm{x}}\,\boxed{\mathrm{y}}\,\underline{\overline{\mathrm{z}}}$ goes to $\boxed{\mathrm{x}}\,\boxed{\mathrm{yz}}$. This way the checking mechanism is less conservative, which means that more programs will be accepted.

The free qualifier can also be extended to the result of a function, this time meaning that the result is in a separate equivalence class from any parameter (and that, therefore, it can be subsequently captured by some balloon, even if some parameter was already captured).

The safe use of the `free` keyword implies that the function containing the keyword must be checked to comply with the obligations stated in the signature. While this is similar to the checking of the constraints that public functions in balloon types must obey under a modular checking mechanism, it means that non-balloon code containing the `free` keyword must undergo a static analysis. In the case of libraries and interfacing between languages, the library provider has to be trusted, as for the balloon information.

While we cannot predict what would be the reaction of programmers to such a keyword, we stress that it is an optional aid, and that the balloon mechanism was conceived without it. We also make the observation that this keyword is meaningful in itself as describing what is to be expected from a function. While some aspects have been traditionally covered in signatures of functions, examples being `var` in Pascal and `const` in C++, the `free` qualifier addresses something which has been neglected. Under one perspective, we can say that there are three kinds of use that a function can give to a reference passed as parameter (or to reachable objects):

- inspect the state of the object;
- modify a 'self-contained' part of the state of the object.
- create object linking, either by storing the reference to the object in 'other' objects or vice-versa.

It is this last case that creates object sharing (static aliasing), and which is the cause of many problems, as the sharing created may remain after the function terminates, and is very difficult to 'track'. In balloon types this last case is forbidden. In non-balloon types, although by default no restriction applies, because this last case is a source of problems, and not always needed, it can be useful to signal that it does not occur for a given parameter of a function. This is precisely what the `free` qualifier does.

## 8.2 Global Variables

Global variables can be accessed by different objects, with no need for parameter passing. One example of global variables in object-oriented languages are the so called *class variables*: global variables with class scope. One of two points of view can be adopted regarding global variables.

Under a permissive point of view, plain balloon types deal exclusively with the graph of objects. Global variables are named entities, which reference objects exactly like local variables do; they are not themselves part of the object graph, and do not take part in the balloon invariant. Global variables exist in a fixed finite number; therefore, the possibility of interference due to global variables would be something to be detected/prevented by an independent mechanism, which could build on approaches such as Reynold's Syntactic Control of Interference [69].

Even adopting this point of view, global variables can be used to propagate references between objects without need for parameter passing. To keep the balloon checking mechanism modular, some restriction must be imposed. An attractive possibility is to have the rule that global variables must be of balloon type: they will contain some information globally accessible, but they will be self-contained and will not contribute to the propagation of sharing through the object graph.

Under a more restrictive point of view, more in the spirit of balloon types, two invocations of a parameterless operation on two different balloon objects should not be able to access common state. The two invocations:

```
x.left.f();
x.right.f();
```

which could arise in a divide-and-conquer situation, should never interfere, something which a compiler should be able to explore; this is one of the motivations for the introduction of balloon types. In this case we must prevent global variables from being manipulated by classes which implement balloon types or by non-balloon classes of objects internal to a balloon.

One way to keep the balloon checking mechanism modular is to make the ability to manipulate global variables explicit in the interface of a data type. Then, if a type is declared not to manipulate global variables, a candidate implementation will be rejected if it directly involves global variables or if it uses a type declared as being able to do so. Balloon types are part of the ones prevented from manipulating global variables, while non-balloon types will be declared as being or not being able to do so.

While this can be seen by some as 'yet another nuisance' in programming a data type, it amounts to documenting the data type regarding the existence of invisible communication channels between instances. Again, we see this as improving the conditions for reasoning about data types.

## 8.3 Subtype Polymorphism

Object-oriented languages support what is known as *subtype polymorphism*. Looking at types as sets of values, the subtype relation can be regarded as set inclusion. If $S$ is subtype of $T$ (which we can write as $S <: T$), then $S$ describes a subset of the values described by $T$.

A type system uses a subtype relation together with a *subsumption* rule: if $S <: T$ and $e : S$ then $e : T$. Subsumption means in practice that we can use an expression of type $S$ where an expression of type $T$ is expected. In the program:

```
x:T;
y:S;
f(p:T) { ... }
x :- y;
f(y);
```

the assignment and invocation are type correct if $S <: T$.

If we make the balloon information part of object types we must also consider it in subtyping. Suppose we have an object type $T$ which is balloon. To have $S <: T$, $S$ must also be balloon. This is because a subtype must respect all invariants present in the supertype, namely the balloon invariant. (Otherwise we could have $x : T$ and $x$ being assigned a reference to an object which does not respect the balloon invariant, and the information that $x$ is of balloon type would be useless.)

If we have $T$ non-balloon, the first impression is that we could have a subtype $S$ which is balloon; after all, a subtype can add invariants which are not present in the supertype. But we are not discussing a type system for functional languages where only *values* matter; mutable object sharing is relevant. If there are restrictions in the use of references to objects of a type $S$, these restrictions must also be present in a supertype $T$. In the case of the balloon mechanism, references of balloon type $S$ are forbidden to be assigned to state variables (by the simple rule). If a non-balloon type $T$ was (wrongly) considered as a supertype of $S$, then a reference of type $S$ could be assigned to a variable of type $T$, which could be subsequently assigned to a state variable of some object, by-passing the simple rule and possibly creating sharing of a balloon object:

```
non-balloon T { ... }
balloon S { ... }
x:T;
y:S;
x :- y;
z.v :- x;
```

From the above it follows that in our proposed model, where any object type must be either balloon or non-balloon, both subtypes and supertypes of a balloon type must also be balloon types, and analogously for non-balloon types. This means having two disjoint type hierarchies, one for balloon types and other for non-balloon types. Within each hierarchy the usual subtyping rules [20, 6] apply.

Having two disjoint hierarchies goes against having a lattice of types with a bottom and a top. Object-oriented type systems sometimes make use of a `Null` type (containing only the `null` value) as the bottom of the hierarchy and a `Any` type as the top of the hierarchy.

`Null` is made the bottom of the hierarchy as it is mathematically 'nice' and so that a variable of any other type can have `null` as a possible value. This is however a rather artificial situation; the `null` value does not represent an object (which can be updated and store information), but is simply a marker for 'no object'. Also it cannot respond to method calls (unless by causing some sort of runtime exception), when it would be required to respond to any possible invocation to be considered a subtype of any other type. The above are some of the reasons why the `null` value is a special case which should be considered as such, instead of making artificial attempts to using a `Null` type as an ordinary object type which is put on the bottom of the hierarchy. The `null` value does not cause problems for making balloons part of the type system. Variables of any object type $T$, balloon or non-balloon, can be allowed to have the `null` value (in addition to referring to objects of some class which implements $T$); it does not interfere with the balloon invariant.

If on the other hand `Null` is to be used as an ordinary object type and put at the bottom of the type hierarchy, then object types cannot be simply classified into balloon/non-balloon: `Null` will have a special status, being neither balloon nor non-balloon.

Contrary to `Null`, the `Any` type poses genuine problems and deserves some attention. As we have discussed, under a binary classification of object types into balloon/non-balloon, it is not possible to have a `Any` object type.

If we do not restrict object types to a binary classification, and assume that some type can be a supertype of both balloon and non-balloon types, then types such as `Any` could exist. However, for the type system to be sound, these types must be subject to restrictions.

Given a type $T$ with $B <: T$ and $N <: T$, where $B$ is balloon and $N$ is non-balloon, a variable $x : T$ can point to both balloon and non-balloon objects. As $x$ can point to balloon objects, the simple rule must apply for type $T$. On the other hand, as $x$ can point to non-balloon objects, not only cannot the balloon invariant be counted upon, but while checking code which uses $T$ it must be assumed that sharing may be created as if $x$ is of non-balloon type. Variables of any type $U$ (including `Any`) supertype of $T$ will also be able to refer to balloon and non-balloon objects, and must be subject to identical treatment. These types, with respect to balloon information, can be positioned as *undefined*. With this third classification,

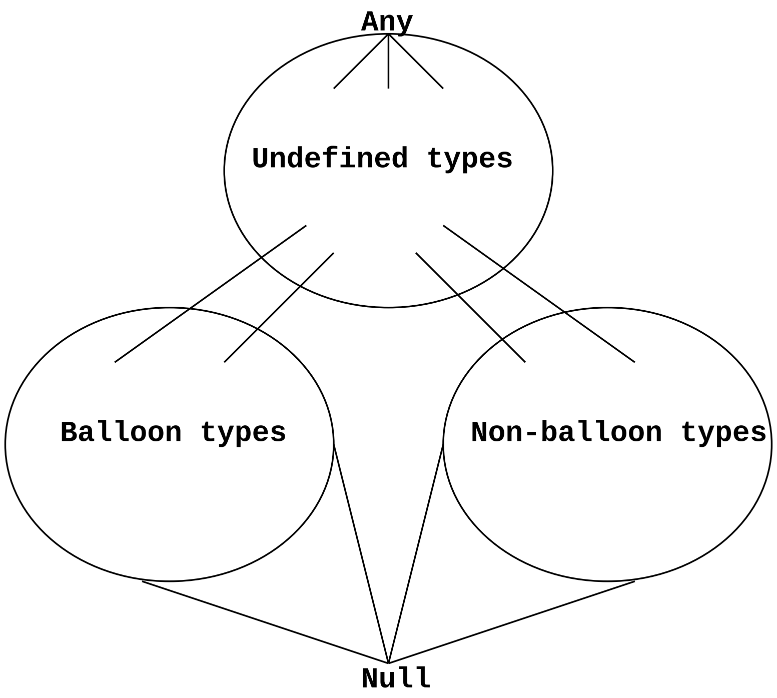


Figure 8.1: Lattice of types given the *undefined* classification

and if the `Null` type is also considered on its own, it is possible to have a lattice of types as in Figure 8.1.

Although a possible option, we do not see it as a good solution. These `undefined` types combine the restrictions of both balloon and non-balloon types with no associated benefit. Unlike non-balloons, they are not much suitable to be used in auxiliary linked structures, due to the simple rule. Unlike balloons, a checking mechanism cannot count upon them to 'take care of themselves' while checking code which uses them. This makes it doubtful whether much useful code can be written with them.

We defend the binary division in balloon/non-balloon for object types of variables (parameters, local variables and state variables), and remark that this division does not apply to higher-order types or type parameters, as we discuss below. The expressiveness which is demanded in many cases when types such as `Any` are used (like in generic container types) can be achieved through parametric polymorphism.

## 8.4 Parametric Polymorphism

When a function has many types given by an expression involving a type parameter, it is said to exhibit parametric polymorphism. In the actual coding the type parameter may be, depending on the language, explicit or implicit. An important example of a language with parametric polymorphism is ML [63], where it is possible, for example, to define a function which gives the length of a list with elements of an arbitrary type:

```
fun length [] = 0
  | length (x::xs) = 1 + length xs
```

Although no explicit type parameter is used (or any type information at all), a polymorphic type is inferred for the function:

$$\forall t. \, \mathrm{List}[t] \rightarrow \mathrm{Int},$$

where we have used universal quantification of a type variable to express the polymorphic type, as in [19].

As well as generic functions, it is useful to be able to define generic data types; this is the case for the so called ‘container’ types. An example is the dictionary data type:

```
type Dictionary[Elem,Key]
{
  insert(Key,Elem);
  search(Key):Elem;
}
```

which can be instantiated for different types of elements or search keys, as in:

```
DictShape = Dictionary[Elem = Shape, Key = String];
ds1, ds2: DictShape;
PhoneBook = Dictionary[Elem = Int, Key = String];
pb1, pb2: PhoneBook;
```

### 8.4.1 Subtype-bounded Parametric Polymorphism

It is important to be able to impose bounds on type parameters (as opposed to unbounded universal quantification), to express requirements that type parameters must conform to. As an example, suppose the `Point` type:

```
type Point = {x:Int; y:Int}
```

and a move procedure:

```
all[P <: Point]
move(p:P, dx:Int, dy:Int)
{
  p.x := p.x + dx;
  p.y := p.y + dy;
}
```

Move is polymorphic (in the type of the first parameter) as it can be applied to an infinite set of types, as long as they are subtypes of `Point` and, therefore, objects of these types have `x` and `y` state variables. We could have, for example:

```
type ColourPoint = {x:Int; y:Int; c:Colour}
cp: ColourPoint;
move(cp, 2, 1);
```

where `ColourPoint` is a subtype of `Point`. The type of `move` is:

$$\forall t <: \text{Point}.\, t \times \text{Int} \times \text{Int} \rightarrow \text{Unit}.$$

We are in the presence of *bounded parametric polymorphism*; as before, we have a universal quantification of the type parameter, which is also bounded using, in this case, the subtype relation. The 'classic' ML parametric polymorphism can be seen as a special case, where the type 'Any' is used as bound:

$$\forall t.\, e(t) = \forall t <: \text{Any}.\, e(t).$$

Bounded parametric polymorphism allows, therefore, expressing a wider range of situations. These issues are discussed in [19].

We discuss now how balloon types can be incorporated in a language, coexisting with parametric polymorphism. Variables in a program (parameters of functions, local variables or state variables of objects) range over object types which, as we have argued, should be either balloon or non-balloon types; we have argued that the *undefined* types in Figure 8.1 should not be allowed for types of variables. However, from what we have just described, if the same applies to type parameters, it will preclude describing useful generic types where a type parameter ranges over both balloon and non-balloon types; it can be overconstraining to be forced to use in the bound a type which is already classified into balloon or non-balloon. In particular, it would mean that type parameters could not be bounded by the `Any` type, precluding important cases of generic code.

For these reasons, type parameters should be able to range or be bounded by, not only balloon or non-balloon types, but also the 'undefined' types. These will be allowed to be used for type parameters but not for types of variables.

It is important to stress that, even though in a generic function or type a variable can be declared to have type $T$, with $T$ a type parameter bounded by an 'undefined' type, when $T$ is instantiated it will be as *either* a balloon or a non-balloon type. This is very different from what we have argued against in the previous section: having a variable of an 'undefined' type which can (by subtype polymorphism) reference balloon or non-balloon objects in different moments at run-time.

An implementation of a generic type containing a type parameter $T$ bounded by an 'undefined' type can be, therefore, checked twice to contemplate both possibilities: $T$ being balloon and $T$ being non-balloon. If there are $n$ type parameters we have $2^n$ possibilities; this is not, however, a practical problem because a generic type has typically very few type parameters, 1 or 2 in most cases. A generic type itself can be left with the balloon classification undefined; each instantiation will define whether it is balloon or non-balloon. A checking of the generic type is made for each case, which means that we have a total of $2^{n+1}$ cases. The dictionary is one example where it would be useful to leave both type parameters (`Elem` and `Key`) and the type itself with an undefined balloon classification; one possible instantiation would be:

```
type Shape = balloon ...
type String = balloon ...
type DictShape = balloon Dictionary[Elem = Shape, Key = String];
ds1, ds2: DictShape;
```

Although more than one checking is needed, we are still close to the 'one checking per generic' philosophy of bounded parametric polymorphism, and not to the brute force 'one checking per instantiation' as in C++'s *templates*.

## 8.4.2 Match-bounded Parametric Polymorphism

As a last point we address the recent developments in object-oriented type systems, which recognise that subtyping is not the best relation on which to base bounded parametric polymorphism. The central issue is that subtyping fails to hold due to the negative occurrences of the recursion variable in a recursive type; essentially due to *binary methods* [14].

In the dictionary example, the search operation could require that `Key` provides some compare operations like `equal` or `less`; these operations take another `Key` as parameter and return `Bool`; they are examples of binary methods. We could hope to use a type `Comparable` as a bound in expressing these requirements in the dictionary type:

```
type Comparable
{
  equal(Comparable):Bool;
  less(Comparable):Bool;
}
type Dictionary[Elem, Key <: Comparable]
  ...
```

and hope to instantiate the dictionary, as above, with the search key being `String`, defined as:

```
type String
{
  equal(String):Bool;
  less(String):Bool;
  concat(String):String;
  ...
}
```

However, `String` is not a subtype of `Comparable`, due to the subtyping rules for recursive types [6]. Indeed, `Comparable` does not admit interesting subtypes at all. This means that subtyping is not appropriate to describe bounded parametric polymorphism in many useful cases.

A relation called *matching* was introduced in Emerald [12] and PolyTOIL [15] as a complement to subtyping. Matching, originally presented as similar to F-bounded subtyping [18], and later [1] based on higher-order subtyping, can be informally seen as describing 'protocol extension'. We have that $X$ matches $Y$ (written $X <\# Y$) if $X$ provides all operations of $Y$ and possibly some more, and (typically) preserves recursive structure. This is exactly what happens between `String` and `Comparable`; we have that String <# Comparable. Matching turns out to be more appropriate than subtyping as the relation on which to base bounded parametric polymorphism, as described in [16].

In what concerns integrating balloon types with match-bounded parametric polymorphism, all that we have said for the subtype-bounded case applies exactly, replacing subtyping by matching. The dictionary example can be successfully written using match-bounded parametric polymorphism:

```
type Dictionary[Elem, Key <# Comparable]
  ...
type Shape = balloon ...
type String = balloon ...
type DictShape = balloon Dictionary[Elem = Shape, Key = String];
ds1, ds2: DictShape;
```

## 8.5 Concurrency and Distributed Systems

Balloons are a general concept. The balloon invariant can be regarded as a graph-theoretical notion: it describes a property of a graph with two 'kinds' of nodes. Balloons give structure to a global *state* (a pool of unnamed objects), regardless of the transient *computations* which are performed and their associated 'temporary views into the state' (i.e. local variables).

We have not mentioned issues such as object location: objects can reside in the memory of a machine, in a persistent object store, or be spread across a distributed system. Defining self-contained sets of objects, balloons are indeed relevant for distributed systems, as candidate units of object placement, migration, replication, and parameter/result passing to remote invocations. In the above cases, single objects would have too fine a granularity and would not be, in general, self-contained.

In what concerns computation, it does not matter whether we have sequential execution or several concurrent threads. The balloon invariant applies to both cases, as it ignores dynamic aliasing involving local variables of threads. An important point is that essentially the same checking mechanism can be applied to a concurrent model of execution. This is because the mechanism works under minimal assumptions about dynamic aliasing and does not consider the internal structure of objects. (Here we assume that local variables are private to each thread and only the pool of objects is shared.)

We make a clear distinction between (active) threads of execution and (passive) objects; and a corresponding distinction between local variables in threads and state variables of objects. In a model like *actors* [3], where objects are 'active', the distinction becomes blurred. However, in 'real' systems there are usually many more passive objects than active threads of execution; therefore, the distinction we make is important.

When several threads operate on shared data, concurrency control mechanisms are needed; balloons are also useful in this respect. In general, more than one thread can have references to a given balloon, it being indeed unrealistic to try to prevent this from happening. Balloons can, however, be used as units of locking, so that only one thread 'enters' a balloon. A thread which enters a *plain* balloon $B$ can safely operate on any object in the cluster of $B$; however, it will have to perform locking of internal balloons, as dynamic references to these objects may exist in other threads.

Opaque balloons are particularly suitable as units of concurrency control: a thread which locks and enters an opaque balloon will be able to operate on *any* internal object with no need for further locking. This does not preclude intra-balloon concurrency: while 'inside' an opaque balloon, several sub-threads can be created to operate concurrently on different internal balloons.

## 8.6 Summary

We have addressed some issues related to incorporating balloon types in programming languages. A basic issue concerns modularity; here we have discussed constraints to be imposed on functions in the interface of a balloon type so that we can obtain a modular checking mechanism. Modularity can be achieved at the cost of using approximations of functions, with the result of accepting fewer programs. Approximations must be used anyway in object-oriented languages with subtype polymorphism (where the code being invoked depends on the class of the object); they are also suitable when using libraries provided with no source code.

A simple option for a modular checking mechanism is to make the unit of modularity coincide with the class. We have discussed the possibility of considering groups of classes simultaneously, useful when some classes cooperate tightly. We have also considered the use of an optional keyword (`free`) in non-balloon parameters of functions. This would enhance the quality of the approximations, being also useful in itself from the programmer's point of view.

Global variables are a hidden source of interference as they can be manipulated with no need for parameter passing. Under a permissive point of view, global variables do not matter for the balloon invariant; but as they can be used to propagate sharing, we must restrict their use; an attractive possibility is to allow global variables only of balloon type. More in the spirit of balloon types, we must prevent global variables from being manipulated by classes which implement balloon types or by non-balloon classes of objects internal to a balloon. One way to keep the checking mechanism modular is to make the ability to manipulate global variables explicit in the interface of a data type.

When integrating balloon types in languages with subtype polymorphism, the balloon property must be considered in subtyping. For types of variables, both the subtypes and supertypes of a balloon type must be balloon, and analogously for non-balloon types. This means having two disjoint hierarchies. The `Null` type is a special case and does not pose problems; `Any` cannot, however, be used as a type of a variable.

Regarding parametric polymorphism, it will be useful to have *undefined* types with respect to the balloon property. Undefined types, such as `Any`, will be able to be used as the bound for type variables. However, a type variable must be instantiated by either a balloon or a non-balloon type in order for a generic type to be able to be used as the type of a variable. An implementation of a generic type with type parameters bounded by undefined types will be checked for the possible combinations of substituting type paramters by balloon or non-balloon types. Balloon types integrate well with parametric polymorphism: unbounded or bounded either by subtyping or (more appropriately) by matching.

We have concluded with a comment on how balloon types are a general concept, discussing, in particular, their usefulness for concurrent and distributed systems.

# Chapter 9

# Conclusion

## 9.1 Discussion

In Chapter 2 we described how accidental state sharing is a source of problems and how current language mechanisms do not provide an appropriate solution. We also compiled a list of key points which should be taken into account.

Our concept, balloon types, meets all these key points: it considers reachable state and substructure sharing; it does not resort to physical containment; it allows duplication of references to be used by local variables; it recognises the different nature of dynamic and static aliasing; it does not use named entities to organise the object graph; it is syntactically simple and conceptually relevant; it makes it possible that primitive types are not longer considered 'special'.

Most approaches we have mentioned fail to address many of these points, but there is one proposal, *Islands* [42], which takes most of these points into account, and which we now compare with our own approach. 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 enables 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*.

  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 a key element in the balloon checking mechanism.

Making the ability to share objects a full part of the type system is a central contribution of balloon types. Most research in type systems has privileged a functional setting, having been done by a community of theory-oriented people, using the lambda calculus as the starting point.

On the other hand, although aliasing has been the topic of much research (see for example the survey by Deutsch [31]), aliasing is mainly considered for implementation purposes. Many static analyses concerning aliasing have been developed, towards some optimisation. However, because aliasing itself is not part of the underlying type-system, there is not much these analyses can 'hold on to'. As very little can be assumed, many difficulties are encountered. In order to avoid being overly conservative, one has to resort to complex global program analyses, with their inherent disadvantages.

Such approaches should not be criticised; they put the emphasis on existing code—many of these analyses were developed for Fortran or C. However, difficulties will persist if sharing remains ignored by the language itself. We are not concerned with short term solutions; our proposal is a long term one, addressing a fundamental and missing element in type systems of imperative languages. Balloon types provide a missing support that further analyses towards optimisation can 'hold on to'. As the balloon mechanism itself uses static analysis, the support it provides could not be offered by a purely syntactical mechanism requiring no effort; this is a hint on the added value of balloon types.

Although we make use of static analysis, the emphasis of balloon types is not on static analysis itself but on what they aim to provide. One of the original ideas behind the mechanism was to provide support to static analysis but not to involve static analysis itself (or at most some simple static analysis); the one we had to resort to was far more complex than expected.

We should also report that it was important to build a static analysis tailored to the mechanism. We did not have much success in reusing previous static analyses. Having worked with static analysis before (in [4]), we started a sketch of the static analysis for balloon types based on existing analyses (namely [21]). We tried to adapt the analysis to our purpose, but due to the assumptions we had to make (about aliasing, modularity, integration with object-oriented languages) and the particularity of the balloon invariant, after some versions and some errors being discovered, we ended up abandoning it. Then we started to use abstract interpretation and built an abstract domain tailored to the assumptions we had to make and to the balloon invariant.

It can be argued that the use of static analysis in the checking mechanism is a potential problem. Indeed, widely used languages do not use static analysis in the type system; it is typically simple for a programmer to know whether a program is type correct and to understand the error messages from the compiler. With the use of a static analysis in type checking, even if we do not have a global program analysis, the definition of what is an acceptable program becomes more complex. About this we can say that under a simple checking more programs would be rejected, even increasing syntactical complexity.

The use of static analysis in the checking mechanism will require more sophisticated error reporting to be developed, in order for the compiler to give the reason why some code may break the balloon invariant. The good side is that, performing a static analysis, the compiler 'knows' more, in general (for example about object creation, and uses of local variables). This means that it can be more easily extended to consider issues other than balloon checking, such as uninitialised variables or null references. This points towards having more 'smart' compilers. Under this perspective, balloon types are an example of how type checking can be extended in order to raise the level of abstraction in imperative languages.

In what concerns balloon types in particular, we emphasise that it is only the checking of code implementing a balloon type that requires a static analysis. For client non-balloon code only the simple rule concerning reference assignment applies. This means that even if it turns out that in some cases beginner programmers may feel some difficulty in understanding error reports for balloon code, every programmer benefits from having balloon types available in libraries constructed by more experienced programmers.

The checking mechanism for opaque balloon types turned out to be much more complex than expected. In a first impression one would think that, after having the balloon invariant, not much more than 'preventing references from being returned' would be needed to obtain opaque balloons. As it turned out, even the definition of opaque balloons itself needed considerable attention. This has happened because we strived to maintain generality, in particular, taking into account recursive data structures of opaque balloons, which led to the introduction of nesting levels.

We should, however, point out that opaque balloon types with recursive structure in the data will not be a common case: to have full freedom of pointer manipulation, recursive structures of non-balloons can be used to build the internal state in the implementation of a data type, which will be made itself an opaque balloon type. An example of this is the `BigInt` type (in Figure 3.6), which makes use of the recursive non-balloon type `Node`. The checking mechanism for a data type which is not recursive in the data will be considerably less involved, as there is no need for the level information.

Opaque balloon types represent truly opaque data abstractions, while plain balloon types still allow references to the state to escape to clients, leaving still open considerable possibilities of interference. This raises the question of whether all balloon types should be opaque, and no plain balloon types should be offered. The issue is that having too many choices available is in itself a source of complexity which programmers must face.

Here we have concentrated on creating the mechanisms on a hierarchical fashion, rather than addressing the issue of which ones to provide in a specific language. This will be a design decision (although an important one) which has to be made after collecting empirical evidence on what can be expressed by the different mechanisms; it will also depend on what other mechanisms are offered by the language in question.

Plain balloon types should be offered in languages without higher-order features, so that objects in a balloon container can be operated upon from client code through dynamic references being returned (as in the dictionary of shapes in Chapter 3). On the other hand, if enough higher-order features are provided, many situations can be expressed with opaque balloon types: instead of returning references to client code, the container type can provide some kind of 'map' function, to which the operation to be applied is passed as argument.

The dynamic aliasing which exists when references are returned will still be an obstacle to analysing code. Due to this, we argue that the combination of higher-order features with opaque balloons, if proven expressive enough, should be preferable.

## 9.2 Summary of Contributions

We have described how accidental state sharing is a source of unexpected program behaviour and an obstacle to reasoning about programs or performing program transformations. We have argued that current language mechanisms provide only a weak form of encapsulation, neglecting the fact that an object is not normally self-contained and failing to deal with substructure sharing. We have exposed weaknesses of other proposals and have compiled a list of relevant points which should be taken into account by language mechanisms. Of these, we stress the need to make an appropriate distinction in the treatment of dynamic and static aliasing.

We have proposed to make the ability to share state a first class property of data types, introducing the concept of balloon types. Every data type becomes either balloon or non-balloon: non-balloon types allow state sharing, while balloon types prevent unwanted sharing guaranteeing a strong form of encapsulation of state through the balloon invariant. This is proposed with minimal syntactical cost: only an extra keyword in the definition of a data type, and no syntactic cost on client code. The use of balloons by client code is subject only to a 'simple rule' regarding reference assignment.

Complexity is hidden from the programmer by relying on a non-trivial compile-time mechanism to check the implementation of balloon types, assuring that the balloon invariant will hold at run-time. The checking mechanism is based on a static analysis which we developed formally using abstract interpretation. Special concern was devoted to the representation of functions, avoiding the naive function space, and obtaining an efficient abstract interpretation.

As a side-effect, balloon checking was in itself an interesting case study in the use of abstract interpretation, from which some lessons were learnt; these were essentially in the design of abstract domains, namely concerning the separation between summarising information about concrete states and merging control paths.

We have developed opaque balloon types, a specialisation of balloon types providing a stronger invariant; client code is assured that no variable references any state reachable by an opaque balloon object. This represents a very strong form of encapsulation of state, present in primitive types such as integer, but not offered for user-defined types by any current data abstraction mechanism. Opaque balloons are truly opaque data abstractions which guarantee that all internal state remains unchanged between invocations of operations from the data type. This is an important property to reason about data types. We have developed an abstract interpretation which forms the basis for a checking mechanism for opaque balloon types, and sketched two alternatives for such a mechanism.

Although these concepts have not been implemented, this has not been merely an abstract mathematical exercise; practical issues were taken into account while developing balloon types. Issues related to incorporating balloon types in programming languages have been addressed: modularity, global variables, subtype and parametric polymorphism, and (shortly) concurrency and distributed systems.

To summarise, we have presented balloon types, a general language/type-system mechanism for imperative languages which: makes the ability to share state a first class property of data types; provides a strong form of encapsulation of state; allows a cluster of objects to be treated as a self-contained composite object; and gives user-defined types the same status as primitive types, which need no longer be considered 'special'. This solves a long-standing flaw in current data abstraction mechanisms.

## 9.3 Research Directions

In this thesis we have defined the balloon mechanism. The next logical step will be to implement the mechanism and to put it into use. One task will be to develop an efficient implementation of the checking mechanism itself. This will involve choosing appropriate data structures. More important are the language issues that will arise from integrating balloon types into languages. Two lines of research can be followed: one is to add balloon types to 'real' languages; the other is to design languages with balloon types.

In the first approach we can chose a popular language and augment it with balloon types. A good candidate will be the Java language: it is widespread, it is relatively simple (compared with languages such as C++), and the memory model is appropriate (objects are passed by reference and contain references to other objects). Java is, indeed, a language where balloon types would be welcome, as the little support to prevent sharing in user-defined types which exists in C++ (physical containment) is not provided in Java. In this approach compatibility is a key issue; many problems to face will be related to language advocacy, as opposed to strictly technical issues.

If we are not so worried about obtaining short term benefits, but more concerned with long term advances in programming languages, the second approach can be more rewarding. The problem with the first approach is that, as balloon types is not a simple 'feature' but a fundamental concept, the best result will be obtained not by adding balloon types to an otherwise unchanged language, but by designing the whole type system taking into account the interactions between the different concepts. As an example, subtyping is strongly affected by balloon types; in order to have a reasonably expressive language, it will be important to have parametric polymorphism together with balloon types. As Java does not support parametric polymorphism, both concepts must be considered and added together. (One can, instead, consider experimental extensions of Java, such as Pizza [65], supporting parametric polymorphism.) We can always reuse the syntactic look of Java; as long as we design the type system as a whole and we are not hindered by backward compatibility issues we will be able to obtain a more advanced language.

Regardless of the approach taken, it will be important to write code and collect empirical evidence about the usability of the concept. This can help making design choices; one example is whether to provide both plain and opaque balloons or only opaque balloons, as we have discussed. We also need to evaluate whether the static checking mechanism allows expressive enough code to be written. Another aspect which will demand research is, as we have already mentioned, error reporting, in order to make the compiler explain in a programmer understandable form why the mechanism has rejected a piece of code.

Another line of research consists of exploring the mechanism towards obtaining better language implementations. Contrary to mechanisms involving unchecked annotations, the invariants provided by the balloon mechanism will be able to be counted upon by the language implementation towards making program transformations.

One such transformation, which motivated the development of balloon types, is parallelisation. Traditionally difficult due to the pervading possibility of aliasing, code analysis towards parallelisation will benefit from the balloon invariant. Static aliasing involving non-balloons internal to some balloon will be confined to a small group of objects as opposed to the whole object graph. In balloons, no static aliasing exists at all and code analysis needs only concentrate on dynamic aliasing. This opens up research possibilities into parallelisation techniques that take advantage of these properties. Opaque balloons types will be specially appealing. An opaque balloon will be able to be treated as a single unit, which means that classic parallelisation techniques dealing with arrays but which work only for primitive types will be able to be adapted for user-defined types.

The knowledge about the organisation of the object graph can also be explored in memory management. It is easy to see a garbage collector taking advantage of the balloon invariants. As an example, if an opaque balloon object becomes garbage while executing client code, all internal objects will also be garbage and can be collected. In the case of plain balloons the situation is not so simple and needs some research.

An important issue is object copying. As we have discussed in Chapter 3, the copying of balloons can be subject to optimisations and the balloon invariant does not have to hold physically, as long as the outcome is the same. This motivates research towards avoiding deep copies and managing physical sharing. In this, it will be relevant to detect whether objects remain immutable or when updates are performed. This requires the development of static analyses, with the possible introduction of some syntactic mechanism.

In value types the (conceptual) copy will be the only variant allowed in assignment or parameter passing; therefore, optimising away copies will be the essential point in future research concerning value types. (The checking mechanism is essentially one for opaque balloon types.) As we have discussed, one possibility is to have simple pointer copies in client code, share physically the objects representing the value, and to detect whether functions in the implementation of the value type would modify the objects manipulated (including `self`) so as to avoid physical copies. Keeping a copy shallow by sharing nested values, and detecting the possibility of update in place must also be pursued. More generally, implementation techniques from functional languages should be considered and reassessed in this context.

Distributed systems is an area where balloon types have great potential: a single object is too small a unit that incurs in large communication overheads due to protocols; balloons provide a way to obtain self-contained clusters of objects which can be 'chunked' and treated as a single unit. This means that balloons can be used to explore new directions in traditional topics in distributed systems, such as object placement and migration, replication, parameter passing to remote invocations, and granularity issues. Balloons can also be useful as candidate units of locking in concurrency control mechanisms.

Other than putting the balloon mechanism to use and exploring it towards improving language implementations, another line of research is extending the mechanism itself.

In the area of the integration between imperative and functional languages, as well as providing value types, research should be made towards providing higher order facilities to languages with balloon types. This is not to mean supporting every combination of features. For example, allowing higher order operations involving non-balloons could lead to much effort developing a corresponding static checking mechanism, when such generality may lead to error-prone programs involving side-effects which are difficult to understand. Instead, careful language design can lead to a more restrictive mechanism, which may be expressive enough, understandable by programmers, and not too complex to implement.

An example is restricting the functions to be manipulated to functions where the ground types are balloon types, allowing the creation of closures that only make use of a self-contained group of objects. Closures could be treated as objects and, therefore, there would be the rule that they cannot share balloon objects owned elsewhere, and can only manipulate copies of such objects. This way, a closure would access a self-contained set of data and would not cause unsuspected side-effects. These are examples of how balloon types can be used in research towards the integration of functional and imperative languages.

Research can also be made towards obtaining extensions of the balloon mechanism which prevent further the possibility of aliasing. Opaque balloon types do not prevent different variables to refer to the same balloon, while value types are used to obtain immutable objects and to program in a functional style. We can think of a specialisation of opaque balloon types which goes further in preventing dynamic aliasing but still allows for mutable objects (i.e 'between' opaque balloon and value types).

An attractive possibility is to explore a specialisation of opaque balloons with the further rule that there cannot be dynamic aliasing involving parameters of a function, as in Pascal or Fortran in call-by-reference. As we have discussed, a static checking mechanism would be overly conservative, but we could use dynamic checking based on simple pointer comparison, which would be a small overhead, as we are dealing with operations on composite objects. As opaque balloons already take care of preventing aliasing involving internal objects, with this extra rule we will have obtained the counterpart for composite objects of what happens in Pascal or Fortran for single objects. The knowledge provided by this absence of aliasing could be explored in the checking mechanism for the implementation of such types, which would be based on the one for opaque balloons. A simple improvement towards enhancing precision would be to explore the absence of 'reaching' between parameters; a more fundamental direction to be explored under the absence of aliasing is to consider the structure of objects (i.e. the state variables) of the class being checked.

# Appendix A

# Mathematical Terminology and Notation

Here we present briefly the mathematical terminology and notation we use in this thesis. Most of it is based on one of [36, 28, 76, 45]. We do not intend nor can we afford a detailed treatment. An introduction to these topics can be found on the above referenced works.

## A.1 Logical Expressions

We use the following operators to build logical expressions:

| | |
|---|---|
| $\neg$ | not |
| $\wedge$ | and |
| $\vee$ | or |
| $\Rightarrow$ | implies |
| $\Leftrightarrow$ | if and only if |
| $\forall$ | for all |
| $\exists$ | exists |

Operator $\neg$ has the highest precedence, followed by $\wedge$ and $\vee$; these are followed by $\Rightarrow$ and this by $\Leftrightarrow$. In $\forall x.\, P(x)$ and $\exists x.\, P(y)$ these operators bind $x$ as far as possible to the right.

## A.2 Sets

A set $X$ with elements $a_1, \ldots, a_n$ is written as $\{a_1, \ldots, a_n\}$. If $x$ belongs to $X$ we write $x \in X$, otherwise $x \notin X$. We write $\{x \in X \mid P(x)\}$ for the set of elements $x$ belonging to $X$ such that predicate $P(x)$ holds; when $X$ is implicit from the context we abbreviate it as $\{x \mid P(x)\}$. We use the following notation regarding sets:

| | |
|---|---|
| $\emptyset$ | empty set |
| $\lvert X \rvert$ | size of $X$ |
| $X \subseteq Y$ | $X$ is subset of $Y$ |
| $\mathcal{P}(X) = \{A \mid A \subseteq X\}$ | powerset of $X$ |
| $X \setminus Y$ | set difference |
| $X \cup Y$ | set union |
| $X \times Y$ | cartesian product |

## A.3 Relations

A (binary) relation $R$: $X \nrightarrow Y$ is a set of ordered pairs $(x, y)$, subset of $X \times Y$. We write $x\ R\ y$ if $(x, y) \in R$, and $x \not R\ y$ otherwise. We also use $x\ R_1\ y\ R_2\ z$ to abbreviate $x\ R_1\ y \wedge y\ R_2\ z$. Given a relation $R$ on a set $X$, we say that:

$R$ is reflexive if $x\ R\ x$ for all $x \in X$;
$R$ is symmetric if $x\ R\ y \Rightarrow y\ R\ x$;
$R$ is antisymmetric if $x\ R\ y \wedge y\ R\ x \Rightarrow x = y$;
$R$ is transitive if $x\ R\ y \wedge y\ R\ z \Rightarrow x\ R\ z$.

For a relation $R$ on a set $X$ and relations $R_1$: $X \nrightarrow Y$ and $R_2$: $Y \nrightarrow Z$, we define:

| | | |
|---|---|---|
| $I_X$ | $= \{(x, x) \mid x \in X\}$ | identity relation on $X$ |
| $R_2 \circ R_1$ | $= \{(x, z) \mid \exists y.\ x\ R_1\ y\ R_2\ z\}$ | composition of $R_1$ and $R_2$ |
| $R^0$ | $= I_X$ | |
| $R^{n+1}$ | $= R \circ R^n$ | |
| $R^+$ | $= \bigcup_{n \in \omega} R^{n+1}$ | transitive closure of $R$ |
| $R^*$ | $= R^+ \cup I_X$ | transitive reflexive closure of $R$ |
| $R \mid A$ | $= \{(x, y) \in A \times A \mid x\ R\ y\}$ | domain restriction of $R$ to $A$ |
| $R \mathbin{\overline{\mid}} A$ | $= R \mid (X \setminus A)$ | domain subtraction of $R$ by $A$ |

For a relation $R$: $X \nrightarrow Y$, we define:

| | |
|---|---|
| $\operatorname{dom} R = \{x \mid \exists y.\ x\ R\ y\}$ | domain of $R$ |
| $R^{op} = \{(y,x) \mid x\ R\ y\}$ | opposite of $R$ |
| $RA = \{y \mid \exists x \in A.\ x\ R\ y\}$ | direct image of $A$ under $R$ |

## A.4 Functions

A function $f : X \to Y$ is a relation such that $\operatorname{dom} f = X$ and such that $x\ f\ y$ and $x\ f\ z$ implies that $y = z$. We write $fx$ for the unique $y$ such that $x\ f\ y$.

We write a function as $\{a \mapsto b, c \mapsto d, \ldots\}$, which is the same as $\{(a,b),(c,d),\ldots\}$, but emphasises that we are dealing with a function and not merely a relation. We write a function also as $\{x \mapsto e(x) \mid P(x)\}$; an example is $\{x \mapsto x+1 \mid x \in \{1,2,3\}\}$.

**Function Update** A common notation in semantics to describe a function which results from another by a modification is the 'function update':

$$f[x \mapsto y] = f \setminus \{(x, fx)\} \cup \{(x,y)\}.$$

We also use a variant with multiple updates:

$$f[x \mapsto e(x) \mid P(x)] = f \setminus \{(x, fx) \mid P(x)\} \cup \{(x, e(x)) \mid P(x)\}.$$

**Lambda Notation** We also make use of lambda notation to define functions. We write:

$$\lambda x.\, e$$

for the function that maps $y$ to the result of evaluating $e$ with the free occurrences of $x$ in $e$ replaced by $y$. When using this notation, either the domain where $x$ lies in is implicit from the context, or we can use the variant $\lambda x : X.\, e$.

## A.5 Order

**Preorder** A preorder on a set $P$ is a reflexive transitive relation $\underset{\sim}{\sqsubset}$ on $P$. When $\underset{\sim}{\sqsubset}$ is implicit from the context we say that $P$ is an ordered set.

**Partial Order** A partial order on a set $P$ is an antisymmetric preorder $\sqsubseteq$ on $P$. A partially ordered set can also be called a poset.

**Comparable and Incomparable** Elements $x$ and $y$ are comparable if either $x \underset{\sim}{\sqsubset} y$ or $y \underset{\sim}{\sqsubset} x$; otherwise they are incomparable.

**Total Order or Chain** A total order on a set $P$ is a partial order $\leq$ on $P$ where all pairs of elements are comparable.

(We tend to use the symbols $\leq$ for a total order, $\sqsubseteq$ for a partial order that is not a total order, and $\underset{\sim}{\sqsubset}$ for a preorder that is not a partial order.)

**Antichain** An antichain is an ordered set where all pairs of elements are incomparable. A set $P$, ordered as an antichain, is written $\overline{P}$.

**Height and Width** The height of a poset is the size of its largest chain; the width of a poset is the size of its largest antichain.

**Monotone** Given ordered sets $A$ and $B$, a function $f : A \to B$ is said to be monotone if $x \underset{\sim}{\sqsubset}_A y$ implies $fx \underset{\sim}{\sqsubset}_B fy$.

**Inflation** A function $f : A \to A$ on an ordered set $A$ is said to be an inflation if $x \underset{\sim}{\sqsubset} fx$ for all $x$ in $A$.

**Order Ideal** A subset $D$ of an ordered set $P$ is an order ideal (or a down-set) if $d \in D$, $x \in P$ and $x \underset{\sim}{\sqsubset} d$ implies $x \in D$. The following constructs result in order ideals:

$$\begin{aligned} {\downarrow}A &= \{y \mid \exists x \in A.\ y \underset{\sim}{\sqsubset} x\} && \text{order ideal generated by } A \\ {\downarrow}x &= \{y \mid y \underset{\sim}{\sqsubset} x\} && \text{principal order ideal generated by } x \end{aligned}$$

The set of order ideals of an ordered set $P$, ordered by set inclusion, is a poset denoted by $\mathcal{O}(P)$.

**Maximal and Greatest, Minimal and Least** An element $x$ of an ordered set $P$ is maximal if for all $y \in P$, $y \neq x$, we have $x \not\underset{\sim}{\sqsubset} y$; it is a greatest element of $P$ if for all $y \in P$, we have $y \underset{\sim}{\sqsubset} x$. Dually, we have that $x$ is minimal if for all $y \in P$, $y \neq x$, we have $y \not\underset{\sim}{\sqsubset} x$. Also, $x$ is a least element of $P$ if for all $y \in P$, we have $x \underset{\sim}{\sqsubset} y$. The set of maximal elements of $P$ is written $\operatorname{Max} P$.

**Top and Bottom** By antisymmetry, a poset contains at most one least and one greatest element. The least element of a poset $P$, if it exists, is called bottom, written $\bot_P$ or simply $\bot$. The greatest element of a poset $P$, if it exists, is called top, written $\top_P$ or simply $\top$.

**Bounds** If $P$ is an ordered set and $S$ a subset of $P$, $x \in P$ is an upper bound of $S$ if $y \precsim x$ for all $y \in S$. The set of all upper bounds of $S$ is written $S^u$.

If $P$ is a poset and $S$ a subset of $P$, the least element of $S^u$, if it exists, is called the least upper bound of $S$, and written $\bigsqcup S$ (read as 'join of $S$'). The least upper bound $\bigsqcup\{x, y\}$, if it exists, can be written as $x \sqcup y$. Dually, we write the set of all lower bounds of $S$ as $S^l$ and the greatest lower bound of $x$ and $y$ as $x \sqcap y$ (read as $x$ meet $y$).

**Join Semilattices** A non-empty poset $P$ is a join semilattice if $x \sqcup y$ exists in $P$ for all $x, y \in P$.

**Lattices** A non-empty poset $P$ is a lattice if $x \sqcup y$ and $x \sqcap y$ exists in $P$ for all $x, y \in P$.

**Cpos** A poset $P$ is a complete partial order (cpo) if it has a least upper bound $\bigsqcup_{n\in\omega} x_n$ in $P$ for all chains $x_0 \sqsubseteq \cdots \sqsubseteq x_n \sqsubseteq \cdots$ in $P$. If a cpo $P$ has a least element $\bot$ we say $P$ is a pointed cpo.

**Continuous Functions and Function Spaces** A monotone function $f : A \to B$ between cpo's $A$ and $B$ is said to be continuous if for all chains $x_0 \sqsubseteq \cdots \sqsubseteq x_n \sqsubseteq \cdots$ in $A$ we have

$$\bigsqcup_{n\in\omega} f x_n = f \bigsqcup_{n\in\omega} x_n.$$

The function space denoted by $[A \to B]$ is a cpo; this is the set of all continuous functions between cpo's $A$ and $B$:

$$\{f : A \to B \mid f \text{ is continuous }\}$$

ordered pointwise, that is:

$$f \sqsubseteq g \Leftrightarrow \forall x \in A.\ fx \sqsubseteq gx.$$

If $B$ is pointed, then $[A \to B]$ is also pointed.

**Fixed Points** A continuous function $f : P \to P$ on a pointed cpo $P$ has a least fixed point $\mathrm{fix}\, f$ (that is, $f(\mathrm{fix}\, f) = \mathrm{fix}\, f$ and for any other $x$ such that $fx = x$ we have $\mathrm{fix}\, f \sqsubseteq x$), given by:

$$\mathrm{fix}\, f = \bigsqcup_{n \in \omega} f^n \bot.$$

**Products and Sums** The product of cpo's $P_1, \ldots, P_n$, is a cpo denoted by $P_1 \times \cdots \times P_n$; it is made up of elements $(x_1, \ldots, x_n)$ from the cartesian product of the underlying sets, with order defined coordinatewise, that is:

$$(x_1, \ldots, x_n) \sqsubseteq (y_1, \ldots y_n) \Leftrightarrow x_1 \sqsubseteq y_1 \wedge \cdots \wedge x_n \sqsubseteq y_n.$$

The sum of cpo's $P_1, \ldots, P_n$, is a cpo denoted by $P_1 + \cdots + P_n$; it is made up of elements from the disjoint union of the underlying sets (using injection functions if they intersect), with two elements being comparable if they 'come' from the same $P_i$ and are comparable there (i.e. as if the diagrams are put side by side).

**Lifting** If we have a cpo $P$, we can construct a pointed cpo $P_\bot$ ($P$ lifted) by adding a $\bot$ and using a function $\lfloor - \rfloor$ such that:

$$\lfloor x \rfloor = \lfloor y \rfloor \Rightarrow x = y, \text{ and } \forall x.\ \lfloor x \rfloor \neq \bot.$$

The cpo $P_\bot$ is made up of elements

$$\{\lfloor x \rfloor \mid x \in P\} \cup \{\bot\}$$

and order

$$x \sqsubseteq y \Leftrightarrow x = \bot \vee \exists x', y'.\ x = \lfloor x' \rfloor \wedge y = \lfloor y' \rfloor \wedge x' \sqsubseteq y'.$$

**Let Construction** It can be useful to evaluate an expression $e_1$ on a lifted cpo $A_\bot$ and depending on the outcome: if it is not $\bot$, pass it to a function $(\lambda x.\, e_2) : A \to B$ ($B$ is pointed); if it is $\bot$, 'bypass' the function and return $\bot$ directly. This is the purpose of the let construct:

$$\mathrm{let}\, x \Leftarrow e_1.\, e_2 \quad = \quad \begin{cases} \bot & \text{if } e_1 = \bot, \\ (\lambda x.\, e_2)y & \text{if } \exists y.\ e_1 = \lfloor y \rfloor \end{cases}$$

## A.6 Substitution

We use the notation $\cdots[y/x]$ to mean $\cdots$ with $x$ replaced by $y$. This notation is used in very different cases, including substituting identifiers in a piece of syntax. It should not be confused with $[x \mapsto y]$ used in the special case of function update; they produce very different results even when applied to a function:

$$\begin{aligned} \{(x,3),(z,5)\}[x \mapsto 4] &= \{(x,4),(z,5)\} \\ \{(x,3),(z,5)\}[y/x] &= \{(y,3),(z,5)\} \end{aligned}$$

We also use the following two variants of this notation to handle multiple substitutions simultaneously: $\cdots[y_1,\ldots,y_n/x_1,\ldots,x_n]$ (here all $x_i$'s must be different); and $\cdots[e(x)/x \mid P(x)]$.

## A.7 Vector Notation

We use the vector notation $\overrightarrow{x_n}$ as a syntactic abbreviation of $x_1,\ldots,x_n$, whatever these elements are. This abbreviation does not include any surrounding element, such as parenthesis. This way we can use $\{\overrightarrow{x_n}\}$ for the set $\{x_1,\ldots,x_n\}$ and $(x_0,\overrightarrow{x_n},1)$ for the tuple $(x_0,x_1,\ldots,x_n,1)$. (We have borrowed this versatile form of vector notation from [45].)

# References

[1] M. Abadi and L. Cardelli. On subtyping and matching. In *Proceedings ECOOP'95*, volume 952 of *LNCS*, pages 145–167, August 1995.

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

[3] Gul Agha and Carl Hewitt. Actors: A conceptual foundation for concurrent object-oriented programming. In P. Wegner and B. Shriver, editors, *Research Directions in Object-Oriented Programming*, pages 49–74. MIT Press, 1987.

[4] P. S. Almeida. Exploração de paralelismo em linguagens orientadas por objectos. Master's thesis, Universidade do Porto, 1994.

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

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

[7] G. E. Andrews. The theory of partitions. In *Encyclopedia of Mathematics and its Applications*, volume 2. Addison-Wesley, 1976.

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

[9] Henry G. Baker. Lively linear lisp—'look ma, no garbage!'. *SIGPLAN Notices*, 27(8):89–98, August 1992.

[10] Henry G. Baker. Equal rights for functional objects or, the more things change, the more they are the same. *ACM OOPS Messenger*, 4(4):2–27, October 1993.

[11] Henry G. Baker. 'Use-once' variables and linear objects—storage management, reflection and multi-threading. *SIGPLAN Notices*, 30(1):45–52, January 1995.

[12] A. Black and N. Hutchinson. Typechecking polymorphism in emerald. Technical Report DEC-CRL-91-1, Digital Equipment Corporation, Cambridge Research Lab, 1991.

[13] Edwin Blake and Steve Cook. On including part hierarchies in object-oriented languages, with an implementation in Smalltalk. In *Proceedings ECOOP'87*, LNCS 276, pages 41–50. Springer-Verlag, 1987.

[14] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Objects Group, G. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[15] K. B. Bruce, A. Schuett, and R. van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In *Proceedings ECOOP'95*, volume 952 of *LNCS*, pages 27–51, August 1995.

[16] Kim B. Bruce, Adrian Fiech, and Leaf Petersen. Subtyping is not a good "match" for object-oriented languages. In *Proceedings ECOOP'97*, LNCS 1241, pages 104–127. Springer-Verlag, 1997.

[17] P. A. Buhr and C. R. Zarnke. Nesting in an object oriented language is not for the birds. In *Proceedings ECOOP'88*, volume 322 of *LNCS*, pages 128–145. Springer-Verlag, August 1988.

[18] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings FPCA'89*, pages 273–280. ACM, 1989.

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

[20] 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.

[21] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. *Proceedings PLDI'90. SIGPLAN Notices*, 25(6):296–310, June 1990.

[22] Franco Civello. Roles for composite objects in object-oriented analysis and design. *Proceedings OOPSLA'93. SIGPLAN Notices*, 28(10):376–393, October 1993.

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

[24] J. R. Cordy. Compile-time detection of aliasing in Euclid programs. *Software – Practice and Experience*, 14(8):775–768, August 1984.

[25] 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.

[26] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

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

[28] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[29] Alain Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proceedings of the IEEE 1992 International Conference on Computer Languages*, pages 2–13. IEEE Press, April 1992.

[30] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond $k$-limiting. *Proceedings PLDI'94. SIGPLAN Notices*, 29(6):230–241, June 1994.

[31] Alain Deutsch. Semantic models and abstract interpretation techniques for inductive data structures and pointers. In *Proceedings ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 226–229, June 1995.

[32] P. Fradet and D. Le Métayer. Shape types. In *Proceedings 24th ACM Symposium on Principles of Programming Languages*, pages 27–39, January 1997.

[33] A. A. Fraenkel, Y. Bar-Hillel, and A. Levy. *Foundations of Set Theory.* North-Holland, 2nd edition, 1973.

[34] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

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

[36] Paul R. Halmos. *Naive Set Theory.* Van Nostrand Reinhold, 1960.

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

[38] Laurie J. Hendren, Joseph Hummel, and Alexandru Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. *Proceedings PLDI'90. SIGPLAN Notices*, 27(7):249–260, July 1992.

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

[40] 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.

[41] 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.

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

[43] R. C. Holt, P. A. Matthews, J. A. Rosselet, and J. R. Cordy. *The Turing Programming Language. Design and Definition.* Prentice Hall, 1988.

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

[45] Neil D. Jones. *Computability and Complexity from a Programming Perspective.* MIT Press, 1997.

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

[47] S. Kent and I. Maung. Encapsulation and aggregation. In *Proceedings TOOLS Pacific 95 (TOOLS 18).* Prentice Hall, 1995.

[48] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, 1977.

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

[50] Won Kim, Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, and Darrel Woelk. Composite object support in an object-oriented database system. *Proceedings OOPSLA'87. SIGPLAN Notices*, 22(12):118–125, December 1987.

[51] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *Proceedings 20th ACM Symposium on Principles of Programming Languages*, pages 196–205, January 1993.

[52] Nils Klarlund and Michael I. Schwartzbach. Graphs and decidable transductions based on edge constraints. In *Trees in Algebra and Programming – CAAP'94*, volume 787 of *LNCS*, pages 187–201, April 1994.

[53] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. The BETA programming language. In P. Wegner and B. Shriver, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. MIT Press, 1987.

[54] Y. Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.

[55] 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.

[56] B. H. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* The MIT Press, 1986.

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

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

[59] Ole Lehrmann Madsen. Block structure and object oriented languages. *SIGPLAN Notices*, 21(10):133–142, October 1986.

[60] Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language.* Addison-Wesley, 1993.

[61] Chris Martin and Chris Hankin. Finding fixed points in finite lattices. In *Proceedings FPCA'87*, volume 274 of *LNCS*, pages 426–445. Springer-Verlag, September 1987.

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

[63] Robin Milner, Mads Tofte, and Robert W. Harper. *The Definition of Standard ML.* MIT Press, 1990.

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

[65] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings 24th ACM Symposium on Principles of Programming Languages*, pages 146–159, January 1997.

[66] G. Plotkin. Lambda definability and logical relations. Technical report, Department of AI, University of Edinburgh, 1973.

[67] G. J. Popek, J. J. Horning, B. W. Lampson, J. G. Mitchell, and R. L. London. Notes on the design of EUCLID. *Proceedings of the ACM Conference on Language Design for Reliable Software. SIGPLAN Notices*, 12(3):11–18, 1977.

[68] Charles Rapin. Block structured object programming. *SIGPLAN Notices*, 32(4), 1997.

[69] J. C. Reynolds. Syntactic control of interference. In *Proceedings 5th ACM Symposium on Principles of Programming Languages*, pages 39–46, January 1978.

[70] Markku Sakkinen. Disciplined inheritance. In *Proceedings ECOOP'89*, pages 39–56. Cambridge University Press, 1989.

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

[72] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, and C. H. A. Koster. *Report on the Algorithmic Language ALGOL 68.* Springer-Verlag, Berlin, 1969.

[73] P. Della Vigna and C. Ghezzi. Context-free graph grammars. *Information and Control*, 37(2):207–233, 1978.

[74] Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods.* IFIP TC2 Working Conference on Programming Concepts and Methods, North-Holland, April 1990.

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

[76] Glynn Winskel. *The Formal Semantics of Programming Languages.* MIT Press, 1993.

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

[78] Niklaus Wirth. On the design of programming languages. In *Information Processing 74*, pages 386–393, 1974.

[79] D. B. Wortman. On legality assertions in EUCLID. *IEEE Transactions on Software Engineering*, SE-5(4), 1979.

[80] W. Wulf and Mary Shaw. Global variable considered harmful. *SIGPLAN Notices*, 8(2):28–34, February 1973.