A Flexible Type System for Fearless Concurrency

Mae Milano
University of California, Berkeley
mpmilano@berkeley.edu

Joshua Turcotti
University of California, Berkeley
jturcotti@berkeley.edu

Andrew C. Myers
Cornell University
Ithaca, NY, USA
andru@cs.cornell.edu

Abstract
This paper proposes a new type system for concurrent programs, allowing threads to exchange complex object graphs without risking destructive data races. While this goal is shared by a rich history of past work, existing solutions either rely on strictly enforced heap invariants that prohibit natural programming patterns or demand pervasive annotations even for simple programming tasks. As a result, past systems cannot express intuitively simple code without unnatural rewrites or substantial annotation burdens. Our work avoids these pitfalls through a novel type system that provides sound reasoning about separation in the heap while remaining flexible enough to support a wide range of desirable heap manipulations. This new sweet spot is attained by enforcing a heap domination invariant similarly to prior work, but tempering it by allowing complex exceptions that add little annotation burden. Our results include: (1) code examples showing that common data structure manipulations which are difficult or impossible to express in prior work are natural and direct in our system, (2) a formal proof of correctness demonstrating that well-typed programs cannot encounter destructive data races at run time, and (3) an efficient type checker implemented in Gallina and OCaml.

CCS Concepts:
- Software and its engineering → Concurrent programming languages; Concurrent programming structures.

Keywords: concurrency, type systems, aliasing

ACM Reference Format:

The promise of a language with lightweight, safe concurrency has long been attractive. Such a language would statically ensure freedom from destructive races, avoiding the cost of synchronization except when concurrent threads explicitly communicate. Our goal is to obtain this "fearless concurrency" [35] for a language with pervasive mutability at its core. Broadly speaking, past efforts to design such a language fall into three camps. Some, like Rust [36], simplify reasoning by severely limiting the shape of representable data structures—making the implementation of common data structures, like the doubly linked list, unapproachable by non-experts1. In others [17, 26, 28, 29, 33, 46], harsh limitations on aliasing cause data structure traversal and manipulation to involve significant mutation of the object graph even for simple computations—for example, in these systems removing the tail of a recursively singly linked list incurs a write to each list node traversed. Existing approaches that avoid either pitfall require significant programmer annotation to explain aliasing information directly to the compiler [8, 12, 13].

This paper introduces a new type system for fearless concurrency. As in prior work, the goal is to statically ensure that at any point during execution, the part of the heap accessible to a given thread—what we call its reservation—is disjoint from the reservations of all other threads. Inspired by Tofte and Talpin [49], the object graph is partitioned into a set of regions, a purely compile-time construct which groups objects that enter or leave a thread’s reservation as a unit. Neither regions nor reservations are fixed; both can and should change during program execution to reflect the movement of objects among threads. As in prior work [17, 26, 28], our type system supports both inter- and intra-region references; intra-region references may freely link objects within the same region, allowing programmers to easily form arbitrary object graphs, while inter-region references are tracked by the type system and stored in appropriately annotated isolated fields. By tracking this information, the type system ensures that threads do not reference objects outside their reservations. Unlike in prior work, this guarantee is provided without requiring that isolated field references satisfy a global domination invariant at all times—and without requiring any annotations from the programmer except at function boundaries.

1That doubly linked lists pose a real challenge is affirmed by top search results for “how to write a doubly linked list in Rust” [18, 41].
For rich object graphs, this increased expressive power poses a challenge: to soundly approximate reservations at run time, the type system must accurately determine to which region each accessed object belongs, and further, which regions are contained within the reservation at run time. This determination is made particularly difficult because reservations can grow and shrink dynamically as threads exchange portions of the object graph.

Our key insight begins by leveraging domination properties in the heap to force isolated field references to dominate [43] their reachable subgraphs, yielding a notion of encapsulation similar to prior work [29]. We then temper this strong and restrictive global domination property with a new focus mechanism inspired by Vault [23]: objects may become temporarily focused, causing their isolated fields’ targets to be explicitly tracked by the type system, and thereby exempted from domination requirements. This weaker heap invariant, which we call tempered domination, allows greater flexibility with lower annotation overhead than in any prior language. It improves on traditional affinity-reference languages by enforcing a tree of regions rather than a tree of objects, allowing more natural structures than are possible in Rust [36]. On the other hand, the focus mechanism skirts the need to maintain a global domination invariant at all times, avoiding the destructive read or swap primitives needed in existing tree-of-regions languages such as L42, LaCasa, Mezzo, and others [3, 4, 17, 26, 28, 46].

Two more novel features enhance expressiveness of our language: (1) a new primitive if disconnected that dynamically determines if a region can be safely split at run time, and (2) expressive function types whose parameters and results need not be dominators.

Our type system can naturally represent many mutable data structures found in prior work, without relying on heavy annotations, unnatural representations, destructive reads, or swap primitives. For example, our type system admits straightforward representations of both doubly linked lists with shared ownership and singly linked lists with recursively linear ownership, improving on a motivating example for much prior work [17, 26, 28] in the first case and offering the celebrated mechanisms of uniqueness and borrowing popularized by Rust [36] in the second.

This work brings together the benefits of two traditional lines of prior work without adding significant complexity. For example, both singly and doubly linked lists support traversal, removal, and insertion functions which look much as they would in an introductory programming class, requiring little annotation or run-time overhead. All these operations enjoy fearless concurrency: added elements may have been received from remote threads and removed elements may be immediately sent to a new thread, all without additional dynamic concurrency control mechanisms or the risk of destructive races. No existing language with fearless concurrency can as naturally express this range of data structures.

Our primary contributions are summarized as follows:

- A new invariant, tempered domination, which allows statically tracked violations of the traditional global domination invariant with a focus construct [23].
- A region-based type system capable of tracking the relationships between regions, without requiring annotations or explicit scopes to do so.
- A formal paper proof of soundness that shows well typed programs have no destructive data races.
- A new primitive to dynamically discover detailed region graphs and expose them to static analysis.
- Expressive function types capable of statically describing complex heap manipulations.
- A type checker implemented in OCaml, and verified in Coq, capable of checking our most complex examples in seconds.

2 A Tail of Two Lists

We begin by explaining key concepts of the new type system, using two linked list implementations as guiding examples.

2.1 Reservations and Tempered Domination

Our language prevents destructive races by dividing the run-time heap into a set of disjoint reservations, one per thread. A thread’s reservation is the portion of the heap that it may access at any particular time. By keeping reservations disjoint, and ensuring no thread attempts to access an object outside its reservation, we guarantee freedom from destructive races; in other words, it is reservation-safe.

As the program executes and threads exchange objects, reservations must shift accordingly. When a thread sends an object to another thread, its reservation must lose access to that object’s reachable subgraph, which includes the object itself as well as all objects transitively reachable from it. Conversely, when a thread receives an object, its reservation expands; the thread gains access to the object and its reachable subgraph.

<table>
<thead>
<tr>
<th>struct sll_node { iso payload : data; iso next : sll_node?; }</th>
<th>struct dll_node { iso payload : data; next : dll_node; prev : dll_node; }</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct sll { iso hd : sll_node; }</td>
<td>struct dll { iso hd : dll_node? }</td>
</tr>
</tbody>
</table>

Figure 1. A singly linked list and circular doubly linked list. Fields are not nullable by default; the ? annotation on types indicates that this field stores a “maybe” of the appropriate type, effectively making it nullable. The iso keyword enforces transitive domination.
The key challenge is ensuring reservation safety at compile time. Consider, for example, a linked list containing some abstract payload type data, used as a messaging queue to communicate with other threads. Two possible definitions of such a list are found in figure 1. While these code examples are simple, they expose two key challenges: the ability to represent cyclic data structures, and the ability to traverse trees of unique references. In order to safely add objects received from other threads to either list, or to remove objects from either list to send to other threads, the compiler must reason about reachability and aliasing, both between the list nodes and their payloads, and between the list nodes themselves.

To make this reasoning tractable for both the compiler and the programmer, our system relies on transitively dominating references: references which lie on all paths from the root of the object graph to all objects transitively reachable from that reference. These references are dominators [43] of entire subgraphs; therefore, a thread which loses access to such a reference, for example by sending it to another thread, also loses access to its reachable subgraph. Hence, marking only this single reference as invalid maintains reservation safety. We use the keyword iso (“isolated”) to describe fields which contain transitively dominating references, thereby exposing knowledge of domination in the object graph to the type system. Looking back to the example code in figure 1, we see that iso appears on the list payloads in both linked list implementations, and that it also appears on the list spine itself in the case of the singly linked list, indicating that the only way to initially reach a singly linked list node is from its predecessor.

If all iso fields contain transitively dominating references, a property we call global domination, then we can safely reason about separation in the heap when accessing such data structures. But global domination is too strong a property to be enforced at all times. For example, consider the code in figure 2, which, given the head node, attempts to remove the final element from a singly linked list, returning a dominating reference. The caller of remove_tail may leverage the separation between the removed node and list parameter to, for example, safely send the removed node to a distinct thread without losing access to the list itself.

In implementing this function, this code first attempts to dereference the argument’s next field, storing it in the variable next. It then checks if next is the tail of the list, removing it from the list and returning its payload if so. Otherwise, it recursively calls remove_tail on the next element. Note something surprising: this code violates global domination! Both the next variable and the list parameter hold references to sll_node’s iso-declared (hence dominating) next field.

In fact, performing a non-destructive traversal of this list while enforcing global domination over all next fields is impossible; all such traversals will require at least a “cursor” variable pointing at the current position in the list, which will necessarily alias the next pointer of that position’s predecessor.

Our language thus does not enforce a traditionally strict global domination invariant; rather than forcing references stored in iso fields to always be transitively dominating, we temper this requirement with a type-level mechanism that explicitly tracks the targets of some references, requiring transitive domination for exactly those references in iso fields which are not explicitly tracked by the type system. We call this weakened property tempered domination.

Tempered domination generalizes prior work that relies on global domination [26–28, 46]. Crucially, tracking, and indeed the decision of which references to track, occurs without explicit user instruction—requiring annotations only at function boundaries. When we describe the mechanisms in place for preserving tempered domination in the remainder of this paper, we refer to transitively dominating references as simply dominating references.

2 Aliasing and Reachable Subgraphs

While an otherwise untracked iso field in some object o is guaranteed to contain a dominating reference, it is not in general guaranteed that o itself is uniquely referenced; in fact many aliases of any given object may be accessible at any particular time. When checking an iso field dereference, it is therefore necessary to ensure the program has not already accessed that same object’s iso field from some other alias.

For example, consider the circular doubly linked list implementation from figure 1. Figure 3 illustrates two possible instances of this list; note that a list of size 1 is represented by a single list node whose prev and next pointers are self-references.

2This is in contrast to existing systems [46], in which similar code would still associate the tail with the list even after returning it, forever entwining the fate of the tail with that of the list.
As with the singly linked list, we might wish to remove the tail from this circular doubly linked list; our first attempt to do so is in figure 4. This code takes advantage of the circular structure of this list, jumping straight to the end via \( \text{hd} \)'s prev pointer. After patching the list pointers to exclude the tail node, we return the iso-annotated tail.payload reference. As in the singly linked list, this function has been declared to return only dominating references, so the caller of remove_tail should be able to use this payload freely without regard for its former attachment to the list\(^3\).

Sadly, this code contains an error. When passed a list of size 2, this code functions as expected; the tail node is excised from the list, removing all external references to the payload except the one returned from the function itself. But when passed a list of size 1, the code behaves differently: \( \text{hd} \) and \( \text{hd}.\text{prev} \) are in fact the same object (fig 3), rendering ineffective the assignments that attempt to remove it from the list. Here, the returned payload actually isn’t a dominating reference; the list retains the same shape as before, and still provides access to the returned payload. While the programmer could eliminate this error by swapping the payload with a dummy value, that fix is undesirable. It would satisfy the type checker, but not remove the bug—replacing a static error with a dynamic one when the dummy value is later unexpectedly encountered.

The correct fix for figure 4 is to add code which handles lists is of length one, perhaps by adding an if-statement. But while this may be sufficient for the programmer to know the size of the list a priori, an if-statement alone would not be enough to allow the type system to make that same deduction.

To solve this, our work introduces a new primitive conditional form called if disconnected. This conditional performs a run-time check to establish if its arguments’ reachable subgraphs are non-intersecting; if they are, it enters the first branch, and otherwise enters the else branch. We see this construct in use in figure 5. Here, the existing logic is enhanced by replacing what was once a plain return of tail.payload to a call to if disconnected, returning tail.payload when it has been successfully disconnected in size 2+ cases, and returning the head’s payload in the size 1 case. Note that the programmer must manually repoint the tail’s next and prev fields away from the remainder of the list, as disconnection is a symmetric property: it is just as essential that tail cannot reach head as it is that head cannot reach tail. Additionally, the type system does not know which of \( \text{hd} \) and \( \text{tail} \) connect to \( \text{hd} \), necessitating that \( \text{hd} \) be reassigned even in the then branch.

Despite its dynamic nature, the run-time complexity for if disconnected is quite reasonable—in this example, it would only require reading the metadata of a single object. Notably, the new if disconnected mechanism cannot be approximated by mechanisms in similar prior work.

3 A Small Language with Dynamic Reservation Safety

We formalize our work as a small core concurrent language with mutable objects, passed by reference.

3.1 Syntax

The syntax of the language can be found in figure 6. Beyond standard imperative constructs, structures, and a first-class “maybe” construct, two novel features stand out: the if disconnected primitive and blocking messaging primitives send-\( \tau \) and recv-\( \tau \).
3.2 Semantics

Figure 7 presents selected rules of the small-step semantics for a single thread; explicit concurrency constructs are added in section 7. The only values are locations. The small-step configuration is largely standard, including a store \( s \) mapping locations to objects, a stack \( t \) mapping variable names to locations, and an expression \( e \) which is evaluated with reference to the store and stack.

The final element of the configuration, \( d \), is not standard; this context models the (dynamic) reservation and is consulted whenever a location is used. For example, rules E2 - Variable-Ref-Step and E5a - Final-Reference-Step – Variable check \( d \) to confine variable and field reads to locations within the reservation, and E8 - Assign-Var-Step and E7a - Final-Assignment-Step – Variable check \( d \) to confine variable and field assignments similarly. If any expression attempts to read or write locations that are not in the current reservation, no rules apply and the program cannot step; the program intentionally “gets stuck.” By augmenting the small-step semantics with this pervasive dynamic reservation check, we can be guaranteed that—provided reservations are always disjoint—no program can destructively race. In section 4 we introduce a type system for which we have section 4, we introduce a type system for which we have

4 Type System

The type system is built around maintaining tempered domination: untracked \( \text{iso} \) fields always dominate their reachable subgraph. To establish this invariant, the type system must be able to determine when two different isolated fields may be aliases. For example, in the doubly linked list example from figure 3, the type system must recognize that \( h.d \) and \( h.d.tail \) may be aliases, and so \( h.d.payload \) and \( h.d.tail.payload \) may be as well. It must also ensure that operations which remove an object from the current thread’s reservation also render all aliases of this object statically unusable.

4.1 Regions

To track aliasing, the type system uses regions [49] to describe disjoint subgraphs of the overall object graph, statically associating each reference with a region in which its target lives. By ensuring that all possible references to the same object are labeled with the same region, the type system can use a set of regions as a conservative compile-time approximation to a run-time reservation. When an object is lost from the reservation, the type system invalidates all references to that object by preventing the use of any references that target its region. Effectively, the type system treats each region as an affine resource which is consumed by reservation-shrinking operations on its constituent objects.

For example, figure 8 circles regions in the doubly linked list instances of figure 3. Entire list spines lie in the same region, which causes the static error from in original attempt: both \( h.d \) and \( h.d.next \) are in the same region, so the type system always treats them as potential aliases.

4.2 Focus

The tempered domination invariant requires that untracked \( \text{iso} \) fields must dominate their reachable subgraph, while tracked \( \text{iso} \) fields are unrestricted. Over the course of program execution, untracked \( \text{iso} \) fields may become tracked, and tracked \( \text{iso} \) fields may in turn become untracked. To allow tracked \( \text{iso} \) fields to be safely untracked, our type system ensures that all tracked \( \text{iso} \) fields have statically known target regions. To avoid unsoundness, we must ensure that potential aliases do not have conflicting static tracking information. To this end, we introduce a focus mechanism, which allows variables to become tracked only in regions in which no other variables are currently tracked. Since variables from distinct regions are necessarily distinct, this ensures no \( \text{iso} \) field ever becomes tracked via multiple aliases. This non-aliasing behavior is formalized as invariant I6 in the appendix.
The variable typing context $\Gamma$ is a largely standard binding environment recording the type and region of variables; the heap context $\mathcal{H}$ is interpreted as a set of tracking contexts of the form $r^\circ(X)$. Each tracking context begins with a region capability $r$, the complete set of which serves to conservatively approximate the dynamic reservation. Were our tracking contexts to contain only this $r$, they would match the tracking context of LaCasa [28, 29]; indeed, several rules—those which introduce, check, and eliminate regions—require only this level of detail.

### 4.4 Expression Typing with Tracking Contexts

In addition to the top-level structure describing the set of tracked regions in $\mathcal{H}$, the full tracking context $r^\circ(x^0[f \mapsto r, \ldots])$ includes a description $x^0[f \mapsto r, \ldots]$ of the region structure discovered by our focus mechanism: namely, tracked variables $x$ in the region $r$, where each $f \mapsto r$ maps tracked fields $f$ to their target regions $r$. Both variables and regions also include a pinning annotation described by the metavariable $\circ$. Pinning a region (resp. variable) prevents any new variables (resp. iso fields) from becoming tracked in that region (resp. variable). Pinning is necessary when the typing context might only have partial static information about the heap, and allows the type system to express abstraction over $\mathcal{H}$.

Figure 10 shows how the context $\mathcal{H}$ is used to type expressions. First, note that $\mathcal{H}$ prevents the type system from confusing an iso field with potential aliases; as shown in T5 - Isolated-Field-Reference, no iso field of some variable may be accessed unless both that variable and its field are already present in the tracking context, and the recorded region targeted by that field is itself present in $\mathcal{H}$.
This tracking context also allows iso fields to be freely reassigned, even if doing so would create cycles in the object graph. This is safe because tempered domination requires domination only on untracked iso fields; fields explicitly mentioned in H are exempt. Consider, for example, type-checking x.f = e with T7 - Isolated-Field-Assignment. This rule places no restrictions on e beyond ensuring that it type-checks, and that x.f remains valid and tracked after checking e. The rule simply updates x.f’s tracking information in the output context.

We sometimes require the tracking context of a region to be empty, containing no tracked variables and thus no tracked fields. As tempered domination weakens global domination only for tracked isolated fields, empty tracking contexts prove that every iso field within that region contains a dominating reference, and thus is safe to transmit between threads via T16 - Send (which requires an empty context) and T17 - Receive (which assumes one).

Note that rules such as T10 - New-Loc, which add regions, variables, or fields to existing contexts, enforce freshness because well-formed contexts cannot duplicate bindings. A notable absence in figure 10 is any rule which introduces or eliminates elements in a tracking context. This role is played by TS1 - Virtual-Transformation-Structural, which allows invariant-preserving virtual transformations to be performed on static contexts.

4.5 Virtual Transformations

Rule TS1 serves to expose a rich language of virtual transformations specified by the V rules in figure 11. These rules manipulate H to match the requirements of the syntax-directed T rules. For example, consider the program x = new-r(τ); x.f. After type-checking the first expression in this sequence via T10 and T8 - Assign-Var, we could obtain the following typing judgment: 

\( x : \perp \perp \tau + \text{new-}r(\tau) \)

If we then moved on to checking x.f, rules T3 - Sequence and T5 - Isolated-Field-Reference would seem natural yet be inapplicable. This is because the output context of new-r(τ)’s derivation has the form \( \tau \); x : \( \tau \), but the field reference rule requires a context like \( \tau \); x [f ↦ r] r’; x : \( \tau \).
Focus

V1 - Focus

V2 - Unfocus

V3 - Explore

V4 - Retract

V5 - Attach

Figure 11. Virtual Transformation Rules.

4.7 Abstraction by Framing and Pinning

Figure 12 introduces rule \(\text{TS2 - Framing-Structural}\), which exposes our second non-syntax directed typing rule: framing. Framing allows our typing rules to ignore irrelevant portions of the static contexts \(\mathcal{H}\) and \(\Gamma\), letting the type checker temporarily frame away regions in \(\mathcal{H}\), variables in \(\Gamma\), and portions of tracking contexts.

While framing is a standard feature when reasoning about separation [45], its inclusion in our system is complicated by tempered domination. Naively allowing variables within tracking contexts to be framed away would seemingly violate tempered domination; it would take an invariant-satisfying context with explicit domination exceptions, and replace it with one in which no record of those exceptions appears—without making corresponding changes to the heap.

The pinning annotation (4.4) solves this problem. Pinning elements of a tracking context indicates that those elements have partial information: that is, it cannot be assumed that untracked iso fields of a pinned region or variable contain dominating references. By leveraging pinning, we can admit framing rules which weaken elements of tracking contexts without introducing unsoundness. Since a pinned context may only be obtained by framing, any pinned context always approximates some fully unpinned context, which avoids the need to further temper dominated information in our proofs of progress and preservation.
A function abstraction should capture all available static variables and regions; and (3) the region tracking information about its arguments as input, and allow arbitrary transformations of that information as output. Following this principle, our system provides function types \( \left( H, \Gamma \right) \Rightarrow \left( H', \Gamma' \right) \) with three main components: (1) an input pair \( \left( H, \Gamma \right) \) in which \( \Gamma \) captures the function’s parameters with their expected region and type, and \( H \) captures the tracking contexts of those regions, possibly closed over the tracked isolated references in those contexts; (2) an output pair \( \left( H', \Gamma' \right) \) which captures the final state of the same variables and regions; and (3) the region \( r \) and type \( \tau \) of the returned value. Rules \( T0 - FUNCTION-DEFINITION \) and \( T9 - FUNCTION-APPLICATION \) integrate these function types. \( T0 \) requires that the function body be well-typed with the given input and output contexts, and \( T9 \) requires that, up to renaming of variables and regions, the call site’s \( H, \Gamma \) match the function’s input \( H, \Gamma \).

At first glance, this reliance on an exact match of contexts may appear restrictive; however, function declarations need only include elements in \( H \) and \( \Gamma \) relevant to that function’s execution. Pinning annotations in the function declaration allow call sites to produce an exact match by using \( TS2 - FRAMING-STRUCTURAL \) to frame away any irrelevant portions of the application context.

### 4.8 Introducing a Function Abstraction

A function abstraction should capture all available static tracking information about its arguments as input, and allow arbitrary transformations of that information as output. Following this principle, our system provides function types \( \left( H, \Gamma \right) \Rightarrow \left( H', \Gamma' \right) \) with three main components: (1) an input pair \( \left( H, \Gamma \right) \) in which \( \Gamma \) captures the function’s parameters with their expected region and type, and \( H \) captures the tracking contexts of those regions, possibly closed over the tracked isolated references in those contexts; (2) an output pair \( \left( H', \Gamma' \right) \) which captures the final state of the same variables and regions; and (3) the region \( r \) and type \( \tau \) of the returned value. Rules \( T0 - FUNCTION-DEFINITION \) and \( T9 - FUNCTION-APPLICATION \) integrate these function types. \( T0 \) requires that the function body be well-typed with the given input and output contexts, and \( T9 \) requires that, up to renaming of variables and regions, the call site’s \( H, \Gamma \) match the function’s input \( H, \Gamma \).

At first glance, this reliance on an exact match of contexts may appear restrictive; however, function declarations need only include elements in \( H \) and \( \Gamma \) relevant to that function’s execution. Pinning annotations in the function declaration allow call sites to produce an exact match by using \( TS2 - FRAMING-STRUCTURAL \) to frame away any irrelevant portions of the application context.

### 4.9 A Usable Function Syntax

The \( H \) and \( \Gamma \) contexts are complex and would be onerous to expect a programmer to write down directly. We therefore expose an alternate surface syntax for describing function types. This syntax is intended to be more intuitive for programmers, while maintaining the full expressive power of the type system.

Two principles drove the design of this user-facing syntax. The first is that programmers should never directly mention regions, as their direct inclusion in syntax here could lead programmers to expect them to be usable elsewhere in the program. The second is to lean on good defaults that match programmer expectations; only exceptional code should require additional annotation.

Following the principle of good defaults, for unannotated functions, three assumptions hold:

- At input, each parameter comes from a distinct unpinned region with no tracking context.
- At output, each parameter remains in that region, which again must be unpinned and empty.
- A returned result is in its own unpinned, empty region.

These assumptions suffice to write functions that perform in-place manipulations of tree-like isolated data structures. Notably, function requirements are only checked at the beginning and end of each function body; function bodies which only temporarily deviate from these expected properties still require no annotation.

In lieu of presenting the full surface language for function declarations, we highlight interesting cases by example in the style of section 2. The concat function in figure 14 illustrates an example of the most commonly needed annotation on functions in our system: consumes, which indicates the annotated input is consumed by the function. A function can consume a parameter in more than one way. Intuitively, it could send that parameter to another thread; in the case of figure 14, the parameter is retracted into an iso field of the other parameter, concatenating the lists together and becoming wholly owned by the larger list in the process. Interestingly, our full implementation of a singly linked list—consisting of 8 functions—requires only this consumes annotation, and even then in just two places.

But there is need for function syntax more expressive than just consumes annotations. Consider for example the get_nth_node function in figure 14. This function takes a circular doubly linked list and returns a mutable reference...
Figure 13. Function application and definition typing rules

```plaintext
def concat(11, 12 : sll_node) : unit consumes 12 {
  let some(11_next) = 11.next in {
    concat(11_next, 12);
  } else (11.next = some 12;)}
def get_nth_node(1 : d1l, pos : int) : d1l_node?
  after: 1.hd ~ result {
    let some(node) = 1.hd in {
      while (pos > 0) {
        node = node.next;
        pos = pos - 1;
      }; some(node)
    } else (none)
```

Figure 14. Concatenating two lists, and returning the nth node of a doubly linked list

5.1 Heuristics for Virtual Transformation Search

As discussed in section 4.6, the TS1 rule in our type system, governing focus, explore, and all other virtual transformations necessary to transform the heap context, is not syntax-directed. Several heuristics implemented by the type checker keep type checking efficient in practice. In particular, we aim to avoid backtracking search when unifying the branches of a conditional.

At the heart of the difficulty in unifying the typing contexts of branches is the information loss associated with key virtual transformations such as V2 - UNFOCUS and V5 - ATTACH. Unification can thus be viewed as the problem of inferring which linear resources must be preserved to type-check a given program suffix. By employing liveness analysis of variables and isolated fields as a unification oracle, our checker can verify our largest examples in a handful of seconds. When necessary, our tool still falls back to search. Other approaches—such as user annotations or an external constraint solver—may be useful for pathological cases. More details appear in the appendix.

5.2 Efficiently Checking Mutual Disconnection

We implemented a version of the if disconnected check (introduced in section 3.2) that is efficient based on two usage assumptions. The first assumption is that data structure designers prefer to keep regions small when possible, placing the iso keyword at abstraction boundaries—for example, collections place their contents in iso fields, as we do in figure 1. The second assumption is that if disconnected is commonly used to detach a small portion of a region—often as small as a single object (as in figure 5).

Following these assumptions, we propose a two-step process for the efficient implementation of if disconnected. First, store a reference count which tracks immediate heap references stored in non-iso fields of structures. This stored reference count is updated only on field assignment, and does not need to be modified—or checked—on assignment to local variables, function invocation, or at any other time. Thus, it is lighter-weight than conventional reference counts.

Second, the if disconnected check itself is implemented via interleaved traversals of the object graphs rooted by its two arguments, ignoring references which point outside the
current region, and stopping when the smaller of the two has been fully explored (or a point of intersection has been found). During this traversal, the algorithm counts the number of times it has encountered each object, assembling a traversal reference count. At the end of the traversal, it compares this traversal reference count with the stored reference count, concluding that the object graphs are disconnected if the counts match, and conservatively assuming that they remain connected if the counts do not match.

The soundness of this strategy relies on two things: tempered domination enforced on iso fields by the type system, and accuracy of the stored heap reference counts. The typing rule for if disconnected ensures that its arguments come from the same region, and that nothing within that region is tracked. Each untracked iso field roots a distinct, fully independent object graph; thus no object beyond an iso field can be the first point of intersection between if disconnected’s arguments. This eliminates any need for the traversal to search beyond an iso field.

Our choice to terminate the traversal after only the smaller graph is explored, meanwhile, is justified by reference counts. The fear here is that, by terminating our exploration early, we may have missed some path from the larger object graph into the smaller. Such a path would necessarily include an unexplored reference targeting an object in the smaller graph. The existence of this unexplored reference would be reflected in the stored reference count, causing the stored reference count to exceed the traversal reference count. Can this check be done efficiently? For cases which follow our expected use patterns—like the one in figure 5, where the smaller graph’s non-iso references point only to the object itself—the traversal terminates immediately after encountering only a single object, or a small number of closely linked objects. But in the worst case, this check may involve traversing an entire region of arbitrary size. Such a traversal would cut against the intended use-cases of if disconnected; we would thus consider these cases more likely to arise as a result of buggy code than of intentional design. In these buggy cases, our if disconnected check would still improve on systems which rely on destructive reads, replacing unexpected run-time crashes later in the program with a static error (or an unexpectedly slow no-op) at the point the bug actually occurs. Returning to figure 5, even were we to introduce a bug by failing to correctly disconnect the object graph—for example by omitting the assignments which immediately precede the if disconnected check—the resulting traversal would incur nearly no additional cost, with if disconnected’s check still terminating after only two objects are encountered.

6 Correctness

We have discussed in detail the surface syntax and small-step semantics of our language, whose rules guarantee that any attempt to access a location outside the dynamic reservation $d$ will arrest the program in a “stuck” state, and we have presented typing rules with a complex context $H$, which statically models capabilities to access a shared heap. The missing piece of the puzzle is a run-time invariant using the information in $H$ to guarantee that well-typed programs never encounter that stuck state. This is easily phrased:

**Definition (Invariant 11 - Reservation-Sufficiency).** All locations that could be the result of stepping a well-typed expression are contained in the dynamic reservation $d$.

An immediate consequence of 11 is that any variables bound in $\Gamma$ to a region tracked in $H$ are mapped (by the dynamic stack $s$) to a location in $d$. This is because $T2$ - VARIABLE-REF guarantees well-typed access to any such variables, and $E2$ - VARIABLE-REF-STEP steps them directly to their bound locations. Similarly, transitive targets of fields are in $d$. Invariant 11 is thus exactly the missing piece to bind well-typedness to reservation safety. Naturally, its preservation as programs step is a nontrivial proof goal, so we introduce a second invariant 12 which implies 11 and is closer to the formalisms of the language:

**Definition (Invariant 12 - Tree-Of-Untracked-Regions).** Any two paths in the dynamic heap that begin in a tracked region and terminate at the same location traverse the same sequence of untracked isolated references.

This invariant is fundamental because it directly encodes the core tempered domination invariant: in particular, that beyond our statically tracked set we can assume that all iso fields contain dominating references.

To further motivate 12, recall that the accepted static evidence for the separation of two objects is their presence in separate regions (consider $T16$ - SEND), and that untracked isolated references are always assumed to point to untracked regions (see $V3$ - EXPLORE). Thus, a necessary condition for safety is that locations serving as the target of untracked isolated references may never be bound to variables in tracked regions; otherwise, that variable could be accessed even after is dropped from the reservation. 12 captures this condition.

The appendix formalizes both 11 and 12, as well as additional formal invariants encoding expected agreement between the static and dynamic contexts. All of these invariants together capture the notion of a sound configuration used in the following theorems.

**Theorem 6.1 (Progress).** Given the well typed expression $H; \Gamma \vdash e : r \rightarrow H'$, there exists a step $(d, h, s, e) \xrightarrow{\text{eval}} (d', h', s', e')$.

**Theorem 6.2 (Preservation).** Given the well typed expression $H; \Gamma \vdash e : r \rightarrow H'$ and step $(d, h, s, e) \xrightarrow{\text{eval}} (d', h', s', e')$, there exist $H, \Gamma$ such that $H; \Gamma \vdash e' : r \rightarrow H'; \Gamma'$ and the configuration $(H, \Gamma, d', h', s')$ is sound.
Proofs of 6.1 and 6.2 are provided in the appendix. Together, these theorems imply that invariants 11 and 12 hold across the execution of a well typed program. This establishes tempered domination is preserved, and it establishes the core safety property of our system: in a well typed program, no thread accesses memory outside its reservation.

7 Concurrency

The results from section 6 show that our system can guarantee the reservation safety of sequential programs. Importantly, this result also means that concurrency is safe.

We model general, message-passing concurrency through the expressions send−τ(e) and recv−τ() (T16 - SEND and T17 - RECEIVE in the type system of section 4).

The concurrent configuration consists of a single shared heap h, and an n-tuple of threads, each with its own reservation d_i, variable store s_i and expression e_i currently under evaluation. Soundness of a concurrent configuration consists of the respective soundness and well-typedness of each thread’s e_i with respect to the configuration (d_i, h, s_i), along with pairwise disjointness of the reservations d_i.

Stepping a concurrent configuration occurs by stepping an individual thread, by updating that thread’s d_i, s_i, e_i as well as the shared h, or by stepping two threads together that have reached a send−τ/recv−τ pair. This stepping rule is illustrated in figure 15. It steps in the context of the shared heap h, but only updates the respective reservations and expressions of the sending and receiving threads. In particular, it identifies the location l_root that the sending thread has chosen, reads h to identify the set d_road of locations that are live (i.e., reachable) from l_root, and steps if d_road is entirely contained within the sending thread’s reservation, transferring it to the receiving thread’s reservation along with access to the location l_root.

Progress and Preservation in the concurrent configuration are also stated and proved in the appendix, notably establishing that no thread’s soundness relies on h outside of its reservation d_i, and that the rules T16 and T17 are sufficient to conclude EC3 - COMMUNICATION-PAIRED-STEP can be applied without getting stuck on ownership transfer, yielding sound post-transfer configurations for both threads.

8 Expressiveness

To explore the expressiveness of the type system, we have written thousands of lines of algorithmic code, data structure manipulations, and experimented with function abstractions ranging from trivial to pathological. Large samples of this code are presented in the appendix, including complete singly and doubly linked lists and a red–black tree.

Our experience suggests that functions in our language place no unnatural restrictions on common coding patterns, requiring annotations only when the iso keywords are added to struct definitions. Further, even functions that manipulate structs with iso fields need no annotation unless they take or return object graphs that violate the tempered domination invariant—for example, overlapping object graphs and non-tree object graphs.

We have found that functions whose arguments’ object graphs overlap (like the get_nth_node example) are usually easy to annotate, while functions that deviate from tempered domination at function boundaries are improved by signature-level annotations describing the shape of their isolated object graph. As an example, the shuffle function of the appendix’s red–black tree takes 7 tree nodes in an arbitrary, possibly deeply aliased state and returns them with a fixed, tree pointer structure. Expressing that information in the signature provides a level of static safety usually found only in dependently typed languages.

Thus, our experience suggests that besides offering strong safety guarantees, this language is intuitively usable.

9 Related Work

The type system we propose owes much to the rich history of related language designs. In particular, it exploits innovations from several important lines of research: ownership types and capabilities, regions, and linear types (and linear regions). We now attempt to broadly characterize notable work from each line of research, and discuss how our work differs.

9.1 Ownership Types and Nonlinear Uniqueness

While we use the terminology of focus [23] and regions [48], the closest antecedent to focus is in CQual [1, 25], while the closest cousin to our regions is ownership contexts [16]. The primary difference between our regions and ownership contexts is that ownership contexts are fixed: objects forever live within a single ownership context, and ownership contexts cannot be merged, consumed, or generated on the fly.

Recognizing these limitations, later work introduced the ability to mix ownership with uniqueness [2, 3, 8, 31, 38, 46]. These languages all enforce uniqueness strictly: a unique reference is the only reference that points to its referent. Clarke and Wrigstad weakened this constraint by introducing the idea of external uniqueness, and with it the idea of a dominating reference: an externally unique reference is traversed on all paths from roots to the object to which it refers [14, 15].

Externally unique references are similar to iso fields, but iso fields dominate all objects reachable from their target, while—in its original formulation—external references dominate just their target. This weaker invariant prevents externally unique references from implying transitive ownership. Other variations on ownership also exist; some work makes owning objects explicit, abstracts them with capabilities, or views them as modifiers [9, 13, 17, 20, 28, 29, 39, 40].

Of particular note is the LaCasa language of Haller and Odersky [28, 29], which our work subsumes. LaCasa’s surface language (and accompanying annotation burden) are

Mae Milano, Joshua Turcotti, and Andrew C. Myers
While Rustaceans have discovered a variety of clever ways to implement the invalidate potential aliases. These systems cannot efficiently using indices into a linearly owned array as a stand-in for references. Recent work into using "ghost cells" to achieve techniques often resemble how our system would behave simulating cyclic data structures within its type system, those languages would not be able to directly represent the doubly cyclic data structure patterns is encouraging, but remains above the annotation budget that we believe is desirable for such common data structures.

9.2 Linear Systems and Regions

Since initially popularized by Wadler [51], many linear languages have been proposed [21, 36, 42, 47, 50, 52] which can prevent destructive races without relying on destructive reads or swapping—but at the cost of making direct representations of graph data structures cumbersome. These languages would not be able to directly represent the doubly linked list from figure 1. Much of the recent interest around this class of languages has centered on Rust [36], the first such language to gain widespread adoption [32, 34, 44, 53]. While Rustaceans have discovered a variety of clever ways to simulate cyclic data structures within its type system, those techniques often resemble how our system would behave were one to have a single object per region; complex graphs are possible, but the cost is a dramatic increase in static tracking, much of it borne directly by the user in the form of extra annotations, a reliance on unsafe code, or "clever hacks" like using indices into a linearly owned array as a stand-in for references. Recent work into using "ghost cells" to achieve

Figure 15. Stepping send/recv pairs in the concurrent configuration

<table>
<thead>
<tr>
<th>h + (d, e; d, e)</th>
<th>→ [comm-eval]</th>
<th>(d, e; d, e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d_{app} = \text{live-set}(r(\cdot); \cdot : l : r \tau; h) )</td>
<td>( h + (d_s \oplus d_{app}, E_a[\text{send-r}(l_{\text{root}})]; d_s, E_a[\text{recv-r}(l)]); )</td>
<td>( \text{comm-eval}</td>
</tr>
</tbody>
</table>
problems interacting safely with Mezzo’s novel take on destructive reads. It is unclear if Mezzo’s adoption mechanism allows the formation of arbitrary graphs, or only DAGs, but it is difficult to see how a doubly linked list could be implemented in Mezzo without relying on implicit nulling.

### 9.3 Immutability and Fractional Permissions

Several related systems offer the ability to temporarily share mutable objects with immutable references, and to recover mutability once all shared references’ lifetimes have ended [17, 23, 27]. This banner feature of Rust [36] appears in Mezzo [4] and was added to Vault through Boyland’s work on fractional permissions [11]. M# [27], an evolution of Sing# [22], also features recovering mutability—later generalized by Pony [17] and L42 [26].

To determine the lifetime of concurrently shared immutable references, these systems all support mutability recovery only when using structured parallelism or explicit recovery scopes: all possible aliases, including those passed to threads, will have been reclaimed by a statically known program point—usually when all other threads involved in communication have died. In contrast, our simple, unstructured send and receive mechanism cannot track which references are transmitted. Threads have no lifetime, so it is impossible to know if a reference sent to another thread is ever returned.

Instead, we expect to take the approach outlined in Galiffrey [37], in which a dynamic mechanism manages shared immutability and mutability by relying on replication. Alternatively we could leverage our equivalent of “lending” references to functions during a function call; making those calls asynchronous, and providing a built-in future mechanism by which “lent” references may be returned, is a promising avenue by which recoverable mutability may be supported.

### 9.4 Significant Complexity

Several systems manage to ensure reservation safety and avoid implicit null (or swap), but introduce significant user-facing complexity [8, 12, 13, 17, 33]. These languages frequently feature explicit, exact region or ownership annotations, provide a type parameterization mechanism which allows the creation of classes whose ownership or region information is determined at instantiation time, or rely on a multitude of reference qualifiers capable of discussing exactly how various objects may relate in the object graph. While such systems are quite flexible, they force the user to reason directly about concepts, like regions and region membership, which we intentionally keep implicit. Here the complexity does not appear to be incidental; it is not clear how to identify a “simple core” language that would be complete on its own. Indeed, our experience designing this type system speaks to the speed at which complexity can creep in from apparently innocuous design choices.

<table>
<thead>
<tr>
<th>Language</th>
<th>sll</th>
<th>dll-repr</th>
<th>Simple</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rust</td>
<td>✓</td>
<td>×</td>
<td>~</td>
</tr>
<tr>
<td>Unique</td>
<td>✓</td>
<td>×</td>
<td>~</td>
</tr>
<tr>
<td>Vault</td>
<td>✓</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>Mezzo</td>
<td>~</td>
<td>~</td>
<td>✓</td>
</tr>
<tr>
<td>LaCasa</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>OwnerJ</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Pony</td>
<td>~</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>M#</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>This paper</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

### 9.5 Comparison with Closely Related Work

The systems that come closest to matching our design goals are summarized in table 1. In the “sll” column, systems are marked that can implement remove_tail from our singly linked list (without requiring O(list-size) object mutations). The “dll-repr” has a check for systems that can directly represent the doubly linked list at all, and the “simple” column marks systems which require few annotations for straightforward implementations of common list mutations. To the best of our knowledge, no previous system is able to represent remove_tail from a doubly linked list without relying on destructive reads or a swap primitive. Finally, the “OwnerJ” row captures the close descendants of original ownership type systems, including PRFJ [8] and AliasJava [2] (section 9.1), while the “Unique” row captures the limitations of type systems in the style of Wadler’s popularization [51].

### 10 Conclusion

We started by observing that expressiveness-limiting heap invariants and intimidating annotations are fatal flaws in existing safe concurrency approaches. Our core insights are that these invariants can be weakened without losing power as long as they stay recoverable through virtual transformations, and that careful type-system design can preserve decidability in lieu of annotation. The result is a type system that replaces stricture with flexibility and caution with fearlessness—a new sweet spot in this design space that lowers the cost of safe concurrency and opens promising avenues for future work.

### Acknowledgments

We thank the anonymous reviewers, Rolph Recto, Tom Maigrino, Gabriel Matute, Justin Lubin, Marco Servetto (author of [46]), and our shepherd, Ralf Jung, for their feedback and suggestions. This work was supported by the National Science Foundation under Grant No. 1717554.
References


