Linear Maps

Shuvendu K. Lahiri
Microsoft Research
shuvendu.lahiri@microsoft.com

Shaz Qadeer
Microsoft Research
qadeer@microsoft.com

David Walker
Princeton University
dpw@cs.princeton.edu

Abstract

Verification of large programs is impossible without proof techniques that allow local reasoning and information hiding. In this paper, we resurrect, extend and modernize an old approach to this problem first considered in the context of the programming language Euclid, developed in the 70s. The central idea is that rather than modeling the heap as a single total function from addresses (integers) to integers, we model the heap as a collection of partial functions with disjoint domains. We call each such partial function a *linear map*. Programmers may *select* objects from linear maps, *update* linear maps or *transfer* addresses and their contents from one linear map to another. Programmers may also declare new linear map variables and pass linear maps as arguments to procedures. The program logic prevents any of these operations from duplicating locations and thereby breaking the key heap representation invariant: the domains of all linear maps remain disjoint. Linear maps facilitate modular reasoning because programs that use them are also able to use simple, classical frame rules to preserve information about heap state across procedure calls. We illustrate our approach through examples, prove that our verification rules are sound, and show that operations on linear maps may be erased and replaced by equivalent operations on a single, global heap.

1. Introduction

Verification of large programs is impossible without proof techniques that allow local reasoning and information hiding. In this paper, we resurrect, extend and modernize an old approach to this problem first considered in the context of the programming language Euclid [17, 20], developed in the 70s. This approach centers around the introduction of a new data type, which we call a *linear map*. Intuitively, linear maps are simply little heaplets: Programmers may *store* objects in linear maps, *look up* objects in linear maps and, most interestingly, *transfer* addresses from one linear map to another. Programmers may also declare new linear map variables, pass them as arguments to functions, and receive them as result. We call these maps *linear* because of their connection to linear type systems [11, 28, 29]: like values with linear type, linear maps are never duplicated nor aliased.

Programs that use linear maps tend to be written in a functional store-passing (*i.e.,* linear-map-passing) style as this style facilitates local reasoning and information hiding. However, despite this functional facade, linear maps programs are actually ordinary imperative programs that load from, and store to, a single, global heap. In order to connect a linear maps program to its corresponding conventional imperative program, we define an *erasure transformation* that erases all linear maps variables, erases all transfer operations and replaces linear map lookups and updates with lookups and updates on the global heap. In the forward direction, the erasure transformation shows that using linear maps incurs no overhead; they are really just a new kind of ghost variable, used, as ghost variables often are, to facilitate modular verification. In the reverse direction, the erasure transformation may be seen as a tactic for proving the correctness of ordinary imperative programs: given an ordinary program, the reverse transformation explains the legal ways to transform it into an easy-to-verify linear maps program without changing its operational behavior.

There are a number of reasons we believe researchers should adopt linear maps as a modular verification technology. First and foremost, the idea is surprisingly simple to understand, to implement and, we hope, to build upon. We believe this is a key contribution. Second, linear maps require no new language of assertions. Generated verification conditions are encoded in first-order logic and may be solved by off-the-shelf SMT solvers such as Z3 [6]. Third, using linear maps enables effective use of the classical frame and hypothetical frame rules, completely unchanged, despite the presence of an imperative heap. Fourth, linear maps technology requires no changes to the overall judgmental apparatus involved in standard, first-order verification condition generation: it does not use non-standard modifies clauses and it does not depend upon sophisticated auxiliary notions such as the footprint or frame of a formula. Consequently, it should be relatively easy to extend any one of a number of standard, existing verification condition generation tools with these new data types.

Our work on linear maps has been directly inspired by several other recent approaches to modular reasoning, including research on capability-based type systems [30, 25, 8, 5], Separation Logic [13, 23, 22], Dynamic Frames [15, 18], Implicit Dynamic Frames [24, 19], and Region Logic [1]. From a technical standpoint, linear maps are clearly different from these other approaches: the commands, their operational semantics, and program logic rules are not the same. However, taking a broader view, linear maps and these other systems all attempt to deal with the issue of "separation" of heap cells in one way or another. In comparison to these other approaches, we believe that the advantage of linear maps are their semantic simplicity: some of these other approaches, when compared with conventional classical Hoare logic, have new kinds of modifies clauses, new kinds of footprint analyses, or new sorts of assertions (such as separating conjunction). Linear maps do not make such changes to conventional Hoare Logic – they are merely a new data type with new operations. Having said that, the additional complexity of other systems generally provides an advantage in brevity and one can see that tradeoff playing out in their implementations: the higher-level concepts they introduce are often implemented by compilation into a series of imperative commands or additional framing axioms or assertions which may then be discharged by classical theorem provers.

2. Key Concepts

Two structural verification rules are required to verify just about any imperative program. The first is the rule of consequence, which states that if a Floyd-Hoare triple $\{P\}C\{Q\}$ is valid and $P' \Rightarrow P$ and $Q \Rightarrow Q'$ then the triple $\{P'\}C\{Q'\}$ is also valid. The second

```
procedure incr(p: int) returns ()
  modifies heap
{
  heap[p] := heap[p] + 1;
}

heap[py] := 42;
call incr(px);
```

Figure 1. Frame rule with ordinary maps

is the (classical) frame rule, which states that if $\{P\}C\{Q\}$ is valid and the set of variables modified by C is disjoint from the set of free variables of R then $\{R \wedge P\}C\{R \wedge Q\}$ is also valid. In other words, the validity of framing formula R may be preserved across any statement that does not modify the frame's free variables.

With that background, consider the procedure `incr`:

```
procedure incr() returns ()
  requires true, ensures true, modifies x
{  x := x + 1; }
```

This procedure has no input arguments and no output arguments. Its specification consists of the precondition `true`, the postcondition `true`, and the guarantee that it does not modify any variable except `x`. Henceforth, our examples will use the convention that a missing `requires` clause indicates the precondition `true`, a missing `ensures` clause indicates the postcondition `true`, and a missing `modifies` clause indicates that the procedure does not modify any variables in the caller's scope.

Consider a call to `incr` in a calling scope that contains another variable `y`. The Floyd-Hoare triple {`y = 42`} `call incr()` {`y = 42`} is easily proved through a combination of the conventional frame and consequence rules.

```
{true} call incr() {true}
---------------------------------------- (Frame)
{y=42 ∧ true} call incr() {y=42 ∧ true}
---------------------------------------- (Consequence)
{y=42} call incr() {y=42}
```

The main reason for the simplicity of the proof is that the set of variables modified by the code fragment `call incr()` is disjoint from the free variables in the assertion `y = 42`.

This simple proof strategy does not quite work when the heap is used to allocate data. The standard method of modeling a heap [4] uses a single map variable mapping memory addresses to their contents. Since a procedure that updates the heap at any address must contain the map variable in its modifies set, the conventional frame rule cannot be used for preserving heap-related assertions across a call to such a procedure. To illustrate the problem, we model the heap as a variable `heap` mapping `int` to `int` and allocate the variables `x` and `y` on the heap with distinct addresses `px` and `py` (Figure 1). Further, we change the procedure `incr` to take the memory address whose contents are to be incremented. Let `C` denote the code fragment in Figure 1 after the definition of `incr`. Then, the triple {`px != py`} `C` {`heap[py] = 42`} cannot be proved using the conventional frame rule.

2.1 Linear Maps

We address this weakness of the conventional frame rule, not by changing it, but by refining our modeling of the heap. Instead of modeling the heap with a single monolithic map, we model it as a collection of partial maps with disjoint domains. We call each such map a linear map, which is essentially a pair comprising a total map representing the contents and a set representing the domain of the linear map. We refer to the underlying total map and domain of a linear map ℓ as $\mathrm{map}(\ell)$ and $\mathrm{dom}(\ell)$ respectively. We augment our

```
procedure incr(p: int, t:lin)
  requires p ∈ dom(t)
  ensures p ∈ dom(t)
{
  t[p] := t[p] + 1;
}

l[py] := 42;
var lx:lin in
  lx := l@{px};
  call incr(px,lx);
  l := lx@{px};
```

Figure 2. Frame rule with linear maps

programming language with operations over linear maps that are guaranteed to preserve the invariant that the domains of all linear maps are pairwise disjoint and their union is the universal set. We refer to this invariant as the disjoint domains invariant.

We now rewrite the program from Figure 1 using linear maps as shown in Figure 2. As we explain in Section 2.2, the program in Figure 1 can be obtained from the program in Figure 2 using the erasure operation; consequently, any properties about the runtime behavior of the latter program are valid for the former as well. The new definition of procedure `incr` takes a pointer `p` and a linear map `t` as arguments.[1] The implementation of `incr` demonstrates that linear maps can be read and written just like ordinary maps. Unlike ordinary maps, a read or a write of a linear map `t` at the address `p` comes with the precondition that `p` is in the domain `t`. The read and write of `t[p]` performed during the increment operation are safe because of the precondition of `incr`.

Let `D` denote the code fragment in Figure 2 after the definition of `incr`. The contents of the heap at the beginning of `D` is modeled by the linear map `l` whose domain includes both addresses `px` and `py`. In order to call `incr` with the pointer `px`, we also need to pass in a linear map whose domain contains the address `px`. We create such a linear map by declaring a new variable `lx`, whose domain is empty initially. We then perform the transfer statement `lx := l@{px}` to move the contents of address `px` from `l` to `lx`; this operation has the precondition that `px` is in the domain of `l`. The linear map `lx` is now passed, along with the pointer `px` to `incr`, thus satisfying the precondition of `incr`.

It is important to note that the procedure call has a side effect on the variable `lx` passed for the linear argument `t`. At entry, the contents of `lx` are transferred into `t`; at exit, the contents of `t` are transferred back into `lx`. This operational behavior is essential to maintain the disjoint domains invariant. After the call, we transfer the pointer `px` from `lx` to `l`. Thus, there is an implicit modifies clause on linear map arguments for a procedure that we do not explicitly show.

Unlike the previous version of the `incr` procedure in Figure 1, the new version of `incr` has the empty modifies specification. Consequently, the triple {`px != py`} `C` {`l[py] = 42`} can be easily verified using the conventional frame rule.

2.2 Erasure

An important aspect of our system is the erasure operation which allows us to connect the operational semantics of the program written using linear maps with the corresponding program written using a global total map. As an example, the erasure of the code in Figure 2 is the code in Figure 1 with `heap` being the unified total map. Section 3 develops a program logic for verifying properties

[1] `tot` and `lin` denote the types of ordinary and linear maps from `int` to `int` respectively.

```
procedure incr(p: int, tm: tot, t: lin)
  requires p ∈ dom(t) ∧ tm = map(t)
  ensures p ∈ dom(t) ∧ t[p] = tm[p]+1
{
  t[p] := t[p] + 1;
}

l[py] := 42;
l[px] := 24;
var lx: lin in
  lx := l@{px};
  var lxm:tot in
    lxm := map(lx);
    call incr(px,lxm,lx);
  l := lx@{px};
```

Figure 3. Ghost variables

of programs that use linear maps. The erasure operation essentially allows us to carry over the runtime properties established by the verification of a program using linear maps over to the erased program using a global map.

The erasure operation is defined both on the state and the program text. The erasure of a state combines all linear map variables in the state into a single unified total map; this transformation is possible because of the disjoint domains invariant. The restrictions on the operations permitted on linear maps have been designed precisely to ensure that the erasure of the state is well-defined. The erasure of the program text removes all occurrences of linear map variables and any transfer operations among them; further, a read or write of a linear map variable is transformed into the corresponding operation on the unified total map. The erasure operation ensures that a program (with linear maps) takes a state s to s' iff the erased program takes the erasure of state s to the erasure of state s'.

2.3 Two-state postconditions

We now augment our increment example from Figure 2 to illustrate another useful feature of our system. The code fragment `E` in Figure 3 assigns the value 24 to `l[px]` before calling `incr`; we would like to show that `l[px] = 25` at the end of `E`. To enable this verification, we must enrich the postcondition of `incr` to relate the value of `t[p]` upon exit with the value of `t[p]` upon entry. It is difficult to express such a postcondition because any reference to the linear argument `t` by default refers to its value in the exit state. To circumvent this problem, we pass an ordinary map `tm` to `incr` as an additional argument and add the precondition `tm = map(t)` indicating the relationship between `tm` and `t`. The presence of the parameter `tm` allows us to enrich the postcondition of `incr` to indicate that the value of `t[p]` at exit is one more than its value at entry. At the call site, we pass a total map whose value is `map(lx)` for the argument `tm`. This operation does not violate our disjoint domains invariant because while `lx` is a linear map, `lxm` is an ordinary total map. The erasure of ghost variables such as `tm` and `lxm` is standard; the erasure operation removes all references to them from the program text. A complete Floyd-Hoare proof for the program fragment `E` is shown in Figure 4. In this proof, we use two macros for compact representation of the assertions: `(l[py] ↦ _)` expands to `(py ∈ dom(l))` and `(l[py] ↦ c)` expands to `(py ∈ dom(l) ∧ l[py] = c)`.

2.4 Aliasing

While linear maps can express separation among heap locations naturally, they allow simple specifications even in the presence of aliasing. Figure 5 shows a procedure `swap` that swaps the contents of two heap cells whose addresses are provided as parameters `a` and `b`. Since `a` and `b` may be aliased with each other (or not), we pass

```
{ px != py ∧ l[py] ↦ _ ∧ l[px] ↦ _ }

l[py] := 42;

{ px != py ∧ l[py] ↦ 42 ∧ l[px] ↦ _ }

l[px] := 24;

{ px != py ∧ l[py] ↦ 42 ∧ l[px] ↦ 24 }

var lx:lin in

  { px != py ∧ l[py] ↦ 42 ∧ l[px] ↦ 24 }

  lx := l@{px};

  { px != py ∧ l[py] ↦ 42 ∧ lx[px] ↦ 24 }

  var lxm:tot in

    { px != py ∧ l[py] ↦ 42 ∧ lx[px] ↦ 24 }

    lxm := map(lx);

    { px != py ∧ l[py] ↦ 42 ∧ lx[px] ↦ 24
      ∧ lxm = map(lx) }

    call incr(px,lxm,lx);

    { px != py ∧ l[py] ↦ 42 ∧ lx[px] ↦ 25 }

  l := lx@{px};

{ px != py ∧ l[py] ↦ 42 ∧ l[px] ↦ 25 }
```

Figure 4. Proof of statement `E`

```
procedure swap(a: int, b: int, tm: tot, t: lin)
  requires {a,b} ⊆ dom(t) ∧ tm = map(t)
  ensures {a,b} ⊆ dom(t) ∧ t[a] = tm[b] ∧ t[b] = tm[a]
{
  var tmp: int in
    tmp := t[a];
    t[a] := t[b];
    t[b] := tmp;
}
```

Figure 5. Swap

the storage for `a` and `b` not as two different linear maps but as a single linear map `t`. Note that this example also uses a ghost total map `tm` to enable writing a precise postcondition.

2.5 Information hiding

Large programs are structured as a collection of modules, each of which offers public procedures as services to clients. An important goal of modular verification is to enable separate verification of the module and its clients. The correctness of a client should depend only on the preconditions and postconditions of the public procedures of the module and not on the private details of the module implementation. This requirement precludes any precondition from referring to the private state of the module; consequently, it becomes difficult to verify the module implementation which often depends on critical invariants at entry into a public procedure. As an example, consider the module shown in Figure 6 that implements a rational number and provides two procedures, `reset` and `floor`. The private representation of this module uses two integer

```
mod [
  num : int := 0;
  den : int := 1;

  invariant den != 0;

  procedure reset(a: int, b: int) returns ()
    requires b > 0;
  {
    num := a;
    den := b;
  }

  procedure floor() returns (res: int)
  {
    res := num/den;
  }
]
```

Figure 6. A simple module

variables `num` and `den` for storing the numerator and denominator respectively. The integer division operation in `floor` fails unless the value of `den` upon entry is different from zero.

This conflict between modular verification and information hiding is well-understood; so is the concept of module invariant as a mechanism for resolving this conflict [12]. A module invariant is an invariant on the private variables of a module that may be assumed upon entry but must be verified upon exit. The module invariant for the rational number module states that `den != 0`; it is preserved by each procedure and allows us to prove the safety of the division operation in `floor`.

There are two important reasons for the soundness of the verification method based on module invariants. First, the module invariant is verified upon exit from the module. Second, the module invariant refers only to the private variables of the module which cannot be accessed by code outside the module. As long as the program uses only scalar variables, verification using module invariants is simple. However, if the representation of a module uses the global heap, which is potentially shared by many different modules, verification becomes difficult since the module invariant cannot refer to the global heap variable. Linear maps come to the rescue just as they did with the framing problem in the presence of the heap; we illustrate their use in the context of information hiding using a memory manager example [22].

Figure 7 shows the memory manager module. This module provides two procedures `alloc` and `free`, used for allocating and freeing a single memory address, respectively. `alloc` returns the freshly allocated address in the return variable `res`. `free` frees the memory address pointed to by the input variable `arg`. It is worth noting that the modified set of both `alloc` and `free` is empty; therefore, they are allowed to modify only the private state of the memory manager module and their respective linear map arguments.

The private representation of the memory manager uses a linear map variable `local` and an integer variable `f`. The value of `f` is a pointer to the beginning of an acyclic list obtained by starting from `f` and applying `local` repeatedly until the special address `nil` is reached. The variable `f` is initialized to a special pointer `nil` and `local` is initialized to the liner map with an empty domain denoted by `[]<>`. The module invariant of the memory manager uses a special set constructor `Btwn` that takes three arguments, a linear map `l`, a pointer `a`, and a pointer `b`. The set `Btwn(l,a,b)` is empty if `a` cannot reach `b` by following `l`. Otherwise, there is a unique acyclic path following `l` from `a` to `b` and `Btwn(l,a,b)` is the set of all pointers on this path including `a` and `b`. The module invariant

```
mod [
  local : lin := []<>;
  f : int := nil;

  invariant
    Btwn(local,f,nil) = (dom(local) ∪ {nil})

  procedure alloc(h:lin) returns (res:int)
    requires dom(h) = ∅
    ensures res != nil ∧ res ∈ dom(h)
  {
    if (f = nil)
      res := malloc(h);
    else {
      res := f;
      f := local[res];
      h := local@{res};
    }
  }

  procedure free(arg:int,h:lin) returns ()
    requires arg != nil ∧ arg ∈ dom(h)
  {
    local := h@{arg};
    local[arg] := f;
    f := arg;
  }
]
```

Figure 7. Memory manager

```
var ref1, ref2 : int in
var tmp1, tmp2 : lin in
  ref1 := alloc(tmp1);
  tmp1[ref1] := 3;
  ref2 := alloc(tmp2);
  assume dom(tmp1) ∩ dom(tmp2) = ∅;
  assert ref1 != ref2;
  tmp1 := tmp2@{ref2};
  tmp1[ref2] := 6;
  assert tmp1[ref1] = 3;
```

Figure 8. Client of memory manager

states that the list hanging from `f` is acyclic and the domain of `local` includes all elements in this list except possibly `nil`.

The procedure `alloc` returns an address from the head of the list if the list is nonempty; before returning, it transfers the returned address from the domain of `local` to `h`. If the list is empty, `alloc` simply calls a lower-level procedure `malloc` with the same interface as `alloc`. Thus, `alloc` and `malloc` model two different operations with same functional specification but with different latency. The module invariant described earlier is crucial for proving the safety of the read and transfer operations on `local`. The procedure `free` appends the pointer `arg` to the beginning of the list and transfers `arg` from the domain of `h` to `local`.

The verification of the client code, given in Figure 8, reveals another interesting feature of our proof system. The statement `assume dom(tmp1) ∩ dom(tmp2) = ∅` allows downstream code to use the assumed fact for verification. It is sound to make this assumption because `tmp1` and `tmp2` are distinct linear map variables whose disjointness is assured by the disjoint domains invariant. This assumption, together with the postcondition of `alloc`, is sufficient to verify the assertion `assert ref1 != ref2` following it. Note that the postcondition of `alloc` is not strong enough to verify this assertion by itself. All necessary disjointedness assumptions could easily be inferred by the system without user annotations – we have

$$
\begin{array}{lll}
\text{types} & \tau & ::= \texttt{int} \mid \texttt{tot} \mid \texttt{lin} \mid \texttt{set} \\
\text{values} & v & ::= n \mid f \mid \ell \mid s \\
\text{logical exps} & e & ::= x \mid v \mid e_1 + e_2 \mid \texttt{sel}(e_1, e_2) \mid \texttt{upd}(e_1, e_2, e_3) \\
 & & \quad \mid \texttt{ite}(e_1, e_2, e_3) \mid \texttt{sel}^{\texttt{L}}(e_1, e_2) \mid \texttt{upd}^{\texttt{L}}(e_1, e_2, e_3) \\
 & & \quad \mid \texttt{map}(e) \mid \texttt{dom}(e) \mid \texttt{lin}(e_1, e_2) \mid \{x \mid F\} \\
\text{formulae} & F & ::= \texttt{true} \mid \texttt{false} \mid \neg F \mid F_1 \vee F_2 \mid F_1 \wedge F_2 \\
 & & \quad \mid F_1 \Rightarrow F_2 \mid \exists x{:}\tau.F \mid \forall x{:}\tau.F \\
 & & \quad \mid e_1 = e_2 \mid e_1 \in e_2
\end{array}
$$

Figure 9. Syntax of values, expressions and logical formulae

not done so in this presentation to highlight the disjoint domains invariant as its own independent, orthogonal concept.

The examples in this section have been mechanized using Boogie. The verification of the memory manager module depends on reasoning about the `Btwn(l,a,b)` set constructor in the presence of updates to `l` and transfers to and from the domain of `l`. We have extended the decision procedure for reachability [16] with a collection of rewrite rules based on e-matching [9] for modeling the interaction between `Btwn` and the semantics of transfer between linear maps. Although examples in this section only illustrate the use of transfer of singleton sets, we have verified an example involving multiple lists that requires transferring the contents of an entire list.

3. Technical Development

This section presents the technical intricacies involved in developing a program logic for imperative programs with linear maps.

3.1 The Assertions

Figure 9 presents the language of assertions. Here and elsewhere, x ranges over variables, n ranges over integers and s ranges over sets of integers. When we want to represent a specific set, we will use standard set-theoretic notation such as $\{x \mid x > 0\}$. Metavariable f ranges over total maps from integers to integers. When we want to represent a specific total map, we will use standard notation from the lambda calculus such as $\lambda x.x + 1$.

Linear maps are simply pairs of a total map f from integers to integers and a domain s. Intuitively, the pair of total map and domain implements a partial map. We let ℓ range over linear maps. In general, when linear map ℓ is the pair f_s, we let $\mathrm{map}(\ell)$ refer to the underlying total map f and $\mathrm{dom}(\ell)$ refer to the underlying domain s. We write $[]_\emptyset$ to refer to a particular linear map whose domain is the empty set.

The logic is built upon a collection of simple expressions e, whose denotations are values with one of four primitive types: integer (`int`), total map (`tot`), linear map (`lin`), and integer set (`set`). The expressions include variables, values of each type, and a collection of simple operations on each type. For total maps, we allow the standard operations select (`sel`) and update (`upd`). For instance, $\texttt{sel}(e_1, e_2)$ selects element e_2 from the total map e_1 while $\texttt{upd}(e_1, e_2, e_3)$ updates total map e_1 at location e_2 with the value denoted by e_3. In addition, we will allow the use of a generalized map update with the form $\texttt{ite}(e_1, e_2, e_3)$ (pronounced "if then else") where e_1 is a set and e_2 and e_3 are two additional total maps. This expression is equal to the total map that acts as e_2 when its argument belongs to the set e_1 and acts as e_3 when its argument does not belong to the set e_1. This non-standard map constructor fits within the framework of de Moura and Bjørner's recent work on generalized array decision procedures [7] and is supported in Z3 [6]; it can be supported in other automated theorem provers that support quantifiers via quantified axioms.

$$\boxed{[\![e]\!]_E = v}$$

$$
\begin{array}{ll}
[\![x]\!]_E & = E[x] \\
[\![v]\!]_E & = v \\
[\![e_1 + e_2]\!]_E & = [\![e_1]\!]_E + [\![e_2]\!]_E \\
[\![\texttt{sel}(e_1, e_2)]\!]_E & = [\![e_1]\!]_E([\![e_2]\!]_E) \\
[\![\texttt{upd}(e_1, e_2, e_3)]\!]_E & = \lambda x.\texttt{if} x = [\![e_2]\!]_E \texttt{then} [\![e_3]\!]_E \texttt{else} [\![e_1]\!]_E(x) \\
[\![\texttt{ite}(e_1, e_2, e_3)]\!]_E & = \lambda x.\texttt{if} x \in [\![e_1]\!]_E \texttt{then} [\![e_2]\!]_E(x) \texttt{else} [\![e_3]\!]_E(x) \\
[\![\texttt{sel}^{\texttt{L}}(e_1, e_2)]\!]_E & = \mathrm{map}([\![e_1]\!]_E)([\![e_2]\!]_E) \\
[\![\texttt{upd}^{\texttt{L}}(e_1, e_2, e_3)]\!]_E & = f_{\mathrm{dom}([\![e_1]\!]_E) \cup \{[\![e_2]\!]_E\}} \\
 & \quad \text{where } f = \lambda x.\texttt{if} x = [\![e_2]\!]_E \texttt{ then } [\![e_3]\!]_E \texttt{ else } \mathrm{map}([\![e_1]\!]_E)(x) \\
[\![\texttt{map}(e)]\!]_E & = \mathrm{map}([\![e]\!]_E) \\
[\![\texttt{dom}(e)]\!]_E & = \mathrm{dom}([\![e]\!]_E) \\
[\![\texttt{lin}(e_1, e_2)]\!]_E & = ([\![e_1]\!]_E)_{[\![e_2]\!]_E} \\
[\![\{x \mid F\}]\!]_E & = \{v \mid E, x = v \models F\}
\end{array}
$$

Figure 10. Denotational Semantics of Expressions

The expressions $\texttt{sel}^{\texttt{L}}(e_1, e_2)$ and $\texttt{upd}^{\texttt{L}}(e_1, e_2, e_3)$ are variants of the standard select and update expressions designed to operate on linear maps. The $\texttt{sel}^{\texttt{L}}(e_1, e_2)$ expression selects e_2 from the underlying total map of e_1. As far as the semantics of logical expressions are concerned, e_2 may lie outside the domain of e_1. In later subsections, the reader will see how the program logic will use explicit domain checks to guarantee that reads and writes of linear map program variables do not occur outside their domains during program execution. The $\texttt{upd}^{\texttt{L}}(e_1, e_2, e_3)$ updates linear map e_1 at location e_2 with the value denoted by e_3. If e_2 does not appear in the domain of e_1, then the domain of the resulting linear map is one element larger than the domain of the initial map. The expressions $\texttt{map}(e)$ and $\texttt{dom}(e)$ extracts the underlying total map and underlying domain of linear map e while the expression $\texttt{lin}(e_1, e_2)$ constructs a linear map from total map e_1 and set e_2. The expression $\{x \mid F\}$ denotes a set of integers x that satisfy formula F. We will freely use other operations on sets such as union and intersection as they may be encoded.

The logical formulae themselves include the usual formulae from first-order logic as well as equality and set inclusion.

Throughout the paper, we will only consider well-typed expressions and formulae. Given a type environment Γ, which is a finite partial map from variables to their types, we write $\Gamma \vdash e : \tau$ to denote that e is a well-formed expression with type τ. Likewise, we write $\Gamma \vdash F : \texttt{prop}$ to denote that formula F is a well-formed formula. The rules for defining these judgments are simple and standard and therefore we omit them.

In Figure 10, expressions are given semantics through a judgement with the form $[\![e_1]\!]_E$. Here, and elsewhere, E is a finite partial map from variables to values. We write $E[x]$ to look up the value associated with x in E. We write $E, x = v$ to extend E with x (assuming x does not already appear in the domain of E). We write $E[x = v]$ to update E with a new value v for x. A value environment E has a type Γ, written $\vdash E : \Gamma$, when the domains of Γ and E are equal and for every binding $x{:}\tau$ in Γ there exists a corresponding value $E[x]$ with type τ.

Given the semantics of expressions, the semantics of formulae is entirely standard. When an environment E satisfies a formula F, we write $E \models F$. When a formula F is valid with respect to any environment with type Γ, we write $\Gamma \models F$.

3.2 Programs

Figure 11 presents the formal syntax of programs. The main syntactic program elements are expressions, statements and modules.

$$
\begin{array}{llll}
\text{impl exps} & Z & ::= & x \mid n \mid Z_1 + Z_2 \\
\text{ghost exps} & S & ::= & e \\
\text{statements} & C & ::= & x := Z \mid x_1 :=^{\mathtt{G}} S \\
 & & \mid & \mathtt{var}\, x{:}\tau\, \mathtt{in}\, C \mid \mathtt{skip} \mid C_1; C_2 \\
 & & \mid & \mathtt{if}\, Z\, \mathtt{then}\, C_1\, \mathtt{else}\, C_2 \\
 & & \mid & \mathtt{while}\, [F]\, Z\, \mathtt{do}\, C \\
 & & \mid & \mathtt{assert}\, F \mid x_3 := g(x_1, x_2) \\
 & & \mid & x_1 :=^{\mathtt{L}} x_2[Z] \mid x[Z_1] :=^{\mathtt{L}} Z_2 \\
 & & \mid & x_1 := x_2@S \mid x_1 :=^{\mathtt{L}} x_2 \\
 & & \mid & \mathtt{assume}\, \mathrm{dom}(x_1) \cap \mathrm{dom}(x_2) = \emptyset \\
\text{mod clause} & \mathit{mod} & ::= & \{x_1, \ldots, x_k\} \\
\text{fun types} & \sigma & ::= & \forall arg_1{:}\tau_1, arg_2{:}\tau_2.F_1 \xrightarrow{mod} \exists ret{:}\tau_3.F_2 \\
\text{mods} & mv & ::= & [E; F_{inv}; g{:}\sigma = C] \\
\text{mod env's} & M & ::= & \cdot \mid M, mv \\
\text{states} & \Sigma & ::= & (M; E) \\
\text{programs} & \mathit{prog} & ::= & (\Sigma; C)
\end{array}
$$

Figure 11. Syntax of Programs

Program expressions are divided into three major categories: *implementation expressions* (Z), *ghost expressions* (S) and *linear expressions*. Implementation expressions are those expressions that are executed unchanged by the underlying abstract machine. Ghost expressions are expressions that are used to help specify the behavior of programs, but are not needed at run time and hence will be erased by the erasure translation. Ghost expressions may include or depend upon implementation expressions, but implementation expressions may not depend upon ghost expressions. Linear expressions are expressions that involve linear maps. These expressions are partially erased: the erasure translation replaces references to linear maps with references to the single underlying heap. Linear expressions must be constrained to ensure they are not copied.

For the purposes of this paper, we segregate the different sorts of expressions using their types. More specifically, the `int` type is our only *implementation type*. The types `tot` and `set` are our *ghost types*. The type `lin` is our *linear type*. We write $\mathtt{impl}(\tau)$ when τ is an implementation type, $\mathtt{ghost}(\tau)$ when τ is a ghost type and $\mathtt{linear}(\tau)$ when τ is a linear type. We write $\mathtt{nonlinear}(\tau)$ when τ is not a linear type. We also use these predicates over closed values, as the type of a closed value is evident from its syntax.

Statements C include standard elements of any imperative language: assignment, `skip`, sequencing, conditionals, while loops, asserts, function calls and local variables. We assume local variables and other binding occurrences alpha-vary as usual. We require function arguments be variables to enable a slight simplification of the verification rules. In addition to a normal assignment, we include a linear assignment and a ghost assignment. Operationally, the linear assignment not only assigns the source to the target, but it also assigns the empty map to the source to ensure locations are not copied and the disjoint domains invariant is preserved. The ghost assignment acts as an ordinary assignment, though the language type system will prevent implementation types from depending upon it. Reading from and writing to total maps is provided via ghost assignments.

To read from location Z in linear map x_2 and assign that value to variable x_1, programmers use the statement $x_1 :=^{\mathtt{L}} x_2[Z]$. To update location Z_1 in linear map x with value Z_2, programmers use the statement $x[Z_1] :=^{\mathtt{L}} Z_2$. The statement $x_1 := x_2@S$ transfers the portion of linear map x_2 with domain S to x_1. The statement $x_1 :=^{\mathtt{L}} x_2$ is a special case of the transfer operation in which the entire contents of x_2 is transferred to x_1. Finally, $\mathtt{assume}\, \mathrm{dom}(x_1) \cap \mathrm{dom}(x_2) = \emptyset$ is a no-op that introduces the fact that two linear maps have disjoint domains into the theorem-proving context.

Modules mv consist of a private environment, an invariant and, for simplicity, a single, non-recursive function. These functions are declared to have a name g, a type σ and a body C. For simplicity again, functions are constrained to take two arguments, where the first is non-linear and the second is a linear map. The first argument is immutable within the body of the function and the second is a mutable input-output parameter. The argument variables arg_1 and arg_2 may appear free in the precondition F_1, the postcondition F_2 and the body of the function. Since arg_1 is immutable in the body of the procedure, its value in the postcondition is the same as its value on entry to the procedure. Since arg_2 is mutable in the body of the procedure, its value in the postcondition is *not* necessarily the same as its value on entry to the procedure – its value will reflect any effects that occur during execution of the procedure. The result variable ret may appear free in the postcondition and may be assigned to in the function body. The set mod on the function type arrow specifies the variables that may be modified during execution of the function. The collection of constraints on the form of a function signature are specified using a judgment with the form $\Gamma \vdash \sigma$ (not shown). For convenience, we often refer to a module using the name of the function that it contains. For instance, given a list of modules M, we select the module containing the function g using the notation $M(g)$. We assume the same function name g is never used twice in a list of modules.

A complete program consists of state Σ and the statement C to execute. A state $\Sigma = (M, E)$ is a list of modules M paired with a global environment E. We assume no variable x is bound both in the global environment E and in some module local environment in M (alpha-converting where necessary). We let $|\Sigma|_{env}$ be the environment formed by concatenating the module environments to the global environment from Σ. We also lift most operations on environments to operations on states in the obvious way. For instance, $\Sigma[x]$ looks up the value bound to x in any environment in Σ and $\Sigma[x = v]$ updates variable x with v in any environment in Σ. $\Sigma, x = v$ extends the global environment in Σ with the binding $x = v$ assuming x does not already appear in Σ. Finally, $[\![e_1]\!]_\Sigma$ abbreviates $[\![e_1]\!]_{|\Sigma|_{env}}$ and $\Sigma \models F$ abbreviates $|\Sigma|_{env} \models F$.

3.3 The Program Logic

The program logic is defined by two primary judgment forms: one for statements and one for modules. The judgment for verification of statements has the form $G; \Gamma; mod \vdash \{F_1\} C \{F_2\}$. Here, G is a function context that maps function variables to their types, Γ is a value type environment that maps value variables to their types and mod is the set of variables that may be modified by the enclosed statement. Given this context, F_1 is the statement precondition, C the statement to be verified, and F_2 is the postcondition. The rules for this judgement form are given in figures 12 and 13.

Figure 12 presents the most basic rules for statement verification including the rule of consequence and the frame rule. This figure contains two rules for assignments: (Asgn) and (Ghst). (Asgn) handles assignment for implementation types and (Ghst) handles assignments for ghost types. The rules are identical, save the type checking component. They are separated to simplify the definition of the erasure translation, which will delete the ghost assignment but leave the implementation assignment untouched. The rule for variables in this figure is standard, though it applies only to introduction of variables with non-linear type. Linear variable declarations (as well as linear assignments) will discussed shortly. We have omitted rules for `skip`, sequencing, if statements, and while loops as they are standard.

Figure 13 presents the verification rules that are concerned with maps and function calls. The first rule in the figure is the linear as-

$\boxed{G;\Gamma;\mathit{mod} \vdash \{F_1\}C\{F_2\}}$

$$\frac{\Gamma \vdash x_1 : \texttt{lin} \quad \Gamma \vdash x_2 : \texttt{lin} \quad x_1, x_2, x_2' \text{ are distinct variables} \quad x_1, x_2 \in \mathit{mod} \quad x_2' \notin FV(F)}{G;\Gamma;\mathit{mod} \vdash \{\mathrm{dom}(x_1) = \emptyset \wedge \forall x_2' {:}\texttt{lin}.\mathrm{dom}(x_2') = \emptyset \Rightarrow F[x_2'/x_2][x_2/x_1]\}x_1 :=^{\texttt{L}} x_2\{F\}} \text{ (Asgn Lin)}$$

$$\frac{x \notin (\mathrm{dom}(\Gamma) \cup FV(F_2)) \quad G;\Gamma, x{:}\texttt{lin};\mathit{mod} \cup \{x\} \vdash \{F_1\}C\{F_2\}}{G;\Gamma;\mathit{mod} \vdash \{\forall x{:}\texttt{lin}.\mathrm{dom}(x) = \emptyset \Rightarrow F_1\}\texttt{var}\ x{:}\texttt{lin}\ \texttt{in}\ C\{F_2\}} \text{ (Var Lin)}$$

$$\frac{\Gamma \vdash x_1 : \texttt{int} \quad \Gamma \vdash x_2 : \texttt{lin} \quad \Gamma \vdash Z : \texttt{int} \quad x_1 \in \mathit{mod}}{G;\Gamma;\mathit{mod} \vdash \{Z \in \mathrm{dom}(x_2) \wedge F[\texttt{sel}^{\texttt{L}}(x_2, Z)/x_1]\}x_1 :=^{\texttt{L}} x_2[Z]\{F\}} \text{ (Linear Map Select)}$$

$$\frac{\Gamma \vdash Z_1 : \texttt{int} \quad \Gamma \vdash Z_2 : \texttt{int} \quad \Gamma \vdash x : \texttt{lin} \quad x \in \mathit{mod}}{G;\Gamma;\mathit{mod} \vdash \{Z_1 \in \mathrm{dom}(x) \wedge F[\texttt{upd}^{\texttt{L}}(x, Z_1, Z_2)/x]\}x[Z_1] :=^{\texttt{L}} Z_2\{F\}} \text{ (Linear Map Update)}$$

$$\frac{\Gamma \vdash x : \texttt{lin} \quad \Gamma \vdash y : \texttt{lin} \quad \Gamma \vdash S : \texttt{set} \quad x, y \in \mathit{mod}}{G;\Gamma;\mathit{mod} \vdash \{S \subseteq \texttt{dom}(y) \wedge F[\texttt{lin}(\texttt{ite}(S, \texttt{map}(y), \texttt{map}(x)), \texttt{dom}(x) \cup S)/x][\texttt{lin}(\texttt{map}(y), \texttt{dom}(y) - S)/y]\}x := y@S\{F\}} \text{ (Transfer)}$$

$$\frac{\Gamma \vdash x_1 : \texttt{lin} \quad \Gamma \vdash x_2 : \texttt{lin} \quad x_1, x_2 \text{ are distinct variables}}{G;\Gamma;\mathit{mod} \vdash \{\mathrm{dom}(x_1) \cap \mathrm{dom}(x_2) = \emptyset \Rightarrow F\}\texttt{assume}\,\mathrm{dom}(x_1) \cap \mathrm{dom}(x_2) = \emptyset\{F\}} \text{ (Assume)}$$

$$\frac{\begin{array}{c} G(g) = \forall \mathit{arg}_1{:}\tau_1, \mathit{arg}_2{:}\tau_2.F_1 \xrightarrow{\mathit{mod}'} \exists \mathit{ret}{:}\tau_3.F_2 \\ \Gamma \vdash x_1 : \tau_1 \quad \Gamma \vdash x_2 : \tau_2 \quad \Gamma \vdash x_3 : \tau_3 \quad (\mathit{mod}' \cup \{x_2, x_3\}) \subseteq \mathit{mod} \quad x_1, x_2, x_3 \notin FV(G(g)) \end{array}}{G;\Gamma;\mathit{mod} \vdash \{F_1[x_1/\mathit{arg}_1][x_2/\mathit{arg}_2]\}x_3 := g(x_1, x_2)\{F_2[x_1/\mathit{arg}_1][x_2/\mathit{arg}_2][x_3/\mathit{ret}]\}} \text{ (Call)}$$

Figure 13. Program Logic: Linear Statements and Function Calls

$\boxed{G;\Gamma;\mathit{mod} \vdash \{F_1\}C\{F_2\}}$

$$\frac{\begin{array}{c} \Gamma \models F_1 \Rightarrow F_1' \\ G;\Gamma;\mathit{mod} \vdash \{F_1'\}C\{F_2'\} \\ \Gamma \models F_2' \Rightarrow F_2 \end{array}}{G;\Gamma;\mathit{mod} \vdash \{F_1\}C\{F_2\}} \text{ (Consequence)}$$

$$\frac{G;\Gamma;\mathit{mod} - FV(R) \vdash \{F_1\}C\{F_2\}}{G;\Gamma;\mathit{mod} \vdash \{F_1 \wedge R\}C\{F_2 \wedge R\}} \text{ (Frame)}$$

$$\frac{\Gamma \vdash x : \tau \quad \texttt{impl}(\tau) \quad \Gamma \vdash Z : \tau \quad x \in \mathit{mod}}{G;\Gamma;\mathit{mod} \vdash \{F[Z/x]\}x := Z\{F\}} \text{ (Asgn)}$$

$$\frac{\Gamma \vdash x : \tau \quad \texttt{ghost}(\tau) \quad \Gamma \vdash S : \tau \quad x \in \mathit{mod}}{G;\Gamma;\mathit{mod} \vdash \{F[S/x]\}x :=^{\texttt{G}} S\{F\}} \text{ (Ghst)}$$

$$\frac{\Gamma \vdash F' : \texttt{prop}}{G;\Gamma;\mathit{mod} \vdash \{F' \wedge F\}\texttt{assert}\,F'\{F\}} \text{ (Assert)}$$

$$\frac{\begin{array}{c} x \notin (\mathrm{dom}(\Gamma) \cup FV(F_2)) \quad \texttt{nonlinear}(\tau) \\ G;\Gamma, x{:}\tau;\mathit{mod} \cup \{x\} \vdash \{F_1\}C\{F_2\} \end{array}}{G;\Gamma;\mathit{mod} \vdash \{\forall x{:}\tau.F_1\}\texttt{var}\ x{:}\tau\ \texttt{in}\ C\{F_2\}} \text{ (Var)}$$

Figure 12. Program Logic: The Basics (Selected Rules)

signment rule (Asgn Lin). This rule demands that the variable x_1 is the empty map prior to assignment and x_2 is the empty map after assignment. The quantified statement in the precondition of the rule states that x_2 may be assigned *any* empty linear map x_2' (*i.e.,* a linear map with empty domain and any underlying total map). These constraints ensure that an assignment neither copies linear map addresses (thereby preserving the disjoint domains invariant) nor overwrites them (thereby simplifying the correspondence between linear maps and heaps in the erasure translation). Note also that both x_1 and x_2 are considered modified by this statement. The second rule (Var Lin) illustrates that declaring a linear variable is the same as declaring a non-linear variable except for the constraint that the linear variable initially contains an empty linear map.

Rules (Linear Map Select) and (Linear Map Update) are modeled after their nonlinear counterparts, with one addition: before using a linear map, a programmer must prove that their linear map access falls within the domain of the linear map.

The (Transfer) rule first checks that the two maps in consideration, x (the map transferred to) and y (the map transferred from) can both be modified. If they can be modified, the Hoare rule itself acts as a specialized assignment rule where a new map that acts as y on S and x elsewhere (*i.e.,* $\texttt{lin}(\texttt{ite}(S, \texttt{map}(y), \texttt{map}(x)), \texttt{dom}(x) \cup S)$) is assigned to x and another new map that acts as y, but has a smaller domain (*i.e.,* $\texttt{lin}(\texttt{map}(y), \texttt{dom}(y) - S)$) is assigned to y.

The second last rule (Assume) allows the theorem proving environment to be extended with the fact that the domains of x_1 and x_2 are disjoint, provide x_1 and x_2 are distict linear map variables. This rule directly exploits the disjoint domains invariant.

The last statement rule is (Call). This rule looks up the function signature in the context and checks its arguments and result have the appropriate types. It also verifies that the variables modified by the function (mod') are subset of those that may be modified in this context (mod). Finally, it checks that both x_2 and x_3 may be

modified. The variable x_3 is clearly modified as it is the target of an assignment. However, beware that x_2 is also modified as it is a linear map and its entire contents are *transferred* to the second parameter of the call upon entry to the function, and then upon return, a mutated linear map is *transferred* back. Such transfers are necessary (as opposed to copies) to maintain the disjoint domains invariant. The first argument to the call x_1 is not mutated: as a non-linear value, it may simply be copied into the parameter. To simplify our formulation of the preconditions and postconditions for the triple, we add the constraint that none of x_1, x_2 or x_3 may appear free in the function signature (either the precondition, postcondition or modifies clause).

Figure 14 defines the judgment form for verification of modules: $G; \Gamma \vdash mv \Rightarrow G'$. Intuitively, the module contents are type checked in one environment $(G; \Gamma)$ and the result is an extended context (G') for the newly declared functions. Hence, this formalizes simple non-recursive modules; extending the system further with mutually recursive modules is orthogonal to the use of linear maps. For simplicity, all module-local variables are private (as opposed to public), and hence, unlike G, Γ is not extended. The most interesting elements of the rule are:

- The private module environment must have some type Γ_E.
- The module invariant F_{inv} is checked for well-formedness with respect only to the private environment ($\Gamma_E \vdash F_{inv} : \texttt{prop}$). This check implies F_{inv} may only contain the private variables of the current module, which may not be modified by code outside the module.
- The module invariant is valid in the initial environment E.
- When checking the body of the module function, F_{inv} is assumed initially and proven upon exit. However, F_{inv} does not appear in σ, meaning it is hidden from module clients.
- The module function may modify any variable in its declared modifies clause as well as the return variables and the private environment. The domain of the private environment does not appear in the function modifies cause (or elsewhere in the function signature), meaning these variables are hidden from clients.

Figure 14 contains definitions for several further judgement forms for verifying lists of modules, states and finally programs as a whole. The judgement $\Gamma \vdash M \Rightarrow G$ simply chains together the verification of all modules M in sequence. This judgment disallows mutual recursion amongst modules. The issues involved with mutual recursion, temporarily broken module invariants, and reentracy are orthogonal to issues involving linear maps. The judgment $\vdash \Sigma \Rightarrow G; \Gamma$ verifies a state, which includes both modules and global environment. Finally, a program *prog* is said to be well-formed and to establish a post-condition F_2 when the judgment $\vdash (\Sigma; C) : F_2$ is valid. This judgment verifies the underlying state Σ and then uses the generated verification context to check the statement C satisfies some appropriate Hoare triple with post-condition F_2.

The program checking rule relies on one other judgment $\vdash E$ wf, whose definition we have omitted, but is easy to define. This latter judgment ensures that the initial environment satisfies the disjoint domains invariant. In practice, a sensible way to perform this global disjoint domains check is to check that all declared linear map variables are initially bound to the empty map (as we have done in our examples), save one, which is bound to the *priomordial map*, a linear map initially containing all addresses. Given a single private priomordial map, it is easy to write an allocator module that hands out addresses to other modules according to any invariant the programmer chooses. In theory, however, it is irrelevant what specific initial conditions are chosen provided that the disjoint domains condition holds.

3.4 Operational Semantics

The operational semantics of our language is specified as a judgment with the form $prog \longrightarrow prog$. To facilitate the proof of soundness, we extend the syntax of statements with one additional statement with the form $g[C]$. This new statement form arises when a function g is called and execution begins on g's body (which will be the statement C inside the square brackets). The $g[\cdot]$ annotation has no real operational effect, but its presence serves as a reminder that code within $g[\cdot]$ has access to the private variables of g's module and must establish the invariant for g's module prior to completion. Figure 15 presents the formal rules. Operational rules for sequencing, if, and while are standard and were omitted.

The first point of interest in the operational semantics involves the linear assignment rule (OS Asgn Lin). This instruction resets the source of the linear assignment to the empty map to prevent duplication of addresses and to maintain the disjoint domains invariant. In the (OS Var) rule, we assume the existence of a function $\mathcal{I}$ that maps types to sets of legal initial values for that type. For integers, sets, and total maps, any initial value may be generated. For linear maps, only the empty map may be generated.

The primary effect of rule (OS Call1) is to look up the module corresponding to the function g in the program state, evaluate the function arguments, and replace the call with $g[C]$ where C is the body of g. In addition, however, the call creates environment bindings for the argument and result variables, sets the linear map argument x_2 to the empty map and sets up the instructions to copy the results ret and arg_2 back to variables visible in the current context (x_3 and x_2 respectively). A linear assignment is used to copy arg_2 back to x_2 after the call, ensuring that at no point is the disjoint domains invariant ever broken. Rule (OS Call2) allows ordinary execution underneath the $g[\cdot]$ annotation and rule (OS Call3) discards the $g[\cdot]$ annotation when control leaves that scope.

The remaining rules are less interesting. We leave the reader to investigate the specifics.

3.5 Soundness

The first key property of our language is that it is *sound*. In other words, execution of verified programs never encounters assertion failures, or fails domain checks on linear maps and, if execution terminates, the postcondition will be valid in the final state. The following definition and theorem state these properties formally. The relation $prog \longrightarrow^* prog$ is the reflexive, transitive closure of $prog \longrightarrow prog$.

Definition 1 (Stuck Program)
A program $(\Sigma; C)$ *is* stuck *if* C *is not* `skip` *and there does not exist another state* $(\Sigma'; C')$ *such that* $(\Sigma; C) \longrightarrow (\Sigma'; C')$ *.*

Theorem 2 (Soundness)
If $\vdash (\Sigma; C) : F_2$ *and* $(\Sigma; C) \longrightarrow^* (\Sigma'; C')$ *then* $(\Sigma'; C')$ *is not stuck and if* $C' = \texttt{skip}$ *then* $\Sigma' \models F_2$ *.*

The proof is carried out using standard syntactic techniques and employs familiar Preservation and Progress lemmas. We have checked all the main top-level cases for these lemmas by hand, but have assumed a number of necessary underlying lemmas such as substitution, weakening, and some others are true without detailed proof. We are confident in our results because the difficult elements of proof have nothing to do with linear maps at all. Rather, difficulties in the proof revolved around the structure of modules, and, in particular, setting up the technical machinery to track the scopes of private module variables and the validity of module invariants as functions are called.

$\boxed{G;\Gamma \vdash mv \Rightarrow G'}$

$$\frac{\begin{array}{c}\sigma = \forall arg_1{:}\tau_1, arg_2{:}\tau_2.F_1 \xrightarrow{mod'} \exists ret{:}\tau_3.F_2 \quad \Gamma \vdash \sigma \\ g \notin \mathrm{dom}(G) \quad (\mathrm{dom}(\Gamma_E) \cup \{arg_1, arg_2, ret\}) \cap \mathrm{dom}(\Gamma) = \emptyset \\ \vdash E : \Gamma_E \quad \Gamma_E \vdash F_{inv} : \mathtt{prop} \quad E \models F_{inv} \\ G;\Gamma,\Gamma_E,arg_1{:}\tau_1,arg_2{:}\tau_2,ret{:}\tau_3;mod' \cup \mathrm{dom}(\Gamma_E) \cup \{arg_2,ret\} \vdash \{F_1 \wedge F_{inv}\}C\{F_2 \wedge F_{inv}\}\end{array}}{G;\Gamma \vdash [E; F_{inv}; g{:}\sigma = C] \Rightarrow G, g{:}\sigma} \text{ (Mod)}$$

$\boxed{\Gamma \vdash M \Rightarrow G}$

$$\frac{}{\Gamma \vdash \cdot \Rightarrow \cdot} \text{ (Mod Env Emp)} \qquad \frac{\Gamma \vdash M \Rightarrow G' \quad G';\Gamma \vdash mv \Rightarrow G''}{\Gamma \vdash M, mv \Rightarrow G''} \text{ (Mod Env)}$$

$\boxed{\vdash \Sigma \Rightarrow G;\Gamma}$

$$\frac{\vdash E : \Gamma \quad \Gamma \vdash M \Rightarrow G}{\vdash (M;E) \Rightarrow G;\Gamma} \text{ (State)}$$

$\boxed{\vdash prog : F_2}$

$$\frac{\vdash \Sigma \Rightarrow G;\Gamma \quad \vdash |\Sigma|_{env}\ \mathtt{wf} \quad \Gamma \vdash F_1 : \mathtt{prop} \quad \Gamma \vdash F_2 : \mathtt{prop} \quad \Sigma \models F_1 \quad G;\Gamma;\mathrm{dom}(\Gamma) \vdash \{F_1\}C\{F_2\}}{\vdash (\Sigma; C) : F_2} \text{ (Programs)}$$

Figure 14. Program Logic: Modules, States and Programs

$$\frac{}{(\Sigma; x := Z) \longrightarrow (\Sigma[x = [\![Z]\!]_\Sigma]; \mathtt{skip})} \text{ (OS Asgn)} \qquad \frac{}{(\Sigma; x_1 :=^{\mathtt{L}} x_2) \longrightarrow (\Sigma[x_1 = [\![x_2]\!]_\Sigma][x_2 = [\,]_\emptyset]; \mathtt{skip})} \text{ (OS Asgn Lin)}$$

$$\frac{}{(\Sigma; x :=^{\mathtt{G}} S) \longrightarrow (\Sigma[x = [\![S]\!]_\Sigma]; \mathtt{skip})} \text{ (OS Asgn Ghst)}$$

$$\frac{v \in \mathcal{I}(\tau) \quad x \notin \mathrm{dom}(|\Sigma|_{env})}{(\Sigma; \mathtt{var}\, x{:}\tau\, \mathtt{in}\, C) \longrightarrow (\Sigma, x = v; C)} \text{ (OS Var)} \qquad \frac{\Sigma \models F}{(\Sigma; \mathtt{assert}\, F) \longrightarrow (\Sigma; \mathtt{skip})} \text{ (OS Assert)}$$

$$\frac{\Sigma(g) = [E'; F_{inv}; g{:}\forall arg_1{:}\tau_1, arg_2{:}\tau_2.F_1 \xrightarrow{mod'} \exists ret{:}\tau_3.F_2 = C] \quad arg_1, arg_2, ret \notin \mathrm{dom}(|\Sigma|_{env}) \quad v_3 \in \mathcal{I}(\tau_3)}{(\Sigma; x_3 := g(x_1, x_2)) \longrightarrow ((\Sigma[x_2 = [\,]_\emptyset]), arg_1{=}[\![x_1]\!]_\Sigma, arg_2{=}[\![x_2]\!]_\Sigma, ret{=}v_3; g[C]; x_3 := ret; x_2 :=^{\mathtt{L}} arg_2)} \text{ (OS Call1)}$$

$$\frac{(\Sigma; C) \longrightarrow (\Sigma'; C')}{(\Sigma; g[C]) \longrightarrow (\Sigma'; g[C'])} \text{ (OS Call2)} \qquad \frac{}{(\Sigma; g[\mathtt{skip}]) \longrightarrow (\Sigma; \mathtt{skip})} \text{ (OS Call3)}$$

$$\frac{[\![Z]\!]_\Sigma = n \quad [\![x_2]\!]_\Sigma = f_s^c \quad n \in s}{(\Sigma; x_1 :=^{\mathtt{L}} x_2[Z]) \longrightarrow (\Sigma[x_1 = f(n)]; \mathtt{skip})} \text{ (OS Linear Map Select)}$$

$$\frac{[\![x]\!]_\Sigma = f_s \quad n_1 \in s \quad [\![Z_1]\!]_\Sigma = n_1 \quad [\![Z_2]\!]_\Sigma = n_2}{(\Sigma; x[Z_1] :=^{\mathtt{L}} Z_2) \longrightarrow (\Sigma[x = (\lambda x.\mathtt{if}\, x = n_1\, \mathtt{then}\, n_2\, \mathtt{else}\, f\, x)_s]; \mathtt{skip})} \text{ (OS Linear Map Update)}$$

$$\frac{[\![x_1]\!]_\Sigma = f_{s_1} \quad [\![x_2]\!]_\Sigma = h_{s_2} \quad [\![S]\!]_\Sigma = s_3}{(\Sigma; x_1 := x_2@S) \longrightarrow (\Sigma[x_1 = (\lambda x.\mathtt{if}\, x \in s_3\, \mathtt{then}\, h\, x\, \mathtt{else}\, f\, x)_{s_1 \cup s_3}][x_2 = h_{s_2 - s_3}]; \mathtt{skip})} \text{ (OS Transfer)}$$

$$\frac{}{(\Sigma; \mathtt{assume}\, \mathrm{dom}(x_1) \cap \mathrm{dom}(x_2) = \emptyset) \longrightarrow (\Sigma; \mathtt{skip})} \text{ (OS Assume)}$$

Figure 15. Operational Semantics (Selected Statements)

3.6 Erasure

A second key property of our language is that all verified programs can be implemented efficiently as ordinary imperative programs. More precisely, we prove that our original operational semantics on linear maps is equivalent to one in which ghost expressions are erased and linear maps are replaced by accesses to a single, global heap. To make these ideas precise, we define an erasure function that maps *linear maps programs* into *heap-based programs*. The main work done by the program erasure function is accomplished by subsidiary functions that erase environments and erase code.

Environment erasure is relatively straightforward, and hence the formal definitions have been omitted. Briefly, the function $\texttt{erase}(\cdot)$ traverses all bindings in an environment, saves the implementation bindings and discards all others (either ghost bindings or linear bindings). An auxiliary function $\texttt{flatten}(\cdot)$ traverses all bindings in an environment, discards all non-linear bindings and uses the linear ones to build a total map (the heap) that acts as the union of all the linear ones on their respective domains. Hence, given an execution environment E for linear maps programs, the corresponding execution environment for heap-based programs is $[\texttt{heap} = \texttt{flatten}(E)], \texttt{erase}(E)$.

Figure 16 explains how to erase code. The key elements of the erasure function on code are: (1) Select and update operations on linear maps become select and update operations on the `heap` variable; (2) Linear map variable declarations, assignments between linear maps, transfer operations, and assume statements are all converted into `skip` statements; (3) Linear map and ghost procedure parameters and results disappear; and (4) Assertion statements disappear. According to soundness, verified programs never suffer from assertion failures and hence erasing assertions will not cause deviations in operational behaviour.

We lift the erasure functions on environments and statements to an erasure function on programs in a natural way. Given these functions, we are now able to prove the following key theorem. As with our other theorem, we have checked the main high-level cases by hand. These high-level cases depend upon a number of simple auxiliary lemmas that we have assumed true without detailed proof.

Theorem 3 (Erasure)
If $\vdash prog : F_2$ *then*
$prog \longrightarrow^* prog'$ *iff* $\texttt{erase}(prog) \longrightarrow^* \texttt{erase}(prog')$

$$\boxed{\texttt{erase}_\Gamma(C) = C'}$$

$$\frac{}{\texttt{erase}_\Gamma(x_1 :=^{\texttt{L}} x_2) = \texttt{skip}}$$

$$\frac{}{\texttt{erase}_\Gamma(x :=^{\texttt{G}} S) = \texttt{skip}}$$

$$\frac{\texttt{impl}(\Gamma(x_1)) \quad \texttt{impl}(\Gamma(x_3))}{\texttt{erase}_\Gamma(x_3 := g(x_1, x_2)) = x_3 := g(x_1)}$$

$$\frac{\texttt{impl}(\Gamma(x_1)) \quad \texttt{ghost}(\Gamma(x_3))}{\texttt{erase}_\Gamma(x_3 := g(x_1, x_2)) = g(x_1)}$$

$$\frac{\texttt{ghost}(\Gamma(x_1)) \quad \texttt{impl}(\Gamma(x_3))}{\texttt{erase}_\Gamma(x_3 := g(x_1, x_2)) = x_3 := g()}$$

$$\frac{\texttt{ghost}(\Gamma(x_1)) \quad \texttt{ghost}(\Gamma(x_3))}{\texttt{erase}_\Gamma(x_3 := g(x_1, x_2)) = g()}$$

$$\frac{\texttt{impl}(\tau) \quad \texttt{erase}_{\Gamma,x:\tau}(C) = C'}{\texttt{erase}_\Gamma(\texttt{var}\, x{:}\tau\, \texttt{in}\, C) = \texttt{var}\, x{:}\tau\, \texttt{in}\, C'}$$

$$\frac{\texttt{ghost}(\tau) \text{ or } \texttt{linear}(\tau) \quad \texttt{erase}_{\Gamma,x:\tau}(C) = C'}{\texttt{erase}_\Gamma(\texttt{var}\, x{:}\tau\, \texttt{in}\, C) = C'}$$

$$\frac{}{\texttt{erase}_\Gamma(\texttt{assert}\, F) = \texttt{skip}}$$

$$\frac{}{\texttt{erase}_\Gamma(x_1 :=^{\texttt{L}} x_2[Z]) = x_1 := \texttt{heap}[Z]}$$

$$\frac{}{\texttt{erase}_\Gamma(x[Z_1] :=^{\texttt{L}} Z_2) = \texttt{heap}[Z_1] := Z_2}$$

$$\frac{}{\texttt{erase}_\Gamma(x_1 := x_2@S) = \texttt{skip}}$$

$$\frac{}{\texttt{erase}_\Gamma(\texttt{assume}\, \mathrm{dom}(x_1) \cap \mathrm{dom}(x_2) = \emptyset) = \texttt{skip}}$$

Figure 16. Erasing Statements (Selected Rules)

4. Related Work

We discuss related work along four different axes—the Euclid programming language, dynamic frames, separation logic, and linear type systems.

4.1 Euclid

Euclid [17, 20] is an imperative programming language derived from Pascal. In order to manage dynamically allocated data structures, Euclid introduced the idea of a *collection*. There are only few ways to use a collection: one may allocate a new object in a collection, deallocate an object in a collection, look up an object in a collection using a pointer to it and pass a collection to a procedure. The static type of a pointer referred to the collection that contained it. Collections satisfy the disjoint domains invariant: two pointers into different collections are guaranteed to point to different objects. Euclid's creators rightly observed that this restriction would facilitate reasoning about pointers and their aliases. However, Euclid's collections are substantially more limited than linear maps as locations could not be transferred from one collection to another, and collections could not be returned from functions. Consequently, many examples presented in this paper could not be supported in Euclid. In addition, the definition of our language and program logic is presented quite differently from Euclid's; we have the benefit of 30 years of technical refinements in programming language semantics to lean on. The design of our program logic also takes recent advances in theorem proving technology into account.

In 1995, Utting [27] again struck upon the idea of a linear map, which he called a local store. Utting considers the idea in the context of a refinement calculus and points out that Euclid's collections are insufficiently flexible without the ability to transfer locations from one store to another. He gives an example of using local stores to refine a functional specification of a queue data structure into one that uses pointers. Utting does not discuss the technical details of how a Hoare proof theory should work (omitting, for instance, discussion of the frame or hypothetical frame rules and the role of assume statements in proofs, and giving only English recommendations on how to enforce anti-aliasing rules), nor does he give an operational semantics for his language, a proof of safety, or

evidence that local stores facilitate automated reasoning using theorem provers (the modern SMT solvers we use, with their extended theory of arrays [7], were not available at that time).

4.2 Dynamic Frames

In more recent years, researchers have developed a variety of powerful new verification tools, proof strategies and experimental language designs based on classical logics, SMT solvers and verification-condition generation. One such research thread is based on the use of *dynamic frames* [15]. A *frame*, also known as a *region* or *footprint*, is the set of heap locations upon which the truth of a formula depends. Intuitively, if the footprint of a formula F is disjoint from the modifies clause of a statement C, the validity of F may be preserved across execution of C. In other words, careful use of footprints gives rise to useful framing rules. Kassios [15] began this line of research by developing a sophisticated refinement calculus that uses higher-order logic together with explicit frames. Leino [18] seized upon these ideas and turned them into an effective new language for verification called Dafny. Dafny compiles to Boogie [2], which in turn generates verification conditions in first-order logic. Dafny has a set of features suitable for doing full functional-correctness verifications, and has been been used to verify a number of challenging heap-manipulating programs. Finally, Banerjee, Naumann, and Rosenberg [1] have developed Region Logic, a further extension of the idea of dynamic frames set in the context of Java. Region Logic includes a rich new form of modifies clause that captures the read, write and allocation effects of a procedure in terms of regions. Important components of Region Logic include a set of subtyping rules for region-based effects, a footprint analysis algorithm for formulae and definitions of separator formulae, which are derived from sets of effects.

Many of the ideas from dynamic frames and Region Logic clearly show up in linear maps. In particular, the domains of linear maps seem analogous to the frames themselves. Moreover, in Dafny and Region Logic, programmers explicitly manipulate frames within the code using ghost variables and assignment in a similar way to which we use transfer operations. There do appear to be at least two key differences between the systems though:
(1) Linear maps obey the disjoint domains invariant whereas dynamic frames and regions do not obey any similar "disjoint frames" invariant. Instead, programmers use logical formulae to express the relationships between various frame variables. (2) Linear maps are pairs of a domain (or frame) and a total map. One consequence of this latter fact is that every reference (select or update) to a linear map unavoidably mentions its domain/frame. The main effect of these two global design differences is that they lead to a substantial simplification of the overall verification system: effects become standard modifies clauses, the "footprint analysis" becomes routine identification of the free variables of a formula, frame and hypothetical frame rules are unchanged from the classic rules, and finally, there is no disruption to the overarching judgmental apparatus for verification-condition generation.

A variant of the dynamic frames approach is the *implicit dynamic frames* approach, which was developed by Smans, Jacobs and Piessens [24] for use in object-oriented programs and by Leino and Müller [19] for use in concurrent programs. This approach involves writing pre- and post-conditions that contain *accessor formulae* similar to those found in the capability calculus [30], alias types [25], or separation logic [13, 23]. The verification system will examine the accessor formulae and then translate them into a series of imperative statements that may be processed by an underlying classical verification-condition generator and solved by a classical SMT solver. These imperative statements perform a similar role as our transfer statements. In the case of Smans's work, they transfer access rights from the caller to the callee during function invocation, and vice-versa on return. In the case of Leino's work, they also transfer privileges to access shared memory objects when locks are acquired and released. In comparison, linear maps are a somewhat simpler, but lower-level abstraction. Consequently, formulae involving linear maps may be interpreted as ordinary first-order formulae without the need for any translation. On the other hand, programs that use linear maps are more verbose than programs that use implicit dynamic frames because of the use of explicit transfer operations. An interesting direction for future research would be to explore compilation of implicit dynamic frames into linear maps. Ideally, such a compilation strategy would be able to avoid the universally-quantified framing axioms that are used by implicit frames to relate heap states before and after function calls, as such quantified formulae are sometimes expensive for a theorem prover to discharge.

4.3 Separation Logic

Over the past decade, many researchers have devoted their attention to the development of the theory and implementation of separation logic [13, 23, 3, 10, 14], an effective framework for supporting modular reasoning in imperative programs. Separation logic has achieved its goals by introducing a new language of assertions that includes $F_1 * F_2$ and $F_1 -\!\!* F_2$. Unfortunately, this new language of assertions is not directly compatible with powerful classical theorem proving engines such as Z3 [6], which process classical formulae. A goal of our work is to give programmers access to the same kind of proof strategies that are used in separation logic, but to do so with minimal extension over a classical theorem proving and verification-condition generation environment.

Despite the differences, it is useful to try to understand the connections between linear maps and separation logic more deeply. One informal observation is that a separating conjunction of precise formulae $F_1 * F_2 * \cdots * F_k$ can be modelled in our context as an ordinary conjunction $F_1(H_1) \wedge F_2(H_2) \wedge \cdots \wedge F_k(H_k)$ where each formula F_i refers to a distinct linear map variable H_i. In separation logic, the separating conjunction ensures that the footprints of each F_i (*i.e.,* the heap locations upon which the F_i depend) are disjoint. In our case, the use of distinct program variables H_i together with the disjoint domains invariant guarantees a similar property. A second observation is that when given a separation logic formula F, one will often use the rule of consequence to prove $F_1 * F_2$ and then call a function g with precondition F_2, saving the information in F_1 across the call using Separation Logic's frame rule. In our case, a similar effect may be achieved using a transfer operation. If $F(H)$ is true initially for some linear map H, then the contents of H may be transferred to two new disjoint linear maps H_1 and H_2, which satisfy $F_1(H_1) \wedge F_2(H_2)$. Next, H_2 can be passed as a parameter to g, satisfying precondition $F_2(H_2)$, and $F_1(H_1)$ (which contains variables disjoint from the parameters of g) can be saved across the call. These observations suggest that it may be possible to compile certain precise fragments of separation logic to linear maps, which would open up new implementation opportunities for the logic using classical theorem proving tools.

Finally, Nanevski *et al.* [21] have developed powerful libraries for reasoning about separation in Coq. In this work, like in the work on linear maps, Nanevski eschews reasoning with separation-logic formulae $*$ and $-\!\!*$. Instead, he develops a theory for working directly with heaps. One of the main contributions is the development of an explicit operator for disjoint union and a demonstration that proofs using this operator can be compact. Nanevski's work is designed for interactive theorem proving in a higher-order logic like Coq, but nevertheless, some of the reasoning principles might translate to the kind of automated, first-order theorem proving environment for which linear maps were designed. This is certainly an interesting direction for future research.

4.4 Linear Type Systems

One final source of inspiration for this work comes from linear type systems [11, 28, 29]. In linear type systems, distinct variables with linear type do not alias one another. Similarly, distinct linear maps have disjoint domains. In addition, values with linear type are neither copied nor discarded (prior to being used). Similarly again, the contents of linear maps are neither copied nor discarded. [2] Hence, although we do not use linear type systems directly in this work, we use similar design principles to architect our language. The *linear* in linear maps is a reminder of these shared principles.

More recently, linear type systems have been combined with dependent types to form rich specification languages for reasoning about memory, or resources in general [30, 25, 8, 5]. The most recent of these approaches, developed by Charguéraud and Pottier, bears quite a number of similarities to work with linear maps. In particular, Charguéraud's capabilities resemble linear maps, and like in work on Euclid, or on region-based type systems [26], Charguéraud's pointer types include the type of the region or capability that they inhabit. Consequently, each pointer may inhabit only one region (*aka.,* collection or linear map), and once again, a variant of the disjoint domains invariant appears. Technically, however, there are quite a number of differences between the two systems. In particular, verification of Charguéraud's language occurs by translating imperative, capability-based programs into functional programs, which are then analyzed in detail in a theorem proving environment for functional programs such as Coq.

5. Conclusions

Linear maps are a simple data type that may be added to imperative programs to facilitate modular verification. Their primary benefits are their simplicity and their compatibility with standard first-order verification condition generators and theorem proving technology. We hope their simplicity, in particular, will make it easy for other researchers to study and build upon these new ideas.

References

[1] A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *European Conference on Object Oriented Programming*, 2008.

[2] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Fourth International Symposium on Formal Methods for Components and Objects*, 2006.

[3] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *Asian Symposium on Programming Languages and Systems*, number 3780 in Lecture Notes in Computer Science, pages 52–68, 2005.

[4] R. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.

[5] A. Charguéraud and F. Pottier. Functional translation of a calculus of capabilities. In *ACM International Conference on Functional Programming*, pages 213–224, 2008.

[6] L. de Moura and N. Bjorner. Z3: An efficient SMT solver. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[7] L. de Moura and N. Bjørner. Generalized, efficient array decision procedures. In *International Conference on Formal Methods in Computer-Aided Design*, pages 45–52, Nov. 2009.

[8] R. DeLine and M. F ahndrich. Enforcing high-level protocols in low-level software. In *International Symposium on Programming Language Design and Implementation (PLDI)*, pages 59–69, 2001.

[9] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[10] D. Distefano and M. Parkinson. jStar: Towards practical verification for Java. pages 213–226, 2008.

[11] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

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

[13] S. Ishtiaq and P. O'Hearn. Bi as an assertion language for mutable data structures. In *ACM Symposium on Principles of Programming Languages*, pages 14–26, London, United Kingdom, January 2001.

[14] B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, 2008.

[15] I. T. Kassios. Dynamic framing: Support for framing, dependencies and sharing without restriction. In *International Conference of Formal Methods Europe*, 2006.

[16] S. K. Lahiri and S. Qadeer. Back to the Future: Revisiting Precise Program Verification using SMT Solvers. In *ACM Symposium on Principles of Programming Languages*, pages 171–182, 2008.

[17] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek. Report on the programming language Euclid. *SIGPLAN Notices*, 12(2):1–79, 1977.

[18] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, 2010. To appear.

[19] K. R. M. Leino and P. M uller. A basis for verifying multi-threaded programs. In *ESOP*, 2009.

[20] R. L. London, J. V. Guttag, J. J. Horning, B. W. Lampson, J. G. Mitchell, and G. J. Popek. Proof rules for the programming language Euclid. In *Program Construction, International Summer School*, pages 133–163. Springer-Verlag, 1979.

[21] A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the verification of heap-manipulating programs. In *ACM Symposium on Principles of Programming Languages*, pages 261–273, 2009.

[22] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL*, pages 268–280, 2004.

[23] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science*, Copenhagen, Denmark, July 2002.

[24] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *European Conference Object-Oriented Programming*, pages 148–172, Berlin, Heidelberg, 2009. Springer-Verlag.

[25] F. Smith, D. Walker, and G. Morrisett. Alias types. In *ESOP*, pages 366–381, Jan. 2000.

[26] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *ACM Symposium on Principles of Programming Languages*, pages 188–201, New York, NY, USA, 1994. ACM.

[27] M. Utting. Reasoning about aliasing. In *Fourth Australasian Refinement Workshop*, pages 195–211, 1995.

[28] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*, 1990.

[29] D. Walker. *Advanced Topics in Types and Programming Languages*, chapter Substructural Type Systems, pages 3–44. MIT Press, 2005.

[30] D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, 2000.

[2] The reader may have noticed that a non-empty linear map may fall out of scope. However, when it does so, it is not removed from the environment, so operationally, the linear location is never really discarded.