Fully-Automatic Type Inference for Borrows with Lifetimes

WILLIAM BRANDON*, Massachusetts Institute of Technology, USA
BENJAMIN DRISCOLL*, Stanford University, USA
FRANK DAI, Unaffiliated, USA
JONATHAN RAGAN-KELLEY, Massachusetts Institute of Technology, USA
MAE MILANO, Princeton University, USA
ALEX AIKEN, Stanford University, USA

We present a new pure functional language and type system with borrows with lifetimes, and a corresponding fully-automatic type inference procedure. Inference provides users the performance benefits of borrows with lifetimes without requiring user annotation. If the user's program cannot be typed, inference inserts a handful of reference count operations so that it can be typed. We provide a heap semantics for our borrowing language and prove a soundness theorem, guaranteeing that well-typed programs do not violate memory safety. We implement our memory management strategy as part of the Morphic language stack and compare it to Perceus, a state-of-the-art reference-counting technique based on linear type inference. We find that our system is able to eliminate almost all reference count operations across a range of programs, reducing reference count increments by 75–100% on all benchmarks with reference count increments under the baseline. As a result, we achieve a 1.48× geomean speedup overall on all benchmarks.

CCS Concepts: • **Software and its engineering** → **Memory management**; • **Theory of computation** → *Type theory*.

Additional Key Words and Phrases: reference counting, ownership, borrowing, type inference

ACM Reference Format:
William Brandon, Benjamin Driscoll, Frank Dai, Jonathan Ragan-Kelley, Mae Milano, and Alex Aiken. 2026. Fully-Automatic Type Inference for Borrows with Lifetimes. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 113 (April 2026), 27 pages. https://doi.org/10.1145/3798221

1 Introduction

Linear type systems support *resources*, i.e., objects that are consumed by operations. Among other things, resources can model physical objects, state machines, or memory. While modeling memory as a resource makes allocation and deallocation points statically apparent, it is quite restrictive without a notion of resource aliases. The Rust programming language [17] features linear types plus *borrowing*, the creation of temporary, statically-tracked resource aliases. Each borrow can be used only for the duration of its (static) *lifetime*, which must end before the original, linear resource (*owner*) is consumed (*moved*).

*Both authors contributed equally to this research.

Authors' Contact Information: William Brandon, Massachusetts Institute of Technology, Cambridge, USA, wbrandon@mit.edu; Benjamin Driscoll, Stanford University, Stanford, USA, bdrisc@cs.stanford.edu; Frank Dai, Unaffiliated, San Francisco, USA, frankdai320@gmail.com; Jonathan Ragan-Kelley, Massachusetts Institute of Technology, Cambridge, USA, jrk@mit.edu; Mae Milano, Princeton University, Princeton, USA, mpmilano@cs.princeton.edu; Alex Aiken, Stanford University, Stanford, USA, aaiken@stanford.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/4-ART113
https://doi.org/10.1145/3798221

As demonstrated by Rust, borrows with lifetimes are expressive enough to efficiently encode a wide variety of programs. Lifetimes allow borrows to be safely returned by functions and stored in data structures. Unfortunately, borrows with lifetimes add significant complexity to the user-facing type system. Furthermore, where borrows with lifetimes cannot express the reason a program is memory safe, one must fall back to *unsafe* code, which is not statically checked for memory safety violations, or a dynamic mechanism, most commonly reference counting via Rust's Rc type. Threading the required Rc through the program can be onerous and create an impedance mismatch with libraries. This situation encourages premature optimization—rewriting code to appease the borrow checker where an Rc would suffice—or pessimization—deep-cloning the relevant object instead of using an Rc or determining the correct borrowing pattern.

We present a system that allows users to write pure, functional code, without the burden of linear types or borrowing, but retain their performance benefits. We infer a borrows-with-lifetimes type and the implied deallocation (*drop*) points fully-automatically. Much as a Rust programmer might fall back on Rc, if a borrows-with-lifetimes type cannot be inferred for a user's program, our algorithm will treat the program as *partial*, and *complete* it by inserting a small number of reference count (RC) increment (*dup*) operations so that it can be given a borrows-with-lifetimes type. As we are focused on the pure setting, we support only the equivalent of Rust's shared/immutable borrows, and do not support exclusive/mutable borrows. In Section 6.6, we discuss how our algorithm can be extended to the impure setting.

We implement our approach as part of the *Morphic* [6] language stack. The algorithm we present is first-order, but our benchmarks include aggressively higher-order programs, such as a parser-combinator-based JSON parser. We handle higher-order user code by pre-processing with lambda set specialization (LSS) [7] to produce first-order programs, see Section 6.1.

Our evaluation shows that borrows-with-lifetimes type inference is able to eliminate almost all of the RC operations in our benchmarks relative to a state-of-the-art baseline where all objects are owned. This baseline is an implementation, in the Morphic compiler, of the Perceus algorithm [22] minus *reuse optimization*, which allows a destructor followed by a constructor to sometimes reuse the same allocation. Highlights from our results are: an 83.1% reduction in RC increments in our trie benchmark, a 100% reduction in our JSON parsing benchmark, and an 85.1% reduction in our metacircular Lisp interpreter benchmark.

We make the following contributions:

- We develop a linear language, $L^{\&}$, with shared borrows and non-lexical lifetimes and the first fully-automatic type inference algorithm for a borrows-with-lifetimes type system.
- We provide a heap semantics for $L^{\&}$ and a proof of the soundness of the type system with respect to this semantics, i.e., that a type checker for $L^{\&}$ can be used to certify the memory safety of programs produced by inference. We do not show that this check will always succeed, i.e., inference will always produce a well-typed result, nor that inference will always find a dup-free solution if one exists. We leave these theorems as future work.
- We implement our technique in the Morphic compiler. On our benchmarks, we achieve a geomean speedup in overall program execution time of 1.48× over a state-of-the-art baseline.

2 Setting and Motivating Example

Consider the following program written in an ML-like language:

$$\begin{array}{l} \textbf{def}\ \mathsf{nth}(\ell : \mathsf{List\ String},\ n : \mathsf{Int}) : \mathsf{Option\ String} = \\ \quad \textbf{case}\ \ell\ \{\ \mathsf{Cons}(h,\ t) \Rightarrow \textbf{if}\ n = 0\ \textbf{then}\ \mathsf{Some}(h)\ \textbf{else}\ \mathsf{nth}(t,\ n-1),\ \mathsf{Nil} \Rightarrow \mathsf{None}\ \}\ \textbf{;} \\ \textbf{let}\ s = \mathsf{Cons}(\texttt{"head"},\ \mathsf{Nil})\ \textbf{;}\ \mathsf{print}(\mathsf{nth}(s,\ 0)) \end{array}$$

The function nth traverses a list of strings, ℓ, and returns the nth element. The program calls nth and prints the result. In a typical functional language, this example would rely on *dynamic memory management*, accepting the overhead of a tracing garbage collector or reference-counting. Prior reference-counting techniques would increment the reference count of (*dup*) each cons cell and each string, then decrement the reference count of (*drop*) each cons cell and all but the final string, which is returned. With access to a borrows-with-lifetimes type system, we can annotate ℓ, h, t, and the returned string as borrows, avoiding any dup or drop operations. Our inference procedure is capable of finding these annotations with no work on the user's part.

Our technique begins with the observation, not unique to this paper, that a handle to a reference counted allocation (an Rc in Rust) can be treated as a resource in its own right, separate from the reference count and payload it points to. Calling dup on an (owned or borrowed) handle increments the reference count of the associated allocation and produces a fresh, owned handle, whose connection to the original handle is not statically tracked. From an owned handle, we can derive any number of borrows without affecting the reference count. Each borrow is statically associated with a *lifetime*, which models the duration for which its owner is guaranteed to remain live. In the dynamics, when an owned handle goes out of scope, the reference count of the underlying object is decremented and the object is deallocated if the count hits zero. In the statics, the lifetimes of the original owner and of all its borrows end, which ensures that the underlying object can no longer be accessed through the discarded handle, though it might be accessible through other handles. Ownership can also be *moved* from an existing owner to a new owner, for example, at a binding site, and this too ends the lifetime of the existing owner. When an owned handle is moved or a borrow goes out of scope, the reference count is not touched.

We will not attempt to borrow types that can be stack allocated, such as Int and Option—they are cheap to copy. Heap allocations, on the other hand, are expensive to copy, so we will reference count them and, where possible, borrow their handles to avoid reference count (RC) operations. The most important examples are arrays and recursive types, which require indirection lest they have unbounded size. In Morphic, the array type is roughly Rust's $\mathsf{Rc}\langle\mathsf{Vec}\langle\mathsf{T}\rangle\rangle$[1] and every recursive occurrence of a recursive type is effectively wrapped in an Rc. Dups arise where an owner is requested, but we cannot fulfill that request with a move because the handle we have is needed again later. So, by passing around a $\&\,\mathsf{Rc}\langle\mathsf{T}\rangle$ (a borrowed handle) instead of an $\bullet\,\mathsf{Rc}\langle\mathsf{T}\rangle$ (an owned handle), we reduce the required number of dups and of drops that are not dynamically no-ops.

Returning to the definition of nth, we can avoid reference counting operations by (statically) borrowing ℓ and returning a borrow of its nth string. Letting $\&\langle\alpha\rangle\,\mathsf{T}$ denote a borrow of a T with lifetime α, nth would have signature:

$$\mathsf{nth}\langle\alpha\rangle(\ell : \&\langle\alpha\rangle\,\mathsf{List}\,(\bullet\,\mathsf{String}),\ n : \mathsf{Int}) \to \mathsf{Option}\,(\&\langle\alpha\rangle\,\mathsf{String})$$

The above signature illustrates the importance of lifetimes in a borrow-with-lifetimes discipline. The strings in the list ℓ are owned by ℓ, so we cannot access them after ℓ is deallocated. The lifetime variable α lets nth tie the lifetime of the returned String borrow to the lifetime of its parameter. This allows the type system, at each call to nth, to see the dependency between the return and the argument and, therefore, to enforce that the returned borrow is not accessed after the lifetime of the argument ends.

The nth function illustrates two additional points. First, we must support borrows nested inside other types. Since we construct the Option inside nth, while we can return its content by borrow, we cannot return the Option itself by borrow, even if we so desire. We need an $\mathsf{Option}\,(\&\langle\alpha\rangle\,\mathsf{String})$.

[1] But, we store the reference count in-line with the array elements.

$$
\begin{array}{llll}
\text{Type} & t & ::= & \mathbf{bool} \mid t_1 \times t_2 \mid \mathbf{rc}\, t \qquad \text{TypeCon}\ \Gamma\ ::=\ \diamond \mid \Gamma, x : t \\
\text{Expr} & e & ::= & \mathbf{true} \mid \mathbf{false} \mid \mathbf{if}\, e\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 \mid (e_1, e_2) \mid \mathbf{projl}\, e \mid \mathbf{projr}\, e \\
 & & & \mid \mathbf{rc}\, e \mid \mathbf{get}\, e \mid \mathbf{let}\, x : t = e_1 \,;\, e_2 \mid x
\end{array}
$$

Fig. 1. Simplified L^{src} syntax.

Second, since every value must have an owner and we pass "head" directly to Cons, Cons must be the owner of "head". Owners might reside on the heap.

Note that nth does not have a principal type. Because we are calling it with s, which is an owned list containing owned strings, for the reasons described in Section 3.3, we cannot type ℓ as $\&\langle\alpha\rangle$ List $(\&\langle\beta\rangle$ String). On the other hand, we cannot directly pass a list of borrowed strings to nth if it is typed as above.

Finally: isn't ℓ used linearly? Why can't we get away with simply moving owned handles here? Since List is a recursive type, each Cons cell is (implicitly) wrapped in an Rc. Hence, casing on the list requires accessing an Rc. In our setting, as in Rust, the content of an Rc can be accessed only by borrow. If the content is needed by ownership, we dup. There are other possible designs, but some dynamic work is certainly needed to convert an $\bullet$ Rc$\langle\bullet$ T$\rangle$ into an $\bullet$ T while keeping the dynamic reference count information coherent with the drops implied by the static ownership information.

In the remainder of this paper, we present a type inference algorithm that is capable of deriving the signature for nth described in this section from the unannotated program.

3 Inference Informally

3.1 Overview

We present a simplified formalism with boolean, product, and reference-counted handle types. This setting suffices for the core ideas of our inference algorithm. In Section 5, we discuss how the system can be extended with recursive types and recursive functions. A formalism and inference algorithm for the extended system are available in the appendix, and we carry out the proof of soundness for the extended system.

The input language for our transformation, L^{src}, is presented in Figure 1. L^{src} does not have ownership or borrowing. It is standard, save the **rc** type, which we use to formally model reference counted types such as Morphic's arrays. Expression **rc** e introduces an **rc**. Expression **get** e eliminates an **rc**. There are four primitive memory management operations that our inference system will add to programs in translating from L^{src} to our target language, $L^{\&}$, which extends L^{src} with ownership and borrowing:

- *Moving* transfers ownership of a handle and *borrowing* creates an alias of a handle. Dynamically, these operations are just ordinary variable occurrences with no extra overhead.
- *Duping* produces a new owned handle from an existing handle. Dynamically, this is a reference count increment.
- *Dropping* consumes an owned handle. Dynamically, this is a reference count decrement. If the reference count hits zero, the underlying allocation is freed. If the handle is an **rc** containing another **rc**, we must drop the inner **rc** before freeing the memory of the outer **rc**.

Our first problem is choosing a representation of borrows for $L^{\&}$. In Rust, "&" is a type former and values of borrowed types are pointers. In our setting, we are free to represent values of $\&t$ and t identically. Borrows are merely a memory management tool, and many of the types Rust distinguishes are not meaningfully different to us. For any t_1, t_2, the types $\&(t_1, t_2)$ and $(\&t_1, \&t_2)$ impose identical memory management requirements. Likewise, $\&$**int** and **int** and, for any t, $\&\&t$

$$\begin{array}{llll} \text{ResourceVar} & \mathfrak{r} & \text{Type } \mathfrak{t} ::= \textbf{bool} \mid \mathfrak{t}_1 \times \mathfrak{t}_2 \mid \mathfrak{r}\,\textbf{rc}\,\mathfrak{t} & \text{TypeCon } \Gamma ::= \diamond \mid \Gamma, x : \mathfrak{t} \\ \text{Expr} & \mathfrak{e} ::= & \textbf{true} \mid \textbf{false} \mid \textbf{if}\,\mathfrak{e}\,\textbf{then}\,\mathfrak{e}_1\,\textbf{else}\,\mathfrak{e}_2 \mid (\mathfrak{e}_1, \mathfrak{e}_2) \mid \textbf{projl}\,\mathfrak{e} \mid \textbf{projr}\,\mathfrak{e} & \\ & & \mid \textbf{rc}\,\mathfrak{e} \mid \textbf{get}\,\mathfrak{e}\,\textbf{as}\,\mathfrak{t} \mid \textbf{let}\,x : \mathfrak{t} = \mathfrak{e}_1\,;\,\mathfrak{e}_2 \mid x\,\textbf{as}\,\mathfrak{t} & \end{array}$$

Fig. 2. Simplified L^{inf} syntax.

and $\&t$ impose identical memory management requirements. This lack of canonicity is a challenge for the design and analysis of an inference algorithm. Therefore, instead of representing borrows via a type former, we introduce *modes* $m \in \{\bullet, \&\}$, and parameterize the **rc** type by whether it is owned, $\bullet\,\textbf{rc}\,t$, or borrowed with some lifetime L, $\&\langle L\rangle\ \textbf{rc}\,t$. Lifetimes are formalized in Section 4.1.

The central challenge for our inference algorithm is that we are told neither modes nor lifetimes. To see why this makes inference challenging, consider the following program—for brevity, we write **rc** instead of **rc int**:[2] $\textbf{let}\,x : \textbf{rc} = \textbf{rc}\,0\,;\,\textbf{let}\,y : \textbf{rc} = x\,;\,\cdots$

If the binding x is not used in "$\cdots$", which of the following $L^{\&}$ programs should we produce?

$$\begin{array}{ll} (1) & \textbf{let}\,x : \bullet\,\textbf{rc} = \textbf{rc}\,0\,;\,\textbf{let}\,y : \&\,\textbf{rc} = \&\,x\,;\,\cdots \\ (2) & \textbf{let}\,x : \bullet\,\textbf{rc} = \textbf{rc}\,0\,;\,\textbf{let}\,y : \bullet\,\textbf{rc} = x\,;\,\cdots \\ (3) & \textbf{let}\,x : \bullet\,\textbf{rc} = \textbf{rc}\,0\,;\,\textbf{let}\,y : \bullet\,\textbf{rc} = \textbf{dup}(x)\,;\,\cdots \end{array}$$

Notice: (1) requires that x be live as long as y is live; (2) requires that x be live when y is created and moves x, ending its lifetime; (3) only requires that x be live when y is created, but incurs a runtime cost. Modes shape the relationships between lifetimes, and lifetimes constrain which modes are possible. Naively, one could search the exponentially large space of possible mode assignments until the first assignment is found where the implied lifetimes are consistent, inserting a dup and restarting the process if no such assignment is found. More practically, one could assume that every mode is owned, which can always be made sound by inserting a dup at all but the final syntactic occurrence of each binding along each control flow path. The resulting algorithm amounts to the Perceus RC technique developed by Reinking et al. [22]. Our technique resolves the mutual recursion between modes and lifetimes by bootstrapping to a fixed point solution: we first infer conservative lifetimes that do not require modes, use those lifetimes to infer modes, and then refine the lifetimes to be precise given the mode assignment. In more detail:

1. First, we lift our L^{src} program to an intermediate language, L^{inf} (Figure 2), by annotating it with fresh *resource variables*, which identify the ownership and lifetime information tied to **rc** values during analysis. In L^{inf}, variable occurrences and **get** operations are annotated with $\&/\bullet$ *casts*. For example, in L^{inf}, if $x : \bullet\,\textbf{rc}$, we can write $x\,\textbf{as}\,\bullet\,\textbf{rc}$. Later, when we produce an $L^{\&}$ program, this might become a **dup** or a move. We delay the decision until we have precise lifetimes. Beyond leaving open whether a **dup** is required, casts are convenient because they have a uniform shape regardless of whether we convert borrowed to borrowed, a no-op in $L^{\&}$, owned to owned, possibly a **dup** in $L^{\&}$, etc.
2. Next, we compute *approximate lifetimes* for bindings and (abstract) heap locations under the maximally pessimistic assumption that, if an **rc**, y, is transitively derived from an **rc**, x, accessing y might require that x is live, though, in fact, this only happens when y borrows x. Comparing approximate lifetimes to lexical scopes amounts to an escape analysis that allows us to derive a system of equations between modes; if a binding x is owned and an occurrence of x escapes the lexical scope of x, the occurrence must be a **dup** or move of x. With our choice of modes fixed, we can, opportunistically, move rather than **dup** at the final

[2] Technically **int** is not an L^{src} type but, like **bool**, it is trivial from a borrowing perspective, so we use it freely in examples.

occurrences of bindings. Finally, we compute *precise lifetimes*, annotations appropriate to $L^{\&}$ that account for which values are borrowed and which are owned.

(3) Given precise lifetimes, we perform *reification* to produce an $L^{\&}$ program. We substitute for the resource variables in our L^{inf} program with the calculated modes and precise lifetimes and lower casts to borrow operations, **dup** operations, or moves. When lowering casts, **dup** operations are inserted at owned occurrences of bindings that are not final occurrences, i.e., are not at the end of the relevant precise lifetime. Final owned occurrences become moves. Finally, reification inserts **drop** operations at the end of the scope of each binding for each owned, top-level **rc** that was not moved, upholding linearity in the lowered program.

We dedicate the remainder of this section to intuition building, deferring a formal presentation of inference to Section 4.4 where Figure 7 details step (1) and Figure 8 details step (2).

3.2 Borrows of Stack Values

We impose two kinds of mode constraints: (1) if an occurrence, x, escapes the scope of its binding and its binding is owned, then the occurrence must be owned; (2) if an occurrence is owned, then its binding must be owned. We will explore why these constraints are necessary through a few examples. Consider the following program in a modestly extended L^{inf} where $\mathfrak{r}_{x_1}$, $\mathfrak{r}_{y_1}$, etc. are resources variables:

$$\mathbf{let}\, x : \mathfrak{r}_{x_1}\ \mathbf{rc} = (\mathbf{let}\, y : \mathfrak{r}_{y_1}\ \mathbf{rc} = (\mathbf{rc}\, 0)\, ;\ (y\ \mathbf{as}\, \mathfrak{r}_{y_2}\ \mathbf{rc}))\, ;\ \mathsf{print}(x\ \mathbf{as}\, \mathfrak{r}_{x_2}\ \mathbf{rc})$$

$\mathfrak{r}_{x_2}$ can be & as print simply reads its argument. Since $(\mathbf{rc}\, 0)$ creates a new **rc** that must be given an owner, $\mathfrak{r}_{y_1}$ must reify as $\bullet$. Since the occurrence of y escapes the lexical scope of y and $\mathfrak{r}_{y_1} = \bullet$, $\mathfrak{r}_{y_2}$ must be $\bullet$, otherwise the print would be a use-after-free. Therefore, $\mathfrak{r}_{x_1}$ must also be $\bullet$ and we are done. Reification yields the program

$$\mathbf{let}\, x : \bullet\ \mathbf{rc} = (\mathbf{let}\, y : \bullet\ \mathbf{rc} = (\mathbf{rc}\, 0)\, ;\ y)\, ;\ \mathsf{print}(\&x)$$

Compare this example to the following, similar situation:

$$\begin{aligned}&\mathbf{let}\, r : \mathfrak{r}_{r_1}\ \mathbf{rc} = (\mathbf{rc}\, 0)\, ;\\ &\mathbf{let}\, x : \mathfrak{r}_{x_1}\ \mathbf{rc} = (\mathbf{let}\, y : \mathfrak{r}_{y_1}\ \mathbf{rc} = (r\ \mathbf{as}\, \mathfrak{r}_{r_2}\ \mathbf{rc})\, ;\ (y\ \mathbf{as}\, \mathfrak{r}_{y_2}\ \mathbf{rc}))\, ;\ \mathsf{print}(x\ \mathbf{as}\, \mathfrak{r}_{x_2}\ \mathbf{rc})\end{aligned}$$

Again, we obtain that $\mathfrak{r}_{x_2}$ can be & and $\mathfrak{r}_{r_1}$ must be $\bullet$. We are left with $\mathfrak{r}_{r_2}$, $\mathfrak{r}_{y_1}$, $\mathfrak{r}_{y_2}$, and $\mathfrak{r}_{x_1}$. By flow analysis, the occurrence of r (at $\mathfrak{r}_{r_2}$) cannot escape the lexical scope of r (at $\mathfrak{r}_{r_1}$), regardless of what we reify the modes in the program to. So, $\mathfrak{r}_{r_2}$ can be &. We bind y to this value, so $\mathfrak{r}_{y_1} = \&$.

The occurrence of y (at $\mathfrak{r}_{y_2}$) escapes the scope of its binding (at $\mathfrak{r}_{y_1}$), but its binding is now a borrow of r, whose scope it does not escape. This means we can let $\mathfrak{r}_{y_2}$ be &. In fact, if the occurrence of y escaped the scope of its lender's binding, we would not have marked its binding as a borrow in the first place. Hence, we need only consider that the binding of y (at $\mathfrak{r}_{y_1}$) is a borrow, not that it is a borrow of r in particular, to know it is safe for the occurrence of y (at $\mathfrak{r}_{y_2}$) to be a borrow. We let $\mathfrak{r}_{y_2}$, and hence $\mathfrak{r}_{x_1}$, be & and we are done. Reification yields the program:

$$\mathbf{let}\, r : \bullet\ \mathbf{rc} = (\mathbf{rc}\, 0)\, ;\ \mathbf{let}\, x : \&\ \mathbf{rc} = (\mathbf{let}\, y : \&\ \mathbf{rc} = \&r\, ;\ y)\, ;\ \mathsf{print}(x)$$

In general, we perform a mode-oblivious flow analysis, which calculates a conservative lifetime, the longest interval where the data backing the occurrence might need to live. If an occurrence escapes the scope of its binding according to the flow analysis, and the binding is owned, then the occurrence must be owned. Otherwise, it can be borrowed, unless it unifies with something that must be owned, e.g., because it is the **then** branch of an **if** and the **else** branch must be owned.

So far, we have simply assigned the mode of a variable binding the mode of its initializer. We have assumed that it is always desirable to borrow where it is sound to do so. In fact, borrowing

can result in additional dups. Consider the following program:

$$\textbf{let}\, x = (\textbf{rc}\, 0)\, \textbf{;}\ \textbf{let}\, z = \textbf{if}\, b\, \textbf{then}\, (\textbf{rc}\, 1)\, \textbf{else}\, (\textbf{let}\, y = x\, \textbf{;}\ y)\, \textbf{;}\ \cdots$$

After inference, the **then** expression, x, and z are owned. By unification, the occurrence of y in the **else** expression must also be owned. There are two $L^{\&}$ programs satisfying these constraints:

$$\textbf{let}\, y : \&\, \textbf{rc} = \&x\, \textbf{;}\ \textbf{dup}(y) \qquad\qquad \textbf{let}\, y : \bullet\, \textbf{rc} = x\, \textbf{;}\ y$$

The former has more borrowed values than the latter, but an additional **dup**. In general, constraining a binding to be owned if any of its occurrences are owned can enable those occurrences to be moves, saving a **dup**. It is common for such patterns to appear in user code because operations on built-in collection types that would be mutating in the imperative setting, e.g., push for arrays, are naturally modeled as accepting owned arguments. As discussed in Section 6.7, this often allows the runtime to perform these logically immutable operations mutably. We will, therefore, produce the second program above, where x is moved into y, by rule (2) from the beginning of this section.

3.3 Borrows in Heap Values

In $L^{\&}$, the operation of borrowing can only be performed at the top level of types, i.e., while we can reinterpret an $\bullet\,\textbf{rc}\,\textbf{int}$ as a $\&\,\textbf{rc}\,\textbf{int}$, we cannot reinterpret an $\bullet\,\textbf{rc}\,(\bullet\,\textbf{rc}\,\textbf{int})$ as an $\bullet\,\textbf{rc}\,(\&\,\textbf{rc}\,\textbf{int})$. Allowing this would break the dynamics of drop as, if the drop of the $\bullet\,\textbf{rc}\,(\&\,\textbf{rc}\,\textbf{int})$ decrements the reference count of the outer **rc** to 0, we will not also decrement the count of the inner **rc**, resulting in a memory leak. We conclude that borrow inference should impose equality constraints between binding and occurrence modes for **rc** types nested inside other **rc** types. Consider the following:

$$\textbf{let}\, x = \textbf{rc}\, (\textbf{rc}\, 0)\, \textbf{;}\ \textbf{let}\, y = x\, \textbf{;} \cdots$$

When we translate into $L^{\&}$, we will infer $x : \bullet\,\textbf{rc}\,(\bullet\,\textbf{rc}\,\textbf{int})$ and either $y : \bullet\,\textbf{rc}\,(\bullet\,\textbf{rc}\,\textbf{int})$ or $y : \&\,\textbf{rc}\,(\bullet\,\textbf{rc}\,\textbf{int})$, but never $y : \&\,\textbf{rc}\,(\&\,\textbf{rc}\,\textbf{int})$. Per the above discussion, we could not obtain an $\&\,\textbf{rc}\,(\&\,\textbf{rc}\,\textbf{int})$ simply via applying "&" or **dup** to x.

3.4 Borrows of Heap Values

RC'd types hold heap data and have some method to access that data by borrow, e.g., **get** for **rc** or indexing for an array. In $L^{\&}$, if we have a borrowed **rc** of borrowed data, e.g. an $\&\,\textbf{rc}\,(\&\,\textbf{rc}\,\textbf{int})$, the lifetime of the output of **get** is the lifetime of the inner borrow. If we have a borrowed **rc** of owned data, e.g. an $\&\,\textbf{rc}\,(\bullet\,\textbf{rc}\,\textbf{int})$, the lifetime of the output of **get** is the lifetime of the outer borrow.

This phenomenon motivates reasoning separately about the modes and lifetimes of the values obtained from a **get** and the modes and lifetimes of the outer **rc** on which you are performing the **get**. Additionally, different top-level **rc** values in the item obtained from a **get** might be subject to different constraints that force some to be **dup**'ed while others can remain borrows. For these reasons, we introduce a fine-grained mechanism for separately tracking how each inner **rc** should participate in **get** operations. At a mechanical level, we treat resource variables on nested **rc** types as if they were top-level for the purpose of borrowing at a **get**. For each nested **rc**, borrow inference tracks two modes, a *storage* mode to determine where it is safe to store borrows to that (abstract) location, and an *access* mode to determine where it is safe to borrow objects from that (abstract) location. The former becomes the type's $L^{\&}$ mode during reification and is equality constrained in accordance with the discussion in Section 3.3. The latter becomes the mode of the output of a **get** on the type and is determined according to the discussion in Section 3.2. Consider the following L^{inf} program where annotation $(\mathfrak{r}_1, \mathfrak{r}_2)$ indicates that $\mathfrak{r}_1$ is the storage mode and $\mathfrak{r}_2$ is the access

mode of the **rc** type:

$$\mathbf{let}\, x : \mathfrak{r}_{x_2}\, \mathbf{rc\, int} = (\mathbf{let}\, y : \mathfrak{r}_{y_1}\, \mathbf{rc}\, ((\mathfrak{r}_{y_2}, \mathfrak{r}_{y_3})\, \mathbf{rc\, int}) = \mathbf{rc}\,(\mathbf{rc}\, 0)\, ;$$
$$\mathbf{get}(y\, \mathbf{as}\, \mathfrak{r}_{y_4}\, \mathbf{rc}\, ((\mathfrak{r}_{y_5}, \mathfrak{r}_{y_6})\, \mathbf{rc\, int}))\, \mathbf{as}\, \mathfrak{r}_{x_1}\, \mathbf{rc\, int})\, ; \cdots$$

Immediately, we have $\mathfrak{r}_{y_1} = \mathfrak{r}_{y_2} = \mathfrak{r}_{y_3} = \bullet$ since we are creating a new **rc** (**rc** 0) value and every value needs an owner. Since storage modes are equality constrained, we also know that $\mathfrak{r}_{y_2} = \mathfrak{r}_{y_5} = \bullet$. Since the output of the **get** escapes the scope of y and $\mathfrak{r}_{y_3} = \bullet$, we know $\mathfrak{r}_{y_6} = \bullet$ and, hence, $\mathfrak{r}_{x_2} = \mathfrak{r}_{x_1} = \bullet$. Since **get** in $L^{\&}$ takes its argument by borrow, $\mathfrak{r}_{y_4}$ is &. Reifying:

$$\mathbf{let}\, x : \bullet\, \mathbf{rc\, int} = (\mathbf{let}\, y : \bullet\, \mathbf{rc}\, (\bullet\, \mathbf{rc\, int}) = \mathbf{rc}\,(\mathbf{rc}\, 0)\, ;\, \mathbf{dup}(\mathrm{get}(\&y)))\, ; \cdots$$

We dup the output of the **get** because **get** in $L^{\&}$ always returns by borrow, but the access mode of the inner **rc** in L^{inf} tells us via the **as** on **get** that it is not safe to borrow from the inner **rc** there. Consider a similar situation where it is safe to borrow—this time we simply fill in the result of mode solving and skip discussion of the underlying constraints:

$$\mathbf{let}\, z : \bullet\, \mathbf{rc}\, ((\bullet, \bullet)\mathbf{rc\, int}) = \mathbf{rc}\,(\mathbf{rc}\, 0)\, ;$$
$$\mathbf{let}\, x : \&\, \mathbf{rc\, int} = (\mathbf{let}\, y : \&\, \mathbf{rc}\, ((\bullet, \&)\, \mathbf{rc\, int}) =$$
$$(z\, \mathbf{as}\, \&\, \mathbf{rc}\, ((\bullet, \&)\, \mathbf{rc\, int}))\, ;\, \mathrm{get}(y\, \mathbf{as}\, \&\, \mathbf{rc}\, ((\bullet, \&)\, \mathbf{rc\, int})))\, ;\, \mathrm{print}(x\, \mathbf{as}\, \&\, \mathbf{rc\, int})$$

While the output of the **get** escapes the scope of y, it does not escape the scope of y's lender. Reifying:

$$\mathbf{let}\, z : \bullet\, \mathbf{rc}\, (\bullet\, \mathbf{rc\, int}) = \mathbf{rc}\,(\mathbf{rc}\, 0)\, ;$$
$$\mathbf{let}\, x : \&\, \mathbf{rc\, int} = (\mathbf{let}\, y : \&\, \mathbf{rc}\, (\bullet\, \mathbf{rc\, int}) = \&z\, ;\, \mathrm{get}(y))\, ;\, \mathrm{print}(x)$$

Finally, we note that, while it may appear storage and access modes can be calculated independently, they are actually mutually dependent through their interaction with the modes of stack variables at **rc** and **get**.

4 Inference Formally

In this section, we precisely formulate lifetimes, $L^{\&}$, and our inference algorithm. Inference largely follows from the observations made in the previous section. In Section 5, we discuss how $L^{\&}$ and inference can be extended to handle recursive types and recursive functions.

4.1 Lifetimes

A lifetime represents the program interval where a value can be safely accessed. Its information content is, for each control flow path, the program point where safety is lost.[3] During inference, we will start from the assumption that values do not have any liveness requirements, updating our assumptions as we traverse the program and encounter expressions requiring liveness, such as dereferences. We therefore ought to choose a representation of lifetimes that admits a simple update function, computing the least upper bound of those requirements we have already discovered and those we newly discover. To that end, unlike prior work (cf. [16, 29]), we give lifetimes a structural interpretation, defining them as trees that "sit atop" the program, picking out the variable occurrences (subexpressions) where safe access to a value is lost. The contribution of this paper is not the notion of a lifetime; it is a formulation of lifetimes suitable for inference.

In our setting, there are two possible control flow relationships between subexpressions within a function. They are sequentially ordered, as with the bound expression and body of a **let**, or they

[3] We are not concerned with where lifetimes begin. This information is implicit in the operation of the typing context.

are unordered, occurring in disjoint program executions, as with the **then** and **else** arms of an **if**. A first attempt to define lifetimes yields:

$$\ell ::= \star \mid \swarrow \ell \mid \searrow \ell \mid (\ell_1 \parallel \ell_2)$$[4]

Here, $\swarrow \ell$ (before) and $\searrow \ell$ (after) are *sequential composition* constructors, e.g., letting one specify either that a lifetime ends in the bound expression of a **let** or the body. $(\ell_1 \parallel \ell_2)$ is an *alternating composition* constructor, e.g., letting one specify where a lifetime ends in both arms of an **if**. $\star$ is the *singleton* lifetime, representing an occurrence of the variable of interest and acting as a base case. The set of program points where safe access to a value is lost then corresponds to the set of paths from root to singleton. Consider the following examples:

$$\begin{array}{ll} \textbf{let}\, x : \bullet\, \textbf{rc} = (\textbf{rc}\, 0)\, ; & \textbf{let}\, x : \bullet\, \textbf{rc} = (\textbf{rc}\, 0)\, ; \\ \textbf{let}\, y : \bullet\, \textbf{rc} = \textbf{if}\, b\, \textbf{then}\, x\, \textbf{else}\, x\, ; \cdots & \textbf{let}\, y : \bullet\, \textbf{rc} = \textbf{if}\, b\, \textbf{then}\, x\, \textbf{else}\, (\textbf{rc}\, 1)\, ; \cdots \end{array}$$

On the left, the lifetime of x is $\searrow(\swarrow(\searrow(\star \parallel \star)))$, the final syntactic occurrences of x in each branch, where it is moved. The first $\searrow$ indicates the body of the first **let**, the $\swarrow$ indicates the bound expression of the second **let**, and the second $\searrow$ indicates the branches of the **if**, which are ordered after the **if** condition.

Now consider the lifetime of x on the right. It is only used in one branch of the **if**. To address this possibility, we add two additional alternating lifetime constructors, $(\ell \parallel -)$ and $(- \parallel \ell)$. The lifetime of x is then $\searrow(\swarrow(\searrow(\star \parallel -)))$. Note that we will always measure lifetimes relative to the beginning of the program or, in general, the enclosing function. Measuring lifetimes relative to a common program point will allow us to express the lifetime constraints for, e.g., **get** operations and moves in the $L^{\&}$ statics.

With lifetimes suitably defined, a least upper bound operation can be obtained by taking the later program point along each branch, e.g.,

$$(\swarrow \ell_1) \sqcup (\searrow \ell_2) = \searrow \ell_2 \qquad\qquad (\ell_1 \parallel \ell_2) \sqcup (\ell_1' \parallel \ell_2') = (\ell_1 \sqcup \ell_1' \parallel \ell_2 \sqcup \ell_2')$$

$\sqcup$ is well-defined so long as its operands are *compatible*, defined inductively by the fact that a singleton is compatible with anything, sequential constructors are compatible with each other, and likewise for alternating constructors. Given a function f, the set of *relevant* lifetimes that "match" f's control flow are always compatible (Appendix D formalizes this, but there are no surprises).

Finally, to represent unused variables, we augment our definition with an *empty* lifetime $\varnothing$. This will be the initial value we assign to lifetimes during inference. Adding $\varnothing$ naively, however, results in non-canonical lifetimes, e.g., $\swarrow \varnothing \simeq \varnothing$. Thus, we stratify our definition into *lifetimes* L and *local lifetimes* ℓ where

$$L ::= \varnothing \mid \ell \qquad\qquad \ell ::= \star \mid \swarrow \ell \mid \searrow \ell \mid (\ell \parallel -) \mid (- \parallel \ell) \mid (\ell_1 \parallel \ell_2)$$

On each set of compatible lifetimes S, $\sqcup$ gives us an algebraic semilattice $(S, \sqcup, \varnothing)$. It is natural to consider the corresponding order-theoretic semilattice $(S, \leq, \varnothing)$, defined by the reflexive closure of

$$\frac{}{\ell \leq \star} \qquad \frac{\ell \leq \ell'}{\swarrow \ell \leq \swarrow \ell'} \qquad \frac{\ell \leq \ell'}{\searrow \ell \leq \searrow \ell'} \qquad \frac{}{\swarrow \ell \leq \searrow \ell'} \qquad \frac{\ell_1 \leq \ell_1' \quad \ell_2 \leq \ell_2'}{(\ell_1 \parallel \ell_2) \leq (\ell_1' \parallel \ell_2')} \qquad \frac{\ell_1 \leq \ell_1'}{(\ell_1 \parallel -) \leq (\ell_1' \parallel \ell_2')}$$

$$\frac{\ell_2 \leq \ell_2'}{(- \parallel \ell_2) \leq (\ell_1' \parallel \ell_2')} \qquad \frac{\ell \leq \ell'}{(\ell \parallel -) \leq (\ell' \parallel -)} \qquad \frac{\ell \leq \ell'}{(- \parallel \ell) \leq (- \parallel \ell')} \qquad \frac{}{\varnothing \leq \ell}$$

Informally, thinking of lifetimes as trees, $L \leq L'$ says: for each branch of L there exists a branch of L' that covers/contains it. One can check by induction that $L \leq L'$ iff $L' = L \sqcup L'$. In a borrows-with-lifetimes type system, it is critical that a long lifetime can be used in place of a shorter lifetime.

[4] This definition is not quite sufficient, as we shall see.

ValueResource	V	$::=$	$\bullet \mid \&\langle L\rangle$ BindingResource B $::=$ $\bullet\langle L\rangle \mid \&\langle L\rangle$
ValueType	τ^v	$::=$	$\mathbf{bool} \mid \tau^v_1 \times \tau^v_2 \mid V\,\mathbf{rc}\,\tau^v$ BindingType τ^b $::=$ $\mathbf{bool} \mid \tau^b_1 \times \tau^b_2 \mid B\,\mathbf{rc}\,\tau^v$
Usage	u	$::=$	$\mathbf{bool} \mid u_1 \times u_2 \mid \mathbf{keep} \mid \mathbf{move} \mid \mathbf{irrel}$ Selector s $::=$ $\mathbf{bool} \mid s_1 \times s_2 \mid \mathbf{select} \mid \mathbf{ignore}$
TypeCon	Γ	$::=$	$\diamond \mid \Gamma, x : \tau^b$ UsageCon Υ $::=$ $\diamond \mid \Upsilon, x : u$
Expr	ϵ	$::=$	$\mathbf{true} \mid \mathbf{false} \mid \mathbf{if}\,\epsilon\,\mathbf{then}\,\epsilon_1\,\mathbf{else}\,\epsilon_2 \mid (\epsilon_1, \epsilon_2) \mid \mathbf{projl}\,\epsilon \mid \mathbf{projr}\,\epsilon$
			$\mid \mathbf{rc}\,\epsilon \mid \mathbf{get}\,\epsilon \mid \mathbf{dup}\,s\,\epsilon \mid \epsilon\,;\,\mathbf{drop}\,\overrightarrow{s\,x} \mid \mathbf{let}\,x : \tau^b = \epsilon_1\,;\,\epsilon_2 \mid \mathbf{use}\,u\,x$

Fig. 3. Simplified $L^{\&}$ syntax.

In the formalism of $L^{\&}$, we shall achieve this by a subtyping rule declaring types contravariant in their lifetimes ordered under $\leq$ (BT-Sub in Figure 4).

In the statics of $L^{\&}$, we shall need to assert that the lifetime of a variable does not extend past the point where it is moved. Let p denote a *path*, that is, a lifetime containing no instances of the $(\ell_1 \parallel \ell_2)$ constructor. If a variable with lifetime L is moved at p, it must hold that no branch of L contains p, i.e., $p \not\leq L$. Orderings on trees are subtle; notice that $p \not\leq L$ is not equivalent to $L \leq p$. Also notice that $p \not\leq L$ behaves intuitively as a bound on L only when the left-hand side is a mere path. In general, $L' \not\leq L$ says: there exists a branch of L' that is not contained in any branch of L. For this reason, we henceforth use dedicated notation $L \prec p$ in place of $p \not\leq L$.

Finally, we define $L \asymp p$ to be the operation that checks whether p is on the boundary of L, that is, coincides exactly with some path in L, and we define $p_1 \cdot p_2$ to be the concatenation of p_1 and p_2, that is, p_1 where the $\star$ has been replaced with p_2.

4.2 Statics of $L^{\&}$

$L^{\&}$ (Figure 3) is similar to L^{src} (Figure 1), but has an explicit **drop** expression, **dup** expression, and mode and lifetime annotations. Two facets of $L^{\&}$ stand out immediately: the data u in the variable occurrence expression $\mathbf{use}\,u\,x$ and the two varieties of types, τ^b and τ^v. As to the first, we observe that if we simply had a borrowing construct $\&\epsilon$, the borrow would happen only after we had evaluated ϵ and moved the relevant binding, rendering the exercise pointless. We, therefore, fuse borrowing with variable occurrence. In $\mathbf{use}\,u\,x$, *usage flag* u specifies **keep** or **move** for each top-level, owned **rc** in x to determine whether to borrow or move that **rc**, and **irrel** for each top-level, borrowed **rc** since borrows are never consumed at usages. The same problem we solve with usage flags Rust solves via C-inspired lvalue/place expressions. In Rust, e.g., $\&x.y$ cannot be understood as $x.y$ followed by $\&$, since the former interpretation would move field y of x, whereas the composite expression borrows field y without moving it. $\&x.y$ is an atomic expression in a DSL for talking about "locations" and how to operate on them. Beyond ease of formalization, a benefit of our solution is that it allows a one-to-one translation of L^{src} occurrences to $L^{\&}$ occurrences, even if we borrow only some fields of a type. For example, L^{src} expression "$\mathbf{let}\,x = (\mathbf{rc}\,0, \mathbf{rc}\,1)\,;\,x$" might become $L^{\&}$ expression "$\mathbf{let}\,x = (\mathbf{rc}\,0, \mathbf{rc}\,1)\,;\,(\mathbf{use}\,(\mathbf{keep} \times \mathbf{move})\,x)$", which has type $(\&\,\mathbf{rc}\,\mathbf{int}) \times (\bullet\,\mathbf{rc}\,\mathbf{int})$, borrows the first field of x, and moves the second. Similarly, **dup** expressions are parameterized by a *selector* s, which indicates, for each top-level **rc** whether to dup (indicated by **select**) or move (indicated by **ignore**). Finally, note that a single **drop** expression can drop multiple bindings. This makes $L^{\&}$ an easier target for inference, since inference computes lifetimes before determining what to drop, and so must anticipate the shape of the final program.

The distinction between *value types* τ^v and *binding types* τ^b stems from an observation about the central role of top-level owned bindings in borrow-with-lifetimes type systems. τ^v ascribes a lifetime to borrows, but not owners. τ^b additionally ascribes a lifetime to each top-level owner. The type of the content of a binding **rc** type is a value type. From a type checking perspective, lifetimes

$$\text{BT-Sub}\quad \frac{\Gamma ; p \vdash \epsilon : \tau_1^v ; \Upsilon \qquad \tau_1^v <: \tau_2^v}{\Gamma ; p \vdash \epsilon : \tau_2^v ; \Upsilon} \qquad \text{BT-True}\quad \frac{}{\Gamma ; p \vdash \mathbf{true} : \mathbf{bool} ; \diamond} \qquad \text{BT-False}\quad \frac{}{\Gamma ; p \vdash \mathbf{false} : \mathbf{bool} ; \diamond}$$

$$\text{BT-If}\quad \frac{\Gamma ; \mathit{seql}(p) \vdash \epsilon : \mathbf{bool} ; \Upsilon_1 \qquad \Gamma ; \mathit{altl}(\mathit{seqr}(p)) \vdash \epsilon_1 : \tau^v ; \Upsilon_2 \qquad \Gamma ; \mathit{altr}(\mathit{seqr}(p)) \vdash \epsilon_2 : \tau^v ; \Upsilon_2 \qquad \Upsilon \leftsquigarrow \mathit{merge}(\Upsilon_1, \Upsilon_2)}{\Gamma ; p \vdash \mathbf{if}\, \epsilon \,\mathbf{then}\, \epsilon_1 \,\mathbf{else}\, \epsilon_2 : \tau^v ; \Upsilon}$$

$$\text{BT-Pair}\quad \frac{\Gamma ; \mathit{seql}(p) \vdash \epsilon_1 : \tau_1^v ; \Upsilon_1 \qquad \Gamma ; \mathit{seqr}(p) \vdash \epsilon_2 : \tau_2^v ; \Upsilon_2 \qquad \Upsilon \leftsquigarrow \mathit{merge}(\Upsilon_1, \Upsilon_2)}{\Gamma ; p \vdash (\epsilon_1, \epsilon_2) : \tau_1^v \times \tau_2^v ; \Upsilon}$$

$$\text{BT-ProjL}\quad \frac{\Gamma ; p \vdash \epsilon : \tau_1^v \times \tau_2^v ; \Upsilon}{\Gamma ; p \vdash \mathbf{projl}\, \epsilon : \tau_1^v ; \Upsilon} \qquad \text{BT-ProjR}\quad \frac{\Gamma ; p \vdash \epsilon : \tau_1^v \times \tau_2^v ; \Upsilon}{\Gamma ; p \vdash \mathbf{projr}\, \epsilon : \tau_2^v ; \Upsilon} \qquad \text{BT-Rc}\quad \frac{\Gamma ; p \vdash \epsilon : \tau^v ; \Upsilon}{\Gamma ; p \vdash \mathbf{rc}\, \epsilon : \bullet\, \mathbf{rc}\, \tau^v ; \Upsilon}$$

$$\text{BT-Get}\quad \frac{\mathit{seqr}(p) \le L \qquad \Gamma ; \mathit{seql}(p) \vdash \epsilon : \&\langle L\rangle\, \mathbf{rc}\, \tau^v ; \Upsilon}{\Gamma ; p \vdash \mathbf{get}\, \epsilon : \&_L(\tau^v) ; \Upsilon}$$

$$\text{BT-Dup}\quad \frac{\Gamma ; \mathit{seql}(p) \vdash \epsilon : \tau_1^v ; \Upsilon \qquad \tau_2^v \leftsquigarrow \mathit{dup\text{-}ty}(\mathit{seqr}(p), s, \tau_1^v)}{\Gamma ; p \vdash \mathbf{dup}\, s\, \epsilon : \tau_2^v ; \Upsilon}$$

$$\text{BT-Drop}\quad \frac{\overrightarrow{x} \text{ distinct} \qquad \Gamma ; \mathit{seql}(p) \vdash \epsilon : \tau^v ; \Upsilon_1 \qquad \overrightarrow{u \leftsquigarrow \mathit{drop\text{-}rsrc}(\mathit{seqr}(p), s, \Gamma(x))} \qquad \Upsilon_2 \leftsquigarrow \mathit{merge}(\Upsilon_1, \{\overrightarrow{x : u}\})}{\Gamma ; p \vdash (\epsilon \,;\, \mathbf{drop}\, \overrightarrow{s\, x}) : \tau^v ; \Upsilon_2}$$

$$\text{BT-Let}\quad \frac{x \notin \Gamma, \Upsilon_1 \qquad \Gamma ; \mathit{seql}(p) \vdash \epsilon_1 : \mathit{val\text{-}ty}(\tau^b) ; \Upsilon_1 \qquad \Gamma, x : \tau^b ; \mathit{seqr}(p) \vdash \epsilon_2 : \tau^v ; \Upsilon_2, x : \mathit{move}(\tau^b) \qquad \Upsilon \leftsquigarrow \mathit{merge}(\Upsilon_1, \Upsilon_2)}{\Gamma ; p \vdash (\mathbf{let}\, x : \tau^b = \epsilon_1 \,;\, \epsilon_2) : \tau^v ; \Upsilon}$$

$$\text{BT-Use}\quad \frac{\tau^v \leftsquigarrow \mathit{use\text{-}ty}(p, u, \Gamma(x))}{\Gamma ; p \vdash \mathbf{use}\, u\, x : \tau^v ; \{x : u\}}$$

$\mathit{seql}, \mathit{seqr}, \mathit{altl}, \mathit{altr} : \text{Path} \to \text{Path}$

$\mathit{seql}(p) = p \cdot \swarrow \star$

$\mathit{seqr}(p) = p \cdot \searrow \star$

$\mathit{altl}(p) = p \cdot (\star \parallel -)$

$\mathit{altr}(p) = p \cdot (- \parallel \star)$

$\mathit{merge} : \text{Usage} \times \text{Usage} \rightharpoonup \text{Usage}$

$\mathit{merge}(\mathbf{bool}, \mathbf{bool}) \rightsquigarrow \mathbf{bool}$

$\mathit{merge}(u_1 \times u_2, u_1' \times u_2') \rightsquigarrow \mathit{merge}(u_1, u_1') \times \mathit{merge}(u_2, u_2')$

$\mathit{merge}(\mathbf{keep}, \mathbf{keep}) \rightsquigarrow \mathbf{keep}$

$\mathit{merge}(\mathbf{keep}, \mathbf{move}) \rightsquigarrow \mathbf{move}$

$\mathit{merge}(\mathbf{move}, \mathbf{keep}) \rightsquigarrow \mathbf{move}$

$\mathit{merge}(\mathbf{irrel}, \mathbf{irrel}) \rightsquigarrow \mathbf{irrel}$

$\mathit{merge} : \text{UsageCon} \times \text{UsageCon} \rightharpoonup \text{UsageCon}$

$$\begin{aligned}\mathit{merge}(\Upsilon_1, \Upsilon_2) \rightsquigarrow\ & \{x : \mathit{merge}(\Upsilon_1(x), \Upsilon_2(x)) \mid x \in \text{dom}(\Upsilon_1) \cap \text{dom}(\Upsilon_2)\} \\ & \cup \{x : \Upsilon_1(x) \mid x \in \text{dom}(\Upsilon_1) \setminus \text{dom}(\Upsilon_2)\} \\ & \cup \{x : \Upsilon_2(x) \mid x \in \text{dom}(\Upsilon_2) \setminus \text{dom}(\Upsilon_1)\}\end{aligned}$$

$\mathit{dup\text{-}ty} : \text{Path} \times \text{Selector} \times \text{ValueType} \rightharpoonup \text{ValueType}$

$\mathit{dup\text{-}ty}(p, \mathbf{bool}, \mathbf{bool}) \rightsquigarrow \mathbf{bool}$

$\mathit{dup\text{-}ty}(p, s_1 \times s_2, \tau_1^v \times \tau_2^v) \rightsquigarrow \mathit{dup\text{-}ty}(p, s_1, \tau_1^v) \times \mathit{dup\text{-}ty}(p, s_2, \tau_2^v)$

$\mathit{dup\text{-}ty}(p, \mathbf{select}, \&\langle L\rangle\, \mathbf{rc}\, \tau^v) \rightsquigarrow \bullet\, \mathbf{rc}\, \tau^v \quad \text{if } p \le L$

$\mathit{dup\text{-}ty}(p, \mathbf{ignore}, \&\langle L\rangle\, \mathbf{rc}\, \tau^v) \rightsquigarrow \&\langle L\rangle\, \mathbf{rc}\, \tau^v$

$\mathit{dup\text{-}ty}(p, s, \bullet\, \mathbf{rc}\, \tau^v) \rightsquigarrow \bullet\, \mathbf{rc}\, \tau^v$

$\mathit{drop\text{-}rsrc} : \text{Path} \times \text{Selector} \times \text{BindingType} \rightharpoonup \text{Usage}$

$\mathit{drop\text{-}rsrc}(p, \mathbf{bool}, \mathbf{bool}) \rightsquigarrow \mathbf{bool}$

$\mathit{drop\text{-}rsrc}(p, s_1 \times s_2, \tau_1^b \times \tau_2^b) \rightsquigarrow \mathit{drop\text{-}rsrc}(p, s_1, \tau_1^b) \times \mathit{drop\text{-}rsrc}(p, s_2, \tau_2^b)$

$\mathit{drop\text{-}rsrc}(p, s, \&\langle L\rangle\, \mathbf{rc}\, \tau^v) \rightsquigarrow \mathbf{keep}$

$\mathit{drop\text{-}rsrc}(p, \mathbf{select}, \bullet\langle L\rangle\, \mathbf{rc}\, \tau^v) \rightsquigarrow \mathbf{move} \quad \text{if } L \prec p \text{ or } L \asymp p$

$\mathit{drop\text{-}rsrc}(p, \mathbf{ignore}, \bullet\langle L\rangle\, \mathbf{rc}\, \tau^v) \rightsquigarrow \mathbf{keep}$

$\mathit{use\text{-}ty} : \text{Path} \times \text{Usage} \times \text{BindingType} \rightharpoonup \text{BindingType}$

$\mathit{use\text{-}ty}(p, \mathbf{bool}, \mathbf{bool}) \rightsquigarrow \mathbf{bool}$

$\mathit{use\text{-}ty}(p, u_1 \times u_2, \tau_1^b \times \tau_2^b) \rightsquigarrow \mathit{use\text{-}ty}(p, u_1, \tau_1^b) \times \mathit{use\text{-}ty}(p, u_2, \tau_2^b)$

$\mathit{use\text{-}ty}(p, \mathbf{irrel}, \&\langle L\rangle\, \mathbf{rc}\, \tau^v) \rightsquigarrow \&\langle L\rangle\, \mathbf{rc}\, \tau^v$

$\mathit{use\text{-}ty}(p, \mathbf{move}, \bullet\langle L\rangle\, \mathbf{rc}\, \tau^v) \rightsquigarrow \bullet\, \mathbf{rc}\, \tau^v \quad \text{if } L \prec p \text{ or } L \asymp p$

$\mathit{use\text{-}ty}(p, \mathbf{keep}, \bullet\langle L\rangle\, \mathbf{rc}\, \tau^v) \rightsquigarrow \&\langle L\rangle\, \mathbf{rc}\, \tau^v$

$\mathit{val\text{-}ty} : \text{BindingType} \to \text{ValueType}$

$\mathit{val\text{-}ty}(\mathbf{bool}) = \mathbf{bool}$

$\mathit{val\text{-}ty}(\tau_1^b \times \tau_2^b) = \mathit{val\text{-}ty}(\tau_1^b) \times \mathit{val\text{-}ty}(\tau_2^b)$

$\mathit{val\text{-}ty}(\&\langle L\rangle\, \mathbf{rc}\, \tau^v) = \&\langle L\rangle\, \mathbf{rc}\, \tau^v$

$\mathit{val\text{-}ty}(\bullet\langle L\rangle\, \mathbf{rc}\, \tau^v) = \bullet\, \mathbf{rc}\, \tau^v$

$\mathit{move} : \text{BindingType} \to \text{Usage}$

$\mathit{move}(\mathbf{bool}) = \mathbf{bool}$

$\mathit{move}(\tau_1^b \times \tau_2^b) = \mathit{move}(\tau_1^b) \times \mathit{move}(\tau_2^b)$

$\mathit{move}(\&\langle L\rangle\, \mathbf{rc}\, \tau^v) = \mathbf{irrel}$

$\mathit{move}(\bullet\langle L\rangle\, \mathbf{rc}\, \tau^v) = \mathbf{move}$

Fig. 4. Simplified $L^{\&}$ statics.

$$
\begin{array}{llll}
\text{Addr} & \iota & & \text{Heap}\ \ h\ ::=\ \diamond \mid h, \iota \coloneqq (n, v) \qquad \text{Value}\ \ v\ ::=\ \mathbf{true} \mid \mathbf{false} \mid (v_1, v_2) \mid \mathbf{rc}\,\iota \\
\text{Cont} & \kappa & ::= & \mathbf{if}\,[\,]\,\mathbf{then}\,\epsilon_1\,\mathbf{else}\,\epsilon_2 \mid \mathbf{if\text{-}then}\,[\,] \mid \mathbf{if\text{-}else}\,[\,] \mid ([\,], \epsilon) \mid (v, [\,]) \mid \mathbf{projl}\,[\,] \mid \mathbf{projr}\,[\,] \\
 & & & \mid \mathbf{rc}\,([\,] : \tau^v) \mid \mathbf{get}\,[\,] \mid \mathbf{dup}\,s\,[\,] \mid [\,]\,;\,\mathbf{drop}\,\vec{s}\,x \mid \mathbf{let}\,x : \tau^b = [\,]\,;\,\epsilon \mid \mathbf{let}\,x : \tau^b = v\,;\,[\,] \mid \blacksquare\,\Upsilon\,[\,] \\
\text{ContStack} & K & ::= & \diamond \mid K\,;\kappa \qquad \text{State}\ \ \sigma\ ::=\ (K \rhd \epsilon \parallel h) \mid (K \lhd v \parallel h)
\end{array}
$$

Fig. 5. Simplified $L^{\&}$ abstract machine.

originate from top-level owned bindings, which confer those lifetimes to the values that borrow them and any owned values they transitively contain through other owned values. The lifetime of a binding is always contained within its lexical scope, and the value in a binding, in accordance with the stack discipline, is always **drop**'ed by the end of the scope unless it is liberated by a move. In contrast, the only constraints on expressions of owned type follow from the grammatical structure of expressions. Borrows are always created, via **use** u x, from bindings.

We are now ready to discuss the typing rules for $L^{\&}$, which are presented in Figure 4 and given in full generality in the appendix (as with our other judgments). First two matters of notation: we typically write "$\rightsquigarrow$" or "$\leftrightsquigarrow$" instead of "=" when dealing with partial functions to bring them to the reader's attention. Often, when a rule hypothesizes a result to a partial function on some input, the mere fact that the function is defined there is the key part of the rule. We write $\vec{x}$ or $\langle x_1, \ldots, x_n \rangle$ for a list of xs in the metatheory.

The judgment $\Gamma\,; p \vdash \epsilon : \tau^v\,; \Upsilon$ should be read: in context Γ, at path p, ϵ has type τ^v and performs moves Υ. Υ tracks, for each top-level **rc** in each binding, whether it is owned and unmoved, **keep**, owned and moved, **move**, or borrowed, **irrel**. Any construct that binds variables, in simplified $L^{\&}$ just **let**, checks in its BT rule that the variables it binds are moved by the end of the scope. Υ exists only to ensure that $L^{\&}$ is linear, not merely affine, so that every heap object not returned by the main function is eventually freed in any well-typed program. This precludes trivial inference.

BT-Sub implements subtyping. Types are contravariant in their lifetimes. That is, $\tau_1^v <: \tau_2^v$ iff $L_2 \le L_1$ for each pair of corresponding lifetimes L_1 in τ_1^v and L_2 in τ_2^v. If we did not have subtyping, we could not type the following program, because the lifetimes of x and y would unify and the **get** of x happens after y is moved and, consequently, after the lifetime of x would end:

$$
\begin{array}{l}
\mathbf{let}\,x = (\mathbf{rc}\,0)\,;\ \mathbf{let}\,y = (\mathbf{rc}\,1)\,;\ \mathbf{let}\,_ = \mathbf{if}\ \cdots\ \mathbf{then}\,(\mathbf{use\,keep}\,x)\,\mathbf{else}\,(\mathbf{use\,keep}\,y)\,; \\
\mathbf{let}\,_ = (\mathbf{use\,move}\,y)\,;\ \mathbf{let}\,_ = \mathbf{get}\,(\mathbf{use\,keep}\,x)
\end{array}
$$

BT-If is interesting because it requires that both branches check under the same moves, Υ_2. If one branch moved an **rc** that the other did not, the program would not be compilable because we must have a static answer to what **rc** values are still available after the **if**. Also, note that BT-If merges the moves from the condition and the branches. The *merge* function takes two Υ contexts and returns a context where **rc** values are moved if they are moved in either input, returning a value only if they are not moved in both inputs, which would be unsound.

BT-Drop and BT-Use potentially move **rc** values, hence they upper bound relevant lifetimes by checking a condition of the form $L \prec p$ or $L \asymp p$ in functions *drop-rsrc* and *use-ty* respectively. Dually, BT-Get and BT-Use potentially access **rc** values, hence they lower bound relevant lifetimes by checking a condition of the form $p \le L$, directly in BT-Get and in function *dup-ty* in BT-Dup. These four rules are where the path tracked by the BT judgment is used.

The statics check subexpressions at paths that are carefully setup so that a type checker for $L^{\&}$ never needs to compare a *parent* and a *child* lifetime—e.g., $\swarrow \star$ is the parent of $\swarrow(\swarrow \star)$. Lifetimes pick out subexpressions. If one cares about how parents compare to children, it is necessary to disambiguate whether a path p refers to the moment immediately before an expression executes or

BE-Rc1
$$\frac{}{K \triangleright \mathbf{rc}\,(\epsilon : \tau^v) \parallel h \hookrightarrow K\,; \mathbf{rc}\,([\,] : \tau^v) \triangleright \epsilon \parallel h}$$

BE-Rc2
$$\frac{\iota \text{ fresh}}{K\,; \mathbf{rc}\,([\,] : \tau^v) \triangleleft v \parallel h \hookrightarrow K \triangleleft \mathbf{rc}\,\iota \parallel h(\iota \mapsto (1, v, \tau^v))}$$

BE-Get2
$$\frac{h(\iota) \rightsquigarrow (n, v, \tau^v)}{K\,; \mathbf{get}\,[\,] \triangleleft \mathbf{rc}\,\iota \parallel h \hookrightarrow K \triangleleft v \parallel h}$$

BE-Drop2
$$\frac{(K', h') \leftsquigarrow \textit{drop-step}(K, h, \overrightarrow{s\,x})}{K\,; ([\,]\,; \mathbf{drop}\,\overrightarrow{s\,x}) \triangleleft v \parallel h \hookrightarrow K' \triangleleft v \parallel h'}$$

BE-Tomb
$$\frac{}{K\,; \blacksquare\,\Upsilon\,[\,] \triangleleft v \parallel h \hookrightarrow K \triangleleft v \parallel h}$$

$$\textit{drop-step} : \text{ContStack} \times \text{Heap} \times \overrightarrow{(\text{Selector} \times \text{Var})} \rightharpoonup \text{ContStack} \times \text{Heap}$$
$$\textit{drop-step}(K, h, \langle\,\rangle) \rightsquigarrow (K, h)$$
$$\textit{drop-step}(K, h, \langle s_x\, x, \overrightarrow{s_y\, y}\rangle) \rightsquigarrow \textit{drop-step}((K\,; \blacksquare\,\{x : u_x\}\,[\,]), h', \overrightarrow{s_y\, y})$$
$$\text{if } h' \leftsquigarrow \textit{dec}(\textit{val-ty}(\textit{types}(K)(x)), s_x, \textit{vars}(K)(x), h) \text{ and } u_x \leftsquigarrow \textit{drop-rsrc}(\textit{seqr}(\textit{path}(K)), s_x, \textit{types}(K)(x))$$

$$\textit{dec} : \text{ValueType} \times \text{Selector} \times \text{Value} \times \text{Heap} \rightharpoonup \text{Heap}$$
$$\textit{dec}(\mathbf{bool}, \mathbf{bool}, \mathbf{true}, h) \rightsquigarrow h \qquad \textit{dec}(\mathbf{bool}, \mathbf{bool}, \mathbf{false}, h) \rightsquigarrow h$$
$$\textit{dec}(\tau_1^v \times \tau_2^v, s_1 \times s_2, (v_1, v_2), h) \rightsquigarrow \textit{dec}(\tau_2^v, s_2, v_2, \textit{dec}(\tau_1^v, s_1, v_1, h))$$
$$\textit{dec}(\bullet\langle L\rangle\, \mathbf{rc}\,\tau^v, \mathbf{select}, \mathbf{rc}\,\iota, h) \rightsquigarrow h(\iota \mapsto (n-1, v, \tau_h^v)) \qquad \text{if } h(\iota) \rightsquigarrow (n, v, \tau_h^v) \text{ and } n > 1$$
$$\textit{dec}(\bullet\langle L\rangle\, \mathbf{rc}\,\tau^v, \mathbf{select}, \mathbf{rc}\,\iota, h) \rightsquigarrow \textit{dec}(\tau^v, \textit{select}(\tau^v), v, (h - \iota)) \qquad \text{if } h(\iota) \rightsquigarrow (1, v, \tau_h^v)$$
$$\textit{dec}(\bullet\langle L\rangle\, \mathbf{rc}\,\tau^v, \mathbf{ignore}, \mathbf{rc}\,\iota, h) \rightsquigarrow h \qquad \textit{dec}(\&\langle L\rangle\, \mathbf{rc}\,\tau^v, s, \mathbf{rc}\,\iota, h) \rightsquigarrow h$$

$$\textit{vars}(\diamond) = \diamond \qquad \textit{types}(\diamond) = \diamond$$
$$\textit{vars}(K\,; \mathbf{let}\,x : t\,\overrightarrow{B} = v\,; [\,]) = \textit{vars}(K), x \coloneqq v \qquad \textit{types}(K\,; \mathbf{let}\,x : t\,\overrightarrow{B} = v\,; [\,]) = \textit{types}(K), x : t\,\overrightarrow{B}$$
$$\textit{vars}(K\,; \kappa) = \textit{vars}(K) \qquad \textit{types}(K\,; \kappa) = \textit{types}(K)$$

Fig. 6. Selected $L^{\&}$ evaluation rules.

immediately after. Parent–child comparisons do show up in the proof of progress and preservation for $L^{\&}$, but we defer that discussion to the appendix.

4.3 Dynamics of $L^{\&}$

Readers who are only interested in how inference is implemented can skip this section. In sum, we prove that $L^{\&}$'s type system is sound with respect to a dynamics modeling the key feature of a real implementation—a heap of reference counted objects whose reference counts are incremented by 1 when they are **dup**'ed and decremented by 1 when they are **drop**'ed and which are removed from the heap when their reference counts hit 0. Soundness of the dynamics implies that an $L^{\&}$ type checker can be used to certify the memory safety of inference results. In fact, Morphic does this. Note that we do not prove that inference will always produce a well-typed result, nor that inference will always find a dup-free solution if one exists, though we conjecture both to be true.

In more detail, we provide a small-step, abstract machine semantics (Figure 5 and Figure 6). Each heap cell contains a reference count, n, a value, v, and a type, which is used in the soundness proof, and the data of an **rc** is an index into the heap. As usual [10, Chapter 28] there are two kinds of machine states: the machine is either evaluating an expression ϵ, state $K \triangleright \epsilon \parallel h$, or returning a value v, state $K \triangleleft v \parallel h$. Here, h denotes the heap and K is a stack of continuations tracking where control flow will resume once the current subexpression has been fully evaluated. The only continuation which is not merely a partially evaluated $L^{\&}$ expression with a hole, $[\,]$, is the *tombstone*, $\blacksquare\,\Upsilon\,[\,]$. Tombstones record the additional **rc** values, Υ, that will be moved by the time execution is resumed at the tombstone. For instance, BE-Drop2 records what it is dropping by pushing tombstones onto K in *drop-step*. Tombstones are designed to allow shared reasoning

about certain inductive invariants between the **drop**, pair, **let**, **if**, and, in the extended version of $L^{\&}$ presented in the appendix, the **case** and call constructs.

The evaluation judgment, $D \vdash \sigma \hookrightarrow \sigma'$, should be read: given definitions D, a machine in state σ steps to state σ'. BE-Rc1 and BE-Rc2 are typical evaluation rules. There is one BE rule for each subexpression of each $L^{\&}$ construct that, like BE-Rc1, recurses into that subexpression and pushes an appropriate continuation to the stack. In addition, there is one BE rule, like BE-Rc2, that runs once all subexpressions have been fully evaluated, performing the work of the operation itself.

BE-Rc2 places a new value in the heap at a fresh address with reference count 1. BE-Get2 takes value $\mathbf{rc}\,\iota$ and reads address ι from the heap, getting stuck if the heap does not contain a value at ι. BE-Drop2 calls *drop-step*, which calls *dec* on the value of each binding in $\overrightarrow{s\,x}$. *dec* decrements the reference count of each owned $\mathbf{rc}\,\iota$ in its argument where the selector is **select**, removing ι from the heap if the reference count hits 0.

We state soundness of the dynamics as a pair of progress and preservation lemmas. For expression states, if $K \rhd \epsilon \parallel h$ ok then there exists a unique σ' such that $K \rhd \epsilon \parallel h \hookrightarrow \sigma'$ and σ' ok. The analogous statement holds for value states. The inductive hypothesis, the ok judgment, entails a number of conditions beyond well-typedness. The most interesting are:

- The reference count of every object is precisely the number of owners of that object on the heap reachable from unmoved owners on the stack through owned values.
- Any **rc** in any stack frame that has been moved according to the running set of moves tracked by the tombstones in K, is dead at the current dynamic path according to its dynamic lifetime. By dynamic path/lifetime we mean the path/lifetime prefixed by the path to the current stack frame implicit in K when $L^{\&}$ is extended with call expressions, as in the appendix.
- If a borrow of $\mathbf{rc}\,\iota$ is live at the current (dynamic) path according to its (dynamic) lifetime, then there is some variable on the stack that guarantees that it is safe to access. That is, there is some owned binding that is live at the current (dynamic) path according to its (dynamic) lifetime from which ι can be transitively reached through owned values.

The proofs of these progress and preservation lemmas are painfully involved. A large section of the appendix is dedicated to this.

4.4 The Inference Procedure

Inference proceeds in two phases, described in Figures 7 and 8. We caution the reader against trying to understand these phases in isolation since the first generates constraints for the second.

The first phase (Figure 7) traverses the source program, introducing fresh resource variables to produce an L^{inf} program and accumulating a set of constraints about the program and the resource variables. The judgment in Figure 7, $\hat{\Gamma}\,; p \vdash e : \mathfrak{t} \rightsquigarrow \mathfrak{e}\,; Q$, should be read: given annotated versions $\hat{\Gamma}$ of the bindings in scope, at path p, L^{src} expression e lifts under type $\mathfrak{t}$ to L^{inf} expression $\mathfrak{e}$ (of type $\mathfrak{t}$) with constraints Q. On left of the "$\rightsquigarrow$" we have the inputs, and on the right the outputs. Since constraints are placed on resource variables, functions and constraints applied to types should be interpreted as broadcast over the resource variables they mention. The conclusion of I-Use asserts that there is a `flow` constraint between the corresponding resources of a binding $\mathfrak{t}_{\text{bind}}$ and its use $\mathfrak{t}_{\text{use}}$, and I-Let asserts that the scope of each resource in the binding $\mathfrak{t}_1$ is *seqr* of the path p to the **let**. We write constraint names in monospaced text to distinguish them from metafunctions.

Non-empty lifetimes originate from the $\texttt{lt}_{\text{approx}}$ and $\texttt{lt}_{\text{prec}}$ constraints emitted by I-Get, in analogy to our informal discussion where we reasoned backward from **get** operations to what must be live where. *get-constr* and *get-constr-heap* are a pair, the former is applied when broadcasting to a resource on a top-level **rc**, and the latter is applied when broadcasting to a resource on a nested **rc**. All of the rules follow from our informal discussion in Section 3, save that, for uniformity, we

$$\text{I-USE}\quad \frac{\hat{\Gamma}(x) = \mathfrak{t}_{\text{bind}}}{\hat{\Gamma}\,;\, p \vdash x\,;\, \mathfrak{t}_{\text{use}} \rightsquigarrow x \;\mathbf{as}\; \mathfrak{t}_{\text{use}}\,;\, \{\texttt{flow}(\mathfrak{t}_{\text{bind}}, \mathfrak{t}_{\text{use}})\}}$$

$$\text{I-RC}\quad \frac{\hat{\Gamma}\,;\, p \vdash e\,;\, \mathfrak{t} \rightsquigarrow \mathfrak{e}\,;\, Q \qquad Q' = Q \cup \{\texttt{access}(\mathfrak{r}) = \bullet, \texttt{access}(\mathfrak{t}) = \texttt{storage}(\mathfrak{t})\}}{\hat{\Gamma}\,;\, p \vdash \mathbf{rc}\, e\,;\, \mathfrak{r}\;\mathbf{rc}\;\mathfrak{t} \rightsquigarrow \mathbf{rc}\;\mathfrak{e}\,;\, Q'}$$

$$\text{I-LET}\quad \frac{\begin{array}{cc}(\mathfrak{t}_1, Q_{\text{fresh}}) \hookleftarrow \mathit{fresh}(t_1) & \hat{\Gamma}\,;\, \mathit{seql}(p) \vdash e_1\,;\, \mathfrak{t}_1 \rightsquigarrow \mathfrak{e}_1\,;\, Q_1 \\ \hat{\Gamma}, x : \mathfrak{t}_1\,;\, \mathit{seql}(\mathit{seqr}(p)) \vdash e_2\,;\, \mathfrak{t}_2 \rightsquigarrow \mathfrak{e}_2\,;\, Q_2 & Q = Q_{\text{fresh}} \cup Q_1 \cup Q_2 \cup \{\texttt{scope}(\mathfrak{t}_1) = \mathit{seqr}(p)\}\end{array}}{\hat{\Gamma}\,;\, p \vdash (\mathbf{let}\, x : t_1 = e_1\,;\, e_2)\,;\, \mathfrak{t}_2 \rightsquigarrow (\mathbf{let}\, x : \mathfrak{t}_1 = \mathfrak{e}_1\,;\, \mathfrak{e}_2)\,;\, Q}$$

$$\text{I-GET}\quad \frac{\begin{array}{c}(\mathfrak{r}\;\mathbf{rc}\;\mathfrak{t}_{\text{src}}, Q_{\text{fresh}}) \hookleftarrow \mathit{fresh}(\mathbf{rc}\; t) \qquad \hat{\Gamma}\,;\, \mathit{seql}(\mathit{seql}(p)) \vdash e\,;\, \mathfrak{r}\;\mathbf{rc}\;\mathfrak{t}_{\text{src}} \rightsquigarrow \mathfrak{e}\,;\, Q \\ Q' = Q \cup Q_{\text{fresh}} \cup \{\texttt{lt}_{\texttt{approx}}(\mathfrak{r}) \geq p, \texttt{lt}_{\texttt{prec}}(\mathfrak{r}) \geq p\} \cup \texttt{flow}(\mathfrak{r}_{\text{src}}, \mathfrak{r}_{\text{dst}}) \cup \texttt{get}(\mathfrak{r}, \mathfrak{t}_{\text{src}}, \mathfrak{t}_{\text{dst}}) \cup \mathit{get\text{-}constr}(\mathfrak{t}_{\text{src}}, \mathfrak{t}_{\text{dst}})\end{array}}{\hat{\Gamma}\,;\, p \vdash \mathbf{get}\, e\,;\, \mathfrak{t}_{\text{dst}} \rightsquigarrow \mathbf{get}\;\mathfrak{e}\;\mathbf{as}\;\mathfrak{t}_{\text{dst}}\,;\, Q'}$$

$\mathit{get\text{-}constr}(\mathfrak{r}_{\text{src}}, \mathfrak{r}_{\text{dst}}) = \{\texttt{access}(\mathfrak{r}_{\text{src}}) = \texttt{access}(\mathfrak{r}_{\text{dst}})\}$

$\mathit{get\text{-}constr\text{-}heap}(\mathfrak{r}_{\text{src}}, \mathfrak{r}_{\text{dst}}) = \{\texttt{access}(\mathfrak{r}_{\text{src}}) = \texttt{access}(\mathfrak{r}_{\text{dst}}), \texttt{storage}(\mathfrak{r}_{\text{src}}) = \texttt{storage}(\mathfrak{r}_{\text{dst}})\}$

$\mathit{fresh} : \text{SourceType} \longrightarrow \text{InferenceType} \times \text{Constraints}$

$\mathit{fresh}(\mathbf{bool}) \rightsquigarrow (\mathbf{bool}, \{\,\})$

$\mathit{fresh}(t_1 \times t_2) \rightsquigarrow (\mathfrak{t}_1 \times \mathfrak{t}_2, Q_1 \cup Q_2) \quad \text{where } (\mathfrak{t}_1, Q_1) \hookleftarrow \mathit{fresh}(t_1),\ (\mathfrak{t}_2, Q_2) \hookleftarrow \mathit{fresh}(t_2)$

$\mathit{fresh}(\mathbf{rc}\; t) \rightsquigarrow (\mathfrak{r}\;\mathbf{rc}\;\mathfrak{t}, \{\texttt{primary}(\mathfrak{r}) = \texttt{access}(\mathfrak{r})\} \cup Q) \quad \text{where } (\mathfrak{t}, Q) \hookleftarrow \mathit{fresh\text{-}heap}(t),\ \mathfrak{r} \text{ is fresh}$

$\mathit{fresh\text{-}heap} : \text{SourceType} \longrightarrow \text{InferenceType} \times \text{Constraints}$

$\mathit{fresh\text{-}heap}(\mathbf{bool}) \rightsquigarrow (\mathbf{bool}, \{\,\})$

$\mathit{fresh\text{-}heap}(t_1 \times t_2) \rightsquigarrow (\mathfrak{t}_1 \times \mathfrak{t}_2, Q_1 \cup Q_2) \quad \text{where } (\mathfrak{t}_1, Q_1) \hookleftarrow \mathit{fresh\text{-}heap}(t_1),\ (\mathfrak{t}_2, Q_2) \hookleftarrow \mathit{fresh\text{-}heap}(t_2)$

$\mathit{fresh\text{-}heap}(\mathbf{rc}\; t) \rightsquigarrow (\mathfrak{r}\;\mathbf{rc}\;\mathfrak{t}, \{\texttt{primary}(\mathfrak{r}) = \texttt{storage}(\mathfrak{r})\} \cup Q) \quad \text{where } (\mathfrak{t}, Q) \hookleftarrow \mathit{fresh\text{-}heap}(t),\ \mathfrak{r} \text{ is fresh}$

Fig. 7. Selected L^{src} to L^{inf} lifting rules.

Solving for approximate lifetimes:

$\texttt{lt}_{\texttt{approx}}(\mathfrak{r}_{\text{bind}}) \geq (\mathbf{if}\ \texttt{flow}(\mathfrak{r}_{\text{bind}}, \mathfrak{r}_{\text{use}})\ \mathbf{then}\ \texttt{lt}_{\texttt{approx}}(\mathfrak{r}_{\text{use}})\ \mathbf{else}\ \varnothing)$

Solving for modes:

$\texttt{access}(\mathfrak{r}_{\text{bind}}) \geq (\mathbf{if}\ \texttt{flow}(\mathfrak{r}_{\text{bind}}, \mathfrak{r}_{\text{use}})\ \mathbf{then}\ \texttt{access}(\mathfrak{r}_{\text{use}})\ \mathbf{else}\ \&)$

$\texttt{access}(\mathfrak{r}_{\text{use}}) \geq (\mathbf{if}\ \texttt{flow}(\mathfrak{r}_{\text{bind}}, \mathfrak{r}_{\text{use}}) \land (\texttt{lt}_{\texttt{approx}}(\mathfrak{r}_{\text{use}}) \not\prec \texttt{scope}(\mathfrak{r}_{\text{bind}}))\ \mathbf{then}\ \texttt{access}(\mathfrak{r}_{\text{bind}})\ \mathbf{else}\ \&)$

$$\left.\begin{array}{l}\texttt{storage}(\mathfrak{r}_{\text{bind}}) \geq (\mathbf{if}\ \texttt{flow}(\mathfrak{r}_{\text{bind}}, \mathfrak{r}_{\text{use}})\ \mathbf{then}\ \texttt{storage}(\mathfrak{r}_{\text{use}})\ \mathbf{else}\ \&) \\ \texttt{storage}(\mathfrak{r}_{\text{use}}) \geq (\mathbf{if}\ \texttt{flow}(\mathfrak{r}_{\text{bind}}, \mathfrak{r}_{\text{use}})\ \mathbf{then}\ \texttt{storage}(\mathfrak{r}_{\text{bind}})\ \mathbf{else}\ \&)\end{array}\right\}\ \text{enforce an equality constraint}$$

Solving for precise lifetimes:

$\texttt{lateral-flow}(\mathfrak{r}_{\text{bind}}, \mathfrak{r}_{\text{use}}) \Leftarrow \texttt{flow}(\mathfrak{r}_{\text{bind}}, \mathfrak{r}_{\text{use}}) \land (\texttt{primary}(\mathfrak{r}_{\text{use}}) = \&)$

$\texttt{vertical-flow}(\mathfrak{r}_{\text{parent}}, \mathfrak{r}_{\text{dst}}) \Leftarrow \texttt{get}(\mathfrak{r}_{\text{parent}}, \mathfrak{r}_{\text{src}}, \mathfrak{r}_{\text{dst}}) \land (\texttt{primary}(\mathfrak{r}_{\text{src}}) = \bullet) \land (\texttt{primary}(\mathfrak{r}_{\text{dst}}) = \&)$[5]

$\texttt{lt}_{\texttt{prec}}(\mathfrak{r}_1) \geq (\mathbf{if}\ \texttt{lateral-flow}(\mathfrak{r}_1, \mathfrak{r}_2) \lor \texttt{vertical-flow}(\mathfrak{r}_1, \mathfrak{r}_2)\ \mathbf{then}\ \texttt{lt}_{\texttt{prec}}(\mathfrak{r}_2)\ \mathbf{else}\ \varnothing)$

Fig. 8. Mode and lifetime inference rules.

track both a storage and access mode on top-level resources. The storage mode is not actually used. We record a `primary` constraint in *fresh* saying that the $L^{\&}$ mode of a top-level resource will be its access mode, and the $L^{\&}$ mode of a nested **rc** will be its storage mode.

[5]Note that, by construction, we always have $\texttt{primary}(\mathfrak{r}_{\text{parent}}) = \&$, and only potentially have $(\texttt{primary}(\mathfrak{r}_{\text{src}}) = \bullet) \land (\texttt{primary}(\mathfrak{r}_{\text{dst}}) = \&)$ for the top-level resources in the $\mathfrak{t}_{\text{src}}$ and $\mathfrak{t}_{\text{dst}}$.

Once we have an L^{inf} program, we compute approximate lifetimes, modes, then precise lifetimes by iterating the rules in Figure 8. Consider the equation under "Solving for approximate lifetimes." We start out with some initial value for $\texttt{lt}_{\texttt{approx}}$, which is a map from resource variables to lifetimes, i.e., with an initial value for the approximate lifetime associated to each resource variable. These values are potentially set by I-Get and are otherwise $\varnothing$. The equation says that, if resource $\mathfrak{r}_{\text{bind}}$ flows into resource $\mathfrak{r}_{\text{use}}$, then the approximate lifetime at $\mathfrak{r}_{\text{bind}}$ is at least the approximate lifetime at $\mathfrak{r}_{\text{use}}$. We repeatedly update $\texttt{lt}_{\texttt{approx}}$ until the equation is satisfied by setting $\texttt{lt}_{\texttt{approx}}(\mathfrak{r}_{\text{bind}}) \coloneqq \texttt{lt}_{\texttt{approx}}(\mathfrak{r}_{\text{bind}}) \sqcup \texttt{lt}_{\texttt{approx}}(\mathfrak{r}_{\text{use}})$ wherever we have a $\texttt{flow}(\mathfrak{r}_{\text{bind}}, \mathfrak{r}_{\text{use}})$ constraint. Because of the way the rules in Figure 7 setup the resource variables, the same variable might show up as the "bind" in one `flow` constraint and as the "use" in another. But, iteration is guaranteed to eventually terminate because the lifetimes compatible with the program form a finite lattice bounded by the branching depth. In the next section, we will discuss how to extend $L^{\&}$ with functions, requiring adjustment to the definition of lifetimes, but it will remain true that the relevant lifetimes form a finite lattice.

Formally: interpreting modes as elements of the lattice $\& < \bullet$, booleans as elements of the lattice **false** < **true**, and lifetimes as elements of the lattice from Section 4.1, each group of equations in Figure 8 defines a monotone function computing one-step consequences over a fixed product of functions from resource variables to lattices. These products are endowed with a component-wise lattice structure, which follows from the pointwise lattice structure on each function induced by its codomain. Existence of a least fixed point for each group of equations is justified by Knaster–Tarski [23], and termination is justified by finiteness. In other words, our equations can be viewed as a Datalog program in the spirit of datalog$^\circ$ [1].

Next, we perform *reification* to turn our L^{inf} program into an $L^{\&}$ program. This process is straightforward and we do not list the rules here, though they are given in the appendix. L^{inf} types are transformed into $L^{\&}$ types by reading off the mode and lifetime information we computed during inference. The $L^{\&}$ mode is whatever mode was declared `primary` by the I rules. Since **let** binds a new variable, it must drop any **rc** values in the binding not moved in its body to uphold linearity. The **else** branch of **if** must drop any variables (from any scope) that are moved along the **then** branch but not moved in the body of the **else**, and symmetrically for the **then** branch. The only other interesting transformations happen at **use** and **get**. Every **use** becomes a **use** followed by a **dup**, yielding three cases: (1) if the usage is by ownership and we are allowed to move because the precise lifetime has ended, then the **use**'s usage flag is set to **move** and the **dup**'s selector is set to **ignore**; (2) if we are not allowed to move, the **use**'s usage flag is set to **keep** and the **dup**'s selector is set to **select**; (3) if the usage is by borrow, then the **use**'s usage flag is set to **irrel** and the **dup**'s selector is set to **ignore**. Every **get** becomes a **get** followed by a **dup**. If we require the output of a **get** by ownership, the **dup**'s selector is set to **select**, otherwise it is set to **ignore**.

A final observation: interestingly, our approximate lifetimes are neither over-approximate nor under-approximate. Precise lifetimes might be shorter than approximate lifetimes because of a **dup** or move that cuts off the flow of lifetimes, as embodied by the `lateral-flow` rule. Surprisingly, they also might be longer. The `vertical-flow` rule says that if we are doing a **get** on an **rc** that owns its content (i.e., $\texttt{primary}(\mathfrak{r}_{\text{src}}) = \bullet$) and not immediately **dup**'ing it (i.e., $\texttt{primary}(\mathfrak{r}_{\text{dst}}) = \&$), then the outer **rc** (i.e., $\mathfrak{r}_{\text{parent}}$) passed to the **get** must be live at least as long as the borrow we return. Approximate lifetimes reason about nested **rc** types as if they are top-level, i.e., the scope assigned to a nested **rc** for the purpose of computing whether it escapes via the output of a **get** is the same as the scope assigned to the top-level **rc**, so there is no soundness issue in escape analysis from failing to propagate lifetimes vertically. However, approximate lifetimes, unlike precise lifetimes, are not sound for the purpose of determining where to insert **drop** operations. In sum, before we

have modes we do not know whether we can drop an outer **rc** while an **rc** that we obtained from it via a **get** is still live, and approximate lifetimes are not conservative in this respect.

5 Handling Real Programs

5.1 Functions

While we track the relationships between program points within the same function in a fine-grained manner, we approximate lifetimes that appear in a function's return type using *lifetime variables*, denoted α, that are ordered after all local lifetimes under $\leq$. Lifetimes become: $L ::= \ell \mid \sqcup_i \alpha_i$, where $\sqcup_i \alpha_i$ is a formal join of lifetimes variables; a join over the empty set is the empty lifetime and $\sqcup_i \alpha_i \leq \sqcup_j \alpha'_j$ iff $\{\vec{\alpha}\} \subseteq \{\vec{\alpha'}\}$. The important information about a function from a caller's perspective is the flow relationship between parameters and returns. In Rust, this relationship is represented by a constraint clause. If we adopted this approach in $L^{\&}$, it would look like:

$$\begin{aligned}&\mathbf{def}\ f\langle\alpha_1,\ \alpha_2\rangle(b:\mathbf{bool},\ \mathrm{tt}:\&\langle\alpha_1\rangle\ \mathbf{rc},\ \mathrm{ff}:\&\langle\alpha_2\rangle\ \mathbf{rc}):\&\langle\alpha_1\rangle\ \mathbf{rc}\times\&\langle\alpha_2\rangle\ \mathbf{rc}\\&\qquad\mathbf{where}\ \alpha_1\leq\alpha_2=(\mathrm{tt},\ \mathbf{if}\ b\ \mathbf{then}\ \mathrm{tt}\ \mathbf{else}\ \mathrm{ff})\end{aligned}$$

Joins of lifetime variables are equally expressive. E.g., the previous example becomes:

$$\mathbf{def}\ f\langle\alpha_1,\ \alpha_2\rangle(b:\mathbf{bool},\ \mathrm{tt}:\&\langle\alpha_1\rangle\ \mathbf{rc},\ \mathrm{ff}:\&\langle\alpha_1\sqcup\alpha_2\rangle\ \mathbf{rc}):\&\langle\alpha_1\rangle\ \mathbf{rc}\times\&\langle\alpha_2\rangle\ \mathbf{rc}=\cdots$$

Using joins of lifetime variables allows an elegant extension of inference to handle functions. When lifting to L^{inf}, we initialize each lifetime in a function's return type with a fresh lifetime variable and each lifetime in the argument type with $\varnothing$. Performing inference as usual will potentially introduce lifetime joins in the argument type, inducing `flow` relationships between argument and return and affecting recursive calls. Iterating to a fixed point will produce the right relationships.

5.2 Recursive Types

Consider the type of linked lists of integers: $\mathbf{list} \coloneqq \mu a.\, \mathbf{unit} + \mathbf{int} \times (\bullet\, \mathbf{rc}\, a)$. Note that we need a layer of indirection mediating the recursive occurrence, a, lest values of **list** be of unbounded stack size. As discussed in Section 3.3, the operation of borrowing can only be performed at the top level. In $L^{\&}$, the mode annotation $\bullet$ on the $\mathbf{rc}\, a$ would be a statement about the mode, not only of the first, top-level **rc** in the **list**, but also all subsequent, nested **rc** values. How, then, can we operationalize borrowing? Rust does not suffer from this problem because & is a type former. In Rust, we can simply place a borrow around the entire list and it distributes appropriately, e.g.

```
enum List { Nil, Cons(i32, Rc⟨List⟩) }
let ℓ : &List = ··· ; match ℓ { Nil ⇒ ··· , Cons(h : &i32, t : &Rc⟨List⟩) ⇒ ··· } ;
```

Standard isorecursive μ types, and nominal types for that matter, compress the information about a type too much for $L^{\&}$. The solution is to require, as a hypothesis to BT-Let, that the type τ^b of the binding is a *guarded* type, in the sense that all μ types in τ^b appear behind an **rc**, or, in general, an RC'd type such as an array.[6] This ensures that there is distinct mode information for top-level **rc** values, allowing us to extend the definition of **use** soundly.

In the Morphic compiler, we take the user's code, roughly an SML program, insert any implicit **rc** types, guard each type to produce an (extended) L^{src} program, and only then perform borrow inference to produce an (extended) $L^{\&}$ program. The process of guarding a user program is not especially challenging, but L^{src} and $L^{\&}$ must support slightly more general type equivalence witnesses than the typical calculus, namely folding and unfolding operations that can be applied at

[6] There is some choice where to assert guardedness, i.e., whether we should assert guardedness where we bind values in **let** etc. or where we consume values in **use** etc. The former choice turns out to be more convenient.

any position, not just at the top level. For example, the following unguarded program

$$\textbf{let}\,x : \textbf{list} = \textbf{fold}(\textbf{inr}((0, \textbf{rc}(\textbf{fold}(\textbf{inl}(\textbf{unit})))))) \,;$$
$$\textbf{case}\,\textbf{unfold}(x)\ \{\ \textbf{inl}(y : \textbf{unit}) \Rightarrow \cdots,\ \textbf{inr}(y : \textbf{int} \times \textbf{rc}\,\textbf{list}) \Rightarrow \textbf{get}(\textbf{projr}(y))\ \}$$

can be transformed into the guarded program

$$\textbf{let}\,x : \textbf{unit} + \textbf{int} \times \textbf{rc}\,\textbf{list} = \textbf{fold}(\textbf{inr}((0, \textbf{rc}(\textbf{fold}(\textbf{inl}(\textbf{unit})))))) \,;$$
$$\textbf{case}\,\textbf{unfold}(x)\ \{\ \textbf{inl}(y : \textbf{unit}) \Rightarrow \cdots,\ \textbf{inr}(y : \textbf{int} \times \textbf{rc}\,(\textbf{unit} + \textbf{int} \times \textbf{rc}\,\textbf{list})) \Rightarrow \textbf{get}(\textbf{projr}(y))\ \}$$

by adjusting the folds to map $(\textbf{unit} + \textbf{int} \times \textbf{rc}\,(\textbf{unit} + \textbf{int} \times \textbf{rc}\,\textbf{list}))$ to $(\textbf{unit} + \textbf{int} \times \textbf{rc}\,\textbf{list})$, rather than $(\textbf{unit} + \textbf{int} \times \textbf{rc}\,\textbf{list})$ to **list**, and adjusting the unfolds to match. The first program is unguarded because x has type **list**. The second is guarded because x has type $\textbf{unit} + \textbf{int} \times \textbf{rc}\,\textbf{list}$—the only μ appears in **list** which appears under an **rc**.

We emphasize that guarding does not impact program execution, but, in some rules, $L^{\&}$ bans one member of what would typically be each μ type's equivalence class, so we must re-type user code under another member before performing borrow inference. We note that the only types that cannot be guarded are those with μ variables that do not appear under an **rc**. Such types are not compilable anyway, unless the backend allows values of unbounded stack size.

6 Implementation Considerations

6.1 Higher-Order and Polymorphic Functions

A production-ready programming language is expected to have features that we have not included in L^{src}, such as higher-order functions and type parameters. These source language features can be compiled to L^{src}. To achieve best results from borrow inference, types and control flow should be concrete, so the most precise lifetimes and modes can be inferred. As such, one should monomorphize to get rid of type parameters, and defunctionalize to get rid of higher-order functions. Lambda set specialization (LSS) [7] can be particularly powerful when combined with borrow inference. LSS avoids the poisoning problem [28] suffered by traditional interprocedural analyses by specializing functions with respect to the possible lambdas at each call site, similar to the way Rust and C++ specialize with respect to a single lambda using type parameters. Performance of function calls does not degrade as new, unrelated calls to those functions are added.

6.2 Mode Specialization

In the mold of LSS, our borrow inference specializes with respect to modes, here to avoid poisoning the determined memory management schemes. Specialization allows programmers to reason modularly at function boundaries, despite the interprocedural nature of the transformation. Consider a function f whose return is related to its argument. Ideally, we can call f with a borrow and get a borrow back. Without mode specialization, if there is even a single instance where the result of a call to f is forced to be owned, the inferred type of f will take an owner and return an owner, adding unnecessary RC operations wherever the f function ought to be called with a borrow. With mode specialization, we get two versions of f, one taking a borrow and returning a borrow, and one taking an owner and returning an owner. To avoid specializations with the same dynamic behavior, a production version of borrow inference might merge specializations whose modes induce the same RC operations.

6.3 Important Optimizations

An implementation of borrow inference encounters several issues not addressed by the formalism:

- Functional programmers often rely on tail-call elimination. Inhibiting such elimination has significant performance ramifications, something we observed in writing our own benchmarks. To avoid inhibiting tail-calls during reification, it suffices to treat variable occurrences in the argument position of a tail-call as escaping, that is, to extend their lifetimes during the $\mathtt{lt}_{\mathsf{approx}}$ analysis. It does not suffice to directly set the modes of those occurrences to owned. The former is a weaker constraint allowing borrows that originate from outside the SCC.
- Because the inference algorithm makes borrowing decisions at variable occurrences, to increase the number of borrowing opportunities, the program should be converted to A-normal form [9] before the algorithm is run. Even if an arm of a case expression is simply an array literal, that array literal should be bound in the transformed program before it is returned. If it is unused, the algorithm can potentially infer the case expression's type to be a borrowed **rc** with the empty lifetime and insert a drop before the array is even returned (our system is more forgiving than Rust in that it allows dead **rc** values to be manipulated so long as their heap content is not accessed).
- Whereas the formalism drops values only at the ends of scopes, an implementation should drop values at the earliest point permitted by their lifetimes, effectively "sliding" drops up as far as possible. This is especially important in pure languages whose implementations perform mutation when the dynamic reference-count is 1.
- As inference is able to move a stack value, but not move a value out of an **rc**, values should be aggressively stack allocated. The only RC'd objects in Morphic are arrays and the pointers used to break up mutually recursive types so that they can be represented in memory.

6.4 Mutually Recursive Types

User-facing languages typically feature mutually recursive nominal types, rather than the singly recursive μ types of the formalism. Supporting such nominal types requires generalizing the guarding procedure. We say that T_1 depends on T_2 if it contains any occurrences of T_2. Now, consider the dependency graph of types. We can guard the set of nominal types as follows. For each type, T, in a cyclic strongly connected component (SCC) of the dependency graph, produce a new "entry" type whose body is the body of T unfolded until all references to itself or other nominal types in the same SCC are behind an **rc**. Replace all external occurrences of nominal types, i.e., occurrences from outside their SCCs (including in the newly produced entry types), with their entry types.

6.5 An Escape Hatch

One can construct L^{src} programs where a dup must be inserted in one of two branches to complete the program, meaning that the best completion depends on the distribution of program inputs—an example is provided in the appendix. In a hot loop, this difference might be critical. A **dup** operation can be added to L^{src} which is, semantically, the identity, but allows the user to guide completion. Then, the user can take control by manually inserting one of the two **dup** operations, freeing inference from making the choice. Note that we do not do this for our benchmarks, since we are trying to demonstrate the power of inference without user intervention.

6.6 Reference Semantics

In this paper, we define a system with value semantics. However, it is possible to extend that system with types that have reference semantics without affecting the existing constructs in any way. For example, we could add an OCaml-style **ref** type. The **get** operation on such a type, unlike the **get**

operation on **rc**, must always return owned values, as the following example demonstrates:

$$\textbf{let}\, a = \textbf{ref}(\textbf{ref}(1));\ \textbf{let}\, b = \textbf{dup}(a);\ \textbf{let}\, x = \textbf{get}(a);$$
$$\textbf{set}(b, \textbf{ref}(0));\ \text{print}(\textbf{get}(x))$$

If x is allowed to be a borrow, then the **set** decrements the reference count of the underlying object to 0, and the **get** inside print is a use-after-free. Something that is sound is a **swap** function that takes an owned **ref** and a new item and replaces the content of the **ref** with the new item, returning the old item. This **ref** is similar to, but not identical to, Rust's Rc⟨Cell⟨T⟩⟩.

6.7 Programming with Reference Counting

The statics of $L^{\&}$ ensure that, if the dynamic reference count of an object is 1 upon entry to a function, the object is exclusively held by that function and the object can never be read through any handle after the function's exit not explicitly yielded back to the caller. This invariant enables a powerful optimization: if we type built-in functions that want to mutate their arguments, such as pushing to an array, as taking owners, we can perform a dynamic check in those functions to see if the reference count is 1 and mutate in-place if so without violating the value semantics of the source language.[7] Inference will propagate ownedness, ensuring that user defined functions that call these built-ins also benefit. Consider the following Morphic code.

```
map_rec(arr : Array t, f : t → u, i : Int) : Array u =
  if i < 0 { [ ] } else { Array.push(map_rec(arr, f, i − 1), f(Array.get(arr, i))) }
map(arr : Array t, f : t → u) : Array u = map_rec(arr, f, len(arr) − 1)
```

When we run this code on an input array, it incrementally builds up the output array element by element. In the process of construction, starting from the empty array, the accumulator always has reference count 1, and the `Array.push` is guaranteed to mutate the array in place. In an implementation with tracing garbage collection that does not reference count arrays, this code will run, unacceptably, in quadratic time, necessitating additional primitives for efficiency.

The two big downsides of reference counting discussed in the literature are its lack of ability to deal with reference cycles and the total time spent on memory management operations as compared to a tracing collector. Morphic sidesteps the problem of cyclic data by being a call-by-value, pure functional language. Because the heap is acyclic, all memory is eventually freed without the need for a cycle collector. With this paper's contribution of borrow inference, it is clear that almost all of the RC increments and decrements of traditional reference-counting are unnecessary, eliminating most of the overhead of reference counting.

7 Evaluation

Morphic is a pure functional language, roughly a subset of SML, that compiles to LLVM IR. Morphic has implementations of three memory management strategies: (1) borrow inference as described in this paper; (2) a reference counting strategy without borrowing, which closely matches the strategy implemented in Perceus (discussed further below) and so serves as comparison point to current state-of-the-art static reference count elision systems; (3) a conservative garbage collector, which serves as a baseline memory management strategy giving a sense of how much improvement is possible using borrow inference over traditional techniques.

Morphic has first class support for arrays with logically immutable operations. In the backend, the Array type is represented as a length, a capacity, and a pointer to a contiguous chunk of memory with a reference count prepended. Under our two RC-based memory management strategies,

[7] The idea of performing an in-place update based on an array's reference count goes back at least to Hudak and Bloss [12].

we perform an RC-1 optimization: operations such as push and pop on an Array with reference count 1 mutate the memory in-place. If the RC is not 1, the entire chunk of memory backing the array is copied, and all objects within the array get their RC incremented, a copy-on-write (COW) strategy. The conservative garbage collector we use [19] is an implementation of Boehm–Demers–Weiser (BDWGC) [5]. The reference count still exists under this strategy, but all code incrementing and decrementing is removed. Without an (accurate) reference count, as we cannot perform the RC-1 optimization and mutate in-place, some benchmarks, like the Sieve of Eratosthenes, have unacceptable quadratic performance. As a result, we implemented a persistent array data structure for use in benchmarks that compare our strategy to BDWGC. In particular, we implemented Clojure's [11] persistent vector, an immutable version of Bagwell's ideal hash tries [2, 3]. We only ever compare program runs of our strategy to BDWGC where persistent arrays are used for both.

The baseline reference-counting strategy is a slight modification of Perceus, the algorithm described by Reinking et al. [22]. Perceus does not have borrowing, assuming all modes are owned and eliding reference counts only where objects are moved, at their last syntactic occurrences. Though the setup of borrow inference is quite different, we can simulate Perceus's behavior by constraining all modes to be owned. Reinking et al. do not describe how they handle built-in operations such as tuple projection, but requiring the argument of a projection to be owned results in duping all the elements of the tuple and dropping all but the projected element, while duping only the projected element obviously suffices. Therefore, to be fair to Perceus, we allow the inputs of certain built-ins to be borrowed, namely when checking the discriminant of a union, projecting a tuple field, indexing an array, getting the length of an array, and outputting to stdout and stderr. Perceus also introduces reuse optimization, allowing a destructor followed by a constructor to reuse the same allocation—a generalized RC-1 optimization. We did not implement this.

We evaluate our borrow inference system on a wide variety of benchmark programs written in Morphic. Five of the benchmarks are from Reinking et al. [22], mainly focusing on functional data structures and algorithms. The other eight benchmarks are new, and cover a variety of programming tasks, intended to be more representative of real-world programming. The benchmarks are:

- `calc`: A calculator utility that parses an arithmetic expression (using the same custom parser combinator library) and then evaluates it.
- `cfold`: Generates a large symbolic expression and performs constant folding.
- `deriv`: Computes a large symbolic derivative of x^x.
- `lisp`: A lexer, parser, and interpreter for a small subset of Scheme running another interpreter for Scheme written in Scheme running a factorial function.
- `nqueens_functional`: Creates a list of solutions to the standard n-queens problem, and prints the number of solutions found. The solution list includes shared sub-solutions. The benchmark recursively builds a tree of all possible solutions by extending partial solutions.
- `nqueens_iterative`: An iterative-flavored solution to n-queens that runs about 4x faster than the recursive version.
- `parse_json`: A parser for JSON implemented with a custom parser combinator library.
- `primes_sieve`: The Sieve of Eratosthenes: counts the number of primes below a number.
- `rbtree`: An implementation of a red-black tree. The benchmark inserts many elements and then does a single fold over the tree.
- `rbtree_ck`: The same red-black tree implementation, but with a snapshot saved in a list every five insertions, thus with more sharing.
- `unify`: A syntactic unification algorithm operating on trees of S-expression-like terms.
- `text_stats`: Counts the occurrences of a given word in a list of words, with a simple fold.
- `words_trie`: A trie data structure, used for counting the number of occurrences of words.

All benchmarks were run on an Intel i5-13600K with 16 GiB RAM. The raw data is available in our artifact as well as data from an Intel Xeon Platinum 8124M CPU with 192 GiB of RAM. For each benchmark, we compare a version with our Perceus baseline to a version with borrow inference. We measure both the number of **dup** instructions called dynamically during a run, which does not depend on the hardware, and separately (with **dup** instrumentation off) the absolute runtime of each version. The results are shown in Figure 9. The number of **drop** instructions is always a constant number greater than the number of **dup** instructions, so we omit the results. We also compare a version of each benchmark with BDWGC to a version with borrow inference, shown in Figure 10. A conservative collector is not as naive a choice as one might think because Morphic only uses heap allocations for arrays and recursive instances of recursive types (scalars, structs, enums, and even closure environments are stack allocated), as demonstrated in Figure 11 where we compare Morphic's performance with BDWGC to OCaml and MLTon.

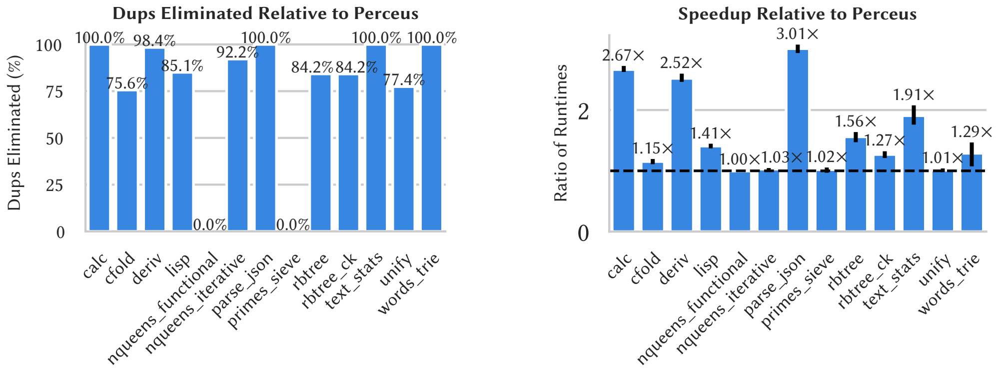

Fig. 9. Morphic with Perceus-style refcounting vs. borrow inference–style refcounting. At their maximum, error bars in the right plot represent a ratio of the slowest run under Perceus-style refcounting to the fastest run under borrow inference–style refcounting, and vice-versa at their minimum. All benchmarks were run with COW arrays; results with persistent arrays were similar and are given in the artifact.

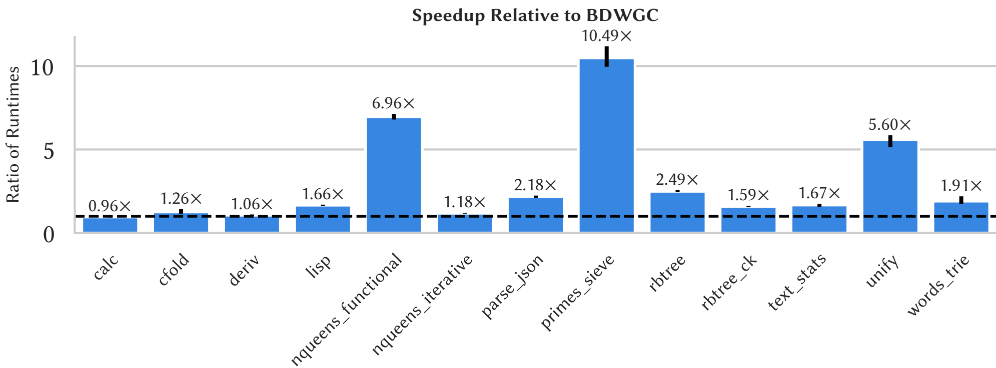

Fig. 10. Morphic with a conservative garbage collector vs. borrow inference–style refcounting. Error bars as in Figure 9. All benchmarks, including borrow inference versions, were run with persistent arrays.

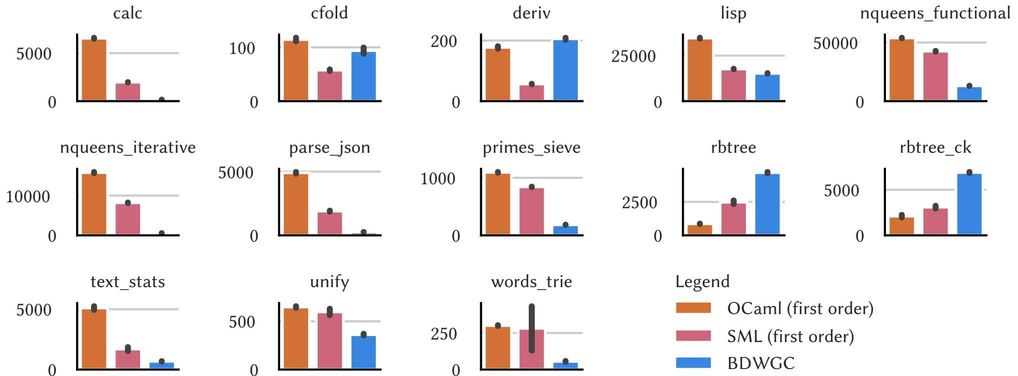

Fig. 11. OCaml 5.3.0 vs. MLTon 20241230 vs. Morphic's performance with conservative collector BDWGC. All benchmarks were translated from Morphic after defunctionalization to OCaml/SML source code. All y-axes are in milliseconds; error bars are standard deviation.

7.1 Benchmark Discussion

Overall, Morphic saved 75-100% of reference counts over the Perceus baseline on 11 of the benchmarks, and matched the baseline (saved 0%) on the other two. Dividing the absolute speedup by the number of reference count operations yields 1-3 nanoseconds saved per dup/drop pair, except for one outlier, `unify` at 0.1 nanoseconds, which is described in more detail below.

Comparing borrow inference to BDWGC shows that reference counting with borrow inference consistently outperforms the conservative garbage collector, with especially large improvements on the three benchmarks with large arrays, `unify`, `primes_sieve`, `nqueens_functional`, due to the RC-1 optimization allowing in-place mutation, rather than slow allocation-heavy persistent array operations, along with the associated garbage collection overhead.

Looking at the OCaml/SML chart, Morphic with a conservative garbage collector is faster than both OCaml and MLTon on 9 of the 13 benchmarks. Most of the performance difference seems to be due to differences in numbers of allocations. MLTon supports unboxed 64-bit integers, but OCaml does not, explaining why MLTon seems to mostly outperform OCaml here. More idiomatic would be OCaml's 63-bit integers, which we did not use for correctness reasons. Morphic's performance increases seem to be due to boxing significantly less than either OCaml or MLTon, with tuples and algebraic data types also being unboxed. The benchmarks where OCaml/MLTon outperform Morphic (`deriv`, `rbtree`, `rbtree-ck`) are the ones heavily manipulating recursive data types. We suspect the difference here is a combination of Morphic's use of a conservative garbage collector causing memory fragmentation, as well as optimizations allowing stack allocation of recursive types not currently implemented in Morphic.

The benchmarks that Morphic's implementation improves the most relative to Perceus are `calc` and `parse_json`, which share the same parser combinator library. There are no dups in the optimized version, compared to millions of dups in the baseline, leading to a 2.5-3× speedup. A parser in the combinator library is represented as a function (Int, Array Byte) → Option (Int, a), and when any combinator is called, there will be a dup/drop pair in the baseline. The number of reference counts saved is consistent with the observation that many combinators may be called for each step forward in the parse state. Reducing the runtime 2.5-3× suggests that the original program was spending over 60% of its time incrementing and decrementing reference-counts!

The `text_stats` benchmark does one fold over an array of strings in its critical section, checking for string equality. With borrow inference, the critical loop goes from "one RC increment, one string equality comparison, and one RC decrement" to "one string equality comparison," saving about 40% of the original runtime.

For the data structure and algorithms benchmarks, `nqueens*`, `words_trie`, `cfold`, `rbtree*`, and `deriv`, empirically borrow inference appears to do better the more read-heavy the workload is. This makes sense, as if a write is already happening, it does not add much overhead to write to the reference count field too, but if there is only reading, then the pair of RC increment and decrement can be costly. The code for `deriv` performs constant scanning through the expression to spot algebraic simplification opportunities. The `rbtree` benchmarks mostly consist of writing into a red-black tree, but rebalancing requires a lot of reading. The relative costs of the reference counts is slightly lowered with some sharing of the red-black tree. The `nqueens` benchmarks have lots of sharing, lots of branching, and not much reading. The `nqueens_functional` benchmark's only array operations are mutating and there are no other non-array RC'd types, so there is no opportunity to take advantage of borrowing. The similar `nqueens_iterative` did have gets and other RC'd types.

`unify` did not speed up significantly, but this is an outlier, as its algorithm should be sensitive to reference count reductions. Investigating further, there are about 6.4% of RC-1 array mutations that fall back to a copy under borrow inference, but succeed under the baseline. Borrowing past a mutation point may inhibit RC-1 optimizations. This could be addressed in future work by taking into account mutation points in the flow analysis. The `unify` program is written with a state monad, and any borrows of the state being extended longer than necessary will cause a copy.

`primes_sieve` has only one RC'd type, an array of integers which gets mutated in place. So, there are no dups in the recorded part of the program, and so none to elide. However, the overall program is still 7% faster. The Perceus and borrow inference versions of the program end up with same dups/drops, but the LLVM we generate is not precisely the same because functions are specialized with respect to modes before we determine dups/drops. The borrow inference version calls different functions with identical implementations where the Perceus version calls a single function. We view the 7% difference here as a signal about benchmark noise that should be kept in mind when interpreting the figures.

The `lisp` benchmark is our biggest and most realistic example at about 600 lines. It implements core Lisp functionality, such as tokenization, parsing of S-expressions, and evaluation. An immutable `HashMap`, implemented with the Array primitive, is used for the lexical environment. The RC-1 optimization implemented for arrays allows for `HashMap` operations to also mutate in place when possible, resulting in reasonably fast performance for the interpreter. The `lisp` benchmark spends less time doing reference counting as a proportion of total runtime compared to the data structure benchmarks.

Finally, we note that, while borrowing can increase the interval where data is live and thereby increase a program's memory consumption (e.g., a list must be live while a borrow of any of its elements is live), when we measured the maximum memory footprints of our benchmarks, only `unify` saw an increase, and that increase was only 2.5%.

8 Related Work

Rust credits Tofte and Talpin [25] and Jim et al. [14] as inspirations for its type system [24, Appendix E]. Tofte and Talpin [25] introduce a static memory management discipline whereby values are stored in *regions*, stacks of unbounded size that are deallocated all at once at the end of their lexical scopes, and give rules for region inference. Jim et al. [14] introduce Cyclone, a dialect of C which

uses regions for safe manual memory management. Unlike the Tofte and Talpin [25] work, users are required to provide region annotations to be checked by the compiler.

There have been a number of recent attempts to formalize the statics and dynamics of Rust. Patina, early work by Reed [21], modeled the safe fragment of Rust's type system as it existed in 2015 (before non-lexical lifetimes) and provided partial proofs of progress and preservation. RustBelt [16] mechanized λ_{Rust}, a continuation-passing style language resembling Rust's mid-level intermediate representation, and formalized how **unsafe** code can be encapsulated by safe interfaces. Oxide [29] formalized a system more closely resembling user-facing Rust. Featherweight Rust [20] presented a simplified view of Rust's type system inspired by Featherweight Java [13]. Stacked Borrows [15] and its follow-up Tree Borrows [27] gave operational semantics for **unsafe** Rust, circumscribing permissible compiler optimizations. Bao et al. [4], inspired by the same line of separation logic work that undergirds RustBelt, took a very different approach than Rust to ownership semantics, selectively restricting sharing instead of reintroducing it.

Like Rust, we distinguish between owned values and borrowed values and we annotate borrows with lifetimes that ensure that they are not used outside the interval over which they are known to be valid. Also like Rust, these lifetimes are non-lexical in the sense that they can describe program intervals that begin and end in the middle of basic blocks, and functions which accept and return borrows can and must be parameterized by lifetime variables, which the caller instantiates at the function call site. Unlike Rust, we have no notion of exclusive references. The problem of inference is ill-posed in a context where the target language has exclusive borrows, but the source language does not. Whereas owners and shared borrows are interchangeable, except insofar as lifetimes are concerned, exclusive references enable fundamentally different user programs. Also unlike Rust, for reasons of inference, we attach modes directly to **rc** values instead of supporting borrows as first-class types. In Rust, the user must specify, for each variable, whether or not it is a borrow, unless this information is available by simple unification.

Another line of recent work optimizes reference counted pure functional programs. Ullrich and de Moura [26] introduce two optimizations: they find opportunities to reuse allocations (a generalized RC-1 check), and they heuristically annotate some function arguments as borrowed, avoiding the need for a reference count increment when calling the function. However, they make no attempt to track lifetimes which means that every other language construct must reference count in the naive way, all the way down to tuple expressions. Perceus [22] implements reference counting by inferring into an ownership type system without borrowing. This makes Perceus's reference counting *precise/garbage free*, i.e., references are dropped as soon as possible, enabling more reuse opportunities in the style of Ullrich and de Moura [26]. Lorenzen [18] builds on Ullrich and de Moura [26] and Perceus, performing Ullrich and de Moura [26] style borrows while maintaining the guarantee that borrowing does not increase peak memory usage by more than a constant amount times the stack depth, a weakened garbage freeness property.

9 Conclusion

In this paper we presented borrow inference, a technique for inferring borrows-with-lifetimes type signatures for functional programs, and completing programs that do not admit such signatures with reference count increment operations. We developed a linear language, $L^{\&}$, supporting borrowing with lifetimes as the target of our inference. We formulated a soundness theorem characterizing the memory safety properties of all well-typed programs in $L^{\&}$. Our implementation of borrow inference yields significant speedups and reductions in reference-counting operations relative to our baseline, the current state-of-the-art in reference-counted functional programming.

Data-Availability Statement

The source code of the Morphic compiler, the source code of our benchmarks, and the benchmark data we collected have been packaged as a Docker image available via Zenodo [8].

Acknowledgments

We thank Wilson Berkow who made important contributions to Morphic's early development. This work was supported by the National Science Foundation under Grant No. 2216964.

References

[1] Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. 2022. Convergence of Datalog over (Pre-) Semirings. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Philadelphia, PA, USA) *(PODS '22)*. Association for Computing Machinery, New York, NY, USA, 105–117. doi:10.1145/3517804.3524140

[2] Phil Bagwell. 2001. *Ideal Hash Trees.* Technical Report EPFL-REPORT-64398. École Polytechnique Fédérale de Lausanne. https://infoscience.epfl.ch/record/64398

[3] Phil Bagwell and Tiark Rompf. 2011. *RRB-Trees: Efficient Immutable Vectors.* Technical Report EPFL-REPORT-169879. École Polytechnique Fédérale de Lausanne. https://infoscience.epfl.ch/record/169879

[4] Yuyan Bao, Guannan Wei, Oliver Bračevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. 2021. Reachability types: tracking aliasing and separation in higher-order functional programs. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 139 (Oct. 2021), 32 pages. doi:10.1145/3485516

[5] Hans-Juergen Boehm and Mark Weiser. 1988. Garbage collection in an uncooperative environment. *Software: Practice and Experience* 18, 9 (1988), 807–820.

[6] William Brandon, Benjamin Driscoll, Frank Dai, and Wilson Berkow. 2020. Morphic Research Language. https://morphic-lang.org [Online; accessed 29-July-2025].

[7] William Brandon, Benjamin Driscoll, Frank Dai, Wilson Berkow, and Mae Milano. 2023. Better Defunctionalization through Lambda Set Specialization. *Proc. ACM Program. Lang.* 7, PLDI, Article 146 (jun 2023), 24 pages. doi:10.1145/3591260

[8] William Brandon, Benjamin Driscoll, Frank Dai, Jonathan Ragan-Kelley, Mae Milano, and Alex Aiken. 2025. *Fully-Automatic Type Inference for Borrows with Lifetimes.* doi:10.5281/zenodo.16605090

[9] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA) *(PLDI '93)*. Association for Computing Machinery, New York, NY, USA, 237–247. doi:10.1145/155090.155113

[10] Robert Harper. 2016. *Practical Foundations for Programming Languages* (2nd ed.). Cambridge University Press, Cambridge.

[11] Rich Hickey. 2006. Clojure. https://clojure.org [Online; accessed 29-July-2025].

[12] Paul Hudak and Adrienne Bloss. 1985. The aggregate update problem in functional programming systems. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '85)*. Association for Computing Machinery, New York, NY, USA, 300–314. doi:10.1145/318593.318660

[13] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (may 2001), 396–450. doi:10.1145/503502.503505

[14] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATEC '02)*. USENIX Association, USA, 275–288.

[15] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked Borrows: An Aliasing Model for Rust. *Proc. ACM Program. Lang.* 4, POPL, Article 41 (dec 2019), 32 pages. doi:10.1145/3371109

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

[17] Steve Klabnik and Carol Nichols. 2019. *The Rust Programming Language.* No Starch Press, San Francisco, CA.

[18] Anton Felix Lorenzen. 2021. *Optimizing Reference Counting with Borrowing.* Master's thesis. Rhenish Friedrich Wilhelm University of Bonn.

[19] Ivan Maidanski. 2025. *Boehm-Demers-Weiser Garbage Collector.* https://github.com/bdwgc/bdwgc

[20] David J. Pearce. 2021. A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust. *ACM Trans. Program. Lang. Syst.* 43, 1, Article 3 (apr 2021), 73 pages. doi:10.1145/3443420

[21] Eric Reed. 2015. *Patina: A Formalization of the Rust Programming Language.* Technical Report UW-CSE-15-03-02. University of Washington. https://dada.cs.washington.edu/research/tr/2015/03/UW-CSE-15-03-02.pdf

[22] Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. Perceus: garbage free reference counting with reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 96–111. doi:10.1145/3453483.3454032

[23] Alfred Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.* 5 (1955), 285–309.

[24] The Rust Project Developers. 2018. *Rust Compiler Development Guide (rustc-dev-guide).* https://github.com/rust-lang/rustc-dev-guide/tree/06424769ca1ec2dee8d80a4be0ea25d563b4f86d

[25] Mads Tofte and Jean-Pierre Talpin. 1997. Region-based memory management. *Information and computation* 132, 2 (1997), 109–176.

[26] Sebastian Ullrich and Leonardo de Moura. 2021. Counting immutable beans: reference counting optimized for purely functional programming. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages* (Singapore, Singapore) *(IFL '19)*. Association for Computing Machinery, New York, NY, USA, Article 3, 12 pages. doi:10.1145/3412932.3412935

[27] Neven Villani, Johannes Hostert, Derek Dreyer, and Ralf Jung. 2025. Tree Borrows. *Proc. ACM Program. Lang.* 9, PLDI, Article 188 (June 2025), 24 pages. doi:10.1145/3735592

[28] Keith Wansbrough and Simon Peyton Jones. 1999. Once upon a polymorphic type. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 15–28. doi:10.1145/292540.292545

[29] Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. 2021. Oxide: The Essence of Rust. arXiv:1903.00982 [cs.PL]

Received 2025-10-09; accepted 2026-02-17