Linear Types for Large-Scale Systems Verification

JIALIN LI, University of Washington, USA
ANDREA LATTUADA, ETH Zurich, Switzerland
YI ZHOU, Carnegie Mellon University, USA
JONATHAN CAMERON, Carnegie Mellon University, USA
JON HOWELL, VMware Research, USA
BRYAN PARNO, Carnegie Mellon University, USA
CHRIS HAWBLITZEL, Microsoft Research, USA

Reasoning about memory aliasing and mutation in software verification is a hard problem. This is especially true for systems using SMT-based automated theorem provers. Memory reasoning in SMT verification typically requires a nontrivial amount of manual effort to specify heap invariants, as well as extensive alias reasoning from the SMT solver. In this paper, we present a hybrid approach that combines linear types with SMT-based verification for memory reasoning. We integrate linear types into Dafny, a verification language with an SMT backend, and show that the two approaches complement each other. By separating memory reasoning from verification conditions, linear types reduce the SMT solving time. At the same time, the expressiveness of SMT queries extends the flexibility of the linear type system. In particular, it allows our linear type system to easily and correctly mix linear and nonlinear data in novel ways, encapsulating linear data inside nonlinear data and vice-versa. We formalize the core of our extensions, prove soundness, and provide algorithms for linear type checking. We evaluate our approach by converting the implementation of a verified storage system (~24K lines of code and proof) written in Dafny, to use our extended Dafny. The resulting system uses linear types for 91% of the code and SMT-based heap reasoning for the remaining 9%. We show that the converted system has 28% fewer lines of proofs and 30% shorter verification time overall. We discuss the development overhead in the original system due to SMT-based heap reasoning and highlight the improved developer experience when using linear types.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; **Formal software verification**.

Additional Key Words and Phrases: linear types, systems verification

ACM Reference Format:
Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2022. Linear Types for Large-Scale Systems Verification. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 69 (April 2022), 28 pages. https://doi.org/10.1145/3527313

69

1 INTRODUCTION

Formal verification allows developers to prove strong functional correctness guarantees about complex systems software. This can significantly increase software reliability, minimizing the risk of runtime errors that can lead to data loss and potentially disastrous consequences. Although verifying large systems is a notoriously time-consuming task [Klein et al. 2014], recent SMT-based

Authors' addresses: Jialin Li, University of Washington, USA; Andrea Lattuada, ETH Zurich, Switzerland; Yi Zhou, Carnegie Mellon University, USA; Jonathan Cameron, Carnegie Mellon University, USA; Jon Howell, VMware Research, USA; Bryan Parno, Carnegie Mellon University, USA; Chris Hawblitzel, Microsoft Research, USA.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2022 Copyright held by the owner/author(s).
2475-1421/2022/4-ART69
https://doi.org/10.1145/3527313

verified systems have achieved reasonable run-time performance with a degree of effort feasible for high-value systems [Hance et al. 2020a; Hawblitzel et al. 2015a].

Unfortunately, existing SMT-based verifiers interact poorly with reasoning about mutable memory in large-scale systems. Interactive proof assistants often use separation logic [Reynolds 2002] to reason about memory and aliasing, but SMT solvers do not provide good support for separation logic. As a result, typical SMT-based program verifiers like Dafny [Leino 2010] dispatch memory reasoning to the SMT solver, forcing the SMT solver to reason about the disjointness of sets of memory locations and updates to a global heap. This approach requires the developers to establish and maintain memory invariants over all mutable objects, a burdensome task. Furthermore, overloading the SMT solver with memory reasoning leads to poor SMT performance (and hence poor interactivity) and confusing error messages when a proof fails.

Linear type systems [Wadler 1990] are gaining traction in practice in Rust [Klabnik et al. 2018; Matsakis and Klock 2014] as a mechanism for controlling and reasoning about aliasing. A growing body of large, performant systems built in Rust [Bhardwaj et al. 2021; Boos et al. 2020; Narayanan et al. 2020] provide evidence that linear types are practical and effective for ensuring program safety. We argue that linear types are effective for SMT-style correctness verification as well.

In our work, we show how to integrate linear types with SMT-based verification. Our approach demonstrates that linear types and SMT solving are complementary: linear types improve SMT performance in the common case where objects are not aliased, and SMT solving enables more expressive verification when aliasing is necessary. When objects are not aliased, the SMT solver can view mutable linear datatypes as immutable mathematical datatypes, speeding solver times. When aliasing is necessary, the SMT solver can precisely reason about the aliasing.

Our approach allows developers to freely mix linear and aliased styles, storing linear data inside nonlinear data and vice-versa. Programs can encapsulate nonlinear, aliased data structures inside linear data structures, using a region-based approach to modularly hide the aliased objects behind a purely linear interface. In the other direction, programs can place linear data inside nonlinear data, using the SMT solver to verify the correct usage of the linear values, avoiding the run-time safety checks needed by other linearly typed languages like Rust. Our approach also supports Rust-style lightweight borrowing of immutable references from linear data, even when the linear data is stored inside aliased, nonlinear objects. We formalize the core of our approach, prove soundness, and prove that the checking of types, linearity, and borrowing is decidable with a straightforward algorithm (Section 2).

The motivation to integrate linear memory reasoning into SMT-based verification is to reduce the cost of practical, large-scale verified development. To evaluate this goal, we introduce the linear type system into Dafny [Leino 2010], a verification language with an SMT backend, which has been used to verify a variety of high performance systems [Hance et al. 2020a; Hawblitzel et al. 2015a, 2014]. Then we took one such system, VeriBetrKV[1] [Hance et al. 2020a], a practical verified storage system (~24K lines of code and proof in the implementation layer), and converted it from vanilla Dafny into our new Linear Dafny. Because Linear Dafny supports hybrid memory reasoning, we could convert VeriBetrKV incrementally over several months, with the system remaining verified at each step throughout the process. The final result is still a hybrid, with 9% using Dafny's vanilla memory reasoning for tricky object relations, and the other 91% exploiting linear memory reasoning.

We find that the linearized VeriBetrKV has 28% fewer lines of proofs, 30% shorter verification time overall, and among slow methods, the median method's interactive verification time is cut nearly in half. We observe that debugging memory errors in linearized code is much simpler, with the linear type system identifying errors at precise lines of code with actionable error messages.

[1] Developed, in part, by a subset of the authors of this work.

The comparison indicates that reasoning about aliasing and mutation via linear types improves the developer experience and reduces development cost over an SMT-only approach. We show that by using better memory reasoning approaches, we can lower verification effort and move a step towards making full verification a realistic engineering choice for a larger set of system software.

Our main contributions are (1) The presentation of a practical design for combining SMT-based heap verification with linear types, as well as enhancements to an existing implementation thereof. (2) The first formalization combining verification condition generation, linear types, and borrowing. (3) A novel region-based mechanism to encapsulate nonlinear data inside linear data and vice-versa. (4) A direct comparison of how SMT-based memory reasoning and the linear-SMT hybrid approach impact the developer experience in the context of a large verified system[2].

The rest of this paper is organized as follows: Section 2 introduces Linear Dafny and presents a detailed formalization; Section 3 compares and contrasts the development experiences of VeriBetrKV and the linearized VeriBetrKV; Section 4 discusses related work on linear type systems.

2 DESIGN AND FORMALIZATION OF LINEAR DAFNY

In this section, we present our design and formalization of the Linear Dafny language[3]. We originally built Linear Dafny in support of the implementation of VeriBetrKV [Hance et al. 2020a], and its basic features and usage are briefly mentioned in Section 5.1 of [Hance et al. 2020a]. Here we present the complete design, including the underlying type-checking algorithm, support for algebraic datatypes, and enhancements to the implementation. We strengthen the previous Linear Dafny implementation with support for `inout` parameters and a library for region-based interoperation between linear and nonlinear data.

We first introduce the basic ideas of adding linearity in Dafny, following earlier work by Wadler [Wadler 1990], on the Cogent language [Amani et al. 2016], and on Rust [Klabnik et al. 2018; Matsakis and Klock 2014]. We then focus on a new region-based formalism that improves the interaction of linear data and nonlinear data in Linear Dafny. Finally, we prove the soundness of the formalized system and prove the soundness and completeness of an algorithm for checking types, linearity, and borrowing in the formalized system.

Section 3 then presents a large-scale evaluation, demonstrating the application of Linear Dafny to 91% of the VeriBetrKV implementation, compared to earlier preliminary experiments converting two “leaf” modules constituting 16% of the implementation [Hance et al. 2020a].

2.1 Basic Features of Linear Dafny

Dafny is a programming language and a verifier with both imperative programming and verification support [Leino 2010]. Dafny supports generic classes and dynamic allocation for writing imperative programs, and provides built-in specification constructs including preconditions and postconditions, mathematical functions, and ghost variables for writing formal specifications. In Dafny, developers write down specifications, proofs and methods, and then ask the verifier to check if methods meet their specifications. Behind the scenes, Dafny translates the program into an intermediate verification language Boogie [Barnett et al. 2006], which then generates verification conditions for the Z3 SMT (Satisfiability Modulo Theories) solver to resolve [de Moura and Bjørner 2008].

Linear Dafny extends the standard Dafny language with a linear type system. Linear type systems disallow the duplication and discarding of linear variables, which helps to control aliasing: by disallowing duplication, the type system can guarantee that a variable pointing to a memory cell is the only variable pointing to that memory cell. In standard Dafny, variables can be ordinary

[2] Available at https://github.com/jialin-li/linear-veribetrkv-artifact

[3] Available at https://github.com/secure-foundations/dafny/tree/oopsla2022

```
method M1(s1:seq<int>, s2:seq<int>)
  returns(x:int)
  requires |s1| >= 10 && |s2| >= 10
{
  x := s1[5] + s2[5];
}

method M2(s1:seq<int>, s2:seq<int>)
  returns(s3:seq<int>, s4:seq<int>, x:int)
  requires |s1| >= 10 && |s2| >= 10
{
  x := M1(s1, s1);
  s3 := s1[5 := 100]; // copy entire seq
  s4 := s2[5 := 200]; // copy entire seq
  assert s3[5] == 100;
}

method M3(a1:array<int>, a2:array<int>)
  requires a1.Length >= 10
  requires a2.Length >= 10
  requires a1 != a2
  modifies a1, a2;
{
  a1[5] := 100;
  a2[5] := 200;
  assert a1[5] == 100; // needs a1 != a2
}
```

```
method M4(shared s1:seq<int>,
          shared s2:seq<int>)
  returns(x:int)
{
  x := seq_get(s1, 5) + seq_get(s2, 5);
}

method M5(linear l1:seq<int>,
          linear l2:seq<int>)
  returns(linear l3:seq<int>,
          linear l4:seq<int>,
          x:int)
  requires |l1| >= 10 && |l2| >= 10
{
  // temporarily borrow l1 as shared:
  x := M4(l1, l1);
  // l1 is linear again here:
  l3 := seq_set(l1, 5, 100); // in place
  l4 := seq_set(l2, 5, 200); // in place
  assert l3[5] == 100;
}
```

Fig. 1. Example code in standard Dafny (left) and Linear Dafny (right)

(unannotated) or `ghost`. We refer to "ordinary" and `ghost` as two distinct *usages* for variables. Linear Dafny extends Dafny's type system with two additional usages: `linear` for linear variables and `shared` for immutably borrowed variables. Ordinary and ghost variables can be freely duplicated, discarded, stored in datatypes, and passed in and out of functions and methods. Ordinary, linear, and shared variables are compiled to executable code, while ghost variables are erased before compilation. Linear variables can be neither duplicated nor discarded, but can be stored in linear datatypes and passed in and out of functions and methods freely. Shared variables can be duplicated and discarded, but have restricted scope. Shared variables cannot be stored in data structures, but data structures containing linear data can be borrowed so that their fields appear shared.

The left half of Figure 1 shows three standard Dafny methods, `M1`, `M2`, and `M3`, that manipulate arrays, where the `seq` is Dafny's sequence (immutable array) type and `array` is Dafny's mutable array type. Mutable arrays can be modified in place (`M3`), but require reasoning about aliasing (`a1 != a2`) so that the SMT solver can verify assertions like `a1[5] == 100`. This reasoning is done through Dafny's dynamic frames [Kassios 2006], in which each method that modifies the heap must provide a `modifies` clause that states the set of heap locations the method changes. Although the specification of non-aliasing in this example is simple (`a1 != a2`), more complex programs require increasingly elaborate specifications of disjointness for the programmer to write and the SMT solver to reason about.

In contrast to Dafny arrays, which are heap objects, Dafny sequences are values. Sequences are easier to reason about than arrays, but do not support in-place updates, which means that updates to sequences require copying the whole sequence at run-time (M2).

As shown in the right half of Figure 1, Linear Dafny supports linear datatypes such as linear sequences. Linear datatypes provide both value semantics and in-place updates, achieving the best of both worlds: in M5, linearity ensures that sequences l1 and l2 do not alias and linearity allows in-place updates to the linear sequences without copying the sequence.

In addition, Linear Dafny allows temporary borrowing of linear variables in the style of Wadler's "let!" [Wadler 1990] and Rust's immutable borrowing. When calling M4, the method M5 temporarily demotes l1 from linear usage to shared usage. While l1 is shared, it can be duplicated, allowing M5 to pass l1 as an argument twice to M4. Like Rust, and unlike Wadler's "let!" [Wadler 1990] and Cogent [Amani et al. 2016], Linear Dafny automatically infers where borrowing occurs, so no explicit programmer annotation is needed to share l1 in the call to M4. Section 2.6 describes how Linear Dafny performs this inference.

For soundness's sake, Linear Dafny must ensure that borrowing is only temporary, so that no copies of a shared reference survive after a variable becomes linear again. Otherwise, a value could be viewed as both linear and shared simultaneously, allowing a program to deallocate the value through the linear usage while still reading the value through the shared usage. Like Cogent, Linear Dafny ensures this by disallowing any shared references from being returned out of the scope of a borrow (see Section 2.2 for the precise rules). As long as they don't escape past a borrow, though, shared references can be freely passed in and out of expressions and functions. For instance, like Cogent, Linear Dafny allows functions to return shared references:

```
method M(b:bool, shared s1:seq<int>, shared s2:seq<int>) returns(shared ret:seq<int>) {
  if b {
    ret := s1;
  } else {
    ret := s2;
  }
}
```

The method M5 consumes its original linear parameters l1 and l2 and produces new linear values l3 and l4 (which are actually the original sequences l1 and l2, updated in place). Rather than forcing programmers to manually pass all linear values in and out of methods, Linear Dafny also supports inout method parameters, so that M5 can be more concisely written as:

```
method M5(linear inout l1:seq<int>, linear inout l2:seq<int>) returns(x:int)
  requires |l1| >= 10 && |l2| >= 10
{
  x := M4(l1, l1);
  l1 := seq_set(l1, 5, 100);
  l2 := seq_set(l2, 5, 200);
  assert l1[5] == 100;
}
```

Arguments to inout parameters can be fields of linear datatypes stored in linear variables or referenced by the inout parameters of the enclosing function. This allows mutably borrowing datatype fields so that programs can efficiently update those fields in place.

Finally, Linear Dafny supports linear algebraic datatypes, which may contain ordinary fields and linear fields. In the List datatype below, the data field is ordinary and the tail field is linear.

When a linear datatype is borrowed as shared, all of its linear fields appear as shared. This allows the while loop shown above to traverse the list without consuming it.

```
linear datatype List<A> = Nil | Cons(data:A, linear tail:List<A>)

// ghost function definining length of List:
function Length<A>(ghost list:List<A>):int {
  match list {
    case Nil => 0
    case Cons(data, tail) => 1 + Length(tail)
  }
}

method GetLength<A>(shared list:List<A>) returns(n:int)
  ensures n == Length(list)
{
  n := 0;
  shared var iter := list;
  while !iter.Nil?
    invariant n + Length(iter) == Length(list)
    decreases iter
  {
    n := n + 1;
    iter := iter.tail;
  }
}
```

In addition to value-based types (sequences and algebraic datatypes), Dafny supports reference-based heap objects (class objects and mutable arrays). Currently, Linear Dafny always treats reference-based heap objects as ordinary rather than shared or linear. It would be straightforward to also allow linear reference-based heap objects, but we didn't have a compelling reason to do so, since linearity already enables mutation of sequences and algebraic datatypes, making linear reference-based heap objects somewhat redundant. Instead, Linear Dafny uses reference-based heap objects for nonlinear data structures, and focuses on enabling interoperation between nonlinear reference-based heap objects and linear value-based types.

2.2 Formalization

This section introduces the syntax and typing rules for a simple model of Linear Dafny. The goal is not to capture all the features of Linear Dafny, but instead to illustrate and clarify the key ideas in Linear Dafny and prove their soundness. We then use the formal model to extend and improve Linear Dafny by breaking Linear Dafny's monolithic heap into regions. This region-based approach yields three novel improvements, all illustrated in our formal model:

(1) Regions allow proper encapsulation of nonlinear data structures stored inside linear data structures, so that any modifications to nonlinear data stay private and are not leaked via Dafny's `modifies` clauses.
(2) Regions enable an elegant form of borrowing for linear data stored inside nonlinear data, replacing the clumsy attribute-based mechanism described in [Hance et al. 2020a].

variable	x		
integer	i	::=	$\ldots, -2, -1, 0, 1, 2, \ldots$
location	ℓ		
struct name	S		
struct instance	s	::=	$\mathrm{S}(v_1, ..., v_n)$
region name	r		
region data	d	::=	$\{\ell_1 \mapsto s_1, \ldots, \ell_n \mapsto s_n\}$
region value	$\mathrm{r}\, d$		
usage	u	::=	linear $\mid$ shared $\mid$ ordinary
	u^{ls}	::=	linear $\mid$ shared
	u^{lo}	::=	linear $\mid$ ordinary
type	τ	::=	int $\mid$ ref(S) $\mid$ region $\mid$ $\langle u^{\mathrm{lo}}_1\, \tau_1, \ldots, u^{\mathrm{lo}}_n\, \tau_n\rangle$
value	v	::=	$i \mid \ell \mid$ null $\mid \mathrm{r}\, d \mid \langle v_1, \ldots, v_n\rangle$
expression	e	::=	$v \mid x \mid e_1 + e_2 \mid e_1;\, e_2 \mid \mathrm{let}\, u\, x = e_1\, \mathrm{in}\, e_2$
		$\mid$	$\langle e_1, \ldots, e_n\rangle \mid e.i \mid \mathrm{let}\, \langle x_1, \ldots, x_n\rangle = e_1\, \mathrm{in}\, e_2$
		$\mid$	new_region() $\mid$ free_region(e) $\mid$ alloc $\mathrm{S}(e_1, \ldots, e_n)$ @ e_0
		$\mid$	read($e_1.i$) @ e_0 $\mid$ write($e_1.i := e_2$) @ e_0 $\mid$ swap($e_1.i := e_2$) @ e_0
location typing	$\mathbb{L}$	::=	$\{\ell_1 \mapsto \mathrm{S}_1, \ldots, \ell_n \mapsto \mathrm{S}_n\}$
region typing	$\mathbb{R}$	::=	$\{\mathrm{r}_1 \mapsto u^{\mathrm{ls}}_1, \ldots, \mathrm{r}_n \mapsto u^{\mathrm{ls}}_n\}$
variable typing	$\mathbb{X}$	::=	$\{x_1 \mapsto u_1\, \tau_1, \ldots, x_n \mapsto u_n\, \tau_n\}$
combined typing	$\mathbb{C}$	::=	$\mathbb{L};\, \mathbb{R};\, \mathbb{X}$

Fig. 2. Syntax of Formal Model

(3) Our regions take advantage of SMT solving to avoid the need for region variables, universal region quantification, and existential region quantification.

Figure 2 shows the syntax of the formal model. Usages u include linear, shared, and ordinary (we omit Linear Dafny's `ghost` usage for simplicity). Types τ include integers, references, regions, and linear tuples. We start by describing the basic features for integers and linear tuples, leaving regions and references for 2.4.

Expressions e for integers include integer constants i and integer addition $e_1 + e_2$. Linear tuple types $\langle u^{\mathrm{lo}}_1\, \tau_1, \ldots, u^{\mathrm{lo}}_n\, \tau_n\rangle$ describe the usage u_i and type τ_i of each field i in the tuple. Fields of linear tuple types cannot have usage shared, since this would allow hiding shared data inside linear data, and thereby escaping the scoping restrictions on shared data. (We use the notation u^{lo} to indicate a usage that can only be linear or ordinary.) Expressions for linear tuples include:

- tuple construction $\langle e_1, \ldots, e_n\rangle$, which creates a tuple of type $\langle u_1\, \tau_1, \ldots, u_n\, \tau_n\rangle$.
- tuple selection e.i, which selects field i from a tuple of type $\langle u_1\, \tau_1, \ldots, u_n\, \tau_n\rangle$.
- tuple deconstruction let $\langle x_1, \ldots, x_n\rangle = e_1$ in e_2, which deallocates a linear tuple e_1 and places its fields in variables $x_1 \ldots x_n$.

Here is an example program that allocates two tuples x and y, placing x inside y, and then deconstructs y and x:

$$
\begin{array}{l}
\mathrm{let\ ordinary}\ a = 10\ \mathrm{in} \\
\mathrm{let\ linear}\ x = \langle\rangle\ \mathrm{in} \\
\mathrm{let\ linear}\ y = \langle x, a\rangle\ \mathrm{in} \\
\mathrm{let\ ordinary}\ b = y.2 + y.2\ \mathrm{in} \\
\mathrm{let}\ \langle x', a'\rangle = y\ \mathrm{in} \\
\mathrm{let}\ \langle\rangle = x'\ \mathrm{in} \\
a + a' + b
\end{array}
$$

The type checking rules use an environment $\mathbb{X} = \{x_1 \mapsto u_1\ \tau_1, \ldots, x_n \mapsto u_n\ \tau_n\}$ that maps variables to their types and usages. The notation $\mathbb{X} = \mathbb{X}_1 \# \mathbb{X}_2$ means that $\mathbb{X}$ is split into two parts, $\mathbb{X}_1$ and $\mathbb{X}_2$, which share the same nonlinear mappings but contain disjoint linear mappings. With this notation, we can write simple type checking rules for integer addition and linear tuple construction (considering, for moment, only 2-tuples for simplicity):

$$\frac{\mathbb{X}_1 \vdash e_1 : \text{ordinary int} \qquad \mathbb{X}_2 \vdash e_2 : \text{ordinary int}}{\mathbb{X}_1 \# \mathbb{X}_2 \vdash e_1 + e_2 : \text{ordinary int}} \qquad \frac{\mathbb{X}_1 \vdash e_1 : u_1\ \tau_1 \qquad \mathbb{X}_2 \vdash e_2 : u_2\ \tau_2}{\mathbb{X}_1 \# \mathbb{X}_2 \vdash \langle e_1, e_2 \rangle : \text{linear}\ \langle u_1\ \tau_1, u_2\ \tau_2 \rangle}$$

When type checking $\langle x, a \rangle$ in the example above, both $\mathbb{X}_1$ and $\mathbb{X}_2$ will contain the nonlinear mapping $a \mapsto$ ordinary int, but only $\mathbb{X}_1$ will contain the linear mapping $x \mapsto$ linear $\langle\rangle$. As is standard in formal presentations of linear type systems [Wadler 1990], the typing rule does not directly determine how to split $\mathbb{X}$ into $\mathbb{X}_1$ and $\mathbb{X}_2$. Section 2.6 presents a simple algorithm for deciding this split.

Formal presentations of borrowing are less common and generally either rely on Wadler's explicit let! notation [Amani et al. 2016; Wadler 1990] or are based on Rust's more complex rules, as in RustBelt [Jung et al. 2017] and Oxide [Weiss et al. 2019]. Here, we introduce rules for borrowing that are simpler than Rust's rules and do not rely on let!. We illustrate this approach with the rule for sequencing expressions $e_1;\ e_2$:

$$\frac{u_1 \neq \text{linear} \qquad \mathbb{X}_1, \text{shared}(\mathbb{X}_b) \vdash e_1 : u_1\ \tau_1 \qquad \mathbb{X}_2, \text{linear}(\mathbb{X}_b) \vdash e_2 : u_2\ \tau_2}{(\mathbb{X}_1 \# \mathbb{X}_2), \text{linear}(\mathbb{X}_b) \vdash e_1;\ e_2 : u_2\ \tau_2}$$

The notation $\mathbb{X} = \mathbb{X}_1, \mathbb{X}_2$ means that $\mathbb{X}$ is split into two parts, $\mathbb{X}_1$ and $\mathbb{X}_2$, which contain disjoint mappings, both for linear and nonlinear mappings. The notation linear$(\mathbb{X})$ denotes an environment in which all usages are linear, while the notation shared$(\mathbb{X})$ denotes the same environment as linear$(\mathbb{X})$, except that all usages are shared. In the rule above, this has the effect of splitting out zero or more linear mappings linear$(\mathbb{X}_b)$ from the overall environment, and then viewing them as shared mappings when checking expression e_1 – in other words, borrowing zero or more linear mappings as shared within e_1. As with the split between $\mathbb{X}_1$ and $\mathbb{X}_2$, the typing rule does not directly determine which $\mathbb{X}_b$ to split out. Section 2.6's algorithm also decides this split.

To help understand the rule above, consider a slight variation of the earlier example, where we use ";" to discard the result of $y.2 + y.2;$ rather than putting the result in a variable b:

$$\begin{array}{l} \ldots \\ \text{let linear}\ y = \langle x, a \rangle\ \text{in} \\ y.2 + y.2; \\ \text{let}\ \langle x', a' \rangle = y\ \text{in} \ldots \end{array}$$

When we apply the typing rule for $e_1;\ e_2$, we use $e_1 = y.2 + y.2$ and $e_2 = \text{let}\ \langle x', a' \rangle = y\ \text{in} \ldots$ and linear$(\mathbb{X}_b) = \{y \mapsto \text{linear}\ \tau_y\}$ and shared$(\mathbb{X}_b) = \{y \mapsto \text{shared}\ \tau_y\}$, where we define $\tau_y = \langle \text{linear}\ \langle\rangle, \text{ordinary int} \rangle$. Because y is shared in shared$(\mathbb{X}_b)$ rather than linear, it may be duplicated in the expression $y.2 + y.2$. After the borrowing finishes, y reverts to its original linear usage in the remaining expression e_2. This is safe because e_1 completely evaluates to a value, which is then discarded, before e_2 begins evaluation, so the program's execution never observes y as both shared and linear simultaneously.

The same reasoning allows for borrowing in let expressions:

$$\frac{u_1 = \text{shared} \Rightarrow \mathbb{X}_b = \varnothing \qquad \mathbb{X}_1, \text{shared}(\mathbb{X}_b) \vdash e_1 : u_1\ \tau_1 \qquad \mathbb{X}_2, \text{linear}(\mathbb{X}_b),\ x \mapsto u_1\ \tau_1 \vdash e_2 : u_2\ \tau_2}{(\mathbb{X}_1 \# \mathbb{X}_2), \text{linear}(\mathbb{X}_b) \vdash \text{let}\ u_1\ x = e_1\ \text{in}\ e_2 : u_2\ \tau_2}$$

Here, the result of evaluating e_1 is not discarded, but the rule prohibits returning a shared result from e_1 if any borrowing occurs. This restriction, which is also made by Cogent's let! expression [Amani et al. 2016], prevents borrowed values from leaking back out through the bound variable x, so that values borrowed by e_1 can flow into e_1 but not out. More generally, the places where borrowing occurs are barriers that block all borrowed variables from flowing out.

As with Wadler's original let! expression, the rule above requires that e_2 consume any borrowed linear variables. At first, this may appear to disallow expressions like "let ordinary $z = y.2 + y.2$ in $z + 1$" that use a borrowed y without consuming y. However, in such cases, the borrowing simply happens in a larger scope. For instance, the borrowing of y occurs in the let for b rather than the let for z in the following example:

$$\begin{array}{l} \ldots \\ \text{let ordinary } b = (\text{let ordinary } z = y.2 + y.2 \text{ in } z + 1) \text{ in} \\ \text{let } \langle x', a' \rangle = y \text{ in} \ldots \end{array}$$

Once an expression has borrowed a linear tuple as a shared tuple, it can select fields from the tuple:

$$\frac{\mathbb{X} \vdash e : \text{shared } \langle u_1\ \tau_1, \ldots, \text{linear } \tau_i, \ldots, u_n\ \tau_n \rangle}{\mathbb{X} \vdash e.i : \text{shared } \tau_i} \qquad \frac{\mathbb{X} \vdash e : \text{shared } \langle u_1\ \tau_1, \ldots, \text{ordinary } \tau_i, \ldots, u_n\ \tau_n \rangle}{\mathbb{X} \vdash e.i : \text{ordinary } \tau_i}$$

Note that the result of selecting a linear field from a shared tuple is itself shared.

Figure 3 shows the complete type checking rules, including rules for regions that will be described in Section 2.4. The rules include the environment $\mathbb{X}$ that maps variables to their types and usages, as well as environments $\mathbb{L}$ for references to locations and $\mathbb{R}$ for regions. The notation $\mathbb{C} = \mathbb{L};\ \mathbb{R};\ \mathbb{X}$ represents the combined typing environment, $!\mathbb{C}$ selects the nonlinear mappings from $\mathbb{C}$, and $¡\mathbb{C}$ selects the linear mappings from $\mathbb{C}$. Borrowing is allowed in three different rules: sequencing $e_1;\ e_2$, let expressions, and tuple deconstruction. This highlights the fact that in Linear Dafny, as in Rust, borrowing is ubiquitous, rather than being restricted to a special expression like let!.

2.3 SMT Solving and Weakest Preconditions

Linear Dafny programs are checked with both a type checker and an SMT solver. To help demonstrate the balance and interplay between type checking and SMT solving, we include a simple SMT checking process in our formal model. In particular, we show:

(1) Since the type system handles linearity checking, the SMT solver can view linear datatypes as simple mathematical datatypes, without worrying about the linearity.
(2) The SMT solver can statically check the correct usage of linear values stored inside nonlinear objects, which Rust needs run-time checks for (Section 2.5).
(3) The SMT solver can track the relation between regions and references into regions, without needing region variables and quantification over region variables (Section 2.4).

Linear Dafny uses Dafny's built-in verification condition generator to verify Linear Dafny code. To model Dafny's verification condition generator, Figure 5 defines a verification condition generator $\text{wp}(e, x.\ f)$ that computes a weakest precondition for any expression e and postcondition f. The postcondition may use the variable x to refer to the final value computed by e. To prove that e evaluates to 4, for instance, we can ask an SMT solver to prove the validity of the formula $\text{wp}(e, x.\ x = 4)$. The definition of $\text{wp}(e, x.\ f)$ shows that $\text{wp}(\text{let ordinary } z = 2 \text{ in } z + 2, x.\ x = 4)$ is equal to $2 + 2 = 4$, which is an easy formula for an SMT solver to prove. For convenience, some of the definitions in Figure 5 use $\forall$ quantifiers to introduce temporary variables; an SMT solver can easily eliminate these using Skolemization since they appear only in positive positions.

Note that $\text{wp}(e, x.\ f)$ ignores linearity completely. Operations on linear tuples, for example, are translated directly into operations on SMT tuples, with no concerns about linearity or borrowing.

Well-typed expression $\mathbb{C} \vdash e : u\ \tau$

$$!\mathbb{C},\ x \mapsto u\,\tau \vdash x : u\ \tau \qquad !\mathbb{C} \vdash i : \text{ordinary int} \qquad !\mathbb{C},\ \ell \mapsto \mathrm{S} \vdash \ell : \text{ordinary ref}(\mathrm{S})$$

$$!\mathbb{C} \vdash \text{null} : \text{ordinary ref}(\mathrm{S})$$

$$\frac{u \neq \text{ordinary} \qquad \text{domain}(d) = \{\ell \in \text{domain}(\mathbb{L}) \mid \text{RegionOfLoc}(\ell) = \mathrm{r}\} \qquad \mathbb{L};\ \mathbb{R} \vdash d : u}{\mathbb{L};\ \mathbb{R} \# \{\mathrm{r} \mapsto u\};\ !\mathbb{X} \vdash \mathrm{r}\,d : u \text{ region}}$$

$$\frac{u_1 \neq \text{linear} \qquad \mathbb{C}_1, \text{shared}(\mathbb{C}_b) \vdash e_1 : u_1\ \tau_1 \qquad \mathbb{C}_2, \text{linear}(\mathbb{C}_b) \vdash e_2 : u_2\ \tau_2}{(\mathbb{C}_1 \# \mathbb{C}_2), \text{linear}(\mathbb{C}_b) \vdash e_1;\ e_2 : u_2\ \tau_2}$$

$$\frac{u_1 = \text{shared} \Rightarrow \mathbb{C}_b = \varnothing;\ \varnothing;\ \varnothing \qquad \mathbb{C}_1, \text{shared}(\mathbb{C}_b) \vdash e_1 : u_1\ \tau_1 \qquad \mathbb{C}_2, \text{linear}(\mathbb{C}_b),\ x \mapsto u_1\,\tau_1 \vdash e_2 : u_2\ \tau_2}{(\mathbb{C}_1 \# \mathbb{C}_2), \text{linear}(\mathbb{C}_b) \vdash \text{let } u_1\ x = e_1 \text{ in } e_2 : u_2\ \tau_2}$$

$$\frac{u \neq \text{ordinary} \qquad \mathbb{C}_1 \vdash e_1 : \text{share_as}(u, u_1)\ \tau_1 \qquad \ldots \qquad \mathbb{C}_n \vdash e_n : \text{share_as}(u, u_n)\ \tau_n}{\mathbb{C}_1 \# \ldots \# \mathbb{C}_n \vdash \langle e_1, \ldots, e_n \rangle : u\ \langle u_1\,\tau_1, \ldots, u_n\,\tau_n \rangle}$$

$$\frac{\mathbb{C}_1 \vdash e_1 : \text{ordinary int} \qquad \mathbb{C}_2 \vdash e_2 : \text{ordinary int}}{\mathbb{C}_1 \# \mathbb{C}_2 \vdash e_1 + e_2 : \text{ordinary int}} \qquad \frac{\mathbb{C} \vdash e : \text{shared } \langle u_1\,\tau_1, \ldots, u_i\,\tau_i, \ldots, u_n\,\tau_n \rangle}{\mathbb{C} \vdash e.i : \text{share_as}(\text{shared}, u_i)\ \tau_i}$$

$$\frac{\mathbb{C}_0, \text{shared}(\mathbb{C}_b) \vdash e_0 : \text{linear } \langle u_1\,\tau_1, \ldots, u_n\,\tau_n \rangle \qquad \mathbb{C}_x, \text{linear}(\mathbb{C}_b),\ x_1 \mapsto u_1\,\tau_1, \ldots,\ x_n \mapsto u_n\,\tau_n \vdash e_x : u_x\ \tau_x}{(\mathbb{C}_0 \# \mathbb{C}_x), \text{linear}(\mathbb{C}_b) \vdash \text{let } \langle x_1, \ldots, x_n \rangle = e_0 \text{ in } e_x : u_x\ \tau_x}$$

$$!\mathbb{C} \vdash \text{new_region}() : \text{linear region} \qquad \frac{\mathbb{C} \vdash e : \text{linear region}}{\mathbb{C} \vdash \text{free_region}(e) : \text{ordinary int}}$$

$$\frac{\begin{array}{c}\text{StructType}(\mathrm{S}) = (u_1\,\tau_1, \ldots, u_n\,\tau_n) \\ \mathbb{C}_0 \vdash e_0 : \text{linear region} \qquad \mathbb{C}_1 \vdash e_1 : u_1\ \tau_1 \qquad \ldots \qquad \mathbb{C}_n \vdash e_n : u_n\ \tau_n\end{array}}{\mathbb{C}_0 \# \mathbb{C}_1 \# \ldots \# \mathbb{C}_n \vdash \text{alloc } \mathrm{S}(e_1, ..., e_n)\ @\ e_0 : \text{linear } \langle \text{ordinary ref}(\mathrm{S}), \text{linear region} \rangle}$$

$$\frac{\text{StructType}(\mathrm{S}) = (u_1\,\tau_1, \ldots, u_i\,\tau_i, \ldots, u_n\,\tau_n) \qquad \mathbb{C}_0 \vdash e_0 : \text{shared region} \qquad \mathbb{C}_1 \vdash e_1 : \text{ordinary ref}(\mathrm{S})}{\mathbb{C}_0 \# \mathbb{C}_1 \vdash \text{read}(e_1.i)\ @\ e_0 : \text{share_as}(\text{shared}, u_i)\ \tau_i}$$

$$\frac{\begin{array}{c}\text{StructType}(\mathrm{S}) = (u_1\,\tau_1, \ldots, \text{ordinary}\,\tau_i, \ldots, u_n\,\tau_n) \\ \mathbb{C}_0 \vdash e_0 : \text{linear region} \qquad \mathbb{C}_1 \vdash e_1 : \text{ordinary ref}(\mathrm{S}) \qquad \mathbb{C}_2 \vdash e_2 : \text{ordinary } \tau_i\end{array}}{\mathbb{C}_0 \# \mathbb{C}_1 \# \mathbb{C}_2 \vdash \text{write}(e_1.i := e_2)\ @\ e_0 : \text{linear region}}$$

$$\frac{\begin{array}{c}\text{StructType}(\mathrm{S}) = (u_1\,\tau_1, \ldots, \text{linear}\,\tau_i, \ldots, u_n\,\tau_n) \\ \mathbb{C}_0 \vdash e_0 : \text{linear region} \qquad \mathbb{C}_1 \vdash e_1 : \text{ordinary ref}(\mathrm{S}) \qquad \mathbb{C}_2 \vdash e_2 : \text{linear } \tau_i\end{array}}{\mathbb{C}_0 \# \mathbb{C}_1 \# \mathbb{C}_2 \vdash \text{swap}(e_1.i := e_2)\ @\ e_0 : \text{linear } \langle \text{linear } \tau_i, \text{linear region} \rangle}$$

Well-typed struct instance $\mathbb{C} \vdash s : \mathrm{S}$ containing u^{ls} **and region data**: $\mathbb{L};\ \mathbb{R} \vdash d : u^{\mathrm{ls}}$

$$\frac{\begin{array}{c}\text{StructType}(\mathrm{S}) = (u_1\,\tau_1, \ldots, u_n\,\tau_n) \\ \mathbb{C}_1 \vdash v_1 : \text{share_as}(u^{\mathrm{ls}}, u_1)\ \tau_1 \qquad \ldots \qquad \mathbb{C}_n \vdash v_n : \text{share_as}(u^{\mathrm{ls}}, u_n)\ \tau_n\end{array}}{\mathbb{C}_1 \# \ldots \# \mathbb{C}_n \vdash \mathrm{S}(v_1, ..., v_n) : \mathrm{S} \text{ containing } u^{\mathrm{ls}}}$$

$$\frac{\mathbb{L};\ \mathbb{R}_1;\ \varnothing \vdash s_1 : \mathbb{L}(\ell_1)\ u^{\mathrm{ls}} \qquad \ldots \qquad \mathbb{L};\ \mathbb{R}_n;\ \varnothing \vdash s_n : \mathbb{L}(\ell_n)\ u^{\mathrm{ls}}}{\mathbb{L};\ \mathbb{R}_1 \# \ldots \# \mathbb{R}_n \vdash \{\ell_1 \mapsto s_1, \ldots, \ell_n \mapsto s_n\} : u^{\mathrm{ls}}}$$

Fig. 3. Type Checking Rules (See Figure 4 for definitions of notation)

$(\mathbb{L}_1;\ \mathbb{R}_1;\ \mathbb{X}_1), (\mathbb{L}_2;\ \mathbb{R}_2;\ \mathbb{X}_2) = \mathbb{L}_1, \mathbb{L}_2;\ \mathbb{R}_1, \mathbb{R}_2;\ \mathbb{X}_1, \mathbb{X}_2$ where "," splits into pieces with disjoint domains.
$(\mathbb{L};\ \mathbb{R}_1;\ \mathbb{X}_1) \# (\mathbb{L};\ \mathbb{R}_2;\ \mathbb{X}_2) = \mathbb{L};\ \mathbb{R}_1 \# \mathbb{R}_2;\ \mathbb{X}_1 \# \mathbb{X}_2$ where:
$\mathbb{R} = \mathbb{R}_1 \# \mathbb{R}_2$ iff $!\mathbb{R} = !\mathbb{R}_1 = !\mathbb{R}_2$ and $¡\mathbb{R} = ¡\mathbb{R}_1, ¡\mathbb{R}_2$ $\qquad$ $\mathbb{X} = \mathbb{X}_1 \# \mathbb{X}_2$ iff $!\mathbb{X} = !\mathbb{X}_1 = !\mathbb{X}_2$ and $¡\mathbb{X} = ¡\mathbb{X}_1, ¡\mathbb{X}_2$

$\mathbb{C} = \mathbb{L};\ \mathbb{R};\ \mathbb{X}$ where $\mathbb{C}_1 \# \mathbb{C}_2 \# \ \dots \ \# \mathbb{C}_n$ denotes $\mathbb{C}_1 \# (\mathbb{C}_2 \# (\dots \# \mathbb{C}_n) \dots)$ for $n \geq 1$ and $!\mathbb{C}$ for $n = 0$

$!(\mathbb{L};\ \mathbb{R};\ \mathbb{X}) = \mathbb{L};\ !\mathbb{R};\ !\mathbb{X}$ and $¡(\mathbb{L};\ \mathbb{R};\ \mathbb{X}) = \varnothing;\ ¡\mathbb{R};\ ¡\mathbb{X}$ where:
$!\mathbb{R} = \{\mathrm{r} \mapsto u \in \mathbb{R} \mid u \neq \mathrm{linear}\}$ $\qquad$ $¡\mathbb{R} = \{\mathrm{r} \mapsto u \in \mathbb{R} \mid u = \mathrm{linear}\}$
$!\mathbb{X} = \{x \mapsto u\,\tau \in \mathbb{X} \mid u \neq \mathrm{linear}\}$ $\qquad$ $¡\mathbb{X} = \{x \mapsto u\,\tau \in \mathbb{X} \mid u = \mathrm{linear}\}$

$\mathrm{linear}(\mathbb{L};\ \mathbb{R};\ \mathbb{X}) = (\varnothing;\ \mathrm{linear}(\mathbb{R});\ \mathrm{linear}(\mathbb{X}))$ and $\mathrm{shared}(\mathbb{L};\ \mathbb{R};\ \mathbb{X}) = (\varnothing;\ \mathrm{shared}(\mathbb{R});\ \mathrm{shared}(\mathbb{X}))$
where linear(...) and shared(...) replace usages u with linear and shared respectively:
$\mathrm{linear}(\mathbb{R}) = \{\mathrm{r} \mapsto \mathrm{linear} \mid \mathrm{r} \mapsto u \in \mathbb{R}\}$ $\qquad$ $\mathrm{shared}(\mathbb{R}) = \{\mathrm{r} \mapsto \mathrm{shared} \mid \mathrm{r} \mapsto u \in \mathbb{R}\}$
$\mathrm{linear}(\mathbb{X}) = \{x \mapsto \mathrm{linear}\,\tau \mid x \mapsto u\,\tau \in \mathbb{X}\}$ $\qquad$ $\mathrm{shared}(\mathbb{X}) = \{x \mapsto \mathrm{shared}\,\tau \mid x \mapsto u\,\tau \in \mathbb{X}\}$

$\mathrm{share_as}(\mathrm{shared}, \mathrm{linear}) = \mathrm{shared}$ and $\mathrm{share_as}(u_1, u_2) = u_2$ otherwise.

Fig. 4. Notation and definitions for type checking

This helps to keep the verification condition formulas small and simple, which in turn helps to keep the SMT validity checking fast and predictable.

2.4 Regions, Part 1: Nonlinear Data Inside Linear Data

Linear typing makes it easy to express tree-shaped data structures, but is less suited to DAGs and cyclic structures. Rust, for example, can express cycles by using a combination of reference counting and the `RefCell` type, but the operations on `RefCell` require run-time safety checks that can panic (abort) if the `RefCell` is not in the expected run-time state.

However, Linear Dafny can draw on Dafny's standard dynamic frames, which can already reason about non-tree data structures. Using dynamic frames, programs can allocate objects in the global heap and can refer to the objects with ordinary references, which can be freely duplicated to create, for example, doubly linked lists. Furthermore, since linear data structures can contain ordinary fields, it is straightforward to combine linear data and nonlinear data by placing ordinary references inside linear datatypes.

However, there is a snag when trying to verify linear data structures that contain heap references: any modifications to the heap must be reported via a Dafny modifies clause. This modifies clause cannot be hidden; if a method `m` modifies the heap, it must report the modifications to any callers of `m`, which then must report the modifications to their callers, and so on up the call stack.

This, unfortunately, can interfere with modularity. Suppose we have a linear queue built out of Linear Dafny's linear sequences, with no heap operations, and therefore no modifies clauses. Then suppose we decide to change the queue's implementation from using sequences to using doubly linked lists built out of heap objects. Suddenly, the queue's public interface must change to report all the heap modifications, even though the doubly linked list is a private implementation detail. Worse, the callers of the queue's public methods must propagate these heap modifications to their callers. This is not just a hypothetical concern: in one place, instead of using heap objects, VeriBetrKV's least-recently-used queue represented a doubly linked list using integers to encode pointers, in the style of old Fortran code, to avoid introducing any modifies clauses in the queue's public interface.

The fundamental issue is not the modifies clauses themselves, but rather that all the modifies clauses refer to a single global heap, and any write to the global heap might affect anyone reading the global heap. If a queue had its own private heap, it could modify the private heap without causing any modifications to the global heap. This is precisely what type systems for *regions* [Tofte and

$$
\begin{array}{rcl}
\text{SMT formula } f & ::= & v \mid x \mid \text{true} \mid \text{false} \mid \forall x.\, f \mid f_1 \wedge f_2 \mid f_1 \Rightarrow f_2 \mid \neg f \\
& \mid & f_1 = f_2 \mid f_1 + f_2 \mid \langle f_1, \ldots, f_n \rangle \mid f.i \mid \text{let } x = f_1 \text{ in } f_2 \\
& \mid & \text{name_of_region}(f) \mid \text{valid_ref}(f_1, f_2) \mid \text{fresh_ref}(f_1) \mathbin{@} f_0 \\
& \mid & \text{read}(f_1.i) \mathbin{@} f_0 \mid \text{modifies}(f_1.i_1, \ldots, f_n.i_n) \mathbin{@} (f_0 \rightarrow f_0')
\end{array}
$$

$$
\begin{array}{rcl}
\text{wp}(i, x.\, f) & = & f[x := i] \\
\text{wp}(\ell, x.\, f) & = & f[x := \ell] \\
\text{wp}(\text{null}, x.\, f) & = & f[x := \text{null}] \\
\text{wp}(\text{r}\, d, x.\, f) & = & f[x := \text{r}\, d] \\
\text{wp}(x_0, x.\, f) & = & f[x := x_0] \\
\text{wp}(e_1 + e_2, x.\, f) & = & \text{wp}(e_1, x_1.\, \text{wp}(e_2, x_2.\, f[x := x_1 + x_2])) \\
\text{wp}(e_1;\, e_2, x_2.\, f) & = & \text{wp}(e_1, x_1.\, \text{wp}(e_2, x_2.\, f)) \\
\text{wp}(\text{let } u\, x_1 = e_1 \text{ in } e_2, x_2.\, f) & = & \text{wp}(e_1, x_1.\, \text{wp}(e_2, x_2.\, f)) \\
\text{wp}(\langle e_1, \ldots, e_n \rangle, x.\, f) & = & \text{wp}(e_1, x_1.\, \ldots \text{wp}(e_n, x_n.\, f[x := \langle x_1, \ldots, x_n \rangle]) \ldots) \\
\text{wp}(e_0.i, x.\, f) & = & \text{wp}(e_0, x_0.\, f[x := x_0.i]) \\
\text{wp}(\text{new_region}(), x.\, f) & = & \forall x. f \\
\text{wp}(\text{free_region}(e_0), x.\, f) & = & \text{wp}(e_0, x_0.\, \forall x. f) \\
\text{wp}(\text{alloc S}(e_1, ..., e_n) \mathbin{@} e_0, x'.\, f) & = & \text{wp}(e_0, x_0.\, \text{wp}(e_1, x_1.\, ...\text{wp}(e_n, x_n. \\
& & \quad \forall x_l'. \forall x_0'. \\
& & \quad\; (\text{valid_ref}(x_l', \text{name_of_region}(x_0)) \wedge \text{fresh_ref}(x_l') \mathbin{@} x_0 \wedge \\
& & \quad\; x_1 = \text{read}(x_l'.1) \mathbin{@} x_0' \wedge ... \wedge x_n = \text{read}(x_l'.n) \mathbin{@} x_0' \wedge \\
& & \quad\; \text{modifies}() \mathbin{@} (x_0 \rightarrow x_0')) \Rightarrow f[x' := \langle x_l', x_0' \rangle])...)) \\
\text{wp}(\text{read}(e_1.i) \mathbin{@} e_0, x.\, f) & = & \text{wp}(e_0, x_0.\, \text{wp}(e_1, x_1. \\
& & \quad \text{valid_ref}(x_1, \text{name_of_region}(x_0)) \wedge \\
& & \quad f[x := \text{read}(x_1.i) \mathbin{@} x_0])) \\
\text{wp}(\text{write}(e_1.i := e_2) \mathbin{@} e_0, x_0'.\, f) & = & \text{wp}(e_0, x_0.\, \text{wp}(e_1, x_1.\, \text{wp}(e_2, x_2. \\
& & \quad \text{valid_ref}(x_1, \text{name_of_region}(x_0)) \wedge \\
& & \quad (\forall x_0'. (x_2 = \text{read}(x_1.i) \mathbin{@} x_0' \wedge \text{modifies}(x_1.i) \mathbin{@} (x_0 \rightarrow x_0')) \Rightarrow f)))) \\
\text{wp}(\text{swap}(e_1.i := e_2) \mathbin{@} e_0, x'.\, f) & = & \text{wp}(e_0, x_0.\, \text{wp}(e_1, x_1.\, \text{wp}(e_2, x_2. \\
& & \quad \text{valid_ref}(x_1, \text{name_of_region}(x_0)) \wedge \\
& & \quad (\forall x_0'. \forall x_2'. \\
& & \quad\; (x_2 = \text{read}(x_1.i) \mathbin{@} x_0' \wedge x_2' = \text{read}(x_1.i) \mathbin{@} x_0 \wedge \\
& & \quad\; \text{modifies}(x_1.i) \mathbin{@} (x_0 \rightarrow x_0')) \Rightarrow f[x' := \langle x_2', x_0' \rangle]))))
\end{array}
$$

Fig. 5. Weakest Precondition Rules (all introduced bound variables are fresh to avoid capturing free variables)

Talpin 1994; Walker and Watkins 2001] allow. In this section, we demonstrate how to incorporate regions, objects, and references into a linear type system with SMT-based verification.

We define a region to be a pair of a region name r and region data d. The region data is essentially a small heap that maps locations ℓ to objects. Each object has a type given by a struct name S, which defines zero or more fields. Each struct field has a type and a usage (either linear or ordinary, as with tuple fields). We assume that there is a mapping StructType(S) from each struct's name to its fields:

$$\text{StructType(S)} = (u_1^{\text{lo}}\, \tau_1, \ldots, u_n^{\text{lo}}\, \tau_n)$$

Each object is then a struct instance $s = \text{S}(v_1, ..., v_n)$. Note that although struct instances contain values, struct instances are not themselves values. Instead, struct instances of type S are referred to by references of type ref(S), and these references are values. Using references allows the for the creation of cyclic data structures, as in the following example, which creates two objects pointing

to each other, and then adds the integers in the objects together to compute $z = 300$:

$$\begin{array}{l} \text{let linear}\, x = \text{new_region}()\ \text{in} \\ \text{let}\ \langle y_1, x\rangle = \text{alloc}\ \mathrm{S}_1(100, \text{null})\ @\ x\ \text{in} \\ \text{let}\ \langle y_2, x\rangle = \text{alloc}\ \mathrm{S}_1(200, y_1)\ @\ x\ \text{in} \\ \text{let linear}\, x = \text{write}(y_1.2 := y_2)\ @\ x\ \text{in} \\ \text{let ordinary}\, z = \text{read}(y_1.1)\ @\ x + \text{read}(y_2.1)\ @\ x\ \text{in} \\ \text{free_region}(x); \\ z \end{array}$$

In this example, we assume $\text{StructType}(\mathrm{S}_1) = (\text{ordinary int}, \text{ordinary ref}(\mathrm{S}_1))$. The example code creates a region x, allocates two objects in the region referred to by references y_1 and y_2, and stores a reference to y_1 in y_2 (via $\text{alloc}\,\mathrm{S}_1(200, y_1)\,@\,x$) and a reference to y_2 in y_1 (via $\text{write}(y_1.2 := y_2)\,@\,x$). The region must be provided to allocation, read, and write using the @ syntax. As shown in the typing rules in Figure 3 and Figure 4, the allocation and writes treat the region linearly, consuming the original region and producing an updated region. Reads treat the region as shared. Freeing a region consumes the region completely; this prevents any further allocations, reads, or writes in the region, since region is no longer available as linear or shared after being consumed.

The weakest precondition definition (see Figure 5) can be used to verify that the example above uses references safely and computes a final value of 300. For verifying region operations, the key elements of weakest preconditions are the modifies predicate, the fresh_ref predicate, and the valid_ref predicate.

The $\text{modifies}(\ell_1.i_1, \ldots, \ell_n.i_n)\ @\ (\mathrm{r}\, d \rightarrow \mathrm{r}\, d')$ predicate is our per-region equivalent of Dafny's global-heap modifies clause. It says that region $\mathrm{r}\, d'$ is the same as $\mathrm{r}\, d$, except that it may contain new allocations and it may contain modifications in the listed fields of the listed locations $\ell_1.i_1, \ldots, \ell_n.i_n$. In the example above, the allocations and write preserve the integer fields, modifying only the $y_1.2$ field, thus allowing the SMT solver to conclude that $\text{read}(y_1.1)\ @\ x + \text{read}(y_2.1)\ @\ x = 300$.

The $\text{fresh_ref}(\ell)\ @\ \mathrm{r}\, d$ predicate says that ℓ is not in the domain of d. When the expression $\text{alloc}\ \mathrm{S}(e_1, \ldots, e_n)\ @\ \mathrm{r}\, d$ allocates a new object in region $\mathrm{r}\, d$, it ensures that the object's location ℓ is fresh ($\text{fresh_ref}(\ell)\ @\ \mathrm{r}\, d$). In the example above, this ensures that the locations y_1 and y_2 are distinct.

The $\text{valid_ref}(\ell, \mathrm{r})$ predicate says that the reference ℓ is a valid reference into the region named r, which is true iff ℓ was allocated in region r using alloc ... @ r. Note that since the $\text{valid_ref}(\ell, \mathrm{r})$ predicate refers only to the region name r, not the region data d, the predicate remains valid forever, even as the region data d changes, and even after the region $\mathrm{r}\, d$ is freed. This makes $\text{valid_ref}(\ell, \mathrm{r})$ ideal for an SMT solver's classical logic, in which a true fact remains true forever. This also leads to a clear division of labor between the SMT solver, which handles $\text{valid_ref}(\ell, \mathrm{r})$, and the type system, which checks that the region is still alive during any read or write to ℓ. These two checks together ensure that ℓ points into r and that r is still alive. This division of labor contrasts with traditional type systems for regions [Fluet et al. 2006; Grossman et al. 2002; Walker and Watkins 2001], which use region variables to ensure that ensure that ℓ points into r, and thus require universal and/or existential quantification of region variables. By offloading the $\text{valid_ref}(\ell, \mathrm{r})$ check to the SMT solver, our system avoids such quantification. In a language like Dafny that already supports SMT solving, this is a significant simplification for the language and for the programmer. For example, Rust supports universal quantification of lifetime variables, but does not yet support existential quantification, making it difficult to encapsulate lifetimes inside data structures.

By contrast, our system can simply use an SMT solver's existing features, including equality, function application, negation, conjunction, and existential quantification, to encapsulate region names inside data structures. Suppose we wanted to encapsulate the cyclic pair of objects y_1, y_2

from the example above as an abstract linear value. We can define a type τ and a well-formedness predicate $wf(a:\tau)$ that ensures that the pair forms a cycle and that the references in the pair are valid:

$$\begin{array}{rcl} \tau & = & \langle \text{linear region, ordinary ref}(\mathrm{S}_1)\rangle \\ wf(a:\tau) & = & \text{let } x = a.1 \text{ in} \\ & & \text{let } y_1 = a.2 \text{ in} \\ & & \text{let } y_2 = \text{read}(y_1.2) \ @ \ x \text{ in} \\ & & \text{valid_ref}(y_1, \text{name_of_region}(x)) \land \\ & & \text{valid_ref}(y_2, \text{name_of_region}(x)) \land \\ & & \neg(y_1 = y_2) \land \\ & & y_1 = \text{read}(y_2.2) \ @ \ x \end{array}$$

The type τ contains the region x and the head pointer y_1. The four conjuncts of $wf(a:\tau)$ ensure that y_1 is valid, that y_2 is valid, that $y_1 \neq y_2$, and that y_2 points to y_1. (The SMT formula name_of_region(x) returns the region name r of a region $x = \mathrm{r}\, d$, and the SMT formula $\text{read}(\ell.i)\ @\ x$ returns the value stored in field i of the object referenced by ℓ in the region's data d.)

Dafny makes it straightforward to define types like τ and predicates $wf(a:\tau)$, and to hide the definitions of τ and $wf(a:\tau)$ behind a module interface. Such a module can also define methods that write to the region stored in τ. These writes will change the value of the region stored in τ, and this change will be captured by a predicate $\text{modifies}(\ldots)\ @\ (x \longrightarrow x')$. However, this predicate is simply local information that the method can use if it desires; the method is under no obligation to report the modification to the method's callers:

```
type t = ...
predicate wf(a:t) { ... }
method m(linear inout a:t)
  requires wf(a)
  ensures wf(a)
  // does not modify the global Dafny heap, so no modifies clause here
{
  ... write to a, optionally taking advantage of region modifies(...) @ (...) ...
}
```

Thus, the region-based approach successfully encapsulates modifications to objects behind a linear interface. Using this approach, we have developed a Linear Dafny library for regions with ML-style ref cells, and we have used this to re-implement VeriBetrKV's doubly linked list using region objects and references[4], rather than integer "pointers". Note that this approach does not require physically switching an implementation to region-based memory management. In fact, since Dafny already includes automated memory management based on garbage collection (when compiling Dafny to C#) and reference counting (when compiling Dafny to C++), our region library uses a purely ghost representation of regions where new_region() and free_region() have no runtime effect. In this approach, the region primitives are just a way to assist verification by applying linear and shared access control to groups of related nonlinear objects.

2.5 Regions, Part 2: Linear Data Inside Nonlinear Data

The previous section discussed storing nonlinear data inside linear data, but the opposite direction is also useful. For example, Rust's type `RefCell<T>` can be used in conjunction with reference counting (`Rc<RefCell<T>>`) to create multiple pointers to a single value of type `T`, while still guaranteeing that the `T` value is used linearly or borrowed immutably. However, Rust depends on

[4] Details are available at https://github.com/jialin-li/linear-veribetrkv-artifact/tree/master/formalization

run-time checks to make this guarantee, and can panic if the `RefCell<T>` is not in the expected state. In particular, `RefCell`'s immutable borrowing operation, `borrow`, panics if the `RefCell` is currently mutably borrowed, and the mutable operations `borrow_mut` and `swap` panic if the `RefCell` is currently mutably or immutably borrowed. These run-time checks add time and space overhead, and introduce places where the entire program may fail because of a panic.

To avoid these overheads and failures, our language uses two techniques that carry no run-time overhead and do not fail at run-time: reading linear values from regions that are shared, and swapping linear values in regions that are linear. The first of these integrates into the type checking rules quite naturally. Consider Figure 3's rule for tuple field selection:

$$\frac{\mathbb{C} \vdash e : \text{shared}\ \langle u_1\ \tau_1, \ldots, u_i\ \tau_i, \ldots, u_n\ \tau_n \rangle}{\mathbb{C} \vdash e.i : \text{share_as}(\text{shared}, u_i)\ \tau_i}$$

This rule combines the two rules from Section 2.2 by using Figure 3's share_as(shared, u_2) function, which demotes u_2 from linear to shared and leaves shared and ordinary unchanged. The rule for reading from region objects is essentially the same, except that instead of requiring a shared tuple, it requires a shared region:

$$\frac{\text{StructType(S)} = (u_1\ \tau_1, \ldots, u_i\ \tau_i, \ldots, u_n\ \tau_n) \qquad \mathbb{C}_0 \vdash e_0 : \text{shared region} \qquad \mathbb{C}_1 \vdash e_1 : \text{ordinary ref(S)}}{\mathbb{C}_0 \# \mathbb{C}_1 \vdash \text{read}(e_1.i)\ @\ e_0 : \text{share_as}(\text{shared}, u_i)\ \tau_i}$$

Using this rule, a program can read a linear field (u_i = linear) as shared (share_as(shared, u_i) = shared), for as long as the region remains shared. Since mutations of the region require a linear region, not a shared region, and since a program can never see a region as linear and shared simultaneously, there is no danger of the program mutating the field while the shared field is in use.

Section 5.1 of [Hance et al. 2020a] also presented a way to borrow a linear value stored inside a nonlinear object, but the mechanism relied on a rather awkward special attribute named `caller_must_be_pure` with special rules to check that no heap modifications occurred within the scope of the borrowing. This special treatment was necessary because [Hance et al. 2020a] could store nonlinear objects only in the global Dafny heap, not in regions, and the global Dafny heap is not a first class linear value like regions are, so there is no natural way to view the heap as a shared value. Furthermore, forcing the entire global heap to be read-only while borrowing is excessively restrictive; using fine-grained regions allows for some regions to be read-only while other regions are mutated.

Figure 3 also shows the rule for swapping linear values in a region object. This rule requires that the region be linear, not shared. In this way, the rule ensures that when the swap happens, the program is not simultaneously viewing the linear field as shared, and in contrast to to Rust's `RefCell` swap operation, this guarantee requires no run-time overhead.

Using swap, a region-based program can implement the `Take` and `Give` operations described in [Hance et al. 2020a] for regions: these support retrieving (`Take`) a linear value from nonlinear data in order to mutate it, and replacing it (`Give`) afterwards. To do this, the program stores the type `Maybe<T>` rather than `T` in a field, where `Maybe<T>` may either be `Some` value of type `T` or `None`. Whether the value is `Some` or `None` is a ghost property that carries no run-time overhead, and the program must prove, using SMT solving, that the value is `Some` to extract the underlying `T` value. SMT verification helps avoid run-time time and space overhead, and avoids unexpected run-time failures (encountering a `None` when a `Some` is expected).

2.6 Type Checking Algorithm

As described in Section 2.2, Figure 3's typing rules use nondeterministic splitting of typing environments to check subexpressions. Fortunately, the programmer does not have to specify this splitting

explicitly. Instead, a very simple and commonly used algorithm for linear type systems computes the splitting lazily. If the type checker needs to split $\mathbb{X} = \mathbb{X}_1, \mathbb{X}_2$ to check e_1 in $\mathbb{X}_1$ and e_2 in $\mathbb{X}_2$, the algorithm first checks e_1 using the entire environment $\mathbb{X}$ and tracks which linear variables e_1 actually consumes; these consumed variables go into $\mathbb{X}_1$, and the remainder may be placed in $\mathbb{X}_2$.

This section demonstrates that this simple, common algorithm can be extended to infer borrowing in a very straightforward way. Given that Rust also automatically infers borrowing, it should not be surprising that such inference is possible. But in case anyone implementing a linear type system is intimidated by the complexity of Rust's inference algorithm, we want to reassure them that even a simple algorithm can be sound and complete for a simple linear type system with borrowing (and we want to encourage them to implement such an algorithm rather than requiring programmers to write explicit let! expressions). We have implemented this inference algorithm as part of Linear Dafny's type checker.

Like the common algorithm for linear type systems without borrowing, our algorithm checks subexpressions using all linear variables, and then tracks which linear variables the subexpressions consume. However, our algorithm also tracks which linear variables the subexpressions borrow:

$$\mathbb{X} \vdash \mathrm{c}\ e : u\ \tau\ \mathrm{borrows}(B)\ \mathrm{consumes}(C)$$

where B and C are sets of variables that are borrowed and consumed. For example, the algorithm's rule for $e_1;\ e_2$ is:

$$\frac{\begin{array}{c}\mathbb{X} \vdash \mathrm{borrow}\ e_1 : u_1\ \tau_1\ \mathrm{borrows}(B_1)\ \mathrm{consumes}(C_1)\\ \mathbb{X} \vdash \mathrm{c}\ e_2 : u_2\ \tau_2\ \mathrm{borrows}(B_2)\ \mathrm{consumes}(C_2)\\ u_1 \neq \mathrm{linear} \qquad C_1 \cap C_2 = \varnothing \qquad C_1 \cap B_2 = \varnothing\end{array}}{\mathbb{X} \vdash \mathrm{c}\ e_1;\ e_2 : u_2\ \tau_2\ \mathrm{borrows}((B_1 \setminus C_2) \cup B_2)\ \mathrm{consumes}(C_1 \cup C_2)}$$

This rule allows e_1 and e_2 to consume non-disjoint sets of variables C_1 and C_2, as in a standard linear type checker without borrowing. It also allows e_1 to borrow variables $B_1 \cap C_2$ that are consumed by e_2. Finally, e_1 and e_2 may borrow additional variables $(B_1 \setminus C_2) \cup B_2$ that must be consumed later.

The one non-obvious aspect of the algorithm is that it relies on an expected "consumption" c that is passed into each expression's type checking, along with the environment $\mathbb{X}$ and the expression e:

$$\mathrm{c} ::= \mathrm{borrow}\ \mid\ \mathrm{consume}$$

If c = borrow, then a linear variable will be borrowed, and if c = consume, then a linear variable will be consumed. For shared variables and ordinary variables, c is ignored. Fortunately, for every expression e in Figure 2 (and in Linear Dafny in general), it is unambiguous which c to use for each of e's subexpressions. For example, for e = free_region(e_1), the algorithm chooses c = consume for e_1, since e_1 must be linear, not shared, and thus if e_1 is a variable, the variable is consumed, not borrowed. Similarly, in $e_0.i$, the subexpression e_0 must be shared, so the algorithm chooses c = borrow for e_0. One limitation of the algorithm is that it relies on the programmer's u annotation in let $u\,x = e_1$ in e_2 to choose c for e_1. On the other hand, this u annotation serves as useful documentation, so it is reasonable to require programmers to provide this information. (Note that Linear Dafny assumes a variable is ordinary if not annotated with linear or shared, so ordinary variables need no annotation.)

We have proven that this algorithm is sound and complete[4]:

Theorem: Type Inference Soundness. For any c, if
$\mathbb{X} \vdash \mathrm{c}\ e : u\ \tau\ \mathrm{borrows}(\varnothing)\ \mathrm{consumes}(\mathrm{domain}(¡\mathbb{X}))$ then $\varnothing;\ \varnothing;\ \mathbb{X} \vdash e : u\ \tau$.
Theorem: Type Inference Completeness. For any c, if $\varnothing;\ \varnothing;\ \mathbb{X} \vdash e : u\ \tau$, and $u = \mathrm{linear} \Longrightarrow \mathrm{c} = \mathrm{consume}$, and $u = \mathrm{shared} \Longrightarrow \mathrm{c} = \mathrm{borrow}$, then $\mathbb{X} \vdash \mathrm{c}\ e : u\ \tau\ \mathrm{borrows}(\varnothing)\ \mathrm{consumes}(\mathrm{domain}(¡\mathbb{X}))$.

$$\begin{array}{llll}
\text{evaluation context} & E & ::= & [\cdot] \mid E_1 + e_2 \mid v_1 + E_2 \mid E_1;\, e_2 \mid \text{let}\, u\, x = E_1 \text{ in } e_2 \\
& & \mid & \langle v_1, \ldots, v_i, E_j, e_k, \ldots, e_n \rangle \mid E.i \mid \text{let}\, \langle x_1, \ldots, x_n \rangle = E_1 \text{ in } e_2 \\
& & \mid & \text{free_region}(E) \mid \text{alloc}\, \mathrm{S}(e_1, \ldots, e_n) \mathbin{@} E_0 \\
& & \mid & \text{alloc}\, \mathrm{S}(v_1, \ldots, v_i, E_j, e_k, \ldots, e_n) \mathbin{@} v_0 \\
& & \mid & \text{read}(e_1.i) \mathbin{@} E_0 \mid \text{read}(E_1.i) \mathbin{@} v_0 \mid \text{write}(e_1.i := e_2) \mathbin{@} E_0 \\
& & \mid & \text{write}(E_1.i := e_2) \mathbin{@} v_0 \mid \text{write}(v_1.i := E_2) \mathbin{@} v_0 \\
& & \mid & \text{swap}(e_1.i := e_2) \mathbin{@} E_0 \mid \text{swap}(E_1.i := e_2) \mathbin{@} v_0 \\
& & \mid & \text{swap}(v_1.i := E_2) \mathbin{@} v_0
\end{array}$$

$$\frac{e \to e'}{E[e] \longrightarrow E[e']} \qquad i_1 + i_2 \to i_3 \text{ where } i_3 \text{ is sum of } i_1, i_2 \qquad v_1;\, e_2 \to e_2 \qquad \text{let}\, u\, x = v_1 \text{ in } e_2 \to e_2[x := v_1]$$

$$\langle v_1, \ldots, v_i, \ldots, v_n \rangle.i \to v_i \qquad \text{let}\, \langle x_1, \ldots, x_n \rangle = \langle v_1, \ldots, v_n \rangle \text{ in } e_0 \to e_0[x_1 := v_1, \ldots, x_n := v_n]$$

$$\text{new_region}() \to \mathrm{r}\, \varnothing \text{ where r is fresh} \qquad \text{free_region}(\mathrm{r}\, d) \to 0 \qquad \text{read}(\ell.i) \mathbin{@} \mathrm{r}\, (d, \ell \mapsto \mathrm{S}(v_1, \ldots, v_n)) \to v_i$$

$$\text{write}(\ell.i := v_i') \mathbin{@} \mathrm{r}\, (d, \ell \mapsto \mathrm{S}(v_1, \ldots, v_i, \ldots, v_n)) \to \mathrm{r}\, (d, \ell \mapsto \mathrm{S}(v_1, \ldots, v_i', \ldots, v_n))$$

$$\text{swap}(\ell.i := v_i') \mathbin{@} \mathrm{r}\, (d, \ell \mapsto \mathrm{S}(v_1, \ldots, v_i, \ldots, v_n)) \to \langle v_i, \mathrm{r}\, (d, \ell \mapsto \mathrm{S}(v_1, \ldots, v_i', \ldots, v_n)) \rangle$$

$$\text{alloc}\, s \mathbin{@} \mathrm{r}\, d \to \langle \ell, \mathrm{r}\, (d, \ell \mapsto s) \rangle \text{ where } \ell \text{ is fresh and RegionOfLoc}(\ell) = \mathrm{r}$$

Fig. 6. Operational semantics

2.7 Semantics and Soundness

Figure 6 shows the operational semantics of the formal model. When allocating fresh locations ℓ for objects, we assume a function RegionOfLoc(ℓ) that maps locations to region names such that each region name r has an infinite number of locations ℓ that satisfy RegionOfLoc(ℓ) = r. This is used in the type checking rules in Figure 3 and we define the SMT formula valid_ref(ℓ, r) to mean RegionOfLoc(ℓ) = r.

Using the operational semantics, type checking rules, and weakest precondition rules, we have proven the soundness of the formal language (see supplemental materials for proofs and further details):

Theorem: Type Preservation. If $\mathbb{L};\ \mathbb{R};\ \mathbb{X} \vdash e : u\ \tau$ and $e \longrightarrow e'$, then there is some $\mathbb{L}'$ and $\mathbb{R}'$ such that $\mathbb{L}';\ \mathbb{R}';\ \mathbb{X} \vdash e' : u\ \tau$.
Theorem: WP Preservation. If $\mathbb{L};\ \mathbb{R};\ \mathbb{X} \vdash e : u\ \tau$ and $e \longrightarrow e'$ and $\mathrm{wp}(e, x.\ f)$ is valid, then $\mathrm{wp}(e', x.\ f)$ is valid.
Theorem: Progress. If $\mathbb{L};\ \mathbb{R};\ \varnothing \vdash e : u\ \tau$ and $\mathrm{wp}(e, x.\ f)$ is valid and e is not a value v, then there is some e' such that $e \longrightarrow e'$.

The operational semantics treat regions as values rather than keeping the region data in a separate global mutable heap. This simplifies the semantics and emphasizes that during verification, the SMT solver views regions as values, so that the weakest preconditions do not need to read and write to any global heap. However, there is a slight danger in this approach that the formal semantics could fail to correspond to the real Linear Dafny implementation, since the semantics creates copies of regions during borrowing (via substitution), while the implementation passes regions by reference, not by copy. Therefore, we have proven an additional theorem[4] that all copies of any region r contain exactly the same data, and can therefore be safely implemented by reference rather than by copy:

Theorem: Region Agreement Preservation. If $\mathbb{L}; \mathbb{R}; \mathbb{X} \vdash e : u\ \tau$ and $e \longrightarrow e'$ and $|\mathrm{versions}_{\mathrm{r}}(e)| \leq 1$, then $|\mathrm{versions}_{\mathrm{r}}(e')| \leq 1$ and if r was not chosen as a fresh region name during $e \longrightarrow e'$, then $|\mathrm{versions}_{\mathrm{r}}(e')| \leq |\mathrm{versions}_{\mathrm{r}}(e)|$.

Here, $\text{versions}_\text{r}(e)$ collects the set of all copies of the region named r appearing in e, so that if the cardinality of this set is less than or equal to 1, there is at most one version of the region, and therefore the region can safely be implemented by reference rather than by copy.

3 CASE STUDY

The previous section showed how our linear type system flexibly nests nonlinear data inside linear data and vice versa. We assert that this mixing exploits the common case of linear data with a fallback when linearity is too restrictive, leading a better development experience. Here we evaluate that assertion via an apples-to-apples comparison of both approaches in a large-scale case study. We evaluate the development experience in terms of proof burden (Section 3.4), interactive performance (Section 3.5), and the interpretability of error messages (Section 3.6).

3.1 VeriBetrKV

The experiments herein were performed on the code base from VeriBetrKV [Hance et al. 2020a], a publicly-available large-scale key-value storage system designed for high performance. VeriBetrKV is a verified crash-safe key-value store built around a write-optimized B^ϵ-tree [Bender et al. 2015]. Verification is worthwhile for a novel storage system design because users are otherwise reluctant to entrust their valuable data to a system that may contain bugs that can corrupt or lose that data. VeriBetrKV comprises 44K lines of code and proof in total (~24K in the implementation level), producing 31K lines of generated C++ code.

We evaluate our linear type system on the VeriBetrKV code artifact because its size and complexity are representative of future verification-driven software development. Because VeriBetrKV is intended for real use, not just research, it prioritizes performance and maintainability. Performance demands integration of complex components, including a cache, a performant cache-eviction policy, a journal, asynchronous disk I/O, and an efficient disk-write scheduling policy that preserves crash-recovery semantics. Performance also demands the developers exercise explicit control over memory allocation and layout, and when necessary, object mutability and aliasing.

Because the system is complex, and because performance optimization demands code evolution, the system also requires a maintainable software engineering strategy, including maintenance of the proof artifact. VeriBetrKV's correctness argument is organized using a conventional refinement hierarchy of state machine models [Lamport 2002]. Models high in the hierarchy are purely functional models that abstract away details to focus on higher-level arguments like crash safety and the equivalence of the rich B^ϵ-tree data structure to dictionary semantics. The bottom layers of the hierarchy introduce imperative code, mutable data structures, and bounded `uint64` integers.

3.2 Challenges with SMT-Based Memory Reasoning

Section 1 introduced the three key challenges with mutable memory reasoning in the SMT-based verification setting, and now we elaborate on them.

The first is **proof burden**: Explicitly-managed dynamic frames are powerful enough to capture arbitrary aliasing relations, but most often that is more power than necessary. Given the common case of unaliased objects, the developer spends many lines of specification describing uninteresting memory invariants.

Second is **verification time**: A verification developer iterates between attempting proof fixes and waiting for feedback from the verifier. The proofs of dynamic frames invariants do not lend themselves to the heuristic automation in the SMT solver, leading to slow verifier response times and hence poor interactivity.

Third is the need for **decipherable error messages**: When memory invariants are insufficient, they manifest as verification errors in later assertions or postconditions about program logic.

Verification developers are accustomed to diagnosing logic errors by trying to understand the logical premises that lead to them. In the presence of memory errors, an assertion failure might mean a logical assertion is actually wrong, or an overlooked alias causes the logical assertion to fail. The latter is a much bigger search space for the human developer to investigate.

These problems are compounded by the system's scale. Object types gain more fields to reference the many data structures that describe the system's state, and tracking sets of accessible objects and their disjointness becomes more complex, compounding the tedium of managing their invariants. As methods interact with complex objects, each simple logical assertion is competing with a compounding number of heap-related invariants that may result in confounding aliasing failures. This manifests as poor interactivity, as well as unhelpful error messages from proof failures.

While developing VeriBetrKV, we discovered that these problems become a bottleneck at scale [Hance et al. 2020a]. We dealt with the second problem, poor interactivity, by splitting implementations into tiny methods, following an only-partly-joking "rule of three semicolons": if you need more than three heap mutating statements in a method, it is time to split it up.

We addressed the third challenge by introducing a "model-implementation split": we inserted a second-lowest layer into our refinement hierarchy with a high-fidelity functional model of the bottom-layer implementation. Because it is functional, the model dispatches all logical reasoning while avoiding all of the challenges above. Because the functional model is high-fidelity, the next implementation layer *only* addresses mutability and hence memory and aliasing reasoning. So an assertion failure in this layer can only refer to a memory invariant or an equivalence problem.

Both mitigation strategies create more tedious overhead.

Our key hypothesis is this: We sometimes want the flexibility of SMT-based memory reasoning, but in most cases we do not need it. Most objects are unaliased; for those objects, memory reasoning can be simple. Integrating linear types into Dafny exploits that simplicity, relieving the burden.

3.3 The VeriBetrKV Conversion Process

To test this thesis, we began with a publicly available version of VeriBetrKV [Hance et al. 2020b] that is entirely based on dynamic frames, and converted it into Linear Dafny. We refer to the dynamic-frame-based system as VeriBetrKV-DF and the linearized system as VeriBetrKV-LT.

In our original work on VeriBetrKV, we presented a small-scale (16.5% of the implementation layer) linear integration study with a primitive version of Linear Dafny [Hance et al. 2020a].

Here we present an enhanced Linear Dafny and a hybrid system, VeriBetrKV-LT, that consists of 91% linearized and 9% nonlinear code and proof in the implementation layer. By fully integrating an advanced linear type system into VeriBetrKV-DF, we can confirm that most system components are amenable to linear types, and can evaluate the development experience of a largely-linear hybrid system.

The conversion only addresses mutable heap reasoning, and thus has no effect on the higher layers of model and refinement (~20K lines) as they only manipulate functional objects. Hence in this analysis we only contrast the implementation layer of VeriBetrKV-DF (~24K lines of the high-fidelity functional model and the imperative implementation) and VeriBetrKV-LT. All improvements stated in the following sections are based on the implementation layer of the code base.

The conversion involves two stages: First, linearization converts unaliased data types to linear types and removes all code and proof used for heap reasoning. Second, the removal of the high-fidelity functional models. With linear types in place, the implementation now has a clean separation between memory reasoning dispatched by the type checker and logical conditions dispatched by the SMT solver, obviating the high-fidelity models. Whenever a model is eliminated, we move its correctness proofs into the implementation.

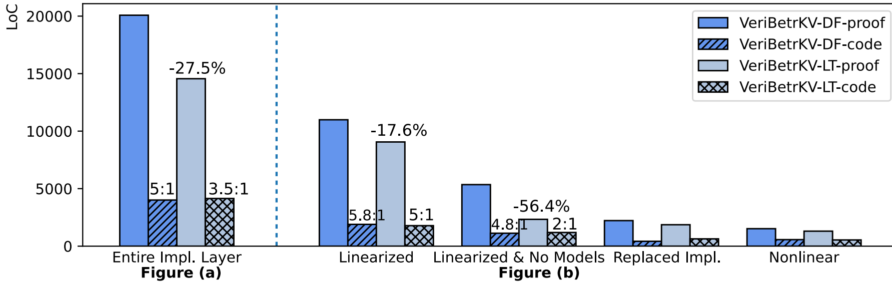

Fig. 7. Figure (a) counts the lines of code and proof in the implementation layer of VeriBetrKV-DF and VeriBetrKV-LT, highlighting the reduction in proof text that improves the proof-to-code ratio. Figure (b) breaks those counts down by how each method was converted. When the complete overhead of nonlinear types is accounted for, such as high-fidelity models, we observe a reduction of 56% in proof text.

When we observe methods with only one caller, we sometime merge the method into its caller, as this artificial split is often a result of the "rule of three semicolons" described above. We did not systematically target this case, however.

The conversion process proceeded incrementally, module-by-module and datatype-by-datatype, keeping the whole system type checking and verifying all the while. This incremental conversion was feasible because our type system supports linear data inside nonlinear data (Section 2.5).

By converting an existing system rather than building a hybrid system from scratch, the study enjoys a direct comparison. Because it inherits an existing proof structure designed for dynamic frames, however, the comparison underestimates the benefits of the hybrid approach.

3.4 Proof Burden

With this context, we first evaluate the effects on proof burden. Proof is all the lines of text a developer has to write, beyond the program itself and its specification, to satisfy the verifier that the program meets its specification. Proof burden comprises both lines of proof text and the effort the developer spends arriving at that text. Since it is difficult to measure the developer's effort, we use the line count of the final proof text as a proxy for the overall burden. This proxy indirectly measures the trials and errors of proof development and predicts how much proof the next developer will need to understand when extending the system with new features.

Figure 7a shows that the implementation layer of VeriBetrKV-LT has 28% fewer lines of proof than VeriBetrKV-DF. Since the two systems have nearly identical implementations, they have comparable amounts of executable code; therefore, the proof-to-code ratio improves by the same ratio, from 5:1 to 3.5:1.

Figure 7b breaks down Figure 7a by method into four groups. *Linearized* comprises data structures and methods that were converted into linear types, but the high-fidelity model was left in place because merging it would be too invasive. This category enjoyed an 18% reduction in proof text. *Linearized & No Models* comprises data structures and methods that were converted into linear types, and then their high-fidelity models were merged into the implementation. This category enjoyed a 56% reduction in proof text.

The *Replaced Impl.* category comprises data structures that were entirely restructured in VeriBetrKV-LT, and thus comparisons are not particularly meaningful. The *Nonlinear* category comprises data structures that remain in the dynamic frames paradigm in VeriBetrKV-LT. The modest improvement in proof burden was an artifact of unrelated refactoring while adapting to an upstream Dafny upgrade.

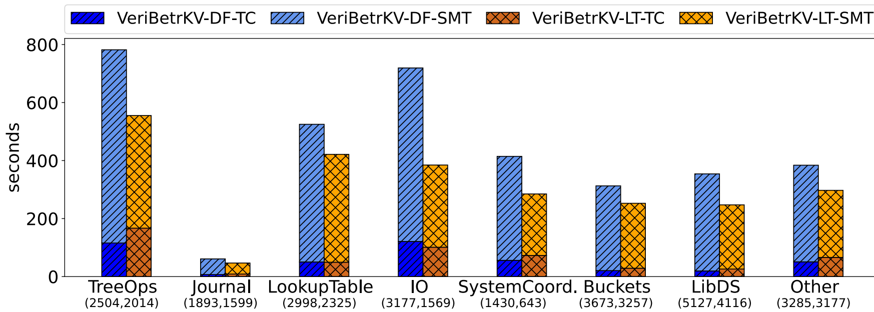

Fig. 8. SMT-solving and type checking time for each module in VeriBetrKV-DF and VeriBetrKV-LT. The size of each module is shown in the format (lines of text in DF, lines of text in LT). Linear type checking typically increases type checking time in exchange for substantially reducing solver time.

The conversion accounts for 90% of the overall proof reduction; the other 10% came from proof enhancement during the conversion development process.

While 91% of the implementation layer was amenable to linearization, we only merged models for 27% of the implementation, where the merge was noninvasive to the proof structure. If the linear type system had been available at the beginning of VeriBetrKV development, we would have created a different proof structure that did not rely on high-fidelity models. Thus the overall improvement of 28% likely underestimates the benefit; working from scratch might extract the benefits shown in the *Linearized & No Models* category across most of the system.

3.5 Verification Time

Short verification times provide a responsive, usable developer experience. Here we look at the time it takes to verify the entire implementation layer and to verify individual methods in VeriBetrKV-DF and VeriBetrKV-LT. Dafny type checks the program before running SMT queries, so we include both type checking and SMT-solving time as the verification time.

Despite identical functionality and nearly identical implementation, VeriBetrKV-LT's overall verification time is reduced by 30%. Figure 8 breaks down those changes by subcomponent. Some VeriBetrKV-LT modules, such as *TreeOps* experience increased type checking time due to the additional linear type checking pass. The increases are offset by significant reductions in the SMT-solving time.

In addition to the overall verification time, the interactivity of the verifier also plays an important role in a developer's experience. To evaluate interactivity, we collect the verification time of individual methods. Predicate, function and lemma definitions are unchanged or omitted in VeriBetrKV-LT and hence omitted from these measurements. Of the 353 methods measured, 284 verify within 5 seconds in VeriBetrKV-DF and exhibit no significant difference in verification time using Linear Dafny. These methods verify in a speedy manner and do not block developers. Thus we restrict our analysis to those 69 methods that take over 5 seconds in VeriBetrKV-DF.

Figure 9 shows the change in the verification time of the 69 methods as a cumulative distribution. A quarter of VeriBetrKV-LT methods take less than a third of their VeriBetrKV-DF verification time, half of the methods take about half of their previous verification time, and 90% verify at least as fast as VeriBetrKV-DF. Not only do most methods see a reduction in verification time, the methods also package more inlined proofs and code, eliminating the boilerplate of inter-method signatures.

To the right of $x = 100\%$, we see eight methods that verify slower in VeriBetrKV-LT. Three of the eight are within 1 second of their VeriBetrKV-DF counterparts. Two inline additional proofs

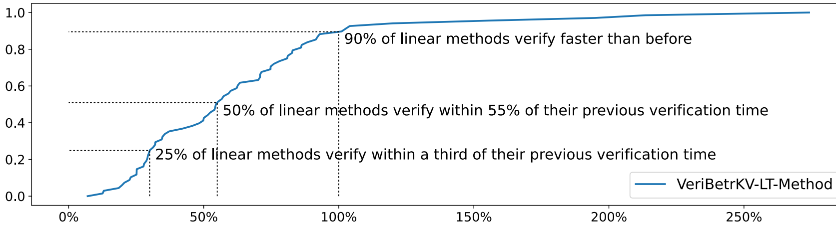

Fig. 9. Cumulative distribution of VeriBetrKV-LT methods' verification time relative to their counterparts in VeriBetrKV-DF for methods taking at least 5s. The vast majority have $x \leq 100\%$, indicating that they verify faster in VeriBetrKV-LT.

and are within 2–4s of their counterparts. The remaining three methods increase by 10–30s and warrant further discussion.

The first outlier increases from 7s to 20s due to inlined correctness proofs; if we add in the time to verify the correctness lemma, VeriBetrKV-DF takes 19s to verify 74 lines of code and proof, while VeriBetrKV-LT takes 20s to verify 28 lines of code and proof. The other two outliers follow the same pattern: One goes from 26s for 77 lines to 37s for 33 lines; the other from 24s for 104 lines to 33s for 33 lines. In every case, the VeriBetrKV-LT implementation trades off verification time to achieve a lower proof burden by inlining proofs.

Above are examples where the developer converting VeriBetrKV-LT collapsed multiple methods into a single method. Some methods collapse model proof into implementation methods. These improvements in proof burden suggest that the analysis understates the available improvement to verification time.

3.6 Decipherable Error Messages

Short verification times help developers by providing quick iterations. Accurate and decipherable verifier error messages help developers by reducing the number of iterations, and by precisely indicating what remediation is required. The verifier can fail this goal two ways (Section 3.6.1): by timing out and providing no error message, or by providing misleading error messages that suggest a logic error where a memory error is responsible. Linear types address both issues: they reduce solver complexity (Section 3.5), and they separate memory errors from logic errors (Section 3.6.2).

3.6.1 Causes of Poor Error Messages. Solver timeouts provide no useful error message, and often arise from complex disjointness invariants. For data structures that hold a dynamic number of heap objects, when heap invariants are insufficiently strong, we frequently observe the SMT solver timing out while repeatedly and ineffectually attempting to instantiate the necessary quantifiers.

For example, VeriBetrKV-DF's `MutCache` class contains a map from `Reference` to `Node` and a ghost `Repr` set that tracks all heap-allocated objects the cache holds: all heap-allocated `Node`s plus the `MutCache` object itself. The methods of `MutCache` export postconditions that are used for the implementation layer and for refinement. If we try to verify one such method when `MutCache`'s heap-related invariants are too weak, the verifier remains inconclusive for over 60 seconds.

Analysis using a custom SMT profiler identifies repeated quantifier instantiations of Dafny axioms related to the heap, which in turn point to a missing invariant related to the disjointness of all entries in the `Repr` set. However, the only information we immediately receive from the timeout is that the solver is likely failing to prove the postcondition, which at first sight may incorrectly appear to be related to a logic failure or an insufficient proof.

Indeed, memory errors emitted by the solver often masquerade as logic errors. When heap invariants or memory reasoning proofs are incomplete, we often observe Dafny report assertion failures that appear unrelated to memory reasoning. For example, we encountered this with VeriBetrKV-DF's `MutBucket`, a data structure that uses a B-tree to store keys and messages. `MutBucket` tracks the root node of its B-tree, a `Repr` set containing itself and the content of the B-tree, and `Bucket`, a high-level representation of the `MutBucket` for refinement purposes.

```
Class MutBucket {                                                                    1
  var tree: BTreeNode; ghost var Repr: set<object>; ghost var Bucket: Bucket;        2
  predicate Inv() reads this, Repr { /* ... */ }                                     3
  method Insert(key: Key, value: Message)                                            4
    requires Inv()                                                                   5
    modifies Repr                                                                    6
    ensures Inv() && Bucket == old(Bucket[key := value]) && /* ... */ {              7
      tree := BTree.Insert(tree, key, value); // modifies tree.Repr                  8
      assert Bucket == old(Bucket[key := value]); // Bucket has the new key-value    9
      // ...                                                                         10
```

When verifying `Insert`, Dafny reports errors related to parts of `Inv` that enforce logical invariants (seemingly unrelated to memory reasoning), and an assertion failure indicating the bucket does not have the correct content (line 9). At first glance, these failures seem related to possibly incompatible postconditions for the `BTree.Insert` method. However, the root cause is actually a missing heap invariant that ensures the disjointness between the `MutBucket` object itself and the nodes in the tree. Without this, in Dafny's dynamic frames model, updating fields in `MutBucket` could in theory corrupt the tree.

Assertion failures like this appear related to incorrect program logic, and mislead the developer who does not immediately consider broken heap reasoning.

3.6.2 Linear Types Provide Unambiguous Errors. In contrast to the experiences above, when we convert code to linear types, the verifier's diagnostics point directly to the root cause and directly guide resolution. We demonstrate this with `LMutCache`, a cache with the same functionality as `MutCache`. It uses a linear hash map as a data store, and it does not need a `Repr` set or any heap invariant, as the disjointness between objects is already captured in the type system.

```
linear datatype LMutCache = LMutCache(content: LinearHashMap<Reference, Node>)
```

The heap invariant that was missing in Section 3.6.1 is implicitly enforced by having the linear `LMutCache` and a linear `LinearHashMap` which holds linear nodes. We demonstrate other common memory reasoning errors in the `Insert` member method of `LMutCache`:

```
linear inout method Insert(ref: Reference, linear node: Node)                        1
  requires Inv() && /* ... */                                                        2
  ensures Inv() && content[ref] == node && /* ... */ {                               3
  shared var replaced :=                                                             4
      LinearHashMap.Insert(inout content, ref, node); // error (1)                   5
  node.FreeNode(); // error (2)                                                      6
  // ...                                                                             7
```

`LinearHashMap.Insert` takes in a linear `Node` that will be inserted at `ref` and returns a linear handle to the `Node` previously associated with `ref`. Line 4 tries to assign this result to a shared variable `replaced`, but this would leak the associated allocation, as there would be no owning linear variable for the returned data.

This error is immediately caught by the type checker with two error messages: *line 4: variable «replaced» must be linear* and *linear variable* [the linear data returned by `Insert`] *must be unavailable at end of block*, addressing both the incorrect usage and the memory leak.

With the previous error fixed, line 6 incorrectly tries to free `node` instead of `replaced`, even though `LinearHashMap.Insert` already took ownership of `node` on line 5. If undetected, there would now be two aliased variables referencing the same linear data: the linear variable `node` and the value for `ref` in the `content` map. The type checker identifies this immediately and provides a clear error message: *linear variable «node» is unavailable here*. This prevents the aliasing and ensures that `node` is not freed while referenced by `content`.

Both errors are generated in seconds (after type checking and before SMT solving), point to the exact line causing the issue, and have immediately actionable messages.

3.7 Limitations in Linear Dafny

Linear Dafny has various limitations in its expressivity when compared to an advanced language with linear types like Rust [Klabnik et al. 2018]. Although the conversion went smoothly overall, we ran into some of these limitations when porting the cache module.

The cache module is responsible for storing in-memory nodes of the B$^{\epsilon}$-tree, providing clients with basic operations like `get`, `insert`, `remove`, and `replace`. In VeriBetrKV-DF, the clients of the cache would get a reference to the node of interest and operate directly on it. In VeriBetrKV-LT, since the cache module is linearized, clients of the cache get an `inout` reference to it. Linear Dafny does not allow calling methods that borrow the `inout` cache to obtain a mutable or shared reference to a node when the reference is then stored in a variable. Returning a mutable reference is currently unsupported, and while it is possible to return shared references, as described in Section 2, their use is limited because Linear Dafny has to restrict their scope to ensure that no copies survive after the borrow. As a result, we added pass-through methods inside the cache to perform node operations on behalf of its clients.

An alternative solution is to remove a node from the cache for client uses, and the client can insert it back in afterwards. What complicates this is that the overall system state changes when the cache is modified. For example, one of our system's state invariants requires that the cache content is consistent with the Least Recently Used (LRU) queue. Removing nodes from the cache would cause inconsistency in the LRU queue and violate the system invariant, which clients often need for their operations. If instead we update both the LRU queue and the cache, we will then need to prove that previously proven properties of the system are unaffected by these changes and do so for all clients.

Another solution for clients is to manually delimit the shared borrow scope by adding an extra method that takes the cache as a shared reference. Within this new method, the client can perform cache operations using the shared reference to the cache and can safely retrieve shared references to nodes. This would have partly relieved the need for pass-through methods but it would also introduce bloat in the client due to redundant function specifications.

One other limitation we encountered was that Linear Dafny does not support storing shared references in datatypes. In VeriBetrKV-DF, iterators for successor queries maintain a reference to the nodes they iterate over. In VeriBetrKV-LT, these iterators are only able to store a non-linear (ordinary) data-structure that mirrors the data in the underlying node. It is often the case that this immutable representation needs to be constructed from the underlying linear node just to be able to store it in the iterator. If Linear Dafny supported storing shared references, the iterators could directly access the node instead.

Rust [Klabnik et al. 2018] has a sophisticated "borrow checker" that tracks the relationship of lifetime variables associated with shared and mutable borrows. Thanks to the borrow checker Rust

supports returning shared and mutable references from methods, and storing them in variables and datatypes. Modern Rust also allows non-lexical lifetimes and does not require borrows to start and end at the boundary of lexical scopes. If we similarly enriched Linear Dafny with lifetime variables and reasoning, and added support for these features, we would be able to remove the pass-through methods in the cache module and reduce its size by 55% (242 lines of code) and could avoid unnecessarily copying data in successor queries.

4 RELATED WORK

Inspired by Girard's linear logic [Girard 1987], various linear type systems have provided different mechanisms to express whether a variable is linear or nonlinear, and have provided different rules for how linear and nonlinear variables interact. Linear Dafny uses a very direct and strict approach: the annotation `linear` declares a variable linear, and linear variables are strictly segregated from ordinary variables, so that programs cannot assign one to the other. (This direct approach was also taken by CIVL [Hawblitzel et al. 2015b], although CIVL did not support borrowing.) This allows Linear Dafny to support both linear and nonlinear usages of the same type, in contrast to other systems like Cogent [Amani et al. 2016] and Rust [Klabnik et al. 2018; Matsakis and Klock 2014]. For example, a `seq<T>` type in an ordinary variable may be duplicated, but not updated in place, but a `seq<T>` type in a linear variable may be updated in place. Both usages of `seq<T>` are encoded identically in the SMT solver and may be used as ghost `seq<T>` values in the same way.

Wadler [Wadler 1990] distinguished linear and nonlinear variables based on the types of the variables. Cogent [Amani et al. 2016] and Rust [Klabnik et al. 2018; Matsakis and Klock 2014] follow this approach, where Cogent uses kinds to distinguish linear types and Rust uses a trait named `Copy` to mark nonlinear types. Linear Haskell follows Girard's original approach, where parameters of function types `A -o B` are linear and parameters of function types `A -> B` are nonlinear. In contrast to Linear Dafny, Cogent, and Rust, nonlinear variables may be assigned to linear variables in Linear Haskell, which makes the type system more flexible but does not guarantee that all linear variables are unaliased, which means that in-place mutation is not safe for types that may be used non-linearly, such as the Haskell list type. Rust and Cogent support borrowing, while Linear Haskell does not. Rust's `&` and `&mut` borrowing correspond roughly to Linear Dafny's `shared` and `inout`, although `inout` does not yet support all the abilities of `&mut`, and Linear Dafny does not yet support Rust's lifetime variables (`&'a` and `&'a mut`), as discussed in Section 3.7.

Cogent supports verification using an interactive theorem prover. Although Rust does not have verification support built in, the Prusti tool [Astrauskas et al. 2019] leverages SMT solving and Rust's advanced linear type system to support verification of Rust code. Prusti encodes Rust's ownership and borrowing information as memory constraints in the verifier's logic, alleviating the need for user-provided memory invariants in the surface syntax, similar to Linear Dafny. This encoding relies on the separation logic features (permissions, magic wand) of the Viper [Müller et al. 2016] verification infrastructure, and is different from Linear Dafny's more direct encoding in Boogie of linear code as mathematical values and expressions. Prusti also supports Rust's more advanced borrowing and partial ownership transfer features, including the ability to return mutable references and store them in variables, which could have been useful to address the difficulties discussed in Section 3.7. Prusti introduces "pledges": a specification construct for functions that return a mutable reference obtained by mutably borrowing one or more of the parameters. Pledges allow a function to specify properties of the borrowed values passed as arguments when the borrow expires. A similar construct would be necessary for Linear Dafny to support returning mutable references.

Both Cogent and Prusti have strong support for verifying linear data structures and borrowed data structures, but have less support for nonlinear data structures like doubly linked lists. We

believe that our novel region-based approach to nonlinear data structures would be applicable to both Cogent and Rust, and perhaps could provide a way to manage Rust's `RefCell` type without requiring reasoning about a global heap.

RustBelt [Jung et al. 2017] can be used to formally reason about unsafe Rust code, which is a subtle and difficult task. Our hope is that by providing an SMT-based verification system with strong support for nonlinear data structures, we can avoid the need for unsafe code in many cases. For example, Linear Dafny can verify doubly linked lists and encapsulate doubly linked lists inside linear data, without requiring extra run-time checks and without relying on unsafe code.

Walker and Watkins [Walker and Watkins 2001] and Fluet, Morrisett, and Ahmed [Fluet et al. 2006] use linear type systems to manage regions. These systems still require explicit region variables, along with universal and existential quantification over those variables, which our approach avoids by relying on SMT solving.

GhostCell [Yanovski et al. 2021] uses linear ghost tokens to apply a single permission to multiple objects, such as all of the nodes in a doubly linked list. This is similar to our use of regions to control access to multiple nodes of a data structure, but with some differences. First, GhostCell ensures just type safety, whereas our approach addresses correctness, which means we have to deal with postconditions, invariants, and modifies clauses. Second, GhostCell builds on Rust's lifetime variables, which is both an advantage and a disadvantage: an advantage because Rust already implements the necessary type checking rules for lifetime variables, but a disadvantage because Rust limits what programs can do with lifetime variables. In particular, the doubly linked list example in [Yanovski et al. 2021] has to expose the GhostToken lifetime variables to the clients of the doubly linked list, since Rust currently lacks existential lifetime variable quantification, whereas our Linear Dafny doubly linked list can hide the use of regions from clients.

Systems like Viper [Müller et al. 2016], SLAyer [Berdine et al. 2011], and Steel [Fromherz et al. 2021; Swamy et al. 2020] combine separation logic with SMT solving. Separation logic can express nonlinear data structures like doubly linked lists more easily than linear type systems can. Viper, SLAyer, and Steel all rely on heuristics to manipulate separation logic formulas and to manage the heap, since the SMT solver does not handle these tasks automatically. By contrast, our approach uses a linear type system, with a complete checker, to check linearity automatically. When pure linearity is insufficient, Linear Dafny programs can still fall back on nonlinear datatypes via modifies clauses and regions.

5 CONCLUSION

Linear types make a valuable complement to an SMT-based verification engine by streamlining common-case memory reasoning. We present the design, formalization, and soundness proof for a hybrid linear type system extension to Dafny that enables explicit dynamic frame reasoning and linear type memory reasoning to symbiotically coexist. Conversion of a large-scale verified systems project from explicit dynamic frame reasoning to mostly-linear reasoning provides a straight-across comparison showing a 27% reduction in proof burden, faster verification of most methods, and clearer error messages.

ACKNOWLEDGMENTS

We would like to thank Oded Padon and the anonymous reviewers for many suggestions that improved the paper. This work is partially supported by the National Science Foundation grant DGE-1762114. Work at CMU was supported, in part, by a gift from VMware, a Google Faculty Fellowship, the Alfred P. Sloan Foundation, and the Intel Corporation. Andrea Lattuada is supported by a Google PhD Fellowship.

REFERENCES

Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. 2016. Cogent: Verifying High-Assurance File System Implementations. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 147 (Oct. 2019), 30 pages. https://doi.org/10.1145/3360573

M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. 2006. Boogie: A modular reusable verifier for object-oriented programs. *Proceedings of Formal Methods for Components and Objects (FMCO)* (2006). https://doi.org/10.1007/11804192_17

Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. 2015. An Introduction to B^{ϵ}-trees and Write-Optimization. *;login: The USENIX Magazine* 40, 5 (Oct. 2015), 22–28.

Josh Berdine, Byron Cook, and Samin Ishtiaq. 2011. SLAyer: Memory Safety for Systems-level Code. In *Conference on Computer Aided Verification (CAV)*.

Ankit Bhardwaj, Chinmay Kulkarni, Reto Achermann, Irina Calciu, Sanidhya Kashyap, Ryan Stutsman, Amy Tai, and Gerd Zellweger. 2021. NrOS: Effective Replication and Sharing in an Operating System. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 295–312. https://www.usenix.org/conference/osdi21/presentation/bhardwaj

Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. 2020. Theseus: an Experiment in Operating System Structure and State Management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1–19. https://www.usenix.org/conference/osdi20/presentation/boos

L. M. de Moura and N. Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems*. https://doi.org/10.1007/978-3-540-78800-3_24

M. Fluet, G. Morrisett, and A. Ahmed. 2006. Linear Regions Are All You Need. In *European Symposium on Programming (ESOP)*. https://doi.org/10.1007/11693024_2

Aymeric Fromherz, Aseem Rastogi, Nikhil Swamy, Sydney Gibson, Guido Martínez, Denis Merigoux, and Tahina Ramananandro. 2021. Steel: Proof-oriented Programming in a Dependently Typed Concurrent Separation Logic. https://www.fstar-lang.org/papers/steel/ In submission.

Jean-Yves Girard. 1987. Linear Logic. In *Theoretical Computer Science*. https://doi.org/10.1016/0304-3975(87)90045-4

D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. 2002. Region-based memory management in Cyclone. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/512529.512563

Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. 2020a. Storage Systems are Distributed Systems (So Verify Them That Way!). In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 99–115. https://www.usenix.org/conference/osdi20/presentation/hance

Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. 2020b. veribetrkv-osdi2020. https://github.com/secure-foundations/veribetrkv-osdi2020/tree/master/docker-hdd/src/veribetrkv-dynamic-frames

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015a. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. https://doi.org/10.1145/2815400.2815428

Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 165–181. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/hawblitzel

Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. 2015b. Automated and Modular Refinement Reasoning for Concurrent Programs. In *Proceedings of the Conference on Computer Aided Verification (CAV)*. https://doi.org/10.1007/978-3-319-21668-3_26

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages. https://doi.org/10.1145/3158154

Ioannis T. Kassios. 2006. Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In *FM 2006: Formal Methods*.

Steve Klabnik, Carol Nichols, and Rust Community. 2018. The Rust Programming Language https://doc.rust-lang.org/book/.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive Formal Verification of an OS Microkernel. *ACM Trans. Comput. Syst.* 32, 1, Article 2 (Feb. 2014), 70 pages.

https://doi.org/10.1145/2560537

Leslie Lamport. 2002. *Specifying Systems: The TLA+ Languange and Tools for Hardware and Software Engineers.* Addison-Wesley.

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR).* https://doi.org/10.1007/978-3-642-17511-4_20

Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology* (Portland, Oregon, USA) *(HILT '14).* Association for Computing Machinery, New York, NY, USA, 103–104. https://doi.org/10.1145/2663171.2663188

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI).*

Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. 2020. RedLeaf: Isolation and Communication in a Safe Operating System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20).* USENIX Association, 21–39. https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science.* https://doi.org/10.1109/LICS.2002.1029817

Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. 2020. SteelCore: An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs. In *25th ACM SIGPLAN International Conference on Functional Programming (ICFP).* https://doi.org/10.1145/3409003

Mads Tofte and Jean-Pierre Talpin. 1994. Implementing the call-by-value lambda-calculus using a stack of regions. In *Principles of Programming Languages (POPL).* https://doi.org/10.1145/174675.177855

Philip Wadler. 1990. Linear Types Can Change the World!. In *Proceedings of the IFIP TC 2 Working Conference on Programming Concepts and Methods.*

David Walker and Kevin Watkins. 2001. On regions and linear types. In *International Converence on Functional Programming (ICFP).* https://doi.org/10.1145/507635.507658

Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. 2019. Oxide: The Essence of Rust. *CoRR* abs/1903.00982 (2019). https://doi.org/10.48550/ARXIV.1903.00982 arXiv:1903.00982

Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. 2021. GhostCell: Separating Permissions from Data in Rust. *Proc. ACM Program. Lang.* 5, ICFP, Article 92 (aug 2021), 30 pages. https://doi.org/10.1145/3473597