# Adoption and Focus: Practical Linear Types for Imperative Programming

Manuel Fähndrich  Robert DeLine
Microsoft Research
One Microsoft Way
Redmond, WA 98052-6399
{maf,rdeline}@microsoft.com

## ABSTRACT

A type system with linearity is useful for checking software protocols and resource management at compile time. Linearity provides powerful reasoning about state changes, but at the price of restrictions on aliasing. The hard division between linear and nonlinear types forces the programmer to make a trade-off between checking a protocol on an object and aliasing the object. Most onerous is the restriction that any type with a linear component must itself be linear. Because of this, checking a protocol on an object imposes aliasing restrictions on any data structure that directly or indirectly points to the object. We propose a new type system that reduces these restrictions with the adoption and focus constructs. Adoption safely allows a programmer to alias objects on which she is checking protocols, and focus allows the reverse. A programmer can alias data structures that point to linear objects and use focus for safe access to those objects. We discuss how we implemented these ideas in the Vault programming language.

## Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; F.3.3 [**Logics and Meaning of Programs**]: Studies of Program Constructs—*Type structure*; F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs—*Pre- and post-conditions, Specification techniques*; D.3.4 [**Programming Languages**]: Processors—*Memory management*

## General Terms

Design, Languages, Reliability, Verification

## Keywords

Linear types, heap aliasing, region-based memory management

## 1. INTRODUCTION

Successful use of a software component often requires more than just familiarity with the types of the functions and data in the component's interface. Rules governing proper interaction with that interface must be gleaned from the component's documentation or, as is often the case, learned from local folklore. These rules, which we call the *interface protocol*, govern the order in which the interface's functions may be called and its data accessed [14]. As a familiar example, a file system's interface protocol typically has the following rules: a file must be opened before it is read or written; a file may be read or written until it is closed; and every file that is opened must eventually be closed. In the context of the Vault programming language, we studied a type system that tracks the lifetime and symbolic state of objects. To enforce our file system protocol in Vault, we give the file type the states "open" and "closed" and specify that the read and write functions expect an "open" file and that the close function changes a file from state "open" to "closed."

Checking the states associated with objects requires the ability to tell different objects apart. For instance, the code sequence `close(a);read(b)` obeys our interface protocol if `a` and `b` refer to different files, but is incorrect if these variables alias the same file. To solve this problem, Vault's type system splits the program's values into two groups: those on which we can check protocols, but to which aliasing restrictions apply (values of linear type); and those on which we cannot check protocols, but which are free of aliasing restrictions (values of nonlinear type). Although this distinction is necessary for tractability, the trade-off between protocol checking and aliasing is an annoyance to programmers.

The division between linear and nonlinear types presents one further annoyance. A linear type system typically restricts how a programmer must design her data structures by forbidding a nonlinear type from having linear components. For instance, if `a` and `b` were nonlinear records whose `f` fields refer to linear files, then the code sequence `close(a.f);read(b.f)` could be unsafe, since `a` and `b` could be aliases due to their nonlinear types. To prevent such safety violations, a linear type system restricts aliasing not only to those objects on which we check protocols, but also to any object which directly or indirectly refers to them.

Because of these restrictions, checking protocols on more than a few types of objects in a large program is impractical. The aliasing restrictions quickly spread to all of the program's data structures, including those, like graphs and caches, which inherently involve aliasing.

To make protocol checking more practical, this paper presents a new type system that removes these restrictions. Our contributions are: 1) We allow linear components in nonlinear containers, but control access to ensure safety. 2) We introduce the adoption construct, which safely allows aliasing of objects on which we check protocols and those that refer to them. 3) We introduce the focus operator, which provides a temporary scope in which we can check a protocol on an aliased object.

Together, these features allow us to check Vault code like the following[1]:

```
struct fileptr { tracked(@open) file f; };

void reset_logs($G:fileptr msgs, $G:fileptr errs)
{
  tracked fileptr log = msgs;
  close(log.f);
  log.f = open(MSG_LOG_FILENAME);

  log = errs;
  close(log.f);
  log.f = open(ERR_LOG_FILENAME);
}
```

Previous linear type systems would reject both the declaration of the nonlinear type `fileptr` due to its linear (`tracked`) field `f` and the code in `reset_logs` due to the potential aliasing between `msgs` and `errs`. Our type system's acceptance of this code demonstrates the focus operation. We automatically infer one focus operation around the first three statements and a second focus operation around the last three statements. The first focus temporarily gives `log`, an alias for `msgs`, a linear type so that its field `f` may be updated. For safety, in the scope of this focus, access to any potential alias of `msgs` (namely `errs`) is illegal. The potential aliases are known due to the presence of *guards* on nonlinear types. The same guard `$G` means `errs` and `msgs` could be aliases. Through the use of guards, focus prevents access to potential aliases. The second focus works similarly. The adoption operation (not shown) allows a `fileptr` object to be allocated at a linear type (to initialize its field), then given a nonlinear type to allow it to be aliased.

## 2. OVERVIEW

We present the details of our techniques in Sections 3 and 4 through a small expression language and a type system. Section 6 discusses extensions and Section 7 describes how the presented ideas are realized and applied in Vault. Section 8 discusses related work.

[1]In Vault, structs are reference types, i.e., pointers.

This section introduces the four concepts we use to relax the boundary between linear and nonlinear types, provides an overview of our techniques, and an example used throughout the paper.

**A unified view of object allocation and deallocation.** In contrast to previous work in which heap objects are either linear or nonlinear, all objects in our model are allocated at linear type and deallocated at linear type. This model is intuitive, since at object creation, there cannot be aliases for the object. Similarly, when an object is freed, all aliases to the object must be dead, if we are to avoid accesses through dangling references.

**Adoption.** Since all objects start out with linear type and since it is impractical to program without aliasing, we need a way to obtain aliases (of nonlinear type) to existing objects. We propose the adoption construct, `adopt` $e_1$ `by` $e_2$, which takes an adoptee $o_1$ (the result of $e_1$) and an adopter $o_2$ (the result of $e_2$), both of linear type. The construct consumes the linear reference to $o_1$ by creating an internal reference from $o_2$ to $o_1$. Therefore, each adoptee has exactly one adopter. The result of the adoption expression is a reference at nonlinear type to the adoptee. This adoptee's nonlinear type is its previous linear type with only the top-level type constructor changed from linear to nonlinear.

The nonlinear reference to the adoptee is valid for the lifetime of the adopter, since we disallow access to $o_1$ through the internal reference of the adopter. Any linear components of object $o_1$ cannot be directly accessed through the nonlinear reference. To do so would lead to uncontrolled shared access to objects of linear type. The linear components of $o_1$ may however be accessed in the scope of a `focus` operation (discussed below).

When the adopter is freed, all nonlinear references to the adopted object become inaccessible, using the mechanism described next. The adopter returns its internal linear reference to the adoptee, thereby reinstating the adoptee's linear type. Through multiple adoption expressions, an adopter object can adopt any number of other objects.

**Access control for nonlinear references.** Because our adoption construct allows temporary aliases to a linear object, our type system must ensure that these aliases become invalid when the object recovers its linear type when it is unadopted. Otherwise, aliases could unsafely witness changes to the object made through the linear type. In our adoption approach, an adoptee recovers its linear type when its adopter is freed. Hence, our approach to invalidating aliases is to tie the lifetime of the aliases to the lifetime of the adopter.

Our solution builds on techniques developed by Crary *et al.* to deal with explicit region deallocation in the Capability Calculus [2], as well as the work on alias types by Walker *et al.* [18]. In this paper, we use a superscripted dot to denote linear types and the absence of a dot to denote nonlinear types. Since our desired lifetime correlation is on a per-object basis, we first give static names to objects of linear type $\tau^\bullet$ so that we can tell them apart. In our type system, an object of linear type is given the singleton type $\mathtt{tr}(\rho)$ for some static name $\rho$. We call the static names *keys* and the singleton types *tracked types.* At each point in the program, our type system maintains the keys of those tracked objects that are allocated before the given program point and deallocated after that point. We call this set of keys the *capabilities.* Each entry $\{\rho \mapsto \tau\}$ in the capabilities means that the object tracked by key $\rho$ has type $\tau$ and is alive at the given program point. In short, a tracked object's allocation lifetime is the set of program points for which its key is in the capabilities.

To correlate a nonlinear reference to an allocation lifetime we use a *guarded type* of the form $\rho \triangleright \tau$. An object of guarded type may be used in an expression requiring a value of type $\tau$ at any program point where key $\rho$ is in the capabilities. Guarded types are nonlinear and can be shared freely. When an object of type $\mathtt{tr}(\rho)$ adopts an object with linear type $\tau^\bullet$, the result of the adoption has guarded type $\rho \triangleright \tau$. When the adopter is freed, the key $\rho$ is removed from the capabilities, and thereafter all references of type $\rho \triangleright \tau$ are invalid.

**Focus.** The final and most important novel concept is a construct we call `focus`, which temporarily provides a linear view on an object of nonlinear type. The insight is that we can violate any type invariant of a nonlinear object (including its very existence), as long as no alias for the object can witness the violation.

The `focus` construct, $\mathtt{let}\, x = \mathtt{focus}\, e_1 \,\mathtt{in}\, e_2$, requires $e_1$ to evaluate to an object of a guarded type $\rho_1 \triangleright \tau$. This object is called the focused object. We change the context in which $e_2$ is type-checked in two ways. First, we bind `x` to the focused object, but we give `x` a tracked type $\mathtt{tr}(\rho)$ for some fresh key $\rho$ and add $\{\rho \mapsto \tau\}$ to the capabilities. Because of `x`'s tracked type, expression $e_2$ can change the underlying type associated with $\rho$, can remove $\rho$ from the capabilities, and can access and replace linear components of $\tau$.

Second, to ensure that no aliases can witness these changes, within the context of $e_2$, we remove the original guarding key $\rho_1$ from the capabilities. By temporarily revoking this key, we are guaranteed that no aliases of the focused object are accessible during $e_2$.

Finally, at the end of $e_2$, the capabilities must contain $\{\rho \mapsto \tau\}$, i.e., $\rho$ is live and the final type $\tau$ matches the initial type of the focus. This guarantees that no alias can witness effects on the focused object outside the scope of the `focus`. We end the focus by revoking the temporary key $\rho$ and granting $\rho_1$, thereby allowing renewed access to all objects guarded by $\rho_1$.

## 2.1 Motivating Example

To illustrate the benefits of our type system, we introduce a simple example here and discuss it throughout the paper. As we showed in previous work [3], checking state-based protocols, like our file example, requires the same typing machinery as checking memory safety. Namely, we use the same type system to check that an object is not referenced after deletion and all objects are eventually deleted. To keep our example simple, we use memory safety as our protocol.

```
fun dict_lookup(phone:int, ssn:int,
                d:dictionary) {
  ... let c = newCell(d) in ...
}

fun add_amount(cell:ref<int[]>, elem:int) {
  ... resize(cell, newsize) ...
}

fun add_entry(d:dictionary, phone:int,
              ssn:int, amount:int) {
  let cell = dict_lookup(phone, ssn, d)
  in
  add_amount(cell, amount)
}
```

**Figure 1: Pseudo code for our motivating example.**

Our example's central data structure is a resizable array. Consider a program that builds a dictionary to hold sequences of monetary amounts, which can be indexed by a person's Social Security Number and phone number, as sketched in Figure 1. The program uses the function `add_entry` to add a new amount to a person's entry in the dictionary. The program stores each person's data in a resizable array, which grows when it is full. The dictionary contains two references to each array, one indexed by the Social Security Number and the other by the phone number.

The natural representation for a resizable array is a mutable cell containing the array (`ref<int[]>`). Since we want to be able to free the array during resize operations, the array should have linear type, but the cell itself is shared and thus should have nonlinear type. In our pseudo code, we assume that the function `dict_lookup` allocates a fresh, but sharable entry via a call to `newCell`, if the entry is not found. Similarly, the function `add_amount` calls `resize` when the array is full. We will show later in detail how the implementation of `newCell` and `resize` are type checked.

The example is ill suited to previous type systems with linearity. In a traditional linear type system, each cell would need to be given the nonlinear type $\mathtt{ref}\langle\mathtt{int}[\,]\rangle$ in order to be shared in the dictionary. To support the destructive resize operation, we want the array to have a linear type. However, since a linear type cannot be a component of a nonlinear type, we must also give the cell a linear type: $\mathtt{ref}\langle\mathtt{int}[\,]^\bullet\rangle^\bullet$. We cannot assign the cells the incompatible types $\mathtt{ref}\langle\mathtt{int}[\,]\rangle$ and $\mathtt{ref}\langle\mathtt{int}[\,]^\bullet\rangle^\bullet$.

In his seminal paper on linear types [17], Wadler began to soften the boundary between linear and nonlinear types. He used a `let!` construct to give a temporary nonlinear type to a linear object. The construct $\mathtt{let!}\;(x)\; y = e_1 \;\mathtt{in}\; e_2$ recursively changes the variable $x$'s linear type to nonlinear form for the scope of the expression $e_1$, then returns it to its linear type for the scope of expression $e_2$. Wadler used a syntactic restriction on the type of $e_1$ to prevent the escape of nonlinear references[17]. Odersky later proposed observer types to prevent this escape [9].

Could we give our cell the linear type $\mathtt{ref}\langle\mathtt{int}[\,]^\bullet\rangle^\bullet$ and then use Wadler's `let!` to obtain a reference of nonlinear type to the cell? Unfortunately, there are two issues that make `let!` unsuitable in this case.

First, while `let!` is sound in its original setting of a functional language, it is unsound if applied to types of mutable storage. As a case in point, applying `let!` to view a cell of linear type $\mathtt{ref}\langle\mathtt{int}[\,]^\bullet\rangle^\bullet$ at nonlinear type $\mathtt{ref}\langle\mathtt{int}[\,]\rangle$, allows us to unsafely update the cell with a shared array, as shown in the following example:

```
fun crash (a:int[], r:ref⟨int[]•⟩•) {
  let! (r) _ = (r := a)
  in free(r); return sub(a,0);
}
```

Here, `let!` gives `r` the type $\mathtt{ref}\langle\mathtt{int}[\,]\rangle$ for the scope of the assignment `r := a`. The assignment overwrites the unique array pointer stored in the cell, thus creating a leak. Furthermore, the subsequent free of the cell also frees the array `a`, which causes the index operation `sub(a,0)` to access freed memory.

The second problem with `let!` is its reliance on a lexical scope, which does not allow the implementation of `newCell`, where a fresh, but sharable object is created and returned to the context under a nonlinear type. In short, `let!` does not allow us to represent our dictionary.

The new type system presented in this paper removes the restriction that nonlinear types may not contain linear types as components and provides adoption as an alternative to `let!`. Adoption handles mutable types and does not rely on lexical scoping. For our example, we track the dictionary under key $\rho_d$ and adopt the cells to the dictionary. This gives the cells the type

$$\rho_d \triangleright \mathtt{ref}\langle\mathtt{int}[\,]^\bullet\rangle$$

which allows the cells to be shared (because they are nonlinear) and the arrays to support deallocation during resize (because they are linear).

Whenever the contained linear array needs to be accessed, a focus operation on the cell will temporarily revoke $\rho_d$, but provide a fresh key $\rho$ for the focused cell. It is then possible to access the contained linear array and to replace it in the resize operation. During the focus, no possible alias of the cell is accessible, since such access would require key $\rho_d$.

## 3. TYPES AND EXPRESSIONS

To illustrate how `adopt` and `focus` allow programs like our dictionary example to type check, this section presents a small type and expression language. Section 4 presents the typing rules.

### 3.1 Types

We use the type language in Figure 2 to formalize the ideas discussed in the previous sections. The type language distinguishes several kinds of types. Heap types $h$ represent mutable tuples and arrays that are stored in the heap. Objects of these types are adoptable and focusable. We represent the type $\mathtt{ref}\langle\mathtt{int}[\,]\rangle$ from the introduction as a 1-tuple of the form $\langle\mathtt{int}[\,]\rangle$.

| | | | |
|---|---|---|---|
| heap type | $h$ | $::=$ | $\langle\sigma_1..\sigma_n\rangle \mid \tau[\,]$ |
| type | $\tau$ | $::=$ | $\mathtt{int} \mid \mathtt{tr}(\rho) \mid G \triangleright h$ |
| | | $\mid$ | $\forall[\Delta].(C_1,\sigma_1) \to (C_2,\sigma_2)$ |
| linear type | $\sigma$ | $::=$ | $\exists[\rho \mid \{\rho \mapsto h\}].\,\mathtt{tr}(\rho) \mid \tau$ |
| guard | $G$ | $::=$ | $\rho$ |
| capability | $C$ | $::=$ | $\cdot \mid \{\rho \mapsto h\}\otimes C \mid \epsilon\otimes C$ |
| type context | $\Delta$ | $::=$ | $\cdot \mid \rho,\Delta \mid \epsilon,\Delta$ |

**Figure 2: Type language**

Ordinary types $\tau$ include integers, tracked types $\mathtt{tr}(\rho)$, and guarded types $G \triangleright h$. Tracked types are singleton types $\mathtt{tr}(\rho)$, where the key $\rho$ is a static name for the address of the object being tracked. Hence, two objects with the same key $\rho$ are stored at the same address in the heap and are hence aliases for the same memory block. The contents of the memory at $\rho$ is of heap type, which is given separately by a capability of the form $\{\rho \mapsto h\}$. Note that guarded types always refer to a heap type.

In this type system, we separate the handle to a linear object, $\mathtt{tr}(\rho)$, from the capability to access that object, $\{\rho \mapsto h\}$. A traditional linear type system couples these two in the linear type $h^\bullet$. To express such a linear type $\sigma$, the handle and its capability are combined in an existential package [18]:

$$h^\bullet = \exists[\rho \mid \{\rho \mapsto h\}].\,\mathtt{tr}(\rho)$$

When storing a linear object in a data structure, the object's particular address is not important, so long as we have the capability to access the object at that address. This use of an existential type binds the address to its capability and makes the address "anonymous." Whereas the universal quantification for function types captures the intuition that functions operate *uniformly* over objects at different addresses, the existential quantification for linear types captures the intuition that a data structure is stored *some particular way* in the heap, although exactly how is irrelevant.

To convert from a linear type $h^\bullet$ to an ordinary tracked type $\mathtt{tr}(\rho)$, the capability is unpacked and added to the capabilities at the current program point. Similarly, to convert from a tracked type $\mathtt{tr}(\rho)$ to a linear type $h^\bullet$, we extract the capability $\{\rho \mapsto h\}$ from the current capabilities. These conversions are captured by the relation $C_1;\sigma_1 \vdash C_2;\sigma_2$ shown in the Appendix.

The spatial conjunction of capabilities is formed via $C_1 \otimes C_2$, expressing that keys in $C_1$ are disjoint from keys in $C_2$.

Function types have the form $\forall[\Delta].(C_1,\sigma_1) \to (C_2,\sigma_2)$, where $\sigma_1$ is the argument type, and $\sigma_2$ the result type. $C_1$ is the "before" capability needed to call the function, and $C_2$ is the "after" capability provided at return. A function that operates only on a heap object stored at a particular address $\rho$ would not be very useful. To avoid this problem, we make functions reusable with parametric polymorphism. The type context $\Delta$ allows a function to abstract over names for keys and capabilities.

$$\begin{array}{lll} e & ::= & x \mid i \mid e.i \mid e.i := e \mid e(e) \mid e[c] \\ & \mid & \texttt{new}\langle i \rangle \mid \texttt{free}\, e \mid \texttt{adopt}\, e{:}h\, \texttt{by}\, e \\ & \mid & \texttt{let}\, x = e\, \texttt{in}\, e \\ & \mid & \texttt{let}\, x = \texttt{focus}\, e\, \texttt{in}\, e \\ & \mid & \texttt{fun}\ f[\Delta](x:\sigma) : \sigma\, \texttt{pre}\ C\, \texttt{post}\ C\ \{e\} \\ \\ c & ::= & \rho \mid C \mid G \end{array}$$

**Figure 3: Expressions**

### 3.2 Expressions

Figure 3 presents a small call-by-value expression language for manipulating integers $i$, updatable tuples, and functions. Expression $e.i$ is used to obtain the $i$th component of a tuple, $e_1.i := e_2$ updates the $i$th component of $e_1$ to $e_2$. Function application of $e_1$ to $e_2$ is written $e_1(e_2)$. A tuple with $i$ components is allocated by expression $\texttt{new}\langle i \rangle$ and freed by expression $\texttt{free}\, e$. Adoption of $e_1$ by $e_2$ is written $\texttt{adopt}\, e_1{:}h\, \texttt{by}\, e_2$, where the type constraint $h$ allows the programmer to specify the form of the adopted type in order to pack particular fields to linear types. The expression $\texttt{let}\, x = e_1\, \texttt{in}\, e_2$ provides name binding with the usual lexical scoping rules. In our examples, we use the expression $e_1; e_2$ as sugar for $\texttt{let}\, \_ = e_1\, \texttt{in}\, e_2$. The focus expression $\texttt{let}\, x = \texttt{focus}\, e_1\, \texttt{in}\, e_2$ focuses on $e_1$ during $e_2$. At the end of $e_2$, the focus is lost.

Expression $\texttt{fun}\ f[\Delta](x{:}\sigma_1){:}\sigma_2\, \texttt{pre}\ C_b\, \texttt{post}\ C_a\ \{e\}$ defines a recursive function $f$, where $x$ is the formal parameter of type $\sigma_1$, and $e$ is the body of type $\sigma_2$. $C_b$ and $C_a$ are respectively the before and after capabilities, and $\Delta$ contains the universally quantified variables. Expression $e[c]$ is used to instantiate polymorphic functions to particular type arguments $c$ (capabilities or keys). In our examples, we will use multi-argument functions without formalizing this trivial extension.

To reduce the size of our language, we do not provide primitive expressions for creating, reading and updating arrays, but rather rely on predefined functions. Our examples only use `newarray(`$n$`)` which allocates a tracked array of size $n$.

### 3.3 Semantics

The operational semantics of the language is relatively straight-forward. Expressions of the form $e[c]$ are operationally equivalent to $e$, ie., the type instantiation has no runtime effect. Similarly, a let-focus expression is operationally equivalent to an ordinary let-expression. The only non-standard construct is adoption. A simple implementation uses two special fields per object: `adoptees`, and `next`. The `adoptees` field of an object $o$ is used to hold the list of objects adopted by $o$, which are chained via their `next` fields. Adoption of $o_1$ by $o_2$ operationally corresponds to

```
o1.next = o2.adoptees;
o2.adoptees = o1;
```

The typing rules guarantee that the structures formed through the `next` and `adoptees` fields are unshared trees. To simplify the language and types, we assume that `free` recursively frees all linear components and all object reachable via `next` and `adoptees` fields. In Section 4.3 we discuss alternative semantics to recover adoptees when adopters are freed.

## 4. TYPING RULES

We are now ready to show the details of how our typing discipline treats adoption and focus. Typing judgements have the form $\Delta; \Gamma; C_1 \vdash e : \tau; C_2$ and express that in type context $\Delta$ and type environment $\Gamma$ (mapping program variables to types $\tau$) and given initial capabilities $C_1$, expression $e$ evaluates to a value of type $\tau$ and final capabilities $C_2$. The capabilities are threaded throughout the expression evaluation and track the state (heap type) and liveness of all tracked objects.

Unlike in standard linear type systems, type environments $\Gamma$ map program variables to nonlinear types $\tau$ only. Linearity is enforced via the capabilities, not via environment splitting. Similarly, the type of an expression is always nonlinear.

The full set of typing rules is given in the Appendix. Here, we highlight the most interesting rules by illustrating them on our example from the introduction.

### 4.1 Adoption

Ideally, we would use tracked types for all objects in our program. However, because tracked types allow no sharing, we use nonlinear guarded types where needed. Objects of guarded type are obtained via *adoption.* An adoption involves an adoptee $o_1$ and an adopter $o_a$. Both objects have to be tracked at the moment of the adoption[2]. Let $\rho_1$ be the key of the adoptee and $\rho_a$ the key of the adopter. The type rule for adoption is

$$\frac{\begin{array}{l}\Delta; \Gamma; C \vdash e_1 : \texttt{tr}(\rho_1); C_1 \\ \Delta; \Gamma; C_1 \vdash e_2 : \texttt{tr}(\rho_a); \{\rho_1 \mapsto h\} \otimes C_2 \\ \Delta \vdash C_2 \leq \{\rho_a\}\end{array}}{\Delta; \Gamma; C \vdash \texttt{adopt}\, e_1{:}h\, \texttt{by}\, e_2 : \rho_a \triangleright h; C_2}\,[\text{adopt}]$$

After typing $e_1$, the adopted object and $e_2$, the adopter, we must have capability $\{\rho_1 \mapsto h\} \otimes C_2$, which makes sure that the to be adopted object is live and has type $h$. We also check that the adopter is accessible, i.e., $\rho_a$ is live by the judgement $\Delta \vdash C_2 \leq \{\rho_a\}$. The result of the adoption has type $\rho_a \triangleright h$, where the guard $\rho_a$ reflects the key of the adopter and $h$ is the heap type of the adopted object. The final capability is $C_2$, reflecting that we have given up the key $\rho_1$ of the adopted object.

Operationally, adoption adds a pointer from the adopter $o_a$ to $o_1$, thereby guaranteeing that we do not loose all references to $o_1$. The invariant that $o_1$ is accessible only via the guarded type $\rho_a \triangleright h$ is enforced because, the key $\rho_1$ for tracked access is consumed, and is not recovered until we free the adopter $o_2$. For the same reason, an object can be adopted by only one adopter at any given time. Thus the pointers from adopters to adoptees form a linear (unshared) tree.

Adoption allows a collection of tracked objects $o_1..o_n$ to be adopted by a single object $o_a$, and thus viewed

[2]We relax this requirement for the adopter later.

under a common type $\rho_a \triangleright h$. References of that type can refer to any of the adopted objects, thus allowing arbitrary aliasing. As an example of adoption, below is the code used by the dictionary to allocate a fresh, but sharable cell.

```
fun newCell(dct : tr(ρd)) : ρd ▷ ⟨int[]•⟩
       pre {ρd ↦ dictionary}
       post {ρd ↦ dictionary}
 {
    let cell : tr(ρ) = new<1> in
    cell.1 := newarray(10);
    adopt cell:ρd ▷ ⟨int[]•⟩ by dct
 }
```

The function takes the dictionary as an argument, allocates a fresh cell, initializes its contents to a fresh array of size 10, and finally has the cell adopted by the dictionary. After the evaluation of `new`, we have capability $\{\rho_d \mapsto \texttt{dictionary}\}\otimes\{\rho \mapsto \langle\texttt{int}\rangle\}$, where $\rho$ is the key of the freshly allocated 1-tuple with initial contents 0 (thus type `int`). Next, the call to `newarray` returns a new tracked array of type $\texttt{tr}(\rho_2)$ for some fresh $\rho_2$, along with a new capability $\{\rho_2 \mapsto \texttt{int[]}\}$. At this point we use the tuple update rule [t-update] from the Appendix:

$$\frac{\begin{array}{l}\Delta;\Gamma;C \vdash e_1 : \texttt{tr}(\rho); C_1\\ \Delta;\Gamma;C_1 \vdash e_2 : \tau; \{\rho \mapsto h\}\otimes C_2\\ h = \langle\sigma_1..\sigma_{i-1},\tau_i,\sigma_{i+1}..\sigma_n\rangle\\ h' = \langle\sigma_1..\sigma_{i-1},\tau,\sigma_{i+1}..\sigma_n\rangle\end{array}}{\Delta;\Gamma;C \vdash e_1.i := e_2 : \tau; \{\rho \mapsto h'\}\otimes C_2}\text{[t-update]}$$

We instantiate the update rule above with

$$\begin{array}{l}\tau = \texttt{tr}(\rho_2)\\ C_2 = \{\rho_d \mapsto \texttt{dictionary}\}\otimes\{\rho_2 \mapsto \texttt{int[]}\}\\ h = \langle\texttt{int}\rangle\\ h' = \langle\texttt{tr}(\rho_2)\rangle\end{array}$$

After the update, the capabilities are $\{\rho_d \mapsto \texttt{dictionary}\}$ $\otimes\{\rho \mapsto \langle\texttt{tr}(\rho_2)\rangle\}\otimes\{\rho_2 \mapsto \texttt{int[]}\}$, i.e., we changed the contents of $\rho$ to refer to the new array of type $\texttt{tr}(\rho_2)$. Note that this is a strong update at the type level, since the content of the cell changed type from `int` to $\texttt{tr}(\rho_2)$.

At the point of adoption, we require the adopted cell to have type $h = \langle\texttt{int[]}^\bullet\rangle$ as stated in the `adopt` expression. The current type associated with $\texttt{tr}(\rho)$ however is $\langle\texttt{tr}(\rho_2)\rangle$. To form the desired linear type, we package the capability $\{\rho_2 \mapsto \texttt{int[]}\}$ together with the reference $\texttt{tr}(\rho_2)$ by applying rule [cap-transform] from the Appendix with judgement

$$\{\rho \mapsto \langle\texttt{tr}(\rho_2)\rangle\}\otimes\{\rho_2 \mapsto \texttt{int[]}\} \;\vdash\; \{\rho \mapsto \langle\texttt{int[]}^\bullet\rangle\}$$

The current capability is now $\{\rho_d \mapsto \texttt{dictionary}\}\otimes\{\rho \mapsto \langle\texttt{int[]}^\bullet\rangle\}$. The adopt rule then consumes key $\rho$ and leaves us with the new cell with guarded type $\rho_d\triangleright\langle\texttt{int[]}^\bullet\rangle$, which matches the expected return type of `newCell`.

## 4.2 Focus

Allowing nonlinear types with linear components would not be useful without the `focus` operation, since the linear components cannot be accessed from the nonlinear container. Focusing on an object with guarded type $G \triangleright h$ allows temporary access to the linear components of $h$. We illustrate `focus` through our running example, by implementing the resize function of resizable arrays. The function takes parameter `cell` of guarded type $\rho_d \triangleright \langle\texttt{int[]}^\bullet\rangle$ containing the old array, and parameter `size` for the new array size. The function expects a capability on entry that includes $\rho_d$, the key for the dictionary guarding our cell.

```
fun resize[ρd](cell:ρd ▷ ⟨int[]•⟩, size:int): int
          pre {ρd ↦ dictionary}
          post {ρd ↦ dictionary}
 {
    let newa = newarray(size) in
    let fcell = focus cell in
    let olda = fcell.1 in
    copy(olda,newa);
    fcell.1 := newa;
    free olda
 }
```

First, we allocate the new array and bind the result to `newa`. Assuming that `newarray` returns a tracked integer array, we assign `newa` the type $\texttt{tr}(\rho_1)$ for a fresh key $\rho_1$. Our current capability is $\{\rho_d \mapsto \texttt{dictionary}\}\otimes\{\rho_1 \mapsto \texttt{int[]}\}$. Next, we apply focus to `cell` and bind the focused reference to `fcell`. The typing rule for focus is

$$\frac{\begin{array}{l}\Delta;\Gamma;C \vdash e_1 : G \triangleright h; C_1\otimes C_2\\ \Delta \vdash C_1 \leq G\\ \rho \text{ fresh}\\ \Delta;\Gamma[x : \texttt{tr}(\rho)]; C_2\otimes\{\rho \mapsto h\} \vdash e_2 : \tau_2; C_3\otimes\{\rho \mapsto h\}\end{array}}{\Delta;\Gamma;C \vdash \texttt{let}\, x = \texttt{focus}\, e_1 \,\texttt{in}\, e_2 : \tau_2; C_1\otimes C_3}$$

We instantiate the rule with

$$\begin{array}{l}G = \rho_d\\ h = \langle\texttt{int[]}^\bullet\rangle\\ C_1 = \{\rho_d \mapsto \texttt{dictionary}\}\\ C_2 = \{\rho_1 \mapsto \texttt{int[]}\}\end{array}$$

We split the current capability into $C_1\otimes C_2$, where $C_1$ is sufficient to prove the guard $G$, and $C_2$ is everything that is not needed to prove the guard. In our case, the dictionary key $\rho_d$ is needed, but the key $\rho_1$ for the new array is not needed. We proceed in the body of `focus` with capability

$$\{\rho \mapsto \langle\texttt{int[]}^\bullet\rangle\}\otimes\{\rho_1 \mapsto \texttt{int[]}\}$$

where $\rho$ is the fresh key giving us linear access to the focused cell. Notice that we no longer have the dictionary key $\rho_d$, because it was used for the focus. This prevents focusing again on an alias to `cell` during the body of the focus.

Next, we type the selection `fcell.1`. The appropriate type rule for indexing from the Appendix is

$$\frac{\begin{array}{l}\Delta;\Gamma;C \vdash e : \texttt{tr}(\rho); \{\rho \mapsto h\}\otimes C_2\\ h = \langle\sigma_1..\sigma_{i-1},\tau_i,\sigma_{i+1}..\sigma_n\rangle\end{array}}{\Delta;\Gamma;C \vdash e.i : \tau_i; \{\rho \mapsto h\}\otimes C_2}\text{[t-index]}$$

The rule requires that the type $\tau_i$ of the $i$th component that is selected is non-linear (not $\sigma$) in order to satisfy our type judgement invariant that expressions evaluate to non-linear types. Thus in our example, the heap type associated with `fcell` ($\rho$) first needs to be transformed by unpacking the linear array type in our cell, via the following judgement (rule [cap-transform] in the Appendix)

$$\{\rho \mapsto \langle \texttt{int[]}^\bullet \rangle\} \;\vdash\; \{\rho \mapsto \langle \texttt{tr}(\rho_2)\rangle\}\otimes\{\rho_2 \mapsto \texttt{int[]}\}$$

This unpacking step simply names the hidden key within $\texttt{int[]}^\bullet$, picking a fresh key name $\rho_2$, and changes the cell type to refer to $\texttt{tr}(\rho_2)$, while adding the capability $\{\rho_2 \mapsto \texttt{int[]}\}$ to the current capability.

Now we can instantiate the [t-index] rule with $h = \langle \texttt{tr}(\rho_2)\rangle$, and thus bind `olda` to $\texttt{tr}(\rho_2)$. The capabilities at this point are $\{\rho \mapsto \langle \texttt{tr}(\rho_2)\rangle\}\otimes\{\rho_2 \mapsto \texttt{int[]}\}\otimes\{\rho_1 \mapsto \texttt{int[]}\}$. Next, we copy the contents of the old array to the new one, assuming that `copy` takes two tracked arrays and does not change the capabilities.

Next, we update the cell via `fcell.1:=newa`, thereby applying rule [t-update] as shown in the previous section. The capabilities after this step are

$$\{\rho \mapsto \langle \texttt{tr}(\rho_1)\rangle\}\otimes\{\rho_1 \mapsto \texttt{int[]}\}\otimes\{\rho_2 \mapsto \texttt{int[]}\},$$

reflecting that `fcell` now contains a pointer to the new array $\rho_1$.

Next, we free `olda`, thereby consuming key $\rho_2$, leaving us with capability $\{\rho \mapsto \langle \texttt{tr}(\rho_1)\rangle\}\otimes\{\rho_1 \mapsto \texttt{int[]}\}$. At this point, the scope of our `focus` ends. To apply the remainder of the focus rule, we need to apply a packing step to our cell $\rho$:

$$\{\rho \mapsto \langle \texttt{tr}(\rho_1)\rangle\}\otimes\{\rho_1 \mapsto \texttt{int[]}\} \;\vdash\; \{\rho \mapsto \langle \texttt{int[]}^\bullet \rangle\}$$

We instantiate the remaining part of the `focus` rule with $C_3 = \cdot$ (the empty capability). Our focus key $\rho$ maps correctly to $h = \langle \texttt{int[]}^\bullet \rangle$, therefore, we can end the focus scope, since all aliases to the focused cell now again see the correct type $h$. Ending the focus consumes the focus key $\rho$ and reinstates the capability $C_1 = \{\rho_d \mapsto \texttt{dictionary}\}$ that was temporarily revoked, leaving us with $\{\rho_d \mapsto \texttt{dictionary}\}$, which matches the final capability of `resize`.

### 4.3 Recovering adoptees on free

To simplify the type systems, we have assumed that freeing an adopter will recursively free all adoptees. This is only one possible way to deal with adoptees. In this section, we briefly contrast three different semantics for free.

**Recursive free.** Possibly the simplest semantics of `free` recursively frees all adoptees. This semantics raises a practical issue: to avoid leaks the operation must also free components of linear types of any object encountered. In order to traverse objects recursively, the generic deallocator needs to understand the layout of all objects in memory, much as a garbage collector would. This requirement goes against the desire to apply the presented techniques in a low level system language, where explicit data layout control is important.

A slightly more restrictive scenario however does not rely on a generic memory traversal and is used widely. The region abstraction for memory allocation allows individual objects to be allocated from a given region. Objects are freed not individually, but collectively by freeing the entire region. This model can be viewed as a special case of adoption, where objects are co-allocated from a common block of memory and immediately adopted by the region (thus no reference of linear type is ever accessible to the new block). If data allocated in regions is restricted to nonlinear components, there is no need to traverse blocks recursively to free components. Thus freeing remains a constant time operation as is expected from most region implementations [15].

**Callback semantics.** Another possibility to avoid the generic object traversal at `free` is to register a callback function with each adoptee at the time of adoption. When the adopter is freed each callback is called with the corresponding adoptee (now of linear type), thereby returning the adoptee to the context. Only the particular callback function of an adoptee needs to understand the layout of the adoptee. This semantics has been applied in the region implementation by Gay and Aiken [4].

**Linear list of adoptees.** The third semantics we find useful requires `free` to return all adoptees via a linear list. The context in which `free` is applied can then choose to deal with the adoptee objects in whatever way is appropriate. This semantics however has the restriction over the previous two proposals that it requires all adoptees to have the same type. We express this by having a special adopter type of the form $\texttt{adopter}\langle h\rangle$ reflecting the type $h$ of the adoptees.

In our experience all three semantics are useful in some programming scenarios. A realistic programming language can provide all three through libraries.

## 5. ALGORITHMICS

The main detail we glossed over in the technical presentation is the treatment of control flow merge points. In our prototype compiler, we insist on obtaining a single capability description (modulo key renaming) per program point. Thus, the capabilities along two control flow edges with the same target have to be compatible. This requirement restricts the set of programs that can be typed, but the techniques described in our earlier paper [3] for encoding correlations between values and capabilities can be used to program around this limitation in many cases.

Given this restriction, the complexity of type checking is $O(ekt)$, where $e$ is the number of edges in the control flow graph of a function, $k$ is the maximal number of keys at any program point, and $t$ is the maximal size of a type expression. In practice, $k$ is relatively small (less than the number of local variables), for only few references refer to tracked objects.

## 6. EXTENSIONS

This section discusses a number of extensions that are important in practice, but would have obscured the previous development unnecessarily.

**Multi guards.** The guards $G$ we discussed so far consist only of the single key name of the adopter. It is straight forward to extend guards to conjunctions of keys $G ::= \cdot \mid \rho \wedge G$. Such guards can be introduced via a *guard strengthening* subtyping step: given an expression $e$ with type $G \triangleright h$, we can also give $e$ the type $(G \wedge \rho) \triangleright h$, for any $\rho$. Strengthening the guard makes it harder to access the object, since more keys must be held at the time of access. Guard strengthening is useful to view objects adopted by different adopters under a common type. For instance, given objects of types $\rho_1 \triangleright h$ and $\rho_2 \triangleright h$, we may want to hold a reference to either of these objects. Since their types are incompatible, this is not possible without weakening each object's type to $(\rho_1 \wedge \rho_2) \triangleright h$.

**Abstract guards.** Our definition of `resize` has the unfortunate property that it mentions in the pre and post capability the fact that the key $\rho_d$ guarding the cell is the key of a dictionary object. The nature of the guard is irrelevant for the code at hand. To solve this problem, we extend guards with abstract guard variables $\gamma$, so that guards have the form $G ::= \ldots \mid \gamma \wedge G$. Using an abstract guard $\gamma$ in place of the explicit guard $\{\rho_d\}$, along with an abstract capability $\epsilon$ instead of the explicit capability $\{\rho_d \mapsto \texttt{dictionary}\}$ makes the type of `resize` applicable in more contexts. The only extra piece of information needed is that capability $\epsilon$ implies the guard $\gamma$. This is recorded in the type context $\Delta$ with $\epsilon \leq \gamma$. The more abstract type of `resize` is then

$$\forall[\gamma, \epsilon \leq \gamma].(\epsilon, \gamma \triangleright \langle \texttt{int}[\,]^\bullet \rangle, \texttt{int}) \to (\epsilon, \texttt{int})$$

and is applicable to guarded cells no matter what the guard is. The idea of using bounded quantification on capabilities has been proposed by Crary et.al. to allow region aliasing in particular contexts [2]. Similarly, abstract guards and bounded capabilities give control over the aliasing granularity of guarded types. Consider the type context

$$\Delta = \gamma_1, \gamma_2, \gamma_3, \epsilon_1 \leq (\gamma_1 \wedge \gamma_2), \epsilon_3 \leq \gamma_3$$

with capability $\epsilon_1 \otimes \epsilon_3$, and three objects of respective types $\gamma_1 \triangleright h$, $\gamma_2 \triangleright h$, and $\gamma_3 \triangleright h$. The capability bounds imply that guard $\gamma_1$ and $\gamma_2$ may overlap. Thus during a focus on $\gamma_1 \triangleright h$, guard $\gamma_2$ is not satisfiable, but guard $\gamma_3$ is, because the capabilities used to prove $\gamma_1$ and $\gamma_3$ are disjoint.

**Tracked vs. linear functions.** So far he have treated function types as ordinary types $\tau$, not heap types $h$, thereby ignoring the space requirement of an associated closure record. If we want programmer control over closure allocation and deallocation, we can treat function types as heap types. Closures then need to be allocated via `new` and result in a tracked type, and they can be explicitly freed at a later point via `free`. Tracked functions can be called any number of times, until the point of deallocation.

In contrast, linear functions as proposed by Wadler in [17] have the property that they can be called only once (so-called *once functions*). Once functions arise naturally in linear type systems if function closures can capture objects of linear type. Note that in our formalism, this is not possible, since the type environment contains no linear types $\sigma$. If a function requires access to a key, the key must be explicitly mentioned in the `pre` capability.

Interestingly, once functions are easily modeled in our formalism as tracked functions, where the function itself deallocates its own closure record. The type of such a function is $\texttt{tr}(\rho_c)$, where the heap type $h$ associated with $\rho_c$ has the form:

$$h = \forall[\epsilon, \rho_c](\{\rho_c \mapsto h\} \otimes \epsilon, \sigma_1) \to (\epsilon, \sigma_2)$$

The `pre` capability contains $\{\rho_c \mapsto h\}$, thereby requiring the key for the closure, but the `post` capability does not contain $\rho_c$, thereby expressing that the function consumes its own closure record. Once functions would be allowed to capture objects of linear type in their closure.

It is also interesting to note here that Wadler's `let!` cannot be applied to linear function types for which it is unsound. Interestingly, adoption is sound for tracked and once functions. As long as the function does not require its own closure record in the pre capability it can be called after adoption. If it does, then the guarded function type will be uncallable, since the key of the closure is not available.

**Autofocus.** We have described `focus` as an explicit construct, where the extent of the focus is a lexical scope. Requiring a full lexical scope for a focus is often impractical, i.e., the focus should be released before the end of the scope[3]. From a static typing perspective, the action occurring at the end of a focus scope is that the temporary key $\rho$ has to be revoked after checking that the associated heap type $h$ is the same as at the beginning of the scope, and to reinstate the revoked capability $C_1$ used to prove the focus guard. To do away with the lexical scope, we simply have to record this information in the capability state for later perusal when the focus should end. We propose therefore to extend capabilities with a linear implication

$$C ::= \ldots \mid \{\rho \mapsto h\} \multimap C$$

along with the focus elimination rule

$$\{\rho \mapsto h\} \otimes \{\rho \mapsto h\} \multimap C \ \vdash \ C$$

Now `focus` can be a simple expression with the following typing rule:

$$\frac{\begin{array}{l}\Delta; \Gamma; C \vdash e : G \triangleright h; C_1 \otimes C_2 \\ \Delta \vdash C_1 \leq G \qquad\qquad \rho \text{ fresh} \\ C_3 = C_2 \otimes \{\rho \mapsto h\} \otimes \{\rho \mapsto h\} \multimap C_1\end{array}}{\Delta; \Gamma; C \vdash \texttt{focus}\, e : \texttt{tr}(\rho); C_3}\text{[focus(e)]}$$

Applying the above rule establishes focus on the result of expression $e$, until the focus elimination rule is applied. A step further is to infer focus expressions automatically whenever a tracked type is expected and a guarded type is provided, thereby providing *autofocus.*

**Self-adoption.** Wadler's `let!` has the advantage over adoption that there is no runtime operation involved, whereas adoption establishes an adoption pointer. There

[3]This issue is similar to the one of early deallocation in region based memory management.

are two ways to provide a similar noop behavior in the adoption model. The lifetime relationship established by adoption states that the adoptee will outlive the adopter. This is a reflexive relation (it is okay, if both lifetimes end simultaneously). Thus we can view an adopter as its own adopter. A simple, but unsound approach of doing so would be to view a reference of tracked type $\mathtt{tr}(\rho)$ with current capability $\{\rho \mapsto h\}$ as $\rho \rhd h$. The approach is unsound because, through the handle of type $\mathtt{tr}(\rho)$, it is possible to change the heap cell from type $h$ to $h'$ via a strong update and then to view the cell through the guarded type under $h$. There are two ways to solve this problem: 1) Provide a virtual adopter $\rho'$ and revoke capability $\rho$, thereby making it impossible to change the type $h$ (except temporarily under focus). To release the adoptee, we use linear implication and the focus elimination rule introduced above, as expressed by the following typing rule

$$\frac{\begin{array}{l}\Delta; \Gamma; C \vdash e : \mathtt{tr}(\rho); \{\rho \mapsto h\} \otimes C_2 \\ \rho' \text{ fresh} \\ C_3 = C_2 \otimes \rho' \mapsto h' \otimes \{\rho' \mapsto h'\} \multimap \{\rho \mapsto h\}\end{array}}{\Delta; \Gamma; C \vdash e : \rho' \rhd h; C_3}\,[\text{self}(1)]$$

Here, $\rho'$ serves as the virtual adopter with $h'$ being an arbitrary type, since there is no actual reference available to $\rho'$. The linear implication $\{\rho' \mapsto h'\} \multimap \{\rho \mapsto h\}$ allows the adoption to be undone by eliminating the virtual adopter.

A second solution is to strengthen the guard to include the heap type under which the alias is established. Given tracked type $\mathtt{tr}(\rho)$ with current capability $\{\rho \mapsto h\}$ we provide an alias with guarded type $\rho{:}h \rhd h$. The guard $\rho{:}h$ is satisfied by a capability $C$ only if $C$ contains $\{\rho \mapsto h\}$. This solution allows an object to be viewed under a tracked type $\mathtt{tr}(\rho)$ and simultaneously under a guarded type $\rho{:}h \rhd h$. If strong updates via the tracked type change $h$ to $h'$, the guarded reference becomes inaccessible.

**Transitive adoption.** The adoption relationship is transitive, i.e., if object $o_1$ is adopted by $o_2$, which in turn is adopted by $o_3$, then $o_1$ is effectively adopted by $o_3$. This observation can be exploited in two ways: First, when object $o_2$ with key $\rho_2$ is adopted by $o_3$ with key $\rho_3$, any object earlier adopted by $o_2$ thereby having type $\rho_2 \rhd h_1$ can be viewed under the new guard $\rho_3 \rhd h_1$. Second, there is no reason that $o_2$ cannot adopt further objects after having been itself adopted by $o_3$. Given a reference to $o_2$ with guarded type $\rho_3 \rhd h_2$, it can adopt any object $\mathtt{tr}(\rho_4)$ with underlying type $h_4$ which can then be viewed under type $\rho_3 \rhd h_4$.

## 7. VALIDATION

We have incorporated the ideas of adoption and focus into the Vault programming language [3]. We briefly describe how Vault makes these features available more conveniently than this paper's "mini" language and show a "real world" example.

### 7.1 Implementation in Vault

We implemented adoption and focus in the Vault programming language, including the extensions of multi-guards, abstract guards, tracked functions, autofocus, and self-adoption. In Vault's surface syntax, the tracked type $\mathtt{tr}(\rho)$ is written `tracked($R)`, and the guarded type $\rho_1 \wedge \ldots \wedge \rho_n \rhd \tau$ is written `{$R1,...,$Rn}:t` or as `$G:t` for an abstract guard. For protocol checking, a programmer may attach a state token to a key, for example `$FILE@open`. Our function syntax combines the pre- and postconditions into a change specification, which states the difference between the before and after capabilities. The constituents of a change specification are the following:

| **Syntax** | **Precondition** | **Postcondition** |
|---|---|---|
| `new $K@s` | $C$ | $C\otimes$`{$K@s}` (fresh `$K`) |
| `+$K@s` | $C$ | $C\otimes$`{$K@s}` |
| `-$K@s` | $C\otimes$`{$K@s}` | $C$ |
| `$K@s` | $C\otimes$`{$K@s}` | $C\otimes$`{$K@s}` |
| `$K@`$s_1$`->`$s_2$ | $C\otimes$`{$K@`$s_1$`}` | $C\otimes$`{$K@`$s_2$`}` |

The `@s` suffix may be omitted for those protocols that track object lifetime but not state. The change specification is both more terse than separate pre- and postconditions and also hides the universally quantified context capability $C$.

Besides self-adoption, Vault provides general adoption in the three forms described in Section 4.3. Region allocation is provided through primitives in the language, while the callback and linear list approaches are provided through two library interfaces. These interfaces have functions to create an adopter, to adopt objects, and to delete the adopter and relinquish the adoptees.

### 7.2 Example: Direct3D vertex buffers

Microsoft's Direct3D graphics library provides an example of a realistic interface protocol. The library supports a "vertex buffer" abstraction for rendering images to a screen. To render an image, the programmer must create a buffer, clear it, draw primitives to it, and then present it. Further, calls to drawing primitives must be directly enclosed by calls to the functions `BeginScene` and `EndScene`. Figure 4 shows a Vault specification of this protocol. The right side of the figure illustrates the interface as a state machine.

To draw a single frame of an animation, a program loops through the cycle in this protocol, taking the buffer through the states `raw`, `clear`, `rendering`, `ready`, and back to `raw`. Within this larger cycle, the program will typically call `DrawPrimitive` many times to draw a complex scene. (Indeed, the function `DrawPrimitive` here is a surrogate for a collection of drawing functions.)

For Direct3D graphics programs, vertex buffers tend to be the program's central objects, referenced from many data structures. The need to alias vertex buffers and the cyclic nature of the protocol lend themselves well to adoption and focus. A graphics program can create vertex buffers and adopt them in the `raw` state. The adopted vertex buffers can then be referenced from many data structures and no static knowledge of their aliasing relations is available.

When the program wants to render an animation frame,

```
interface VERTEX_BUFFER {
  type buffer;
  tracked($B) buffer CreateVertexBuffer () [new $B@raw];
  void Clear (tracked($B) buffer) [$B@raw->clear];
  void BeginScene (tracked($B) buffer) [$B@clear->rendering];
  void DrawPrimitive ($B:buffer, ...) [$B@rendering];
  void EndScene (tracked($B) buffer) [$B@rendering->ready];
  void Present (tracked($B) buffer) [$B@ready->raw];
}
```

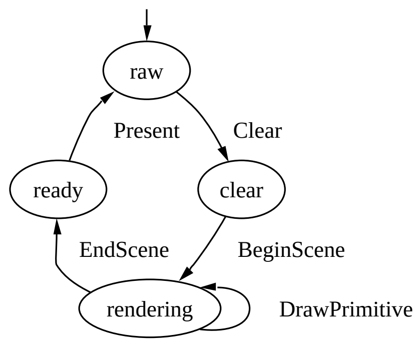

**Figure 4: The interface to vertex buffer objects in the Direct3D library.**

it focuses on a particular vertex buffer in order to call the interface's state-changing operations. By the end of the focus, the buffer is back in the `raw` state, leaving any alias of the buffer unaware of the intermediate states.

This example illustrates how adoption and focus allow an object to switch back and forth from supporting aliasing to supporting protocol checking over the course of its lifetime.

## 8. RELATED WORK

We briefly review the related work on reasoning about state changes in programs with dynamically allocated memory.

**Region-based memory management.** Tofte and Talpin present an inference system for classifying all allocated data of a program into regions and deducing a safe lifetime for each region [16], which enables provably memory-safe implementations of ML-like languages without a garbage collector. Crary *et al.*'s Capability Calculus extends this work by allowing explicit region allocation and deletes, while making sure that all data accesses to a region happen during its lifetime [2]. Similarly, Niss and Henglein study an explicit region calculus, albeit for first order programs [5].

The commonality of these systems is that only regions are treated linearly; all other objects are allocated within regions and have types akin to guarded types. Regions are not first-class values and cannot be stored in data structures.

**Linear type systems.** Starting with Wadler [17], linear types systems have been used in purely functional languages to enforce single threading on the state of the world or to implement operations like array updating without the cost of a full copy [12]. Linear type systems enable resource management at the granularity of a single object. Every use of an object of linear type consumes the object, leading to a programming style where linear objects are threaded through the computation. Wadler's `let!` construct, or its variations [9], can be used to give a temporary nonlinear type to a object of linear type. Walker and Watkins [19] study a type system with three kinds of objects: linear, reference counted, and region allocated. The kind of an object is fixed at allocation without a means to change kind. They provide `let!` only for regions.

**Alias type systems.** Originally developed to track incremental initialization in TAL, alias types provide compile time names for unshared objects [13]. Walker extends this approach with existentially bound capabilities to support recursive data structures [18], bringing back the full expressibility of linear types and its fine grained resource management. The advantage of alias types over linear types is that uses are not destructive and local aliasing can be handled. This leads to a more natural imperative programming style, since the threading of values in linear type systems is done at the type level. Walker does not examine ways to switch between linear and nonlinear views of objects.

**Object-oriented analyses.** The problem of representation exposure, namely uncontrolled sharing of the internals of an abstraction, is most problematic in object oriented systems, where sharing and destructive updates are common. A number of different approaches to declaring and checking representation invariants have been proposed, all with incomparable expressivity and enforceability. For example, the pivots used in ESC correspond to a linear component of a possibly shared container, but the formalism developed by Leino *et al.*, does not allow pivots to be reused [8]. Roles [7] enable the description of precise heap structures, since they allow more than a single reference to tracked objects, as long as all references are known. However, if an object's role is the equivalent of a guarded type, i.e., with an unknown number of references, the object's role is frozen (no more type changes), and it can only be freed by a garbage collector. The alias burying system presented by Boyland also allows unique pointers in shared data structures, but access control is exercised through read effects, rather than the possible aliasing we express via guards [1]. Furthermore, the alias burying approach does not provide lifetime control.

**Heap logic.** The Logic of Bunched Implications (BI) [6] is a formalism that allows one to reason about heap allocated structures and their sharing properties. At its core, BI uses spatial conjunction to reason about disjoint parts of the heap. We are not aware of any automatic reasoning systems based on BI.

Shape analysis based on 3-valued logic (TVLA) [11] is another formalims for reasoning about heap structures. TVLA is promising in that there exists an effective algorithm for inferring rather precise heap properties, albeit with high worst case complextiy.

## 9. FUTURE WORK

The current paper does not address the problem of sharing objects between multiple threads. Whereas exclusive object access is easily modeled with keys, actually sharing data between threads is not. Adoption does not provide a solution here, since the adopter's key would still be available only to one thread at a time. We are investigating mechanisms for sharing keys. In shared key scenarios, the focus operation on shared keys would have to become an explicit locking operation.

## 10. CONCLUSION

Enforcing interface protocols, data structure invariants, and proper memory management at compile time greatly improves the quality of imperative resource-handling software. Our work aims to develop techniques and language features to make such static enforcement practical and mainstream. This paper provides a large step in that direction by enabling property checking and data structure invariants for a much larger class of programming idioms. Still, there are data structure scenarios that our type system cannot handle. For instance, once an object is adopted, it cannot be recovered independently of all other adoptees of the same adopter. Furthermore, the present techniques do not address issues related to concurrency.

## REFERENCES

[1] J. Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 31(6):533–553, May 2001.

[2] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In POPL'99 [10].

[3] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2001.

[4] D. Gay and A. Aiken. Memory management with explicit regions. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, number 33:5 in SIGPLAN notices, pages 313–323, June 1998.

[5] F. Henglein, H. Makholm, and H. Niss. A direct approach to control-flow sensitive region-based memory management. In *ACM Conference on Principles and Practice of Declarative Programming*, Sept. 2001.

[6] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Conference Record of the 28th Annual ACM Symposium on Principles of Programming Languages*, pages 14–26. ACM Press, Jan. 2001.

[7] V. Kuncak, P. Lam, and M. C. Rinard. Role analysis. In *Conference Record of the 29th Annual ACM Symposium on Principles of Programming Languages*, Jan. 2002.

[8] K. R. M. Leino and G. Nelson. Data abstraction and information hiding. Technical Report 160, Compaq SRC, nov 2000.

[9] M. Odersky. Observers for linear types. In B. Krieg-Brückner, editor, *ESOP '92: 4th European Symposium on Programming, Rennes, France, Proceedings*, pages 390–407, New York, 1992. Springer-Verlag.

[10] *Conference Record of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM Press, Jan. 1999.

[11] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In POPL'99 [10], pages 105–118.

[12] S. Smetsers, E. Barendsen, M. v. Eekelen, and R. Plasmeijer. Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. *Lecture Notes in Computer Science*, 776:358–379, 1994.

[13] F. Smith, D. Walker, and J. G. Morrisett. Alias types. In *European Symposium on Programming*, pages 366–381, 2000.

[14] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *tose*, SE-12(1):157–171, Jan. 1986.

[15] M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T. H. Olesen, P. Sestoft, and P. Bertelsen. Programming with regions in the ml kit (for version 3). Technical Report 98/25, Department of Computer Science, University of Copenhagen, 1998.

[16] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value $\lambda$-calculus using a stack of regions. In *Conference Record of the 21st Annual ACM SSymposium on Principles of Programming Languages*, pages 188–201, Jan. 1994.

[17] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods.* Apr. 1990. IFIP TC 2 Working Conference.

[18] D. Walker and G. Morrisett. Alias types for recursive data structures. In *Proceedings of the 4th Workshop on Types in Compilation*, Sept. 2000.

[19] D. Walker and K. Watkins. On linear types and regions. In *Proceedings of the International Conference on Functional Programming (ICFP '01)*, Sept. 2001.

## APPENDIX

$\boxed{\Delta;\Gamma;C \vdash e : \tau; C}$

$$\frac{\Gamma(x) = \tau}{\Delta;\Gamma;C \vdash x : \tau; C}\text{[var]} \qquad \frac{}{\Delta;\Gamma;C \vdash i : \texttt{int}; C}\text{[int]}$$

$$\frac{\begin{array}{l}\Delta;\Gamma;C \vdash e : \texttt{tr}(\rho); \{\rho \mapsto h\}\otimes C_2 \\ h = \langle \sigma_1..\sigma_{i-1}, \tau_i, \sigma_{i+1}..\sigma_n\rangle\end{array}}{\Delta;\Gamma;C \vdash e.i : \tau_i; \{\rho \mapsto h\}\otimes C_2}\text{[t-index]} \qquad \frac{\begin{array}{l}\Delta;\Gamma;C \vdash e : G \triangleright h; C_2 \\ \Delta \vdash C_2 \leq G \\ h = \langle \sigma_1..\sigma_{i-1}, \tau_i, \sigma_{i+1}..\sigma_n\rangle\end{array}}{\Delta;\Gamma;C \vdash e.i : \tau_i; C_2}\text{[g-index]}$$

$$\frac{\begin{array}{l}\Delta;\Gamma;C \vdash e_1 : \texttt{tr}(\rho); C_1 \\ \Delta;\Gamma;C_1 \vdash e_2 : \tau; \{\rho \mapsto h\}\otimes C_2 \\ h = \langle \sigma_1..\sigma_{i-1}, \tau_i, \sigma_{i+1}..\sigma_n\rangle \\ h' = \langle \sigma_1..\sigma_{i-1}, \tau, \sigma_{i+1}..\sigma_n\rangle\end{array}}{\Delta;\Gamma;C \vdash e_1.i := e_2 : \tau; \{\rho \mapsto h'\}\otimes C_2}\text{[t-update]} \qquad \frac{\begin{array}{l}\Delta;\Gamma;C \vdash e_1 : G \triangleright h; C_1 \\ h = \langle \sigma_1..\sigma_{i-1}, \tau_i, \sigma_{i+1}..\sigma_n\rangle \\ \Delta;\Gamma;C_1 \vdash e_2 : \tau_i; C_2 \\ \Delta \vdash C_2 \leq G\end{array}}{\Delta;\Gamma;C \vdash e_1.i := e_2 : \tau_i; C_2}\text{[g-update]}$$

$$\frac{\begin{array}{l}\tau_f = \forall[\Delta_1](C_b, \sigma_1) \rightarrow (C_a, \sigma_2) \\ C_b; \sigma_1 \vdash C_1; \tau_1 \\ \Delta, \Delta_1; \Gamma[f : \tau_f][x : \tau_1]; C_1 \vdash e : \tau_2; C_2 \\ C_2; \tau_2 \vdash C_a; \sigma_2\end{array}}{\Delta;\Gamma;C \vdash \begin{array}{l}\texttt{fun}\ f[\Delta_1](x : \sigma_1) : \sigma_2 \\ \texttt{pre}\ C_b\,\texttt{post}\ C_a\ \{e\}\end{array} : \tau_f; C}\text{[fun]} \qquad \frac{\begin{array}{l}\Delta;\Gamma;C \vdash e_1 : \forall[].(C_b, \sigma_1) \rightarrow (C_a, \sigma_2); C_1 \\ \Delta;\Gamma;C_1 \vdash e_2 : \tau_2; C_2 \\ C_2; \tau_2 \vdash C_b; \sigma_1 \\ C_a; \sigma_2 \vdash C_3; \tau_3\end{array}}{\Delta;\Gamma;C \vdash e_1(e_2) : \tau_3; C_3}\text{[app]}$$

$$\frac{\Delta;\Gamma;C \vdash e : \forall[\rho', \Delta'].\tau_f; C_1}{\Delta;\Gamma;C \vdash e[\rho] : \forall[\Delta'].(\tau_f[\rho/\rho']); C_1}\text{[key]} \qquad \frac{\Delta;\Gamma;C \vdash e : \forall[\gamma, \Delta'].\tau_f; C_1}{\Delta;\Gamma;C \vdash e[G] : \forall[\Delta'].(\tau_f[G/\gamma]); C_1}\text{[guard]}$$

$$\frac{\begin{array}{l}\Delta;\Gamma;C \vdash e : \forall[\epsilon \leq G, \Delta'].\tau_f; C_1 \\ \Delta \vdash C_0 \leq G\end{array}}{\Delta;\Gamma;C \vdash e[C_0] : \forall[\Delta'].(\tau_f[C_0/\epsilon]); C_1}\text{[cap]} \qquad \frac{\begin{array}{l}\Delta;\Gamma;C \vdash e_1 : \tau_1; C_1 \\ \Delta;\Gamma[x : \tau_1]; C_1 \vdash e_2 : \tau_2; C_2\end{array}}{\Delta;\Gamma;C \vdash \texttt{let}\,x = e_1\,\texttt{in}\,e_2 : \tau_2; C_2}\text{[let]}$$

$$\frac{h = \langle \underbrace{\texttt{int}..\texttt{int}}_{n} \rangle \qquad \rho \text{ fresh}}{\Delta;\Gamma;C \vdash \texttt{new}\langle n \rangle : \texttt{int}; \{\rho \mapsto h\}\otimes C}\text{[new]} \qquad \frac{\Delta;\Gamma;C \vdash e : \texttt{tr}(\rho); \{\rho \mapsto h\}\otimes C_1}{\Delta;\Gamma;C \vdash \texttt{free}\,e : \texttt{int}; C_1}\text{[free]}$$

$$\frac{\begin{array}{l}\Delta;\Gamma;C \vdash e_1 : \texttt{tr}(\rho_1); C_1 \\ \Delta;\Gamma;C_1 \vdash e_2 : \texttt{tr}(\rho_2); \{\rho_1 \mapsto h\}\otimes C_2 \\ \Delta \vdash C_2 \leq \{\rho_2\}\end{array}}{\Delta;\Gamma;C \vdash \texttt{adopt}\ e_1{:}h\,\texttt{by}\,e_2 : \rho_2 \triangleright h; C_2}\text{[adopt]} \qquad \frac{\begin{array}{l}\Delta;\Gamma;C \vdash e_1 : G \triangleright h; C_1\otimes C_2 \\ \Delta \vdash C_1 \leq G \\ \rho \text{ fresh} \\ \Delta;\Gamma[x : \texttt{tr}(\rho)]; C_2\otimes\{\rho \mapsto h\} \vdash e_2 : \tau_2; C_3\otimes\{\rho \mapsto h\}\end{array}}{\Delta;\Gamma;C \vdash \texttt{let}\,x = \texttt{focus}\,e_1\,\texttt{in}\,e_2 : \tau_2; C_1\otimes C_3}\text{[focus]}$$

$$\frac{\begin{array}{l}\Delta;\Gamma;C \vdash e : \tau; C_1 \\ C_1 \vdash C_2\end{array}}{\Delta;\Gamma;C \vdash e : \tau; C_2}\text{[cap-transform]}$$

$\boxed{C \vdash C}$

$$\frac{C_1 \vdash C_2}{\{\rho \mapsto h\}\otimes C_1 \vdash \{\rho \mapsto h\}\otimes C_2}$$

$$\frac{C_1; \sigma_i \vdash C_2; \sigma_i'}{\{\rho \mapsto \langle \sigma_1..\sigma_n \rangle\}\otimes C_1 \vdash \{\rho \mapsto \langle \sigma_1..\sigma_i'..\sigma_n \rangle\}\otimes C_2}$$

$\boxed{C; \sigma \vdash C; \sigma}$

$$\frac{}{C; \sigma \vdash C; \sigma}$$

$$\frac{}{C_1; \exists[\rho|\{\rho \mapsto h\}].\,\texttt{tr}(\rho) \vdash C_1\otimes\{\rho \mapsto h\}; \texttt{tr}(\rho)}\text{[unpack]}$$

$$\frac{}{C_1\otimes\{\rho \mapsto h\}; \texttt{tr}(\rho) \vdash C_1; \exists[\rho|\{\rho \mapsto h\}].\,\texttt{tr}(\rho)}\text{[pack]}$$

$\boxed{\Delta \vdash C \leq G}$

$$\frac{}{\Delta \vdash C \leq \cdot} \qquad \frac{\begin{array}{l}\Delta \vdash C \leq G_1 \\ \Delta \vdash C \leq G_2\end{array}}{\Delta \vdash C \leq G_1 \wedge G_2}$$

$$\frac{\Delta \vdash C_1 \leq G}{\Delta \vdash C_1\otimes C_2 \leq G} \qquad \frac{\Delta \vdash C_2 \leq G}{\Delta \vdash C_1\otimes C_2 \leq G}$$

$$\frac{}{\Delta \vdash \{\rho \mapsto h\} \leq \{\rho\}} \qquad \frac{}{\Delta, \epsilon \leq G_1 \wedge G_2 \vdash \epsilon \leq G_1}$$