

# Validity Invariants and Effects

Yi Lu, John Potter, and Jingling Xue

Programming Languages and Compilers Group  
School of Computer Science and Engineering  
University of New South Wales, Sydney  
{ylu,potter,jingling}@cse.unsw.edu.au

**Abstract.** Object invariants describe the consistency of object states, and are crucial for reasoning about the correctness of object-oriented programs. However, reasoning about object invariants in the presence of object abstraction and encapsulation, arbitrary object aliasing and re-entrant method calls, is difficult.

We present a framework for reasoning about object invariants based on a behavioural contract that specifies two sets: the *validity invariant*—objects that must be valid before and after the behaviour; and the *validity effect*—objects that may be invalidated during the behaviour. The overlap of these two sets is critical because it captures precisely those objects that need to be re-validated at the end of the behaviour. When there is no overlap, no further validity checking is required.

We also present a type system based on this framework using ownership types to confine dependencies for object invariants. In order to track the validity invariant, the type system restricts updates to permissible contexts, even in the presence of re-entrant calls. Object referencing and read access are unrestricted, unlike earlier ownership type systems.

## 1 Introduction

The flexibility and extensibility offered in object-oriented programming is both a boon and a curse. Classes provide an encapsulated definition of object data and behaviour; subclassing allows the extension of existing definitions with reuse of code that depends on a parent class; heap-based allocation allows objects to persist beyond the scope of their creator; object references provide a disciplined use of pointers. However this programming flexibility comes at a cost. Reasoning about the behaviour of objects is difficult. More specifically, the presence of complex object dependencies, object aliasing, and arbitrary method call-backs (re-entrant calls) makes it difficult to reason about object-oriented code. The particular problem we focus on in this paper is how we can guarantee the validity of objects within such a programming context.

Formal verification techniques for structured programming encapsulate the program state within local variables and arguments of procedures. This makes it feasible to provide a complete axiomatic semantics based on a pre/post-condition style of reasoning about code. A critical assumption in such reasoning systems, is that the program state that is being reasoned about is encapsulated within code

blocks—a stack-based memory model is assumed. Such reasoning techniques cannot directly cope with reasoning about programs with pointers—the essential problem is that heap-based data may be accessed and modified indirectly via pointers. Currently the most promising approach for formally dealing with pointers appears to be Separation Logic [6, 30, 16, 7] which supports reasoning about the distinctness of regions of the heap.

Design by contract is a technique for reasoning about objects introduced by Meyer [22] and provided with programming language support in Eiffel [21]. Design by contract relies on the specification of object behaviours via the combination of object invariants describing valid object states, and pre/post-conditions for object behaviours. Although design by contract does provide a good conceptual basis for assisting designers, its formal basis is weakened by the presence of inter-object dependencies, aliasing and call-backs.

We extend an example used by Leino and Müller [18] to illustrate the effect of call-backs on object invariants. If  $P$  calls back on  $m$  then a divide-by-zero error may occur. The problem is that  $P$  is called when the current invariant is broken, and cannot be relied on in subsequent re-entrant calls. On the other hand, if  $Q$  in  $n$  is re-entrant there is no harm, assuming it maintains invariants.

```
class C {
  int a, b;      invariant: 0 <= a < b;
  C() { a = 0;   b = 3; }
  void m() { int k = 100/(b-a);
           a = a+3;  P(...);  b = (k+4)*b; }
  void n() { int k = 100/(b-a);  Q(...); } }
```

In order to reason about this type of code, we need to be able to track the effect and dependency of possible call-backs on the current object invariant. This is our goal.

The difficulty of modular verification of object-oriented programs has been recognized for a long time [17]. Recently researchers have made some in-roads, with the use of fixed ownership-based schemes in Universe Types [25, 23, 14, 27], and dynamic ownership used to track object validity in the Boogie methodology, as manifested in Spec# [3, 18, 28, 4]. Our own work draws strongly on the ideas of object validity of this work. Further comparisons will follow in Section 5.

In this paper we present a general framework for tracking validity within object systems which is independent of any particular language framework. The motivation for adopting a language-free approach is to emphasize the underlying principles. The key idea of our model is to capture the effect of code blocks (such as method bodies) on the validity of the objects in the system. The approach we take requires code blocks to specify a pair of object sets  $\langle J, \mathcal{E} \rangle$ .  $J$  is the *validity invariant* for the code, which specifies those objects that must be valid before and after the code executes.  $\mathcal{E}$  is the *validity effect* which specifies those objects that may be invalidated during execution of the code. During execution, nested code blocks or method calls  $\langle J', \mathcal{E}' \rangle$  must satisfy some consistency criteria with respect to their calling context. When the sets  $J$  and  $\mathcal{E}$  are disjoint, there is nothing to

check, other than consistency criteria for the code—system validity is automatically guaranteed to hold. If  $\mathcal{J}$  and  $\mathcal{E}$  are not disjoint, validity of the objects within their overlap must be established by other means (presumably relying on a more detailed program logic). It is this ability to hone in on a restricted critical set of objects whose validity must be established after code execution which provides the novelty in our model. When object validity is specified *via object invariants*, *our model imposes certain structural constraints* that relate the form of  $\mathcal{J}$  and  $\mathcal{E}$  to the dependencies inherent in the object invariants.

A general model is only useful if it can be realized in some concrete form. To that end, we offer a small language, Oval, that realizes the model using an ownership-based type system. In Oval, an object’s invariant can only depend on its own fields, and on other objects that it contains, as determined by the ownership structure. For simplicity, we restrict the overlap between  $\mathcal{J}$  and  $\mathcal{E}$  to be, at most, a single object which is always the current active object. Consequently, *only local (per-object) reasoning is necessary to establish system-wide validity*.

Within Oval, we do not impose restrictions on object reference or read access; our system does not rely on alias protection. Instead, interpreting the consistency criteria of nested code blocks (method calls) from our general model, we restrict what write access is allowed in different contexts. It is this mechanism that allows us to keep track of which objects are valid at any particular time; for example, if a call-back requires an object to be valid, we can prohibit it if that object may be invalid but allow it if we know it must be valid.

Interestingly, it is straightforward to create immutable objects in Oval. Objects high up in the ownership hierarchy are more accessible for update (and more likely to be invalid). For us, immutable objects are those created at the very bottom of the ownership hierarchy, where nothing has write access. Note that immutability is determined by the object’s creation type (instantiating the object’s owner to be `bot`) rather than the object’s class (where the owner is a formal parameter, rather than a concrete context). Our system also provides a more general model than approaches based on read-only annotations. In Oval, the context of the reference holder determines its update capability.

Our system is also able to express encapsulation-aware read-only references. They typically use hidden contexts (equivalent to existentially abstract contexts) to forbid access to methods that have some write capability on the objects encapsulated by the context where the reference is held.

In summary, specific contributions of this paper include:

- a validity specification for blocks of code—defining a set of objects  $\mathcal{J}$  that must be valid before and after, and a set of objects  $\mathcal{E}$  that may be invalidated during code execution;
- identification of where explicit validity checks are required within code blocks, and where validity can be assumed to hold;
- a model of structure based on the object dependency implied by object invariants;
- a small language, Oval, with an ownership-based type system—where an object’s invariant can only depend on an its own fields and its owned objects, and system validity can be achieved with per-object checks;

- immutable objects and a generalization of read-only annotations arise as a special case of the type system.

This paper is organized as follows. Section 2 details our general approach for describing system validity and the behavioural abstraction  $\langle \mathcal{J}, \mathcal{E} \rangle$  introduced above. In particular we outline the consistency conditions for nested code with the underlying rationale. Our initial model is descriptive, rather than focused on specific language mechanisms for programming objects. Section 3 introduces the Oval language with examples. Here, our intention is to illustrate language mechanisms and type rules to support our general model of Section 2 rather than to design a realistic programming language. Unlike most previous ownership schemes [13, 12, 10, 8, 11], we do not use the object ownership structure to restrict object references and aliasing, but rather, we use it to specify the sets of objects that must be valid, or may be invalid, during method calls. This use of ownership builds on earlier work extending the use and flexibility of ownership type systems [20, 19]. With Oval we have opted for simplicity rather than a fully expressive model. Oval requires that object invariants are only invalidated one object at a time; possible extensions avoiding this limitation are discussed in this section. In Section 4 we provide a static and dynamic semantics for our language and formalize properties that demonstrate how this language implements the general model. Section 5 addresses other related work, including Boogie/Spec# and Universes, Ownership Types and read-only systems. Section 6 briefly concludes the paper.

## 2 A Model for Object Validity

### 2.1 The Validity Contract $\langle \mathcal{J}, \mathcal{E} \rangle$

The key idea for our model is very simple: whenever an object is active, it may be invalid. If an object is not active (and still alive) then it must be valid. But what determines when an object is valid? For now, it suffices to consider an object to be valid when it satisfies its specified invariant—this is the standard notion of object validity. Later on, in Subsection 2.3, we will refine this notion somewhat, in order to handle object invariants that depend on more than one object. So, for the moment, to maintain consistency with the refined notion, we will simply assume that there is some notion of object validity, and at any time, there is a set of all valid objects, that we call the *validity set*  $\text{VALID}$ .

For any particular structured block of code, we specify a behavioural abstraction: the *validity contract*  $\langle \mathcal{J}, \mathcal{E} \rangle$ . The *validity invariant*  $\mathcal{J}$  specifies a set of objects that must be valid both before and after the code executes. Clearly this imposes an obligation on the caller (see Subsection 2.2) and the code itself (see below). The *validity effect*  $\mathcal{E}$  specifies those objects that may be invalidated during execution of the code.

A given validity contract  $\langle \mathcal{J}, \mathcal{E} \rangle$  for a block of code provides constraints on  $\text{VALID}$  at different execution timepoints as shown in Table 1. In the following we denote the validity set at the start of code execution by  $\text{VALID}_0$ , so that at

any timepoint,  $\text{VALID}_0 - \text{VALID}$  represents invalidated objects—those that were initially valid, but are currently invalid. Interpreting constraints, we first have,

Before:	$\mathcal{J} \subseteq \text{VALID}$
During:	$\mathcal{J} - \mathcal{E} \subseteq \text{VALID}$ $\text{VALID}_0 - \text{VALID} \subseteq \mathcal{E}$
At End:	$\mathcal{J} \cap \mathcal{E} \subseteq \text{VALID}$ <i>to be checked</i>
After:	$\mathcal{J} \subseteq \text{VALID}$ $\text{VALID}_0 - \text{VALID} \subseteq \mathcal{E} - \mathcal{J}$

**Table 1.** Validity Contract: Constraints on the Validity Set

immediately before the code executes, that all objects in  $\mathcal{J}$  must be valid. During execution, only those objects in  $\mathcal{E}$  may become invalid, and so the remainder,  $\mathcal{J} - \mathcal{E}$  must still be valid. At the end of execution, we have a proof obligation for the code: the validity of the *critical set* of objects  $\mathcal{J} \cap \mathcal{E}$  must be checked. Then, immediately after execution, we have ensured that all objects in  $\mathcal{J}$  are still valid, and that only objects in  $\mathcal{E} - \mathcal{J}$  may have been invalidated. These constraints will be the basis for the rules for subcontracting, coming up next.

We have chosen our model to be flow insensitive for simplicity. It is indeed possible to provide a stronger validity contract in which we provide separate pre- and post-conditions for validity. For consistency, the post-condition would need to be stronger than the disjunction of the pre-condition and the validity effect. We leave the pursuit of this more general form of validity contract for another time.

## 2.2 Validity Subcontract for Nested Behaviours

With structured code, we can nest behaviours in various ways, such as by making method calls, or entering nested blocks. From a caller’s perspective, we can also think of a method override as being a nested version of the overridden method. Method calls must respect the validity contract of their calling context; nested code blocks may have their own contract, but must respect the contract of their containing block; and the contract for a method override must be consistent with the contract for the method being overridden. All of these nested behaviours must conform to the contract for the surrounding behaviour in the same way.

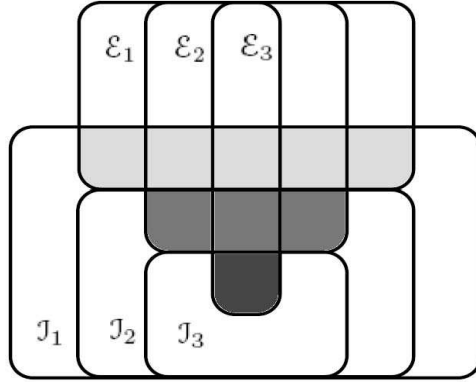
In Design by Contract [22] there is a notion of subcontracting for subclasses, and in particular, for overridden methods. The subcontract rules (preconditions may only be weakened, postconditions may only be strengthened) provide a guarantee that the overriding behaviour conforms to the contract of the overridden method, which allows clients to reason about method calls without worrying

about whether the method has been overridden in a subclass. In formal specification this notion corresponds to that of operation refinement. We adopt a similar approach for validity subcontracts, but we take into account the constraints on validity imposed by a contract, as formulated above.

Within the context of a particular validity contract  $\langle \mathcal{J}, \mathcal{E} \rangle$ , suppose we execute code that is known to meet another validity contract  $\langle \mathcal{J}', \mathcal{E}' \rangle$ . According to the constraints of Table 1, on entry to  $\langle \mathcal{J}', \mathcal{E}' \rangle$  we require  $\mathcal{J}' \subseteq \text{VALID}$ , but we know that during the execution of  $\langle \mathcal{J}, \mathcal{E} \rangle$  that  $\mathcal{J} - \mathcal{E} \subseteq \text{VALID}$ . We therefore require that  $\mathcal{J}' \subseteq \mathcal{J} - \mathcal{E}$ . From the perspective of  $\langle \mathcal{J}, \mathcal{E} \rangle$ , we do not care what becomes invalid during execution of  $\langle \mathcal{J}', \mathcal{E}' \rangle$ , provided that it exits satisfying the constraints imposed by  $\langle \mathcal{J}, \mathcal{E} \rangle$ . When  $\langle \mathcal{J}', \mathcal{E}' \rangle$  exits, we know that the only objects to have been invalidated by its execution,  $\text{VALID}'_0 - \text{VALID}$ , lie within  $\mathcal{E}' - \mathcal{J}'$ . However we know that before  $\langle \mathcal{J}', \mathcal{E}' \rangle$  executes, that  $\text{VALID}_0 - \text{VALID}'_0$  is within  $\mathcal{E}$ . It follows that  $\text{VALID}_0 - \text{VALID}$  will be within  $\mathcal{E}$  as required, provided  $\mathcal{E}' - \mathcal{J}' \subseteq \mathcal{E}$ . This reasoning leads us to the following definition for a *validity subcontract*. Note that the definition makes no mention of the validity set  $\text{VALID}$ . The subcontract definition simply relates two different validity contracts, irrespective of what the validity set may be.

**Definition 1 (Validity Subcontract)**  $\langle \mathcal{J}', \mathcal{E}' \rangle$  is a validity subcontract of  $\langle \mathcal{J}, \mathcal{E} \rangle$  if:

$$\begin{aligned} \mathcal{J}' &\subseteq \mathcal{J} - \mathcal{E} \\ \mathcal{E}' - \mathcal{J}' &\subseteq \mathcal{E} \end{aligned}$$



**Fig. 1.** Validity Subcontracts: The Relationship between Nested Contracts. Validity of objects in  $\mathcal{E}_i \cap \mathcal{J}_i$  (shaded) must be separately established.

Figure 1 captures the nesting properties of the validity invariants and effects, for subcontracts. Here  $\langle \mathcal{J}_3, \mathcal{E}_3 \rangle$  is a subcontract of  $\langle \mathcal{J}_2, \mathcal{E}_2 \rangle$ , which is, in turn, a subcontract of  $\langle \mathcal{J}_1, \mathcal{E}_1 \rangle$ . Intuitively, we think of this in terms of nested method calls. The validity invariant is weakened, the deeper the call structure. On successive calls it is safe to reduce the size of the validity effect. However, most

interesting is where the effect may be increased. This critical set is the overlap of the validity invariant and effect for the new call. It is precisely this critical set of objects whose validity must be re-established at the end of the call.

Before introducing our language Oval, we will discuss the relationship between the validity set and object invariants for our general model.

### 2.3 Object Invariants and Dependency Structure

In abstract terms an object's invariant captures the commonality between the pre- and post-conditions for all of the object's methods. By specifying an object invariant, we effectively constrain the allowable behaviours of the object, thereby defining the safe or consistent states of the object. An object's invariant ultimately resolves to constraints on fields, not only of the current object, but also, possibly, of other objects. It is therefore possible for an object's invariant to be satisfied, even though it depends on another object whose invariant does not hold. To avoid this worrisome situation, we refine the idea of object validity.

**Definition 2 (Object Dependency)** *An object (transitively) depends on another if its invariant depends directly, or indirectly, on the fields of the other object. We write  $\succsim$  for this preorder between objects.*

**Definition 3 (Object Validity)** *An object is valid if its invariant is satisfied and if all of the objects that it depends on are also valid. The validity set VALID is the largest set of objects that are valid.*

Because the definition of *valid object* is recursive, we have specified the validity set to be the largest possible set of objects satisfying the constraint.

It is possible to have objects whose invariant holds, but that are not valid by our definition. For us, the important point is that from a system-wide perspective, validity is a closed property. The validity of an object only depends on its own state, and the validity of other objects that its invariant depends on.

**Property 1 (Validity Closure)** *If  $o \in \text{VALID}$  and  $o \succsim o'$  then  $o' \in \text{VALID}$ .*

This asserts that the validity set is down-closed with respect to the object dependency induced by the invariants. Consequently in specifying validity contracts, it makes sense to insist that the validity invariant  $\mathcal{J}$  (valid objects) is down-closed, and that the validity effect  $\mathcal{E}$  (possibly invalid objects) is up-closed with respect to object dependency. Figure 1 was drawn to illustrate the closure properties of  $\mathcal{J}$  and  $\mathcal{E}$ , assuming that object dependency is downwards in the diagram.

We now begin to see some hope of how to achieve a syntactic representation of validity contracts in programming languages. Assuming we can explicitly list the objects at the top of  $\mathcal{J}$  and the bottom of  $\mathcal{E}$ , and assuming we can capture the object dependency, then we should be able to reason about validity contracts syntactically. In Section 3 we do so by using ownership types to constrain the object dependency, and specifying the extrema of  $\mathcal{J}$  and  $\mathcal{E}$  as single objects, that is, as ownership contexts I and E respectively.

### 3 Oval: A Language with Ownership-Based Validity

Oval is a simple object-oriented programming language that incorporates ownership types to support validity contracts. Oval realises our general model by using object ownership to provide the structure for specifying validity invariants and effects. Oval presumes that object dependency (for invariants) is constrained by object ownership.

#### 3.1 Ownership Types and Effect Encapsulation

Object encapsulation techniques hide an object’s internal representation and force any access from outside to be made via the object’s interface. Ownership types [29, 13, 12, 10] improve on previous work on full encapsulation techniques [15, 2] by allowing outgoing references while still preventing representation exposure; objects cannot be referenced from outside of the encapsulation.

In ownership type systems, each object has one fixed owning object, given at the time of creation. The ownership relation partitions objects into an *ownership tree*, whose root is a conceptual object called **top** in this paper. Objects are confined so that they can only reference or access objects owned by their (transitive and reflexive) owners.

Ownership type systems are essentially access control systems where accessibility is determined by the position of the target object in the ownership tree. Such systems achieve a form of information hiding. However, the problem of maintaining object validity is fundamentally a problem of effects: only updates can break object invariants. In other words, read access can always be considered safe. Our earlier work on *effect encapsulation* [20] has provided a technique for encapsulating side-effects on an object without restricting referenceability or read access. Our work in this paper provides a significant extension to effect encapsulation; we allow each method to specify the validity invariant, that is, what object invariants can be expected to hold. By tracking permissible updates via the validity effect, the type system checks if each expected validity invariant can be met.

#### 3.2 Overview of Oval

In Oval, we use ownership types to structure objects, specify validity contracts and confine dependencies for object invariants.

Oval programs specify a validity invariant and effect for each method as a *validity contract*,  $\langle I, E \rangle$ .  $I$  and  $E$  are ownership contexts in our type system. The corresponding validity invariant  $\mathcal{J}$  is the context  $I$  and all contexts owned (transitively) by  $I$ ; the corresponding effect  $\mathcal{E}$  is the context  $E$  and all owning contexts of  $E$ ; in the ownership order,  $I$  is the top of the down-set  $\mathcal{J}$ , and  $E$  is the bottom of the up-set  $\mathcal{E}$  (which is a branch of the ownership tree). The only overlap allowed for  $\mathcal{J}$  and  $\mathcal{E}$  is the local context **this**. This yields just two interesting cases. When the validity invariant and effect are disjoint, the critical set is empty, and  $I$  must be strictly within  $E$  in the ownership order. Otherwise, when they overlap,

they must satisfy  $I = E = \text{this}$ ; the critical set just contains the current object. As we shall see, the type system also checks that the subcontract rule is satisfied by any method calls. Consequently when the critical set is empty, there is no further reasoning about object validity required in the current method body. When the critical set is a single object, only the validity of the current object needs to be re-established at the end of the method body. By incorporating the idea of validity contracts in the type system, Oval provides a frame for localized reasoning about object validity.

The constraint that  $I$  and  $E$  can only overlap at the current object, allows us to keep both the syntax and type system simple in Oval. Allowing a bigger overlap is feasible: it is a trade-off between greater expressiveness and increased complexity. The added complexity arises from syntactic issues (describing sets of contexts rather than singletons), the type system (the static dependency ordering is trickier), and reasoning for re-establishment of invariants with multiple objects simultaneously. These sorts of extensions have been made for `Spec#` where objects may have peers, and whole peer sets can be simultaneously invalidated.

An object's invariant may only depend on its member fields and fields of (transitively) owned objects. For expressiveness, we allow fields that are not part of the object's representation to be declared `nonrep`. `nonrep` fields cannot be mentioned in the class invariant. Fields which are not explicitly declared `nonrep` are part of the object's representation and may participate in its object invariant. Some other languages have a similar design, for instance, Javari [5] uses the keyword `mutable` to annotate those `nonrep` fields. Because there is no object ownership structure in Javari, object invariants can never depend on `mutable` fields. In our language, we do allow the owning object's invariant to depend on any fields of owned objects (including `nonrep` fields). An object's invariant can only depend on its `rep` fields and objects that it (transitively) owns. So updates on `rep` fields may affect the target's invariant, and so (in Oval) should only be allowed by the target's own methods. Updates on `nonrep` fields can only affect an owner's invariant, and that owner must already have an active method that is responsible for re-establishing the owner's invariant.

A constructor must always establish its object's invariant. For simplicity, we consider constructors to have no effect on validity—they simply initialize the object in a valid state.

In addition to the usual `top` context, we introduce another context `bot`. The `bot` context is the bottom of the ownership hierarchy; it is inside/owned by all other contexts. We syntactically restrict its use in Oval to the validity invariant part of a contract. When a method's validity invariant is `bot`, it neither requires nor ensures any object validity; it provides a vacuous constraint for the validity set.

### 3.3 Oval by Example

The validity contract allows us to solve the re-entrancy problem discussed in the Introduction section.

```

class C {
  int a, b;
  invariant: 0 <= a < b;
  C() { a = 0; b = 3; }
  void m() <this, this> { int k = 100/(b-a);
                        a = a+3; P(...); b = (k+4)*b; }
  void n() <this, top> { int k = 100/(b-a);
                       Q(...); } }

```

In the above example, the validity contract for method `m` is `<this, this>`, which means the method requires the current object (`this`) to be valid and the method is allowed to break `this`'s invariant via either direct assignment or indirect method calls. What is interesting here is that our type system guarantees `P` cannot callback on `m`. In order for `P` to callback on `m`, `P` must meet the required validity invariant—`this` must be valid. However, object `this` is invalid in `P` because the validity effect of `m` has invalidated `this`.

On the other hand, the contract for method `n` is `<this, top>`. It does expect the current object (`this`) to be valid, but it does not invalidate any object except the conceptual context `top`. This time, `Q` may callback on `n`, because `this` is valid in `Q` and so it will be able to satisfy the required validity invariant of `n` on re-entry.

A method can require a validity invariant other than its target object (`this`), as in the following code using class `C` from above.

```

class D { nonrep int f; ... }

class G [p, q] where p < q {
  void ie(C<p> x, D<q> y) <p, q> { y.f = 100/(x.b-x.a); } }

```

Every class has a formal parameter which refers to the owner of the current object. By default, when the owner parameter is not used by the class definition, it may be omitted, as for class `D`. Moreover, since field `f` is declared as `nonrep`, updates on field `f` will not break the invariant of class `D`, because the local invariant cannot depend on its `nonrep` fields. But updates on `f` may break the invariant of its owning object, whose invariant is allowed to depend on any field of owned objects. In fact, in our type system, fields not declared to be `nonrep` can only be updated by the object itself.

As part of its validity contract, method `ie` states that `p` is the validity invariant; since the parameter `x` is owned by `p`, `x` must be valid, that is, its invariant must hold—so we know there will be no divide-by-zero error. Note that, to reason in this way, the programmer needs to know class `C`'s invariant—`C`'s invariant needs to be *visible*. We will discuss this in Section 5.

The method contract of `ie` specifies its validity effect as `q`; this permits `q` to be invalidated. Since the object in `y` is owned by `q`; assignment on `y` may invalidate `q` and any objects owning `q`; this is allowed by the method's validity effect.

To illustrate a more realistic use of our type system, we adopt an example from a tutorial on Spec# [31].

```

class Person {
    int freeday;
    Meeting<this> next;
    invariant: if next!=null then next.day!=freeday;
    int travel(int d) <this, this> {
        freeday = d;
        if (next!=null) next.reschedule((d+1)%7); } }

class Meeting {
    int day;
    invariant: 0 <= day < 7;
    Meeting(int d) { day = d; }
    void reschedule(int d) <this, this> { day = d; } }

```

The invariant of `Person` depends on the `day` field of the `Meeting` object `next`. Consequently, if the `day` field of the `Meeting` object is changed, the invariant of the `Person` may be broken. Ownership types capture this kind of dependency by enforcing the `Meeting` object to be owned by `Person`—the owner argument of `next` is `this` which refers to the current `Person` object.

Both the validity invariant and effect of the method `reschedule` in class `Meeting` are the current object `this`. The method is allowed to update its `day` field, and it will require and ensure its local invariant holds before and after the method call.

Back to the `travel` method in class `Person` whose validity contract is also `<this, this>`. The method is allowed to call `next.reschedule` because it can meet `reschedule`'s validity invariant requirement, and `next.reschedule` can meet `travel`'s requirement for validity effect. Method `next.reschedule` requires `next` to be valid before calling it; within the method `travel` the local invariant of the `Person` object may be violated, but all objects owned by the `Person` object, including the object in `next`, are valid. On the other hand, method `travel`'s validity effect insists that only objects (transitively and reflexively) owning `this` may be invalidated within its body, more precisely for this case, it insists that the object in `next` must be valid after the call to `next.reschedule`. The validity invariant on method `next.reschedule` ensures that the object in `next` is validated after the call.

Linked lists are a popular example for illustrating language features dealing with restricted use of references. Ownership type systems are known to be inadequate for expressing cross-encapsulation references such as needed by iterators, because of their restrictive reference containment. We show how iterators are expressed in our language, explaining how our type system is able to relax the usual ownership restriction on reference containment while still keeping the iterators safe.

```

class List[o, d] {
    Node<this, d> head;
    void insert(Data<d> data) <this, this> { head.insert(data); }
    Iterator<o, this, d> getIterator() <bot, top> {
        return new Iterator<o, this, d>(head); } }

```

```

class Node[l, d] {
  nonrep Data<d> data;          nonrep Node<l, d> next;
  Node(Node<l, d> next, Data<d> data) {
    this.next = next;          this.data = data; }
  void insert(Data<d> data) <bot, l> {
    next = new Node<l, d>(next, data); } }

class Iterator[o, l, d] {
  nonrep Node<l, d> current;
  Iterator(Node<l, d> node) { current = node; }
  void next() <bot, o> { current = current.next; }
  int element() <bot, top> { return current.data; }
  void insert(Data<d> data) <bot, l> { current.insert(data); } }

// client code
List<o, d> list = ...;  Data<d> data = ...;  list.insert(data);
Iterator<o, *, d> iterator = list.getIterator();
while iterator.element() {
  int data = iterator.element();  iterator.next(); }
list.insert(data) // OK
iterator.insert(data); // error, effect is unknown in current context
Node<*, d> node = list.head; // OK, * can abstract any context
node.next = ... // error, owner of node.next is hidden

```

The `List` class provides methods for adding new elements to the list and for returning iterators for the list. The implementation of the `List` class is almost the same as for ownership types except the methods have been annotated with a validity contract. Method `insert` will update the current object and can be called by clients, so its validity invariant and effect are the current object. Method `getIterator`'s validity contract is `<bot, top>`; it makes no assumptions about validity (by the `bot` invariant), and does not invalidate any object (by the `top` effect).

In the `Node` class, the field `next` is declared to be `nonrep` so that it is part of its owner's representation (the owner being the list, in this example). We can allow the field `next` to be part of the node's representation if the programmer wants that—this is an explicit design choice: if the field is part of the object's representation, then the iterator will not be able to insert objects into the list, because it will not be able to specify the required effect on the node as part of its validity contract.

The field `current` in class `Iterator` may refer to the node objects owned by list (note that the class parameter `l` is bound to the list object). The method `insert` may add new elements into the list. The method `next` allows the iterator to step to the next link in the list object. Method `element` will return the data at the current position of traversal. None of these methods constrain the validity, because the iterator has no internal representation: its fields are owned by the list, and any invariant will be associated with the list rather than the iterator.

### 3.4 Abstract Contexts and Mutability from Top to Bot

In the list example above, the client can give the correct types for list and data objects, but it cannot give the precise type for iterators. This is because the second formal parameter of the iterators is bound to the internal context of the list, which cannot be named from the client outside. To solve this problem, we use the context abstraction techniques introduced in our earlier papers [20, 19] to abstract the name of list's internal context, so that the client can declare the correct type to hold a reference to the iterators. This is how our type system frees programs from the restrictive reference containment enforced by ownership type system. We allow arbitrary reference structure, yet, we ensure safety of these references. In this example, the client cannot use the iterators to insert elements into the list, because this will mutate and break the list's invariant. In order for the client to insert new elements, it must use the list's interface which allows the list's local invariant to be re-established by itself. The last two lines of the client code illustrate that the nodes of the list can be referenced by the external client code, but they cannot directly update the nodes because they cannot specify a concrete context for the owner of the nodes' fields (the list in this case).

However, our iterators are *not* read-only references. Iterators are able to return `Data` objects which can be mutated by the client. This is very different to previous read-only systems which use read-only references to allow iterators to cross the boundary of object encapsulation. More precisely, read-only references can only call pure (side-effect free) methods and return references in read-only mode. For instance, the `Data` objects returned by their read-only iterators cannot be mutated.

Moreover, our iterators are able to mutate the list's representation in its `insert` method depending on the caller's context. In this example, the client sits outside of the list context so it cannot use iterators to mutate the list. But objects inside the list may call `insert` on iterators to modify the list. In this sense, all references in our language are context-aware, they can be used to read anything, but write capability depends on the contexts where the references are held.

Oval provides the ability to create immutable objects explicitly. Immutable objects cannot be mutated after initialization. In Oval, immutable objects are those objects whose owner is the special context `bot`. By declaring an object to have `bot` as owner, the system allows any other object's invariant to depend on this object. Consequently, mutation of an object owned by `bot` could break the invariants of those objects. This is disallowed by prohibiting `bot` from being used as a validity effect. No well-formed method is capable of mutating objects owned by `bot`—effectively they become immutable.

There are some notable special cases for validity contracts. The validity contract for the main method is  $\langle \text{top}, \text{top} \rangle$ . Validity invariant `top` means all objects in the heap are valid. At the beginning of the main method (before execution of the program), there is no object in the heap. At the end of the main method (after execution of the program), all objects created by the program must be valid.

Another special contract is  $\langle \text{bot}, \text{top} \rangle$ . Methods with a  $\langle \text{bot}, \text{top} \rangle$  contract do not require any objects to be valid, and do not break any validity that objects may have. By the subcontracting rule, any calls within such methods must obey the same contract. The  $\langle \text{bot}, \text{top} \rangle$  contract can be used to default the validity contract of methods in legacy code such as Java programs; any callbacks from legacy code must be restricted to  $\langle \text{bot}, \text{top} \rangle$  methods.

## 4 The Oval Type System

We present a type system for a core part of the Oval language to demonstrate that the general object validity model of Section 2 is realizable within a type system. The Oval type system is based on our earlier effective ownership type system [20]. There, methods are declared with an effective owner,  $E$ , which serves two purposes: it determines the write capability of the method body, and it restricts access to call contexts where  $E$  is known. In Oval, we replace the effective owner,  $E$ , with a validity contract  $\langle I, E \rangle$ . The Oval type system adopts our earlier scheme [20, 19], based on abstraction of ownership contexts, which provides for liberal read access to owned objects in an ownership type systems.

The most important rules of the Oval type system are field assignment and method call. Method call is constrained by the subcontracting rules of Section 2.2. We do not formalize the expression of object invariants and their dependence on fields; informally, we simply require that object invariants only depend on the object’s owned fields (those declared without `nonrep`) and on fields of other objects that it (transitively) owns. However we do formalize object validity by presenting a dynamic semantics which tracks a set  $\Sigma$  of valid objects, assuming that the object invariant is re-established for critical method calls (where  $I = E = \text{this}$ ).

### 4.1 Syntax and Static Semantics

The abstract syntax is given in Table 2.  $X$  ranges over formal context parameters; and  $x$  ranges over variable names including `this` used to reference the target object for the current call. Note that `this` is also used as a context. The overbar is used for a sequence of constructs; for example,  $\bar{e}$  is used for a possibly empty sequence  $e_1..e_n$ ,  $\bar{T} \bar{x}$  stands for a possibly empty sequence of pairs  $T_1 x_1..T_n x_n$ .

The first context argument of a type determines the actual owner of objects of the type. The abstract context `*` is used to hide context arguments when they are not nameable. We do not distinguish syntactically between concrete and abstract contexts in forming types. However such distinctions are important in our type rules. For example, the object creation type in a `new` expression must be concrete.

Programs consist of a collection of classes with an expression to be evaluated. Classes are parameterized with context parameters having some assumed ordering expressed by the where clause. The first formal context parameter of a class

$T$	$::= C(\overline{K})$	types
$K, I, E$	$::= X \mid \text{this} \mid \text{top} \mid \text{bot} \mid *$	contexts
$P$	$::= \overline{L} e$	programs
$L$	$::= \text{class } C[\overline{X}] \triangleleft T \text{ where } \overline{X} \prec \overline{X} \{ \overline{F}; \overline{M} \}$	classes
$F$	$::= [\text{nonrep}]_{\text{opt}} T f$	fields
$M$	$::= T m(\overline{T} \overline{x}) \langle I, E \rangle \{e\}$	methods
$e$	$::=$	terms
	$x$	variable
	$\mid \text{new } T(\overline{e})$	new
	$\mid e.f$	select
	$\mid e.m(\overline{e})$	call
	$\mid e.f = e$	assignment

**Table 2.** Abstract Syntax for the Oval Source Language

$K, I, E$	$::= \dots \mid ? \mid K_{\text{rep}}$	contexts
$\Gamma$	$::= \bullet \mid \Gamma, X \prec X \mid \Gamma, x : T$	environments
$S$	$::= \langle K, I, E \rangle$	stack frames

**Table 3.** Extended Syntax for Oval’s Type Rules

determines the owner of `this` object within the class. Like the original type systems, the owner of an object is determined by its creation type, and is fixed for the lifetime of the object. Unlike the original ownership type systems we do not insist that the owner is within the other formal parameters of the class, because we do not use ownership to constrain object references. In the class production, inheritance  $\triangleleft T$  is optional because our type system does not need a top type. Each field may optionally be declared to be `nonrep`, implying that the object’s own invariant should *not* depend on it. Each method defines a pair of contexts  $\langle I, E \rangle$  which specify the validity contract. We interpret the validity invariant as the down-set of  $I$  in the ownership ordering of contexts, and the validity effect is the up-set of  $E$ . The actual validity contract for a method call is determined by the context binding for the type of the target object (see the lookup [LKP-DEF] in Table 8). Our term syntax is kept as simple as possible; our interest is focused on field assignment and method call—the operations which can make changes, direct or indirect, to objects.

Table 3 extends the abstract syntax for use in the type system. This syntax is not expressible in programs. The context  $?$  denotes an existential context, that is, an unknown context. It replaces the general context  $*$  when a type is opened up in a dot expression for a field access or method call (see the [LKP-DEF] rule in Table 8). Different occurrences of the existential context cannot be compared, because they represent arbitrarily different, but unknown contexts. Similarly  $K_{\text{rep}}$  denotes an existential context whose owner is  $K$ . This allows us to expand the fields of a non-local object without needing to name it as a context (see the lookups [LKP-FLD-EXP], [LKP-INT’] rules in Table 8). Our earlier paper [19] has more detail on the topic of context abstractions and existential contexts.

Type environments  $\Gamma$  record the assumed ordering between context parameters, and the types of variables. Stack frames  $S$  keep track of the current active object and the validity contract for the active call. In the dynamic semantics  $K$

[TYP-OBJ]	$\frac{\text{defin}(C(\overline{K}), \_) = \text{class } C \dots \text{ where } \overline{K}' \prec \overline{K}'' \dots \\ * \notin \overline{K} \quad \Gamma \vdash \overline{K}' \prec \overline{K}''}{\Gamma \vdash_{\circ} C(\overline{K})}$
[TYPE]	$\frac{\Gamma \vdash_{\circ} T' \quad \vdash T' < T}{\Gamma \vdash T}$
[SUB-EXT]	$\frac{\text{class } C[\overline{X}] \triangleleft T' \dots \quad T = [\overline{K}/\overline{X}]T'}{\vdash C(\overline{K}) < T}$
[SUB-RFL]	$\overline{\vdash T < T}$
[SUB-TRA]	$\frac{\vdash T < T'' \quad \vdash T'' < T'}{\vdash T < T'}$
[BIN-ABS]	$\frac{\vdash \overline{K} \subseteq \overline{K}'}{\vdash C(\overline{K}) <: C(\overline{K}')}$
[BIN-SUB]	$\frac{\vdash T < T'' \quad \vdash T'' <: T'}{\vdash T <: T'}$

**Table 4.** Type, Subtype, and Binding Rules

corresponds to the current object location; in the static semantics,  $K$  is bound to this in method bodies.  $\langle I, E \rangle$  denotes the validity contract of the current method.

The type rules proper follow in the next four tables. In addition, Table 8 defines some auxiliary definitions to be used by the type system. We put substitutions and lookups in the auxiliary definitions to keep the type rules simple.

Table 4 provides rules for type well-formedness and subtyping rules for expressible types. We introduce a separate judgement for bindability to handle types with existential contexts (which only occur in the type system after field or method lookup). For expressible types, there is no difference between subtyping and bindability. However bindability is not reflexive, because existential types can only occur on the right-hand side of the binding relation.

Concrete object types are formed by substituting concrete contexts into class definitions as in [TYP-OBJ]. This rule explicitly excludes context abstraction. By [TYPE] well-formed expressible types are those that have a valid concrete subtype. The rules for subtyping are based on substitution in class inheritance [SUB-EXT], and by reflexive and transitive closure [SUB-RFL], [SUB-TRA]. The binding rules are governed by a combination of context abstraction [BIN-ABS] and subtyping [BIN-SUB].

Table 5 defines the context ordering for concrete contexts. It also defines the rules for context abstraction. [ABS-RFL] ensures that the existential contexts  $?$  and  $K_{\text{rep}}$  abstract nothing. Combined with the earlier bindability rule [BIN-ABS] this ensures that types with existential contexts cannot be associated with the target of a binding. Finally [SUBCONTRACT] describes rules for enforcing validity subcontracting, which correspond to the earlier subcontracting definition of Section 2.2. The fourth case is equivalent to  $\Gamma \vdash E \preceq E' \vee E = \text{owner}_{\Gamma}(E')$ . This highlights the fact that calls that affect the validity invariant can only be made from the owner context.

[ORD-ENV]	$\frac{K \prec K' \in \Gamma}{\Gamma \vdash K \prec K'}$
[ORD-OWN]	$\frac{K \neq \text{bot}}{\Gamma \vdash K \prec \text{owner}_\Gamma(K)}$
[ORD-BOT]	$\frac{K \neq \text{bot}}{\Gamma \vdash \text{bot} \prec K}$
[ORD-REP]	$\frac{}{\Gamma \vdash K_{\text{rep}} \prec K}$
[ORD-TRA]	$\frac{\Gamma \vdash K \prec K' \quad \Gamma \vdash K' \prec E}{\Gamma \vdash K \prec E}$
[ORD-REF]	$\frac{\Gamma \vdash K \prec K' \quad \text{or} \quad K = K'}{\Gamma \vdash K \preceq K'}$
[ABS-ANY]	$\frac{}{\vdash K \subseteq *}$
[ABS-RFL]	$\frac{K \neq ? \quad K \neq_{\text{-rep}}}{\vdash K \subseteq K}$
[SUBCONTRACT]	$\frac{\begin{array}{l} \Gamma \vdash I \prec E \implies \Gamma \vdash I' \preceq I \\ \Gamma \vdash I' \prec E' \implies \Gamma \vdash E \preceq E' \\ I = E \implies \Gamma \vdash I' \prec I \\ I' = E' \implies \Gamma \vdash E \preceq \text{owner}_\Gamma(E') \end{array}}{\Gamma; \langle K, I, E \rangle \vdash \langle I', E' \rangle}$

**Table 5.** Context Ordering, Abstraction and Subcontracting Rules

The rules for well-formed program definitions and declarations are in Table 6. The main expression has an empty environment, and uses the `top` context to express the validity contract. A valid class must have valid fields and methods, assuming the given context ordering. The `[METHOD]` checks the validity contract is in one of the two correct forms, where the critical set is just `this` or is empty, and checks the method body within the appropriate stack frame. If the method is overriding a superclass method, its validity contract must be a subcontract of its parent.

Table 7 defines expression types. `[EXP-NEW]` requires that the object creation type is a valid object type, and that the constructor arguments are bindable. Field assignment `[EXP-ASS]` requires that the assigned expression type be bindable to the field type. If the field type involves a hidden context, this is not possible. For default field accessibility, the target expression `e` must be `this`, and it must be allowed to break its validity. For nonrep fields, the validity of the owner of the target expression must be able to be modified. Method call `[EXP-CAL]` is governed by the validity subcontracting rule; note that this check applies in the context of the call, so the same call may be allowed in some calling contexts and not in others. This contributes to the context sensitivity of the type system.

## 4.2 Dynamic Semantics and Properties

To formulate reduction rules, in Table 9 we extend the syntax of terms and contexts with typed object locations; they provide the runtime context bindings

[PROGRAM]	$\frac{\vdash \bar{L} \quad \bullet; \langle \text{top}, \text{top}, \text{top} \rangle \vdash e : T}{\vdash \bar{L} e}$
[CLASS]	$\frac{\Gamma = \bar{X}' \prec \bar{X}'', \text{this} : C(\bar{X}), \text{super} : T \quad \Gamma \vdash T \quad \Gamma \vdash \bar{M} \quad \Gamma \vdash \bar{F} \quad \text{owner}_{\Gamma}(\text{this}) = \text{owner}_{\Gamma}(\text{super}) \quad \text{range}(\bar{F}) \cap \text{dom}(\text{fields}(T, \text{this})) = \bullet}{\vdash \text{class } C[X] \triangleleft T \text{ where } \bar{X}' \prec \bar{X}'' \{ \bar{F}; \bar{M} \}}$
[METHOD]	$\frac{S = \langle \text{this}, I, E \rangle \quad \Gamma, \bar{x} : \bar{T}; S \vdash e : T'' \quad \Gamma \vdash T, \bar{T} \quad \vdash T'' \prec T \quad I = E = \text{this} \text{ or } \Gamma \vdash I \prec E \quad \text{method}(\Gamma(\text{super}), \text{this}, m) = T' \ m(\bar{T}' \_) \langle I', E' \rangle \dots \implies \quad \vdash T \prec T' \quad \vdash \bar{T}' \prec \bar{T} \quad \Gamma; S \vdash \langle I', E' \rangle}{\Gamma \vdash T \ m(\bar{T} \bar{x}) \langle I, E \rangle \{e\}}$
[FIELD]	$\frac{\Gamma \vdash T}{\Gamma \vdash [\text{nonrep}]_{\text{opt}} T \ f}$

**Table 6.** Program, Class, Method and Field Rules

[EXP-VAR]	$\frac{\Gamma(x) = T}{\Gamma; S \vdash x : T}$
[EXP-NEW]	$\frac{\Gamma \vdash_{\circ} T \quad \text{fields}(T, ?) = \_ \bar{T} \quad \Gamma; S \vdash \bar{e} : \bar{T}' \quad \vdash \bar{T}' \prec \bar{T}}{\Gamma; S \vdash \text{new } T(\bar{e}) : T}$
[EXP-SEL]	$\frac{\text{fields}_S^{\Gamma}(e)(f) = T}{\Gamma; S \vdash e.f : T}$
[EXP-ASS]	$\frac{\Gamma; S \vdash e.f : T \quad \Gamma; S \vdash e' : T' \quad \vdash T' \prec T \quad f \notin \text{nrfields}_S^{\Gamma}(e) \implies \Gamma; S \vdash \langle \text{bot}, e \rangle \wedge S = \langle e, \dots \rangle \quad f \in \text{nrfields}_S^{\Gamma}(e) \implies \Gamma; S \vdash \langle \text{bot}, \text{owner}_{\Gamma}(e) \rangle}{\Gamma; S \vdash e.f = e' : T'}$
[EXP-CAL]	$\frac{\text{method}_S^{\Gamma}(e, m) = T \ m(I, E)(\bar{T} \_) \quad \Gamma; S \vdash \bar{e} : \bar{T}' \quad \vdash \bar{T}' \prec \bar{T} \quad \Gamma; S \vdash \langle I, E \rangle}{\Gamma; S \vdash e.m(\bar{e}) : T}$

**Table 7.** Typing Rules for Expressions

which serve to structure the heap. Objects are modeled as mappings from fields to locations (for convenience, the object type is encoded in the object's location rather than in the object itself). The heap maps locations to objects.

Amongst the extra definitions in Table 10, note that the well-formedness of the validity set requires that it contains all objects within the validity invariant  $I$ , and  $I$  itself when it is distinct from the validity effect  $E$ .

Table 11 presents a big-step reduction semantics. It is of particular interest to trace how the validity set is affected by the reductions, depending on the validity contract for method call, and how default and nonrep field assignment differ by either invalidating the target of the call or its owner.

Finally, we present some of the key properties of the type system. The main result *validity preservation* is stated together with the conventional *type preservation* property in Theorem 1. The first line of the *if-then* block is the validity invariant property; and the second line of the *if-then* is the type preservation property. Corollary 1 states that all objects created by the program must be valid at the end of the execution.

[LKP-DEF]	$\frac{L = \text{class } C[\bar{X}] \dots \quad \bar{K}' = [?/*]\bar{K}}{\text{defin}(C\langle\bar{K}\rangle, K) = [\bar{K}'/\bar{X}, K/\text{this}]L}$
[LKP-FLD]	$\frac{\text{defin}(T, K) = \text{class } \dots \triangleleft T' \{[\text{nonrep}]_{\text{opt}} \bar{T} \bar{f}; \dots\}}{\text{fields}(T, K) = \bar{f} \bar{T}, \text{fields}(T', K)}$
[LKP-FLD-EXP]	$\frac{\Gamma; S \vdash e : T}{\text{fields}_S^T(e) = \text{fields}(T, \text{internal}_S^T(e))}$
[LKP-MUT]	$\frac{\text{class } C \dots \triangleleft T \{ \dots \text{nonrep } \bar{T} \bar{f} \dots; \dots \}}{\text{nrfields}(C\langle..\rangle) = \bar{f}, \text{nrfields}(T)}$
[LKP-MUT-EXP]	$\frac{\Gamma; S \vdash e : T}{\text{nrfields}_S^T(e) = \text{nrfields}(T)}$
[LKP-MTH]	$\frac{\text{defin}(T, K) = \text{class } \dots T' m(\bar{T} \bar{x})\{e\} \dots}{\text{method}(T, K, m) = T' m(\bar{T} \bar{x})\{e\}}$
[LKP-MTH']	$\frac{\text{defin}(T, K) = \text{class } \dots \triangleleft T' \{ \dots; \bar{M} \} \quad m \notin \bar{M}}{\text{method}(T, K, m) = \text{method}(T', K, m)}$
[LKP-MTH-EXP]	$\frac{\Gamma; S \vdash e : T}{\text{method}_S^T(e, m) = \text{method}(T, \text{internal}_S^T(e), m)}$
[LKP-OWN]	$\frac{\Gamma; \bullet \vdash e : T}{\text{owner}_T(e) = \text{owner}(T)}$
[LKP-OWN']	$\frac{}{\text{owner}(C\langle K, .. \rangle) = K}$
[LKP-INT]	$\frac{S = \langle K, I, E \rangle}{\text{internal}_S^T(K) = K}$
[LKP-INT']	$\frac{S = \langle K, I, E \rangle \quad e \neq K \quad K' = \text{owner}(T)}{\text{internal}_S^T(e) = K'_{\text{rep}}}$

**Table 8.** Auxiliary Definitions for Typing Rules

**Theorem 1 (Validity Preservation and Type Preservation)** *Given  $\vdash P$ ,  $\vdash H$  and  $H \vdash S$ ,*

*if  $\left\{ \begin{array}{l} H; S \vdash \Sigma \\ \bullet; S \vdash e : T \end{array} \right.$  and  $H; \Sigma; e \Downarrow_S H'; \Sigma'; l$ , then  $\left\{ \begin{array}{l} H'; S \vdash \Sigma' \\ \bullet; S \vdash l : T', \vdash T' <: T \text{ and } \vdash H' \end{array} \right.$*

**Proof.** The proof proceeds by induction on the form of  $H; \Sigma; e \Downarrow_S H'; \Sigma'; l$ . Due to the limited space, we only sketch the proof for validity preservation. Let  $S = \langle K, I, E \rangle$ . Case [RED-SEL] is trivial. Case [RED-NEW]: we show that the newly created object is valid by the [OBJECT] rule, then by the [VALIDITY SET] rule we have the result. Case [RED-ASS]: we need to show assignments cannot invalidate objects in  $\Sigma$ , by the [EXP-ASS] and the [SUBCONTRACT] rules we have that the object invalidated by the assignment is equal or outside of  $E$ . Since  $H \vdash S$ , and by the [STACKFRAME] rule we have  $I$  is inside or equal to the invalidated object, depending on whether or not  $I$  and  $E$  are the same. Then by the [OBJECT] and [VALIDITY SET] rules, we have the result. Case [RED-CAL]: as for the [RED-ASS] case, except the target object of the call may be invalidated during the call and is re-validated after the call.

**Corollary 1 (System Validity)** *Given  $\vdash P$ , if  $P \Downarrow H; \Sigma; l$  then  $\Sigma = \text{dom}(H)$ .*

$l, l_T$		typed locations
$e$	$::= \dots \mid l$	terms
$K, I, E$	$::= \dots \mid l$	contexts
$o$	$::= \bar{f} \mapsto \bar{l}$	objects
$H$	$::= \bar{l} \mapsto \bar{o}$	heaps
$\Sigma$	$::= \text{bot}, \bar{l}$	valid objects

**Table 9.** Extended Syntax for Dynamic Semantics

[EXP-LOC]	$\overline{\Gamma; S \vdash l_T : T}$
[HEAP]	$\frac{\forall l \in \text{dom}(H) \cdot \bullet; \bullet \vdash l : T \quad H(l) = \bar{f} \mapsto \bar{l} \quad \text{fields}(T, l) = \bar{f} \bar{T} \quad \bullet; \bullet \vdash \bar{l} : \bar{T}' \quad \vdash \bar{T}' <: \bar{T}}{\vdash H}$
[STACKFRAME]	$\frac{\{K, I, E\} \subseteq \text{dom}(H) \cup \{\text{top}\} \quad \bullet \vdash I < E \text{ or } K = I = E}{H \vdash \langle K, I, E \rangle}$
[OBJECT]	$\frac{\bullet \vdash l' \preceq l \implies l' \in \Sigma}{\Sigma \vdash l}$
[VALIDITY SET]	$\frac{\Sigma \subseteq \text{dom}(H) \cup \{\text{top}\} \quad \bullet \vdash I < E \implies \Sigma \vdash I \quad I = E \implies \Sigma, I \vdash I}{H; \langle K, I, E \rangle \vdash \Sigma}$

**Table 10.** Auxiliary Definitions for Dynamic Features

**Proof.** After applying Theorem 1 to the [EXECUTION] rule, we have the result by the [VALIDITY SET] rule.

## 5 Discussion and Related Work

### 5.1 Boogie and Universes

Our model for object validity is general, and not tied to any particular language. The Oval language provides just one possible realization of our general model for validity contracts. It has been inspired, not only by our own attempt to use ownership types to reason about object invariants and effects [20], but by various work on the Boogie methodology [18] and Universes [25]. We now compare the Universes and Boogie approaches with our general model.

Universes uses a relatively simple ownership type system to arrange objects into layers. Object invariants may depend on objects in the same and inner layers. Read-only annotations are used to allow object dependencies between layers, without providing a general update capability. The layering of objects corresponds to our object preorder  $\succcurlyeq$ . The validity invariant should be interpreted as the set of layers below the currently active layer, and the validity effect corresponds to the current layer. In Universes, there is no attempt to track the (possibly) invalid call context  $E$ , and so its up-closure is not required.

The Boogie methodology, as manifested in `Spec#`, is based on a program logic, accompanied by a model for protecting the validity of object state, which dynamically tracks the ownership of objects. The basic model describes objects as either consistent or valid or neither. Consistent or valid objects must satisfy their own invariant, and may only own consistent objects. Within the scope of

[EXECUTION]	$\frac{\bullet; \text{bot}; e \Downarrow_{(\text{top}, \text{top}, \text{top})} H; \Sigma; l}{\bar{l} e \Downarrow H; \Sigma; l}$
[RED-CAL]	$\frac{\begin{array}{l} H; \Sigma; e \Downarrow_S H'; \Sigma'; l \quad H'; \Sigma'; \bar{e} \Downarrow_S H''; \Sigma''; \bar{l} \\ \text{method}(l, m) = \dots \langle I, E \rangle (- \bar{x}) \{e'\} \quad H''; \Sigma''; [\bar{l}/\bar{x}]e' \Downarrow_{\langle l, I, E \rangle} H'''; \Sigma'''; l' \\ l = I = E \implies \Sigma'''' = \Sigma''', l \quad \Gamma \vdash I \prec E \implies \Sigma'''' = \Sigma''' \end{array}}{H; \Sigma; e.m(\bar{e}) \Downarrow_S H'''; \Sigma''''; l'}$
[RED-NEW]	$\frac{H; \Sigma; \bar{e} \Downarrow_S H'; \Sigma'; \bar{l} \quad l_T \notin \text{dom}(H') \quad \text{fields}(T, l_T) = \bar{f} \_}{H; \Sigma; \text{new } T(\bar{e}) \Downarrow_S H', l_T \mapsto \{\bar{f} \mapsto \bar{l}\}; \Sigma', l_T; l_T}$
[RED-ASS]	$\frac{\begin{array}{l} H; \Sigma; e \Downarrow_S H'; \Sigma'; l \quad H'; \Sigma'; e' \Downarrow_S H''; \Sigma''; l' \\ f \notin \text{nrfields}(l) \implies \Sigma''' = \Sigma'' - l \\ f \in \text{nrfields}(l) \implies \Sigma''' = \Sigma'' - \text{owner}(l) \end{array}}{H; \Sigma; e.f = e' \Downarrow_S [l \mapsto H''(l)] [f \mapsto l'] H''; \Sigma'''; l'}$
[RED-SEL]	$\frac{H; \Sigma; e \Downarrow_S H'; \Sigma'; l}{H; \Sigma; e.f \Downarrow_S H'; \Sigma'; H'(l)(f)}$

**Table 11.** Reduction Rules

an object pack/unpack block, a valid object may be updated and its invariant may be broken within that scope. For us, the validity invariant corresponds to the set of valid or consistent objects, and the validity effect corresponds to those objects that have currently been unpacked. Interestingly, this model allows dynamic transfer of ownership. For us, this implies that the preorder  $\succcurlyeq$  is dynamic, so when reasoning about  $I$  and  $E$ , we need to allow for different closures as the ownership structure changes. This may require some extra consistency constraints, but we believe that our model for validity contracts as summarized in Table 1 is still applicable. This is worthy of further investigation. In general, program logics like Boogie increase the specification overhead significantly. We believe the use of an Oval-like type system can provide framing assumptions for more detailed specifications, thereby allowing those specifications to be more concise.

Ownership is useful for expressing partially ordered *one-to-many* dependency relation, that is, one object may depend on multiple other objects. This kind of dependency is particularly useful for enforcing locality in object-oriented programs to allow localized reasoning on object invariants. However, ownership is not enough for expressing cyclic, many-to-one or many-to-many dependency relations. For instance, in a double-linked list, a node object has two fields—a **previous** field to reference its predecessor node in the list and a **next** field to reference its successor. The invariant of such a node typically requires the **next** field in its predecessor and the **previous** field in its successor to reference back the node itself. As a result, the invariants of two adjacent nodes are mutual. Ownership cannot capture such an invariant precisely, instead it requires such an invariant to be maintained by the list (i.e., the owner of the nodes). This is a sound solution, but in practice, it may complicate specification and reasoning [18]. In order to verify the invariant of the list, one has to consider all node objects owned by the list, but in fact modification of one node can affect only its predecessor and successor nodes.

Universes and Boogies allow local invariants to depend on the states of objects which are not owned by the current object; those objects are typically required to be within the same owner or sufficiently unpacked. For example, a node object’s invariant may mention fields in its predecessor and successor nodes. This kind of invariant is called a *visibility-based invariant* [23, 18], because it requires an invariant to be visible in every method that might violate the invariant, typically restricted to be within a module or friend classes. Visibility-based invariants essentially allow programmers to trade the locality offered by ownership for flexibility in the object invariant. They need to check states which are not local, and consequently generate more complicated proof obligations. Our general model introduced in Section 2 is independent of the form of the invariant, whether ownership or visibility-based. The Oval language has only used an ownership-based invariant in demonstrating the general model. This results in several advantages including less specification overhead (only type annotations are needed) and significantly simplified proof obligations for verification. In the future, we may consider extending Oval to handle visibility-based invariants as well.

## 5.2 Ownership Type Systems

Ownership Types [29, 13, 12, 10] allow programmers to enforce a tree-based encapsulation by declaring owners of objects as part of their types. Traditional object-oriented programs offer no object structure, and can be considered as special case of ownership types where all objects are owned by a single universal context `top`.

*Effective ownership* [20] allows programmers to add contexts to methods as effect owners. It is an encapsulation-aware effect system which frees ownership types from reference constraint, i.e. it allows arbitrary reference structure, but still retains a strict encapsulation on object representation. The *effect encapsulation* property guarantees that any update to an object’s internal state must occur (directly or indirectly) via a call on a method of its owner. Ownership types can be generalized as a special case of effective ownership where all effect owners are the current context `this`.

The type system we present in this paper generalizes effective ownership by adding invariant constraints to methods. The effect owner used in effective ownership is indeed the validity effect we have used in our new type system. Effective ownership is a special case of this type system where all validity invariants are the special context `bot`. The actual formalization of effective ownership is slightly different, but we are able to encode them in our type system. Consequently, the *effect encapsulation* property is also true in our type system.

The original ownership types hide knowledge of the identity of an object outside itself so that it cannot be named from outside. This kind of naming restriction may limit the expressiveness of ownership types. In particular, ownership types are known to be unable to express some common design patterns such as iterators or callbacks, which typically need to cross the boundary of encapsulation.

There have been a number of proposals made to improve the expressiveness of ownership types. *JOE* [11] allows internal contexts to be named through read-only local variables so that internal representation can be accessed from outside. *Ownership Domains* [1] use a similar method where read-only fields (final fields in Java) are used to name internal domains (partitions of contexts) instead of read-only variables. The *inner class* solution [10, 9] allows inner classes to name the outer object directly. These proposals tend to break the strict encapsulation of ownership types, and do not support localized reasoning on object invariants.

### 5.3 Read-only Systems

Read-only systems [24, 26, 5] use read-only references to cross the boundary of encapsulation, called *observational representation exposure* in [5]. Read-only references are considered harmless because they are restricted—they can only call pure methods (methods with no side effect) and return other read-only references. For example, they are able to express iterators by using a read-only reference to access the internal implementation of the list object.

However, these iterators can only return data elements in read-only mode, that is, the elements stored in the list cannot be updated in this way (unless using dynamic casting with its associated runtime overheads [26]). This is due to the fact that traditional read-only references are plain read-only—they are unaware of any encapsulation structure.

Our proposal here allows references to cross encapsulation boundaries but ensures these references cannot update the internal states directly, that is, any update still has to be initiated by the owner object. The novelty is that all references in our system are encapsulation-aware. A reference can never be used to mutate states encapsulated by the context where the reference is held, yet they can always be used to mutate states of or outside of the current context. For example, iterator objects in our program can be used by a client to mutate data elements stored by a list object, while still protecting the list’s representation from external modification. Moreover, the iterator objects may or may not be used to update the list’s internal implementation depending on the contexts where the iterators are used—the `insert` method may be called from within the list context but never from outside of the list.

## 6 Conclusion

In this paper, we have presented a general framework for tracking object invariants. The novelty of this model is a behavioural abstraction that specifies two sets, the *validity invariant* and the *validity effect*. The overlap of these two sets captures precisely those objects that need to be re-validated at the end of the behaviour. To support our general model, we have also presented an object-oriented programming language that uses ownership types to confine dependencies for object invariants, and restricts permissible updates to track where object invariants hold even in the presence of re-entrant calls.

## Acknowledgments

We thank the anonymous reviewers for their comments, one in particular being very detailed. This work has been supported by the Australian Research Council, Grant DP0665581.

## References

1. J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *European Conference on Object-Oriented Programming (ECOOP)*, July 2004.
2. P. S. Almeida. Balloon types: Controlling sharing of state in data types. *Lecture Notes in Computer Science*, 1241:32–59, 1997.
3. M. Barnett, R. DeLine, M. Fahndrich, K. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.
4. M. Barnett and D. Naumann. Friends need a bit more: Maintaining invariants over shared state. *Mathematics of Program Construction*, pages 54–84, 2004.
5. A. Birka and M. D. Ernst. A practical type system and language for reference immutability. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 35–49. ACM Press, 2004.
6. R. Bornat. Proving pointer programs in Hoare logic. *Mathematics of Program Construction*, pages 102–126, 2000.
7. R. Bornat, C. Calcagno, P. O’Hearn, and M. Parkinson. Permission accounting in separation logic. *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 259–270, 2005.
8. C. Boyapati, B. Liskov, and L. Shriram. Ownership types for object encapsulation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 213–223. ACM Press, 2003.
9. C. Boyapati, B. Liskov, and L. Shriram. Ownership types for object encapsulation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 213–223. ACM Press, 2003.
10. D. Clarke. *Object Ownership and Containment*. PhD thesis, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia, 2001.
11. D. G. Clarke and S. Drossopoulou. Ownership, encapsulation and disjointness of type and effect. In *17th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.
12. D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In *European Conference on Object-Oriented Programming (ECOOP)*, 2001.
13. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 48–64. ACM Press, 1998.
14. W. Dietl and P. Muller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 2005.

15. J. Hogg. Islands: aliasing protection in object-oriented languages. In *OOPSLA '91: Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 271–285, New York, NY, USA, 1991. ACM Press.
16. S. Ishtiaq and P. W. O'Hearn. Bi as an assertion language for mutable data structures. In *Proceedings of the 28th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*. ACM Press, 2001.
17. G. Leavens. Modular specification and verification of object-oriented programs. *Software, IEEE*, 8(4):72–80, 1991.
18. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *Proceedings of the 18th European Conference on Object-Oriented Programming*, pages 491–516. Springer-Verlag, 2004.
19. Y. Lu and J. Potter. On ownership and accessibility. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 99–123. Springer-Verlag, 2006.
20. Y. Lu and J. Potter. Protecting representation with effect encapsulation. In *Proceedings of the 33th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 2006.
21. B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1992.
22. B. Meyer. *Object-oriented software construction*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 2nd edition, 1997.
23. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. Springer, 2002.
24. P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. *Programming Languages and Fundamentals of Programming*, 1999.
25. P. Müller and A. Poetzsch-Heffter. *Universes: A Type System for Alias and Dependency Control*. Fernuniv., Fachbereich Informatik, 2001.
26. P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.
27. P. Muller, A. Poetzsch-Heffter, and G. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 2006.
28. D. Naumann and M. Barnett. Towards imperative modules: reasoning about invariants and sharing of mutable state. *Logic in Computer Science, 2004. Proceedings of the 19th Annual IEEE Symposium on*, pages 313–323, 2004.
29. J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *European Conference on Object-Oriented Programming (ECOOP)*, 1998.
30. J. Reynolds. Separation logic: a logic for shared mutable data structures. *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74, 2002.
31. W. Schulte. *Towards a Verifying Compiler: The Spec# Approach*. Microsoft Research, <http://research.microsoft.com/specsharp/>, 2006.