Specification and Verification of Object-Oriented Software

K. Rustan M. LEINO
Microsoft Research, Redmond, WA, USA
leino@microsoft.com

Abstract. The specification of object-oriented and other pointer-based programs must be able to describe the structure of the program's dynamically allocated data as well as some abstract view of what the code implements. The verification of such programs can be done by generating logical verification conditions from the program and its specifications and then analyzing the verification conditions by a mechanical theorem prover.

In these lecture notes, I present an object-based language, Dafny, whose specifications use the style of dynamic frames. I show how to write and specify programs in Dafny. Most of the material is devoted to how to build a first-order automatic program verifier for Dafny programs, generating the verification conditions as input to an automatic satisfiability-modulo-theories solver.

Keywords. Automatic program verification, program specification, information hiding and abstraction, objects and dynamic allocation, dynamic frames, Dafny

0. Introduction

A system for specifying, writing, and verifying programs has many components. One component is the programming language itself and the specification constructs it contains. In some cases, specifications are not part of the program, but are provided separately. Another component is the compiler, which generates executable code from the program. Yet another component is the program verifier, which itself can include a subcomponent that formalizes the semantics and proof obligations of the program, a subcomponent that generates logical formulas, called *verification conditions*, from the formalization, and a subcomponent, called the *theorem prover*, that analyzes the verification conditions and searches for proofs or counterexamples. In these lecture notes, I describe a system with the components mentioned here, but I omit a description of the compiler and theorem prover. After following these lecture notes, you should be able to build your own basic program verifier for your own language.

In each component and subcomponent, the system designer makes many choices. In order to cover a whole system, I will make a particular set of design choices in these lecture notes. There are many other choices already in existence, and future research is likely to invent and explore new choices.

The choice I make for the programming language is to include some basic features found in imperative languages that allow references (pointers) to dynamically allocated data. The language, which I call *Dafny*, is type safe and centers around simple

classes. I omit higher-order features like lambda terms (closures) and map-reduce operations. Other facilities, like additional statements, exceptional control flow, immutable data structures, generic types, subclassing, unsafe memory operations, and various concurrency features can be designed as variations of what I present. Figure 0 shows a Dafny program that implements a linked list.

The choice I make for specifications is based on *dynamic frames* [28]. Dynamic frames provide a flexible style of describing classes built in layers, where one object is implemented in terms of other, more primitive objects. The effect of using dynamic frames on the specification features required in the language is modest. Although they can be a little verbose, specifications tend to be highly stylized. Experience with the dynamic-frames style will reveal common specification idioms for which one can invent terser syntax (*e.g.*, [57]), possibly at the loss of flexibility.

The choice I make for formalizing the semantics and proof obligations of the language is to translate the language into an intermediate verification language. The intermediate language is called *Boogie* [36,4], of which I will describe the features we need. Just like compiler writers have found it desirable to separate concerns in the compilation process by using an intermediate representation (sometimes more than one) (*cf.* [1]), builders of automatic program verifiers have also found that an intermediate language separates concerns in the process and lends itself to a componentized design of the program verifier [42,4,21].

The choice I make for generating verification conditions from the intermediate program is *weakest preconditions* [17,48]. In a practical system, there are several important optimizations to make in this subcomponent of the system [23,35,6,37], but the unoptimized version I will cover is still adequate for verifying many programs. I use weakest preconditions that let us prove that programs avoid irrecoverable errors, but I ignore issues of program termination.

My presentation stops at the level of verification conditions, and I will not describe the theorem provers that analyze these formulas. There are several good satisfiability-modulo-theories (SMT) solvers that can be used by automatic program verifiers ((*e.g.*, [15,9,14]), and such provers can be used as off-the-shelf components. In fact, the step of generating verification conditions from the intermediate language can also be done by existing tools (like the Boogie tool [4] and the Why tool [19]), which further justifies spending fewer words on that part in these lecture notes.

In these lecture notes, I start in Fig. 1 by presenting some example programs written in Dafny, which gives a flavor of the kinds of programs we aim to write a verifier for. The example programs show the dynamic-frames style of specifications in Dafny. Starting in Section 2, I turn to the definition of the verifier itself, starting bottom up: Sections 2 and 3 introduce the intermediate language Boogie and its weakest preconditions, and Section 4 presents the source language Dafny in more detail and defines its semantics as a translation of its classes, fields, methods, and functions into Boogie.

The general ideas and many details of the translation apply to a variety of object-based languages. The streamlined support of dynamic-frame specifications in Dafny comes from its **modifies** and **reads** clauses, which frame the modifications of methods and dependencies of functions; from the well-definedness checks imposed on functions, which build on the frames; and from the stylized use of set-valued (ghost) fields in specifications.

```
class Data { }

class Node {
  var list: seq<Data>;
  var footprint: set<Node>;

  var data: Data;
  var next: Node;

  function Valid(): bool
    reads this, footprint;
  {
    this ∈ footprint ∧ ¬(null ∈ footprint) ∧
    (next = null ⟹ list = [data]) ∧
    (next ≠ null ⟹
        next ∈ footprint ∧ next.footprint ⊆ footprint ∧
        ¬(this ∈ next.footprint) ∧
        list = [data] + next.list ∧
        next.Valid())
  }

  method Init(d: Data)
    modifies this;
    ensures Valid() ∧ fresh(footprint − {this});
    ensures list = [d];
  {
    data := d;  next := null;
    list := [d];  footprint := {this};
  }

  method SkipHead() returns (r: Node)
    requires Valid();
    ensures r = null ⟹ |list| = 1;
    ensures r ≠ null ⟹ r.Valid() ∧ r.footprint ⊆ footprint;
    ensures r ≠ null ⟹ r.list = list[1..];
  {
    r := next;
  }

  method Prepend(d: Data) returns (r: Node)
    requires Valid();
    ensures r.Valid() ∧ fresh(r.footprint − old(footprint));
    ensures r.list = [d] + list;
  {
    r := new Node;
    r.data := d;  r.next := this;
    r.footprint := {r} ∪ footprint;
    r.list := [r.data] + list;
  }

  // ReverseInPlace shown in next figure
}
```

Figure 0. An example linked-list program written in Dafny.

```
method ReverseInPlace() returns (reverse: Node)
  requires Valid();
  modifies footprint;
  ensures reverse ≠ null ∧ reverse.Valid();
  ensures fresh(reverse.footprint − old(footprint));
  ensures |reverse.list| = |old(list)|;
  ensures (∀ i: int • 0 ≤ i ∧ i < |old(list)| ⟹
              old(list)[i] = reverse.list[|old(list)|−1−i]);
{
  var current: Node;
  current := next;
  reverse := this;
  reverse.next := null;
  reverse.footprint := {reverse};
  reverse.list := [data];

  while (current ≠ null)
    invariant reverse ≠ null ∧ reverse.Valid();
    invariant reverse.footprint ⊆ old(footprint);
    invariant current = null ⟹ |old(list)| = |reverse.list|;
    invariant current ≠ null ⟹
      current.Valid() ∧
      current ∈ old(footprint) ∧ current.footprint ⊆ old(footprint) ∧
      current.footprint ⋕ reverse.footprint ∧
      |old(list)| = |reverse.list| + |current.list| ∧
      (∀ i: int • 0 ≤ i ∧ i < |current.list| ⟹
          current.list[i] = old(list)[|reverse.list|+i]);
    invariant (∀ i: int • 0 ≤ i ∧ i < |reverse.list| ⟹
                  old(list)[i] = reverse.list[|reverse.list|−1−i]);
  {
    var nx: Node;
    nx := current.next;
    assert nx ≠ null ⟹
        (∀ i: int • 0 ≤ i ∧ i < |nx.list| ⟹
            current.list[1+i] = nx.list[i]);

    // The state looks like: ..., reverse, current, nx, ...
    assert current.data = current.list[0];
    current.next := reverse;
    current.footprint := {current} ∪ reverse.footprint;
    current.list := [current.data] + reverse.list;

    reverse := current;
    current := nx;
  }
}
```

Figure 1. The Node.ReverseInPlace method, which performs an *in situ* list reversal.

Likely, the best way to master the material in these lecture notes is to implement your own verifier for a language like Dafny. I recommend it. But you are also welcome to try out my Dafny verifier, which is available as part of the Spec# download [60]. In my implementation, I also support **break** and **return** statements, generic types, and type inference.

1. Dynamic Frames in Dafny

In this section, I describe three example Dafny programs.

1.0. Linked List with Reverse Method

The Dafny linked-list implementation in Figs. 0 and 1 exhibits idiomatic specifications in the style of dynamic frames [28]. The term *frame* refers to a set of memory locations, and an expression denoting a frame is *dynamic* in the sense that as the program executes, the set of locations denoted by the frame can change. For simplicity, frames in Dafny are at the granularity of (all fields of) objects, not individual object fields. A dynamic frame is thus denoted by a set-valued expression (in particular, a set of object references), and this set is idiomatically stored in a field. Figure 0 shows that field as footprint. I shall call footprint a *ghost field*, because it is introduced for the purpose of writing specifications and it need not be included in a compiled program; however, Dafny has no special declaration for ghost fields, so the distinction between regular fields and ghost fields is just part of our mental model.

Class Node declares four fields. The fields data and next are the usual fields in a linked-list data structure. The other two are ghost fields: list is the sequence of data values stored in a node and its successors, and footprint is a set consisting of the node and its successors. Part of Dafny's simplicity—as well as the fact that Dafny's verification conditions seem to be handled swiftly by a modern SMT solver, even for programs with recursive data structures and recursive predicates—comes from the explicit representation of abstract entities like list and footprint as ghost fields (*cf.* [39]). The price to pay for this simplicity of the language is that the fields have to be explicitly updated in the program, as we shall soon see.

Idiomatically, Node also declares a function Valid, which has the return value true if the node and, recursively, its successors satisfy the intended steady-state invariant for these objects. Functions in Dafny have to declare the frames they read; Valid says it may read the state of any object in the set

```
{this} ∪ this.footprint
```

where **this** denotes the receiver object of the method. (Here and elsewhere, the receiver object can be implicit, as is customary in object-oriented languages. So, **this**.footprint can be written simply as footprint.) This **reads** set is used in two ways (see Section 4.5): first, it induces a well-founded order among function applications, which is used in checking that the definitions of given functions are logically consistent; second, it gives rise to an axiom that is needed in this and other programs to show that certain state changes have no effect on the value returned by a function application.

The definition of Valid spells out the expected invariants about the node's abstract value, like the subexpression

```
n.list = [n.data] + n.next.list
```

It also spells out properties of footprint. The interplay between Valid (which defines the intended contents of footprint) and footprint (which defines what Valid is allowed to read) is handled by carefully revealing that an object is a member of footprint before it is derefer-

enced. For example, the reading of next.footprint is preceded by next $\in$ footprint. We shall see this left-to-right ordering reflected in the definedness checks on functions (funcdf) in Section 4.5.

To keep the language simple, Dafny separates allocation, which is done by a built-in statement, from initialization, which is defined by an ordinary method. That is, there are no constructors like those built into Java-like languages. To construct a singleton Node object, a client writes:

```
s := new Node;  call s.Init(d);
```

(which is similar to how it is done in Modula-3 [49]).

Methods in Dafny have to declare the frames they write. In the case of method Init, the frame is the state of the objects in the (singleton) set {**this**}. The postcondition of Init says about the receiver object that Valid holds and that list is the singleton sequence [d].

It is also important that the postcondition say something about the new value of **this**.footprint, since that is a field that the method declares that it may modify. The body of Init sets footprint to the singleton set {**this**}, but the general case is that a method may grow the footprint of objects. If a footprint grows as the result of a method call, then a caller needs to know how the new footprint may overlap with other footprints in the caller's purview. The postcondition

```
fresh(footprint - {this})
```

says that the objects in footprint, with the exception of **this**, are freshly allocated during the execution of the method. This means that the overlap of the final value of footprint with footprints previously known to the caller is at most {**this**}. For non-Init methods that mutate an object's state, the specification idiom is

```
modifies footprint;
ensures fresh(footprint - old(footprint));
```

which says that the possible growth of footprint does not introduce any overlap with footprints previously known to the caller. This postcondition expresses what is known as the *swinging pivots restriction* [40,28,57].

As a final remark about Init, note that the ghost fields list and footprint are updated explicitly in the method body.

Methods SkipHead (the familiar cdr) and Prepend (the familiar $cons$) operate on an initialized object, as reflected by the precondition Valid. The methods do not modify any previously allocated state, so their **modifies** clauses are empty (and thus omitted). Note in both cases how the footprint of the result is described in the postcondition. This lets the caller know that there may be overlap between **this**.footprint and r.footprint after a call, which means that modifying something in the footprint of either of these objects may affect the validity of the other. A different design, leading to different implementations of these methods, would be to instead us the postcondition

```
ensures footprint ⫫ r.footprint;
```

which says that the footprints of **this** and r are disjoint.

Method ReverseInPlace, shown in Fig. 1, performs an *in situ* reversal of the list, which is to say that it reuses the storage cells used by the original list. Its specification says that **this**.footprint, on which **this**.Valid depends, may change, but says nothing about the value of **this**.Valid after the call, which in effect prevents the caller from continuing to use **this** as a valid list. The last two postconditions say that the returned list really does represent the reverse of the initial list.

In the body of method ReverseInPlace, the loop not only updates the link next between nodes, but also adjusts the ghost fields accordingly. The loop invariant is big, describing the intermediate states of the reversal process. The condition

```
current.footprint ⫫ reverse.footprint
```

says that the sets current.footprint and reverse.footprint do not overlap, a condition that separation logic specializes in expressing (with the separating-conjunction operator $*$) [55].

Finally, it is interesting to note that the specifications in this example program do not make use of any reachability predicate [47]. This is good news for automatic checking, because trying to find a practical encoding of the reachability predicate in trigger-based SMT solvers has proved to be difficult [27,30].

In my implementation of Dafny, which is built on Boogie [4], this program yields the following output when using Z3 [14] as the underlying SMT solver:

```
Dafny program verifier version 0.90, Copyright (c) 2003-2008, Microsoft.
Parsing ListContents.dfy

Running abstract interpretation...
  [0.063 s]

Verifying CheckWellformed$$Node.Valid ...
  [0.199 s]  verified

Verifying Node.Init ...
  [0.013 s]  verified

Verifying Node.SkipHead ...
  [0.01 s]  verified

Verifying Node.Prepend ...
  [0.014 s]  verified

Verifying Node.ReverseInPlace ...
  [1.07 s]  verified

Dafny program verifier finished with 5 verified, 0 errors
```

The output shows the verification times in seconds on my 2.4 GHz desktop machine.

1.1. A Common Specification Idiom

Dynamic-frame specifications are often idiomatic. Figure 2 shows a code skeleton that uses a recurring pattern of specification in Dafny.

The class Coo declares a ghost field footprint for storing the set of objects used to represent a Coo object, and the function Valid returns true when the object satisfies its invariant. The **reads** clause of function Valid says that the fields of all objects in footprint are allowed to be read by Valid. As defined by its body, function Valid returns true only if **this** $\in$ footprint, but in order to bootstrap the reading process, the **reads** clause of Valid also mentions {**this**}—this gives the body permission to mention **this**.footprint in the first place.

```
class Coo {
  var footprint: set<object>;
  // other fields go here...
  function Valid(): bool
    reads this, footprint;
  {
    this ∈ footprint // something more substantial goes here...
  }
  method Init()
    modifies this;
    ensures Valid() ∧ fresh(footprint − {this});
  {
    footprint := {this};
    // more initialization goes here...
  }
  method Mutate()
    requires Valid();
    modifies footprint;
    ensures Valid() ∧ fresh(footprint − old(footprint));
  {
    // mutations of the object go here...
  }
}
```

Figure 2. A code skeleton that illustrates a common specification idiom in Dafny.

An initialization method, like Init in class Coo, modifies the fields of the receiver object, and it establishes validity of the object. The effect on the footprint is as in the Node class in Section 1.0.

The mutating method in Fig. 2 both requires and ensures validity of the receiver. It declares that it may modify any field of any object in footprint, and it promises to extend footprint only by newly allocated objects. When verifying that the body of the method satisfies its **modifies** clause, it is necessary to have information about which objects are in footprint, and that information comes from the definition of function Valid and the fact that the method requires Valid as a precondition.

Let me extend the example by showing the skeleton of a class that is implemented on top of Coo, see Fig. 3. Class Bill includes an integer field and a reference to a Coo object. By the definition of Bill.Valid, the validity of a Bill object b implies the validity of its underlying Coo object b.c and the inclusion of the footprint of b.c in the footprint of b. In fact, b.c.footprint will be a proper subset of b.footprint, because b is excluded from b.c.footprint. Methods Init and Mutate have the same specifications as in class Coo.

It is instructive to consider the proof of method Bill.Mutate. In the initial state of the method, Valid() holds, which implies ¬(**this** ∈c.footprint). This tells us two important things. First, it tells us that changes to fields of **this** do not affect the state of the objects in c.footprint, and so by the **reads** clause of c.Valid, changes to x and footprint have no effect on the value of c.Valid(). Second, it tells us that the call to c.Mutate, whose **modifies** clause is c.footprint, has no effect on the fields of **this**. Now, then: The update of **this**.x is allowed because the precondition Valid() implies **this** ∈footprint. The precondition of the call is met, because Valid() implies c.Valid(). The mutations performed by the call are allowed, because c.footprint is a subset of footprint. The last assignment adds to footprint any objects added to c.footprint by the call. Finally, Valid() holds in the final state, because

```
class Bill {
  var x: int;
  var c: Coo;

  var footprint: set<object>;

  function Valid(): bool
    reads this, footprint;
  {
    this ∈ footprint ∧
    c ≠ null ∧ c ∈ footprint ∧
    ¬(this ∈ c.footprint) ∧ c.footprint ⊆ footprint ∧
    0 ≤ x ∧ c.Valid()
  }

  method Init()
    modifies this;
    ensures Valid() ∧ fresh(footprint − {this});
  {
    var coo := new Coo;
    call coo.Init();
    x := 0;
    c := coo;
    footprint := {this} ∪ c.footprint;
  }

  method Mutate()
    requires Valid();
    modifies footprint;
    ensures Valid() ∧ fresh(footprint − old(footprint));
  {
    x := x + 1;
    call c.Mutate();
    footprint := footprint ∪ c.footprint;
  }
}
```

Figure 3. A code skeleton that illustrates how one class (Bill) is built on top of another (Coo in Fig. 2).

x was only incremented and the call maintained validity of c and added only newly allocated objects to c.footprint (which also implies that **this**, which was allocated before the call, was not added to c.footprint).

The previous paragraph includes many subtle points. Luckily, checking them does not require hard mathematics, but only an obsessive attention to detail. This makes the task well suited for an automatic theorem prover.

1.2. Queue

As a final example before we go into the details of a verifier for Dafny, let us consider the implementation of a queue. The queue is implemented as a linked list of nodes, with a pointer to both the first and last nodes of the list, see Fig. 4. To simplify the code, the list is always nonempty, keeping a sentinel object at the head of the list. Enqueue operations are done at the tail, dequeue operations at the head.

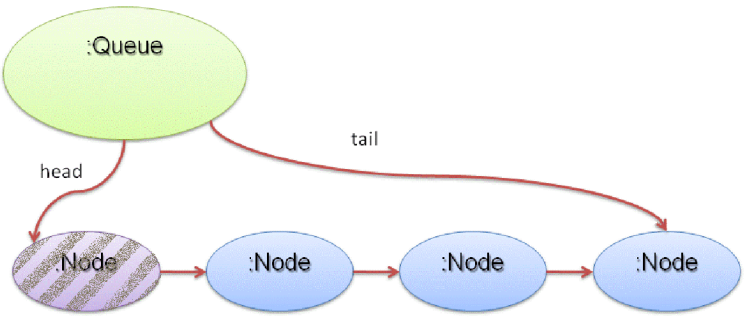

Figure 4. A pictorial view of the Queue implementation. The head field of a queue object points to the head of a linked list and the tail field points to the tail of that list. The first node in the list is a sentinel node.

The start of the class declaration for Queue is given in Fig. 5. In addition to the head and tail fields, the class declares three ghost fields: contents represents the sequence of data elements stored by the queue, footprint is the usual dynamic-frame field, and spine is the subset of footprint that contains just the Node objects along the linked list.

The invariant of a Queue object, as reflected in the definition of Valid, is more complicated than in the previous examples. In particular, it quantifies over all nodes in the spine, saying that each such node is valid with a footprint that is contained in the queue's footprint. The quantification also says that only one node in the spine has a next field of **null**, and that is the tail node. The second quantification says that the spine is closed under next.

Note that the validity condition does not say that all nodes along the spine are reachable from head. In fact, this under-specification lets spine contain additional nodes whose next fields eventually lead to one of the linked-list nodes of interest; it would even permit cycles of nodes. Still, the under-specification is good enough to verify the methods of the class, and this style of specification avoids talking about reachability, which can be difficult to reason about with SMT solvers.

Class Node is declared in Fig. 7. Its validity condition says that the footprint contains the footprint of the successor node. It also declares the intention to maintain tailContents as the data fields of all successor nodes. In this implementation of Queue, which puts the sentinel object at the head, it is convenient not to include the value of a node's data field in the node's own tailContents field.

The initialization methods of Queue (Fig. 5) and Node (Fig. 7) offer no surprises. They mainly use the idiomatic specification for Init methods as shown in the Bill and Coo examples, with the addition of postconditions that are specific to Queue and Node.

Similarly, the implementations of Front and Dequeue (Fig. 6) offer no surprises. Other than the idiomatic specification for mutating methods shown in the Bill and Coo examples, the specification of Dequeue is just:

```
requires 0 < |contents|;
ensures contents = old(contents)[1..];
```

Note that Front has an empty (that is, omitted) **modifies** clause, since it does not change the state, and note that Dequeue explicitly updates the ghost field contents in accordance

```
class Data { }

class Queue {
  var head: Node;
  var tail: Node;

  var contents: seq<Data>;
  var footprint: set<object>;
  var spine: set<Node>;

  function Valid(): bool
    reads this, footprint;
  {
    this ∈ footprint ∧ spine ⊆ footprint ∧
    head ≠ null ∧ head ∈ spine ∧
    tail ≠ null ∧ tail ∈ spine ∧
    tail.next = null ∧
    (∀ n • n ∈ spine ⟹
        n ≠ null ∧ n.Valid() ∧
        n.footprint ⊆ footprint ∧
        (n.next = null ⟹ n = tail)) ∧
    (∀ n • n ∈ spine ⟹
        n.next ≠ null ⟹ n.next ∈ spine) ∧
    contents = head.tailContents
  }

  method Init()
    modifies this;
    ensures Valid() ∧ fresh(footprint − {this});
    ensures |contents| = 0;
  {
    var n := new Node;
    call n.Init();
    head := n;  tail := n;
    contents := n.tailContents;
    footprint := {this} ∪ n.footprint;
    spine := {n};
  }

  // other methods shown in next figure
}
```

Figure 5. The declaration of the fields of the Queue, their validity condition, and initializer.

with the validity condition. Because of the under-specification in Valid, no change is needed for footprint and spine.

The implementation of Enqueue is more interesting. As expected, it allocates a node Node, which it initializes and adds to the tail of the linked list. But it also needs to update the ghost fields. Since the footprint of each node contains the the footprint of the successor node, and the tailContents of a node is defined in terms of the tailContents of the successor node, it is necessary to update the footprint and tailContents fields of all the nodes. This is conveniently performed by the **foreach** statement, which simultaneously sets a field of many objects. Note that the append and union operations on these two fields maintain the invariants (stated in Queue.Valid and Node.Valid) about all Node objects,

```
method Front() returns (d: Data)
  requires Valid();
  requires 0 < |contents|;
  ensures d = contents[0];
{
  d := head.next.data;
}

method Dequeue()
  requires Valid();
  requires 0 < |contents|;
  modifies footprint;
  ensures Valid() ∧ fresh(footprint − old(footprint));
  ensures contents = old(contents)[1..];
{
  var n := head.next;
  head := n;
  contents := n.tailContents;
}

method Enqueue(d: Data)
  requires Valid();
  modifies footprint;
  ensures Valid() ∧ fresh(footprint − old(footprint));
  ensures contents = old(contents) + [d];
{
  var n := new Node;
  call n.Init();  n.data := d;
  tail.next := n;
  tail := n;

  foreach (m ∈ spine) {
    m.tailContents := m.tailContents + [d];
  }
  contents := head.tailContents;

  foreach (m ∈ spine) {
    m.footprint := m.footprint ∪ n.footprint;
  }
  footprint := footprint ∪ n.footprint;

  spine := spine ∪ {n};
}
```

Figure 6. The familiar methods of a queue abstraction. Methods Front and Dequeue operate on the head of the linked-list representation, whereas Enqueue, which adds a node at the tail, operates on the ghost fields of all linked-list nodes.

even in light of the under-specification that allows spine to contain nodes outside the linked-list nodes of interest.

Although one can imagine compiling and executing such a **foreach** statement, the Queue class uses the statement only to update ghost fields, so a compiler could easily ignore the statement when generating executable code.

Finally, let us look at an example use of the Queue class, shown in Fig. 8. Method Main allocates and initializes two queues, q0 and q1. It enqueues x and y into q0 and z

```
class Node {
  var data: Data;
  var next: Node;

  var tailContents: seq<Data>;
  var footprint: set<object>;

  function Valid(): bool
    reads this, footprint;
  {
    this ∈ footprint ∧
    (next ≠ null ⟹ next ∈ footprint ∧ next.footprint ⊆ footprint) ∧
    (next = null ⟹ tailContents = []) ∧
    (next ≠ null ⟹ tailContents = [next.data] + next.tailContents)
  }

  method Init()
    modifies this;
    ensures Valid() ∧ fresh(footprint − {this});
    ensures next = null;
  {
    next := null;
    tailContents := [];
    footprint := {this};
  }
}
```

Figure 7. The Node class that is part of the Queue implementation.

into q1, after which it asserts the length of q0 to be 2. It then inspects the elements at the front of q0, with an intervening dequeue operation, after which it checks that both queues have length 1.

The correctness of the Main method relies on q0 and q1 to operate independently of each other. This is achieved by the idiomatic specifications. The allocation of q1 yields a previously unallocated object, which implies that q1 is not a member of q0.footprint, all of whose objects are allocated (as guaranteed by the Dafny language and the encoding we shall see in later sections). The call to q1.Init may change fields of q1, which includes q1.footprint, but it promises to enlarge the footprint only by newly allocated objects. Thus, after the two queues have been initialized, their footprints can be proved to be disjoint. The remaining operations also enlarge footprints only by newly allocated objects, so the disjointness is maintained. Since q1.Valid reads only the fields of the objects in q1.footprint, methods that operate on the fields of the objects in q0.footprint have no effect on q1.Valid.

2. Intermediate Language

In this section, I give a first introduction to the Boogie language, as I use it in subsequent sections to encode the semantics and proof obligations of Dafny. I will present some other parts of Boogie when needed. A full definition of the language is found in the Boogie 2 language reference manual [36].

```
method Main(x: Data, y: Data, z: Data)
{
  var q0 := new Queue;
  call q0.Init();
  var q1 := new Queue;
  call q1.Init();

  call q0.Enqueue(x);
  call q0.Enqueue(y);

  call q1.Enqueue(z);

  assert |q0.contents| = 2;

  var w;
  call w := q0.Front();
  assert w = x;
  call q0.Dequeue();

  call w := q0.Front();
  assert w = y;

  assert |q0.contents| = 1;
  assert |q1.contents| = 1;
}
```

Figure 8. An example method that uses the Queue class. The Dafny verifier verifies the correctness of this method automatically, including the **assert** statements.

Boogie consists of a mathematical part and an imperative part. The mathematical part has declarations of (uninterpreted) types, constants, and first-order functions, as well as axioms, which are used to state properties about these types, constants, and functions. The imperative part has declarations of variables and procedures.

Mathematical declarations Here is a small example that shows Boogie's mathematical declarations:

$$\begin{array}{l}
\textbf{type}\ Keyboard;\\
\textbf{const}\ Yamaha_DX7\colon Keyboard;\\
\textbf{function}\ keys(Keyboard)\ \textbf{returns}\ (\textbf{int});\\
\textbf{axiom}\ (\forall\, k\colon Keyboard \bullet\ 0 \leqslant keys(k)\ );\\
\textbf{axiom}\ keys(\ Yamaha_DX7) = 61;
\end{array}$$

These declarations introduce a type to represent musical keyboards, a constant to represent the classic DX7 synthesizer, and a function that returns the number of keys a given instrument has. The first axiom postulates that a keyboard cannot have a negative number of keys, and the second axiom says that the DX7 has 61 keys.

Expressions The expression language of Boogie includes usual boolean and integer operators. All expressions in Boogie are total; for example, even division by zero results in some value that is a (fixed but unknown) function of its arguments. The language includes map types and two operations on maps: the expression $m[j]$ accesses the value of map m at domain element j (in other words, it applies map m to argument j and

returns the result), and the expression $m[j := x]$ returns a map that is like m except that it maps j to x. Maps can also take multiple arguments, in which case j is a list of domain elements. Finally, the expression language includes universal and existential quantifications.

Procedures A procedure is declared as follows:

$$\textbf{procedure}\ P(ins)\ \textbf{returns}\ (outs);\ Spec$$

where P is the name of the procedure, ins is the list of formal in-parameters, $outs$ is the list of formal out-parameters, and $Spec$ is the procedure's specification. If $outs$ is empty, then the entire **returns** clause can be dropped. The procedure specification $Spec$ consists of clauses of the forms

$$\textbf{requires}\ Expr;$$

which declares a *precondition* (a condition that is to hold on every invocation of the procedure), and

$$\textbf{modifies}\ xs;$$

which declares a *modifies clause* (which says which global variables the procedure is allowed to change), and

$$\textbf{ensures}\ Expr;$$

which declares a *postcondition* (a condition that is to hold on exit from the procedure implementation). Multiple **requires** clauses are equivalent to one **requires** clause that conjoins the conditions, and similar for multiple **ensures** clauses. Multiple **modifies** clauses are equivalent to one **modifies** clause that gives all the variables in one list. A postcondition, and also any expression inside an implementation body, can use the expression $\textbf{old}(E)$, which stands for the value of expression E on entry to the procedure.

Statements Procedures can have any number of in- and out-parameters. A procedure implementation consists of the declaration of a number of local variables followed by a list of statements that follow this grammar:

$$\begin{array}{rcl}
Stmt & ::= & xs := Exprs; \\
& | & x[Exprs] := Expr; \\
& | & \textbf{havoc}\ xs; \\
& | & \textbf{if}\ (Expr)\ \{\ Stmts\ \}\ \textbf{else}\ \{\ Stmts\ \} \\
& | & \textbf{while}\ (Expr)\ Invs\ \{\ Stmts\ \} \\
& | & \textbf{assert}\ Expr; \\
& | & \textbf{assume}\ Expr; \\
& | & \textbf{call}\ xs := P(Exprs);
\end{array}$$

where x is an identifier, xs is a list of distinct identifiers, P is the name of a declared procedure, $Expr$ is an expression, $Exprs$ is a list of expressions, $Stmts$ is a list of statements, and $Invs$ is a list of loop-invariant declarations each of which has the form:

$$\textbf{invariant}\ Expr;$$

Multiple **invariant** declarations are equivalent to one **invariant** declaration that conjoins the conditions (and no **invariant** declaration is equivalent to **invariant true**). Some common (and "obvious") restrictions apply; for example, the number of left-hand variables in the assignment statement must equal the number of right-hand expressions, and the types of the guard expressions in the **if** and **while** statements must be boolean.

For any list of variables xs, $xs := EE;$ is the usual parallel assignment statement, which first evaluates all the right-hand expressions and then assigns these values to the respective left-hand variables. The map-update statement $m[jj] := E;$ changes map variable m so that it maps jj to E. Stated differently, it assigns to m the value $m[jj := E]$. The **havoc** statement takes a list of variables and sets each one of them to an arbitrary (blindly chosen, demonically chosen) value. The variables assigned in these statements, and also the variables used as actual out-parameters of call statements, must be global variables, out-parameters, or local variables; in-parameters are immutable.

The **if** statement is the usual conditional statement. The **while** statement is the usual loop statement that iteratively executes $Stmts$ while the guard $Expr$ is true. At the top of each loop iteration (in other words, immediately before each evaluation of the guard expression; or, stated differently, immediately before the loop and also immediately after each execution of $Stmts$), each of the declared invariants must hold, or else the loop execution results in an irrecoverable error.

The **assert** statement records a proof obligation. If the given condition holds, the statement acts like a no-op. If the condition does not hold, the execution (stops and) results in an irrecoverable error. So, the assert statement can be used to express a condition that, in order for the source program to be correct, must hold at the given program point. A statement is said to *go wrong* if it results in an irrecoverable error.

The **assume** statement restricts the set of feasible executions defined by the Boogie program. If the given condition holds, the statement acts like a no-op. An execution is considered infeasible if it would have led to an assume statement where the expression would evaluate to false. So, the assume statement can be used to express that only certain (namely, the feasible) executions are to be considered by the program verifier. For example, an instance of a common idiomatic use of **havoc** and **assume** is:

$$\textbf{havoc}\ x;\ \ \textbf{assume}\ 0 \leqslant x;$$

which ("first") picks any value for x but ("then") considers the execution feasible only if x is non-negative. In other words, this use of the important havoc-assume idiom expresses "set x to an arbitrary value that satisfies $0 \leqslant x$".

Finally, the call statement $\textbf{call}\ xs := P(EE);$ invokes procedure P, where EE is the list of actual in-parameters and xs is the list of actual out-parameters.

3. Verification Conditions

A Boogie program is correct if all procedure implementations satisfy their specifications. To check that a procedure implementation satisfies its specification, Boogie performs a syntactic check (for modifies clauses) and generates a verification condition to be discharged by a theorem prover. The proof obligations encoded by the verification condition arise from the postcondition of the procedure being verified, the preconditions of called procedures, and the conditions in assert statements.

The meaning of simple statements The semantics of statements is given by the weakest-precondition transformer wp [17,48]. I use the following definition: For any statement S and condition Q on the post-state of S, the *weakest precondition* of S with respect to Q, denoted $\mathsf{wp}[\![S, Q]\!]$, is the condition that characterizes those pre-states from which every feasible execution of S does not go wrong and, if it terminates, terminates in a state satisfying Q. In short, the formula $\mathsf{wp}[\![S, Q]\!]$ tells us what must hold in the pre-state of S in order for S to correctly establish Q. Weakest preconditions are defined inductively on the structure of basic statements:

$$
\begin{array}{lcl}
\mathsf{wp}[\![xs := EE;,\ Q]\!] & = & Q[\![EE \mathbin{/\!/} xs]\!] \\
\mathsf{wp}[\![\mathbf{havoc}\ xs;,\ Q]\!] & = & (\forall\, xs \bullet\ Q\ ) \\
\mathsf{wp}[\![\mathbf{assert}\ E;,\ Q]\!] & = & E \ \wedge\ Q \\
\mathsf{wp}[\![\mathbf{assume}\ E;,\ Q]\!] & = & E \ \Rightarrow\ Q \\
\mathsf{wp}[\![S\ T,\ Q]\!] & = & \mathsf{wp}[\![S,\ \mathsf{wp}[\![T, Q]\!]]\!] \\
\mathsf{wp}[\![\mathbf{if}\ (E)\ \{\ S\ \}\ \mathbf{else}\ \{\ T\ \},\ Q]\!] & = & \\
\quad (E \ \Rightarrow\ \mathsf{wp}[\![S, Q]\!]) \ \wedge\ (\neg E \ \Rightarrow\ \mathsf{wp}[\![T, Q]\!]) & &
\end{array}
$$

where $Q[\![EE \mathbin{/\!/} xs]\!]$ denotes the simultaneous capture-avoiding substitution of the expressions EE for the variables xs in Q. As a notational convention, I use double brackets, as in $\mathsf{wp}[\![\cdot, \cdot]\!]$ and $\cdot[\![\cdot \mathbin{/\!/} \cdot]\!]$, for transformations that produce formulas or other program snippets.

To better understand the more complicated statements below, it can be helpful to read out the semantics defined by these simpler equations. The first three say the following. Statement $xs := EE;$ correctly establishes Q if what Q says about xs holds for EE in the pre-state. Statement $\mathbf{havoc}\ xs;$ correctly establishes Q if Q holds in the pre-state for all values of xs—since the $\mathbf{havoc}$ statement may set xs to any value, correctness is guaranteed by making sure Q holds for all xs. Statement $\mathbf{assert}\ E;$ correctly establishes Q if E holds in the pre-state—otherwise, the statement goes wrong—and Q already holds—since the statement does not change the program state.

The meaning of map updates The map-update statement $m[jj] := E;$ is simply a shorthand for the assignment statement $m := m[jj := E]$. Thus, its weakest precondition is:

$$\mathsf{wp}[\![m[jj] := E;,\ Q]\!] \quad = \quad Q[\![m[jj := E] \mathbin{/\!/} m]\!]$$

The meaning of loops Operationally, a loop keeps iterating while its guard evaluates to true. To generate a verification condition that closely mimics this operational behavior involves computing a fix-point. Fortunately, we can abstract from the iterations of the loop and instead define the semantics of the loop in terms of a given loop invariant. In essence, this replaces the problem of finding a fix-point with the easier problem of verifying a supposed fix-point. Such a loop invariant typically comes from a combination of (i0) properties that always hold in the source language, (i1) rules enforced by an accompanying programming discipline, (i2) invariants inferred by some inference engine, and (i3) loop invariants declared in the source language by the programmer. In Section 4.4, I will use (i0), (i1), and (i3) in the translation from Dafny into Boogie. For the important study of (i2), see for example the rich literature on abstract interpretation [12].

Let me first define the *syntactic assignment targets* of a statement. A statement S *assigns to a variable* x if x occurs in the left-hand side of an assignment statement in

S, x is the map in a map-update statement in S, x occurs in a **havoc** statement in S, x is an actual out-parameter of a call statement in S, or x is mentioned in the modifies clause of a procedure called by S. The syntactic assignment targets of a statement S is the set of variables to which S assigns.

Consider a loop

$$\textbf{while}\ (E)\ \textbf{invariant}\ J;\ \{\ S\ \}$$

and let xs denote the syntactic assignment targets of S. I encode the meaning of this **while** statement as:

$$\begin{array}{l}\textbf{assert}\ J;\\ \textbf{havoc}\ xs;\ \ \textbf{assume}\ J;\\ \textbf{if}\ (E)\ \{\ S\ \ \textbf{assert}\ J;\ \ \textbf{assume false};\ \}\ \textbf{else}\ \{\ \}\end{array}$$

Intuitively, the first assert statement checks that the declared loop invariant holds on entry to the loop. The effect of the havoc-assume idiom is to "fast forward" the execution of the loop to the top of an arbitrary loop iteration—at the top of an arbitrary loop iteration, the syntactic loop targets can have any values that satisfy the declared loop invariant. If the loop guard does not hold in that state, the loop terminates and execution continues after the loop. If the loop guard holds, the encoding checks that the loop body maintains the declared loop invariant, that is, that the loop invariant holds after an arbitrary iteration of the loop body.

The role of the final "**assume false**;" in the encoding of the loop strikes many as mysterious, but understanding and making use of such encodings is key to defining semantics effectively in the present style. The idea is to include all executions that start in an arbitrary loop state and go wrong, and to ignore executions through the loop body that do not go wrong. The final "**assume false**;" treats any execution that correctly gets past the "S **assert** J;" as infeasible, thus having the effect of being ignored.

Another way to understand the loop semantics is to calculate the weakest precondition of the encoding and try to understand the resulting formula. For any predicate Q,

$$\begin{array}{ll}
 & \mathsf{wp}[\![\ \textbf{while}\ (E)\ \textbf{invariant}\ J;\ \{\ S\ \},\ Q\ ]\!]\\
= & \quad \langle\ \text{semantic encoding of } \textbf{while}\ \rangle\\
 & \mathsf{wp}[\![\ \textbf{assert}\ J;\ \textbf{havoc}\ xs;\ \textbf{assume}\ J;\\
 & \quad \textbf{if}\ (E)\ \{\ S\ \ \textbf{assert}\ J;\ \textbf{assume false};\ \}\ \textbf{else}\ \{\ \},\ Q\ ]\!]\\
= & \quad \langle\ \mathsf{wp}\ \text{of}\ \textbf{if}\ \rangle\\
 & \mathsf{wp}[\![\ \textbf{assert}\ J;\ \textbf{havoc}\ xs;\ \textbf{assume}\ J;,\\
 & \quad (E\ \Rightarrow\ \mathsf{wp}[\![\ S\ \ \textbf{assert}\ J;\ \textbf{assume false};,\ Q\ ]\!])\ \wedge\ (\neg E\ \Rightarrow\ Q)\ ]\!]\\
= & \quad \langle\ \mathsf{wp}\ \text{of}\ \textbf{assume}\ \rangle\\
 & \mathsf{wp}[\![\ \textbf{assert}\ J;\ \textbf{havoc}\ xs;\ \textbf{assume}\ J;,\\
 & \quad (E\ \Rightarrow\ \mathsf{wp}[\![\ S\ \ \textbf{assert}\ J;,\ \textbf{false}\ \Rightarrow\ Q\ ]\!])\ \wedge\ (\neg E\ \Rightarrow\ Q)\ ]\!]\\
= & \quad \langle\ \text{boolean simplification, and } \mathsf{wp}\ \text{of}\ \textbf{assert}\ \rangle\\
 & \mathsf{wp}[\![\ \textbf{assert}\ J;\ \textbf{havoc}\ xs;\ \textbf{assume}\ J;,\\
 & \quad (E\ \Rightarrow\ \mathsf{wp}[\![\ S,\ J\ \wedge\ \textbf{true}\ ]\!])\ \wedge\ (\neg E\ \Rightarrow\ Q)\ ]\!]\\
= & \quad \langle\ \text{boolean simplification, and } \mathsf{wp}\ \text{of}\ \textbf{assume}\ \rangle\\
 & \mathsf{wp}[\![\ \textbf{assert}\ J;\ \textbf{havoc}\ xs;,\ J\ \Rightarrow\ (E\ \Rightarrow\ \mathsf{wp}[\![\ S,\ J\ ]\!])\ \wedge\ (\neg E\ \Rightarrow\ Q)\ ]\!]\\
= & \quad \langle\ \mathsf{wp}\ \text{of}\ \textbf{havoc}\ \rangle
\end{array}$$

$$
\begin{array}{ll}
 & \mathsf{wp}[\![\, \mathbf{assert}\ J;\, ,\ (\,\forall\, xs \bullet\ J\ \Rightarrow\ (E\ \Rightarrow\ \mathsf{wp}[\![\, S,\ J\,]\!])\ \wedge\ (\neg E\ \Rightarrow\ Q)\ )\,]\!] \\
= & \qquad \langle\ \mathsf{wp}\ \text{of}\ \mathbf{assert}\ \rangle \\
 & J\ \wedge\ (\,\forall\, xs \bullet\ J\ \Rightarrow\ (E\ \Rightarrow\ \mathsf{wp}[\![\, S,\ J\,]\!])\ \wedge\ (\neg E\ \Rightarrow\ Q)\ ) \\
= & \qquad \langle\ \text{distribution of}\ \forall,\ \Rightarrow\ ,\ \text{and}\ \wedge\ \rangle \\
 & J\ \wedge\ (\,\forall\, xs \bullet\ J\ \wedge\ E\ \Rightarrow\ \mathsf{wp}[\![\, S,\ J\,]\!]\ )\ \wedge\ (\,\forall\, xs \bullet\ J\ \wedge\ \neg E\ \Rightarrow\ Q\ )
\end{array}
$$

This formula, which often is given as the way to compute the wp of a loop, lends itself to the following reading: the **while** statement correctly establishes Q if, in the initial state of the loop:

- the loop invariant J holds,
- for any values of the loop targets xs, if the loop invariant and guard hold, then the loop body S maintains the invariant, and
- for any values of the loop targets xs, if the loop invariant and negation of the guard hold, then so does Q.

The meaning of calls Somewhat akin to the way loops are defined via their invariants, procedure calls are defined via their specifications. So, one reasons about a call in terms of the procedure's specification, not its implementation. In addition to avoiding issues of fix-points, this approach has the advantage that it permits information hiding (callers cannot, and do not need to, depend on the details of a particular implementation) and lends itself to modular verification [52].

Consider any procedure:

procedure $P(ins)$ **returns** $(outs)$;
  requires Pre;
  modifies gs;
  ensures $Post$;

and a call to this procedure:

call $xs := P(EE)$;

I encode the meaning of this call statement as follows. First, introduce in the calling context fresh variables ins', out', and gs', one for each variable in ins, $outs$, and gs, respectively. Then, let Pre' be Pre in which each variable from ins is replaced by the corresponding variable from ins'. Let $Post'$ be $Post$ in which each variable from ins is replaced by the corresponding variable from ins', each variable from $outs$ is replaced by the corresponding variable from $outs'$, and each occurrence of a variable from gs inside an **old** expression is replaced by the corresponding variable from gs', after which every $\mathbf{old}(E)$ is replaced by just E. The meaning of the call is:

$ins' := EE$;	// evaluate in-parameters
assert Pre';	// check precondition
$gs' := gs$;	// remember old values of variables in modifies clause
havoc $gs, outs'$;	// set out-parameters and modified global variables to
	// arbitrary values
assume $Post'$;	// ... such that the postcondition holds
$xs := outs'$;	// set actual out-parameters from formal out-parameters

The wp of the call is computed from these statements.

Verifying procedure implementations Every procedure implementation is checked to satisfy its specification. The modifies clause is checked syntactically, the rest of the specification is checked semantically.

The modifies clause of a procedure frames the global variables on which the procedure is allowed to have an effect. This is enforced by a (strict and) simple syntactic check: all global variables among the syntactic assignment targets of the implementation body must be listed in the modifies clause. As I mentioned earlier, the syntactic assignment targets are also allowed to contain local variables and the out-parameters of the procedure, but not the in-parameters, which are immutable in the procedure body.

The pre- and postconditions of the procedure, as well as other proof obligations in the procedure body itself, are checked semantically, that is, by generating a verification condition that is to be discharged by the theorem prover.

The implementation may assume the procedure's preconditions to hold on entry—in other words, it is as if the implementation began with an **assume** statement for each precondition. The postcondition is checked on exit from the procedure—in other words, it is as if the implementation body ended with an **assert** statement for each postcondition.

More formally, consider any procedure:

$$\begin{array}{l}\textbf{procedure}\ P(ins)\ \textbf{returns}\ (outs);\\ \quad \textbf{requires}\ Pre;\\ \quad \textbf{modifies}\ gs;\\ \quad \textbf{ensures}\ Post;\end{array}$$

with an implementation

$$\textbf{var}\ locals;\ \ stmts$$

Let gs' denote a list of fresh variables, one for each variable in gs, and let $stmts'$ denote $stmts$ in which map-update statements, **while** statements, and **call** statements have been expanded according to their semantic encodings given above. Let $Impl$ denote

$$\textbf{assume}\ Pre;\ \ gs' := gs;\ \ stmts'';\ \ \textbf{assert}\ Post';$$

where $stmts''$ and $Post'$ are $stmts'$ and $Post$, respectively, in which each occurrence of a variable from gs inside an **old** expression is replaced by the corresponding variable from gs', after which every $\textbf{old}(E)$ is replaced by just E. Let Axs denote the conjunction of axioms declared in the program. The verification condition for this implementation of P is given by:

$$Axs\ \Rightarrow\ \mathsf{wp}[\![\ Impl,\ \textbf{true}\ ]\!]$$

This formula says that, under the given axioms, the implementation executes correctly.

This concludes the introduction to Boogie and the definition of its semantics. We are now ready to look at the Dafny language in more detail and define, by a Boogie program, its semantics and associated proof obligations.

4. Dafny

A Dafny program consists of a set of named classes:

$$\begin{array}{rcl} \mathit{Program} & ::= & \mathit{Classes} \\ \mathit{Class} & ::= & \textsf{class}\ \mathit{Id}\ \{\ \mathit{Members}\ \} \\ \mathit{Member} & ::= & \mathit{Field} \mid \mathit{Method} \mid \mathit{Function} \end{array}$$

A class declares fields, methods, and functions. In this section, I explain these member declarations, expressions, and statements, and show how to model them by translation into Boogie. The language uses conventional scope and type checking rules; I omit the formal detail thereof and simply assume that the program to be translated has already been properly type checked.

The Boogie translation consists of a prelude of declarations, which encodes some properties of all Dafny programs, and the Boogie declarations $\mathsf{decl}[\![\ d\ ]\!]$ for every Dafny class declaration d.

4.0. Classes

The most interesting part of the translation of a class stems from the members of the class. Each class as a whole merely contributes into the Boogie program a constant that represents the class name. So, the prelude declares a type

$$\textbf{type}\ \mathit{ClassName};$$

and each class declaration is translated as follows:

$$\begin{array}{l} \mathsf{decl}[\![\ \textsf{class}\ C\ \{\ mm\ \}\ ]\!]\ = \\ \quad \textbf{const unique}\ \mathit{class}.C\colon \mathit{ClassName}; \\ \quad \mathsf{decl}^*[\![\ mm\ ]\!] \end{array}$$

Some explanation is in order. First, identifiers in Boogie can include several non-alphanumeric characters, including "$.$", so the name of the constant introduced is "$\mathit{class}.C$". Second, the Boogie modifier **unique** declares that the constant has a value that is different from the values of other **unique** constants. Third, I write $\mathsf{decl}^*[\![\ mm\ ]\!]$ to denote the application of decl to every member in mm, and similarly for other translation functions.

4.1. Fields

A *field*, also known as an *instance variable*, is declared to be of some given type:

$$\begin{array}{rcl} \mathit{Field} & ::= & \textsf{var}\ \mathit{Id} : \mathit{Type}\ ; \\ \mathit{Type} & ::= & \textsf{bool} \mid \textsf{int} \mid \mathit{Id} \mid \textsf{object} \mid \textsf{set<}\mathit{Type}\textsf{>} \mid \textsf{seq<}\mathit{Type}\textsf{>} \end{array}$$

Types include booleans and integers, references to instances of a class, denoted by the name of the class, and references to instances of any class, denoted **object**. In addition, types include sets and sequences of a given type.

To model types, the Dafny-to-Boogie translation introduces into the target Boogie program a nullary type constructor Ref to model references and two unary type constructors Set and Seq to model sets and sequences:

$\textbf{type}\ Ref;$
$\textbf{type}\ Set\ \alpha;$
$\textbf{type}\ Seq\ \alpha;$

The translation into Boogie needs to include appropriately defined operations on sets and sequences, but I omit the details. The set operations are defined in terms of set membership, which in these notes I typeset as the standard "$\in$". The translation also introduces a constant that stands for Dafny's **null** reference:

$\textbf{const}\ null\colon Ref;$

The translation of Dafny types into Boogie goes as follows:

$\mathsf{type}[\![\ \textbf{bool}\ ]\!] = \textbf{bool}$
$\mathsf{type}[\![\ \textbf{int}\ ]\!] = \textbf{int}$
$\mathsf{type}[\![\ Id\ ]\!] = Ref$
$\mathsf{type}[\![\ \textbf{object}\ ]\!] = Ref$
$\mathsf{type}[\![\ \textbf{set}\langle T\rangle\ ]\!] = Set\ \mathsf{type}[\![\ T\ ]\!]$
$\mathsf{type}[\![\ \textbf{seq}\langle T\rangle\ ]\!] = Seq\ \mathsf{type}[\![\ T\ ]\!]$

Note that the types in the Boogie translation are more coarsely grained than in the Dafny program. In particular, all Dafny class types are modeled by the Boogie type Ref. To distinguish between references of different Dafny types, the translation includes a function that maps each reference to its allocated type:

$\textbf{function}\ dtype(Ref)\ \textbf{returns}\ (ClassName);$

When modeling the semantics of a source language, one of the most important considerations is the decision of how to model memory. Dafny includes dynamically allocated objects and references to these. Different possibilities exist for how to model such a memory. For a discussion of these, see [36]; for some specific choices, see [32,53,54,42,4,20,11]. Here, I choose to model memory as a map from object references and field names to values. Updates of the memory are then modeled as updates of this map. The declarations introduced by the translation into Boogie are:

$\textbf{type}\ Field\ \alpha;$
$\textbf{type}\ HeapType = \langle\alpha\rangle[Ref,\ Field\ \alpha]\,\alpha;$
$\textbf{var}\ \mathcal{H}\colon HeapType;$

Global variable $\mathcal{H}$ represents the memory, the heap. Its type is a polymorphic map that for any type α maps each $(Ref, Field\ \alpha)$ pair to an α.

Each field in the Dafny program gives rise to a distinct value of the appropriate $Field$ type. For any field f in a class C:

$\mathsf{decl}[\![\ \textbf{var}\ f\colon T;\ ]\!] =$
$\quad \textbf{const unique}\ C.f\colon\ Field\ \mathsf{type}[\![\ T\ ]\!]\ ;$

Conveniently, Boogie's admission of non-alphanumeric characters like "." in identifiers makes it easy to produce readable, fully qualified names.

Since map types in Boogie are total, the heap maps all possible references and fields to values. It even maps reference-field pairs to values when such a reference-field pair

would be ill-typed in Dafny, but the target Boogie program never makes use of the values at such reference-field pairs. Also, the domain of the map includes allocated as well as unallocated references. To distinguish between these two, I add a ghost field $alloc$ that I arrange to set to $\mathbf{true}$ when an object is allocated:

$$\mathbf{const\ unique}\ alloc\colon Field\ \mathbf{bool};$$

The heap of a Dafny program has properties that not every polymorphic map has. To distinguish Dafny heaps from other maps, the translation introduces a predicate

$$\mathbf{function}\ GoodHeap(HeapType)\ \mathbf{returns}\ (\mathbf{bool});$$

Various axioms state properties that hold of all heaps. For example, if the Dafny program includes a class C with a field f of a reference type D, then the following axiom is included to say that $C.f$ yields a value of the appropriate type and that $C.f$ is closed under allocation, that is, that an allocated object only reaches allocated objects:

$$\begin{array}{l}\mathbf{axiom}\ (\forall\, h\colon HeapType,\ o\colon Ref\ \bullet\\ \quad GoodHeap(h) \wedge o \neq null \wedge h[o, alloc]\\ \quad\ \Rightarrow \mathsf{GoodRef}[\![\, h[o, C.f],\ D,\ h\,]\!]\ );\end{array}$$

Translation function $\mathsf{GoodRef}$ is defined as follows:

$$\mathsf{GoodRef}[\![\, t, T, h\,]\!] = \begin{cases} t = null \vee (h[t, alloc] \wedge dtype(t) = class.T) & \text{if } T \text{ is a class name}\\ t = null \vee h[t, alloc] & \text{if } T \text{ is } \textbf{object}\end{cases}$$

Note that the type **object** gives no information about the allocated type of an object.

If the type of the field f is a set of D references, then the following axiom is introduced:

$$\begin{array}{l}\mathbf{axiom}\ (\forall\, h\colon HeapType,\ o\colon Ref,\ t\colon Ref\ \bullet\\ \quad GoodHeap(h) \wedge o \neq null \wedge h[o, alloc] \wedge t \in h[o, C.f]\\ \quad\ \Rightarrow \mathsf{GoodRef}[\![\, t, D, h\,]\!]\ );\end{array}$$

This axiom is what allows the verifier to prove that new objects are not in any footprint (see the discussion about the correctness of Main at the end of Section 1.2). If the type of the field f is a sequence of D references, then the axiom introduced is:

$$\begin{array}{l}\mathbf{axiom}\ (\forall\, h\colon HeapType,\ o\colon Ref,\ i\colon Ref\ \bullet\\ \quad GoodHeap(h) \wedge o \neq null \wedge h[o, alloc] \wedge\\ \quad 0 \leqslant i \wedge i < SeqLength(h[o, C.f])\\ \quad\ \Rightarrow \mathsf{GoodRef}[\![\, SeqIndex(h[o, C.f], i),\ D,\ h\,]\!]\ );\end{array}$$

The translation also includes similar axioms for fields of more complex types, like sets of sets of references and sets of sequences of references.

4.2. Expressions

Dafny expressions include common boolean, integer, set, and sequence operators. Their semantics is defined by two translation functions: $\mathsf{df}[\![\, E\,]\!]$ generates a Boogie predicate

that says whether or not E is well defined in Dafny, and $\mathsf{tr}[\![\, E \,]\!]$ generates an expression that for any well-defined E evaluates to E. Most of these definitions are straightforward; the following examples illustrate the idea:

$\cdot$	$\mathsf{df}[\![\, \cdot \,]\!]$	$\mathsf{tr}[\![\, \cdot \,]\!]$
x	$\mathbf{true}$	x
this	$\mathbf{true}$	$this$
$E + F$	$\mathsf{df}[\![\, E \,]\!] \wedge \mathsf{df}[\![\, F \,]\!]$	$\mathsf{tr}[\![\, E \,]\!] + \mathsf{tr}[\![\, F \,]\!]$
$E \,/\, F$	$\mathsf{df}[\![\, E \,]\!] \wedge \mathsf{df}[\![\, F \,]\!] \wedge \mathsf{tr}[\![\, F \,]\!] \neq 0$	$\mathsf{tr}[\![\, E \,]\!] \,/\, \mathsf{tr}[\![\, F \,]\!]$
$E \wedge F$	$\mathsf{df}[\![\, E \,]\!] \wedge (\mathsf{tr}[\![\, E \,]\!] \Rightarrow \mathsf{df}[\![\, F \,]\!])$	$\mathsf{tr}[\![\, E \,]\!] \wedge \mathsf{tr}[\![\, F \,]\!]$
$E.f$	$\mathsf{df}[\![\, E \,]\!] \wedge \mathsf{tr}[\![\, E \,]\!] \neq null$	$\mathcal{H}[\mathsf{tr}[\![\, E \,]\!], C.f]$

In the last line above, C denotes the class that defines field f. Note that in Dafny, the "." in the expression $E.f$ denotes member selection, whereas in Boogie, the "." in $C.f$ is just another character in the identifier's name. Also, note the encoding of Dafny's short-circuit boolean operators: the definedness of F matters only if E evaluates to true, and in this left-to-right fashion, one can assume E to hold when verifying the definedness of F. I alluded to this left-to-right rule when discussing the interplay between Valid and footprint in Section 1.0.

Like Boogie, Dafny supports two-state expressions, that is, expressions that refer not only to the current state but also to the initial state of the method. Two-state expressions are allowed in postconditions and in method bodies. One two-state expression is **old**(E), which stands for the value of E on entry to the enclosing method. Another two-state expression is **fresh**(E), where E is a set of object references. It says that each of the references is either **null** or was not allocated in the method's pre-state. Finally, if E is of a reference type, then the two-state expression **fresh**(E) is a shorthand for **fresh**($\{E\}$). Formally:

$\cdot$	$\mathsf{df}[\![\, \cdot \,]\!]$	$\mathsf{tr}[\![\, \cdot \,]\!]$
old(E)	$\mathbf{old}(\mathsf{df}[\![\, E \,]\!])$	$\mathbf{old}(\mathsf{tr}[\![\, E \,]\!])$
fresh(E)	$\mathsf{df}[\![\, E \,]\!]$	$(\forall\, o\colon Ref \;\bullet\; o \in \mathsf{tr}[\![\, E \,]\!] \Rightarrow$ $o = null \vee \neg\mathbf{old}(\mathcal{H})[o, alloc]\,)$

The translations of set and sequence constructors, like $\{\,\}$, $\{a, b\}$, $[\,]$, and $[x, y, z]$, are also interesting, but I omit the details, which depend on the axiomatization of sets and sequences. I will present the translation of calls to user-defined Dafny functions in Section 4.5.

4.3. Methods

A *method* is a class-bound procedure that can change the program state. It is declared with a specification and an implementation.

$$\begin{array}{rcl} Method & ::= & \textbf{method}\ Id(Params)\ \textbf{returns}\ (Params)\ Specs\ \{\ Stmts\ \} \\ Param & ::= & Id : Type \\ Spec & ::= & \textbf{requires}\ Expr\ ; \mid \textbf{modifies}\ Exprs\ ; \mid \textbf{ensures}\ Expr\ ; \end{array}$$

The **returns** clause can be dropped if there are no out-parameters. As in Boogie, the **requires** and **ensures** clauses declare pre- and postconditions, and multiple specifica-

$$
\begin{array}{l}
\mathsf{decl}[\![\ \textbf{method}\ M(ins)\ \textbf{returns}\ (outs) \\
\quad\quad \textbf{requires}\ Pre; \\
\quad\quad \textbf{modifies}\ mts; \\
\quad\quad \textbf{ensures}\ Post; \\
\quad \{\ stmts\ \}\ ]\!] \\
= \\
\quad \textbf{procedure}\ C.M(this\colon Ref,\ \mathsf{decl}^*[\![\ ins\ ]\!])\ \textbf{returns}\ (\mathsf{decl}^*[\![\ outs\ ]\!]) \\
\quad\quad \textbf{free requires}\ GoodHeap(\mathcal{H}) \wedge CanAssumeFunctionDefs; \\
\quad\quad \textbf{free requires}\ this \neq null \wedge \mathsf{GoodRef}[\![\ this, C, \mathcal{H}\ ]\!]; \\
\quad\quad \textbf{free requires}\ \mathsf{isAllocated}^*[\![\ ins\ ]\!]; \\
\quad\quad \textbf{free requires}\ \mathsf{df}[\![\ Pre\ ]\!]; \\
\quad\quad \textbf{requires}\ \mathsf{tr}[\![\ Pre\ ]\!]; \\
\quad\quad \textbf{modifies}\ \mathcal{H}; \\
\quad\quad \textbf{free ensures}\ GoodHeap(\mathcal{H}); \\
\quad\quad \textbf{free ensures}\ \mathsf{boilerplate}_{mts}[\![\ \textbf{old}(\mathcal{H})\ ]\!]; \\
\quad\quad \textbf{free ensures}\ \mathsf{isAllocated}^*[\![\ outs\ ]\!]; \\
\quad\quad \textbf{free ensures}\ \mathsf{df}[\![\ Post\ ]\!]; \\
\quad\quad \textbf{ensures}\ \mathsf{tr}[\![\ Post\ ]\!]; \\
\quad \{\ \textbf{var}\ \mathsf{locals}^*[\![\ stmts\ ]\!];\ \ \mathsf{stmt}^*_{mts}[\![\ stmts\ ]\!]\ \} \\
\quad \textbf{procedure}\ C.M.WellDefined\ \ldots \text{(see text)}
\end{array}
$$

$$
\mathsf{decl}[\![\ x\colon T\ ]\!]\ =\ x\ :\ \mathsf{type}[\![\ T\ ]\!]
$$

$$
\mathsf{isAllocated}[\![\ x\colon T\ ]\!]\ =\ \begin{cases} \mathsf{GoodRef}[\![\ x, T, \mathcal{H}\ ]\!] & \text{if } T \text{ is a reference type} \\ \textbf{true} & \text{otherwise} \end{cases}
$$

$$
\begin{array}{l}
\mathsf{boilerplate}_{mts}[\![\ prevHeap\ ]\!]\ = \\
\quad (\forall\, \langle\alpha\rangle\ o\colon Ref,\ f\colon Field\ \alpha \bullet \\
\quad\quad\quad \mathcal{H}[o,f] = prevHeap[o,f] \vee \mathsf{CanWrite}_{mts}[\![\ o\ ]\!]\ ) \wedge \\
\quad (\forall\, o\colon Ref \bullet\ prevHeap[o, alloc] \Rightarrow \mathcal{H}[o, alloc]\ )
\end{array}
$$

$$
\begin{array}{l}
\mathsf{CanWrite}_{mts}[\![\ o\ ]\!]\ = \\
\quad o \in \textbf{old}(\mathsf{tr}[\![\ mts\ ]\!]) \vee \neg\textbf{old}(\mathcal{H})[o, alloc]
\end{array}
$$

Figure 9. The translation into a Boogie procedure of a Dafny method M declared in a class C.

tion clauses of the same kind can be combined. Unlike in Boogie, the Dafny **modifies** clause takes a list of expressions. The type of each such expression must be a reference type or a set of references; an expression E of a reference type is simply a shorthand for the singleton $\{E\}$. The **modifies** clause says that the method may modify the state of any object referenced in the set. In addition, the method is allowed to allocate new objects and modify their state.

The specification is used when reasoning about a call to the method. The given method implementation is checked to satisfy the specification.

A method is translated into a procedure in Boogie, making the implicit receiver parameter **this** explicit, see Fig. 9. As I am about to describe, the specification of the procedure comes not only from the method's specification, but also from the types of the parameters, properties that hold of all Dafny programs, and details of the encoding.

In the procedure specification in Fig. 9, some of the specification clauses are marked with **free**, which in Boogie means that they are assumed but not checked [36]. Some

properties needed in the verification, like $this \neq null \ \wedge \ \mathcal{H}[this, alloc]$, hold in every execution of the source language. There is no reason to spend time proving these with every program verification. Rather, such properties can be justified by a meta-argument (which can be formalized and proved once and for all, perhaps using a mechanical proof assistant) and are then conveniently introduced in the Boogie translation with **free** conditions.

I will now describe the specification clauses shown in Fig. 9, but in different order than they appear in the figure.

First, the Dafny method pre- and postconditions Pre and $Post$ are translated into corresponding pre- and postconditions for the procedure:

```
free requires df⟦ Pre ⟧;
requires tr⟦ Pre ⟧;
free ensures df⟦ Post ⟧;
ensures tr⟦ Post ⟧;
```

The definedness conditions for Pre and $Post$ are marked as **free**, because they are checked separately by the following procedure:

```
procedure C.M.WellDefined(this: Ref, decl*⟦ ins ⟧) returns (decl*⟦ outs ⟧)
  free requires GoodHeap(H) ∧ CanAssumeFunctionDefs;
  free requires this ≠ null ∧ GoodRef⟦ this, C, H ⟧;
  free requires isAllocated*⟦ ins ⟧;
  modifies H;
{
  assert df⟦ Pre ⟧;
  assume tr⟦ Pre ⟧;
  havoc H;
  assume GoodHeap(H) ∧ boilerplate_mts⟦ old(H) ⟧;
  assume isAllocated*⟦ outs ⟧;
  assert df⟦ Post ⟧;
}
```

An alternative to generating procedure $C.M.WellDefined$ would be to include Pre and $Post$ in procedure $C.M$ (Fig. 9) simply as:

```
requires df⟦ Pre ⟧ ∧ tr⟦ Pre ⟧;
ensures df⟦ Post ⟧ ∧ tr⟦ Post ⟧;
```

Although slightly simpler, this alternative has the drawback of having to verify the definedness of Pre at every call site.

Second, as we shall see in Section 4.4, **modifies** clauses in Dafny are enforced at every statement. Therefore, a method's effect on the heap shows up in the encoding as a **free** condition [29,57], and it also shows up in the syntactically enforced Boogie **modifies** clause that allows $\mathcal{H}$ to be an assignment target in the procedure body:

```
modifies H;
free ensures boilerplate_mts⟦ old(H) ⟧;
```

The first conjunct of boilerplate (defined in Fig. 9) says that a heap location (o, f) can have changed only if o is an object reference included in the set denoted by the **modifies**

clause mts, which is interpreted in the method's pre-state, or if o is a newly allocated object reference. The second conjunct mentions the possible ways that the $alloc$ field can change, namely that allocated objects remain allocated, which is a property of the Dafny language. Note that boilerplate mentions three heaps: the current heap $\mathcal{H}$, the heap $\mathbf{old}(\mathcal{H})$ at the beginning of the method, and the heap parameter $prevHeap$. Here, $prevHeap$ is instantiated with $\mathbf{old}(\mathcal{H})$, but for loops (Section 4.4), it is instantiated with the pre-loop heap.

Third, the values passed as in- and out-parameters in Dafny are members of the respective parameter types and, for reference types, are allocated. This is encoded in the specifications:

$$\begin{array}{l}\mathbf{requires}\ this \neq null \wedge \mathsf{GoodRef}[\![\, this, C, \mathcal{H} \,]\!];\\ \mathbf{requires}\ \mathsf{isAllocated}^*[\![\, ins \,]\!];\\ \mathbf{ensures}\ \mathsf{isAllocated}^*[\![\, outs \,]\!];\end{array}$$

where GoodRef is defined in Section 4.1 and isAllocated is defined in Fig. 9.

Fourth, the heap on entry to and exit from the method is a map that satisfies our axiomatized heap properties. Therefore, the procedure encoding includes:

$$\begin{array}{l}\mathbf{free\ requires}\ GoodHeap(\mathcal{H});\\ \mathbf{free\ ensures}\ GoodHeap(\mathcal{H});\end{array}$$

Fifth, functions declared in Dafny give rise to axioms that can be used only in certain places. As I describe in Section 4.5, those axioms mention a constant

$$\mathbf{const}\ CanAssumeFunctionDefs\colon \mathbf{bool};$$

in their antecedent. Since the axioms are allowed to be used when reasoning about method bodies, the encoding of the method includes the precondition

$$\mathbf{free\ requires}\ CanAssumeFunctionDefs;$$

The assumptions about $GoodHeap$ and isAllocated that I have shown here also need to be done on loop boundaries and for local variables. Boogie provides **where** clauses that automatically insert such assumptions. I refer to the Boogie 2 language reference manual for details [36].

4.4. Statements

Statements follow the grammar in Fig. 10. The **else** branch can be dropped if its second $Stmts$ is empty, and the " := " is dropped if the list of actual out-parameters xs in the **call** statement is empty. The two identifiers (x) in the **foreach** statement must be the same.

In Dafny, local variables can be declared among the statements, whereas Boogie lists all local variables up front in the procedure body. Translation function locals, defined in Fig. 11, returns the set of Boogie local variables that the translation of each Dafny statement gives rise to. The translation function locals is used in the procedure body shown in Fig. 9.

Dafny statements are translated into Boogie as shown in Fig. 12. I give the following explanations.

$$
\begin{array}{rcl}
\mathit{Stmt} & ::= & \textbf{var}\ x : \mathit{Type}\ ; \\
& | & x := \mathit{Expr}\ ; \\
& | & \mathit{Expr}\ .\ f\ :=\ \mathit{Expr}\ ; \\
& | & x := \textbf{new}\ T\ ; \\
& | & \textbf{assert}\ \mathit{Expr}\ ; \\
& | & \textbf{if}\ (\mathit{Expr})\ \{\ \mathit{Stmts}\ \}\ \textbf{else}\ \{\ \mathit{Stmts}\ \} \\
& | & \textbf{while}\ (\mathit{Expr})\ \mathit{Invs}\ \{\ \mathit{Stmts}\ \} \\
& | & \textbf{foreach}\ (x \in \mathit{Expr})\ \{\ x.f := \mathit{Expr}\ ;\ \} \\
& | & \textbf{call}\ xs := \mathit{Expr}\ .\ \mathit{Id}(\mathit{Exprs})\ ; \\
\mathit{Inv} & ::= & \textbf{invariant}\ \mathit{Expr}\ ;
\end{array}
$$

Figure 10. Grammar of Dafny statements, where x denotes any variable identifier, xs denotes any list of distinct variables, f denotes any field, and T denotes any class.

$$
\begin{array}{l}
\mathsf{locals}[\![\ \textbf{var}\ x\colon T;\ ]\!] \ =\ x : \mathsf{type}[\![\ T\ ]\!] \\
\mathsf{locals}[\![\ x := E;\ ]\!] \ =\ \text{// empty} \\
\mathsf{locals}[\![\ E.f := E;\ ]\!] \ =\ \text{// empty} \\
\mathsf{locals}[\![\ x := \textbf{new}\ T;\ ]\!] \ =\ \text{// empty} \\
\mathsf{locals}[\![\ \textbf{assert}\ E;\ ]\!] \ =\ \text{// empty} \\
\mathsf{locals}[\![\ \textbf{if}\ (E)\ \{\ S0\ \}\ \textbf{else}\ \{\ S1\ \}\ ]\!] \ =\ \mathsf{locals}[\![\ S0\ ]\!],\ \mathsf{locals}[\![\ S1\ ]\!] \\
\mathsf{locals}[\![\ \textbf{while}\ (E)\ \mathit{invs}\ \{\ S\ \}\ ]\!] \ =\ \mathit{prevHeap},\ \mathsf{locals}[\![\ S\ ]\!] \\
\mathsf{locals}[\![\ \textbf{foreach}\ (x \in R)\ \{\ x.f := E;\ \}\ ]\!] \ =\ \mathit{prevHeap} \\
\mathsf{locals}[\![\ \textbf{call}\ xs := E.M(EE);\ ]\!] \ =\ \text{// empty}
\end{array}
$$

Figure 11. A translation function that collects the Boogie local variables used by the translation. I assume the local variables in Dafny have first been suitably renamed to avoid name clashes. For each **while** and **foreach** statement, $\mathit{prevHeap}$ denotes a fresh variable introduced for the translation of that statement.

After a Dafny local-variable declaration has been moved to the beginning of the Boogie procedure, what remains in the translation of **var** x:T; is $\mathbf{havoc}\ x;$, which ensures that the variable starts off with an arbitrary value, even if it occurs inside a loop.

The assignment to a local variable translates into an assignment in Boogie, preceded by a check that the right-hand side is defined.

A field update x.f := E; turns into a heap update, roughly like $\mathcal{H}[x, f] := E;$, preceded by various checks. The checks enforce that all expressions involved are defined, that **null** is not dereferenced, and that the enclosing method's **modifies** clause permits the update. Like all statements that directly update the heap, the translation of the field-update statement ends by saying the resulting heap satisfies the $\mathit{GoodHeap}$ predicate.

Using the havoc-assume idiom, the allocation statement sets its left-hand side to an arbitrary non-null, unallocated object reference of the appropriate type. It then marks that object reference as being allocated.

Dafny's **assert** statement first checks that the given expression is defined and then checks that it evaluates to true.

Similarly, Dafny's **if** statement translates into Boogie's $\mathbf{if}$ statement, preceded by a check that the guard is defined.

For every **while** statement (and also every **foreach** statement), the translation introduces a new local variable $\mathit{prevHeap}$. The encoding starts by recording into $\mathit{prevHeap}$

$$
\begin{array}{l}
\mathsf{stmt}_{mts}[\![\ \textsf{var}\ x\colon T;\ ]\!]\ = \\
\quad \textbf{havoc}\ x; \\
\mathsf{stmt}_{mts}[\![\ x := E;\ ]\!]\ = \\
\quad \textbf{assert}\ \mathsf{df}[\![\ E\ ]\!]; \\
\quad x := \mathsf{tr}[\![\ E\ ]\!]; \\
\mathsf{stmt}_{mts}[\![\ E0.f := E1;\ ]\!]\ = \\
\quad \textbf{assert}\ \mathsf{df}[\![\ E0\ ]\!] \wedge \mathsf{df}[\![\ E1\ ]\!] \wedge \mathsf{tr}[\![\ E\ ]\!] \neq \mathit{null}; \\
\quad \textbf{assert}\ \mathsf{CanWrite}_{mts}[\![\ \mathsf{tr}[\![\ E0\ ]\!]\ ]\!]; \\
\quad \mathcal{H}[\mathsf{tr}[\![\ E0\ ]\!], C.f] := \mathsf{tr}[\![\ E1\ ]\!]; \\
\quad \textbf{assume}\ \mathit{GoodHeap}(\mathcal{H}); \\
\mathsf{stmt}_{mts}[\![\ x := \textsf{new}\ T;\ ]\!]\ = \\
\quad \textbf{havoc}\ x;\ \ \textbf{assume}\ x \neq \mathit{null} \wedge \neg\mathcal{H}[x, \mathit{alloc}] \wedge \mathit{dtype}(x) = \mathit{class}.T; \\
\quad \mathcal{H}[x, \mathit{alloc}] := \textbf{true}; \\
\quad \textbf{assume}\ \mathit{GoodHeap}(\mathcal{H}); \\
\mathsf{stmt}_{mts}[\![\ \textsf{assert}\ E;\ ]\!]\ = \\
\quad \textbf{assert}\ \mathsf{df}[\![\ E\ ]\!]; \\
\quad \textbf{assert}\ \mathsf{tr}[\![\ E\ ]\!]; \\
\mathsf{stmt}_{mts}[\![\ \textsf{if}\ (E)\ \{\ S0\ \}\ \textsf{else}\ \{\ S1\ \}\ ]\!]\ = \\
\quad \textbf{assert}\ \mathsf{df}[\![\ E\ ]\!]; \\
\quad \textbf{if}\ (\mathsf{tr}[\![\ E\ ]\!])\ \{\ \mathsf{stmt}^*_{mts}[\![\ S0\ ]\!]\ \}\ \textbf{else}\ \{\ \mathsf{stmt}^*_{mts}[\![\ S1\ ]\!]\ \} \\
\mathsf{stmt}_{mts}[\![\ \textsf{while}\ (E)\ \textsf{invariant}\ J;\ \{\ S\ \}\ ]\!]\ = \\
\quad \mathit{prevHeap} := \mathcal{H}; \\
\quad \textbf{while}\ (\mathsf{tr}[\![\ E\ ]\!]) \\
\quad\quad \textbf{invariant}\ \mathsf{df}[\![\ J\ ]\!] \wedge \mathsf{tr}[\![\ J\ ]\!]; \\
\quad\quad \textbf{invariant}\ \mathsf{df}[\![\ E\ ]\!]; \\
\quad\quad \textbf{free invariant}\ \mathsf{boilerplate}_{mts}[\![\ \mathit{prevHeap}\ ]\!]; \\
\quad \{\ \mathsf{stmt}^*_{mts}[\![\ S\ ]\!]\ \} \\
\mathsf{stmt}_{mts}[\![\ \textsf{foreach}\ (x \in R)\ \{\ x.f := E;\ \}\ ]\!]\ = \\
\quad \textbf{assert}\ \mathsf{df}[\![\ R\ ]\!]; \\
\quad \textbf{assert}\ (\forall\, x\colon \mathit{Ref}\ \bullet\ x \in \mathsf{tr}[\![\ R\ ]\!]\ \Rightarrow\ x \neq \mathit{null} \wedge \mathsf{df}[\![\ E\ ]\!]\ ); \\
\quad \textbf{assert}\ (\forall\, x\colon \mathit{Ref}\ \bullet\ x \in \mathsf{tr}[\![\ R\ ]\!]\ \Rightarrow\ \mathsf{CanWrite}_{mts}[\![\ x\ ]\!]\ ); \\
\quad \mathit{prevHeap} := \mathcal{H}; \\
\quad \textbf{havoc}\ \mathcal{H}; \\
\quad \textbf{assume}\ (\forall\, \langle\alpha\rangle\ o\colon \mathit{Ref},\ g\colon \mathit{Field}\ \alpha\ \bullet \\
\quad\quad\quad\quad \mathcal{H}[o, g] = \mathit{prevHeap}[o, g] \vee (o \in \mathsf{tr}[\![\ R\ ]\!] \wedge g = C.f)\ ); \\
\quad \textbf{assume}\ (\forall\, x\colon \mathit{Ref}\ \bullet\ \mathcal{H}[x, C.f] = \mathsf{tr}[\![\ E\ ]\!][\![\mathit{prevHeap}\,/\!/\,\mathcal{H}]\!]\ ); \\
\quad \textbf{assume}\ \mathit{GoodHeap}(\mathcal{H}); \\
\mathsf{stmt}_{mts}[\![\ \textsf{call}\ xs := E.M(EE);\ ]\!]\ = \\
\quad \textbf{assert}\ \mathsf{df}[\![\ E\ ]\!] \wedge \mathsf{df}^*[\![\ EE\ ]\!] \wedge \mathsf{tr}[\![\ E\ ]\!] \neq \mathit{null}; \\
\quad \textbf{assert}\ (\forall\, o\colon \mathit{Ref}\ \bullet\ o \in \mathsf{tr}[\![\ MT[\![EE\,/\!/\,\mathit{args}]\!]\ ]\!]\ \Rightarrow\ \mathsf{CanWrite}_{mts}[\![\ o\ ]\!]\ ); \\
\quad \textbf{call}\ xs := C.M(\mathsf{tr}[\![\ E\ ]\!], \mathsf{tr}^*[\![\ EE\ ]\!]);
\end{array}
$$

Figure 12. Translation of Dafny statements into Boogie statements. C is used to denote the class that declares field f and method M. In the **while** and **foreach** statements, $\mathit{prevHeap}$ denotes the variable introduced by the translation for this statement; this is the same variable as is returned by the translation function locals (Fig. 11). The capture-avoiding substitution $\mathsf{tr}[\![\ E\ ]\!][\![\mathit{prevHeap}\,/\!/\,\mathcal{H}]\!]$ in the **foreach** statement leaves **old** expressions unchanged. For the call, MT denotes the **modifies** clause of the method that is called and args denotes its formal in-parameters.

the value of the heap on entry to the loop. The target loop invariant checks, at the top of each loop iteration, that the loop invariant is defined, that the loop invariant holds, and that the loop guard is defined. The encoding also records the **free** condition that the current heap stands in relation to the pre-loop heap according the enclosing method's **modifies** clause, because that **modifies** clause is enforced in the translation of the loop body.

When discussing the meaning of Boogie loops in Section 3, I mentioned four contributors to loop invariants. The second conjunct of the boilerplate predicate is an (i0) contributor, a property that always holds in the source language. The first conjunct of boilerplate is an (i1) contributor, because Dafny's programming discipline enforces the rule of checking **modifies** clauses at each update. And the checking of the loop invariant itself is an (i3) contributor.

The **foreach** statement checks that all expressions involved are defined and that the enclosing method is allowed to update the field $x.f$ for every x in R. Using the havoc-assume idiom, it then changes the heap at the affected fields (and only those—as prescribed by the first quantification) to set these to the right-hand E (as prescribed by the second quantification). Note that the syntax of the **foreach** statement, which forces the left-hand side of the assignment to be a field of an object reference drawn from a set, guarantees that the fields to be updated are distinct; hence, the order of the assignments is immaterial.

Finally, a method call in Dafny is translated into a call to the corresponding procedure in Boogie, after checking the definedness of all expressions involved and checking that the caller is allowed to update the memory locations that the callee may update.

4.5. Functions

So far, we have the basic verification machinery in place, but there is no abstraction in the language to allow us to specify programs modularly. Functions provide that abstraction.

$$\begin{array}{rcl} \mathit{Function} & ::= & \textbf{function}\ \mathit{Id}(\mathit{Params}) : \mathit{Type}\ \mathit{FSpecs}\ \{\ \mathit{Expr}\ \} \\ \mathit{Param} & ::= & \mathit{Id} : \mathit{Type} \\ \mathit{FSpec} & ::= & \textbf{requires}\ \mathit{Expr}\ ; \mid \textbf{reads}\ \mathit{Exprs}\ ; \end{array}$$

Multiple **requires** clauses are equivalent to one **requires** clause that conjoins the conditions, and multiple **reads** clauses are equivalent to one **reads** clause that gives all the expressions.

In the rest of this subsection, I consider what to do for a function F declared as follows in a class C:

function $F(ins)$: T **requires** R; **reads** rd; { $body$ }

A Dafny function is modeled by a Boogie function, so $\mathsf{decl}[\![$ **function** $F \ldots]\!]$ produces:

$$\textbf{function}\ C.F(h\colon \mathit{HeapType},\ \mathit{this}\colon \mathit{Ref},\ \mathsf{decl}^*[\![\ \mathit{ins}\ ]\!])\ \textbf{returns}\ (\mathsf{type}[\![\ T\ ]\!]);$$

It also produces a procedure and two axioms, as I will describe shortly.

A function can be used in expressions. It is defined wherever its precondition holds:

$$
\begin{array}{l}
\mathsf{df}[\![\, E.F(EE) \,]\!] \;= \\
\quad \mathsf{df}[\![\, E \,]\!] \wedge \mathsf{df}^*[\![\, EE \,]\!] \wedge \mathsf{tr}[\![\, E \,]\!] \neq \mathit{null} \wedge \\
\quad \mathsf{df}[\![\, R[\![EE \mathbin{/\!/} \mathit{ins}]\!] \,]\!] \wedge \mathsf{tr}[\![\, R[\![EE \mathbin{/\!/} \mathit{ins}]\!] \,]\!] \\
\mathsf{tr}[\![\, E.F(EE) \,]\!] \;= \\
\quad C.F(\mathcal{H},\ \mathsf{tr}[\![\, E \,]\!],\ \mathsf{tr}^*[\![\, EE \,]\!])
\end{array}
$$

To reason about a function $C.F$, the translation into Boogie introduces a couple of axioms about $C.F$.

The first axiom uses the precondition and body of the function, and it gives a precise definition of the value returned by the function:

$$
\begin{array}{l}
\textbf{axiom}\ \mathit{CanAssumeFunctionDefs} \;\Rightarrow \\
\quad (\,\forall\, \mathcal{H}\colon \mathit{HeapType},\ \mathit{this}\colon \mathit{Ref},\ \mathsf{decl}^*[\![\, \mathit{ins} \,]\!] \bullet \\
\qquad \mathit{GoodHeap}(\mathcal{H}) \wedge \mathit{this} \neq \mathit{null} \wedge \mathsf{df}[\![\, R \,]\!] \wedge \mathsf{tr}[\![\, R \,]\!] \\
\qquad\ \Rightarrow\ C.F(\mathcal{H}, \mathit{this}, \mathit{ins}) = \mathsf{tr}[\![\, \mathit{Body} \,]\!]\ )
\end{array}
$$

Generating an axiom to define a possibly-recursive function is dicey, because the definition may be inconsistent (*cf.* [13,0,56]). If the axiom generated is inconsistent, one can prove anything, including any proof obligation intended to check the function definition to be consistent! For that reason, the translation uses $\mathit{CanAssumeFunctionDefs}$ as an antecedent of the axioms produced. Since the verification of methods is allowed to use the function axioms, the procedures to which methods translate include $\mathit{CanAssumeFunctionDefs}$ as a **free** precondition. The justification for making this precondition **free** is that the obligation to prove the function axioms consistent rests with special function well-definedness checks, described next.

To check that the function is well defined, the translation generates a proof obligation that all calls go to functions with a strictly smaller **reads** clause. The proof obligation is formulated as a Boogie procedure; discharging the proof obligation thus comes down to verifying the correctness of the procedure's implementation:

$$
\begin{array}{l}
\textbf{procedure}\ C.F.\mathit{WellDefined}(\mathit{this}\colon \mathit{Ref},\ \mathsf{decl}^*[\![\, \mathit{ins} \,]\!]) \\
\quad \textbf{free requires}\ \mathit{GoodHeap}(\mathcal{H}); \\
\quad \textbf{free requires}\ \mathit{this} \neq \mathit{null} \wedge \mathsf{GoodRef}[\![\, \mathit{this}, C, \mathcal{H} \,]\!]; \\
\quad \textbf{free requires}\ \mathsf{isAllocated}^*[\![\, \mathit{ins} \,]\!]; \\
\{ \\
\quad \textbf{assume}\ \mathsf{df}[\![\, R \,]\!] \wedge \mathsf{tr}[\![\, R \,]\!]; \\
\quad \textbf{assert}\ \mathsf{funcdf}_{rd}[\![\, \mathit{body} \,]\!]; \\
\}
\end{array}
$$

Translation function funcdf is like df, except for field selection expressions and function calls, which check that the heap is read in accordance with a given **reads** set r:

$$
\begin{array}{l}
\mathsf{funcdf}_r[\![\, E.f \,]\!] \;= \\
\quad \mathsf{funcdf}_r[\![\, E \,]\!] \wedge \mathsf{tr}[\![\, E \,]\!] \neq \mathit{null} \wedge \mathsf{tr}[\![\, E \,]\!] \in \mathsf{tr}[\![\, r \,]\!] \\
\mathsf{funcdf}_r[\![\, E.G(EE) \,]\!] \;= \\
\quad \mathsf{funcdf}_r[\![\, E \,]\!] \wedge \mathsf{funcdf}_r^*[\![\, EE \,]\!] \wedge \mathsf{tr}[\![\, E \,]\!] \neq \mathit{null} \wedge \\
\quad \mathsf{funcdf}_r[\![\, Q[\![EE \mathbin{/\!/} \mathit{args}]\!] \,]\!] \wedge \mathsf{tr}[\![\, Q[\![EE \mathbin{/\!/} \mathit{args}]\!] \,]\!] \wedge \\
\quad \mathsf{tr}[\![\, s \,]\!] \subsetneq \mathsf{tr}[\![\, r \,]\!]
\end{array}
$$

where G is a function declared with in-parameters args, precondition Q, and **reads** clause s. (We can be a little bit more permissive for boolean functions: if the definition

of a boolean function calls another boolean function in a positive position, then it suffices to check that the callee's **reads** clause is no larger than the caller's [61,34,50,57].)

Note that, unlike the procedures generated from methods, procedure $C.F.WellDefined$ does not have any precondition that mentions $CanAssumeFunctionDefs$. Therefore, it is not possible to use the function axioms when doing the proof of well-definedness for a function.

It seems that having the definition of the function trumps any other axiom about it. However, if the definition is recursive, it becomes difficult to prove that a change of the heap does not affect the value of the function. In particular, doing such a proof typically requires an induction principle, which is not automatically supported by SMT solvers. Also, there are times when one might want to hide the actual definition in parts of the program (*cf.* [57,32,45]). Then, it is useful to have the following frame axiom, which builds on the function's **reads** clauses and says what parts of memory the function depends on:

$$
\begin{array}{l}
\textbf{axiom}\ CanAssumeFunctionDefs\ \Rightarrow \\
\quad (\,\forall\, \mathcal{H}\colon HeapType,\ \mathcal{K}\colon HeapType,\ this\colon Ref,\ \mathsf{decl}^*[\![\, ins \,]\!]\ \bullet \\
\qquad GoodHeap(\mathcal{H}) \ \wedge\ GoodHeap(\mathcal{K}) \ \wedge \\
\qquad (\,\forall\, \langle\alpha\rangle\ o\colon Ref,\ f\colon Field\ \alpha\ \bullet\ o \neq null \ \wedge\ o \in \mathsf{tr}[\![\, rd \,]\!] \ \Rightarrow\ \mathcal{H}[o,f] = \mathcal{K}[o,f]\ ) \\
\qquad \Rightarrow\ C.F(\mathcal{H}, this, ins) = C.F(\mathcal{K}, this, ins)\ )
\end{array}
$$

The consistency of this axiom follows from the fact that the function's body adheres to the function's declared **reads** clauses, which in turn follows from the proof obligation $C.F.WellDefined$ above. The proof of class Node in Section 1.0, whose function Valid is recursive, needs this frame axiom.

5. Related Reading

Dynamic frames were introduced by Kassios [28] and were first implemented in an automatic program verifier by Smans *et al.* [59,58,57].

A prevalent architecture of such verifiers first translates the source language to a primitive intermediate verification language, and then generates theorem-prover input from the intermediate language. ESC/Modula-3 [16] and ESC/Java [22] used early forms of this architecture, which is now further developed in Boogie [4,36] and Why [19]. A pedagogical development of the architecture for a core object-oriented language with subclassing and invariants is given in previous Marktoberdorf lecture notes [43].

The style of dynamic-frames specifications bears some resemblance to the valid/-state specification idiom in ESC/Modula-3 [16,40], to data groups [33,41], and to separation logic [55] with predicates [51,50]. Alternatives are explored in JML [31], which uses universe types [46], and Spec# [7], which uses the Boogie methodology [5,38,8,44].

Separation logic [55] provides a reasoning logic that hides the explicit representation of dynamic frames. Recently, some promising checkers based on separation logic have sprung up [10,18,26]. These operate in a way that is more similar to Floyd/Hoare logic [24,25] for enumerated straight-line paths of a program than to verification-condition generation via weakest preconditions. Region logic [3] uses Floyd/Hoare style reasoning with *regions* that explicitly let the logic talk about dynamic frames. Region logic has also been encoded as a translation to Boogie [2].

Acknowledgments

I thank Jan Smans for various discussions on this topic, and for useful comments on a draft of these lecture notes. Michał Moskal and Wolfram Schulte helped write a previous version of the Queue example in Boogie, developing the use of the "bulk update" (in Dafny, the **foreach** statement) as a way to update all relevant ghost fields of the linked list. I also thank the curious students and other participants at the Summer School on Logic and Theorem-Proving in Programming Languages in Eugene, OR (July 2008) and the International Summer School Marktoberdorf, Germany (August 2008), as well as the organizers of those summer schools.

References

[0] Ádám Darvas and K. Rustan M. Leino. Practical reasoning about invocations and implementations of pure methods. In Matthew B. Dwyer and Antónia Lopes, editors, *Fundamental Approaches to Software Engineering, 10th International Conference, FASE 2007*, volume 4422 of *Lecture Notes in Computer Science*, pages 336–351. Springer, March–April 2007.

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Anindya Banerjee, Mike Barnett, and David A. Naumann. Boogie meets regions: A verification experience report. In Natarajan Shankar and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments, Second International Conference, VSTTE 2008*, volume 5295 of *Lecture Notes in Computer Science*, pages 177–191. Springer, October 2008.

[3] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In Jan Vitek, editor, *ECOOP 2008 — Object-Oriented Programming, 22nd European Conference*, volume 5142 of *Lecture Notes in Computer Science*, pages 387–411. Springer, July 2008.

[4] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, September 2006.

[5] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[6] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In Michael D. Ernst and Thomas P. Jensen, editors, *Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'05*, pages 82–87. ACM, September 2005.

[7] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005.

[8] Mike Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In Dexter Kozen and Carron Shankland, editors, *Seventh International Conference on Mathematics of Program Construction (MPC 2004)*, volume 3125 of *Lecture Notes in Computer Science*, pages 54–84. Springer, July 2004.

[9] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer, July 2007.

[10] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, September 2006.

[11] Jeremy Condit, Brian Hackett, Shuvendu Lahiri, and Shaz Qadeer. Unifying type checking and property checking for low-level code. In *Proceedings of POPL 2009*. ACM, January 2009. To appear.

[12] Patrick Cousot and Rhadia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM, January 1977.

[13] Ádám Darvas and Peter Müller. Reasoning about method calls in interface specifications. *Journal of Object Technology*, 5(5):59–85, June 2006.

[14] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, March–April 2008.

[15] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.

[16] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.

[17] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[18] Dino Distefano and Matthew J. Parkinson. jStar: Towards practical verification for Java. In Gail E. Harris, editor, *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008*, pages 213–226. ACM, October 2008.

[19] Jean-Christophe Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.

[20] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In *Sixth International Conference on Formal Engineering Methods (ICFEM)*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29. Springer, November 2004.

[21] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, July 2007.

[22] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37, number 5 in *SIGPLAN Notices*, pages 234–245. ACM, May 2002.

[23] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Conference Record of the 28th Annual ACM Symposium on Principles of Programming Languages*, pages 193–205. ACM, January 2001.

[24] R. W. Floyd. Assigning meaning to programs. In *Mathematical Aspects of Computer Science*, pages 19–32. XIX American Mathematical Society, 1967.

[25] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, October 1969.

[26] Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, August 2008.

[27] Rajeev Joshi. Extended static checking of programs with cyclic dependencies. In James Mason, editor, *1997 SRC Summer Intern Projects*, Technical Note 1997-028. Digital Equipment Corporation Systems Research Center, 1997.

[28] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer, August 2006.

[29] Viktor Kuncak and K. Rustan M. Leino. In-place refinement for effect checking. In *Second International Workshop on Automated Verification of Infinite-State Systems (AVIS'03)*, April 2003.

[30] Shuvendu K. Lahiri and Shaz Qadeer. Verifying properties of well-founded linked lists. In Greg Morrisett and Simon Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006*, pages 115–126. ACM, January 2006.

[31] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[32] K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technol-

ogy, 1995. Technical Report Caltech-CS-TR-95-03.

[33] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*, volume 33, number 10 in *SIGPLAN Notices*, pages 144–153. ACM, October 1998.

[34] K. Rustan M. Leino. A SAT characterization of boolean-program correctness. In Thomas Ball and Sriram K. Rajamani, editors, *Model Checking Software (SPIN 2003)*, volume 2648 of *Lecture Notes in Computer Science*, pages 104–120. Springer, May 2003.

[35] K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, March 2005.

[36] K. Rustan M. Leino. This is Boogie 2. Manuscript KRML 178, 2008. Available at `http://-research.microsoft.com/~leino/papers.html`.

[37] K. Rustan M. Leino, Michał Moscal, and Wolfram Schulte. Verification condition splitting. Submitted manuscript, September 2008.

[38] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In Martin Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, June 2004.

[39] K. Rustan M. Leino and Peter Müller. A verification methodology for model fields. In Peter Sestoft, editor, *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*, pages 115–130. Springer, March 2006.

[40] K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, September 2002.

[41] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37, number 5 in *SIGPLAN Notices*, pages 246–257. ACM, May 2002.

[42] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. In Bart Jacobs, Gary T. Leavens, Peter Müller, and Arnd Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*, Technical Report 251. Fernuniversität Hagen, May 1999.

[43] K. Rustan M. Leino and Wolfram Schulte. A verifying compiler for a multi-threaded object-oriented language. In *2006 Marktoberdorf Summer School on Programming Methodology*, NATO ASI Series. Springer, 2007.

[44] K. Rustan M. Leino and Angela Wallenburg. Class-local object invariants. In *First India Software Engineering Conference (ISEC 2008)*. ACM, February 2008.

[45] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002. PhD thesis, FernUniversität Hagen.

[46] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.

[47] Greg Nelson. Verifying reachability invariants of linked structures. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 38–47. ACM, January 1983.

[48] Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, 1989.

[49] Greg Nelson, editor. *Systems Programming with Modula-3*. Series in Innovative Technology. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[50] Matthew J. Parkinson. *Local Reasoning for Java*. PhD thesis, Computer Laboratory, University of Cambridge, 2005. UCAM-CL-TR-654.

[51] Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005*, pages 247–258. ACM, January 2005.

[52] D. L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972.

[53] Arnd Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitationsschrift, Technische Universität München, 1997.

[54] Arnd Poetzsch-Heffter and Peter Müller. Logical foundations for typed object-oriented languages. In David Gries and Willem-Paul de Roever, editors, *Programming Concepts and Methods, PROCOMET '98*, pages 404–423. Chapman & Hall, 1998.

[55] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74. IEEE Computer Society, 2002.

[56] Arsenii Rudich, Ádám Darvas, and Peter Müller. Checking well-formedness of pure-method specifications. In Jorge Cuellar, Tom Maibaum, and Kaisa Sere, editors, *FM 2008: Formal Methods, 15th International Symposium on Formal Methods*, volume 5014 of *Lecture Notes in Computer Science*, pages 68–83. Springer, May 2008.

[57] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. In Marieke Huisman, editor, *Formal Techniques for Java-like Programs (FTfJP 2008)*, pages 1–12, July 2008. Proceedings appear as Technical Report ICIS-R08013, Radboud University Nijmegen.

[58] Jan Smans, Bart Jacobs, and Frank Piessens. VeriCool: An automatic verifier for a concurrent object-oriented language. In Gilles Barthe and Frank S. de Boer, editors, *Formal Methods for Open Object-Based Distributed Systems, 10th IFIP WG 6.1 International Conference, FMOODS 2008*, volume 5051 of *Lecture Notes in Computer Science*, pages 220–239. Springer, June 2008.

[59] Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. An automatic verifier for Java-like programs based on dynamic frames. In José Luiz Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008*, volume 4961 of *Lecture Notes in Computer Science*, pages 261–275. Springer, March–April 2008.

[60] Spec# home page. `http://research.microsoft.com/specsharp/`, 2008.

[61] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.