Prototype Verification System

Lecture Notes

Daniele Parmeggiani

February 29, 2024

Contents

Chapter 1

Introduction

In this chapter we will introduce PVS (the Prototype Verification System),[1] and we will install a modern environment for using PVS. Later, we'll also see how to prove some simple properties of a hashmap using PVS.

1.1 Introduction to PVS

PVS is a powerful tool designed for formal specification and verification of software and hardware systems. It is a language coupled with a complete environment for developing formal mathematical specifications and then verifying that those meet certain properties.

PVS comes with an expressive specification language that allows you to describe your systems in a precise and formal manner. One of the core functionalities of PVS is its ability to prove theorems about your specifications. This involves using mathematical reasoning to demonstrate that certain properties hold true for your system under specified conditions.

PVS includes a set of built-in libraries that cover a wide range of mathematical theories and common programming constructs. These libraries provide a foundation for specifying and verifying a variety of systems. Notably among them, we can find NASALib,[2] a vast set of libraries developed by the Formal Methods teams at the NASA Langley Research Center, along with the Stanford Research Institute, the National Institute of Aerospace, and the larger PVS community.

PVS is often used to verify critical software systems where correctness is of utmost importance, such as avionics and aerospace software, medical devices, or other safety-critical applications.

[1] `https://pvs.csl.sri.com/`

[2] `https://github.com/nasa/pvslib`

It is also extensively employed in the verification of hardware systems, and can be used to formally specify and verify communication protocols, cryptographic and concurrent algorithms, and other complex systems.

1.2 Installing PVS

The recommended development environment for PVS is NASA's plugin for Visual Studio Code:[3]

1. download and install Visual Studio Code for your platform from `https://code.visualstudio.com/download`;
2. navigate to the extensions tab in VS Code (the blocks icon at the left edge of the screen) and search for "pvs";
3. the first item in the list should be called "PVS" and authored by Paolo Masci;
4. click the install button and follow the on-screen installation instructions.

Note that this procedure installs the PVS binaries and libraries in your system as well, at a path that you need to specify.

To confirm that everything was installed correctly, open a new VS Code workspace; i.e. create and select a folder from your file system. Then, navigate to the newly appeared PVS tab at the left edge of the screen, right-click on the folder's name and select "New PVS File." In the new file insert the following content:

```
hello: THEORY
  BEGIN

  world: CONJECTURE
    TRUE
END hello
```

Now, right-click the name of the new file on the left panel and select "Rerun All Proofs:" you should see a new tab called `hello.summary` displaying the above conjecture `world` with the status `proved`.

For further help installing this plugin, please refer to `https://github.com/nasa/vscode-pvs#installation-instructions`.

[3] `https://github.com/nasa/vscode-pvs`

1.3 Proving Properties of a Hashmap

Let us introduce the core concepts and features of the PVS proof assistant with a commonly known data structure. We will try to prove a hashmap has the following features:

1. it should be possible to insert a new (key, value) pair;
2. it should be possible, given a key, to retrieve its corresponding value; and
3. it should be possible to remove a (key, value) pair from the hashmap.

Let's name these operations *insert*, *lookup*, and *delete*. Since we'll be talking about keys, values and maps, we should let PVS know about them. Specifically, we introduce new *uninterpreted* types; i.e. types that PVS knows nothing about: whether there would be 0, 1, or infinite instances of these types, for example. PVS instead is thus only told that a key `K` is something different from a value `V`.

```
K: TYPE
V: TYPE
```

Now we define what a hashmap is, and we use this notation:

```
M: TYPE = [K -> V]
```

This tells PVS that a hashmap `M` is a function from keys to values. We will later define this function as a sequence of associations. You can also think of this `TYPE` as an array of (key, value) pairs.[4]

We could define `M` as a partial function (that is, a function defined over a subset of its domain), but efficient theorem proving strongly encourages the use of total functions. One way to denote the fact that some $k \in K$ is not to be found in the hashmap, is to identify some particular value that indicates this. Let's choose an arbitrary element of `V`:

```
null: V
```

So that, we will expect a hashmap to be empty iff $\forall k \in K, m(k) = \texttt{null}$. We can write this in PVS too:

[4]PVS also allows expressing exactly *an array of tuples*, but it's able to reason very efficiently with functions, which is why we choose this representation.

```
empty: M
empty_ax: AXIOM
    FORALL (k: K): empty(k) = null
```

We have declared a hashmap to be the *one* empty hashmap, and we have axiomatically defined `empty` as the hashmap in which all keys point to `null`. This may surprise a programmer who's used to think that there may be multiple instances of a hashmap class that all happen to be empty. From the point of view of PVS, though, all those instances are exactly the same. A program may distinguish them based on their memory address, but PVS has no notion of this, instead it only knows abstractly about the types we have defined so far. Think of it mathematically instead of programmatically: the number 0 is exactly the number 0 no matter how many times we may write it on a piece of paper.

Also, do not consider the axiom itself to be computationally expensive since it associates `null` to the entire key space. Quite the contrary: in the carrying out of a proof, PVS will know that whatever we may write next, the value of any key in `empty` will be `null`, without employing any memory to store the possible associations.

Now, let's "implement" the hashmap operations:

```
lookup: [M, K -> V]
lookup_ax: AXIOM FORALL (m: M) (k: K):
    lookup(m, k) = m(k)

insert: [M, K, V -> M]
insert_ax: AXIOM FORALL (m: M) (k: K) (v: V):
    insert(m, k, v) = m WITH [(k) := v]

delete: [M, K -> M]
delete_ax: AXIOM FORALL (m: M) (k: K):
    delete(m, k) = m WITH [(k) := null]
```

Notice some of the notations we have used so far: `m(k)` is exactly equivalent to the mathematical notation of function application, the `AXIOM` and `FORALL` keywords refer to their usual meanings, and the keyword `WITH` has been used to add an association to the function `m` above.

Also, notice the *functional* specification style: we pass the whole "state of the system" (i.e. the entire hashmap) to the functions `lookup`, `insert`, and `delete`. Consider `insert`: the input hashmap is somehow discarded in the operation and we instead "return" a novel hashmap that has the

added association. As previously said, do not consider that PVS will actually allocate and deallocate memory to represent these hashmaps.

If we were programming a hashmap and we felt content of what we wrote, we may turn to writing tests. But with PVS we instead want to write theorems about our hashmaps: they are general statements that we think should be true if our specification does what it ought to. This can provide much more information than a single test case, since with one theorem we would be running an entire *class* of test cases.

A suitable challenge for our specification may be: "if I add a key `k` with a value `v` to a hashmap and then I lookup `k`, I should get back `v`." We can write it as:

```
insert_then_lookup: CONJECTURE FORALL (m: M) (k: K) (v: V):
    lookup(insert(m, k, v), k) = v
```

In your VS Code editor, you should see some links above the definition of `insert_then_lookup`. We invoke the prover by clicking "prove":

```
Starting prover session for insert_then_lookup


insert_then_lookup :

|-------
{1}   FORALL (m: M) (k: K) (v: V): lookup(insert(m, k, v), k) =
v

>>
```

This is a *sequent*: in general there will be several numbered formulas above the turnstile symbol `|-------` (antecedent), and several below (consequent). The idea is that we have to establish that the conjunction (and) of formulas in the antecedent implies the disjunction (or) of the formulas in the consequent. Therefore, a sequent is true if any antecedent is the same as any consequent, if any antecedent is false, or if any consequent is true.

The prompt indicates that PVS is waiting for us to submit a prover command, in Lisp syntax. The commands provided by PVS can be categorized as basic commands and strategies, which in turn are compositions of basic commands. Let us disregard enumerating all of the various commands available and their semantics during this introduction (they will be explored later in §???) and let us instead introduce a few commands and consider how a proof with PVS can be carried out in general.

The highest-level strategy is called `grind`. It does skolemization, heuristic instantiation, propositional simplification, if-lifting, rewriting, and applies decision procedures for linear arithmetic and equality. It can take several optional arguments to specify the formulas that can be used for automatic rewriting (below, we'll be specifying that it can use the axioms in the theory `hashmap`).

Following the execution of a command, PVS may be required to split the proof into two or more parts. The PVS prover itself maintains a proof tree; the act of proving the specification can be thought of as traversing the tree, with the goal of showing that each leaf of the tree is true.

The `grind` strategy satisfies the proof for `insert_then_lookup`:

```
>> (grind :theories ("hashmap"))

Q.E.D.
```

Following the display of `Q.E.D.`, you should see a pop-up indicating that the proof has been saved. We will mostly run our proofs in the interactive environment, but after a proof is successfully completed, it can be re-run in batch mode, without user interactions. Interactive proofs can also be suspended with `(quit)` and later resumed.

In the example shown above the proof tree had just one trivial branch. Now let's turn to another less trivial conjecture:

```
insert_then_delete: CONJECTURE FORALL (m: M), (k: K), (v: V):
    delete(insert(m, k, v), k) = m
```

Intuitively, the above states that inserting a key `k` into a hashmap `m` and then deleting `k`, yields `m` unchanged. The first proof went very smoothly, let's see about this one. The same strategy used before produces the following result:

```
Starting prover session for insert_then_delete


insert_then_delete :

|-------
{1}   FORALL (m: M), (k: K), (v: V): delete(insert(m, k, v), k)
= m

>> (grind :theories ("hashmap"))
```

```
Trying repeated skolemization, instantiation, and if-lifting

insert_then_delete :

|-------
{1}   m!1 WITH [(k!1) := null] = m!1

>>
```

The identifiers with `!` in them are Skolem constants—arbitrary representatives for quantified variables. Notice that PVS has already simplified away one of the two `WITH` expressions: if it only substituted the axioms, it could have equivalently written `m!1 WITH [(k!1) := v!1] WITH [(k!1) := null] = m!1`.

The resulting sequent is requesting us to prove that two functions (i.e. hashmaps) are equal. We try to appeal to the principle of extensionality, which states that two functions are equal if their values are the same for every point of their domains:

```
>> (apply-extensionality)


Applying extensionality

insert_then_delete :

|-------
{1}   m!1 WITH [(k!1) := null](x!1) = m!1(x!1)
[2]   m!1 WITH [(k!1) := null] = m!1

>> (delete 2)


Deleting some formulas

insert_then_delete :

|-------
[1]   m!1 WITH [(k!1) := null](x!1) = m!1(x!1)
```

```
>>
```

At any point we can invoke the `delete` command to remove an element of a sequent; here the original formula was deleted to reduce clutter, since it is equivalent to the more interesting extensional form.[5]

This sequent is asking us to show that the value associated with an arbitrary key `x!1` is the same before and after `m` was updated for `k!1`. We can do a case analysis here to consider whether `x!1` = `k!1`.

```
>> (lift-if)


Lifting IF-conditions to the top level

insert_then_delete :

|-------
{1}   IF x!1 = k!1 THEN null = m!1(x!1) ELSE TRUE ENDIF

>> (ground)


Applying propositional simplification and decision procedures

insert_then_delete :

{-1}   x!1 = k!1
|-------
{1}   null = m!1(x!1)

>>
```

Let's look closely at this sequent: it is asking to demonstrate that if $x_1 = k_1$, then $m_1(x_1) =$ `null`, but modus ponens, we also are trying to show that $m_1(k_1) =$ `null`. That is, that the hashmap's value for k_1 was `null` before we inserted k_1. But it doesn't necessarily have to be so! If the value associated with k_1 was a non-null value, say v_2, then the `insert` operation *updates* the association and the later `delete` associates k_1 to `null`. Thus, our

[5]Recall that to prove a sequent we need to prove that the conjunction of the antecedents implies the disjunction of the consequents.

conjecture (that the hashmap is unchanged following an `insert` + `delete`) is true only under the assumption that the key we add to the hashmap wasn't already in the hashmap (i.e. $m(k) = \texttt{null}$).

If we change our conjecture to

```
insert_then_delete: CONJECTURE FORALL (m: M), (k: K), (v: V):
    lookup(m, k) = null => delete(insert(m, k, v), k) = m
```

then, we can prove it with

```
(grind :theories ("hashmap"))
(apply-extensionality)
(delete 2)
(grind :theories ("hashmap"))
```

1.3.1 Axiomatic Issues

Let's now introduce two seemingly innocuous axioms, extending our specification the same way we have done before:

```
is_in?: [M, K -> bool]
is_in_ax: AXIOM FORALL (m: M), (k: K):
    m(k) /= null

insert_is_in_ax: AXIOM FORALL (m: M), (k: K), (v: V):
    is_in?(insert(m, k, v), k)
```

The semantics of these axioms should be fairly self-explanatory: a key is in the hashmap if it is not null, and after inserting some key k in a hashmap `m`, `k` is in `m`.

But these axioms hide an inconsistency!

Exercise 1.3.1. Prove, without using PVS, that true = false, based on the axioms provided so far.

Hint: start from `is_in?(insert(empty, k, null), k)`.

In fact, it shouldn't be possible that after associating a key k with `null`, we find that the value associated with k is not `null`! We have clearly abused axiomatic definitions and PVS will happily run this impossible proof for us, given that the inconsistency is within the axioms themselves, that PVS assumes to hold. We'll see later how to avoid abusing axioms, but for now, let's try to carry out the exercise proof using PVS.

```
Starting prover session for what


what :

|-------
{1}   FORALL (k: K): is_in?(insert(empty, k, null), k)

>> (skolem!)


Skolemizing

what :

|-------
{1}   is_in?(insert(empty, k!1, null), k!1)

>> (use "insert_ax")


Using lemma insert_ax

what :

{-1}   insert(empty, k!1, null) = empty WITH [(k!1) := null]
|-------
[1]   is_in?(insert(empty, k!1, null), k!1)

>> (use "is_in_ax")


Using lemma is_in_ax

what :

{-1}   insert(empty, k!1, null)(k!1) /= null
[-2]   insert(empty, k!1, null) = empty WITH [(k!1) := null]
|-------
[1]   is_in?(insert(empty, k!1, null), k!1)
```

```
>> (use "empty_ax")


Using lemma empty_ax

what :

{-1}   empty(k!1) = null
[-2]   insert(empty, k!1, null)(k!1) /= null
[-3]   insert(empty, k!1, null) = empty WITH [(k!1) := null]
|-------
[1]   is_in?(insert(empty, k!1, null), k!1)

>> (ground)

Q.E.D.
```

The complete `.pvs` file for this example can be found in §A.1.

1.4 Exercises

Exercise 1.4.1. Encode in PVS grammar the following properties of natural numbers, without proving them. Note that in PVS to denote that a variable is a natural number, we write e.g. `FORALL (n: nat)`.

1. Let $n \in \mathbf{N}$, $n > 5 \Rightarrow n > 10$.
2. Let $x, y, z \in \mathbf{N}$. If $x = y + z$, then $x^2 = yx + zx$, $xy = y^2 + zy$, and $xz = z^2 + yz$.
3. Let $x, y, z \in \mathbf{N}$. If $x = y + z$ and the above formulae hold, then $x^2 = y^2 + z^2 + 2yz$, and $(y + z)^2 = y^2 + z^2 + 2yz$.

What happens when you try to prove these properties using `(grind)`?

Chapter 2

Fundamentals of PVS Syntax and its Prover

You can read more on the "PVS Language Reference" (Owre et al., 2020) and the "PVS Prover Guide" (Shankar et al., 2020).

A language cheat-sheet can be found in Appendix B.

Chapter 3

Concurrent Reference Counting

Let us now go through a more interesting specification: one that deals with concurrent memory management. In this chapter we'll practice ...

As a basis, we observe that computer memory is an array of bytes. On a 64-bit computer, you can address memory from byte number 0, to byte number $2^{64} - 1$. This is not a simplistic assumption that we're making here: it's the abstraction that all modern computers and operating systems use. Of course, memory is more complicated than this: there is virtual memory, kernel-space memory, file-system mapped memory, and process-specific memory, to name a few.

In here, we cannot go in so deep as to take all of these different kinds of memory into consideration. Instead, we'll be specifying the memory for one single process, as if all the rest is not there.

Another assumption we'll make is that the machine memory is unbounded. This sort of makes memory management redundant, but here we're interested in specifying the behavior of a management system irrespective of the amount of memory it can dispose of.

We can be free to add this constraint later, and we will, after we dealt with the other issues that will come up shortly.

To keep things simple, we'll specify a reference counting-based, object-oriented scheme. That is, we assume that our model process has knowledge of thing called objects, that may represent different classes of data. In managing memory, though, what the classes of data are isn't important. Each object has an amount of memory associated with it and however large or small it may be, we can assume the correct amount will be allocated (because we assumed memory is infinite), and deallocated once the object is freed.

3.1 Specification

The following rules are the behavior of the memory management system we would like to specify:

Rule 1. When an object is allocated, its reference count will be 1.

Rule 2. Any object can be freed if its reference count is 0.

Rule 3. When a new reference is made to an existing object, the object's reference count must be increased by 1.

Rule 4. When an existing reference to an object is removed, the object's reference count must be decreased by 1.

The rules, that will be verified, are based on these intuitive notions:

1. when an object is allocated, the returned pointer is the only reference to that object;

2. such is the case when no other object or variable is referencing it, thus it is safe to reclaim the memory;

3-4. the stored reference count must be kept consistent with its semantics.

The rules above ensure essentially that, in principle, any allocated object can be eventually freed, when its presence is no longer necessary.

Notice though, how Rule (2) doesn't specify that the object *is* freed, only that it *can* be freed. This will become important later, but first we need to talk about why these properties alone are not sufficient.

There are important features of a memory management system that we're not guaranteeing: eventual reclamation, and use-after-free avoidance. As already stated, Rule (2) only requires the possibility of freeing; we want to enforce that an object that can be freed, will be freed.

Use-after-free is a commonly known bug of programs that don't employ a memory management system, or that use it incorrectly. What happens is that a piece of memory is freed and then used again, as the name implies, either for a read or for a write. At that point, the program generally crashes with a segmentation fault. A more subtle, but possibly worse, manifestation of this bug is when the program requests to allocate memory and the allocation routine returns some address x, the program then frees x, requests more

memory to be allocated, and the allocation routine returns x again.[1] If we distinguish the two states of the object at x with A and B, we may be surprised when using the pointer x as if it was A, but instead it is B. This may not be a problem per se, but generally can be a very tricky situation in a concurrent system. In fact it could be that one thread t_1 frees A and allocates an unrelated object B, while another thread t_2 still had a pointer x that it considered to pertain to A. This of course is a bug in the overall program. In a reference count-based memory scheme, a possible cause of such a bug is that the code of t_2 didn't correctly account for a new reference to A, or that t_1 incorrectly decremented the reference count.

Let us add more rules to deal with these problems.

Rule 5. When the reference count of an object is 0, it remains 0 until the object is freed.

Rule 6. If an object is being freed, its reference count must be 0.

Rule 7. If an object has been freed, no thread tries to use it.

Rule 8. After the reference count of an object becomes 0,[2] the object is eventually freed.

3.2 Pseudocode

In this section we'll be constructing a program in C-like pseudocode to use as a reference for what we intend to specify later in PVS. Some base-level C knowledge is required to read this section.

First, we can easily declare our object, which we'll be using throughout our pseudocode.

```
typedef struct {
    int refcount;
    bool allocated;
    bool freed;
} object;
```

[1] This is not an unlikely scenario, many memory management systems allocate chunks of memory. When a call to the free routine is made, the system marks that piece of memory within a chunk as free, and when a successive call to the allocation routine for the same size is made, the previously freed piece is unmarked and returned.

This behavior is useful to improve performance by not continuously relying on calls to the kernel when allocations are frequent. Object-oriented and functional language runtimes often exhibit this usage pattern.

[2] Such is a definite, unique point in the program execution due to the combination of Rules (1), and (5).

It is a very simple abstraction for an object. Remember as we've said previously, we don't want to actually store any kind of data in this object. We'll be doing it later. So, for now, we only care about an object's reference count (the `refcount` field). We also add two flags which are only used to simplify our specification: in a real system the fact that a portion of memory is allocated or free is stored in kernel data structures (and possibly also in user-space when a memory management system is employed). We do not care about those details here. Instead, we attach this information to the objects themselves. It would be of course a significant waste in space, but we have an infinite amount of it here ;)

Next, we'll be doing something weird. Each thread has its own set of registers, which we can imagine as its current view of the shared memory. As you may know, in C there is no direct access to registers,[3] the mapping of variables to registers is left to the compiler. So, in order to make clear the usage of registers by a thread, we introduce an explicit `registers` array:

```
object *registers[10] = {NULL, NULL, NULL, NULL,
    NULL, NULL, NULL, NULL, NULL, NULL};
```

Another thing we generally don't get to see is the stack. Memory on the heap is explicitly allocated in a C program, but not the variables that are allocated onto the stack. Again, we want to make it clear that a thread is viewing a certain set of variables:

```
object *variables[10] = {NULL, NULL, NULL, NULL,
    NULL, NULL, NULL, NULL, NULL, NULL};
```

Both of these arrays should be thought of as thread-local storage, with each thread seeing a different array. This is implicit in the pseudo program, because we avoid explicitly starting threads.

With the above definition of an object, the `free` routine becomes straightforward, beacuse we just need to set the `freed` flag:

```
void free()
{
    CAS(registers[0]->freed, false, true);
}
```

There are a couple of things to note here. First, we're using a `CAS` function, which we don't declare nor define, that denotes the usage of a hardware compare-and-set instruction to perform this write to memory. This pseudo-function returns a boolean value, which we just don't care to look at here.

[3]Unless you embed assembly code.

We'll take care of the fact that the object wasn't already freed by other means.

Second, this routine takes no input parameters, while the usual C `free` function does require the pointer to the memory that needs to be freed. Here, we're making a choice to simplify further the PVS specification that will follow. Instead of specifying that there is a parameter, we assume that the parameter is in register number 0. Note that this is not a PVS-specific way of doing this, we're just making this deliberate choice on how we model a real program execution.

Next, the allocation routine:

```
void alloc()
{
    do {
        object *obj = choose();
    } while (!CAS(obj->allocated, false, true));
    obj->refcount = 1;
    registers[0] = obj;
}
```

Weirdness abounds. Let's break it down. First, the signature is again unexpected. Again, the reason is simplifying the PVS specification: instead of specifying a return mechanism we rely on the assumption that the returned value is found in register number 0, instead of actually modeling the return register.

Then, what's that `choose()` doing? As said, `malloc` and the like have special data structures that are used to determine which address in memory is the next free address that can be allocated, but we don't care to map it to our specification. So, instead, we "choose" an address. That is, we *non-deterministically* pick one at random until we find an object that wasn't already allocated. Non-determinism is certainly not a characteristic usually found in computer programs. You can think of it as a non-deterministic Turing machine that outputs a random address. With non-determinism, we can also "be lucky" and find a non-allocated object on our first try. Which is exactly what we'll do later in the PVS specification. Here, the check inside the while is kept in order to be clear about the state of `obj` after the loop.

Now we define the reference count increment and decrement routines, and we'll look at them together, given that they're pretty similar:

```
void inc_ref()
{
    object *obj = registers[0];
```

```
    do {
        int refcount = obj->refcount;  // registers[1]

        if (obj->refcount == 0) {
            registers[0] = NULL;
            return;
        }
    } while (!CAS(obj->refcount, refcount, refcount + 1));
}

void dec_ref()
{
    object *obj = registers[0];

    do {
        int refcount = obj->refcount;

        if (obj->refcount == 0) {
            registers[0] = NULL;
            return;
        }
    } while (!CAS(obj->refcount, refcount, refcount - 1));
    refcount--;

    if (refcount == 0) {
        free(obj);
    }
}
```

These two routines shouldn't be very surprising, given what we've already seen so far. In both, we read the current reference count of the input object and attempt to modify it accordingly. In case we fail, because another thread modified the count concurrently, we read it again and retry. This should be the expected way to correctly implement a concurrent counter.

The decrement routine also takes care of freeing the object in case its reference count reaches 0. This leads to a subtle problem. Consider this execution: two threads t_1 and t_2 are respectively trying to incref and decref a shared object x. Both read refcount$(x) = 1$. t_2 proceeds before t_1 and decrements the reference count. Additionally, t_2 executes the call to `free` before t_1 attempts to CAS the reference count. The object has thus been

freed before t_1 executes the CAS. When t_1 tries to execute the CAS, it breaks Rule (7).

We have thus determined that the decref routine cannot also take care of freeing the object that has reached a reference count of 0. In fact, if it was also done inside the same routine, the thread executing it would need to wait for an indefinite amount of time before executing free. Instead, we'll employ a scheme known as Quiescent State Based Reclamation (QSBR): we'll defer the call to free until we know no thread is trying to access objects that have refcount$(x) = 0$. That is, we need a Garbage Collector (GC).

Instead of calling `free` in `dec_ref`, we append the decref'ed object to a GC tracking list. When the GC will run a collection cycle, the tracked objects will be freed.

Let's add the GC structures at the top of our pseudocode:

```
extern int T;  // number of threads (T >= 1)

// ...

typedef struct GC {
    object *obj;
    struct GC *next;
} GC;

GC *gc_head = NULL;
bool STW_world_stopped = false;  // stop the world
bool STW_requested = false;
int STW_count_down = T;
```

As you can already see, we'll be implementing a very simple stop-the-world GC later, based on a single linked list of tracked objects to be freed. For now, the result is that inside the `dec_ref` function we replace the call to free with the following linked-list append:

```
GC *gc = new GC();

gc->obj = obj;
gc->next = NULL;

do {
    gc_tail = gc_head;
    while (gc_tail != NULL) {
        gc_tail = gc_tail->next;
```

```
    }
} while (!CAS(gc_tail->next, NULL, gc);
```

A very simple GC cycle can be implemented like this:

```
void collect()
{
    GC *gc = gc_head;

    while (gc != NULL) {
        free(gc->obj);
        gc = gc->next;
    }

    gc_head = NULL;
}
```

As you can see, it's a very trivial list traversal algorithm which calls `free`. The complexity will stem from the stop-the-world part of this GC. When a thread becomes the GC (i.e. initiates a GC cycle), it will need to make sure that no other thread is executing an incref or a decref. As said, it will do so by locking all other threads out of execution. The mechanism is not very complicated: first we signal all threads that a stop-the-world (STW) request has occurred, then we wait for all other threads to acknowledge the request, and finally we signal that we have successfully stopped the world. After the call to collect completes, we reset the STW state.

```
void gc_cycle()
{
    if (!CAS(STW_requested, false, true)) {
        return;
    }

    while (STW_count_down - 1 > 0) { }  // wait for all threads
                                        // (except this one)
    STW_world_stopped = true;

    collect();

    STW_count_down = T;
    STW_requested = false;
    STW_world_stopped = false;  // resume the world
```

```
                                // (must be last)
}
```

Exercise 3.2.1. Based on the above `gc_cycle` routine come up with a `STW_wait` routine, in pseudocode, that performs its counterpart: it is called by all the threads that did not attempt to request a STW operation, and possibly makes them wait for the STW operation to complete before returning. A solution can be found in §A.3.

Finally, we need to write a `main` function that is the entry point of all threads. Again, we're not going to explicitly start any thread, we simply assume that T threads start executing the `main` function. In particular, we want it to be an infinite loop over the possible operations that a thread can do: allocate, decref, or incref an object, or start a GC cycle. No program would exactly behave like this, but this is a close-enough model. A real program would have some possibly complicated logic on what actions it needs to take, but from the point of view of a memory management system, all that logic is transparent. Such system would only be able to observe the program continuously requesting one of those operations that we just mentioned. Therefore, a `while (true) { switch (operation) {…} }` loop is already a sufficiently detailed representation of a program that uses our concurrent memory management system. Thus, the beginning of our `main` function can look like this:

```
int main()
{
    while (true) {
        STW_wait();  // maybe wait if the world is stopped,
                     // or if a stop is requested

        int action = choose();  // registers[5]
        int variable = choose();  // registers[6]

        switch (action) {
```

We make again use of non-determinism: the action and the variable upon which the action is performed are chosen at random. For our purposes, this pseudocode maps to a situation in which PVS needs to make sure that our proofs hold for whichever value `action` and `variable` may take, which is exactly what we want. Next we go through the four possible actions: `alloc`, `dec_ref`, `inc_ref`, and `GC`. The order in which we do so, and the integer numbers we map the actions to, is completely arbitrary.

```
case 1:  // dec_ref
    if (variables[variable] == NULL) {
        break;
    }

    registers[0] = variables[variable];
    dec_ref();

    if (registers[0] == NULL) {
        goto error;
    }

    variables[variable] = NULL;
    registers[0] = NULL;
    break;
```

Notice how we first check that the chosen variable makes sense: that it actually holds a reference to an object. Given that we're using non-determinism, we need to make sure that we didn't non-deterministically choose something that would break our logic. In this case, we don't want to decref `NULL`. Then, in keeping with our calling convention, we populate register 0 to pass the argument to the `dec_ref` function. If it returned `NULL`, it means that the reference count of the object in our variables registry reached 0 before the `dec_ref` routine could decrement it. This is considered an impossibility: how could it reach 0 if we were holding a reference to it? To represent this impossibility we create a new error state, outside the main loop body:

```
int main()
{
    while (true) {
        // ...

        switch (action) {
            // ...
        }
    }

    error:
    while (true) { }
}
```

A program execution that reaches this state, never goes back to a correct state. This is a common pattern in formal verification. In the following section, we will somehow encode that a valid program execution is one that never enters the error state.

Now let's look at the `inc_ref` action, where we want to acquire a reference to an object that was already allocated, possibly by a different thread:

```
case 2:  // inc_ref
        if (variables[variable] != NULL) {
            break;
        }

        object *obj = choose();  // registers[7]
        if (!obj->allocated || obj->freed) {
            break;
        }

        variables[variable] = obj;
        registers[0] = variables[variable];
        inc_ref();

        if (registers[0] == NULL) {
            variables[variable] = NULL;
        }

        registers[0] = NULL;
        break;
```

Here, we want to make sure that we are not replacing an already populated variable. Next, we want to randomly pick[4] an object from memory. The condition that we need to satisfy to cope with the non-determinism is to check that the object was previously allocated, and has not been freed. Notice that a real program doesn't generally have the opportunity to check for these states. The implicit assumption here is that the program logic we're glossing over, also takes care of not using a faulty pointer.

If the call to `inc_ref` fails, we don't need to go into the error state: the fact that an incref doesn't work is perfectly normal, we don't incur in a use-after-free because of our QSBR scheme. It simply means that we didn't manage to acquire a new reference to the object, and thus need to refrain from doing so (and also clean up the reference in the variables array).

[4]A real program wouldn't pick an object at random, instead picking one according to its logic, which here we model with non-determinism.

The GC action is very trivial: simply call the `gc_cycle` function to maybe run a GC cycle. The allocation part is very similar to what we've already seen, you can read it in the full pseudocode in §A.3.

3.3 PVS Specification

The full specification can be found in §A.5. We won't go through it entirely, skipping the intuitive parts.

First of all, we declare that our specification is parameterized on the number of threads:

```
refcount[T: {x: nat | x >= 1}]: THEORY
BEGIN
```

This means that we may prove the correctness of our specification, irrespective of the number of threads T. That number T needs to be fixed and known for our proofs to work, but the proof holds independently of any value of $T \geq 1$. If you recall that the way the STW mechanism works is by decrementing a counter initialized with the number of threads, then you'll immediately see why we need to have T as a fixed number.

One important aspect of the specification is pertaining states: we wish to encode in PVS that the program execution is in a specific state. That is, one global state for all threads, comprising a complete state of the global memory, of the registers and variables local to each thread, the global STW flags and counter, the global GC linked list, and the program counter of each thread. We'll use the latter to describe what instruction a thread should be executing next, which is a natural way of encoding the fact that each real thread does have its own program counter. All of this information combined indicates one definite state of the execution of our program, which we can use to check some invariants to prove correctness.

```
    GC: TYPE = list[Address]

    State: TYPE = [#
        memory: [Address -> Object],

        registers: [Thread -> [int -> int]],
        variables: [Thread -> [int -> int]],

        gc_head: GC,
```

```
        STW_world_stopped: bool,
        STW_requested: bool,
        STW_count_down: int,

        pc: [Thread -> nat]
    #]
```

As you can see, the `GC` type was very simple to define: PVS already has built-in[5] datatypes for linked lists, among many other utilities.

We'll take a look at the specification of the `free` function first, since it is rather simple.

```
free(t, s1, s2): bool = s1`pc(t) = 36 and (
    if s1`memory(s1`registers(t)(0))`freed = false then
        s2 = s1 with [
            (memory)(s1`registers(t)(0)) :=
                    s1`memory(s1`registers(t)(0)) with [
                        (freed) := true
            ],
            (pc)(t) := 107
        ]
    else
        s2 = s1 with [
            (pc)(t) := 107
        ]
    endif
)
```

First of all, you should notice that the way we specify the behavior of the single line of pseudocode in the free routine is by specifying a function that returns true if the state `s2` is the one state that could be reached from `s1` if thread `t` happened to be the one in execution and its program counter happened to be at line 36. Notice how we're not explicitly modifying the state, instead we're asserting whether one state is the successor of another. We do so by asserting whether the successor state is equal to the predecessor state, with the modification we expect from the execution of the line of code. In this sense, a line of code (or rather, an assembly instruction, but we're not going to that much detail), is simply a specification of how the state of the machine should be modified. A computer programmer should be

[5]Or rather, *prelude* datatypes: you can find them in your PVS installation folder, in a file called `prelude.pvs`.

familiar with this concept, even though they never used it. Unlike a normal computer program, here we're interested in specifying all the possible such modifications so as to prove that all the traces (i.e. executions) that can be produced by the correct sequencing of those modifications, are correct.

Also notice how we are modeling the fact that only one thread modifies the state at a given time. The line numbers refer to the pseudocode that we prepared in the above section.

Talking about the behavior of `free` specifically, we can also observe the modeling of the behavior of a CAS instruction. In the same state change, we both check that the memory contains the expected value and we either modify it, or don't. In this case, we don't take a different path (i.e. the next program counter is the same in both branches of the if) when the CAS fails, but you can see how we could have easily specified that.

Also, you may have noticed that we set an absolute line number as the next program counter. This should surprise programmers who are used to be able to call functions from anywhere in their programs. We have purposefully written our pseudocode to contain at most one call site for each function in it. This property enables us to employ this trick of explicitly setting the next PC. If instead we didn't have that property, we would in principle be forced to also model a call stack, like the one that normal computer programs have. That would render our PVS specification much more difficult to prove.

Now let's consider the `alloc` routine, where we make use of non-determinism to get the address of newly allocated memory:

```
alloc(t, s1, s2): bool = s1`pc(t) = 28 and (
    exists (a: Address): (
        (
            % preconditions
            s1`memory(a) = (# refcount := 0,
                              allocated := false,
                              freed := false #)
        )
    ) => (
        % mutation (allocation)
        s2 = s1 with [
            (memory)(a) := (# refcount := 1,
                              allocated := true,
                              freed := false #),
            (registers)(t)(0) := a,
            (pc)(t) := 154
        ]
```

```
    )
)
```

As you can see the magic of non-determinism here is simply expressed by an existence: we say that if there exist an address, such that it satisfies some conditions, then we can proceed with the state transition. The precondition here is that there exist an unused address. This is always going to be the case since we have infinite memory, so there are infinitely many objects like this one,[6] therefore the preconditions are always satisfied. In order to simplify our writing we both set the `allocated` field of the object to `true` and set register 0 to the pointer we just found. Since the register is local, we can collapse those two states into one, without compromising the correctness in spite of concurrency. Also note, that by simply saying $\exists a$, we have successfully avoided having to choose an address efficiently like `malloc` does. This doesn't mean that an actual implementation of this prototype cannot have an efficient mechanism for allocation here: we just don't care to specify it.

Let's now jump forward to the last line of the decref routine (`dec_ref_4`), focusing on the branch taken when the CAS succeeds:

```
s2 = s1 with [
    (memory)(s1`registers(t)(0)) :=
        s1`memory(s1`registers(t)(0)) with [
            (refcount) := s1`registers(t)(1) - 1
        ],
    (pc)(t) := 84
]
```

Here we're decrementing the counter. If we had specified that our `Object` type has a `refcount` of type `nat` instead of `int`, then we would have had to deal with TCCs at this line (and other lines like this one). This would have been impossible to prove: the fact that `s1`registers(t)(0)` stores a number $\geq 1$ is derived from a trace of transitions, and cannot be evinced by simply looking at this line on its own. Instead, we can say that the `refcount` has type `int` and leave this pseudo-type checking to our invariants.

In `dec_ref_7` we have taken a deliberate shortcut:

```
dec_ref_7(t, s1, s2): bool = s1`pc(t) = 87 and (
    s2 = s1 with [
        (gc_head) := append(s1`gc_head,
            cons(s1`registers(t)(0), null)
        ),
```

[6] You'll also want to check the definition of an initial state, explained later.

```
        (pc)(t) := 107
    ]
)
```

This one state change pertains lines 87 to 97, and treats them as one atomic operation. We shouldn't do this in principle. Nevertheless, we can assume that a safe way to achieve this change that employs several transitions exist, both in PVS and in the real world (the pseudocode provided should be a convincing example), and thus we need not fully specify the atomicity of a linked-list append, as such would be trivial, but cumbersome to prove.

Just below, we have taken a similar shortcut for the GC collect routine:

```
s2 = s1 with [
    (gc_head) := cdr(s1`gc_head),
    (memory)(car(s1`gc_head)) :=
            s1`memory(car(s1`gc_head)) with [
            (freed) := true
        ],
    (pc)(t) := 107
]
```

As you can see, we modify both the `gc_head` shared variable, and free the object, skipping the actual call to `free`, which is unrealistic. This is again fine, around this `collect` call we have already worked on holding the related STW locks, thus all modifications we make to memory can be thought of as atomic, from the perspective of the remaining threads.

Exercise 3.3.1. Consider `gc_cycle_4`. From the proof point of view, what, if anything, would change if it was split into multiple functions, each mutating the global state one variable at a time? Explain your answer.

Exercise 3.3.2. Consider `main_2` where non-determinism is used to choose what action a thread takes. It doesn't guarantee that globally some thread eventually chooses to become the GC, which is required in order to prove Rule (8). Modify the relevant parts of the pseudocode and of the PVS specification to ensure that rule.

Going further down, let's take a look at the `step` function:

```
step(t, s1, s2): bool =
    alloc(t, s1, s2)
    or free(t, s1, s2)
    or inc_ref_1(t, s1, s2)
```

```
    % ...
    or main_23(t, s1, s2)
    or error(t, s1, s2)
```

With this function we enumerate all the possible transitions that can take place in between two states. Notice again how only one thread can make any progress in the execution of one step. The non-exclusive OR works here because each of those functions has a condition like $\mathrm{pc}(t) = x$, where x is a line number so that it is disjunct w.r.t. all other transition functions. If we used XOR, the proofs would take longer to complete because PVS would also have to check that exclusivity.

Later, we define what an initial state is allowed to look like:

```
init_empty_memory(s): bool =
    forall (a: Address):
        s`memory(a) = (# refcount := 0, allocated := false, freed := false #)

init_empty_registers(s): bool =
    forall (t: Thread, r: Register):
        s`registers(t)(r) = NULL

init_empty_variables(s): bool =
    forall (t: Thread, v: Variable):
        s`variables(t)(v) = NULL

init_starting_pc(s): bool =
    forall (t: Thread):
        s`pc(t) = 146

init(s): bool =
    true
    and init_empty_memory(s)
    and init_empty_registers(s)
    and init_empty_variables(s)
    and init_starting_pc(s)
    and s`gc_head = null
    and s`STW_world_stopped = false
    and s`STW_requested = false
    and s`STW_count_down = T
```

This should be rather intuitive. We can point out that `init_empty_memory` ensures that the precondition of `alloc` can always be met. Or equivalently, that can only be unmet after an infinite number of allocations.

Then, we define some invariants that we want all the states to hold:

```
INV(s): bool =
    true
    and rule_1(s)
    and rule_6(s)
    and rule_7(s)
    and not_in_error_state(s)
```

This represents the set of "good" states that we may encounter in an execution of our program. Then, we want to prove two core theorems: the initial states are "good," and from a "good" state we will necessarily go into another "good" state:

```
init_implies_invariants: THEOREM
    forall (s: State):
        init(s) => INV(s)

step_implies_invariants: THEOREM
    forall (t: Thread, s1: State, s2: State):
        step(t, s1, s2) and INV(s1) => INV(s2)
```

Proving these two theorems provides the full proof that our theory about concurrent reference counting is correct.

3.4 Big Data

Let's add data to our objects! How much data? All the data!

Let's take a step back. We're trying to reason on the correctness of a memory management system. In particular, of an abstract system that can use an infinite amount of space! It turns out that with infinite space we could store a single bit per object and still have the possibility of encoding all the natural numbers. In fact, we don't even need that bit! We can encode whatever information we need simply in the address of an object. Since there can be an infinite amount of objects, there must be an infinite number of addresses. Assuming that the memory allocator gives out an object at a perfectly random address, after an infinite number of attempts, we will manage to get the object at the exact address that we need.

Except of course if that object had been previously allocated. Alright, we do have this limitation, fair enough. But it can be easily solved! Let's add a pointer field to every object, such that if the pointer of object α points to the address of α, then the semantics are those previously described: the

object α encodes the number α. Otherwise, if it points to another object β, then it means that α represents the number β.[7]

This poses a problem for our GC. If there exist two distinct objects α and β, such that α points to β, then β cannot be freed before α is freed.

- modify the previous pseudocode and pvs spec;
- prove that every object is eventually freed

[7]This is a ludicrous introduction. In fact, to store a pointer to an unbounded number of memory locations, it's necessary to store a pointer of infinite size. Nevertheless, what follows does hold for garbage collection in general.

Appendix A

Solutions

A.1 The simple hashmap example

```
hashmap: THEORY
BEGIN
    K: TYPE
    V: TYPE
    M: TYPE = [K -> V]

    null: V
    empty: M
    empty_ax: AXIOM
        FORALL (k: K): empty(k) = null

    lookup: [M, K -> V]
    lookup_ax: AXIOM FORALL (m: M) (k: K):
        lookup(m, k) = m(k)

    insert: [M, K, V -> M]
    insert_ax: AXIOM FORALL (m: M) (k: K) (v: V):
        insert(m, k, v) = m WITH [(k) := v]

    delete: [M, K -> M]
    delete_ax: AXIOM FORALL (m: M) (k: K):
        delete(m, k) = m WITH [(k) := null]

    is_in?: [M, K -> bool]
    is_in_ax: AXIOM FORALL (m: M), (k: K):
```

```
        m(k) /= null

    insert_is_in_ax: AXIOM FORALL (m: M), (k: K), (v:
       V):
        is_in?(insert(m, k, v), k)

    insert_then_lookup: CONJECTURE FORALL (m: M) (k: K
       ) (v: V):
        lookup(insert(m, k, v), k) = v

    insert_then_delete: CONJECTURE FORALL (m: M), (k:
       K), (v: V):
        lookup(m, k) = null => delete(insert(m, k, v),
            k) = m

    what: CONJECTURE FORALL (k: K):
        is_in?(insert(empty, k, null), k)
END hashmap
```

A.2 Exercise 1.4.1

```
chapter_1: THEORY
BEGIN

    ex_1: CONJECTURE FORALL (n: nat):
        n > 5 => n > 10

    ex_2_1?(x,y,z: nat): bool = x^2 = y*x + z*x

    ex_2_2?(x,y,z: nat): bool = x*y = y^2 + z*y

    ex_2_3?(x,y,z: nat): bool = x*z = z^2 + y*z

    ex_2: CONJECTURE FORALL (x,y,z: nat):
        x = y + z => (ex_2_1?(x,y,z) and ex_2_2?(x,y,z
           ) and ex_2_3?(x,y,z))

    ex_3: CONJECTURE FORALL (x,y,z: nat):
        (x = y + z and ex_2_1?(x,y,z) and ex_2_2?(x,y,
           z) and ex_2_3?(x,y,z)) =>
            (x^2 = y^2 + z^2 + 2*y*z and (y + z)^2 = y
```

```
                ^2 + z^2 + 2*y*z)

END chapter_1
```

A.3 Chapter 3 Pseudocode

```
extern int T;  // number of threads (T >= 1)

typedef struct {
    int refcount;
    bool allocated;
    bool freed;
} object;

// each thread sees its own registers and variables
// think of them as thread-local storage
object *registers[10] = {NULL, NULL, NULL, NULL, NULL,
    NULL, NULL, NULL, NULL, NULL};
object *variables[10] = {NULL, NULL, NULL, NULL, NULL,
    NULL, NULL, NULL, NULL, NULL};

typedef struct GC {
    object *obj;
    struct GC *next;
} GC;

GC *gc_head = NULL;
bool STW_world_stopped = false;  // stop the world
bool STW_requested = false;
int STW_count_down = T;


void alloc()
{
    do {
        object *obj = choose();
    } while (!CAS(obj->allocated, false, true));
    obj->refcount = 1;
    registers[0] = obj;
}
```

```
void free()
{
    CAS(registers[0]->freed, false, true);
}

void inc_ref()
{
    object *obj = registers[0];

    do {
        int refcount = obj->refcount;  // registers[1]

        if (obj->refcount == 0) {
            registers[0] = NULL;
            return;
        }
    } while (!CAS(obj->refcount, refcount, refcount +
       1));
}

void dec_ref_naif()
{
    object *obj = registers[0];

    do {
        int refcount = obj->refcount;

        if (obj->refcount == 0) {
            registers[0] = NULL;
            return;
        }
    } while (!CAS(obj->refcount, refcount, refcount -
       1));
    refcount--;

    if (refcount == 0) {
        free(obj);
    }
}

void dec_ref()
```

```
{
    object *obj = registers[0];

    do {
        int refcount = obj->refcount;  // registers[1]

        if (obj->refcount == 0) {
            registers[0] = NULL;
            return;
        }
    } while (!CAS(obj->refcount, refcount, refcount -
       1));
    refcount--;

    if (refcount == 0) {
        GC *gc = new GC();  // simplified as one
           atomic operation in `dec_ref_7`

        gc->obj = obj;
        gc->next = NULL;

        do {
            gc_tail = gc_head;
            while (gc_tail != NULL) {
                gc_tail = gc_tail->next;
            }
        } while (!CAS(gc_tail->next, NULL, gc);
    }
}

void collect()
{
    GC *gc = gc_head;

    while (gc != NULL) {
        free(gc->obj);
        gc = gc->next;
    }

    gc_head = NULL;
}
```

```
    void gc_cycle()
    {
115     if (!CAS(STW_requested, false, true)) {
            return;
        }

        while (STW_count_down - 1 > 0) { }  // wait for
           all threads (except this one)
120     STW_world_stopped = true;

        collect();

        STW_count_down = T;
125     STW_requested = false;
        STW_world_stopped = false;  // resume the world (
           must be last)
    }

    void STW_wait()
130 {
        while (STW_world_stopped) { }

        if (STW_requested) {
            do {
135             int count_down = STW_count_down;
            } while (!CAS(STW_count_down, count_down,
               count_down - 1));

            while (STW_requested || STW_world_stopped) { }
        }
140 }


    int main()
    {
145     while (true) {
            STW_wait();  // maybe wait if the world is
               stopped, or if a stop is requested

            int action = choose();  // registers[5]
```

```
        int variable = choose();  // registers[6]

        switch (action) {
        case 0:  // alloc
            if (variables[variable] != NULL) {
                break;
            }

            alloc();

            object *obj = registers[0];
            if (obj == NULL) {
                goto error;
            }
            variables[variable] = obj;
            registers[0] = NULL;

            break;
        case 1:  // dec_ref
            if (variables[variable] == NULL) {
                break;
            }

            registers[0] = variables[variable];
            dec_ref();

            if (registers[0] == NULL) {
                goto error;
            }

            variables[variable] = NULL;
            registers[0] = NULL;
            break;
        case 2:  // inc_ref
            if (variables[variable] != NULL) {
                break;
            }

            object *obj = choose();  // registers[7]
            if (!obj->allocated || obj->freed) {
                break;
```

```
190             }

                variables[variable] = obj;
                registers[0] = variables[variable];
                inc_ref();
195
                if (registers[0] == NULL) {
                    variables[variable] = NULL;
                }

200             registers[0] = NULL;
                break;
            case 3:  // GC
                gc_cycle();
                break;
205         default:
                break;
            }
        }

210     error:
        while (true) { }
    }
```

A.4 Exercise 3.3.1

A.5 Chapter 3 PVS Specification

```
refcount[T: {x: nat | x >= 1}]: THEORY
BEGIN

    Object: TYPE = [#
        refcount: int,
        allocated: bool,
        freed: bool
    #]

    Address: TYPE = int
    NULL: Address = 0
```

```
Register: TYPE = upto[10]
Variable: TYPE = upto[10]
Thread: TYPE = upto[T]
t, t1, t2: VAR Thread

% GC: DATATYPE
% BEGIN
%     gc_NULL: gc_NULL?
%     gc(obj: Address, next: GC): gc?
% END GC

GC: TYPE = list[Address]

State: TYPE = [#
    memory: [Address -> Object],

    registers: [Thread -> [int -> int]],
    variables: [Thread -> [int -> int]],

    gc_head: GC,

    STW_world_stopped: bool,
    STW_requested: bool,
    STW_count_down: int,

    pc: [Thread -> nat]
#]

s, s1, s2: VAR State


% operations
alloc(t, s1, s2): bool = s1`pc(t) = 28 and (
    exists (a: Address): (
        (
            % preconditions
            s1`memory(a) = (# refcount := 0,
               allocated := false, freed := false
               #)
        )
    ) => (
```

```
        % mutation (allocation)
        s2 = s1 with [
            (memory)(a) := (# refcount := 1,
               allocated := true, freed := false
               #),
            (registers)(t)(0) := a,
            (pc)(t) := 154
        ]
    )
)

free(t, s1, s2): bool = s1`pc(t) = 36 and (
    if s1`memory(s1`registers(t)(0))`freed = false
        then
        s2 = s1 with [
            (memory)(s1`registers(t)(0)) := s1`
               memory(s1`registers(t)(0)) with [
                 (freed) := true
            ],
            (pc)(t) := 107
        ]
    else
        s2 = s1 with [
            (pc)(t) := 107
        ]
    endif
)

inc_ref_1(t, s1, s2): bool = s1`pc(t) = 44 and (
    s2 = s1 with [
        (registers)(t)(1) := s1`memory(s1`
           registers(t)(0))`refcount,
        (pc)(t) := 46
    ]
)

inc_ref_2(t, s1, s2): bool = s1`pc(t) = 46 and (
    if s1`registers(t)(1) = 0 then
        s2 = s1 with [
            (pc)(t) := 47
        ]
```

```
    else
        s2 = s1 with [
            (pc)(t) := 50
        ]
    endif
)

inc_ref_3(t, s1, s2): bool = s1`pc(t) = 47 and (
    s2 = s1 with [
        (registers)(t)(0) := NULL,
        (pc)(t) := 196
    ]
)

inc_ref_4(t, s1, s2): bool = s1`pc(t) = 50 and (
    if s1`memory(s1`registers(t)(0))`refcount /=
       s1`registers(t)(1) then
        s2 = s1 with [
            (pc)(t) := 44
        ]
    else
        s2 = s1 with [
            (memory)(s1`registers(t)(0)) := s1`
               memory(s1`registers(t)(0)) with [
                (refcount) := s1`registers(t)(1) +
                    1
            ],
            (pc)(t) := 196
        ]
    endif
)

dec_ref_1(t, s1, s2): bool = s1`pc(t) = 77 and (
    s2 = s1 with [
        (registers)(t)(1) := s1`memory(s1`
           registers(t)(0))`refcount,
        (pc)(t) := 79
    ]
)

dec_ref_2(t, s1, s2): bool = s1`pc(t) = 79 and (
```

```
    if s1`registers(t)(1) = 0 then
        s2 = s1 with [
            (pc)(t) := 80
        ]
    else
        s2 = s1 with [
            (pc)(t) := 83
        ]
    endif
)

dec_ref_3(t, s1, s2): bool = s1`pc(t) = 80 and (
    s2 = s1 with [
        (registers)(t)(0) := NULL,
        (pc)(t) := 175
    ]
)

dec_ref_4(t, s1, s2): bool = s1`pc(t) = 83 and (
    if s1`memory(s1`registers(t)(0))`refcount /=
       s1`registers(t)(1) then
        s2 = s1 with [
            (pc)(t) := 77
        ]
    else
        s2 = s1 with [
            (memory)(s1`registers(t)(0)) := s1`
               memory(s1`registers(t)(0)) with [
                (refcount) := s1`registers(t)(1) -
                    1
            ],
            (pc)(t) := 84
        ]
    endif
)

dec_ref_5(t, s1, s2): bool = s1`pc(t) = 84 and (
    s2 = s1 with [
        (registers)(t)(1) := s1`registers(t)(1) -
           1,
        (pc)(t) := 86
```

```
    ]
)

dec_ref_6(t, s1, s2): bool = s1`pc(t) = 86 and (
    if s1`registers(t)(1) = 0 then
        s2 = s1 with [
            (pc)(t) := 87
        ]
    else
        s2 = s1 with [
            (pc)(t) := 175
        ]
    endif
)

dec_ref_7(t, s1, s2): bool = s1`pc(t) = 87 and (
   % perform lines 87 to 97 as one atomic
   operation
    s2 = s1 with [
        (gc_head) := append(s1`gc_head, cons(s1`
           registers(t)(0), null)),
        (pc)(t) := 107
    ]
)

collect(t, s1, s2): bool = s1`pc(t) = 107 and (
    if s1`gc_head = null then
        s2 = s1 with [
            (pc)(t) := 124
        ]
    else
        s2 = s1 with [
            (gc_head) := cdr(s1`gc_head),
            (memory)(car(s1`gc_head)) := s1`memory
               (car(s1`gc_head)) with [
                (freed) := true
            ],
            (pc)(t) := 107
        ]
    endif
)
```

```
gc_cycle_1(t, s1, s2): bool = s1`pc(t) = 115 and (
    if s1`STW_requested = false then
        s2 = s1 with [
            (STW_requested) := true,
            (pc)(t) := 119
        ]
    else
        s2 = s1 with [
            (pc)(t) := 204
        ]
    endif
)

gc_cycle_2(t, s1, s2): bool = s1`pc(t) = 119 and (
    if s1`STW_count_down - 1 > 0 then
        s2 = s1 with [
            (pc)(t) := 119
        ]
    else
        s2 = s1 with [
            (pc)(t) := 120
        ]
    endif
)

gc_cycle_3(t, s1, s2): bool = s1`pc(t) = 120 and (
    s2 = s1 with [
        (STW_world_stopped) := true,
        (pc)(t) := 107
    ]
)

gc_cycle_4(t, s1, s2): bool = s1`pc(t) = 124 and (
    s2 = s1 with [
        (STW_world_stopped) := false,
        (STW_requested) := false,
        (STW_count_down) := T,
        (pc)(t) := 204
    ]
)
```

```
STW_wait_1(t, s1, s2): bool = s1`pc(t) = 131 and (
    if s1`STW_world_stopped then
        s2 = s1 with [
            (pc)(t) := 131
        ]
    else
        s2 = s1 with [
            (pc)(t) := 133
        ]
    endif
)

STW_wait_2(t, s1, s2): bool = s1`pc(t) = 133 and (
    if s1`STW_requested then
        s2 = s1 with [
            (pc)(t) := 135
        ]
    else
        s2 = s1 with [
            (pc)(t) := 148
        ]
    endif
)

STW_wait_3(t, s1, s2): bool = s1`pc(t) = 135 and (
    s2 = s1 with [
        (registers)(t)(1) := s1`STW_count_down,
        (pc)(t) := 136
    ]
)

STW_wait_4(t, s1, s2): bool = s1`pc(t) = 136 and (
    if s1`registers(t)(1) = s1`STW_count_down then
        s2 = s1 with [
            (STW_count_down) := s1`registers(t)(1)
                - 1,
            (pc)(t) := 138
        ]
    else
        s2 = s1 with [
```

```
            (pc)(t) := 135
        ]
    endif
)

STW_wait_5(t, s1, s2): bool = s1`pc(t) = 138 and (
    if s1`STW_requested or s1`STW_world_stopped
       then
        s2 = s1 with [
            (pc)(t) := 138
        ]
    else
        s2 = s1 with [
            (pc)(t) := 148
        ]
    endif
)

main_1(t, s1, s2): bool = s1`pc(t) = 146 and (
    s2 = s1 with [
        (pc)(t) := 131
    ]
)

main_2(t, s1, s2): bool = s1`pc(t) = 148 and (
    exists (action: upto[3]):
        s2 = s1 with [
            (registers)(t)(5) := action,
            (pc)(t) := 149
        ]
)

main_3(t, s1, s2): bool = s1`pc(t) = 149 and (
    exists (variable: upto[3]):
        s2 = s1 with [
            (registers)(t)(6) := variable,
            (pc)(t) := 151
        ]
)

main_4(t, s1, s2): bool = s1`pc(t) = 151 and (
```

```
    cond
        s1`registers(t)(5) = 0 -> s2 = s1 with [
            (pc)(t) := 153
        ],
        s1`registers(t)(5) = 1 -> s2 = s1 with [
            (pc)(t) := 168
        ],
        s1`registers(t)(5) = 2 -> s2 = s1 with [
            (pc)(t) := 183
        ],
        s1`registers(t)(5) = 3 -> s2 = s1 with [
            (pc)(t) := 203
        ],
        else -> s2 = s1 with [
            (pc)(t) := 146
        ]
    endcond
)

main_5(t, s1, s2): bool = s1`pc(t) = 153 and (
    if s1`variables(t)(s1`registers(t)(6)) /= NULL
        then
        s2 = s1 with [
            (pc)(t) := 146
        ]
    else
        s2 = s1 with [
            (pc)(t) := 157
        ]
    endif
)

main_6(t, s1, s2): bool = s1`pc(t) = 157 and (
    s2 = s1 with [
        (pc)(t) := 28
    ]
)

main_7(t, s1, s2): bool = s1`pc(t) = 168 and (
    if s1`registers(t)(0) = NULL then
        s2 = s1 with [
```

```
            (pc)(t) := 211
        ]
    else
        s2 = s1 with [
            (pc)(t) := 163
        ]
    endif
)

main_8(t, s1, s2): bool = s1`pc(t) = 163 and (
    s2 = s1 with [
        (variables)(t)(s1`registers(t)(6)) := s1`
           registers(t)(0),
        (registers)(t)(0) := NULL,
        (pc)(t) := 146
    ]
)

main_9(t, s1, s2): bool = s1`pc(t) = 168 and (
    if s1`variables(t)(s1`registers(t)(6)) = NULL
       then
        s2 = s1 with [
            (pc)(t) := 146
        ]
    else
        s2 = s1 with [
            (pc)(t) := 172
        ]
    endif
)

main_10(t, s1, s2): bool = s1`pc(t) = 172 and (
    s2 = s1 with [
        (registers)(t)(0) := s1`variables(t)(s1`
           registers(t)(6)),
        (pc)(t) := 173
    ]
)

main_11(t, s1, s2): bool = s1`pc(t) = 173 and (
    s2 = s1 with [
```

```
        (pc)(t) := 74
    ]
)

main_12(t, s1, s2): bool = s1`pc(t) = 175 and (
    if s1`registers(t)(0) = NULL then
        s2 = s1 with [
            (pc)(t) := 211
        ]
    else
        s2 = s1 with [
            (pc)(t) := 179
        ]
    endif
)

main_13(t, s1, s2): bool = s1`pc(t) = 179 and (
    s2 = s1 with [
        (variables)(t)(s1`registers(t)(6)) := NULL
           ,
        (registers)(t)(0) := NULL,
        (pc)(t) := 146
    ]
)

main_14(t, s1, s2): bool = s1`pc(t) = 183 and (
    if s1`variables(t)(s1`registers(t)(6)) /= NULL
        then
        s2 = s1 with [
            (pc)(t) := 146
        ]
    else
        s2 = s1 with [
            (pc)(t) := 187
        ]
    endif
)

main_15(t, s1, s2): bool = s1`pc(t) = 187 and (
    exists (a: Address):
        s2 = s1 with [
```

```
            (registers)(t)(7) := a,
            (pc)(t) := 188
        ]
)

main_16(t, s1, s2): bool = s1`pc(t) = 188 and (
    if (not s1`memory(s1`registers(t)(7))`
       allocated) or (s1`memory(s1`registers(t)(7)
       )`freed) then
        s2 = s1 with [
            (pc)(t) := 146
        ]
    else
        s2 = s1 with [
            (pc)(t) := 192
        ]
    endif
)

main_17(t, s1, s2): bool = s1`pc(t) = 192 and (
    s2 = s1 with [
        (variables)(t)(s1`registers(t)(6)) := s1`
           registers(t)(7),
        (registers)(t)(0) := s1`registers(t)(7),
        (pc)(t) := 194
    ]
)

main_18(t, s1, s2): bool = s1`pc(t) = 194 and (
    s2 = s1 with [
        (pc)(t) := 44
    ]
)

main_19(t, s1, s2): bool = s1`pc(t) = 196 and (
    if s1`registers(t)(0) = NULL then
        s2 = s1 with [
            (pc)(t) := 197
        ]
    else
        s2 = s1 with [
```

```
            (pc)(t) := 200
        ]
    endif
)

main_20(t, s1, s2): bool = s1`pc(t) = 197 and (
    s2 = s1 with [
        (variables)(t)(s1`registers(t)(6)) := NULL
           ,
        (pc)(t) := 200
    ]
)

main_21(t, s1, s2): bool = s1`pc(t) = 200 and (
    s2 = s1 with [
        (registers)(t)(0) := NULL,
        (pc)(t) := 146
    ]
)

main_22(t, s1, s2): bool = s1`pc(t) = 203 and (
    s2 = s1 with [
        (pc)(t) := 115
    ]
)

main_23(t, s1, s2): bool = s1`pc(t) = 204 and (
    s2 = s1 with [
        (pc)(t) := 146
    ]
)

error(t, s1, s2): bool = s1`pc(t) = 211 and (
    s2 = s1 with [
        (pc)(t) := 211
    ]
)


% transitions
step(t, s1, s2): bool =
```

```
alloc(t, s1, s2)
or free(t, s1, s2)
or inc_ref_1(t, s1, s2)
or inc_ref_2(t, s1, s2)
or inc_ref_3(t, s1, s2)
or inc_ref_4(t, s1, s2)
or dec_ref_1(t, s1, s2)
or dec_ref_2(t, s1, s2)
or dec_ref_3(t, s1, s2)
or dec_ref_4(t, s1, s2)
or dec_ref_5(t, s1, s2)
or dec_ref_6(t, s1, s2)
or dec_ref_7(t, s1, s2)
or collect(t, s1, s2)
or gc_cycle_1(t, s1, s2)
or gc_cycle_2(t, s1, s2)
or gc_cycle_3(t, s1, s2)
or gc_cycle_4(t, s1, s2)
or STW_wait_1(t, s1, s2)
or STW_wait_2(t, s1, s2)
or STW_wait_3(t, s1, s2)
or STW_wait_4(t, s1, s2)
or STW_wait_5(t, s1, s2)
or main_1(t, s1, s2)
or main_2(t, s1, s2)
or main_3(t, s1, s2)
or main_4(t, s1, s2)
or main_5(t, s1, s2)
or main_6(t, s1, s2)
or main_7(t, s1, s2)
or main_8(t, s1, s2)
or main_9(t, s1, s2)
or main_10(t, s1, s2)
or main_11(t, s1, s2)
or main_12(t, s1, s2)
or main_13(t, s1, s2)
or main_14(t, s1, s2)
or main_15(t, s1, s2)
or main_16(t, s1, s2)
or main_17(t, s1, s2)
or main_18(t, s1, s2)
```

```
    or main_19(t, s1, s2)
    or main_20(t, s1, s2)
    or main_21(t, s1, s2)
    or main_22(t, s1, s2)
    or main_23(t, s1, s2)
    or error(t, s1, s2)


% psuedo-type-checking invariants
refcounts_non_neg(s): bool =
    forall (a: Address):
        s`memory(a)`refcount >= 0


% state invariants
init_empty_memory(s): bool =
    forall (a: Address):
        s`memory(a) = (# refcount := 0, allocated
           := false, freed := false #)

init_empty_registers(s): bool =
    forall (t: Thread, r: Register):
        s`registers(t)(r) = NULL

init_empty_variables(s): bool =
    forall (t: Thread, v: Variable):
        s`variables(t)(v) = NULL

init_starting_pc(s): bool =
    forall (t: Thread):
        s`pc(t) = 146

init(s): bool =
    true
    and init_empty_memory(s)
    and init_empty_registers(s)
    and init_empty_variables(s)
    and init_starting_pc(s)
    and s`gc_head = null
    and s`STW_world_stopped = false
    and s`STW_requested = false
```

```
    and s`STW_count_down = T

rule_1(s): bool =
    forall (t: Thread):
        pc(s)(t) = 154 => memory(s)(registers(s)(t
           )(0)) = (# refcount := 1, allocated :=
           true, freed := false #)

% rule_3(s1, s2): bool =
%     forall (t: Thread):
%         pc(s1)(t) = 44 =>

% rule_5(s): bool =

rule_6(s): bool =
    forall (t: Thread):
        pc(s)(t) = 36 => (
            s`memory(s`registers(t)(0))`refcount =
                0
            and s`memory(s`registers(t)(0))`freed
               = false
            and s`memory(s`registers(t)(0))`
               allocated = true
        )

rule_7(s): bool =
    forall (t: Thread, v: Variable):
        variables(s)(t)(v) /= NULL => memory(s)(
           variables(s)(t)(v))`freed = false

not_in_error_state(s): bool =
    forall (t: Thread):
        pc(s)(t) /= 211


INV(s): bool =
    true
    % and refcounts_non_neg(s)
    and rule_1(s)
    and rule_6(s)
    and rule_7(s)
```

```
        and not_in_error_state(s)


    % intermediate proofs
    dec_ref_4_implies_dec_ref_2: LEMMA
        forall (t: Thread): forall(s1: State, s2:
           State):
            (
                INV(s1) and s1`pc(t) = 79 and s1`
                   registers(t)(1) > 0 and step(t, s1,
                    s2)
            ) => (
                INV(s2) and s2`pc(t) = 83 and s2`
                   registers(t)(1) > 0
            )


    % core proofs
    % @QED init_implies_state_invariant proved by dp
       on Thu, 06 Jun 2024 15:45:12 GMT
    init_implies_invariants: THEOREM
        forall (s: State):
            init(s) => INV(s)

    step_implies_invariants: THEOREM
        forall (t: Thread, s1: State, s2: State):
            step(t, s1, s2) and INV(s1) => INV(s2)

END refcount
```

Appendix B

PVS Cheat Sheet

In the next page.

PVS Cheat Sheet

Theories

```
function_properties[D, R: TYPE]: THEORY
BEGIN
    % this is a comment

    f, g: VAR [D -> R]
    x, x1, x2: VAR D
    y: VAR R

    injective?(f): bool = (FORALL x1, x2: (f(x1) = f(x2)) => (x1 = x2))
    surjective?(f): bool = (FORALL y: (EXISTS x: f(x) = y))
END function_properties

finite[T: TYPE]: THEORY
BEGIN
    IMPORTING function_properties

    is_finite: bool = (EXISTS (n: nat), (f: [upto[n] -> T]): surjective?(f))
END finite
```

Constants

`some: int`	An undefined integer constant
`some: int = 10`	The integer number 10
`abs(x: int): nat = if x >= 0 then x else -x endif`	Function definition

Expressions

`=`	Equality
`/=`	Inequality
`true, false, and, or, not, =>, <=>, FORALL, EXISTS`	Logical
`if x = 0 then 1 elsif x = 1 then 1 else x / 2 endif`	If expression

`CASES x OF`
` lst(val, nxt): lst(1, x)`
` ELSE lst(1, null)`
`ENDCASES`
	Pattern matching

`(lambda x: x + 1)`	Function expression
`f with [(0) := 1, (1) := 0]`	Function update
`{x: int | x < 10}`	Set expression
`(# amount := 10, curr := EUR #)`	Record construction
`(# amount := 10, curr := EUR #)`` `amount`	Record field access
`(# amount := 10, curr := EUR #) with [amount := 0]`	Record update
`(1, true, (lambda x: x + 1))`	Tuple construction
`proj_3((1, true, (lambda x: x + 1)))`	Tuple projection
`(1, true, (lambda x: x + 1)) with [2 := false]`	Tuple update
`a + b:nat`	Type coercion

Variables

`x, y, z: VAR int`	Three named variables of `int` type
`f: VAR [int -> int]`	A variable function
Usage	
`abs(x): nat = if x >= 0 then x else -x endif`	Avoided type declaration
`bound(x, y, z, f): bool = f(x, y) > z`	Not necessarily the same `x`

Types

`foo: TYPE`	Uninterpreted type
`spam: TYPE+`	Non-empty type
`positive_int: TYPE = {x: integer | x > 0}`	Subtype
`bin_int_f: TYPE = [int, int -> int]`	Function type
`bin_int_f: TYPE = ARRAY[int, int -> int]`	Equivalent to above
`tup: TYPE = [int, bool, [int -> int]]`	Tuple type
`currency: TYPE = {USD, EUR, JPY}`	Enumeration type
`value: TYPE = [# amount: nat, curr: currency #]`	Record type (struct)

Recursive and higher-order types

A higher-order type can be thought of like a generic type in Java or C++.

```
linked_list[T: TYPE]: DATATYPE
BEGIN
    null: null?
    lst (value: T, next: linked_list): lst?
END linked_list

a: linked_list[nat] = lst(1, lst(2, lst(3, null)))
empty: linked_list[bool] = null
```

The recursion is the `lst ...: lst?` part of the datatype definition.

Recursive functions

```
factorial(n: nat): RECURSIVE nat =
    if n = 0 then 1 else n * factorial(n-1) endif
MEASURE (LAMBDA n: n)  % the measure must always decrease
```

Prover commands

Control	
`(quit)`	Close prover session
`(postpone)`	Defer evaluation of this proof branch
`(undo)`	Revert the changes of the previous command
`(help)`	General help
`(help` *command*`)`	Help on a specific command
Strategies	
`(grind)`	And pray for a `Q.E.D.`
`(tcc)`	Apply common rules for type-checking conditions
`(assert)`	Simplify using decision procedures
Structural Rules	
`(copy` *fnum*`)`	Copy the formula *fnum*
`(delete` *fnum*`)`	Delete the formula *fnum*
Propositional Rules	
`(bddsimp)`	Propositional simplification using BDDs
`(case "`*condition*`")`	Split into cases; e.g. `(case "x > 0")`
`(flatten)`	Simplify disjunctions
`(lift-if)`	Lift embedded *if* expressions
`(prop)`	Propositional simplifications
Quantifier Rules	
`(skolem!)`	Skolemize with generated names
Using definitions and lemmas	
`(expand "`*name*`")`	Apply the definition of *name*
`(use "`*name*`")`	Apply lemma *name*