A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust

DAVID J. PEARCE, Victoria University of Wellington

Rust is a relatively new programming language which has gained significant traction since its v1.0 release in 2015. Rust aims to be a systems language that competes with C/C++. A claimed advantage of Rust is a strong focus on memory safety without garbage collection. This is primarily achieved through two concepts, namely *reference lifetimes* and *borrowing*. Both of these are well known ideas stemming from the literature on *region-based memory management* and *linearity / uniqueness*. Rust brings both of these ideas together to form a coherent programming model. Furthermore, Rust has a strong focus on stack-allocated data and, like C/C++ but unlike Java, permits references to local variables.

Type checking in Rust can be viewed as a two-phase process: firstly, a traditional type checker operates in a flow-insensitive fashion; secondly, a *borrow checker* enforces an ownership invariant using a flow-sensitive analysis. In this paper, we present a lightweight formalism which captures these two phases using a flow-sensitive type system that enforces "*type and borrow safety*". In particular, programs which are type and borrow safe will not attempt to dereference dangling pointers. Our calculus core captures many aspects of Rust, including copy- and move-semantics, mutable borrowing, reborrowing, partial moves, and lifetimes. In particular, it remains sufficiently lightweight to be easily digested and understood and, we argue, still captures the salient aspects of reference lifetimes and borrowing. Furthermore, extensions to the core can easily add more complex features (e.g. control-flow, tuples, method invocation, etc). We provide a soundness proof to verify our key claims of the calculus. We also provide a reference implementation in Java with which we have model checked our calculus using over 500 billion input programs. We have also fuzz tested the Rust compiler using our calculus against 2 billion programs and, to date, found one confirmed compiler bug and several other possible issues.

CCS Concepts: • **Software and its engineering** → **Memory management**; **Formal language definitions**; *Imperative languages*; • **Theory of computation** → *Type theory*.

Additional Key Words and Phrases: Rust, Ownership, Type Theory, Model Checking

ACM Reference Format:
David J. Pearce. 2020. A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust. *ACM Trans. Program. Lang. Syst.* 1, 1 (June 2020), 72 pages. https://doi.org/???/???

1 INTRODUCTION

Dangling pointers arise when dynamically allocated memory is freed or when stack-allocated data goes out of scope. These are common problems in imperative languages like C/C++ which, unfortunately, cause many reliability and security problems [26, 47, 116, 119]. Various solutions are known, of which garbage collection is perhaps the most widely adopted [64]. For systems languages like C/C++, garbage collection is considered prohibitively expensive and, hence, manual memory allocation prevails. Alternatives have been explored extensively [33, 54, 68, 107, 124, 127, 129]. Of

Author's address: David J. Pearce, david.pearce@ecs.vuw.ac.nz, Victoria University of Wellington, Wellington, New Zealand.

0164-0925/2020/6-ART $15.00
https://doi.org/???/???

these region-based memory management is perhaps the most promising, having been pioneered for Standard ML [124, 127] and subsequent systems, such as Cyclone [49, 53, 54, 61, 67, 123], X10 [31] and the Real-Time Java Specification [107]. Another common approach is to employ linear [51, 130] or unique [23, 38] type systems which ensure that, when a unique/linear variable goes out of scope, its storage can be reclaimed. This is a well-trodden idea in C++ where smart pointers, such as `auto_ptr` and `unique_ptr`, have been used for over twenty years [92]. Indeed, the advent of C++11 has further improved this situation by providing proper support for move constructors and *rvalue references* [42, 122].

Rust is a relatively new systems language from Mozilla which provides safe memory management without garbage collection and, in many ways, provides a logical extension of smart pointers in C++. Rust has gained significant traction already and is used in production by companies such as Dropbox, Yelp, Xero and Chef. Rust aspires to replace C/C++ by bringing safe, zero-cost memory management to systems programming through *reference lifetimes* and *borrowing*. The former is roughly analogous to the concept of a region, whilst the latter comes from work on linearity/uniqueness. Taking heavy influence from the earlier work on Cyclone [49, 61, 123], Rust brings ideas from both region-based memory management and linearity/uniqueness together to form a coherent programming model. Other important benefits have also arisen from this, including the ability to protect against iterator invalidation and certain forms of data race [8].

At this relatively early stage in the evolution of Rust, there is already a burgeoning literature exploring different aspects of the language [4, 6, 10, 41, 66, 69, 70, 80, 109, 110]. The recent work of Jung *et al.* is an excellent example which provides a comprehensive, machine-checked formalisation for a realistic subset of Rust [70]. This includes various notions of concurrency and extends to libraries using `unsafe` features by identifying *library-specific verification conditions* which must be satisfied to ensure overall safety. However, concessions were understandably necessary given the enormity of this formalisation task (which, in fact, amounts to roughly 17.5KLOC of Coq). For example, the system presented does not resemble the surface syntax of Rust but, rather, is more akin to the *Mid-level Intermediate Representation (MIR)* used within the Rust compiler. As such, we believe there is a need for formalisations of the language which provide complementary benefits (e.g. ease of experimenting with new features, establishing claims regarding specific features, communicating key ideas behind its semantics, fuzz testing the compiler, etc).

We present a lightweight formalism of Rust, called FR, which captures both *type checking* (a flow-insensitive activity) and *borrow checking* (a flow-sensitive activity). The goal was to make FR as accessible as possible and easy to experiment with. In doing this, we take inspiration from the development of Featherweight Java (FJ) [63]:

> "*Our main goal in designing FJ was to make a proof of type soundness ("well-typed programs do not get stuck") as concise as possible, while still capturing the essence of the soundness argument for the full Java language ... Any language feature that made the soundness proof longer without making it significantly different was a candidate for omission*"

As such FR provides a lightweight calculus that is effectively a true subset of the Rust language. Our formalism does not attempt to model all aspects of the Rust language, and is intended to provide insight into the language and serve as the basis for future work (e.g. formalising simple language extensions, etc). Our formalism attempts to characterise the essential problems of memory management whilst ignoring other aspects (such as those relating to concurrency). More specifically, it provides: protection against dangling references in the presence of deallocation; and, enforcement of a strong ownership invariant subject to controlled, temporary breakages (i.e. borrowing). The accompanying proof of the key *type and borrow safety* theorem is, likewise, lightweight in nature and easy to digest. In stripping Rust down to a minimal core, we find the

resulting calculus extremely helpful in understanding the deeper aspects of lifetimes and borrowing.

1.1 Contributions

The main contributions of this paper are:

- **(Calculus)** We present a lightweight calculus for describing (what is effectively) a subset of Rust. This includes the salient aspects of lifetimes and borrowing, including copy- and move-semantics, mutable borrowing, reborrowing, partial moves, and lifetimes. A flow-sensitive type system encodes the primary functions of type and borrow checking.
- **(Proof)** We provide a soundness proof alongside the calculus to verify our claims that a *type and borrow safe* program will not get stuck and borrows safely. This implies (amongst other things) that programs cannot access dangling references, and that they adhere to the rules of lifetimes and borrowing.
- **(Implementation)** We provide a reference implementation in Java of our calculus which retains a strong connection with the rules presented in this paper. In particular, one can easily check by hand that the implementation matches the presented rules. Using this, we have model checked our calculus using over 500B input programs. In addition we have used our calculus to fuzz test the Rust compiler using over 2B input programs and compare results and, to date, found one confirmed compiler bug and identified several other possible issues.
- **(Extension)** We explore several simple extensions to our calculus which model missing parts of the Rust language. This includes a detailed examination of adding conditional statements to illustrate control flow, and tuples to illustrate compound data types. We also sketch how method invocation can be added. These extensions highlight both the ease with which our formalisation can be extended, and the inherent flexibility of its core.

We imagine many uses for our calculus going forward. In the first instance, it provides a simple and easy to digest formalisation of the salient aspects of type and borrow checking in Rust. This provides an ideal platform on which to experiment with new language features and/or to design similarly-minded languages. But there are many other use cases. For example, since our formalisation is effectively a subset of Rust, it can be used as an oracle for fuzz testing the Rust compiler (as illustrated in §5.3).

2 OVERVIEW OF RUST

Rust is a multi-paradigm programming language that claims to be *"a systems programming language that runs blazingly fast, prevents segmentation faults, and guarantees thread safety"* [113]. Rust was initially developed by Graydon Hoare whilst at Mozilla. Since then, further development has been funded by Mozilla with an aim to replacing some — or all — of the C++ code making up the Firefox web browser.[1] Nicholas D. Matsakis is credited for introducing lifetimes and borrowing, presumably based on his earlier work [86–89]. We now give an overview, whilst paying particular attention to reference lifetimes and borrowing. A more detailed introduction can be found elsewhere [114, 115].

2.1 (Im)mutability

Rust's syntax was heavily influenced by both functional and imperative languages [4]. Following functional languages, local variables correspond to immutable *variable bindings*. By adding the

[1] At this time, some Rust code is present in released versions of Firefox and Thunderbird. Furthermore, an experimental rendering engine called Servo is under development to gain experience with the Rust language and to guide its future development [4].

modifier `mut`, variables can also be treated as mutable like those of imperative languages. The following illustrates a simple program accepted by the Rust compiler:

```
fn f() -> i32 {
  let x = 1;
  return x+1;
}
```

Here, `i32` denotes a 32-bit signed integer (likewise `u8` for 8-bit unsigned integers, `i16` for 16-bit signed integers, etc). The type of `x` is inferred from the assigned expression and, since `x` is an *immutable binding*, we cannot reassign it. The following illustrates such an attempt:

```
fn f() -> i32 {
  let x = 1;
  x = x + 1;
  return x;
}
```

Attempting to compile the above program gives an error "`cannot assign twice to immutable variable ‘x‘`". Although variables are immutable by default, they can be declared as mutable using the `mut` keyword as follows:

```
fn f() -> i32 {
  let mut x = 1;
  x = x + 1;
  return x;
}
```

The above is accepted by the Rust compiler and, in essence, the mutable binding `x` is comparable to a local variable in languages like C/C++. The Rust book gives the following justification here [114]:

> *"There is no single reason that bindings are immutable by default, but we can think about it through one of Rust's primary focuses: safety. If you forget to say* **`mut`***, the compiler will catch it, and let you know that you have mutated something you may not have intended to mutate. If bindings were mutable by default, the compiler would not be able to tell you this. If you did intend mutation, then the solution is quite easy: add* **`mut`***."*

Whilst we are not specifically concerned with the choice of immutable-by-default here, as we will see later, an explicit notion of *immutable references* versus *mutable references* is critical. Finally, Rust follows many functional languages in allowing variable bindings to be *rebound*. For example, the following is perfectly acceptable:

```
fn f() -> i32 {
  let x = 1;
  let x = x + 1;
  return x;
}
```

Here, the first declaration of `x` is said to be *shadowed* by the second. Note that we are declaring distinct bindings here, and the second declaration should not be viewed as simply an assignment to variable `x`. Whilst shadowing may seem a potentially hazardous feature, it is useful as it effectively allows a variable to be given a different type.

2.2 Ownership

The concept of *ownership* is a central facet of the Rust language and draws heavily from the literature on linearity and uniqueness. The Rust book makes the following comments [114]:

> "*Variable bindings have a property in Rust: they 'have ownership' of what they're bound to. This means that when a binding goes out of scope, Rust will free the bound resources.*"

A variable is bound to a chunk of allocated storage (of required size) which we refer to as an *abstract location*, ℓ. Such locations must have a statically-known size and, for now, can be regarded as always stack allocated. Rust enforces an *ownership invariant* where a variable is said to "own" the value it contains such that no two variables can own the same value (though, as we'll see later, this can be relaxed through *borrowing*).

The concept of ownership has important implications for the permitted behaviour of programs in Rust. Consider the following simple example:

```
fn dup(x: Vec<i32>) -> (Vec<i32>,Vec<i32>) {
  let y = x;
  return (x, y);
}
```

To understand this we must view the declaration, `let y = x`, in terms of ownership. Unlike for many languages (e.g. Java, C/C++, etc), this does not default to *copying* the value from `x` to `y` as might be expected (i.e. as this would break ownership). Instead, such an assignment by default *moves* the value from `x` to `y` and, thus, leaves `x` subsequently *undefined*. And, indeed, the Rust compiler complains that "`value used here after move`" regarding the use of `x` in the return statement. The key here is that, after the move, variable `x` is considered no longer usable and the borrow checker enforces this. Observe that rewriting the above example to the following equivalent form does not prevent the error:

```
fn dup(x: Vec<i32>) -> (Vec<i32>,Vec<i32>) {
  return (x, x);
}
```

Here the Rust compiler observes the values returned may, at the point of invocation, be assigned to different variables and, hence, rejects the above program. For example, allowing the assignment `let (a,b) = dup(vec![1,2,3]);` would violate the ownership invariant (where the `vec![1,2,3]` macro constructs an instance of `Vec<i32>`). To work around this, we must create a new value which can be assigned to one of the variables:

```
fn dup(x: Vec<i32>) -> (Vec<i32>,Vec<i32>) {
  let y = x.clone();
  return (x, y);
}
```

The above is now accepted by the Rust compiler. The `clone()` method is provided by types implementing the `Clone` trait and, generally speaking, performs a deep clone up to borrowed references (i.e. roughly approximating a *sheep clone* [82, 98]).

2.2.1 Boxes. The ability to dynamically allocate memory without requiring manual deallocation or garbage collection is a critical feature of Rust. For example, `std::Vec<T>` provides a resizeable array type (roughly like `ArrayList` in Java). Since the array portion of this type has arbitrary size, it cannot be stored on the stack and, instead, must be dynamically allocated. To the programmer,

however, this is transparent as it otherwise behaves like a regular type (as illustrated in the examples above). For the purposes of this paper, however, we restrict ourselves to the `Box<T>` type. This provides a unique pointer to a given data type allocated on the heap. The following illustrates a simple example accepted by the Rust compiler:

```
fn alloc(n : i32) -> Box<i32> { return Box::new(n); }
```

The key is that, whilst boxes require explicit allocation by the programmer, their deallocation is implicit. In particular, when the location owning a box type goes out of scope, its dynamically allocated memory is automatically released.

2.2.2 Move vs Copy Semantics. A fundamental and, at first, challenging aspect of Rust is the distinction made between data types supporting *move semantics* and those which additionally support *copy semantics*. To the inexperienced developer, it can seem mysterious as to why some types only exhibit move semantics whilst others are more flexible. The fundamental distinction is whether or not a type implements the `Copy` trait, which characterises those types that can be duplicated simply through a bitwise copy. As expected, primitive types (such as `i32`) implicitly implement the `Copy` trait and, hence, exhibit copy semantics. This means, for example, that the following program compiles without problem:

```
fn f(x: i32) -> i32 {  let y = x; return x + y; }
```

Of course, some types are or contain references to heap allocated memory (e.g. `Vec<T>` or `Box<T>`). They cannot be copied using a simple bitwise copy as this would prevent their safe deallocation and, instead, exhibit move semantics. User-defined types (e.g. `struct`s) employ move semantics by default and, when it makes sense to do so, the programmer can explicitly override this by implementing the `Copy` trait.

Finally, the `Copy` trait is known to the borrow checker so that it can treat types with move semantics differently from those with copy semantics. In our calculus, we include types which exhibit both behaviour to ensure a balanced representation of Rust.

2.3 Borrowing

An important concept is that of *borrowing* which enables controlled breakages of the ownership invariant [23, 38, 51, 52, 130, 131]. For example, borrowing is the essential mechanism for accessing shared resources where multiple writers to the same shared resource are prohibited. Likewise, borrowing prevents unsafe concurrent modification of collections in Rust. The following example, which compiles without problem, illustrates borrowing in Rust:

```
fn is_nat(x : &i32) -> bool {
  if *x >= 0 { return true; } else { return false; }
}
```

Here, `&i32` indicates an *immutable borrowed reference* to a value of type `i32`. In contrast, `&mut i32` would indicate a *mutable* borrowed reference. A borrowed reference refers to a location defined elsewhere and, thus, clearly breaks the ownership invariant. Such a breakage should (generally speaking) be temporary in nature. A given location may have multiple immutable borrowed references when there are no mutable borrowed references (i.e. multiple readers, no writers); otherwise, at most one (mutable) borrowed reference is permitted (i.e. no readers, exactly one writer). In some sense, we can think of borrowed references as being a restricted form of pointers/references in languages like C/C++/Java.

The following example, which compiles without problem, illustrates the use of borrowing to effect a temporary breakage of the ownership invariant:

```
fn f() -> (i32,bool) {
  let x = 0;
  let y = is_nat(&x);
  return (x,y);
}
```

Contrasting this example with `dup()` from §2.2 we see that, having borrowed via `&x`, we can subsequently use `x` after the invocation as, once the borrow is completed, `x` is restored as the owner of its location.

2.3.1 Borrow Checking. Rust permits multiple immutable borrowed references to coexist for the same location and, during this time, any existing immutable binding may continue to be used. The following illustrates a simple example accepted by the Rust compiler:

```
fn f() -> i32 {
  let x = 1;
  let y = &x;
  let z = &x;
  return x + *y + *z;
}
```

Here, an immutable binding, `x`, and two immutable borrowed references, `y` and `z`, happily coexist. In contrast, only one borrowed reference of any kind can exist for a given location if that reference is mutable. Thus, immutable and mutable borrowed references cannot coexist for the same resource. Consider the following `swap()` function which compiles without problem:

```
fn swap(x : &mut i32, y : &mut i32) { let z = *x; *x = *y; *y = z; }
```

The type modifier `&mut` indicates that `x` and `y` are *mutable borrowed references*. The function `swap()` simply accepts two mutable references and swaps their contents. Since Rust prohibits multiple mutable references to the same location, we know `x != y`. Hence, the following use of `swap()` is rejected by the Rust compiler:

```
  let mut x = 1;
  swap(&mut x,&mut x);
```

Note, `&mut x` indicates that we are making a mutable borrow of variable `x` and, for this to be permitted, it follows that `x` must itself be mutable. The above program yields the following error:

```
error[E0499]: cannot borrow `x` as mutable more than once at a time
  |
9 |     swap(&mut x,&mut x);
  |              -      ^- first borrow ends here
  |              |      |
  |              |      second mutable borrow occurs here
  |              first mutable borrow occurs here
```

As an aside, error messages regarding borrowing seem surprisingly good and this is perhaps an attempt to mitigate the perceived difficulty in understanding and working with the borrow checker.

2.3.2 Reborrowing. A common scenario arising in Rust is the need to pass a mutable borrow into a method, whilst retaining it for use after the invocation. The following illustrates:

```
fn put(q : &mut i32, v : i32) { *q = v; }

    ...
    let mut x = 0;
    let mut p = &mut x;
    put(p, 1);
    *p = 2;
```

Whilst this is an artifical example, it serves to illustrate the problem. Since `p` is a mutable borrow it has move semantics and, hence, passing it into `put()` should cause a move rendering `p` subsequently undefined. One solution is for `put()` to "give back" (i.e. return) the mutable borrow so it can be reassigned to `p`. However, since threading mutable borrows through methods in this fashion is rather tedious, Rust offers an alternative solution. In fact, the above program compiles without problem in Rust because, roughly speaking, an implicit coercion is applied. More specifically, the term `put(p, 1);` can be viewed as `put(&mut *p, 1);` where `&mut *p` is referred to as a (mutable) *reborrow*.

One can think of a reborrow `&mut *p` as taking a mutable borrow to the location referred to by `p`. In such case, `p` must itself have ownership over the location to which it refers. Then, for the life of the reborrow, the original borrow cannot be used. Once the reborrow is over, the original borrow can again be reused as before. Furthermore, one can reborrow a mutable borrow as an immutable borrow (which temporarily changes the type of the original borrow to immutable).

2.3.3 Flow Sensitivity. Another interesting aspect of the borrow checker is its *flow sensitive* nature. Specifically, the act of borrowing can have flow sensitive effects and this distinguishes it from regular type checking (i.e. a flow *insensitive* activity). As an example, making an immutable borrow to a mutable binding has the effect of *freezing* that binding for the duration:

```
fn f() -> i32 {
  let mut x = 1;
  let y = &x;
  x = x + 1;
  return x + *y;
}
```

Compiling this yields the error "`cannot assign to ‘x‘ because it is borrowed`" as we are attempting to mutate `x` within the scope of the borrow for `y`. In essence, when borrowing `x` we are implicitly changing its type to immutable for the life of the borrow. This ensures the immutable borrow does indeed refer to an immutable value. Without the ability to freeze mutable bindings, one could never immutably borrow them. To further illustrate, consider the effect of borrowing *part* of a variable binding (where `Point` has type `struct {x:i32, y:i32}`):

```
fn f() -> i32 {
  let mut p = Point{x:1,y:2};
  let br = &mut p.y;
  return p.x + *br;
}
```

The above compiles without problem, thus indicating the borrow checker is quite sophisticated when reasoning about borrows. Since we have mutably borrowed part of variable `p` above, we

might reasonably expect `p` was now frozen in its entirety. However, the borrow checker is smart enough to freeze only that which is borrowed (i.e. `p.y`), leaving us free to use other parts (i.e. `p.x`). Finally, although the borrow checker can reason precisely about `struct`s, it remains conservative when reasoning about arrays:

```
fn f() -> i32 {
  let mut p = [1,2,3];
  let br = &mut p[1];
  return p[0] + *br;
}
```

We can reason that, although `p[1]` is mutably borrowed, `p[0]` is not so the above program is safe. Rust's borrow checker does not reason so precisely and, instead, regards the borrow of `p[1]` as a borrow to the whole of `p`. Thus, the above program is rejected by the Rust compiler.

2.3.4 Moving Out. An interesting and sometimes unexpected aspect of Rust is that you cannot move an item out of a borrow, which is referred to as "moving out". The following illustrates:

```
struct Item { value : i32 }

fn main() {
  let x = Item{value:2};
  let y = &x;
  let z = *y;
}
```

Compiling this produces the error "`cannot move out of borrowed content`" which, while strange, is an entirely rational response from the borrow checker. Recall `structs` have move semantics by default. Thus, allowing the above would result in both `x` and `z` owning the same location and, hence, both would be responsible for deallocating it. In contrast, by changing the last line as follows, it will compile:

```
  ...
  let z = y.value;
```

The reason for this is simply that `y.value` has type `i32` which implicitly implements `Copy`. Hence, it's value is copied out of the borrow rather than being moved.

2.4 Reference Lifetimes

We tacitly referred above to the *life* of a borrowed reference without clarifying this. In fact, the concept of a reference lifetime is quite involved. For example, Blandy and Orendorff describe them as follows [17]:

> *"A lifetime is some stretch of your program for which a reference could be safe to use: a lexical block, a statement, an expression, the scope of some variable, or the like."*

We can use explicit blocks in Rust to illustrate lifetimes in a more concrete fashion, as follows:

```
fn main() { let x = 1; { let y = 2; } ... }
```

Here, the lifetime of the respective variables is determined (roughly speaking) by their scope.[2] These lifetimes are anonymous and have not been given explicit names. We say that the lifetime of `x` *outlives* that of `y`. Thus, no reference to `y` can exist outside the innermost scope. A motivating use case for reference lifetimes is to prevent stack-allocated data from escaping its allocation scope. The following illustrates the canonical example:

```
fn f(p : &i32) -> &i32 {
  let x = 1;
  let y = &x;
  return y;
}
```

This illustrates an attempt to return a borrowed reference to a local variable (i.e. `&x`) whose lifetime (i.e. allocation context) has expired. For this program, Rust reports the following error:

```
error[E0515]: cannot return value referencing local variable `x`
7 |   let y = &x;
  |           -- `x` is borrowed here
8 |   return y;
  |          ^ returns a value referencing data owned by the current function
```

This error highlights how the Rust compiler prohibits dangling references by reasoning about reference lifetimes. In this case, the lifetime of the returned reference must match that of the parameter (see §2.4.3 for more on why) which, by construction, outlives the function. Since the compiler must prevent borrows from outliving their referents, the above is rejected.

In Rust, there is also a *global* lifetime called `static` which (roughly speaking) corresponds to a `static` variable in C/C++ or in Java. The following illustrates:

```
static VALUE: i32 = 1;

fn f() -> &'static i32 { return &VALUE; }
```

Here, we have effectively declared a global variable `VALUE` and, hence, the above is accepted by the Rust compiler as we can return a reference to this by specifying its lifetime explicitly.

2.4.1 Lifetime of Borrows. The lifetime of a borrowed reference is constrained by the variable to which it is bound. If that variable goes out of scope then the borrow has certainly expired (and may even have expired before this). The key, however, is that assigning a borrowed reference to another variable can affect its lifetime. The following example, which compiles without problem, illustrates:

```
fn f() -> i32 {
  let x = 0;
  let y;
  {
    let z = &x;
    y = z;
  }
  return *y;
}
```

[2]We say "roughly speaking" here since, in practice, a variable's lifetime may be smaller than its enclosing scope in certain situations.

Here, the lifetime of the borrow `&x` is initially constrained to the innermost block. However, the subsequent assignment to `y` extends it to that of the enclosing function.

2.4.2 Lifetime Polymorphism. Previously, we illustrated the use of the global lifetime `static`. However, we can also use explicit lifetime parameters to improve flexibility. The following illustrates a simple example:

```
fn f<'a>() -> &'a i32 {
    let x = 1;
    let y = &x;
    return y;
}
```

Here, we used an explicit lifetime parameter (i.e. `'a`) to specify the lifetime of the returned reference. The Rust compiler reports the following error:

```
error: `x` does not live long enough
let y = &x;
        ^
note: reference must be valid for the lifetime 'a ...
```

We can see from this error that the Rust compiler is reasoning about the lifetime of variable `x`. Lifetime parameters can also be given for `struct` types, as the following illustrates:

```
struct pInt<'a> { value : &'a i32 }
   ...
   let x = 1;
   let y = pInt{ value: &x };
```

This is acceptable to the Rust compiler which infers that `y` is a `struct` containing a borrowed reference to an integer in an enclosing lifetime.

2.4.3 Lifetime Elision. In certain situations, the Rust compiler will automatically infer lifetimes, thereby allowing the user to omit them. The reason for this is presumably to simplify common cases, particularly for novices getting started with the language. The mechanism for this, unfortunately, is subtle. For each elided lifetime amongst the parameter type(s), the compiler creates a fresh lifetime parameter. If the return type does not contain a borrowed reference with an elided lifetime, then this is already sufficient. Otherwise, the compiler must infer these elided lifetimes from amongst the available lifetime parameters (including those arising from elisions amongst the parameter types). If only one such lifetime parameter exists, this will be chosen for all elided lifetimes in the return type(s). Otherwise, the program is rejected as there is no obvious way to choose. The following illustrates:

```
fn f(x: &str) -> &str { return x; }
```

Here, the lifetime for variable `x` as been elided. As a result, the Rust compiler automatically infers the lifetime of the return value to match that of `x`. Thus, the compiler infers the following signature for `f`:

```
fn f<'a>(x: &'a str) -> &'a str
```

The lifetime variable inferred by the compiler is, of course, fresh and will not clash with any other declared lifetime variable.

3 CALCULUS

We now present our calculus, FR, for reasoning about lifetimes and borrow checking in Rust. This is intentionally minimal in nature to capture only the salient aspects. FR most closely resembles that of Rust 1.0 where lifetimes were largely based on the lexical structure of programs. However, since 2018, Rust moved to supporting *Non-Lexical Lifetimes (NLL)* which offers greater flexibility and can accept more programs as correct.[3] Nevertheless, FR remains a useful building block for exploring further aspects of the language. FR employs flow-sensitive rules for type and borrow checking and obtains a *type and borrow safety* theorem for well-typed programs. FR is imperative in nature and supports *copy- and move-semantics*, *mutable/immutable borrowing*, *partial moves*, and *reference lifetimes*. It does not employ λ-values as the primary construct, for example, as found in the λ-calculus. Instead, the main construct is that of a statement block which defines a lifetime for variables declared within. FR also does not capture unsafe code and does not support non-lexical lifetimes. We note also the approach to borrowing and lifetimes in FR differs significantly from the implementation found in `rustc` but (we believe) offers similar expressiveness.[4]

3.1 Syntax

The syntax for FR is given in Figure 1 and its primary constructs are *terms*, *lvals*, *partial values*, *values*, *partial types* and *types*. We now highlight the main features of FR:

(1) **Types.** A borrowed reference is either *immutable* (e.g. "&x") or *mutable* (e.g. "&mut x"). Supporting both kinds allows us to capture their differences, namely that the former has copy semantics whilst the latter has move semantics. Note, we will often write "&[mut] x" to match borrowed references which are *either* immutable *or* mutable. Additionally, borrow types can target multiple locations (e.g. "&x, y") and arbitrary lvals (e.g. "&∗x").[5] The primitive integer type (i.e. int) has copy semantics, whilst box types (e.g. □T) have move semantics and represent dynamically allocated memory (recall §2.2.1). Partial Types are types where one or more components are currently *undefined*. A type is always a partial type, but not necessarily the other way around. An undefined portion of a partial type is denoted by ⌊T⌋ (e.g. "⌊&x⌋") and represents a location which is currently inaccessible (i.e. because it was previously moved).

(2) **Blocks** "$(\{\overline{t}\}^{l})$" are sequences of terms with assigned lifetimes, l. We regard semicolons as *separators*, rather than *terminators*. Thus, for example, "$\{x = y;\}^{l}$" is (technically speaking) syntactically incorrect. However, for brevity, we allow this as short-hand notation for "$\{x = y; \epsilon\}^{l}$". Lifetimes are assumed to form a partial order denoted by $l \succeq m$, which is taken to mean that lifetime m is *inside* l. Since we have a partial order, it follows that a lifetime is always inside itself (i.e. $l \succeq l$ always holds). The ordering between lifetimes assigned to blocks is assumed to reflect their relative nesting. For example, for a block "$\{\{t\}^{m}\}^{l}$" we assume $l \succeq m$.

(3) **Values.** The special *unit* value corresponds with the empty tuple in Rust and is produced from executing a statement (e.g. "let mut x = 0"). Integer values are assumed to be drawn from some finite set, such as those corresponding with a 32-bit two's complement representation. Reference values are split into *owning* references (e.g. "$\ell_x^{\bullet}$") and *borrowed* references (e.g. "ℓ_y°"). The former have responsibility for recursively dropping (i.e. deallocating) the location they refer to when they are dropped, whilst the latter do not. This distinction is necessary to properly separate semantics from typing.

[3] See RFC2094 for more on Non-Lexical Lifetimes (https://github.com/rust-lang/rfcs/blob/master/text/2094-nll.md).
[4] Indeed the experimental borrow checker, Polonius, available in `rustc` (nightly) is perhaps bringing them closer together.
[5] We note LVals are referred to as *place expressions* or simply *places* in the Rust compiler and MIR.

t ::=		*Terms*
	$\{\overline{\mathtt{t}}\}^{\mathtt{l}}$	*block*
	`let mut x = t`	*declaration*
	`w = t`	*assignment*
	`box t`	*heap allocation*
	`&[mut] w`	*(mutable) borrow*
	`w`	*move*
	$\hat{\mathtt{w}}$	*copy*
	`v`	*value*
w ::=		*LVals*
	`x`	*variable*
	`*w`	*dereference*
$\mathtt{v}^{\perp}$ ::=		*Partial Values*
	`v`	*value*
	$\perp$	*undefined*

v ::=		*Values*
	ϵ	*unit*
	`c`	*integer*
	$\ell^{\bullet}, \ell^{\circ}$	*reference*
$\widetilde{\mathtt{T}}$ ::=		*Partial Types*
	`T`	*type*
	$\square\widetilde{\mathtt{T}}$	*partial box*
	$\lfloor \mathtt{T} \rfloor$	*undefined*
T ::=		*Types*
	ϵ	*unit*
	`int`	*integer*
	`&mut` $\overline{\mathtt{w}}$	*mutable borrow*
	$\&\overline{\mathtt{w}}$	*immutable borrow*
	$\square$`T`	*box*

Fig. 1. Syntax for FR.

As an example, a simple program written in FR is given below to illustrate:

$$\{\texttt{let mut x = box 0; } \{\texttt{let mut y = \&mut x; *y = box 1;}\}^{\mathtt{m}} \texttt{ let mut z = x;}\}^{\mathtt{l}} \tag{1}$$

This is a valid program containing an inner block which borrows and mutates variable `x`, before releasing its ownership back to the enclosing block. The syntax "`box e`" is equivalent to `Box::new(e)` and, hence, "`box 0`" allocates a new box on the heap initialised to `0`.[6] The final statement moves the value of `x` to `z` and, hence, this is a valid FR program. Explicit syntax, $\hat{\mathtt{x}}$, is required to indicate when a variable *copy* should occur as opposed to a *move*. This syntax is necessary to correctly model runtime memory management in the calculus.[7] As such, the following minor variation is not a valid FR program:

$$\{\texttt{let mut x = box 0; } \{\texttt{let mut y = \&mut x; *y = box 1;}\}^{\mathtt{m}} \texttt{ let mut z = } \hat{\mathtt{x}}\texttt{;}\}^{\mathtt{l}} \tag{2}$$

Unlike before, `x` is now copied in the last statement and this leads to a violation of the ownership invariant, since both `x` and `z` now refer to the same heap location (see Figure 2). Regardless, in this particular case, the program still successfully reduces to ϵ.

Finally, *source-level* terms are those which could be written by a programmer, whilst other terms arise only during execution. Specifically, source-level terms cannot contain reference values. For example, "$\{$`let mut x = box 1`$\}$" is a source level term where "$\{$`let mut x =` $\ell^{\bullet}\}$" is not. We also note FR allows terms which are forbidden in Rust (e.g. "`let mut x = (let mut y = 0);`") and we tacitly assume these are not expressible at the source-level.

[6]The "box e" syntax is valid Rust, though not yet available in the stable compiler.
[7]In fact, the Mid-level Intermediate Representation (MIR) used in the Rust compiler does something similar.

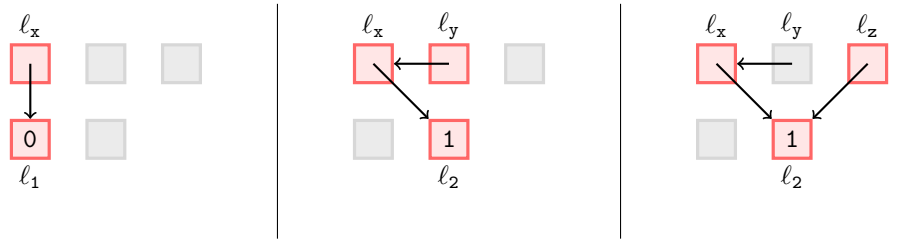

Fig. 2. Visualising three stages of the execution of program (2). The leftmost diagram illustrates the state after executing the first statement; the middle diagram the state after executing the inner block; and, finally, the rightmost diagram illustrates the state after executing the final statement. Here, location ℓ_x is that bound to x, location ℓ_y is that bound to y, etc.

3.2 Semantics

The semantics for FR is presented in the form of a small-step operational semantics which lends itself naturally to proofs of *progress* and *preservation* [134]. The small-step semantics for the core language is presented as reduction rules of the general form $\langle\ \mathcal{S} \triangleright \mathtt{t} \longrightarrow \mathcal{S}' \triangleright \mathtt{t}'\ \rangle^{\mathtt{l}}$. Here, $\mathcal{S} \triangleright \mathtt{t}$ represents the state of the machine *before* term t begins evaluation (i.e. the *pre-state*). Likewise, $\mathcal{S}' \triangleright \mathtt{t}'$ represents the state of the machine *after* term t has taken a single step (i.e. the *post-state*). Furthermore, $\mathcal{S}$ and $\mathcal{S}'$ represent the *program store* which maps locations to partial "slot" values, $\langle \mathtt{v}^{\perp} \rangle^{\mathtt{m}}$, extended with their allocated lifetime m. The *lifetime context*, l, identifies the enclosing lifetime in which t is reducing as it is sometimes necessary to know this (e.g. when declaring new variables). For brevity, we often omit the lifetime context when unnecessary. The transition $\langle\ \{\ell_x \mapsto \langle 1 \rangle^{\mathtt{m}}\} \triangleright \mathtt{x} \longrightarrow \{\ell_x \mapsto \langle 1 \rangle^{\mathtt{m}}\} \triangleright 1\ \rangle^{\mathtt{l}}$ illustrates a simple reduction where variable x is represented by a *named* location, ℓ_x, which was allocated in lifetime m and currently has value 1. In contrast, box values are represented in the semantics as *unnamed* (i.e. heap) locations, ℓ_n.

At this point it is worth noting that variables in FR are, intuitively, quite different from those in, for example, the λ-calculus. This is because, in the latter, variables are simply let-bound and do not represent mutable state. In contrast, a variable in FR corresponds to a *mutable* location that may be updated as the program proceeds and whose lifetime is bound to that of its enclosing block. Furthermore, when execution proceeds into a block, a fresh mutable location is created for each declared variable which is subsequently dropped (i.e. deallocated) when the block completes.[8] Our treatment of variables as mutable locations means that FR does not support variable shadowing which, although not ideal, provides important simplifications. Furthermore, it is relatively easy to simulate shadowing through a simple renaming process prior to reduction.

3.2.1 Preliminaries. Before presenting the reductions for FR, we first consider some necessary support functions. The most important of these is, perhaps, the partial function $\mathtt{loc}(\mathcal{S}, \mathtt{w})$:

Definition 3.1 (Locate). The partial function $\mathtt{loc}(\mathcal{S}, \mathtt{w})$ determines the location associated with a given lval in a given store:

$$\begin{array}{rcll} \mathtt{loc}(\mathcal{S}, \mathtt{x}) & = & \ell_x & \\ \mathtt{loc}(\mathcal{S}, *\mathtt{w}) & = & \ell & \textbf{where } \mathtt{loc}(\mathcal{S}, \mathtt{w}) = \ell_w \textbf{ and } \mathcal{S}(\ell_w) = \langle \ell^* \rangle^{\mathtt{m}} \end{array}$$

Here, "ℓ^*" is used to match both owning and borrowed references. Thus, for example, we have $\mathtt{loc}(\{\ell_x \mapsto \langle 1 \rangle^{\mathtt{m}}, \ell_p \mapsto \langle \ell_x^{\circ} \rangle^{\mathtt{n}}\}, *\mathtt{p}) = \ell_x$. For the core calculus, $\mathtt{loc}(\mathcal{S}, \mathtt{w})$ is defined quite simply (as above) and assumes a straightfoward mapping between variables (e.g. x) and their corresponding

[8] In the vernacular of Rust, the Drop trait provides something akin to a destructor as found in C++.

locations (e.g. ℓ_x). However, extensions to the calculus may refine this to support more complex associations (e.g. to support references to locations *within* compound values, or to support *dynamic* binding with stack frames). Finally, observe that $\texttt{loc}(\mathcal{S}, \texttt{w})$ is a *partial* function when the program store does not match the given lval (e.g. $\texttt{loc}(\{\ell_x \mapsto \langle 1 \rangle^{\texttt{m}}\}, *\texttt{x})$ is undefined). This is not a specific cause for concern, as the machine simply becomes stuck in such cases.

In addition to a mechanism for locating lvals, we additionally require the ability to *read* from them and *write* to them. As for $\texttt{loc}(\mathcal{S}, \texttt{w})$, these provide hooks which can be refined by extensions to the core calculus as necessary. For now, however, they are defined trivially as follows:

Definition 3.2 (Read). The partial function $\texttt{read}(\mathcal{S}, \texttt{w})$ retrieves the valued stored in a given lval:

$$\texttt{read}(\mathcal{S}, \texttt{w}) \quad = \quad \mathcal{S}(\ell_{\texttt{w}}) \quad \textbf{where } \texttt{loc}(\mathcal{S}, \texttt{w}) = \ell_{\texttt{w}}$$

Definition 3.3 (Write). The partial function $\texttt{write}(\mathcal{S}, \texttt{w}, \texttt{v}^{\perp})$ updates the value stored in a given lval:

$$\texttt{write}(\mathcal{S}, \texttt{w}, \texttt{v}^{\perp}) \quad = \quad \mathcal{S}[\ell_{\texttt{w}} \mapsto \langle \texttt{v}^{\perp} \rangle^{\texttt{m}}] \quad \textbf{where } \texttt{loc}(\mathcal{S}, \texttt{w}) = \ell_{\texttt{w}} \textbf{ and } \mathcal{S}(\ell_{\texttt{w}}) = \langle \cdot \rangle^{\texttt{m}}$$

The notation $\mathcal{S}[\ell \mapsto \langle \texttt{v}^{\perp} \rangle^{\texttt{m}}]$ returns a program store identical to $\mathcal{S}$ except where location ℓ now has (partial) value $\langle \texttt{v}^{\perp} \rangle^{\texttt{m}}$. Likewise, $\langle \cdot \rangle^{\texttt{m}}$ represents a slot with some arbitrary (i.e. *don't care*) value. Note also how the lifetime of $\ell_{\texttt{w}}$ is unchanged by this operation. Finally, note both $\texttt{read}(\mathcal{S}, \texttt{w})$ and $\texttt{write}(\mathcal{S}, \texttt{w}, \texttt{v}^{\perp})$ are partial functions since $\texttt{loc}(\mathcal{S}, \texttt{w})$ is a partial function.

3.2.2 Expressions. We now proceed to present and explain the reduction rules for FR starting with expressions. The first rule is R-Copy for reducing copy expressions:

$$\frac{\texttt{read}(\mathcal{S}, \texttt{w}) = \langle \texttt{v} \rangle^{\texttt{m}}}{\langle\ \mathcal{S} \triangleright \hat{\texttt{w}} \longrightarrow \mathcal{S} \triangleright \texttt{v}\ \rangle^{1}} \qquad \text{(R-Copy)}$$

This rule simply copies the value stored in the location denoted by w, but does not perform a destructive read. To guarantee memory safety, the borrow checker should only permit this for values which exhibit copy semantics (more later). In the core calculus, only mutable references and boxes do not exhibit copy semantics. For values with move semantics, the following should be used instead:

$$\frac{\texttt{read}(\mathcal{S}_1, \texttt{w}) = \langle \texttt{v} \rangle^{\texttt{m}} \quad \mathcal{S}_2 = \texttt{write}(\mathcal{S}_1, \texttt{w}, \perp)}{\langle\ \mathcal{S}_1 \triangleright \texttt{w} \longrightarrow \mathcal{S}_2 \triangleright \texttt{v}\ \rangle^{1}} \qquad \text{(R-Move)}$$

This rule is responsible for reducing an lval by moving (i.e. rather than copying) and illustrates how this affects the program store. The rule implements a *destructive read* whereby the lval w is subsequently rendered unusable by effectively removing its location from the resulting program store. This is achieved using the special *undefined item* $\perp$ which is *not* a value in FR and cannot be returned by $\texttt{read}(\mathcal{S}, \texttt{w})$. Thus, for example, $\texttt{read}(\{\texttt{x} \mapsto \langle \perp \rangle^{\texttt{m}}\}, \texttt{x})$ is undefined and any attempt to do this renders a stuck program.

To handle heap allocations, rule R-Box creates a fresh location in the store representing the box being created:

$$\frac{\ell_{\texttt{n}} \notin \textbf{dom}(\mathcal{S}_1) \quad \mathcal{S}_2 = \mathcal{S}_1[\ell_{\texttt{n}} \mapsto \langle \texttt{v} \rangle^{*}]}{\langle\ \mathcal{S}_1 \triangleright \texttt{box v} \longrightarrow \mathcal{S}_2 \triangleright \ell_{\texttt{n}}^{\bullet}\ \rangle^{1}} \qquad \text{(R-Box)}$$

Here, heap locations are given the *global lifetime*, $*$, which all other lifetimes are assumed to be inside. Finally, the rule for borrows determines the location of the lval being borrowed:

$$\frac{\mathtt{loc}(\mathcal{S},\mathtt{w}) = \ell_{\mathtt{w}}}{\langle\ \mathcal{S} \triangleright \&[\mathtt{mut}]\ \mathtt{w} \longrightarrow \mathcal{S} \triangleright \ell^{\circ}_{\mathtt{w}}\ \rangle^{1}} \quad \text{(R-Borrow)}$$

3.2.3 Statements. We now proceed to consider the reduction of statements in FR whilst, for now, ignoring the question of how subterms are reduced (more later). We first consider rule R-Assign for the reduction of assignments:

$$\frac{\mathtt{read}(\mathcal{S}_1,\mathtt{w}) = \langle \mathtt{v}_1^{\perp}\rangle^{\mathtt{m}} \quad \mathcal{S}_2 = \mathtt{drop}(\mathcal{S}_1,\{\mathtt{v}_1^{\perp}\}) \quad \mathcal{S}_3 = \mathtt{write}(\mathcal{S}_2,\mathtt{w},\mathtt{v}_2)}{\langle\ \mathcal{S}_1 \triangleright \mathtt{w} = \mathtt{v}_2 \longrightarrow \mathcal{S}_3 \triangleright \epsilon\ \rangle^{1}} \quad \text{(R-Assign)}$$

The function $\mathtt{drop}(\mathcal{S},\{\mathtt{v}^{\perp}\})$ is responsible for dropping (i.e. deallocating) any locations owned by $\mathtt{v}^{\perp}$ (more on this later). The rule for handling variable declarations determines the variable's corresponding location, and updates the program store accordingly:

$$\frac{\mathcal{S}_2 = \mathcal{S}_1[\ell_{\mathtt{x}} \mapsto \langle \mathtt{v}\rangle^{1}]}{\langle\ \mathcal{S}_1 \triangleright \mathtt{let\ mut\ x = v} \longrightarrow \mathcal{S}_2 \triangleright \epsilon\ \rangle^{1}} \quad \text{(R-Declare)}$$

Observe that, in this case, the function $\mathtt{write}(\mathcal{S}_1,\mathtt{x},\mathtt{v})$ cannot be used because this requires x to have already been declared and, hence, associated with a lifetime. Instead, the store is updated directly using the lifetime l of the enclosing block. The reduction of a sequence of terms proceeds by removing those completed from the left:

$$\frac{\mathcal{S}_2 = \mathtt{drop}(\mathcal{S}_1,\{\mathtt{v}\})}{\langle\ \mathcal{S}_1 \triangleright \mathtt{v};\overline{\mathtt{t}} \longrightarrow \mathcal{S}_2 \triangleright \overline{\mathtt{t}}\ \rangle^{1}} \quad \text{(R-Seq)}$$

The reduction of blocks continues via R-BlockA until only a single value remains. Recall from §3.1 that such a value must exist because semicolons are separators, not terminators. Thus, for example, "x = y;" is short hand for "x = y; ϵ", etc. Once only a single value remains, the block is reduced entirely via rule R-BlockB:

$$\frac{\langle\ \mathcal{S}_1 \triangleright \overline{\mathtt{t}_1} \longrightarrow \mathcal{S}_2 \triangleright \overline{\mathtt{t}_2}\ \rangle^{\mathtt{m}}}{\langle\ \mathcal{S}_1 \triangleright \{\overline{\mathtt{t}_1}\}^{\mathtt{m}} \longrightarrow \mathcal{S}_2 \triangleright \{\overline{\mathtt{t}_2}\}^{\mathtt{m}}\ \rangle^{1}} \quad \text{(R-BlockA)} \qquad \frac{\mathcal{S}_2 = \mathtt{drop}(\mathcal{S}_1,\mathtt{m})}{\langle\ \mathcal{S}_1 \triangleright \{\mathtt{v}\}^{\mathtt{m}} \longrightarrow \mathcal{S}_2 \triangleright \mathtt{v}\ \rangle^{1}} \quad \text{(R-BlockB)}$$

A key aspect of this rule is the deallocation of any remaining owned locations. Observe that dropping locations allocated in one block may also cause locations dynamically allocated elsewhere to be dropped (as can happen after one location is moved into another). The function $\mathtt{drop}(\mathcal{S},\mathtt{m})$ is responsible for this and recursively drops locations declared in lifetime m:

Definition 3.4 (Drop). Let $\mathcal{S}$ be a program store. Then, the function $\mathtt{drop}(\mathcal{S},\mathtt{m})$ is defined as $\mathtt{drop}(\mathcal{S},\psi)$ where $\psi = \{\ell^{\bullet} \mid \ell \mapsto \langle \mathtt{v}^{\perp}\rangle^{\mathtt{m}} \in \mathcal{S}\}$. Here, $\mathtt{drop}(\mathcal{S},\psi)$ recursively deallocates owned locations as follows:

$$\begin{array}{rcll} \mathtt{drop}(\mathcal{S},\emptyset) & = & \mathcal{S} & \\ \mathtt{drop}(\mathcal{S},\psi \cup \{\mathtt{v}^{\perp}\}) & = & \mathtt{drop}(\mathcal{S},\psi) & \textbf{if } \mathtt{v}^{\perp} \neq \ell^{\bullet} \\ \mathtt{drop}(\mathcal{S},\psi \cup \{\ell^{\bullet}\}) & = & \mathtt{drop}(\mathcal{S} - \{\ell \mapsto \langle \mathtt{v}^{\perp}\rangle^{*}\},\psi \cup \{\mathtt{v}^{\perp}\}) & \textbf{where } \mathcal{S}(\ell) = \langle \mathtt{v}^{\perp}\rangle^{*} \end{array}$$

The function $\mathtt{drop}(\mathcal{S},\psi)$ traverses owning references dropping location as necessary. Here, ψ is referred to as the *drop set* which identifies those locations allocated by a given block and, hence, which should be dropped when it completes. Observe that partial values are dropped by traversing the defined portions. As an aside, the order in which locations are dropped in Rust is important as it determines the invocation order for destructors. If the ordering is wrong, this can result in an attempt to release data already released which in most languages (e.g. C/C++) results in undefined

behaviour. From our perspective here, this is not a concern as attempting to remove an item from a store which does not contain that item has no effect.

Finally, we consider how subterms are reduced. To simplify the presentation, we employ a single rule for this based around the notion of an evaluation context:

Definition 3.5 (Evaluation Context). An *evaluation context* is a term containing a single occurrence of $[\![\cdot]\!]$ (the hole) in place of a subterm. Evaluation contexts are defined as follows:

$$E \quad ::= \quad [\![\cdot]\!] \mid E \,;\, \overline{\mathtt{t}} \mid \mathtt{let\ mut\ x} = E \mid \mathtt{w} = E \mid \mathtt{box}\ E$$

Here, for example, if E is "$\mathtt{let\ mut\ x} = [\![\cdot]\!]$" then $E[\![\mathtt{y}]\!]$ gives "$\mathtt{let\ mut\ x} = \mathtt{y}$", etc. Using this, we have the following reduction rule:

$$\frac{\langle\ \mathcal{S}_1 \triangleright \mathtt{t}_1 \longrightarrow \mathcal{S}_2 \triangleright \mathtt{t}_2\ \rangle^1}{\langle\ \mathcal{S}_1 \triangleright E[\![\mathtt{t}_1]\!] \longrightarrow \mathcal{S}_2 \triangleright E[\![\mathtt{t}_2]\!]\ \rangle^1} \qquad \text{(R-Sub)}$$

Thus, the transition $\langle\ \{\ell_\mathtt{y} \mapsto \langle 1\rangle^1\} \triangleright\ \mathtt{box\ y} \longrightarrow \{\ell_\mathtt{y} \mapsto \langle 1\rangle^1\} \triangleright\ \mathtt{box\ 1}\ \rangle^1$ occurs via R-Sub, as does $\langle\ \{\ell_\mathtt{y} \mapsto \langle 1\rangle^1\} \triangleright\ \mathtt{y} = 2; \mathtt{x} = \mathtt{y} \longrightarrow \{\ell_\mathtt{y} \mapsto \langle 2\rangle^1\} \triangleright\ \epsilon; \mathtt{x} = \mathtt{y}\ \rangle^1$. Observe that, whilst sequences are evaluation contexts, blocks are not. This is because blocks require careful handling of the enclosing lifetime (which, instead, is managed via R-BlockA and R-BlockB).

3.2.4 Worked Example. To help understand our operational semantics, we now consider a short worked example. The purpose of this is to highlight the main features, in particular copy- versus move-semantics and location dropping when a block completes. The initial state of our reduction is the following:

$$\emptyset \triangleright \{\ \mathtt{let\ mut\ x} = 1;\ \mathtt{let\ mut\ y} = \mathtt{box}\ \hat{\mathtt{x}};\ \{\ \mathtt{let\ mut\ z} = \mathtt{box}\ 0;\ \mathtt{y} = \&\mathtt{z};\ \mathtt{y} = \mathtt{z};\ *\mathtt{y}\ \}^\mathtt{m}\ \}^1 \quad (3)$$

During evaluation of the first two statements three locations are created (one each for x and y, and one dynamically allocated via "box $\hat{\mathtt{x}}$"). After they have completed, we are left in the following state:

$$\{\ell_\mathtt{x} \mapsto \langle 1\rangle^1, \ell_\mathtt{y} \mapsto \langle \ell_1^\bullet\rangle^1, \ell_1 \mapsto \langle 1\rangle^*\} \triangleright \{\ \{\ \mathtt{let\ mut\ z} = \mathtt{box}\ 0;\ \mathtt{y} = \&\mathtt{z};\ \mathtt{y} = \mathtt{z};\ *\mathtt{y}\ \}^\mathtt{m}\ \}^1 \quad (4)$$

We can see that x is still present since its value was copied rather than moved into the box created for y. Likewise, y holds an owning reference to location ℓ_1 (and, hence, at this exact moment, ℓ_1 stands to be dropped if y were dropped). Furthermore, ℓ_1 has global lifetime $*$ since it was dynamically allocated. The leftmost diagram of Figure 3 provides a pictorial view of the store at this point. After evaluating the next statement two more locations are created (for z and box 0), leaving us in the following state:

$$\{\ell_\mathtt{x} \mapsto \langle 1\rangle^1, \ell_\mathtt{y} \mapsto \langle \ell_1^\bullet\rangle^1, \ell_1 \mapsto \langle 1\rangle^*, \ell_\mathtt{z} \mapsto \langle \ell_2^\bullet\rangle^\mathtt{m}, \ell_2 \mapsto \langle 0\rangle^*\} \triangleright \{\ \{\ \mathtt{y} = \&\mathtt{z};\ \mathtt{y} = \mathtt{z};\ *\mathtt{y}\ \}^\mathtt{m}\ \}^1 \quad (5)$$

Again, location ℓ_2 created by box 0 has global lifetime and this is critical as it allows the location to (subsequently) be moved out of the innermost block's scope. The middle diagram of Figure 3 provides a pictorial view of the store at this point. The next statement "y = &z" would not type check in Rust but, nevertheless, serves to further illustrate our semantics. Executing this statement creates a borrowed reference, leaving us in the following state:

$$\{\ell_\mathtt{x} \mapsto \langle 1\rangle^1, \ell_\mathtt{y} \mapsto \langle \ell_\mathtt{z}^\circ\rangle^1, \ell_\mathtt{z} \mapsto \langle \ell_2^\bullet\rangle^\mathtt{m}, \ell_2 \mapsto \langle 0\rangle^*\} \triangleright \{\ \{\ \mathtt{y} = \mathtt{z};\ *\mathtt{y}\ \}^\mathtt{m}\ \}^1 \quad (6)$$

The remaining assignment statement is perhaps the most interesting, as this implements a move from z to y. In turn, this drops reference $\ell_\mathtt{z}^\circ$ which has no effect since this is not an owning reference. After executing this we are in the following state:

$$\{\ell_\mathtt{x} \mapsto \langle 1\rangle^1, \ell_\mathtt{y} \mapsto \langle \ell_2^\bullet\rangle^1, \ell_2 \mapsto \langle 0\rangle^*\} \triangleright \{\ \{\ *\mathtt{y}\ \}^\mathtt{m}\ \}^1 \quad (7)$$

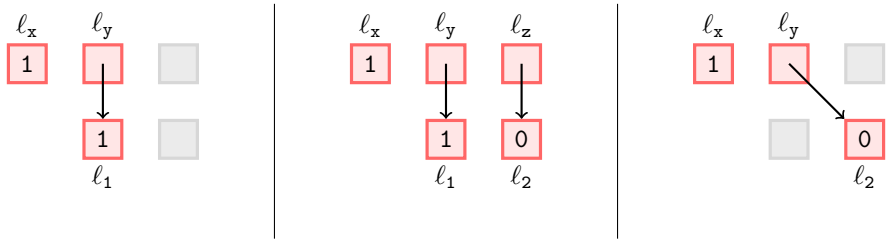

Fig. 3. Illustrating a visualisation of the program store during evaluation of our worked example, where light gray locations are either not (yet) allocated or have been deallocated.

The rightmost diagram of Figure 3 provides a pictorial view of the store at this point. The reduction of this assignment statement gives us insight into the purpose of ownership in Rust, namely to enable implicit memory management without garbage collection. If the statement "`y = ẑ;`" were executed instead, we would end up with both ℓ_y and ℓ_z pointing to ℓ_2 making implicit memory management unsafe. This is because, when the inner block completed, ℓ_z would be dropped which, in turn, would recursively drop ℓ_2 — leaving ℓ_y holding a dangling reference. In contrast, if dropping a location did not recursively drop its owned locations, then dynamically allocated locations would never be reclaimed. Finally, executing the dereference term leaves things as follows:

$$\{\ell_x \mapsto \langle 1 \rangle^1, \ell_y \mapsto \langle \ell_2^\bullet \rangle^1, \ell_2 \mapsto \langle \bot \rangle^*\} \triangleright \{\ \{\ \texttt{0}\ \}^m\ \}^1 \tag{8}$$

The execution of $*$`y` was achieved with R-Move which destructively read lval $*$`y`, leaving ℓ_2 holding $\bot$. At this point, the inner block has not yet completed and requires two further steps before the entire term is reduced to ϵ and all locations are dropped via R-BlockB.

3.3 Typing Judgments

We now consider the question of what it means for programs in our calculus to be *type and borrow safe*. Programs which are type and borrow safe cannot, for example, use dangling references or break the ownership invariant for mutable borrowed references. Of course, not all programs meet these requirements. For example, consider this minimal program:

$$\{\ \texttt{let mut x = 0; let mut y = \&mut x; x = 1;}\ \}^1 \tag{9}$$

This program will successfully evaluate to ϵ but, nevertheless, is not considered type and borrow safe as variable `x` is assigned whilst borrowed to `y` (i.e. whilst it is frozen). Here's another example:

$$\{\texttt{let mut x = 0; let mut y = \&mut x; }\{\texttt{let mut z = 0; y = \&mut z;}\}^m\ \texttt{let mut w = y;}\}^1 \tag{10}$$

This program is incorrect because it attempts to create a borrowed reference to variable `z` that exists outside of its lifetime. This program is not considered type and borrow safe because, in the assignment "`y = &mut z`", the type of "`&mut z`" is not a subtype of `y`. The purpose of the type system presented as part of our calculus is to identify erroneous programs, such as these.

Since borrow checking (as opposed to just type checking) is an inherently flow-sensitive activity, our presentation employs flow-sensitive typing rules (a.k.a *flow typing*). In particular, our typing rules determine both the type for a given term, as well as its effect(s). Judgments have the form "$\Gamma_1 \vdash \langle\ \texttt{t} : \texttt{T}\ \rangle^{\texttt{l}}_{\sigma} \dashv \Gamma_2$" which are taken to mean: firstly, within lifetime `l`, term `t` has type `T` under typing environment Γ_1; and, secondly, that evaluating term `t` under typing environment Γ_1 produces the (potentially updated) environment Γ_2. Thus, we can see the *effect* of term `t` in the difference between the two environments. An environment, Γ, maps each variable to a "slot" type $\langle \tilde{\texttt{T}} \rangle^m$

with allocated lifetime m. Finally, following Pierce, the *store typing*, σ, is needed for locations "in flight" [104]. To understand this, consider the following transition:

$$\emptyset \triangleright \{\texttt{let mut x = box 0;}\}^1 \longrightarrow \{\ell_1 \mapsto \langle 0 \rangle^*\} \triangleright \{\texttt{let mut x = } \ell_1^\bullet\texttt{;}\}^1 \tag{11}$$

The challenge here lies in typing the partially reduced `let` statement after the transition. Specifically, this refers to the heap-allocated location ℓ_1 which is not represented in any typing environment. Hence, the store typing is used to resolve this (where $\sigma \vdash \ell_1^\bullet : \square\texttt{int}$ in this case).

Borrowing. With respect to borrow checking, there are two essential challenges to be addressed: firstly, the *permanent* movement of locations; secondly, the *temporary* borrowing of locations. To understand the former, consider the following program:

$$\{\ \texttt{let mut x = box 0;}\ \{\texttt{let mut y = x;} \ldots\}^m\ \}^1 \tag{12}$$

Once the box reference stored in x is moved to y it takes on the lifetime of y and, thus, responsibility for its deallocation now rests with y. As such, the visibility of x is permanently lost due to the move. This contrasts with the following alternative:

$$\{\ \texttt{let mut x = box 0;}\ \{\texttt{let mut y = \&mut x;} \ldots\}^m\ \}^1 \tag{13}$$

In this case, the visibility of x is lost only within the inner scope and, once evaluation of the inner block completes, x regains ownership of its location. As such, the visibility of x is temporarily lost due to the mutable borrow and y takes on no responsibilities regarding deallocation.

To properly manage borrowing, the type system must therefore be able to determine the *borrow status* of a given variable. To this end, a variable's borrow status is determined by examining the environment to look for any variables of — or containing — type "&x" or "&mut x".

Reborrowing. Borrowing of variables in Rust extends to general lvals where it is often referred to as reborrowing (recall §2.3.2). The following illustrates:

$$\{\ \texttt{let mut x = 0;}\ \{\texttt{let mut y = \&mut x;}\ \{\texttt{let mut z = \&*y;}\ \ldots\}^n\}^m\}^1 \tag{14}$$

The innermost assignment reborrows the value held by y meaning both y and z now refer to x. Since the reborrow is immutable in this case, y can still be used for reading (though not writing). Furthermore, once the reborrow expires (i.e. when the innermost block completes), then y is fully restored as a mutable borrow. To help understand this, consider the following runtime environment (left) and its corresponding typing environment (right):

$$\{\ell_x \mapsto \langle 0 \rangle^1, \ell_y \mapsto \langle \ell_x^\circ \rangle^m, \ell_z \mapsto \langle \ell_x^\circ \rangle^n\} \sim \{\texttt{x} \mapsto \langle \texttt{int} \rangle^1, \texttt{y} \mapsto \langle \texttt{\&mut x} \rangle^m, \texttt{z} \mapsto \langle \texttt{\&*y} \rangle^n\} \tag{15}$$

In this case, reading from x is prohibited (i.e. because of the type retained for y), but reading from either $*$y or $*$z is permitted. However, as expected, writing to $*$y is prohibited by the type for z.

Box Types. With respect to typing heap locations, the calculus takes a different direction. Recall that the syntax for a box type is given by $\square$T. This is the type of references to heap locations. Since heap locations always have a single owner, we do not represent them individually within the typing environment. To understand this, consider the following runtime environment (left) and its corresponding typing environment (right):

$$\{\ell_x \mapsto \langle \ell_1^\bullet \rangle^1, \ell_1 \mapsto \langle \ell_2^\bullet \rangle^*, \ell_2 \mapsto \langle 0 \rangle^*, \ell_y \mapsto \langle \ell_3^\bullet \rangle^1, \ell_3 \mapsto \langle 0 \rangle^*\} \sim \{\texttt{x} \mapsto \langle \square\square\texttt{int} \rangle^1, \texttt{y} \mapsto \langle \square\texttt{int} \rangle^1\} \tag{16}$$

Here, ℓ_x refers to a heap location ℓ_1 which, in turn, refers to another ℓ_2 holding an integer. As such, the type $\square\square$int given for x captures the two levels of ownership. As an aside, an alternative to using types of the form $\square$T, perhaps, would be to explicitly represent heap locations in the typing

environment. However, this would require a notion of equivalence between isomorphic typing environments, and additional rules to ensure such locations were uniquely referenced, etc.

3.4 Typing Rules

We now examine the typing rules for our calculus. These encode both the rules for type checking and for borrow checking as necessary for correctly determining when move- versus copy-semantics applies.

3.4.1 Preliminaries. Before presenting the typing rules for FR, we first consider some necessary support functions. As before, these provide important hooks when extending the core calculus. The simplest of these identifies which types exhibit copy semantics. Following Rust, mutable references and boxes are the only types in the core calculus which do not exhibit copy semantics:[9]

Definition 3.6 (Copy Types). A type $\mathtt{T}$ has copy semantics, denoted by $\mathtt{copy}(\mathtt{T})$, when $\mathtt{T} = \mathtt{int}$ or $\mathtt{T} = \&\overline{\mathtt{w}}$.

Here, the copy status of a type is fixed. However, for extensions to the core calculus, this need not be the case. For example, the copy status of a pair $(\mathtt{T}_1, \mathtt{T}_2)$ is determined by the copy status of its elements. Another important function is that for determining the join of two types and, by extension, that for two environments:

Definition 3.7 (Type Strengthening). Let $\tilde{\mathtt{T}}_1$ and $\tilde{\mathtt{T}}_2$ be partial types. Then $\tilde{\mathtt{T}}_1$ *strengthens* $\tilde{\mathtt{T}}_2$, denoted as $\tilde{\mathtt{T}}_1 \sqsubseteq \tilde{\mathtt{T}}_2$, according to the following rules:

$$\frac{}{\tilde{\mathtt{T}}_1 \sqsubseteq \tilde{\mathtt{T}}_1}\ \text{(W-Reflex)} \qquad \frac{\tilde{\mathtt{T}}_1 \sqsubseteq \tilde{\mathtt{T}}_2}{\square\tilde{\mathtt{T}}_1 \sqsubseteq \square\tilde{\mathtt{T}}_2}\ \text{(W-Box)} \qquad \frac{\overline{\mathtt{u}} \subseteq \overline{\mathtt{w}}}{\Gamma \vdash \&[\mathtt{mut}]\ \overline{\mathtt{u}} \sqsubseteq \&[\mathtt{mut}]\ \overline{\mathtt{w}}}\ \text{(W-Bor)}$$

$$\frac{\mathtt{T}_1 \sqsubseteq \mathtt{T}_2}{\lfloor \mathtt{T}_1 \rfloor \sqsubseteq \lfloor \mathtt{T}_2 \rfloor}\ \text{(W-UndefA)} \qquad \frac{\mathtt{T}_1 \sqsubseteq \mathtt{T}_2}{\mathtt{T}_1 \sqsubseteq \lfloor \mathtt{T}_2 \rfloor}\ \text{(W-UndefB)} \qquad \frac{\tilde{\mathtt{T}}_1 \sqsubseteq \lfloor \mathtt{T}_2 \rfloor}{\square\tilde{\mathtt{T}}_1 \sqsubseteq \lfloor \square\mathtt{T}_2 \rfloor}\ \text{(W-UndefC)}$$

Again, W-Bor requires the same mutability on both sides and, hence, both $\&\mathtt{x} \sqsubseteq \&\mathtt{x},\mathtt{y}$ and $\&\mathtt{mut}\ \mathtt{x} \sqsubseteq \&\mathtt{mut}\ \mathtt{x},\mathtt{y}$ hold but never $\&\mathtt{mut}\ \mathtt{x} \sqsubseteq \&\mathtt{x},\mathtt{y}$ nor $\&\mathtt{x} \sqsubseteq \&\mathtt{mut}\ \mathtt{x},\mathtt{y}$.

Definition 3.8 (Type Join). Let $\tilde{\mathtt{T}}_1$ and $\tilde{\mathtt{T}}_2$ be partial types. Then their *join*, denoted $\tilde{\mathtt{T}}_1 \sqcup \tilde{\mathtt{T}}_2$, is a partial function returning the *strongest* $\tilde{\mathtt{T}}_3$ such that $\tilde{\mathtt{T}}_1 \sqsubseteq \tilde{\mathtt{T}}_3$ and $\tilde{\mathtt{T}}_2 \sqsubseteq \tilde{\mathtt{T}}_3$.

In essence, the above allows borrows to be combined in a coherent fashion. For example, $\&\mathtt{x} \sqcup \&\mathtt{y}$ gives the type $\&\mathtt{x},\mathtt{y}$ which represents an immutable borrow to either $\mathtt{x}$ or $\mathtt{y}$. Observe it is undefined for types that cannot be combined (e.g. $\mathtt{int} \sqcup \square\mathtt{int}$ is undefined). Again, this is not a cause for concern as, in such case, typing cannot succeed. Also, the treatment of undefined types must be conservative in nature. For example, consider the meaning of "$\square\lfloor\square\mathtt{T}\rfloor \sqcup \square\square\lfloor\mathtt{T}\rfloor$". This can be thought of as determining the type of a variable at a meet point which, on one control-flow path, has type $\square\lfloor\square\mathtt{T}\rfloor$ and, on another, has type $\square\square\lfloor\mathtt{T}\rfloor$. The first type (roughly speaking) describes "a box to nothing" whilst the second describes "a box to a box to nothing". Since the first offers the least information, it must be taken (in this case) to retain soundness. The join relation is then lifted to typing environments in the expected fashion:

Definition 3.9 (Environment Strengthening). Let Γ_1 and Γ_2 be typing environments. Then Γ_1 *strengthens* Γ_2, denoted $\Gamma_1 \sqsubseteq \Gamma_2$, iff $\mathbf{dom}(\Gamma_1) = \mathbf{dom}(\Gamma_2)$ and, for all $\mathtt{x} \in \mathbf{dom}(\Gamma_1)$ where $\Gamma_1(\mathtt{x}) = \langle\tilde{\mathtt{T}}_1\rangle^1$, we have $\Gamma_2(\mathtt{x}) = \langle\tilde{\mathtt{T}}_2\rangle^1$ where $\tilde{\mathtt{T}}_1 \sqsubseteq \tilde{\mathtt{T}}_2$.

[9] In practice, the definition of a copyable type in Rust is determined simply by whether or not it implements the Copy trait.

Definition 3.10 (Environment Join). Let Γ_1 and Γ_2 be environments. Then their *join*, denoted $\Gamma_1 \sqcup \Gamma_2$, is a partial function returning the *strongest* Γ_3 such that $\Gamma_1 \sqsubseteq \Gamma_3$ and $\Gamma_2 \sqsubseteq \Gamma_3$.

In essence, environments are combined by joining the types of all variables in both. We additionally require that variables are declared in the same lifetime but, in fact, this will always be the case in practice. Another important piece of functionality is that for typing lvals:

Definition 3.11 (LVal Typing). An lval $\mathtt{w}$ is said to be typed with respect to an environment Γ, denoted $\Gamma \vdash \mathtt{w} : \langle \tilde{\mathtt{T}} \rangle^{\mathtt{m}}$, according to the following rules:

$$\frac{\Gamma(\mathtt{x}) = \langle \tilde{\mathtt{T}} \rangle^{\mathtt{m}}}{\Gamma \vdash \mathtt{x} : \langle \tilde{\mathtt{T}} \rangle^{\mathtt{m}}}\ (\text{T-LvVar}) \quad \frac{\Gamma \vdash \mathtt{w} : \langle \square \tilde{\mathtt{T}} \rangle^{\mathtt{m}}}{\Gamma \vdash *\mathtt{w} : \langle \tilde{\mathtt{T}} \rangle^{\mathtt{m}}}\ (\text{T-LvBox}) \quad \frac{\Gamma \vdash \mathtt{w} : \langle \&[\mathtt{mut}]\ \overline{\mathtt{u}} \rangle^{\mathtt{n}} \quad \overline{\Gamma \vdash \mathtt{u} : \langle \mathtt{T} \rangle^{\mathtt{m}}}}{\Gamma \vdash *\mathtt{w} : \langle \bigsqcup_{\mathtt{i}} \mathtt{T_i} \rangle^{\sqcap_{\mathtt{i}} \mathtt{m_i}}}\ (\text{T-LvBor})$$

For example, given $\Gamma = \{\mathtt{x} \mapsto \mathtt{int}, \mathtt{p} \mapsto \square\mathtt{int}, \mathtt{q} \mapsto \&\mathtt{x}\}$ it follows that $\Gamma \vdash \mathtt{x} : \mathtt{int}$, $\Gamma \vdash *\mathtt{p} : \mathtt{int}$ and $\Gamma \vdash *\mathtt{q} : \mathtt{int}$. Furthermore, lvals can have partial types provided their internal "path" is defined. Thus, for $\{\mathtt{x} \mapsto \square[\square\mathtt{int}]\}$ it follows that both $\mathtt{x}$ and $*\mathtt{x}$ can be typed, but not $**\mathtt{x}$. Finally, we note that $\sqcap_{\mathtt{i}} \mathtt{m_i}$ returns the least (i.e. innermost) lifetime from $\mathtt{m_0} \ldots \mathtt{m_n}$. Since, at any point in a term, the active lifetimes form a strict linear sequence, this is always well defined. In essence, this identifies the lifetime in which all borrows contained within the type can safely exist.

The ability to determine when a location is mutably or immutably borrowed is imperative to enforcing the safety guarantees of FR. For example, in the environment $\{\mathtt{x} \mapsto \langle \mathtt{int} \rangle^{\mathtt{l}}, \mathtt{y} \mapsto \langle \square\&\mathtt{x} \rangle^{\mathtt{l}}\}$ variable $\mathtt{x}$ is immutably borrowed and, hence, cannot be assigned. In this case, we say that $\mathtt{x}$ is *write prohibited* by $\mathtt{y}$. To determine whether a variable is read or write prohibited requires a mechanism for identifying variables whose types contain conflicting borrows (e.g. $\&\mathtt{x}$). The following provides a foundation for this:

Definition 3.12 (Path). A *path*, π, is a sequence of zero or more *path selectors*, ρ, which is either empty ($\pi \triangleq \epsilon$) or composed by appending a selector onto another path ($\pi \triangleq \pi' \cdot \rho$).

Definition 3.13 (Path Selector). A *path selector*, ρ, is always a dereference ($\rho \triangleq *$).

In the core calculus, paths always constitute sequences of zero or more dereferences (though for extensions can be more complex). For reference, we note the rightmost path selector corresponds with the *innermost* selection.[10] We can now capture the notion of when two paths conflict with each other. In the following, "$\mathtt{u} \triangleq \pi \mid \mathtt{x}$" denotes a *destructuring* of an lval $\mathtt{u}$ into its base ($\mathtt{x}$) and path (π):

Definition 3.14 (Path Conflict). Let $\mathtt{u} \triangleq \pi_{\mathtt{u}} \mid \mathtt{x}$ and $\mathtt{w} \triangleq \pi_{\mathtt{w}} \mid \mathtt{y}$ be lvals. Then, $\mathtt{w}$ is said to *conflict* with $\mathtt{u}$, denoted $\mathtt{u} \bowtie \mathtt{w}$, if $\mathtt{x} = \mathtt{y}$.

As such, path conflicts are fairly coarse-grained in the core calculus as, in essence, any paths involving the same variable conflict. For example, $\mathtt{x} \bowtie *\mathtt{x}$, $*\mathtt{x} \bowtie *\mathtt{x}$ and $\mathtt{x} \bowtie **\mathtt{x}$ all hold. However, for extensions to the core, more interesting conflicts are possible. For example, with tuples, a borrow of one element should not conflict with that of another element (i.e. $\mathtt{x.0} \bowtie \mathtt{x.1}$ should not hold).

Definition 3.15 (Type Containment). Let Γ be an environment where $\Gamma(\mathtt{x}) = \langle \tilde{\mathtt{T}} \rangle^{\mathtt{l}}$ for some $\mathtt{l}$. Then, $\Gamma \vdash \mathtt{x} \rightsquigarrow \mathtt{T_y}$ denotes that variable $\mathtt{x}$ *contains* type $\mathtt{T_y}$ and is defined as $\mathtt{contains}(\Gamma, \tilde{\mathtt{T}}, \mathtt{T_y})$ where:

$$\mathtt{contains}(\Gamma, \tilde{\mathtt{T}}, \mathtt{T_y}) = \begin{cases} \mathtt{contains}(\Gamma, \tilde{\mathtt{T}}', \mathtt{T_y}) & \textbf{if } \tilde{\mathtt{T}} = \square\tilde{\mathtt{T}}', \\ \mathtt{true} & \textbf{if } \tilde{\mathtt{T}} = \mathtt{T_y}, \\ \mathtt{false} & \textbf{otherwise}. \end{cases}$$

[10] For example, in the extension for tuples considered in §6.2, the path described in $(*\mathtt{x}).1$ would be "$\epsilon \cdot 1 \cdot *$", etc.

Type containment is about exploring the type of a given variable looking for a subcomponent (usually a borrow of some kind). Thus, $\Gamma \vdash \mathtt{x} \rightsquigarrow \&\mathtt{y}$ holds if $\Gamma(\mathtt{x}) = \langle \&\mathtt{y} \rangle^1$ or $\Gamma(\mathtt{x}) = \langle \Box\&\mathtt{y} \rangle^1$ or $\Gamma(\mathtt{x}) = \langle \Box\Box\&\mathtt{y} \rangle^1$, etc. Note, however, that containment ignores undefined types as these represent inactive portions of the environment. Thus, $\Gamma \vdash \mathtt{x} \rightsquigarrow \&\mathtt{y}$ is defined but does not hold if $\Gamma(\mathtt{x}) = \langle \lfloor \&\mathtt{y} \rfloor \rangle^1$. Using this, we finally define the concepts of being *read* and *write prohibited* as follows:

Definition 3.16 (Read Prohibited). In an environment Γ, an lval $\mathtt{w}$ is said to be *read prohibited*, denoted $\mathtt{readProhibited}(\Gamma, \mathtt{w})$, when some $\mathtt{x}$ exists where $\Gamma \vdash \mathtt{x} \rightsquigarrow \&\mathtt{mut}\ \overline{\mathtt{u}}$ and $\exists_{\mathtt{i}}(\mathtt{u}_{\mathtt{i}} \bowtie \mathtt{w})$.

Definition 3.17 (Write Prohibited). In an environment Γ, an lval $\mathtt{w}$ is said to be *write prohibited*, denoted $\mathtt{writeProhibited}(\Gamma, \mathtt{w})$, when either some $\mathtt{x}$ exists where $\Gamma \vdash \mathtt{x} \rightsquigarrow \&\overline{\mathtt{u}} \wedge \exists_{\mathtt{i}}(\mathtt{u}_{\mathtt{i}} \bowtie \mathtt{w})$ or $\mathtt{readProhibited}(\Gamma, \mathtt{w})$ holds.

An important requirement is the ability to model the effect of moving values out of lvals. The following illustrates a simple example:

$$\{\ \mathtt{let\ mut\ x = box\ 0;\ let\ mut\ y = x;}\ \}^1$$

In the last statement, the box moves from $\mathtt{x}$ to $\mathtt{y}$, and this is reflected in the type of $\mathtt{x}$ going from $\Box\mathtt{int}$ (before) to $\lfloor\Box\mathtt{int}\rfloor$ (after). The partial type $\lfloor\Box\mathtt{int}\rfloor$ signals a slot which *can* hold a box that refers to an integer, but which is currently *undefined*.

Another important aspect of Rust is that of *partial moves*. This arises when only part of a structure is moved (e.g. moving out one component of a tuple leaving the remainder intact). In the core calculus, only boxes support partial moves as the following illustrates:

$$\{\ \mathtt{let\ mut\ x = box\ box\ 0;\ let\ mut\ y = *x;}\ \}^1$$

At the end of this sequence, the typing environment would be $\{\mathtt{x} \mapsto \langle \Box\lfloor\Box\mathtt{int}\rfloor \rangle^1, \mathtt{y} \mapsto \langle \Box\mathtt{int} \rangle^1\}$. Again, $\mathtt{x}$ has a partial type indicating it refers to a box which can contain a value of type $\Box\mathtt{int}$ but is currently undefined. Although not possible in the core calculus, moving values out of partial types is permitted in some circumstances. For example, if tuples are added to the core, then the first component of $(\Box\mathtt{int}, \lfloor\mathtt{int}\rfloor)$ can be moved out (but not the second).

The $\mathtt{move}(\Gamma, \mathtt{w})$ function is responsible for determining the environment after the value of an lval $\mathtt{w}$ is moved out:

Definition 3.18 (Move). Let Γ be an environment where $\Gamma(\mathtt{x}) = \langle \tilde{\mathtt{T}}_1 \rangle^1$ for some lifetime $\mathtt{l}$, and $\mathtt{w}$ an lval where $\mathtt{w} \triangleq \pi_{\mathtt{x}} \mid \mathtt{x}$. Then, $\mathtt{move}(\Gamma, \mathtt{w})$ is a partial function defined as $\Gamma[\mathtt{x} \mapsto \langle \tilde{\mathtt{T}}_2 \rangle^1]$ where $\tilde{\mathtt{T}}_2 = \mathtt{strike}(\pi_{\mathtt{x}} \mid \tilde{\mathtt{T}}_1)$:

$$\begin{array}{rcl} \mathtt{strike}(\epsilon \mid \mathtt{T}) & = & \lfloor \mathtt{T} \rfloor \\ \mathtt{strike}\big((\pi \cdot *) \mid \Box\tilde{\mathtt{T}}_1\big) & = & \Box\tilde{\mathtt{T}}_2 \quad \textbf{where}\ \tilde{\mathtt{T}}_2 = \mathtt{strike}(\pi \mid \tilde{\mathtt{T}}_1) \end{array}$$

Here, the syntax $(\pi \cdot *)$ indicates a path containing at least one dereference, whilst the pattern matching syntax "$\pi \mid \mathtt{T}$" represents a path being applied to a type and is used for "unravelling" lvals. For example, if $\Gamma(\mathtt{x}) = \langle \Box\mathtt{int} \rangle^1$ then $\mathtt{move}(\Gamma, *\mathtt{x})$ reduces first to $\mathtt{strike}(* \mid \Box\mathtt{int})$ and then $\Box\mathtt{strike}(\mathtt{int})$ before finally giving $\Box\lfloor\mathtt{int}\rfloor$. Observe there is no case above for handling borrows which simply reflects that one cannot move out of a borrow in Rust.

3.4.2 Expressions. We now consider the typing rules for expressions in FR. The first rule handles the unit value, integer constants and reference values using the store typing as necessary:

$$\frac{\sigma \vdash \mathtt{v} : \mathtt{T}}{\Gamma \vdash \langle\ \mathtt{v} : \mathtt{T}\ \rangle^1_\sigma \dashv \Gamma} \qquad \text{(T-Const)}$$

Rule T-Copy handles copying out of an lval. Amongst other things, to maintain the ownership invariant, the type in question must have copy semantics:

$$\frac{\Gamma \vdash \mathtt{w} : \langle \mathtt{T} \rangle^{\mathtt{m}} \quad \mathtt{copy(T)} \quad \neg\mathtt{readProhibited}(\Gamma, \mathtt{w})}{\Gamma \vdash \langle\ \hat{\mathtt{w}} : \mathtt{T}\ \rangle^{1}_{\sigma} \dashv \Gamma} \qquad \text{(T-Copy)}$$

This rule is fairly straightforward and simply returns the type of the lval in question without updating the environment. Observe that only fully defined types (i.e. those which do not have undefined components) can be copied. Rule T-Copy ignores the lifetime determined for w since it represents a copy and, hence, the value returned will take on a new lifetime. Furthermore, w cannot be read prohibited (i.e. mutably borrowed) as, otherwise, this could break the invariant that mutable borrows have unique access. This contrasts with the rule for moves:

$$\frac{\Gamma \vdash \mathtt{w} : \langle \mathtt{T} \rangle^{\mathtt{m}} \quad \neg\mathtt{writeProhibited}(\Gamma_1, \mathtt{w}) \qquad \Gamma_2 = \mathtt{move}(\Gamma_1, \mathtt{w})}{\Gamma_1 \vdash \langle\ \mathtt{w} : \mathtt{T}\ \rangle^{1}_{\sigma} \dashv \Gamma_2} \qquad \text{(T-Move)}$$

This is similar to before but requires w is not write prohibited (i.e. borrowed), and captures the move by (effectively) removing w from the resulting environment. More specifically, the type of w is relegated to its corresponding undefined type (i.e. $\lfloor \mathtt{T} \rfloor$). This indicates w is no longer live whilst retaining the necessary structural information about its slot. For example, if we simply removed w from the resulting environment altogether then subsequent reassignments, such as in the following, would be rejected:

$$\{\ \mathtt{let\ mut\ x = 0;\ let\ mut\ y = x;\ \ldots\ ; x = 1;}\ \}^{1}$$

After the second statement above, any attempts to use x prior to its reassignment are rendered impossible as the environment carries only the undefined "shadow" of its type, rather than its actual type. This shadow, firstly, indicates that x is a declared variable and, secondly, allows us to prohibit *incompatible* assignments (more on this later).

Rule T-MutBorrow for mutable borrowing requires lval w is not *write* prohibited, whilst T-ImmBorrow requires only that it is not *read* prohibited:

$$\frac{\Gamma \vdash \mathtt{w} : \langle \mathtt{T} \rangle^{\mathtt{m}} \quad \mathtt{mut}(\Gamma_1, \mathtt{w}) \quad \neg\mathtt{writeProhibited}(\Gamma, \mathtt{w})}{\Gamma \vdash \langle\ \mathtt{\&mut\ w} : \mathtt{\&mut\ w}\ \rangle^{1}_{\sigma} \dashv \Gamma}\ \text{(T-MutBorrow)} \qquad \frac{\Gamma \vdash \mathtt{w} : \langle \mathtt{T} \rangle^{\mathtt{m}} \quad \neg\mathtt{readProhibited}(\Gamma, \mathtt{w})}{\Gamma \vdash \langle\ \mathtt{\&w} : \mathtt{\&w}\ \rangle^{1}_{\sigma} \dashv \Gamma}\ \text{(T-ImmBorrow)}$$

Observe that both rules above require lval w to have a defined type (i.e. not a partial type). This reflects the fact that, in Rust, one cannot borrow a partial type. Furthermore, T-MutBorrow requires mutable access to the location being borrowed. This is determined as follows:

Definition 3.19 (Mutable). Let Γ be an environment where $\Gamma(\mathtt{x}) = \langle \tilde{\mathtt{T}} \rangle^{\mathtt{l}}$ for some lifetime l, and w an lval where $\mathtt{w} \triangleq \pi_{\mathtt{x}} \mid \mathtt{x}$. Then, $\mathtt{mut}(\Gamma, \mathtt{w})$ is a partial function defined as $\mathtt{mutable}(\Gamma, \pi_{\mathtt{x}} \mid \tilde{\mathtt{T}})$ that determines whether w is *mutable*:

$$\begin{array}{rcl} \mathtt{mutable}(\Gamma, \epsilon \mid \mathtt{T}) & = & \mathtt{true} \\ \mathtt{mutable}(\Gamma, (\pi \cdot *) \mid \square\mathtt{T}) & = & \mathtt{mutable}(\Gamma, \pi \mid \mathtt{T}) \\ \mathtt{mutable}(\Gamma, (\pi \cdot *) \mid \mathtt{\&mut}\ \overline{\mathtt{w}}) & = & \bigwedge_{\mathtt{i}} \mathtt{mut}(\Gamma, \pi \cdot \mathtt{w_i}) \end{array}$$

In essence this requires that, for a given lval, the path it describes never traverses an immutable borrow. Since variables are always declared mutable in FR (e.g. "`let mut x = 0`"), an lval can only be immutable if it involves an immutable borrow. Also, whilst $\mathtt{mut}(\Gamma, \mathtt{w})$ is concerned solely with w, $\mathtt{writeProhibited}(\Gamma, \mathtt{w})$ is concerned with externalities (i.e. borrows) which *restrict* w. In other

words, $\texttt{mut}(\Gamma, \texttt{w})$ protects against obtaining *mutable access through an immutable borrow*, whilst $\texttt{writeProhibited}(\Gamma, \texttt{w})$ prevents *mutable access to something borrowed*. The following illustrates:

$$\{\ \texttt{let mut x = 0; let mut y = \&x; let mut p = ...}\ \}^1$$

Here, initialising p with "&mut *y" is prohibited since we have $\neg\texttt{mut}(\Gamma, *\texttt{y})$ whilst "&mut x" is prohibited since we have $\texttt{writeProhibited}(\Gamma, \texttt{x})$, where $\Gamma = \{\texttt{x} \mapsto \langle \texttt{int} \rangle^1, \texttt{y} \mapsto \langle \&\texttt{x} \rangle^1\}$.

Dynamic allocation is handled by returning a box type, which represents an owned pointer (rather than a borrowed reference) to a location dynamically allocated in the heap:

$$\frac{\Gamma_1 \vdash \langle\ \texttt{t} : \texttt{T}\ \rangle^1_\sigma \dashv \Gamma_2}{\Gamma_1 \vdash \langle\ \texttt{box t} : \Box\texttt{T}\ \rangle^1_\sigma \dashv \Gamma_2} \quad \text{(T-Box)}$$

We see here that the effect of evaluating term t is propagated outwards. Thus, for example, if t results in some variable being moved then "box t" includes this effect.

3.4.3 Statements. We now consider typing rules for statements in FR:

$$\frac{\Gamma_1 \vdash \langle\ \texttt{t}_1 : \texttt{T}_1\ \rangle^1_\sigma \dashv \Gamma_2 \quad \ldots \quad \Gamma_n \vdash \langle\ \texttt{t}_n : \texttt{T}_n\ \rangle^1_\sigma \dashv \Gamma_{n+1}}{\Gamma_1 \vdash \langle\ \overline{\texttt{t}} : \texttt{T}_n\ \rangle^1_\sigma \dashv \Gamma_{n+1}} \quad \text{(T-Seq)}$$

For a sequence of terms, the environment generated after each is simply fed into the next. Observe that the type of a sequence is determined by the final term (which, for example, might be ϵ for a statement). The rule for handling blocks relies on T-Seq for handling the body and exploits the unique lifetime associated with a given block to determine which variables should be dropped:

$$\frac{\Gamma_1 \vdash \langle\ \overline{\texttt{t}} : \texttt{T}\ \rangle^{\texttt{m}}_\sigma \dashv \Gamma_2 \quad \Gamma_2 \vdash \texttt{T} \geq \texttt{l} \quad \Gamma_3 = \texttt{drop}(\Gamma_2, \texttt{m})}{\Gamma_1 \vdash \langle\ \{\overline{\texttt{t}}\}^{\texttt{m}} : \texttt{T}\ \rangle^1_\sigma \dashv \Gamma_3} \quad \text{(T-Block)}$$

Definition 3.20 (Environment Drop). The *environment drop* deallocates locations by removing them from an environment, Γ, as follows: $\texttt{drop}(\Gamma, \texttt{m}) = \Gamma - \{\texttt{x} \mapsto \langle \tilde{\texttt{T}} \rangle^{\texttt{m}} \mid \texttt{x} \mapsto \langle \tilde{\texttt{T}} \rangle^{\texttt{m}} \in \Gamma\}$.

In T-Block, the requirement, $\Gamma_2 \vdash \texttt{T} \geq \texttt{l}$, forms the closest thing to a subtyping requirement in the calculus and states that T is *well-formed* with respect to lifetime l. More specifically, that every borrowed reference contained in T lives *at least as long* as lifetime l. Here, we are effectively subtyping over lifetimes, but not over types in general. In fact, at the time of writing, Rust itself does not support any form of subtyping other than for lifetimes.

Definition 3.21 (Well-Formed Type). For an environment Γ, a type T is said to be well-formed with respect to a lifetime l, denoted $\Gamma \vdash \texttt{T} \geq \texttt{l}$, according to the following rules:

$$\frac{}{\Gamma \vdash \texttt{int} \geq \texttt{l}} \text{ (L-Int)} \qquad \frac{\overline{\Gamma \vdash \texttt{u} : \langle \texttt{T} \rangle^{\texttt{m}}} \quad \overline{\texttt{m} \geq \texttt{l}}}{\Gamma \vdash \&[\texttt{mut}]\ \overline{\texttt{u}} \geq \texttt{l}} \text{ (L-Borrow)} \qquad \frac{\Gamma \vdash \texttt{T} \geq \texttt{l}}{\Gamma \vdash \Box\texttt{T} \geq \texttt{l}} \text{ (L-Box)}$$

Variable declarations capture the creation of a new (owned) location whose lifetime matches that of the enclosing block.

$$\frac{\texttt{x} \notin \mathbf{dom}(\Gamma_1) \qquad \Gamma_1 \vdash \langle\ \texttt{t} : \texttt{T}\ \rangle^1_\sigma \dashv \Gamma_2 \qquad \Gamma_3 = \Gamma_2[\texttt{x} \mapsto \langle \texttt{T} \rangle^1]}{\Gamma_1 \vdash \langle\ \texttt{let mut x = t} : \epsilon\ \rangle^1_\sigma \dashv \Gamma_3} \quad \text{(T-Declare)}$$

Handling of assignments is trickier. For example, $\texttt{mut}(\Gamma, \texttt{w})$ must be implied as a location cannot be assigned whilst borrowed (i.e. it is frozen for the duration to protect against dangling references).

$$\frac{\Gamma_1 \vdash \mathtt{w} : \langle \widetilde{\mathtt{T}}_1 \rangle^{\mathtt{m}} \quad \Gamma_1 \vdash \langle\, \mathtt{t} : \mathtt{T}_2 \,\rangle^{1}_{\sigma} \dashv \Gamma_2 \quad \Gamma_2 \vdash \widetilde{\mathtt{T}}_1 \approx \mathtt{T}_2 \quad \Gamma_2 \vdash \mathtt{T}_2 \geq \mathtt{m} \quad \Gamma_3 = \mathtt{write}^{0}(\Gamma_2, \mathtt{w}, \mathtt{T}_2) \quad \neg \mathtt{writeProhibited}(\Gamma_3, \mathtt{w})}{\Gamma_1 \vdash \langle\, \mathtt{w} = \mathtt{t} : \epsilon \,\rangle^{1}_{\sigma} \dashv \Gamma_3} \quad \text{(T-Assign)}$$

Observe that, unlike other rules, w is permitted to have a partial type (i.e. since it is being overwritten anyway). Here, the importance of the undefined "shadow" of a type comes into play as these ensure reassignments are *compatible*. For example, the following is not permitted in Rust:

$$\{\ \mathtt{let\ mut\ x = 0;\ let\ mut\ y = x;\ \ldots\ ; x = \&y;\ }\}^{1}$$

This is prohibited because the declared type of x is not compatible with that being reassigned (i.e. &y). Catching this requires knowledge of the original type given to x by its declaration and this is achieved by retaining its shadow. In rule T-Assign, we refer to $\mathtt{T}_1 \approx \mathtt{T}_2$ as a *shape requirement*. Specifically, that $\mathtt{T}_1$ and $\mathtt{T}_2$ have compatible shape which, in Rust, is primarily about ensuring identical memory layouts. For example, &x is compatible with &y provided the type of x is compatible with that of y. And yet, at the same time, the following is rejected by the Rust compiler:

```
fn main() { let mut y = Box::new(0); { let mut z = 1; y = &mut z; } }
```

This gives an error "`expected struct `std::boxed::Box`, found mutable reference`" in Rust which amounts to saying that &mut z is incompatible with □int.[11]

Definition 3.22 (Compatible Shape). For an environment Γ, two partial types $\widetilde{\mathtt{T}}_1$ and $\widetilde{\mathtt{T}}_2$ are said to be *shape compatible*, denoted as $\Gamma \vdash \widetilde{\mathtt{T}}_1 \approx \widetilde{\mathtt{T}}_2$, according to the following rules:

$$\frac{}{\Gamma \vdash \mathtt{int} \approx \mathtt{int}}\ \text{(S-Int)} \qquad \frac{\Gamma \vdash \widetilde{\mathtt{T}}_1 \approx \widetilde{\mathtt{T}}_2}{\Gamma \vdash \square\widetilde{\mathtt{T}}_1 \approx \square\widetilde{\mathtt{T}}_2}\ \text{(S-Box)} \qquad \frac{\forall_{i,j}.\big(\Gamma \vdash \mathtt{u}_i : \widetilde{\mathtt{T}}_1 \approx \widetilde{\mathtt{T}}_2 : \mathtt{w}_j \dashv \Gamma\big)}{\Gamma \vdash \&[\mathtt{mut}]\ \overline{\mathtt{u}} \approx \&[\mathtt{mut}]\ \overline{\mathtt{w}}}\ \text{(S-Bor)}$$

$$\frac{\Gamma \vdash \mathtt{T}_1 \approx \widetilde{\mathtt{T}}_2}{\Gamma \vdash \lfloor \mathtt{T}_1 \rfloor \approx \widetilde{\mathtt{T}}_2}\ \text{(S-UndefL)} \qquad \frac{\Gamma \vdash \widetilde{\mathtt{T}}_1 \approx \mathtt{T}_2}{\Gamma \vdash \widetilde{\mathtt{T}}_1 \approx \lfloor \mathtt{T}_2 \rfloor}\ \text{(S-UndefR)}$$

Here, S-Bor requires the same mutability on both sides and, hence, both $\Gamma \vdash \&\mathtt{x} \approx \&\mathtt{y}$ and $\Gamma \vdash \&\mathtt{mut\ x} \approx \&\mathtt{mut\ y}$ can hold (depending on x and y), but never $\Gamma \vdash \&\mathtt{mut\ x} \approx \&\mathtt{y}$. Two rules are also given for handling undefined types which simply check compatibility of underlying types. This follows given the primary purpose of retaining the undefined "shadows" is to enable exactly this. Finally, we note that compatibility does not consider lifetimes at all and, in fact, borrows can be compatible even when the locations to which they refer have different lifetimes.

Considering rule T-Assign again, the type of lval w is updated via a dedicated $\mathtt{write}^{0}(\Gamma, \mathtt{w}, \mathtt{T})$ function. This is necessary to ensure its type reflects the lifetime of any borrows it now holds. The following illustrates:

$$\{\mathtt{let\ mut\ x = 0;\ \{let\ mut\ y = 1;\ \{let\ mut\ z = \&mut\ y;\ z = \&mut\ x;\}^{n}\ \}^{m}\ \}}^{1} \tag{17}$$

Here, variable z initially has type "&mut y" which, after the final assignment, becomes "&mut x". This *retyping* is critical because z now refers to a location with a different lifetime. In this case, the movement of lifetimes goes from inner to outer; however, the direction of movement does not ultimately matter, provided z remains inside. The $\mathtt{write}^{k}(\Gamma, \mathtt{w}, \mathtt{T})$ function is defined as follows:

[11] The need for this distinction presumably arises because of their differing behaviour in certain respects. For example, dropping a Box<T> requires deallocating memory, whilst for a mutable borrow it does not. Likewise, Box<T> is covariant whilst a mutable borrow is invariant.

Definition 3.23 (Write). Let Γ be an environment where $\Gamma(\mathtt{x}) = \langle \tilde{\mathtt{T}}_1 \rangle^{\mathtt{l}}$ for some lifetime $\mathtt{l}$ and lval $\mathtt{w}$ where $\mathtt{w} \triangleq \pi_{\mathtt{x}} \mid \mathtt{x}$. Then, $\mathtt{write}^{\mathtt{k}}(\Gamma, \mathtt{w}, \mathtt{T})$ is a partial function defined as $\Gamma_2[\mathtt{x} \mapsto \langle \tilde{\mathtt{T}}_2 \rangle^{\mathtt{l}}]$ for some rank $\mathtt{k} \geq \mathtt{0}$ where $(\Gamma_2, \tilde{\mathtt{T}}_2) = \mathtt{update}^{\mathtt{k}}(\Gamma, \pi_{\mathtt{x}} \mid \tilde{\mathtt{T}}_1, \mathtt{T})$:

$$
\begin{array}{rcll}
\mathtt{update}^{\mathtt{0}}(\Gamma, \epsilon \mid \tilde{\mathtt{T}}_1, \mathtt{T}_2) & = & (\Gamma, \mathtt{T}_2) & \\
\mathtt{update}^{\mathtt{k}\geq\mathtt{1}}(\Gamma, \epsilon \mid \mathtt{T}_1, \mathtt{T}_2) & = & (\Gamma, \mathtt{T}_1 \sqcup \mathtt{T}_2) & \\
\mathtt{update}^{\mathtt{k}}(\Gamma_1, (\pi \cdot *) \mid \square\tilde{\mathtt{T}}_1, \mathtt{T}) & = & (\Gamma_2, \square\tilde{\mathtt{T}}_2) & \textbf{where } \overline{(\Gamma_2, \tilde{\mathtt{T}}_2) = \mathtt{update}^{\mathtt{k}}(\Gamma_1, \pi \mid \tilde{\mathtt{T}}_1, \mathtt{T})} \\
\mathtt{update}^{\mathtt{k}}(\Gamma, (\pi \cdot *) \mid \&\mathtt{mut}\ \overline{\mathtt{u_i}}, \mathtt{T}) & = & (\bigsqcup_{\mathtt{i}} \Gamma_{\mathtt{i}}, \&\mathtt{mut}\ \overline{\mathtt{u_i}}) & \textbf{where } \overline{\Gamma_{\mathtt{i}} = \mathtt{write}^{\mathtt{k}+\mathtt{1}}(\Gamma, \pi \mid \mathtt{u_i}, \mathtt{T})}
\end{array}
$$

The two cases for variables and boxes are straightforward at rank $\mathtt{0}$. For example, we have $\mathtt{write}^{\mathtt{0}}(\{\mathtt{x} \mapsto \langle \lfloor \mathtt{int} \rfloor \rangle^{\mathtt{1}}\}, \mathtt{x}, \mathtt{int}) = \{\mathtt{x} \mapsto \langle \mathtt{int} \rangle^{\mathtt{1}}\}, \mathtt{write}^{\mathtt{0}}(\{\mathtt{x} \mapsto \langle \square \lfloor \mathtt{int} \rfloor \rangle^{\mathtt{1}}\}, *\mathtt{x}, \mathtt{int}) = \{\mathtt{x} \mapsto \langle \square \mathtt{int} \rangle^{\mathtt{1}}\}$. We refer to these cases as applying a *strong update* because they wholly replace the type of the assigned location.[12] The case for mutable borrows is more subtle as, in the general case, one cannot safely apply a strong update. To understand this, let us consider the interpretation of $\Gamma(\mathtt{p}) = \langle \&\mathtt{mut}\ \mathtt{x}, \mathtt{y} \rangle^{\mathtt{1}}$. This should not be read as saying $\mathtt{p}$ refers to $\mathtt{x}$ *and* $\mathtt{y}$ (since this is not physically possible). Rather, it must be read as as saying $\mathtt{p}$ refers to $\mathtt{x}$ *or* $\mathtt{y}$. Following this, assigning through $\mathtt{p}$ updates either $\mathtt{x}$ or $\mathtt{y}$ (though we cannot tell which) and, hence, we must conservatively retain their original types. Furthermore, we note that borrowed locations cannot have partial types and, hence, the terminating case for rank $\mathtt{k} \geq \mathtt{1}$ need not consider them. Finally, observe that $\mathtt{mut}(\Gamma, \mathtt{w})$ is implied when $\mathtt{write}^{\mathtt{0}}(\Gamma, \mathtt{w}, \mathtt{T})$ is defined (i.e. since it has no case for immutable borrows).

An interesting question is how a type such as $\langle \&\mathtt{mut}\ \mathtt{x}, \mathtt{y} \rangle^{\mathtt{1}}$ arises in practice. If the calculus is extended with conditional control-flow, then such types can easily arise from assignments on either side of a conditional. However, the core calculus has no notion of control-flow and, as a result, such types do not *have* to arise. More specifically, if $\mathtt{write}^{\mathtt{k}}(\Gamma, \mathtt{w}, \mathtt{T})$ were modified so as to perform a strong update when assigning through a borrow with only one referent, then multi-referent borrows could not be created. Instead, our choice above follows Rust which does not perform strong updates in such situations. For example, the following is rejected by the Rust compiler:

```
let mut x = 1;
let mut y = 2;
let mut p = &mut x;
let mut q = &mut p;
*q = &mut y;
x + *p
```

In principle, this could be safely accepted by the Rust compiler since the assignment through $*\mathtt{q}$ must overwrite the value held by $\mathtt{p}$. The reason this is not supported remains unclear, but it seems likely that such situations arise rarely in practice.

Finally, the above definition for $\mathtt{write}^{\mathtt{k}}(\Gamma, \mathtt{w}, \mathtt{T})$ guarantees to produce a type (i.e. not a partial type). However, extensions to the core can produce partial types. For example, "$(\lfloor \mathtt{int} \rfloor, \lfloor \mathtt{int} \rfloor)$" could become "$(\mathtt{int}, \lfloor \mathtt{int} \rfloor)$", etc.

4 SOUNDNESS

We now present the main result obtained for FR, namely that *type and borrow safe* programs do not get stuck and preserve the borrowing invariant (i.e. borrows outlive their referents, etc). Since programs in FR refer to locations which may be deallocated, this result implies that *type and borrow safe* programs do not attempt to dereference dangling pointers. Traditionally, type soundness

[12] We reuse terminology from the literature on pointer analysis here [81, 102].

(e.g. for the simply-typed lambda calculus) is split into the so-called *progress* and *preservation* lemmas [104]. However, since our type system is formulated in a flow-sensitive fashion, we must adapt these theorems for our setting. Note, proofs for the following lemmas and theorems can be found in the Appendix.

4.1 Valid States

A key precursor to establishing progress and preservation is the notation of a *valid state*. For example, the state $\{\ell_1 \mapsto \langle 1 \rangle^1, \ell_2 \mapsto \langle \ell_1^\bullet \rangle^1\} \triangleright \{\ \texttt{let mut x} = \ell_1^\bullet\ \}^1$ should be considered invalid because there are two owning references to ℓ_1 in play. Starting from this state, there is no hope to establish progress or preservation!

Definition 4.1 (Valid Term). Let $\texttt{t}$ be a term where $\overline{\texttt{v}} \in \texttt{t}$ is the sequence of distinct values it contains. Then, $\texttt{t}$ is *well-formed* if $\neg\exists_{\texttt{i},\texttt{j}}.\big(\texttt{i} \neq \texttt{j} \wedge \exists_{\ell^\bullet}.(\texttt{v}_\texttt{i} = \texttt{v}_\texttt{j} = \ell^\bullet)\big)$.

The above notion essentially says that no valid term can containing distinct owning references to the same location. In fact, in the calculus core, this is only possible for blocks (e.g. "$\{\ell^\bullet; \ell^\bullet\}$" is not valid). Extensions to the core may introduce additional compound types (e.g. tuples) where this can arise in a more critical fashion. We note also that source-level terms (recall §3.1) are never invalid as, by definition, they cannot involve reference values. In a similar fashion, we can define the notion of a valid program store:

Definition 4.2 (Valid Store). Let $\mathcal{S}$ be a program store where $\overline{\texttt{v}} \in \texttt{t}$ is the sequence of distinct values contained in any $\ell \in \mathbf{dom}(\mathcal{S})$. Then, $\mathcal{S}$ is said to be *valid* when $\neg\exists_{\texttt{i},\texttt{j}}.\big(\texttt{i} \neq \texttt{j} \wedge \exists_{\ell^\bullet}.(\texttt{v}_\texttt{i} = \texttt{v}_\texttt{j} = \ell^\bullet)\big)$.

The intuition here is, as before, that a valid store cannot hold distinct owning references to the same location. Thus, for example, $\{\ell_1 \mapsto \langle 1 \rangle^1, \ell_2 \mapsto \langle \ell_1^\bullet \rangle^1, \ell_3 \mapsto \langle \ell_1^\bullet \rangle^1\}$ is an invalid store. Finally, we can bring these two notions together to define the concept of a valid state:

Definition 4.3 (Valid State). Let $\mathcal{S} \triangleright \texttt{t}$ be a program state where both $\mathcal{S}$ and $\texttt{t}$ are valid. Let $\overline{\texttt{v}} \in \texttt{t}$ and $\overline{\texttt{u}} \in \texttt{t}$ be the sequence of distinct values contained in (respectively) $\mathcal{S}$ and $\texttt{t}$. Then, $\mathcal{S} \triangleright \texttt{t}$ is valid when $\neg\exists_{\texttt{i},\texttt{j}}.\big(\texttt{i} \neq \texttt{j} \wedge \exists_{\ell^\bullet}.(\texttt{v}_\texttt{i} = \texttt{u}_\texttt{j} = \ell^\bullet)\big)$.

The rough intuition here is that a well-typed term, when starting from a valid state, is guaranteed to eventually reduce to a value. In particular, "$\emptyset \vdash \texttt{t}$" is a valid state when $\texttt{t}$ is a source-level term and, hence, should reduce to a value when $\texttt{t}$ is well typed. To show this, however, requires establishing further connections between program stores and typing environments.

4.2 Safe Abstractions

Another important property is the connection between runtime program stores and typing environments. That is, the typing environment determined for a given program point should always be a *safe abstraction* of any possible program store at that point. For example, the typing environment $\{\texttt{x} \mapsto \langle \texttt{\&mut y} \rangle^1, \texttt{y} \mapsto \langle \texttt{int} \rangle^1\}$ does not safely abstract the program store $\{\ell_\texttt{x} \mapsto \langle 1 \rangle^1\}$ for two reasons: firstly, the type of $\texttt{x}$ does not correspond with its runtime value; secondly, variable $\texttt{y}$ has a type but does not actually exist! We now progressively build up the machinery necessary for establishing when a typing environment safely abstracts a runtime program store.

Definition 4.4 (Valid Type). Let $\mathcal{S}$ be a program store, $\texttt{v}^\perp$ a partial value and $\tilde{\texttt{T}}$ a partial type. Then, $\texttt{v}^\perp$ is abstracted by $\tilde{\texttt{T}}$ in $\mathcal{S}$, denoted $\mathcal{S} \vdash \texttt{v}^\perp \sim \tilde{\texttt{T}}$, according to the following rules:

$$\frac{}{\mathcal{S} \vdash \epsilon \sim \epsilon}\ \text{(V-Unit)} \qquad \frac{}{\mathcal{S} \vdash \mathtt{c} \sim \mathtt{int}}\ \text{(V-Int)} \qquad \frac{}{\mathcal{S} \vdash \bot \sim \lfloor \mathtt{T} \rfloor}\ \text{(V-Undef)}$$

$$\frac{\exists_{\mathtt{i}}.\big(\mathtt{loc}(\mathcal{S}, \mathtt{w_i}) = \ell\big)}{\mathcal{S} \vdash \ell^{\circ} \sim \&[\mathtt{mut}]\ \overline{\mathtt{w}}}\ \text{(V-Borrow)} \qquad \frac{\mathcal{S}(\ell^{\bullet}) = \langle \mathtt{v}^{\bot} \rangle^{\mathtt{l}} \quad \mathcal{S} \vdash \mathtt{v}^{\bot} \sim \widetilde{\mathtt{T}}}{\mathcal{S} \vdash \ell^{\bullet} \sim \Box \widetilde{\mathtt{T}}}\ \text{(V-Box)}$$

Here, V-Borrow conservatively requires the borrowed reference matches *one* of the borrowed locations, but not all (i.e. since this is physically impossible). We extend this notion to store typings as needed for typing runtime values (i.e. those which can only arise during execution):

Definition 4.5 (Valid Store Typing). Let $\mathcal{S}$ be a program store, σ a store typing and $\mathtt{t}$ a term where $\overline{\mathtt{v}} \in \mathtt{t}$ is the sequence of distinct values it contains. Then, σ is valid for state $\mathcal{S} \triangleright \mathtt{t}$, denoted $\mathcal{S} \triangleright \mathtt{t} \vdash \sigma$, if $\forall \mathtt{i}.\big(\mathcal{S} \vdash \mathtt{v_i} \sim \sigma(\mathtt{v_i})\big)$.

Recall from §3.3 that the store typing is needed for typing reference values which arise during the execution of a term. Thus, as corollary, we have $\mathcal{S} \triangleright \mathtt{t} \vdash \emptyset$ for any source-level term $\mathtt{t}$. In other words, that any source-level term can be typed without the need for a store typing. We can now develop the notion of a *safe abstraction*, where $\mathcal{L}$ represents the set of all heap locations ℓ_n:

Definition 4.6 (Variable Projection). For a set of variable identifiers, ϕ, $\Theta(\phi) = \{\ell_{\mathtt{x}} \mid \mathtt{x} \in \phi\}$.

Definition 4.7 (Safe Abstraction). Let Γ be a typing environment and $\mathcal{S}$ a program store. Then, $\mathcal{S}$ is *safely abstracted* by Γ, denoted $\mathcal{S} \sim \Gamma$, iff $(\mathbf{dom}(\mathcal{S}) - \mathcal{L}) = \Theta(\mathbf{dom}(\Gamma))$ and for all $\mathtt{x} \in \mathbf{dom}(\Gamma)$ we have $\big(\mathcal{S} \vdash \mathtt{v}^{\bot} \sim \widetilde{\mathtt{T}}\big)$ where $\mathcal{S}(\ell_{\mathtt{x}}) = \langle \mathtt{v}^{\bot} \rangle^{\mathtt{l}}$ and $\Gamma(\mathtt{x}) = \langle \widetilde{\mathtt{T}} \rangle^{\mathtt{l}}$.

Definition 4.7 says that a runtime program store is safely abstracted by a typing environment if they describe the same set of variables (ignoring heap locations) and where the runtime value for every variable has a valid type. The projection, $\Theta(\cdot)$, maps variable identifiers to their corresponding locations. Also, the reason for ignoring heap locations here is that they are not reflected in the typing environment, but are present in the program store. For example, a type $\Box\mathtt{T}$ in Γ effectively accounts for one heap location in $\mathcal{S}$. We note the notion of a safe abstraction permits unreachable heap locations to remain. In other words, we are not enforcing that a well-typed program reclaims all dynamically allocated memory and, instead, enforce the weaker condition that, for those locations which were reclaimed, it was safe to do so.

4.3 Borrow Invariance

An important invariant over typing environments is that every borrow is for a valid lval (i.e. is not dangling). We refer to this as the *borrow invariant* and capture it in the *well-formedness* property over environments (recall type containment from Definition 3.15):

Definition 4.8 (Well-Formed Environment). A typing environment Γ is *well-formed with respect to some lifetime* $\mathtt{l}$ iff: (i) for all $\mathtt{x} \in \mathbf{dom}(\Gamma)$ and $\overline{\mathtt{w}} \in \mathtt{LVal}$ where $\Gamma \vdash \mathtt{x} \rightsquigarrow \&[\mathtt{mut}]\ \overline{\mathtt{w}} \wedge \Gamma(\mathtt{x}) = \langle \cdot \rangle^{\mathtt{n}}$ we have $\overline{\Gamma \vdash \mathtt{w} : \langle \mathtt{T} \rangle^{\mathtt{m}} \wedge \mathtt{m} \geq \mathtt{n}}$; (ii) for all $\mathtt{x} \in \mathbf{dom}(\Gamma)$ where $\Gamma(\mathtt{x}) = \langle \cdot \rangle^{\mathtt{n}}$ we have $\mathtt{n} \geq \mathtt{l}$.

The first part of the well-formedness property simply ensures that every borrowed lval is well typed and has a lifetime that outlives the borrow. The second part of the well-formedness property ensures every slot lives as long as the given lifetime $\mathtt{l}$. To understand the need for this, consider a typing $\Gamma_1 \vdash \langle\ \mathtt{t} : \mathtt{T}\ \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2$ where $\Gamma_1(\mathtt{x}) = \langle \mathtt{T} \rangle^{\mathtt{m}}$ for some arbitrary $\mathtt{x}$. Clearly, if $\mathtt{l} \geq \mathtt{m}$ then something has gone wrong as this indicates state remains for a lifetime which no longer exists!

We can now present the *borrow invariance* lemma which establishes that typing a statement with a well-formed environment produces a well-formed environment:

Lemma 4.9 (Borrow Invariance). *Let $\mathcal{S}_1 \triangleright \mathtt{t}$ be a valid state; let σ be a store typing where $\mathcal{S}_1 \triangleright \mathtt{t} \vdash \sigma$; let Γ_1 be a well-formed typing environment with respect to a lifetime $\mathtt{l}$ where $\mathcal{S}_1 \sim \Gamma_1$ and Γ_2 be an arbitrary typing environment; let $\mathtt{t}$ be a term; and, let $\mathtt{T}$ be a type. If $\Gamma_1 \vdash \langle\ \mathtt{t} : \mathtt{T}\ \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2$, then $\Gamma_2[\gamma \mapsto \langle \mathtt{T} \rangle^{\mathtt{l}}]$ is well formed with respect to $\mathtt{l}$ for arbitrary $\gamma \in$ fresh.*

A subtle aspect of establishing borrow invariance is that the return value must be included. To understand this consider an assignment "$\mathtt{x} = \mathtt{t}$" and corresponding typing $\Gamma_1 \vdash \langle\ \mathtt{t} : \mathtt{T}\ \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2$. If Γ_1 is well-formed, does it follow that $\Gamma_2[\mathtt{x} \mapsto \langle \mathtt{T} \rangle^{\mathtt{l}}]$ is well-formed? This is important as, roughly speaking, it models the effect of an assignment. Unfortunately, well-formedness does not always follow here and there are two cases to consider. Firstly, if $\mathtt{x}$ is borrowed in Γ_2 (e.g. $\Gamma_2(\mathtt{y}) = \langle \&\mathtt{x} \rangle^{\mathtt{m}}$ for some $\mathtt{y}$) then $\Gamma_2[\mathtt{x} \mapsto \langle \mathtt{T} \rangle^{\mathtt{l}}]$ may break the borrowing invariant (e.g. if $\mathtt{m} \geq \mathtt{l}$). Fortunately, this case is easy to dismiss since we prohibit assignment to borrowed variables. Secondly, if $\mathtt{x}$ is assigned some borrow (e.g. $\mathtt{T} = \&\mathtt{y}$) to state which doesn't exist (i.e. has already been dropped) then $\Gamma_2[\mathtt{x} \mapsto \langle \mathtt{T} \rangle^{\mathtt{l}}]$ clearly breaks the borrowing invariant! For example, this could arise if a term such as $\mathtt{t} \triangleq \{\mathtt{let\ mut\ y = 0; \&y}\}^{\mathtt{l}}$ were permitted. Since this has no side effects we would have $\Gamma_1 \vdash \langle\ \mathtt{t} : \&\mathtt{y}\ \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2$ where $\Gamma_1 = \Gamma_2$ and, hence, well-formedness of Γ_2 would follow immediately from Γ_1. In short, looking only at the resulting environment is insufficient as it misses critical information from the type itself (i.e. $\&\mathtt{y}$). To address this, we must establish $\Gamma_2[\gamma \mapsto \langle \mathtt{T} \rangle^{\mathtt{l}}]$ is well-formed for some arbitrary $\gamma \in$ *fresh*. Here, γ represents an *anonymous* variable that, for example, cannot collide with other local variables. Our formulation may seem strange but, in saying $\Gamma_2[\gamma \mapsto \langle \mathtt{T} \rangle^{\mathtt{l}}]$ is well formed, we are saying that the "combination" of Γ_2 and $\mathtt{T}$ is well formed (in some sense).

4.4 Progress and Preservation

Using Definition 4.7 we can now present the *progress* lemma which establishes that a well-typed program which has not already terminated is guaranteed to execute at least one more step:

Lemma 4.10 (Progress). *Let $\mathcal{S}_1 \triangleright \mathtt{t}_1$ be a valid state; let σ be a store typing where $\mathcal{S}_1 \triangleright \mathtt{t}_1 \vdash \sigma$; let Γ_1 be a well-formed typing environment with respect to a lifetime $\mathtt{l}$ where $\mathcal{S}_1 \sim \Gamma_1$; let Γ_2 be a typing environment; and, let $\mathtt{T}$ be a type. If $\Gamma_1 \vdash \langle\ \mathtt{t}_1 : \mathtt{T}\ \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2$ then either $\mathtt{t}_1 \in \mathtt{Value}$ or $\langle\ \mathcal{S}_1 \triangleright \mathtt{t}_1 \longrightarrow \mathcal{S}_2 \triangleright \mathtt{t}_2\ \rangle^{\mathtt{l}}$ for some state $\mathcal{S}_2 \triangleright \mathtt{t}_2$.*

Accompanying the notion of progress is that of *preservation*. This is a guarantee that, having executed zero or more steps, a well-typed program remains well-typed.

Lemma 4.11 (Preservation). *Let $\mathcal{S}_1 \triangleright \mathtt{t}$ be a valid state and $\mathcal{S}_2 \triangleright \mathtt{v}$ a terminal state; let σ be a store typing where $\mathcal{S}_1 \triangleright \mathtt{t} \vdash \sigma$; let Γ_1 be a well-formed typing environment with respect to a lifetime $\mathtt{l}$ where $\mathcal{S}_1 \sim \Gamma_1$; let Γ_2 be a typing environment; and, let $\mathtt{T}$ be a type. If $\Gamma_1 \vdash \langle\ \mathtt{t} : \mathtt{T}\ \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2$ and $\langle\ \mathcal{S}_1 \triangleright \mathtt{t} \rightsquigarrow \mathcal{S}_2 \triangleright \mathtt{v}\ \rangle^{\mathtt{l}}$ then $\mathcal{S}_2 \triangleright \mathtt{v}$ remains valid where $\mathcal{S}_2 \sim \Gamma_2$ and $\mathcal{S}_2 \vdash \mathtt{v} \sim \mathtt{T}$.*

The key here is that we consider the reduction of an entire term at a time, which may involve zero or more steps (denoted by $\rightsquigarrow$) to reduce terms contained therein. This is necessitated by the flow-sensitive nature of our system as $\Gamma_1 \vdash \langle\ \mathtt{t} : \mathtt{T}\ \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2$ does not imply Γ_2 is a safe abstraction of all intermediate stores generated during the reduction of $\mathtt{t}$. For example, suppose $\mathtt{t}$ is $\{\mathtt{let\ mut\ x = 0}\}^{\mathtt{m}}$. Then, $\emptyset \vdash \mathtt{t} : \epsilon \dashv \emptyset$ but, clearly, some intermediate state $\mathcal{S}' = \{\ell_{\mathtt{x}} \mapsto \langle \mathtt{0} \rangle^{\mathtt{m}}\}$ exists where $\mathcal{S}' \not\sim \emptyset$.

Another key aspect of the preservation lemma is the handling of variables dropped (i.e. deallocated) as a result of assignments. For example, rule R-Blocks drops locations declared within. Since the function $\mathtt{drop}(\mathcal{S}, \psi)$ additionally deallocates any owned locations recursively (recall Definition 3.4), it must be shown that this doesn't drop something still needed. To this end, two sub-lemmas are employed (Drop Preservation and Update Preservation) to ensure the preservation of a safe abstraction across variable drops.

4.5 Type and Borrow Safety

Finally, we can now present the *type and borrow safety* theorem which establishes the property that a well-typed program is guaranteed to continue executing until a terminal state (i.e. a value) is reached. This theorem essentially follows trivially from Lemma 4.10 and Lemma 4.11.

Theorem 4.12 (Type and Borrow Safety). *Let $\mathcal{S}_1 \triangleright \mathtt{t}$ be a valid state; let σ be a store typing where $\mathcal{S}_1 \triangleright \mathtt{t} \vdash \sigma$; let Γ_1 be a well-formed typing environment with respect to a lifetime $\mathtt{l}$ where $\mathcal{S}_1 \sim \Gamma_1$; let Γ_2 be a typing environment; and, let $\mathtt{T}$ be a type. If $\Gamma_1 \vdash \langle\ \mathtt{t} : \mathtt{T}\ \rangle^1_\sigma \dashv \Gamma_2$, then $\langle\ \mathcal{S}_1 \triangleright \mathtt{t} \rightsquigarrow \mathcal{S}_2 \triangleright \mathtt{v}\ \rangle^1$ for some terminal state $\mathcal{S}_2 \triangleright \mathtt{v}$.*

We note the accompanying proofs for the main theorem and the borrow invariance, progress and preservation lemmas can be found in the Appendix.

4.5.1 Borrow Safety. A key observation is that Theorem 4.12 does not explicitly prohibit, for example, multiple mutable borrows of the same location arising. Consider a transition $\langle\ \mathcal{S}_1 \triangleright \mathtt{y} \longrightarrow \mathcal{S}_2 \triangleright \ell^\circ_\mathtt{x}\ \rangle^1$ where $\mathcal{S}_1 = \{\ell_\mathtt{x} \mapsto \langle \mathtt{0} \rangle^1, \ell_\mathtt{y} \mapsto \langle \ell^\circ_\mathtt{x} \rangle^1, \ell_\mathtt{z} \mapsto \langle \ell^\circ_\mathtt{x} \rangle^1\}$. Since immutable / mutable borrows are indistinguishable at runtime, one cannot tell whether $\mathcal{S}_1$ contains conflicting borrows to the same location (e.g. two mutable borrows, or one mutable and one immutable, etc). Indeed, the distinction between an immutable / mutable borrow is a strictly static notion. As such, one might expect to see the key properties regarding mutable borrows reflected in typing environments. Unfortunately this is not the case as, for example, when $\Gamma_1 = \{\mathtt{x} \mapsto \langle \mathtt{int} \rangle^1, \mathtt{y} \mapsto \langle \mathtt{\&mut\ x} \rangle^1, \mathtt{z} \mapsto \langle \mathtt{\&mut\ x} \rangle^1\}$, it holds that $\mathcal{S}_1 \sim \Gamma_1$ and Theorem 4.12 can apply. Instead, we require an additional property which captures our intuition about mutable borrows:

Definition 4.13 (Borrow Safe Environment). Let Γ be a well-formed program store with respect to some lifetime $\mathtt{l}$. Then, Γ is said to be *borrow safe* iff for all $\mathtt{x}, \mathtt{y} \in \mathbf{dom}(\Gamma)$ and for all $\overline{\mathtt{u}}, \overline{\mathtt{w}} \in \mathtt{LVal}$ where $\Gamma \vdash \mathtt{x} \rightsquigarrow \mathtt{\&mut}\ \overline{\mathtt{w}} \wedge \Gamma \vdash \mathtt{y} \rightsquigarrow \mathtt{\&[mut]}\ \overline{\mathtt{u}} \wedge \exists_{\mathtt{i},\mathtt{j}}.(\mathtt{w_i} \bowtie \mathtt{u_j})$ we have $\mathtt{x} = \mathtt{y}$.

This property simply ensures that at most one mutable borrow per lval can exist in a borrow safe environment. In fact, despite our observation that Γ_1 above does not meet this requirement, it remains the case that FR can guarantee borrow safety. More specifically, whilst we cannot make guarantees when starting from arbitrary program stores (as above), we can make guarantees from "safe" starting points (e.g. the empty store). We formalise this as the following corollary:

Corollary 4.14 (Borrow Safety). *Let $\mathcal{S}_1 \triangleright \mathtt{t}_1$ and $\mathcal{S}_2 \triangleright \mathtt{t}_2$ be valid states; let σ be a store typing where $\mathcal{S}_1 \triangleright \mathtt{t}_1 \vdash \sigma$; let Γ_1 be a well-formed borrow safe typing environment with respect to a lifetime $\mathtt{l}$ where $\mathcal{S}_1 \sim \Gamma_1$; let Γ_2 be a typing environment; and, let $\mathtt{T}_1, \mathtt{T}_2$ be types. If $\Gamma_1 \vdash \langle\ \mathtt{t}_1 : \mathtt{T}_1\ \rangle^1_\sigma \dashv \Gamma_2$ where $\langle\ \mathcal{S}_1 \triangleright \mathtt{t}_1 \rightsquigarrow \mathcal{S}_2 \triangleright \mathtt{t}_2\ \rangle^1$ then, for arbitrary $\gamma \in$ fresh, a well-formed and borrow safe typing environment $\Gamma_3[\gamma \mapsto \langle \mathtt{T}_2 \rangle^1] \sqsubseteq \Gamma_2[\gamma \mapsto \langle \mathtt{T}_1 \rangle^1]$ exists where $\mathcal{S}_2 \sim \Gamma_3$.*

For the calculus core, Corollary 4.14 can be immediately strengthened with $\Gamma_2 = \Gamma_3$ (see Appendix for the strengthened form). In other words, *the calculus core maintains borrow safety when starting from a borrow safe environment.* The key observation is that, when the typing environment indicates no borrow exists for a given lval, then no such borrow can exist in any program store it abstracts. In such case, it is always safe to take a mutable borrow of that lval. Indeed, this is a requirement for T-MutBorrow — which is the only way to create a mutable borrow in FR. A similar argument applies when taking immutable borrows.

Finally, for extensions to the calculus core, Corollary 4.14 cannot always be strengthened in the same manner (e.g. for the control-flow extension discussed in §6.1). This can arise when Γ_2 conservatively abstracts *multiple execution paths*. However, since execution in FR is deterministic, *only one* execution path can be executed in practice, and Corollary 4.14 simply allows us to identify a more precise environment for that execution path. We return to discuss this further in §6.1.3.

4.5.2 (Non-)Termination. Another observation regarding Theorem 4.12 is that it requires termination. For the calculus presented thus far (and the extensions considered later) this is indeed the case. Nevertheless, extending FR with loops and recursion is desirable, but this requires a weaker theorem than that presented thus far. Such a theorem would need to characterise intermediate states to account for non-termination. For example, suppose some reduction $\langle\ \mathcal{S}_1 \triangleright \mathtt{t}_1 \longrightarrow \mathcal{S}_2 \triangleright \mathtt{t}_2\ \rangle^1$ (where $\mathtt{t}_2$ is not yet a value) and corresponding typing $\Gamma_1 \vdash \langle\ \mathtt{t}_1 : \mathtt{T}\ \rangle^1_\sigma \dashv \Gamma_3$ (where $\mathcal{S}_1 \sim \Gamma_1$, etc). Then, some intermediate typing environment, Γ_2, must exist which establishes that $\mathtt{t}_2$ remains well typed (i.e. where $\mathcal{S}_2 \sim \Gamma_2$ and $\Gamma_2 \vdash \langle\ \mathtt{t}_2 : \mathtt{T}\ \rangle^1_\sigma \dashv \Gamma_3$). Showing that such an environment always exists could easily be proven as a theorem in its own right, but would require some (albeit straightforward) reworking of our proof structure.

5 IMPLEMENTATION

A reference implementation (in Java) of FR is publicly available on GitHub.[13] A key goal was that one could easily check by hand that the implementation is consistent with the rules presented in §3.[14] Thus, the implementation is not the most efficient possible, but one which has a close correspondence with the rules as presented in this paper. Having a reference implementation offers many benefits. For example, we can use it to easily explore extensions to the calculus. Likewise, in the spirit of lightweight mechanisation [74, 111], we can model check against large numbers of automatically generated programs or fuzz test `rustc`.

5.1 Overview

The reference implementation consists of three main components: an *abstract syntax tree*; a *type and borrow checker*; and, an *interpreter*. For example, recall the rule R-Move from Page 15:

$$\frac{\mathtt{read}(\mathcal{S}_1, \mathtt{w}) = \langle \mathtt{v} \rangle^{\mathtt{m}} \quad \mathcal{S}_2 = \mathtt{write}(\mathcal{S}_1, \mathtt{w}, \bot)}{\langle\ \mathcal{S}_1 \triangleright \mathtt{w} \longrightarrow \mathcal{S}_2 \triangleright \mathtt{v}\ \rangle^1} \qquad \text{(R-Move)}$$

In our reference implementation, this rule is written as follows:

```
protected Pair<State, Term> reduceMove(State S1, Lifetime l, LVal w) {
  // Read contents of slot at given location
  Value v = S1.read(w);
  // Apply destructive update
  State S2 = S1.write(w, null);
  // Return value read
  return new Pair<>(S2, v);
}
```

A program store, $\mathcal{S}$, from our operational semantics is implemented using an immutable structure, `State`, which maps variables to their bound locations and locations to their assigned values. Using an immutable structure is not the most efficient but, as we can see above, it more closely follows the original rules.

In a similar fashion, the rules for borrow checking are implemented using immutable structures. Recall the rule T-Declare from page 24:

$$\frac{\mathtt{x} \notin \mathbf{dom}(\Gamma_1) \qquad \Gamma_1 \vdash \langle\ \mathtt{t} : \mathtt{T}\ \rangle^1_\sigma \dashv \Gamma_2 \qquad \Gamma_3 = \Gamma_2[\mathtt{x} \mapsto \langle \mathtt{T} \rangle^1]}{\Gamma_1 \vdash \langle\ \mathtt{let\ mut\ x = t} : \epsilon\ \rangle^1_\sigma \dashv \Gamma_3} \qquad \text{(T-Declare)}$$

[13] https://github.com/DavePearce/FeatherweightRust

[14] This bears some semblance with Pollack's general approach for gaining confidence in the correctness of a proof checker [105].

In our reference implementation, this rule is written as follows:

```
Pair<Environment, Type> apply(Environment R1, Lifetime l, Term.Let t) {
  String x = t.variable();
  Slot Sx = R1.get(x);
  check(Sx == null, VARIABLE_ALREADY_DECLARED, t);           // Check not declared
  Pair<Environment, Type> p = apply(R1, l, t.initialiser()); // Type operand
  Environment R2 = p.first();
  Type T = p.second();
  Environment R3 = R2.put(x, T, l);                          // Update environment
  return new Pair<>(R3, Type.Unit);
}
```

Here, we can see typing environments, Γ, are implemented using the immutable structure `Environment` which maps variable names to instances of `Slot` which contain the variable's type and lifetime information. Furthermore, environment updates are handled using `put` where, for example, $\Gamma_3 = \Gamma_2[\mathtt{x} \mapsto \langle \mathtt{T} \rangle^{\mathtt{l}}]$ corresponds to `R3 = R2.put(x, T, l)`. The requirement $\mathtt{x} \notin \mathbf{dom}(\Gamma_1)$ is implemented using a special `check` function which throws an exception when the given condition is false. Thus, our implementation checks all requirements stated in our typing judgments. Finally, note that no equivalent of the store typing, σ, is present in our implementation. This is because we type and borrow check programs written in the source-level syntax before executing them and, hence, the store typing is unnecessary.[15]

5.2 Bounded Model Checking

Using our reference implementation, we have model checked the typing rules of FR against its operational semantics. As is the case with bounded model checking, this demonstrates correctness for the specific range of programs examined. Nevertheless, we were able to check a very large number of programs (500B+) chosen from the space of all programs, giving a high degree of confidence. The relative conciseness of FR is one reason this was possible. Another was the judicious use of constraints to dramatically reduce the space of programs (e.g. using one representative for programs which are otherwise identical up to variable renaming).

5.2.1 Program Domains. To model check our reference implementation, we exhaustively generate input programs within certain limits. To enable this, we define the (parameterised) "space" of all possible programs as follows:

Definition 5.1. A finite space of FR programs is denoted by $\mathcal{P}_{i,v,d,w}$ where: i defines the number of distinct integer literals; v the number of distinct variable names; d the maximum nesting of statement blocks; and w the maximum width of any statement block.

The four parameters determine the size of space described and, of course, are used to limit the number of programs to something which can be practically enumerated in reasonable time. Observe that, since FR does not support any arithmetic operators, we are interested only in the number of distinct integer literals, rather than their exact values (i.e. using a set $\{0, 1, 2\}$ versus $\{-1, 0, 1\}$ makes no difference). Also, since every FR program consists of an outermost statement block, we have $d \geq 1$. Furthermore, the width of a statement block is the number of statements it contains and, for simplicity, we omit empty blocks (i.e. because they have no effect). We also ignore non-sensical statement forms (e.g. "`let mut x = let mut y = 1;`") and, furthermore, retain a strict

[15] Recall the store typing is necessary for typing locations which arise from reducing certain expressions, such as box e. Such locations are intermediate values which are not expressible in the source-level syntax.

separation between statements and expressions. The latter means that blocks are never positioned as expressions (e.g. as in "`let mut x = {1}`") since this is of little value in the core calculus. We also limit the depth of expressions and, again, for simplicity assume at most one occurrence of `box` in any given expression.[16] As an example, the space $\mathcal{P}_{1,1,1,1}$ is given in Figure 4 and, for reference, we note the following:

$$
\begin{array}{lcllcl}
|\mathcal{P}_{1,1,1,1}| & = & 54 & |\mathcal{P}_{1,2,2,3}| & = & 6.2102 \times 10^{20} \\
|\mathcal{P}_{1,1,1,2}| & = & 2970 & |\mathcal{P}_{1,2,3,3}| & = & 2.3951 \times 10^{62} \\
|\mathcal{P}_{1,1,2,2}| & = & 9147600 & |\mathcal{P}_{1,3,2,3}| & = & 6.3054 \times 10^{17} \\
|\mathcal{P}_{1,2,2,2}| & = & 1766058600 & |\mathcal{P}_{1,3,3,2}| & = & 1.7116 \times 10^{21} \\
|\mathcal{P}_{2,2,2,2}| & = & 2217326832 & |\mathcal{P}_{1,3,3,3}| & = & 2.507 \times 10^{53}
\end{array}
$$

Generating all elements of a given program space is relatively straightforward but, of course, requires time linear in its size. Our implementation maps each element to an integer index allowing the corresponding program to be selected in constant time without holding all elements in memory (i.e. roughly similar to the approach taken in Feat [34, 43]). This means, for example, we can sample uniformly from a space using Knuth's Algorithm S [75].

Unfortunately, many interesting program spaces remain intractably large. A key litmus test for the suitability of a program space is whether or not certain programs, such as the following, are included:

$$\{\texttt{let mut x = 0; let mut y = \&mut x; \{let mut z = 1; y = \&mut z;\}}^{m}\ \}^{1} \tag{18}$$

This program should not type check because its execution leads to a dangling reference. This program is important as *there is no smaller program which causes a dangling reference solely through incorrect lifetime inclusion.* That is, one whose detection relies exclusively on a lifetime inclusion

[16]Note, this is not so significant since expressions with multiple occurrences always have counterparts in three-address form. For example, whilst "`let mut x = box box 0;`" is omitted, "`let mut x = box 0; let mut y = box x;`" is not.

{ let mut x = 0; }	{ x = 0; }	{ *x = 0; }
{ let mut x = x; }	{ x = x; }	{ *x = x; }
{ let mut x = *x; }	{ x = *x; }	{ *x = *x; }
{ let mut x = x̂; }	{ x = x̂; }	{ *x = x̂; }
{ let mut x = *x̂; }	{ x = *x̂; }	{ *x = *x̂; }
{ let mut x = &mut x; }	{ x = &mut x; }	{ *x = &mut x; }
{ let mut x = &mut *x; }	{ x = &mut *x; }	{ *x = &mut *x; }
{ let mut x = &x; }	{ x = &x; }	{ *x = &x; }
{ let mut x = &*x; }	{ x = &*x; }	{ *x = &*x; }
{ let mut x = box 0; }	{ x = box 0; }	{ *x = box 0; }
{ let mut x = box x; }	{ x = box x; }	{ *x = box x; }
{ let mut x = box *x; }	{ x = box *x; }	{ *x = box *x; }
{ let mut x = box x̂; }	{ x = box x̂; }	{ *x = box x̂; }
{ let mut x = box *x̂; }	{ x = box *x̂; }	{ *x = box *x̂; }
{ let mut x = box &mut x; }	{ x = box &mut x; }	{ *x = box &mut x; }
{ let mut x = box &mut *x; }	{ x = box &mut *x; }	{ *x = box &mut *x; }
{ let mut x = box &x; }	{ x = box &x; }	{ *x = box &x; }
{ let mut x = box &*x; }	{ x = box &*x; }	{ *x = box &*x; }

Fig. 4. The complete space $\mathcal{P}_{1,1,1,1}$.

check (e.g. premise "$\mathtt{T}_2 \vdash \mathtt{T}_2 \geq \mathtt{m}$" in rule T-Assign). To see this, we must consider the essential ingredients for creating such a dangling reference:

(1) A variable (y) which will refer to another variable (z) whose location has been dropped.
(2) A nested block declaring the location (z) which is to be dropped.
(3) An initial target (x) for the referent variable (y) as required for its declaration.

Perhaps one surprise here is the need for variable x. At first glance, it might appear we could avoid this using a box initialiser:

$$\{\mathtt{let\ mut\ y = box\ 0;\ \{let\ mut\ z = 1;\ y = \&mut\ z;\}^m\}^1} \tag{19}$$

Unfortunately, whilst this program does create a dangling reference, it can be detected without lifetime inclusion. Specifically, this program does not type check under rule T-Assign (recall page 25) because the type of y ($\Box\mathtt{int}$) is not shape compatible (recall Definition 3.22) with that being assigned ($\mathtt{\&mut\ z}$). Whilst no program smaller than (18) exists which creates a dangling reference, other programs of the same size do exist. For example, the following variation:

$$\{\mathtt{let\ mut\ x = 0;\ let\ mut\ y = \&x;\ \{let\ mut\ z = 1;\ y = \&z;\}^m\}^1} \tag{20}$$

We can reason that, for a space to be considered interesting, it must *at least* contain one or more programs which can create a dangling reference. Here, $\mathcal{P}_{1,3,2,3}$ is one such space, as is $\mathcal{P}_{1,3,3,2}$.

5.2.2 Constrained Program Domains. The spaces $\mathcal{P}_{1,3,2,3}$ and $\mathcal{P}_{1,3,3,2}$ are of interest (as above) but remain intractable in terms of their size. One approach is to sample from these domains, and this can be done relatively easily. Unfortunately, this is rarely fruitful as, with high probability, all programs sampled are invalid (i.e. do not type and borrow check) [30]. Thus, a borrow checker which returned `false` in all cases would appear effective.

Instead of sampling, we adopt an alternative approach to constraining such program spaces which eliminates many invalid programs, whilst retaining a healthy mix of both valid and invalid programs. We adopt the common strategies of eliminating unbound variables [74] and isomorphs [111]:

Definition 5.2. Let $\mathcal{P}_{i,v,d,w}$ be a finite space of FR programs. Then, $\mathcal{P}^{\mathsf{def},b}_{i,v,d,w} \subseteq \mathcal{P}_{i,v,d,w}$ is a constrained program domain where each program is restricted as follows: (1) every variable is defined before being used; (2) programs are unique up to variable renaming; (3) there are at most b blocks.

Since, we are largely interested in borrow checking it makes sense to ignore programs which fail definite assignment (point (1) above). Likewise, we are not specifically concerned with variable naming and, hence, there is little point in checking all isomorphs (with respect to variable renaming) of a given program (point (2) above); finally, we further constrain the number of blocks. For example, the following program is a member $\mathcal{P}_{1,1,2,1}$:

$$\{\ \{\mathtt{let\ mut\ x = 0;}\}^m\ \{\mathtt{let\ mut\ x = 0;}\}^n\ \}^1 \tag{21}$$

Since this program contains three blocks, it is not present in $\mathcal{P}^{\mathsf{def},2}_{1,1,2,1}$. **Crucially, however, our program of interest** (18) **and its variation** (20) **are contained in** $\mathcal{P}^{\mathbf{def},2}_{1,3,2,3}$. Furthermore, these constrained program spaces are significantly more tractable. For example, $|\mathcal{P}^{\mathsf{def},2}_{1,3,2,3}| = 418496660$.

A key challenge in generating such constrained domains is that they are no longer uniform in nature. For example, consider the (non-constrained domain) $\mathcal{P}_{i,v,d,w}$: the dimensions of this can be determined without enumerating its elements; and, more importantly, the domain supports random access of its elements. In short, generating the elements of such a domain is straightforward. In contrast, the dimensions of $\mathcal{P}^{\mathsf{def},b}_{i,v,d,w}$ cannot be determined beforehand, and random access of elements is not possible. Nevertheless, it is possible (with care) to iterate all elements of the domain.

Domain	Size	Valid	Invalid	False Pos	False Neg	Time
$\mathcal{P}_{1,1,1,1}$	54	2	52	0 (0%)	0	0s
$\mathcal{P}_{1,1,1,2}$	2970	12	2958	52 (1.8%)	0	0s
$\mathcal{P}_{1,1,2,2}$	9147600	260	9147340	15444 (0.2%)	0	30s
$\mathcal{P}_{1,2,2,2}$	1766058600	7174	1766051426	153494 (0.01%)	0	5237s
$\mathcal{P}_{2,2,2,2}$	2217326832	37028	2217289804	479996 (0.02%)	0	5978s
$\mathcal{P}^{\mathsf{def},2}_{1,2,2,2}$	9332	623	8709	2749 (31.6%)	0	0s
$\mathcal{P}^{\mathsf{def},2}_{2,2,2,2}$	22824	1954	20870	6918 (33.2%)	0	0s
$\mathcal{P}^{\mathsf{def},2}_{1,2,2,3}$	182401748	220991	182180757	26761299 (14.7%)	0	929s
$\mathcal{P}^{\mathsf{def},2}_{1,3,2,3}$	418496660	876174	417620486	64946576 (15.6%)	0	1918s
$\mathcal{P}^{\mathsf{def},3}_{1,2,2,2}$	21432	2067	19365	5401 (27.9%)	0	0s
$\mathcal{P}^{\mathsf{def},3}_{2,2,2,2}$	82360	10054	72306	20722 (28.7%)	0	0s
$\mathcal{P}^{\mathsf{def},3}_{1,2,2,3}$	500246168816	24376766	500221792050	27300386528 (5.46%)	0	-

Table 1. Illustrating the main results for model checking FR across various program spaces, where: **size** denotes each program space's cardinality; **valid** denotes the number of programs that passed type and borrow checking; **invalid** denotes the number that failed type and borrow checking; **false pos** indicates the number that failed type and borrow checking but did not generate a runtime fault during execution; **false neg** denotes the number which passed type and borrow checking and generated a runtime fault during execution; and, finally, **time** denotes the time take to check each space.

This is done by maintaining meta-information regarding the number of variables declared and blocks created during the traversal. To understand this, consider the following:

$$\{\ \{\ \texttt{let mut x = 0;} \bullet\ \}^1 \tag{22}$$

Here, • indicates the "cursor" position and suppose we wish to generate all statements which could replace it. For $\mathcal{P}_{i,v,d,w}$ this is straightforward — we simply enumerate all possible statements (including, for example, "`let mut x = 0`"). For the constrained domain, however, the set of possible statements at this point is *context sensitive*. For example, "`let mut x = 0`" is not permitted (as this would be a redeclaration of `x`) and, likewise, "`let mut y = z`" is not permitted (as `z` is not declared). Thus, as the above example suggests, we generate programs in a left-to-right fashion whilst maintaining necessary meta-information as each cursor position. Further details of the algorithm can be found in our implementation.

5.2.3 Discussion. The results from model checking our calculus across a range of program spaces are presented in Table 1. The experiments were conducted on an 32-core Dell Precision 7920R running Arch Linux and are included only to give an indication of how long they took (note $\mathcal{P}^{\mathsf{def},3}_{1,2,2,3}$ required completion via a grid of machines — see below). Most importantly, the spaces $\mathcal{P}^{\mathsf{def},2}_{1,3,2,3}$ and $\mathcal{P}^{\mathsf{def},3}_{1,2,2,3}$ *were exhaustively checked with zero false negatives being raised.* A false negative would indicate a program that passed type and borrow checking, but raised a fault at runtime. In our implementation, faults are raised for the obvious reasons (e.g. accessing undeclared or uninitialised variables, attempting to treat integers as references, etc). In addition, whenever a location is dropped we check for dangling references and raise a fault if any are found — thus, for example, executing program (18) above leads to a runtime fault. We include information about the number of false positives purely for interest. This indicates programs which failed type and borrow checking for some reason but, when executed, did not generate a fault.

Finally, we note that during the development and testing for these experiments we did identify bugs in earlier formulations of FR (which, of course, we corrected).

5.3 Fuzz Testing

Given our ability to generate both valid and invalid programs according to the type and borrow checking rules of FR, an obvious question is how this compares with the actual Rust implementation. To that end, we have generated a large number of input programs, compiled them with `rustc` version `1.39.0` and compared the output (i.e. whether compilation succeeded or failed) against our calculus. In doing this, we targeted the 2015 edition of Rust which provides a slightly better comparison with FR.

5.3.1 Translation. Whilst our calculus is effectively a subset of Rust, some massaging is required to get programs to compile with `rustc`:

- (**Function Syntax**) Each FR program was given the header `fn main()` in the generated Rust program.
- (**Box Syntax**) The syntax `box e` is not supported by default in Rust. Therefore, we generated `Box::new(e)` whenever box e is used in FR.
- (**Lifetime Syntax**) The explicit syntax used for associating statement blocks with lifetimes (i.e. $\{\ldots\}^l$) is not valid Rust syntax. Therefore, lifetime annotations were simply dropped when generating Rust programs.
- (**Variable Shadowing**). Since Rust permits variable shadowing (where FR does not), some valid Rust programs are invalid FR programs. For example, "`{let mut x = 0; let mut x = 1;}`" is not a valid FR program, whilst its translation is a valid Rust program. To handle this, programs which shadow variables were simply ignored.
- (**Non-Lexical Lifetimes**) In 2018, Rust adopted a new borrow checker which supported Non-Lexical Lifetimes (NLL). For example, "`{let mut x = 0; let mut y = &mut x; x = 0;`" is not a valid FR program because x is mutably borrowed at the final assignment. However, its translation is a valid Rust program:

  ```
  fn main() { let mut x = 0; let mut y = &mut x; x = 0; }
  ```

  This is accepted by `rustc` because it terminates the mutable borrow before the end of the enclosing block. Indeed, this is apparent from the following which is not accepted:

  ```
  fn main() { let mut x = 0; let mut y = &mut x; x = 0; y; }
  ```

  As expected, this program fails because the mutable borrow of `x` is active at the final assignment. To work around this, we simply introduce uses of variables which are live at the end of their declaring block (as above).
- (**Move/Copy Syntax**) The use of explicit moves and copies in FR is an important difference from Rust. In `rustc`, the choice to copy variables is made by looking for the `Copy` trait. Furthermore, whilst it is possible to force a copy in Rust by generating `*&x` for $\hat{x}$, it is not possible to force a *move*. Consider the FR program "`{let mut x = 1; let mut y = x; let mut z = x;}`" which is invalid as x was moved before the final statement. This is translated as follows:

  ```
  fn main() { let mut x = 1; let mut y = x; let mut z = x; y; z; }
  ```

  This is a valid Rust program because `rustc` infers a copy from `x` to `y`, rather than a move. In other words, the program understood by `rustc` is *not* equivalent to the original FR program. To resolve this issue, programs containing variable copies were ignored, such as

Domain	Size	Ignored	Valid	Invalid	Inconsistent
$\mathcal{P}_{1,1,1,1}$	54	12 (22%)	2	40	0
$\mathcal{P}_{1,1,1,2}$	2970	1360 (45.8%)	9	1601	0
$\mathcal{P}_{1,1,2,2}$	9147600	6428212 (70.3%)	153	2719235	0
$\mathcal{P}_{1,2,2,2}$	1766058600	1471593416 (83.3%)	1362	294463772	50
$\mathcal{P}_{2,2,2,2}$	2217326832	1822114504 (82.2%)	7142	395205086	100
$\mathcal{P}^{\mathsf{def},2}_{1,2,2,2}$	9332	3640 (39%)	401	5241	50
$\mathcal{P}^{\mathsf{def},2}_{2,2,2,2}$	22824	8144 (35.7%)	1366	13214	100
$\mathcal{P}^{\mathsf{def},2}_{1,2,2,3}$	182401748	117782248 (64.6%)	77326	64513929	28245
$\mathcal{P}^{\mathsf{def},2}_{1,3,2,3}$	418496660	271930568 (65%)	355864	146035341	174887
$\mathcal{P}^{\mathsf{def},3}_{1,2,2,2}$	21432	8344 (38.9%)	1362	11676	50
$\mathcal{P}^{\mathsf{def},3}_{2,2,2,2}$	82360	29264 (35.5%)	7142	45854	100

Table 2. Results from fuzz testing `rustc` with FR input programs, where: **size** denotes each program space's cardinality; **ignored** indicates number of inputs ignored; **valid** denotes the remaining programs that passed type and borrow checking; **invalid** denotes the remainder that failed type and borrow checking; and, **inconsistent** indicates number of inputs with inconsistent results between `rustc` and FR.

"{let mut x = box 0; let mut y = $\hat{x}$;}". All remaining FR programs were then subjected to a *copy inference* which mimics `rustc` by inserting variable copies into FR programs. Specifically, types which support copy semantics are always copied and others are always moved.

To illustrate our translation, consider the following FR program:

$$\{\ \texttt{let mut x = 1;}\ \texttt{let mut y = box}\ \hat{x};\ \{\texttt{let mut z = box 0;}\ \texttt{y = z;}\}^{m}\ \}^{l} \tag{23}$$

The above FR program is translated into the following Rust program:

```
fn main() {let mut x=1; let mut y=Box::new(x); {let mut z=Box::new(0); y=z;} y; x;}
```

Using this translation we were able to pass a large number of FR programs through the Rust compiler.

5.3.2 Discussion. The results from fuzz testing against `rustc` are presented in Table 2. Timing data is not reported as these results were generating using the departmental grid (running Sun Grid Engine version `8.1.6`) comprising approximately 400 machines. Roughly speaking, the experiments took around one week to complete. The main observation from Table 2 is that there are relatively few inconsistencies between FR and `rustc`. We note the number of ignored inputs is seen to increase proportionally with the size of the underlying space. This is to be expected as the probability of an FR program containing a variable copy is higher than that of a program which contains no copies.

To help understand the observed inconsistencies between FR and `rustc` in Table 2, further analysis was performed. The results of this are shown in Table 3 and uncovered several issues:

(1) **(E0506)**. These correspond to situations where a program is accepted by FR but rejected by `rustc` with an `E0506` error code. The following illustrates such a program:

```
fn main() {let mut x=Box::new(0); let mut y=Box::new(&mut x); y=Box::new(*y);}
```

This program appears safe since the borrow `&mut x` is simply moved into the new box. The reason `rustc` rejects this is most likely because it incorrectly applies an implicit reborrow.

Domain	Total Inconsistent	`rustc`	FR		Other
		E0506	Deref Coercions	Cyclic Assignment	
$\mathcal{P}_{1,2,2,2}$	50	3	41	6	0
$\mathcal{P}_{2,2,2,2}$	100	6	82	12	0
$\mathcal{P}^{\mathsf{def},2}_{1,2,2,2}$	50	3	41	6	0
$\mathcal{P}^{\mathsf{def},2}_{2,2,2,2}$	100	6	82	12	0
$\mathcal{P}^{\mathsf{def},2}_{1,2,2,3}$	28245	828	21393	3725	2299
$\mathcal{P}^{\mathsf{def},2}_{1,3,2,3}$	174887	5978	129989	20253	18667
$\mathcal{P}^{\mathsf{def},3}_{1,2,2,2}$	50	3	41	6	0
$\mathcal{P}^{\mathsf{def},3}_{2,2,2,2}$	100	6	82	12	0

Table 3. Results from analysing inconsistencies identified between `rustc` and FR in Table 2.

(2) **(Deref Coercions)**. These correspond to problems arising from the lack of support for deref coercions in FR. The following illustrates:

```
fn main() { let x = 0; { let mut y = Box::new(&x); y = Box::new(&y); } }
```

This program is accepted by `rustc` but rejected by FR under rule T-Assign. More specifically, the type of `y` is "`□&int`" but, in the last statement, it is assigned a value of (incompatible) type "`□&□&int`". Indeed, this appears to unsoundly create a cycle from `y` back to itself! However, in fact, Rust employs implicit *deref coercions* here, meaning the last statement is effectively treated as though it was `y = Box::new(**&y)`. [17]

(3) **(Cyclic Assignments)**. These correspond with programs containing statements that have no effect, but are rejected by FR. For example, consider the following:

```
fn main() { let mut x = 0 ; let mut y = &x ; y = &*y; }
```

This program is accepted by `rustc` but rejected by FR under rule T-Assign because `y` is write protected by the "self borrow" `&*y`. Such borrows are problematic for FR though could be handled with some special-case simplification rules. This issue arises in various guises (e.g. replacing `&x` and `&*y` with `&mut x` and `&mut *y` above). Since all cases have a common theme (e.g. assigning a variable an expression involving itself), these can easily be detected and classified. We note that our classification does not include trivial assignments of the form `y = y;` or `*y = *y;`, etc. Also, those cases which can be resolved through deref coercions are reported as above.

(4) **(Other)**. Finally, whilst most inconsistencies arise from implicit coercions, a small number remain. For example, the following is rejected by `rustc`:

```
fn main() {let mut x = 0; {let mut y = Box::new(&x); let mut z = 0; *y = &z;}}
```

This program is rejected because "`` `z` does not live long enough ``". This arises because of the ordering in which Rust drops variables (i.e. `z` first in this case). In contrast, FR drops all variables declared in a block simultaneously. Indeed, the above is accepted by `rustc` when the declaration order for `y` and `z` is reversed.

[17] https://stackoverflow.com/questions/57655286/

Overall, we found fuzz testing against `rustc` to be an extremely useful exercise which has resulted in much greater confidence that FR closely models Rust. This work also uncovered some previously unknown issues in `rustc` as well. Specifically, the following was uncovered by fuzzing:

```
fn main() { let mut x = Box::new(0); let mut y = &mut x; y = y; }
```

An error `E0506` was reported (in e.g. `rustc` version `1.35.0`) for the last statement stating that one cannot assign to `y` because it is already borrowed. As a result of this work, an issue was raised with the `rustc` developers which was accepted and eventually fixed.[18]

Through fuzzing, we have also noticed changes between different versions of the Rust compiler. The following illustrates one such example:

```
fn main() { let mut x = Box::new(0); let mut y = &x; y = &mut y; }
```

This program was accepted by `rustc` version `1.35.0` (edition 2015). However, for a while, later versions of `rustc` (e.g.`1.36.0`) reported a `W0506` warning:

```
warning: this error has been downgraded to a warning for backwards compatibility
    with previous releases
warning: this represents potential undefined behavior in your code and this
    warning will become a hard error in the future
```

And yet, more recent versions (e.g. `1.39.0`) now accept this program again without such a warning! We believe automated fuzzing using FR could ensure better continuity between releases of `rustc`.

6 EXTENSIONS

The syntax given for FR is considerably smaller than for full Rust and, in fact, is not a subset though it does come close (i.e. as illustrated in §5.3). Many obvious things are missing such as conditional statements, loops and method invocation. Likewise, the range of types is fairly limited and does not include **`struct`**s, tuples, booleans or immutable variables. To this end we now present two completed extensions to FR for control-flow (§6.1) and tuples (§6.2) and sketch another for functions (§6.3) which, nevertheless, is relatively complete.

6.1 EXTENSION: Control Flow

The lack of control-flow in FR is a useful simplification and, in many ways, relatively little is lost through this. In fact, adding control-flow is about the simplest extension possible to FR. One important aspect here is that of merging typing environments at join points in the control-flow graph. Consider the following example which compiles in Rust:

```
fn f(n: i32) -> i32 {
    let mut x = 0;
    {
        let mut y = 1;
        let mut r = &x;
        if n == 0 { r = &y; }
        return *r;
}   }
```

[18] https://github.com/rust-lang/rust/issues/63719

t ::=	*Terms*	v ::=	*Values*
…		…	
`if` t $\{\overline{\texttt{t}}\}^{\texttt{m}}$ `else` $\{\overline{\texttt{t}}\}^{\texttt{n}}$	*conditional*	`true`, `false`	*boolean*
$\texttt{t}_1$ == $\texttt{t}_2$	*equality*		
…			
		T ::=	*Types*
		…	
		`bool`	*boolean*

Fig. 5. Syntactic extensions to FR as required for conditional statements.

The question here is what type does `r` have at the **`return`** statement? In FR, it would be given type &y on the true branch and carry through type &x on the false branch. But, how would these be combined together? In fact, the machinery for determining this is already available in FR via Definition 3.10 (page 21). Specifically, the environments arising from the true and false branches are joined together, giving a type of "&x, y" for `r` at the **`return`** statement.

To illustrate such an extension, we consider now the addition of simple **`if`** statements to FR. This includes extending the *syntax*, *operational semantics* and *typing rules* of FR.

6.1.1 Syntax & Semantics. Figure 5 illustrates the syntactic extensions required to support conditionals in FR. This includes the addition of boolean types and values, as well as an equality comparator (and more could easily be envisaged here). In addition, corresponding semantic reductions are required for basic equality and the reduction of conditionals:

$$\frac{}{\mathcal{S} \triangleright \texttt{v} == \texttt{v} \longrightarrow \mathcal{S} \triangleright \texttt{true}} \quad \text{(R-EqualT)} \qquad \frac{\texttt{v}_1 \neq \texttt{v}_2}{\mathcal{S} \triangleright \texttt{v}_1 == \texttt{v}_2 \longrightarrow \mathcal{S} \triangleright \texttt{false}} \quad \text{(R-EqualF)}$$

$$\frac{}{\mathcal{S} \triangleright \texttt{if true } \{\overline{\texttt{t}}\}^{\texttt{m}} \texttt{ else } \{\overline{\texttt{t}}\}^{\texttt{n}} \longrightarrow \mathcal{S} \triangleright \{\overline{\texttt{t}}\}^{\texttt{m}}} \quad \text{(R-IfT)}$$

$$\frac{}{\mathcal{S} \triangleright \texttt{if false } \{\overline{\texttt{t}}\}^{\texttt{m}} \texttt{ else } \{\overline{\texttt{t}}\}^{\texttt{n}} \longrightarrow \mathcal{S} \triangleright \{\overline{\texttt{t}}\}^{\texttt{n}}} \quad \text{(R-IfF)}$$

The above rules are fairly straightforward, though we must additionally extend the notion of an evaluation context as follows to ensure R-Sub (recall page 17) still applies:

Definition 6.1 (Extended Evaluation Context). An *evaluation context* is a term containing a single occurrence of $[\![\cdot]\!]$ (the hole) in place of a subterm. Evaluation contexts are defined as follows:

$$E \quad ::= \quad [\![\cdot]\!] \mid \ldots \mid \texttt{if } E \ \{\overline{\texttt{t}}\}^{\texttt{m}} \texttt{ else } \{\overline{\texttt{t}}\}^{\texttt{n}} \mid E == \texttt{t} \mid \texttt{v} == E$$

Observe that the true and false branches are not considered evaluation contexts. This ensures the reduction of a conditional must go through either of the rules R-IfT or R-IfF and, hence, that erroneous terms (e.g. "`if 1` $\{\overline{\texttt{t}}\}^{\texttt{m}}$ `else` $\{\overline{\texttt{t}}\}^{\texttt{n}}$") are stuck.

6.1.2 Typing. The typing rule for conditional terms employs both the type join (recall Definition 3.8, page 20) and environment join (Definition 3.10, page 21) operators as follows:

$$\frac{\Gamma_1 \vdash \langle\ \texttt{t} : \texttt{bool}\ \rangle^{\texttt{l}}_{\sigma} \dashv \Gamma_2 \quad \Gamma_2 \vdash \langle\ \{\overline{\texttt{t}}\}^{\texttt{n}} : \texttt{T}_1\ \rangle^{\texttt{l}}_{\sigma} \dashv \Gamma_3 \quad \Gamma_2 \vdash \langle\ \{\overline{\texttt{s}}\}^{\texttt{m}} : \texttt{T}_2\ \rangle^{\texttt{l}}_{\sigma} \dashv \Gamma_4}{\Gamma_1 \vdash \langle\ \texttt{if t } \{\overline{\texttt{t}}\}^{\texttt{n}} \texttt{ else } \{\overline{\texttt{s}}\}^{\texttt{m}} : \texttt{T}_1 \sqcup \texttt{T}_2\ \rangle^{\texttt{l}}_{\sigma} \dashv \Gamma_3 \sqcup \Gamma_4} \quad \text{(T-If)}$$

Here, the resulting type from each block is joined to form the resulting type of the conditional itself, meaning it can be used as a general term (e.g. on the right-hand side of an assignment, etc). For completeness, a rule for typing equality comparators is also required:

$$\frac{\Gamma_1 \vdash \langle\ \mathtt{t}_1 : \mathtt{T}_1\ \rangle^1_\sigma \dashv \Gamma_2 \quad \Gamma_2[\gamma \mapsto \langle \mathtt{T}_1 \rangle^1] \vdash \langle\ \mathtt{t}_2 : \mathtt{T}_2\ \rangle^1_\sigma \dashv \Gamma_3 \quad \Gamma_4 = \Gamma_3 - \{\gamma \mapsto \langle \mathtt{T}_1 \rangle^1\} \quad \Gamma_4 \vdash \mathtt{T}_1 \approx \mathtt{T}_2 \quad \mathtt{copy}(\mathtt{T}_1) \quad \mathtt{copy}(\mathtt{T}_2) \quad \gamma \in \mathit{fresh}}{\Gamma_1 \vdash \langle\ \mathtt{t}_1 == \mathtt{t}_2 : \mathtt{bool}\ \rangle^1_\sigma \dashv \Gamma_4} \qquad \text{(T-Equal)}$$

When typing the right operand, the left operand type is again anonymously included in the typing environment (via γ) to protect the ownership invariant (e.g. `&mut x == &mut x` is not permitted in Rust). Likewise, the left and right operand must be copy and they must be compatible to ensure that equality is meaningful. The final environment (Γ_4) does not include the type of either the left or the right operand, allowing some limited non-lexical scoping of borrows (e.g. for `&*x == &*y` neither `*x` nor `*y` is considered borrowed afterwards). This reflects the fact that, once the comparison is complete, the values are discarded.

Finally, we note some apparent differences between the rule T-Equal above and Rust. This relates to variables with move semantics, as illustrated by the following program:

$$\{\ \mathtt{let\ mut\ x = box\ 0;\ let\ mut\ y = box\ 0;\ if\ x == y\ \{\ let\ mut\ z = *x;\ \}^n\ else\ \{\}^m\ \}^1$$

This program is not valid under the typing rules above because neither `x` and `y` have copy semantics. However, such a program is accepted by Rust because it applies a *deref coercion* to both operands and, thus, compares the *contents* of their referents rather than the underlying references themselves (and, hence, the true branch would be taken above).[19]

6.1.3 Discussion. At this point, updating the soundness proof is straightforward and requires minor updates to case analyses of the various lemmas. We note also that our reference implementation includes an extension for conditionals.

An interesting question remains as to what further machinery is necessary for loops. The main challenge arises around typing loop bodies. As is common for dataflow analyses, this requires computing a *fixed-point* of typing environments [97]. The following snippet in Rust illustrates:

```
let (x, y) = (1, 2);
let (mut p, mut q) = (&x,&x);
while ... {
  q = p;
  p = &y;
}
```

Here, the type for both `p` and `q` after the loop should be `&x, y`. Furthermore, to arrive at this, *two passes through the loop body are required during typing*. To understand why, observe that the types for `p` and `q` after a single pass through the loop body are (respectively) `&y` and `&x`. Unfortunately, joining this with the environment from before the loop (i.e. for when the body is never executed) gives `&x, y` for `p` but only `&x` for `q`. This arises because the assignment to `q` occurs before the type of `p` is updated and, hence, a second iteration is needed to propagate this information back around the loop and obtain the correct typing.

We return to reflect upon the borrow safety corollary (recall Corollary 4.14, page 30). To better understand the need for this corollary as stated, consider the following Rust program which compiles without problem:

[19] As an aside, we note the only way to compare references directly (i.e. rather than their contents) is via `std::ptr::eq`.

```
let mut a = 0;
let mut x = 0;
let mut y = 0;
let mut p = &mut x;
let mut q = &mut y;
//
if ... { p = &mut a; } else { q = &mut a; }
```

The key problem here is that, after the conditional statement, the typing environment in FR would be: $\Gamma_1 = \{\mathtt{a} \mapsto \langle \mathtt{int} \rangle^1, \ldots, \mathtt{p} \mapsto \langle \mathtt{\&mut\ x}, \mathtt{a} \rangle^1, \mathtt{q} \mapsto \langle \mathtt{\&mut\ y}, \mathtt{a} \rangle^1\}$. Clearly, this is not a borrow safe environment (recall Definition 4.13, page 30)! And yet, intuitively, it is clear that this program is borrow safe. More specifically, whilst multiple mutable borrows occur in the typing environment *these are for different execution paths through the program*. Hence, they do not conflict in practice. Indeed, we can extract *precise* typing environments for each path by strengthening them accordingly. For the true and false branches we have (respectively) the environments $\Gamma_2 = \{ \ldots, \mathtt{p} \mapsto \langle \mathtt{\&mut\ a} \rangle^1, \mathtt{q} \mapsto \langle \mathtt{\&mut\ y} \rangle^1 \}$ and $\Gamma_3 = \{ \ldots, \mathtt{p} \mapsto \langle \mathtt{\&mut\ x} \rangle^1, \mathtt{q} \mapsto \langle \mathtt{\&mut\ a} \rangle^1 \}$ where $\Gamma_2 \sqsubseteq \Gamma_1$ and $\Gamma_3 \sqsubseteq \Gamma_1$. Here, a precise environment is one which cannot be further strengthened (whilst still abstracting the program store in question). Such an environment has exactly one target for every borrow (e.g. "`&mut a`" rather than "`&mut x, a`", etc) and, furthermore, for any given program store exactly one such environment exists. Of course, it remains to show that such an environment is borrow safe but this follows straightforwardly. Roughly speaking, we can flatten a term into a straight-line sequence by specialising for the given execution path taken, thereby reducing it to a term in the calculus core.

6.2 EXTENSION: Tuples

We now consider the problem of adding tuples to FR, which is a somewhat more involved task than for adding control-flow. A particular challenge here is that a tuple changes the shape of expressions from strictly linear (e.g. `Box::new(Box::new(1))`) to supporting trees (e.g. `(&mut x, &mut y)`). As for equality, care is needed when typing such expressions to ensure, for example, `(&mut x,&mut x)` is prohibited (i.e. as it breaks the ownership invariant). Again, this is achieved by anonymously including the i^{th} operand type in the typing environment when typing the i^{th}+1 operand.

Tuples in Rust support indexing (e.g. `(x,y).1`) to access elements, and partial moves (recall §2.3.3). The following program, which compiles, illustrates:

```
fn f() -> i32 {
  let mut p = (Box::new(1), Box::new(2)); let x = p.0; return *(p.1) + *x;
}
```

After the declaration of `x` above, variable `p` has a partial type indicating that the first element is undefined (i.e. since it was moved to `x`). However, the second element remains accessible via `p.1`. Furthermore, observe that `p.1` is an lval and, hence, element indexing is a form of path expression. As such, element indexing offers all the benefits afforded other forms of lval. For example, element indices can be borrowed individually as the following illustrates:

```
fn f() -> i32 { let mut p = (1,2); let p0 = &mut p.0; return p.1 + *p0; }
```

As a result of this flexibility, adding tuples requires a number of definitions from the FR core to be updated. Nevertheless, most are straightforward and we now work through them in more detail.

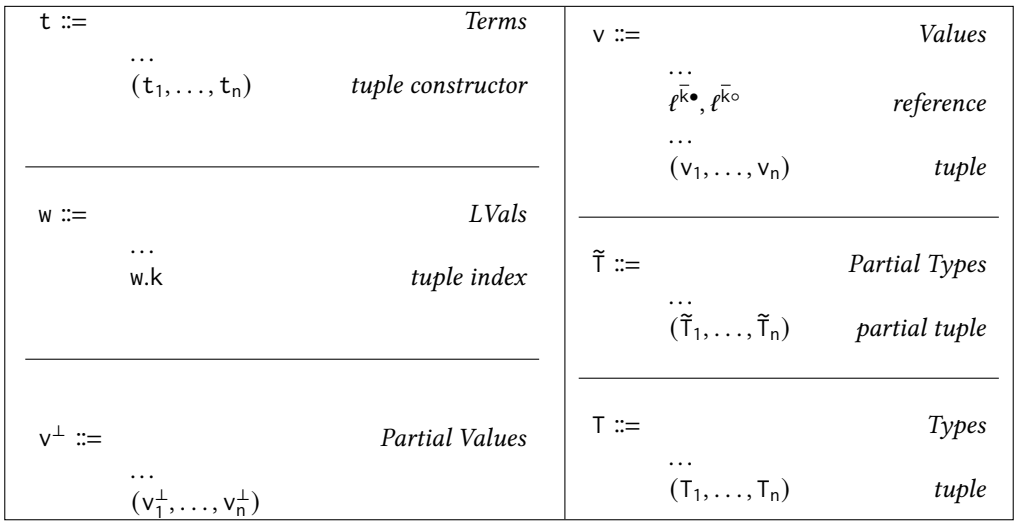

Fig. 6. Syntactic extensions to FR as required for tuples.

6.2.1 Syntax & Semantics. Figure 6 illustrates the syntactic extensions required to support tuples in FR. In fact, the syntactic extensions required are relatively minimal, and mostly serve to distinguish terms from values and partial types from types, etc. We tacitly extend the concept of a path expression so that, for example, "$(1 \cdot *) \mid x$" is the destructured form of "$(*x).1$", etc. Perhaps the most unexpected change is the need to extend the syntax for references with a sequence of zero or more integer "divisors". The divisors are needed to subdivide locations containing a compound value (such as, but not restricted to, tuples). Thus, ℓ refers to a location as before, whilst ℓ^1 refers to the second element of a location, $\ell^{0;1}$ refers to the second element of the first element of a location, etc. Then, borrowed and owned references are extended accordingly. However note that, in fact, owning references never require divisors (i.e. since one cannot obtain an owning reference for *part* of a location).

Perhaps surprisingly, the operational semantics requires no additional evaluation rules since the reduction of tuple terms to tuple values can be handled by R-Sub (recall page 17). To make this work, we must again extend the notion of an evaluation context as follows:

Definition 6.2 (Extended Evaluation Context). An *evaluation context* is a term containing a single occurrence of $[\![\cdot]\!]$ (the hole) in place of a subterm. Evaluation contexts are defined as follows:

$$E \quad ::= \quad [\![\cdot]\!] \mid \ldots \mid (v_1, \ldots, v_{k-1}, E, t_{k+1}, \ldots, t_n)$$

As expected, this enforces a left-to-right evaluation order. Further extensions to the calculus core are also required to connect element indexes to sublocations, and to enable reading / writing from sublocations. In addition, we must ensure that dropping a tuple recursively drops its (defined) elements. To begin, we have two support functions for accessing sublocations within a value. These are defined here for tuples, but could easily be extended for other compound types (e.g. arrays, `struct`s, etc):

Definition 6.3 (Extract). The partial function $\texttt{get}(v^\perp, \overline{k})$ extracts the value at a given sublocation:

$$\begin{array}{rcl} \texttt{get}(v^\perp, \epsilon) & = & v^\perp \\ \texttt{get}\big((\ldots, v_k^\perp, \ldots), k; \overline{k}\big) & = & \texttt{get}(v_k^\perp, \overline{k}) \end{array}$$

Definition 6.4 (insert). The partial function $\texttt{put}(\mathsf{v}_1^\perp, \overline{\mathsf{k}}, \mathsf{v}_2^\perp)$ overwrites the value at a given sublocation:

$$\begin{array}{rcl} \texttt{put}(\mathsf{v}_1^\perp, \epsilon, \mathsf{v}_2^\perp) & = & \mathsf{v}_2^\perp \\ \texttt{put}\big((\ldots,\mathsf{v}_\mathsf{k}^\perp,\ldots),\mathsf{k};\overline{\mathsf{k}},\mathsf{v}^\perp\big) & = & (\ldots,\mathsf{v}_\mathsf{n}^\perp,\ldots) \quad \textbf{where } \mathsf{v}_\mathsf{n}^\perp = \texttt{put}(\mathsf{v}_\mathsf{k}^\perp,\overline{\mathsf{k}},\mathsf{v}^\perp) \end{array}$$

Thus, for example, we have $\texttt{get}\big((\texttt{true},\texttt{false}),1\big) = \texttt{false}$ whilst $\texttt{get}\big((\texttt{true},\texttt{false}),2\big)$ is undefined. Likewise, $\texttt{put}\big((\texttt{false},\texttt{false}),1,\texttt{true}\big) = (\texttt{false},\texttt{true})$, etc. Following this, we now extend $\texttt{loc}(\mathcal{S},\mathsf{w})$ from Definition 3.1 (page 14) with support for sublocations and tuple indices:

Definition 6.5 (Extended Locate). The partial function $\texttt{loc}(\mathcal{S},\mathsf{w})$ determines the location associated with a given lval in a given store:

$$\begin{array}{rcll} \texttt{loc}(\mathcal{S},\mathsf{x}) & = & \ell_\mathsf{x} & \\ \texttt{loc}(\mathcal{S},*\mathsf{w}) & = & \ell^{\overline{\mathsf{j}}} & \textbf{where } \texttt{loc}(\mathcal{S},\mathsf{w}) = \ell_\mathsf{w}^{\overline{\mathsf{k}}} \textbf{ and } \texttt{get}(\mathcal{S}(\ell_\mathsf{w}),\overline{\mathsf{k}}) = \ell^{\overline{\mathsf{j}}*} \\ \texttt{loc}(\mathcal{S},\mathsf{w}.\mathsf{k}) & = & \ell^{\overline{\mathsf{k}};\mathsf{k}} & \textbf{where } \texttt{loc}(\mathcal{S},\mathsf{w}) = \ell^{\overline{\mathsf{k}}} \end{array}$$

In the above, observe that the case for dereferences also changed from the original definition, since it must now support dereferencing a reference to a sublocation. In a similar fashion, the functions $\texttt{read}(\mathcal{S},\mathsf{w})$ (Definition 3.2) and $\texttt{write}(\mathcal{S},\mathsf{w},\mathsf{v}^\perp)$ (Definition 3.23) are updated as follows:

Definition 6.6 (Extended Read). The partial function $\texttt{read}(\mathcal{S},\mathsf{w})$ retrieves the valued at a given lval:

$$\texttt{read}(\mathcal{S},\mathsf{w}) \quad = \quad \texttt{get}(\mathcal{S}(\ell),\overline{\mathsf{k}}) \quad \textbf{where } \texttt{loc}(\mathcal{S},\mathsf{w}) = \ell^{\overline{\mathsf{k}}}$$

Definition 6.7 (Extended Write). The partial function $\texttt{write}(\mathcal{S},\mathsf{w},\mathsf{v}^\perp)$ updates the value at a given lval:

$$\begin{array}{rcll} \texttt{write}(\mathcal{S},\mathsf{w},\mathsf{v}_1^\perp) & = & \mathcal{S}[\ell_\mathsf{w} \mapsto \langle \mathsf{v}_3^\perp \rangle^\mathsf{m}] & \textbf{where } \texttt{loc}(\mathcal{S},\mathsf{w}) = \ell_\mathsf{w}^{\overline{\mathsf{k}}} \textbf{ and } \mathcal{S}(\ell_\mathsf{w}) = \langle \mathsf{v}_2^\perp \rangle^\mathsf{m} \\ & & & \textbf{and } \texttt{put}(\mathsf{v}_2^\perp,\overline{\mathsf{k}},\mathsf{v}_1^\perp) = \mathsf{v}_3^\perp \end{array}$$

Finally, we must update $\texttt{drop}(\mathcal{S},\psi)$ to destructure compound values so as to ensure that owning references within them are deallocated. For this, we employ a simple destructuring notation $\overline{\mathsf{v}} \in \mathsf{v}$ which extracts all immediate sublocations $\overline{\mathsf{v}}$ for a given value v. For atomic values (e.g. integers), this is always the empty sequence. For tuples we have, for example, $\mathsf{v}_1, \mathsf{v}_2 \in (\mathsf{v}_1, \mathsf{v}_2)$ and $\mathsf{v}_1, (\mathsf{v}_2, \mathsf{v}_3) \in \big(\mathsf{v}_1, (\mathsf{v}_2, \mathsf{v}_3)\big)$, etc. This is also defined for partial values in the obvious fashion, hence $\mathsf{v}, \perp \in (\mathsf{v}, \perp)$, etc.

Definition 6.8 (Extended Drop). The function $\texttt{drop}(\mathcal{S},\psi)$ recursively deallocates owned locations as follows:

$$\texttt{drop}(\mathcal{S},\psi \cup \{\mathsf{v}^\perp\}) \quad = \quad \begin{array}{c}\ldots\\ \texttt{drop}(\mathcal{S},\psi \cup \{\overline{\mathsf{v}^\perp}\}) \\ \ldots\end{array} \quad \textbf{if } \mathsf{v}^\perp \neq \ell^\bullet \textbf{ where } \overline{\mathsf{v}^\perp} \in \mathsf{v}^\perp$$

The other two cases for this function are unchanged from Definition 3.4 (recall page 16). At this stage, we now have a simple and powerful system for handling the evaluation of compound values which can be easily extended beyond tuples.

6.2.2 Typing. To preserve the ownership invariant, the typing rule for tuple constructors must ensure the type of each operand is temporarily included in the environment when typing remaining operands. As before, this is done using temporary fresh locations and we introduce some shorthand notation, referred to as *carry typing*, for this:

Definition 6.9 (Carry Typing). Let Γ and Γ' be typing environments and $\mathtt{l}$ a lifetime. Let $\overline{\mathtt{t}}$ be a sequence of zero or more terms and $\overline{\mathtt{T}}$ a matching sequence of types. Then, the *carry typing* over $\overline{\mathtt{t}}$ gives a left-to-right typing of terms, denoted $\Gamma \vdash \langle\ \overline{\mathtt{t} : \mathtt{T}}\ \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma'$, and defined as a partial function $\Gamma' = \Gamma_n - \{\overline{\gamma \mapsto \Gamma_n(\gamma)}\}$ where $\Gamma_n = \mathtt{carry}^0(\Gamma, \overline{\mathtt{t}})$:

$$\begin{array}{rcl} \mathtt{carry}^k(\Gamma_k, \emptyset) & = & \Gamma_k \\ \mathtt{carry}^k(\Gamma_k, \mathtt{t}_k\ \overline{\mathtt{t}}) & = & \mathtt{carry}^{k+1}(\Gamma'_k[\gamma_k \mapsto \langle \mathtt{T}_k \rangle^{\mathtt{l}}], \overline{\mathtt{t}}) \quad \textbf{where } \Gamma_k \vdash \langle\ \mathtt{t}_k : \mathtt{T}_k\ \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma'_k \end{array}$$

In the above, γ_k represents a sequence of anonymous variables freshly introduced during the evaluation of $\mathtt{carry}(\Gamma, \overline{\mathtt{t}})$. As alluded to above, the key difficulty when typing a tuple $(\mathtt{t}_1, \ldots, \mathtt{t}_n)$ arises when some $\mathtt{t}_k$ acquires a mutable reference (e.g. to a variable $\mathtt{x}$) which must be carried through the remaining terms. For example, with $\Gamma_k \vdash \langle\ \mathtt{t}_k : \mathtt{T}_k\ \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma'_k$ the effect of $\mathtt{t}_k$ cannot be reflected in Γ'_k since its result is "in flight" — i.e. has not yet been assigned to some location. However, by assigning $\mathtt{T}_k$ to an anonymous variable γ_k in Γ'_k, we bring to bear all information about borrowed locations when typing $\mathtt{t}_{k+1}$, etc. With carry typing, the rule for typing tuples is straightforward:

$$\frac{\Gamma_1 \vdash \langle\ \overline{\mathtt{t} : \mathtt{T}}\ \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2}{\Gamma_1 \vdash \langle\ (\overline{\mathtt{t}}) : (\overline{\mathtt{T}})\ \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2} \quad \text{(T-Tuple)}$$

In addition to T-Tuple, we must also update a large number the various support functions used within other typing rules and we now detail these. Perhaps the simplest of these is Definition 3.6 (page 20) for identifying which types are *copy*:

Definition 6.10 (Extended Copy Types). A type $\mathtt{T}$ has copy semantics, denoted by $\mathtt{copy}(\mathtt{T})$, when $\mathtt{T} = \mathtt{int}$ or $\mathtt{T} = \&\overline{\mathtt{w}}$ or $\mathtt{T} = (\mathtt{T}_1, \ldots, \mathtt{T}_n)$ and $\bigwedge_i \mathtt{copy}(\mathtt{T}_i)$.

This defines the copy status of a tuple in terms of its elements. Thus, for example, $\mathtt{copy}(\mathtt{int}, \mathtt{int})$ and $\mathtt{copy}\big((\mathtt{int}, \&\mathtt{x}, \&\mathtt{y})\big)$ hold but not $\mathtt{copy}\big((\mathtt{int}, \&\mathtt{mut}\ \mathtt{x})\big)$ or $\mathtt{copy}\big((\Box\mathtt{int}, \mathtt{int})\big)$, etc.

Definition 6.11 (Extended Type Strengthening). Let $\widetilde{\mathtt{T}}_1$ and $\widetilde{\mathtt{T}}_2$ be partial types. Then $\widetilde{\mathtt{T}}_1$ *strengthens* $\widetilde{\mathtt{T}}_2$, denoted as $\widetilde{\mathtt{T}}_1 \sqsubseteq \widetilde{\mathtt{T}}_2$, according to the following rules:

$$\cdots \qquad\qquad \cdots$$

$$\frac{\widetilde{\mathtt{T}}_1 \sqsubseteq \widetilde{\mathtt{S}}_1 \cdots \widetilde{\mathtt{T}}_n \sqsubseteq \widetilde{\mathtt{S}}_n}{(\widetilde{\mathtt{T}}_1, \ldots, \widetilde{\mathtt{T}}_n) \sqsubseteq (\widetilde{\mathtt{S}}_1, \ldots, \widetilde{\mathtt{S}}_n)}\ \text{(W-TupA)} \qquad \frac{\widetilde{\mathtt{T}}_1 \sqsubseteq \lfloor \mathtt{S}_1 \rfloor \cdots \widetilde{\mathtt{T}}_n \sqsubseteq \lfloor \mathtt{S}_n \rfloor}{(\widetilde{\mathtt{T}}_1, \ldots, \widetilde{\mathtt{T}}_n) \sqsubseteq \lfloor (\mathtt{S}_1, \ldots, \mathtt{S}_n) \rfloor}\ \text{(W-TupB)}$$

This updates Definition 3.7 (page 20) for strengthening tuple types, where cases for other types are unchanged from the original definition. This in turn extends the type join operator (Definition 3.8, page 20) so, for example, $(\mathtt{int}, \&\mathtt{x}) \sqcup (\mathtt{int}, \&\mathtt{y})$ gives $(\mathtt{int}, \&\mathtt{x},\mathtt{y})$, etc.

Definition 6.12 (Extended LVal Typing). An lval $\mathtt{w}$ is said to be typed with respect to an environment Γ, denoted $\Gamma \vdash \mathtt{w} : \langle \widetilde{\mathtt{T}} \rangle^{\mathtt{m}}$, according to the following rules:

$$\cdots \qquad \cdots \qquad \cdots \qquad \frac{\Gamma \vdash \mathtt{w} : \langle (\ldots, \widetilde{\mathtt{T}}_k, \ldots) \rangle^{\mathtt{m}}}{\Gamma \vdash \mathtt{w}.\mathtt{k} : \langle \widetilde{\mathtt{T}}_k \rangle^{\mathtt{m}}}\ \text{(T-LvTup)}$$

This updates Definition 3.11 (page 21) for typing tuple lvals in the expected fashion. Thus, for example, $\Gamma \vdash \mathtt{x}.1 : \langle \Box\mathtt{int} \rangle^{\mathtt{m}}$ and $\Gamma \vdash *(\mathtt{x}.1) : \langle \mathtt{int} \rangle^{\mathtt{m}}$ where $\Gamma(\mathtt{x}) = \langle (\mathtt{int}, \Box\mathtt{int}) \rangle^{\mathtt{m}}$.

Definition 6.13 (Extended Type Containment). Let Γ be an environment where $\Gamma(\mathtt{x}) = \langle \widetilde{\mathtt{T}} \rangle^{\mathtt{l}}$ for some $\mathtt{l}$. Then, $\Gamma \vdash \mathtt{x} \rightsquigarrow \mathtt{T}_y$ denotes that variable $\mathtt{x}$ *contains* type $\mathtt{T}_y$, and is defined as $\mathtt{contains}(\Gamma, \widetilde{\mathtt{T}}, \mathtt{T}_y)$ where:

$$\mathtt{contains}(\Gamma, \widetilde{\mathtt{T}}, \mathtt{T}_y) = \begin{cases} \ldots \\ \bigvee_k \mathtt{contains}(\Gamma, \widetilde{\mathtt{T}}_k, \mathtt{T}_y) & \textbf{if } \widetilde{\mathtt{T}} = (\widetilde{\mathtt{T}}_1, \ldots, \widetilde{\mathtt{T}}_n) \\ \ldots \end{cases}$$

This extends Definition 3.15 (page 21) so that type containment explores tuple elements when looking for a specific type. Thus, for example, $\Gamma \vdash \mathtt{x} \rightsquigarrow \&\mathtt{y}$ when $\Gamma = \{\mathtt{x} \mapsto \langle(\mathtt{int}, \&\mathtt{y})\rangle^{\mathtt{l}}, \ldots\}$ allowing us to identify that $\mathtt{y}$ is write prohibited in this context.

Definition 6.14 (Extended Path Selector). A *path selector*, ρ, is either a *dereference* ($\rho \triangleq *$) or a *tuple selector* ($\rho \triangleq \mathtt{n}$, for some natural $\mathtt{n}$).

Definition 6.15 (Extended Path Conflict). Let $\mathtt{u} \triangleq \pi_{\mathtt{u}} \mid \mathtt{x}$ and $\mathtt{w} \triangleq \pi_{\mathtt{w}} \mid \mathtt{y}$ be lvals. Then, $\mathtt{w}$ is said to *conflict* with $\mathtt{u}$, denoted $\mathtt{u} \bowtie \mathtt{w}$, if $\mathtt{x} = \mathtt{y}$ and either $\pi_{\mathtt{u}} = (\pi \cdot \pi_{\mathtt{w}})$ or $\pi_{\mathtt{w}} = (\pi \cdot \pi_{\mathtt{u}})$ for some π.

Here, the intuition is that lvals referring to different elements of the same tuple do not conflict. Thus, $\mathtt{x} \bowtie \mathtt{x}$, $*\mathtt{x} \bowtie \mathtt{x}$, $** \mathtt{x} \bowtie *\mathtt{x}$ all conflict as before, as do $\mathtt{x} \bowtie \mathtt{x}.1$, $(*\mathtt{x}).1 \bowtie (*\mathtt{x})$ now, etc. However, for example, $\mathtt{x}.0 \not\bowtie \mathtt{x}.1$ and $(*\mathtt{x}).0 \not\bowtie (*\mathtt{x}).1$ do not conflict.

Definition 6.16 (Extended Move). Let Γ be an environment where $\Gamma(\mathtt{x}) = \langle \tilde{\mathtt{T}}_1 \rangle^{\mathtt{l}}$ for some lifetime $\mathtt{l}$, and $\mathtt{w}$ an lval where $\mathtt{w} \triangleq \pi_{\mathtt{x}} \mid \mathtt{x}$. Then, $\mathtt{move}(\Gamma, \mathtt{w})$ is a partial function defined as $\Gamma[\mathtt{x} \mapsto \langle \tilde{\mathtt{T}}_2 \rangle^{\mathtt{l}}]$ where $\tilde{\mathtt{T}}_2 = \mathtt{strike}(\pi_{\mathtt{x}} \mid \tilde{\mathtt{T}}_1)$:

$$\mathtt{strike}\big((\pi \cdot \mathtt{k}) \mid (\tilde{\mathtt{T}}_1, \ldots, \tilde{\mathtt{T}}_{\mathtt{k}}, \ldots \tilde{\mathtt{T}}_{\mathtt{n}})\big) \quad \overset{\cdots}{=} \quad (\tilde{\mathtt{T}}_1, \ldots, \tilde{\mathtt{T}}'_{\mathtt{k}}, \ldots \tilde{\mathtt{T}}_{\mathtt{n}}) \quad \textbf{where } \tilde{\mathtt{T}}'_{\mathtt{k}} = \mathtt{strike}(\pi \mid \tilde{\mathtt{T}}_{\mathtt{k}})$$

This updates Definition 3.18 (page 22) to account for tuple indexing, whilst the other two rules are unchanged from the original definition. Observe that the definition allows moving out from a partial type. For example, $\mathtt{move}(\Gamma, \mathtt{x}.1)$ gives $(\lfloor \mathtt{int} \rfloor, \lfloor \square \mathtt{int} \rfloor)$ when $\Gamma(\mathtt{x}) = \langle(\lfloor \mathtt{int} \rfloor, \square \mathtt{int})\rangle^{\mathtt{l}}$.

Definition 6.17 (Extended Mutability). Let Γ be an environment where $\Gamma(\mathtt{x}) = \langle \tilde{\mathtt{T}} \rangle^{\mathtt{l}}$ for some lifetime $\mathtt{l}$, and $\mathtt{w}$ an lval where $\mathtt{w} \triangleq \pi_{\mathtt{x}} \mid \mathtt{x}$. Then, $\mathtt{mut}(\Gamma, \mathtt{w})$ is a partial function defined as $\mathtt{mutable}(\Gamma, \pi_{\mathtt{x}} \mid \tilde{\mathtt{T}})$:

$$\mathtt{mutable}\big(\Gamma, (\pi \cdot \mathtt{k}) \mid (\ldots, \mathtt{T}_{\mathtt{k}}, \ldots)\big) \quad \overset{\cdots}{=} \quad \mathtt{mutable}(\Gamma, \pi \mid \mathtt{T}_{\mathtt{k}})$$

Again, the other rules are unchanged as from Definition 3.19 (page 23). This allows us to, for example, mutably borrow an lval via a tuple element when the element in question is also mutable. For example, $\mathtt{mut}(\Gamma, *(\mathtt{x}.1))$ holds when $\Gamma(\mathtt{x}) = \langle(\mathtt{int}, \square \mathtt{int})\rangle^{\mathtt{l}}$.

Definition 6.18 (Extended Compatible Shape). For an environment Γ, two partial types $\tilde{\mathtt{T}}_1$ and $\tilde{\mathtt{T}}_2$ are said to be *shape compatible*, denoted as $\Gamma \vdash \tilde{\mathtt{T}}_1 \approx \tilde{\mathtt{T}}_2$, according to the following rules:

$$\cdots \qquad \cdots \qquad \cdots \qquad \cdots \qquad \frac{\overline{\Gamma \vdash \tilde{\mathtt{T}}_{\mathtt{i}} \approx \tilde{\mathtt{S}}_{\mathtt{i}}}}{\Gamma \vdash (\tilde{\mathtt{T}}_1, \ldots, \tilde{\mathtt{T}}_{\mathtt{n}}) \approx (\tilde{\mathtt{S}}_1, \ldots, \tilde{\mathtt{S}}_{\mathtt{n}})} \ (\textsc{S-Tuple})$$

Here, Definition 3.22 (page 25) extends easily to tuples, where the other rules are unchanged from the original definition. In essence, two tuples are compatible if they have equal size and corresponding elements are themselves compatible.

Definition 6.19 (Extended Well-Formed Type). For an environment Γ, a type $\mathtt{T}$ is said to be well-formed with respect to a lifetime $\mathtt{l}$, denoted $\Gamma \vdash \mathtt{T} \geq \mathtt{l}$, according to the following rules:

$$\cdots \qquad \cdots \qquad \cdots \qquad \frac{\overline{\Gamma \vdash \mathtt{T} \geq \mathtt{l}}}{\Gamma \vdash (\mathtt{T}_1, \ldots, \mathtt{T}_{\mathtt{n}}) \geq \mathtt{l}} \ (\textsc{L-Tuple})$$

Again, Definition 3.21 (page 24) extends quite naturally to handle tuples, where the other rules are unchanged from the original definition. Here, a lifetime is considered within a tuple if it is within every element of that tuple.

Definition 6.20 (Extended Write). Let Γ be an environment where $\Gamma(\mathtt{x}) = \langle \tilde{\mathtt{T}}_1 \rangle^{\mathtt{l}}$ for some lifetime $\mathtt{l}$ and lval $\mathtt{w}$ where $\mathtt{w} \triangleq \pi_{\mathtt{x}} \mid \mathtt{x}$. Then, $\mathtt{write}^{\mathtt{k}}(\Gamma, \mathtt{w}, \mathtt{T})$ is a partial function defined as $\Gamma_2[\mathtt{x} \mapsto \langle \tilde{\mathtt{T}}_2 \rangle^{\mathtt{l}}]$ for some rank $\mathtt{k} \geq 0$ where $(\Gamma_2, \tilde{\mathtt{T}}_2) = \mathtt{update}^{\mathtt{k}}(\Gamma, \pi_{\mathtt{x}} \mid \tilde{\mathtt{T}}_1, \mathtt{T})$:

$$\mathtt{update}^{\mathtt{k}}(\Gamma_1, (\pi \cdot \mathtt{j}) \mid (\tilde{\mathtt{T}}_1, \ldots \tilde{\mathtt{T}}_{\mathtt{j}}, \ldots \tilde{\mathtt{T}}_{\mathtt{n}}), \mathtt{T}) \quad \begin{matrix} \ldots \\ = \end{matrix} \quad \begin{matrix} \ldots \\ (\Gamma_2, (\tilde{\mathtt{T}}_1, \ldots \tilde{\mathtt{T}}'_{\mathtt{j}}, \ldots, \tilde{\mathtt{T}}_{\mathtt{n}})) \\ \textbf{where } (\Gamma_2, \tilde{\mathtt{T}}'_{\mathtt{j}}) = \mathtt{update}^{\mathtt{k}}(\Gamma_1, \pi \mid \tilde{\mathtt{T}}_{\mathtt{j}}, \mathtt{T}) \end{matrix}$$

Again, the other rules are unchanged as from Definition 3.23 (page 26). Observe this allows writes into partial types which yield partial types. For example, suppose $\Gamma = \{\mathtt{x} \mapsto \langle (\lfloor \mathtt{int} \rfloor, \square \mathtt{int}) \rangle^{\mathtt{l}}\}$ (i.e. that $\mathtt{x}$ is a tuple value whose first element is undefined). Then, $\mathtt{write}^{0}(\Gamma, \mathtt{x}.0, \mathtt{int})$ gives $\Gamma = \{\mathtt{x} \mapsto \langle (\mathtt{int}, \square \mathtt{int}) \rangle^{\mathtt{l}}\}$ where the first element of $\mathtt{x}$ is no longer undefined.

6.2.3 Discussion. At this point, there is relatively little remaining to do in updating the soundness proof, such as updating the various case analyses and adding an appropriate rule to Definition 4.4. We note also that our reference implementation includes an extension for tuples.

6.3 EXTENSION: Functions

We now sketch an approach for adding function declarations and invocations to FR. A key issue here is the lack of support for *stack frames* in FR, meaning that a given variable (e.g. x) can only be instantiated once. This arises because of the $1-1$ mapping between named locations (e.g. $\ell_{\mathtt{x}}$) and variable identifiers (e.g. x). Of course, in the presence of recursion, a given function parameter x may be instantiated arbitrarily many times and the usual approach is to employ a stack of frames binding variables to locations. In fact, FR already has most of the machinery required for this. If an explicit stack of frames were used, there would be one frame for every block currently executing. The key insight is that the executing blocks themselves form a stack and, hence, a separate stack is unnecessary. Since each block is associated with a lifetime l, this can uniquely identify variables declared within. That is, a variable x declared in lifetime l now maps to a location $\ell_{\mathtt{l::x}}$. Consider the following:

$$\emptyset \triangleright \{\ \mathtt{let\ mut\ x = 1;}\ \{\ \mathtt{let\ mut\ x = 2;\ let\ mut\ y = x}\ \}^{\mathtt{m}}\ \}^{\mathtt{l}} \tag{24}$$

Executing this program for several steps would yield the following state:

$$\{\ell_{\mathtt{l::x}} \mapsto \langle 1 \rangle^{\mathtt{l}}, \ell_{\mathtt{m::x}} \mapsto \langle 2 \rangle^{\mathtt{m}}\} \triangleright \{\ \{\ \mathtt{let\ mut\ y = x;}\ \}^{\mathtt{m}}\ \}^{\mathtt{l}} \tag{25}$$

Here, two locations have been created for x — one for the outer declaration, and one of the inner. To execute the final statement, we must resolve x to the correct location. We can exploit the nesting of lifetimes here. That is, when resolving a variable x in lifetime m we first look for location $\ell_{\mathtt{m::x}}$. If this doesn't exist, we explore the strictly enclosing lifetime (i.e. looking for $\ell_{\mathtt{l::x}}$ next, etc). This continues until either x is resolved or no resolution exists and the program is stuck.

An interesting question is how function invocation could be handled in FR. The idea is to simply inline the function body as a block. To understand this, imagine executing $\{\mathtt{f}();\}^{\mathtt{l}}$ where $\mathtt{f} \triangleq \{\overline{\mathtt{t}}\}^{\mathtt{m}}$. Inlining the body gives $\{\{\overline{\mathtt{t}}\}^{\mathtt{m}}\}^{\mathtt{l}}$, but there are two problems: firstly, this is not well formed unless (by chance) $\mathtt{l} \geq \mathtt{m}$; secondly, if $\mathtt{f} \triangleq \{\mathtt{f}();\}^{\mathtt{m}}$, then repeated execution yields $\{\{\{\ldots\}^{\mathtt{m}}\}^{\mathtt{m}}\}^{\mathtt{l}}$ which is also inconsistent. The solution here is to *substitute* lifetimes upon inlining. That is, when inlining $\{\overline{\mathtt{t}}\}^{\mathtt{m}}$ into $\{\ldots\}^{\mathtt{l}}$ we substitute m for a *fresh* lifetime n where $\mathtt{l} \geq \mathtt{n}$. Thus, repeated inlining of the same block (i.e. for recursion) yields instances of the same block with unique lifetimes.

6.3.1 Syntax & Semantics. Figure 7 illustrates the syntactic extensions required to support function declarations and invocations in FR. The following illustrates a simple program:

$$\mathtt{fn\ id(mut\ x : \&'a\ int) \rightarrow \&'a\ int\ \{\ x\ \}^{m}\ \{let\ mut\ u = 0;\ let\ mut\ v = id(\&u);\}^{l}} \tag{26}$$

p ::=		*Programs*	S ::=		*Signatures*
	fn f($\overline{\texttt{mut x : S}}$) → S{$\overline{\texttt{t}}$}$^{\texttt{l}}$ p	*Function*		ϵ	*unit*
	t	*Main*		int	*integer*
				&'l mut S	*mutable borrow*
t ::=		*Terms*		&'l S	*immutable borrow*
	...			□S	*box*
	f($\overline{\texttt{t}}$)				

Fig. 7. Syntactic extensions to FR as required for function declarations and invocations.

Note that, for consistency, our syntax requires parameters be `mut` and avoids explicit lifetime parameters for convenience. Whilst the latter could be added (along with checks to ensure parameter signatures only use declared lifetimes) this adds unnecessary clutter to the typing rules.

We extend the small-step semantics with a declaration context, $\mathcal{D}$, such that all reductions have the form $\langle\ \mathcal{D}; \mathcal{S} \triangleright \texttt{t} \longrightarrow \mathcal{D}'; \mathcal{S}' \triangleright \texttt{t}'\ \rangle^{\texttt{l}}$. Furthermore, we tacitly assume all existing rules of FR are extended to this form in the obvious way. Using this, programs are executed as follows:

$$\frac{\mathcal{D}' = \mathcal{D}[\texttt{f} \mapsto \lambda(\overline{\texttt{x}})\{\overline{\texttt{t}}\}^{\texttt{m}}]}{\langle\ \mathcal{D}; \mathcal{S} \triangleright \texttt{fn f}(\overline{\texttt{mut x : S}}) \rightarrow \texttt{S}\{\overline{\texttt{t}}\}^{\texttt{m}}\ \texttt{p} \longrightarrow \mathcal{D}'; \mathcal{S} \triangleright \texttt{p}\ \rangle^{\texttt{l}}} \qquad \text{(R-Program)}$$

This essentially loads functions into the declaration context until, eventually, only a term `t` remains to be reduced as before. The rule for reducing invocations is thus:

$$\frac{\mathcal{D}(\texttt{f}) = \lambda(\overline{\texttt{x}})\{\overline{\texttt{t}}\}^{\texttt{m}} \quad \Theta(\texttt{l} \Rightarrow \{\overline{\texttt{t}}\}^{\texttt{m}}) = \{\overline{\texttt{t}}\}^{\texttt{n}} \quad \mathcal{S}_2 = \mathcal{S}_1[\overline{\ell_{\texttt{n::x}} \mapsto \langle \texttt{v} \rangle^{\texttt{n}}}]}{\langle\ \mathcal{D}; \mathcal{S}_1 \triangleright \texttt{f}(\overline{\texttt{v}}) \longrightarrow \mathcal{D}; \mathcal{S}_2 \triangleright \{\overline{\texttt{t}}\}^{\texttt{n}}\ \rangle^{\texttt{l}}} \qquad \text{(R-Invoke)}$$

Here, $\Theta(\texttt{l} \Rightarrow \texttt{t})$ is the *lifetime instantiation* function which recursively instantiates all lifetimes in a given term `t` to fresh lifetimes within `l`, whilst respecting the existing nesting of lifetimes (i.e. as discussed above). At this stage, it remains only to address the problem of associating a given variable `x` with its corresponding location in the program store. This is challenging as we may have multiple locations representing different instantiations of variable in the store (e.g. $\ell_{\texttt{l::x}}$ versus $\ell_{\texttt{m::x}}$). As such, we extend $\texttt{loc}(\mathcal{S}, \texttt{w})$ from Definition 3.1 (page 14) to identify the enclosing instantiation:

Definition 6.21 (Extended Locate). The partial function $\texttt{loc}(\mathcal{S}, \texttt{w})$ determines the location associated with a given lval in a given store:

$$\begin{array}{rcl} \texttt{loc}(\mathcal{S}, \texttt{x}) & = & \ell_{\texttt{m::x}} \quad \textbf{where } \mathcal{S}(\ell_{\texttt{m::x}}) = \langle\cdot\rangle^{\texttt{m}} \textbf{ and } \neg\exists \texttt{n}.\big(\texttt{m} \geq \texttt{n} \wedge \mathcal{S}(\ell_{\texttt{n::x}}) = \langle\cdot\rangle^{\texttt{n}}\big) \\ & \ldots & \end{array}$$

Observe that functions $\texttt{read}(\mathcal{S}, \texttt{w})$ (Definition 3.2, page 15) and $\texttt{write}(\mathcal{S}, \texttt{w}, \texttt{v}^{\perp})$ (Definition 3.23, page 26) automatically benefit from this extension. However, we must still update R-Declare to handle the introduction of locations as follows:

$$\frac{\mathcal{S}_2 = \mathcal{S}_1[\ell_{\texttt{l::x}} \mapsto \langle \texttt{v} \rangle^{\texttt{l}}]}{\langle\ \mathcal{D}; \mathcal{S}_1 \triangleright \texttt{let mut x = v} \longrightarrow \mathcal{D}; \mathcal{S}_2 \triangleright \epsilon\ \rangle^{\texttt{l}}} \qquad \text{(R-Declare)}$$

At this point, we now have a workable mechanism for the semantics of function invocations.

6.3.2 Typing. We now consider the typing rules necessary for implementing function invocations. A key challenge lies in the connection between *signatures* and *types*. For example, consider the signature "`&'a int`" given for `x` in our `id()` function from before. This is not a type in FR though, clearly, it has some connection here. To resolve this we *lower* signatures from declarations to typing

environments, denoted $\Gamma \vdash (\overline{\mathsf{S}}) \to (\mathsf{S}) \Longrightarrow (\overline{\mathsf{T}}) \to (\mathsf{T})$. For the `id()` function we might have the lowering $\Gamma \vdash (\&'\mathtt{a}\ \mathtt{int}) \to (\&'\mathtt{a}\ \mathtt{int}) \Longrightarrow (\&\gamma) \to (\&\gamma)$ where $\Gamma = \{\gamma \mapsto \langle \mathtt{int} \rangle^{\mathsf{a}}\}$. Here, γ is an anonymous variable introduced into the environment in order to form (in this case) suitable parameter and return types. We now present the typing rules for programs and functions:

$$\frac{\begin{array}{c}\mathcal{D}_1;\Gamma_1 \vdash \langle\ \mathtt{fn}\ \mathtt{f}(\overline{\mathtt{mut}\ \mathtt{x} : \mathsf{S}}) \to \mathsf{S}\{\overline{\mathsf{t}}\}^{\mathsf{m}} : \epsilon\ \rangle^{1}_{\sigma} \dashv \mathcal{D}_1;\Gamma_1 \\ \mathcal{D}_2 = \mathcal{D}_1[\mathtt{f} \mapsto (\overline{\mathsf{S}}) \to (\mathsf{S})] \qquad \mathcal{D}_2;\Gamma_1 \vdash \langle\ \mathtt{p}\ \rangle^{1}_{\sigma} \dashv \mathcal{D}_3;\Gamma_3\end{array}}{\mathcal{D}_1;\Gamma_1 \vdash \langle\ \mathtt{fn}\ \mathtt{f}(\overline{\mathtt{mut}\ \mathtt{x} : \mathsf{S}}) \to \mathsf{S}\{\overline{\mathsf{t}}\}^{\mathsf{m}}\ \mathtt{p} : \epsilon\ \rangle^{1}_{\sigma} \dashv \mathcal{D}_3;\Gamma_3} \quad \text{(T-Program)}$$

The above is fairly straightforward and simply adds each function to the declaration context and types them. The main responsibility for typing function declarations, however, resides with a supplementary rule:

$$\frac{\begin{array}{c}\Gamma \subseteq \Gamma_1 \quad \Gamma_1 \vdash (\overline{\mathsf{S}}) \to (\mathsf{S}) \Longrightarrow (\overline{\mathsf{T}}) \to (\mathsf{T}) \\ \mathcal{D};\Gamma_1[\overline{\mathsf{x} \mapsto \mathsf{T}}] \vdash \langle\ \{\overline{\mathsf{t}}\}^{\mathsf{m}} : \mathsf{T}\ \rangle^{1}_{\sigma} \dashv \mathcal{D};\Gamma_2\end{array}}{\mathcal{D};\Gamma \vdash \langle\ \mathtt{fn}\ \mathtt{f}(\overline{\mathtt{mut}\ \mathtt{x} : \mathsf{S}}) \to \mathsf{S}\{\overline{\mathsf{t}}\}^{\mathsf{m}} : \epsilon\ \rangle^{1}_{\sigma} \dashv \mathcal{D};\Gamma} \quad \text{(T-Function)}$$

The key mechanism for lowering signatures into typing environments is non-deterministic. Observe that Γ may not include locations suitable for lowering into. Indeed, in the core calculus, it necessarily cannot hold any location suitable for lowering a borrow (i.e. because we have no `static` variables). As such, non-determinism is empowered to introduce anonymous locations (such as γ above) suitable for lowering and the constraint $\Gamma \subseteq \Gamma_1$ enables this. Observe, to ensure soundness such locations are never aliased. In other words, every unknown location implied in a signature maps to a unique anonymous location in the typing environment.

Example (Parameter Aliasing). When typing function declarations, important questions arise about the potential aliasing (or lack thereof) between parameters. Following T-Function, anonymous variables must be introduced to satisfy the lowering constraint. For example, the lowering constraint $\emptyset \vdash (\&'\mathtt{a}\ \&'\mathtt{b}\ \mathtt{mut}\ \mathtt{int}) \to (\mathtt{int}) \Longrightarrow (\cdot) \to (\mathtt{int})$ is unsatisfiable for the type parameter because no suitable variable exists in the environment to form the necessary borrow type. Furthermore, since the constraint $\Gamma \subseteq \Gamma_1$ in T-Function is non-deterministic, there are (in principle) different ways variables can be introduced (e.g. with varying aliasing structures, etc). However, in practice, the lowering process is fairly prescriptive. For example, consider the following:

$$\mathtt{fn}\ \mathtt{f0}(\mathtt{mut}\ \mathtt{p} : \&'\mathtt{a}\ \mathtt{mut}\ \&'\mathtt{b}\ \mathtt{int}, \mathtt{mut}\ \mathtt{q} : \&'\mathtt{a}\ \mathtt{mut}\ \&'\mathtt{b}\ \mathtt{int}, \mathtt{mut}\ \mathtt{r} : \&'\mathtt{b}\ \mathtt{int}) \to \mathtt{int}\ \{\ \ldots\ \}^{1} \quad (27)$$

If $\Gamma_1 = \{\gamma_1 \mapsto \langle \mathtt{int} \rangle^{\mathsf{b}}, \gamma_2 \mapsto \langle \mathtt{int} \rangle^{\mathsf{b}}, \gamma_3 \mapsto \langle \mathtt{int} \rangle^{\mathsf{b}}, \gamma_4 \mapsto \langle \&\gamma_1 \rangle^{\mathsf{a}}, \gamma_5 \mapsto \langle \&\gamma_2 \rangle^{\mathsf{a}}\}$ where (implicitly) $\mathsf{b} \geq \mathsf{a}$, then the corresponding lowering constraint is easily satisfied with $(\&\mathtt{mut}\ \gamma_4, \&\mathtt{mut}\ \gamma_5, \&\gamma_3) \to (\mathtt{int})$. Indeed, one may wonder whether it is satisfiable when $\Gamma_1 = \{\gamma_1 \mapsto \langle \mathtt{int} \rangle^{\mathsf{b}}, \gamma_2 \mapsto \langle \&\gamma_1 \rangle^{\mathsf{a}}, \gamma_3 \mapsto \langle \&\gamma_1 \rangle^{\mathsf{a}}\}$ with $(\&\mathtt{mut}\ \gamma_2, \&\mathtt{mut}\ \gamma_3, \&\gamma_1) \to (\mathtt{int})$ or even $(\&\mathtt{mut}\ \gamma_2, \gamma_3, \&\mathtt{mut}\ \gamma_2, \gamma_3, \&\gamma_1) \to (\mathtt{int})$, etc. However, whilst the former is perhaps reasonable, neither can satisify the lowering constraint simply because aliasing between variables is not permitted. We argue (without futher elaboration) that this is sound for two reasons: firstly, mutable borrows are already unique; secondly, non-local state can only be modified via mutable borrows and, hence, strong updates do not apply. ■

We now consider the typing rule for invocations itself. This employs carry typing of arguments (recall Definition 6.9) and a mechanism for *lifting* types from typing environments to signatures, denoted $\Gamma_1 \vdash (\overline{\mathsf{S}}) \to (\mathsf{S}) \Longleftarrow (\overline{\mathsf{T}}) \to (\mathsf{T}) \dashv \Gamma_2$. Here, any possible effects on the environment arising from the invocation are captured in the difference between Γ_1 and Γ_2. Lifting is more involved than lowering and requires constraint solving to allow information flow in both directions (i.e. from

signatures to types, and vice versa). To understand the difference, consider typing the invocation `id(&u)` from the example in §6.3.1. Since the environment immediately before is $\Gamma = \{\mathtt{u} \mapsto \langle \mathtt{int} \rangle^1\}$, the (solved) lifting constraint is $\Gamma \vdash (\&'\mathtt{a}\ \mathtt{int}) \to (\&'\mathtt{a}\ \mathtt{int}) \Longleftarrow (\&\mathtt{u}) \to (\&\mathtt{u}) \dashv \Gamma$. In this case, constraint solving determines the return type from the parameter types using the function signature. As such, the rule itself is given as follows:

$$\frac{\mathcal{D}(\mathtt{f}) = (\overline{\mathtt{S}}) \to (\mathtt{S}) \quad \Gamma_1 \vdash \langle\ \overline{\mathtt{t} : \mathtt{T}}\ \rangle^1_\sigma \dashv \Gamma_2 \qquad \Gamma_2 \vdash (\overline{\mathtt{S}}) \to (\mathtt{S}) \Longleftarrow (\overline{\mathtt{T}}) \to (\mathtt{T}) \dashv \Gamma_3}{\mathcal{D}; \Gamma_1 \vdash \langle\ \mathtt{f}(\overline{\mathtt{t}}) : \mathtt{T}\ \rangle^1_\sigma \dashv \mathcal{D}; \Gamma_3} \qquad \text{(T-Invoke)}$$

Example (Conservative Returns). A simple observation about lifting is that it must conservatively approximate the set of values which could be returned. The following illustrates:

$$\begin{array}{l}\mathtt{fn\ f1(mut\ p : \&'a\ mut\ int, mut\ q : \&'a\ mut\ int)} \to \mathtt{\&'a\ mut\ int\ \{\ p\ \}}^1 \\ \mathtt{\{\ let\ mut\ u = 0;\ let\ mut\ v = 0;\ let\ mut\ w = f1(\&mut\ u, \&mut\ v);\ \}}^{\mathtt{m}}\end{array} \tag{28}$$

Since we cannot know at the call-site which of the two mutable borrows `f1()` will return, the type given to `w` must *conservatively approximate both*. In this case, the lifting constraint would be solved as $\Gamma \vdash (\&'\mathtt{a}\ \mathtt{mut}\ \mathtt{int}, \&'\mathtt{a}\ \mathtt{mut}\ \mathtt{int}) \to (\&'\mathtt{a}\ \mathtt{mut}\ \mathtt{int}) \Longleftarrow (\&\mathtt{mut}\ \mathtt{u}, \&\mathtt{mut}\ \mathtt{v}) \to (\&\mathtt{mut}\ \mathtt{u}, \mathtt{v}) \dashv \Gamma$. Observe this is the only valid solution and, for example, the constraint cannot be solved with a return type of either `&mut u` or `&mut v`. ■

Example (Conservative Lifetimes). Another observation about lifting is that it must conservatively choose lifetimes as necessary. The following illustrates:

$$\begin{array}{l}\mathtt{fn\ f2(mut\ p : \&'a\ mut\ int, mut\ q : \&'a\ mut\ int)} \to \mathtt{\&'a\ mut\ int\ \{\ p\ \}}^1 \\ \mathtt{\{\ let\ mut\ u = 0;\ \{let\ mut\ v = 0;\ let\ mut\ w = f2(\&mut\ u, \&mut\ v);\ \}}^{\mathtt{n}}\ \}^{\mathtt{m}}\end{array} \tag{29}$$

Here, `u` and `v` have *different* lifetimes, but `f2()` expects they are the same. Nevertheless, this is solved as $\Gamma \vdash (\&'\mathtt{a}\ \mathtt{mut}\ \mathtt{int}, \&'\mathtt{a}\ \mathtt{mut}\ \mathtt{int}) \to (\&'\mathtt{a}\ \mathtt{mut}\ \mathtt{int}) \Longleftarrow (\&\mathtt{mut}\ \mathtt{u}, \&\mathtt{mut}\ \mathtt{v}) \to (\&\mathtt{mut}\ \mathtt{u}, \mathtt{v}) \dashv \Gamma$ where the lifetime `n` is bound to `'a`. ■

Example (Precise Reasoning). Another observation about lifting is that it allow precise reasoning about mutable borrows in certain circumstances. The following illustrates:

$$\begin{array}{l}\mathtt{fn\ f3(mut\ p : \&'a\ mut\ int)} \to \mathtt{int\ \{\ 0\ \}}^1 \\ \mathtt{\{\ let\ mut\ u = 0;\ let\ mut\ w = f3(\&mut\ u);\ u\ \}}^{\mathtt{m}}\end{array} \tag{30}$$

One might expect that `u` remains mutably borrowed after the invocation and, hence, this will fail. However, the lifting constraint is solved as $\Gamma \vdash (\&'\mathtt{a}\ \mathtt{mut}\ \mathtt{int}) \to (\mathtt{int}) \Longleftarrow (\&\mathtt{mut}\ \mathtt{u}) \to (\mathtt{int}) \dashv \Gamma$. Thus, knowledge of `&mut u` is not retained in the typing environment after the invocation, thereby allowing `u` to be used freely. ■

Example (Incompatible Binding). Another key observation about lifting is that it must be able to find a compatible binding for lifetimes. The following illustrates:

$$\begin{array}{l}\mathtt{fn\ f4(mut\ p : \&'a\ mut\ \&'b\ int, mut\ q : \&'a\ mut\ \&'b\ int)} \to \mathtt{int\ \{\ 0\ \}}^1 \\ \mathtt{fn\ f5(mut\ r : \&'c\ int, mut\ s : \&'d\ int)} \to \mathtt{int\ \{\ f4(\&mut\ r, \&mut\ s)\ \}}^{\mathtt{m}}\end{array} \tag{31}$$

The challenge is that `r` and `s` have different lifetimes, but `f4()` requires they are the same. This time there is no suitable lifetime available to use. Thus, we arrive at an *unsatisfiable* lifting constraint

$\Gamma \vdash (\&'\texttt{a mut } \&'\texttt{b int}, \&'\texttt{a mut } \&'\texttt{b int}) \rightarrow (\texttt{int}) \not\Longleftarrow (\&\texttt{mut r}, \&\texttt{mut s}) \rightarrow (\texttt{int}) \dashv \Gamma.$ ■

Example (Side Effects). An important challenge faced in typing function invocations is to provide a conservative handling of side effects. The following illustrates:

$$\begin{array}{l}\texttt{fn f6(mut p : \&'a mut \&'b int, mut q : \&'b : int)} \rightarrow \epsilon\ \{\ \ldots\ \}^{1} \\ \{\ \texttt{let mut x = 0;\{ let mut y = 1; let mut u = \&x; f6(\&mut u, \&y); u\}}^{n}\}^{m}\end{array} \tag{32}$$

The challenge here is that the type of u must be conservatively updated from &x (before the invocation) to &x, y (afterward). This happens regardless of the actual body given for f6(), since we must conservatively assume that $*\texttt{p} = \texttt{q}$ could occur (even if it actually does not). Thus, the typing environment beforehand is $\Gamma = \{\texttt{x} \mapsto \langle \texttt{int} \rangle^{m}, \texttt{y} \mapsto \langle \texttt{int} \rangle^{n}, \texttt{u} \mapsto \langle \&\texttt{x} \rangle^{n}\}$, and we have the satisfiable lifting constraint $\Gamma \vdash (\&'\texttt{a mut } \&'\texttt{b int}, \&'\texttt{b int}) \rightarrow (\epsilon) \Longleftarrow (\&\texttt{mut u}, \&\texttt{y}) \rightarrow (\epsilon) \dashv \Gamma[\texttt{u} \mapsto \langle \&\texttt{x}, \texttt{y} \rangle^{n}]$. As expected, however, this constraint prevents the subsequent use of u from being well typed. Observe also the constraint is only satisfiable because the type of u is updated appropriately. ■

6.3.3 Discussion. We note that work remains on extending the notion of a safe abstraction (recall Definition 4.7, page 28). The key issue is that of relating a program store at the point of an invocation with the typing environment used for typing the function in question. Unlike before, the typing environment may track fewer variables than actually present in the program store (i.e. it won't have knowledge of variables visible at the call site). We also note that function declarations could be easily extended with lifetime inclusion constraints (as for Rust), given that most of the relevant machinery is hidden within the lifting and lowering processes. Finally, our reference implementation includes an extension for functions, and this naturally provides further clarification on the lifting and lowering mechanisms employed above.

7 RELATED WORK

Over the past two decades or more, significant work has been done on the development of systems that manage aliasing (in some sense) for benefit. From the perspective of this paper, there are two main areas of relevance: that relating to *regions* (i.e. lifetimes in Rust) and that relating to uniqueness/linearity (i.e. ownership in Rust). We also examine some of the literature on Rust itself.

7.1 Rust

Perhaps most relevant here is the recent work of work Jung *et al.* which provides a machine checked safety proof for a realistic subset of Rust [70]. Their focus was on establishing safety proofs in the presence of `unsafe` code, arguing these play a fundamental role in any practical usage of Rust. For example, inside an `unsafe` block one can mutate locations via immutable references, thereby potentially breaking the ownership invariant. Similarly, methods like `String::from_utf8_unchecked()` (which converts a byte sequence into a `String`) are marked `unsafe` as they make assumptions about their inputs (in this case, that the bytes are valid `utf8` character sequences). Indeed, several bugs have arisen in libraries using `unsafe` code (some of which are subtle, requiring interactions across multiple libraries). The challenge arises when developers believe their uses of `unsafe` code are properly encapsulated when, in fact, this is not the case.

The formalisation of Jung *et al.*, called $\lambda_{\texttt{Rust}}$, employs a substructural type system and permits one to establish an appropriate verification condition for a given library using `unsafe` which, when satisfied, ensures safety of the overall system. A key challenge they faced is that the standard approach to proving safety properties — namely, *progress* and *preservation* – does not easily extend to mixing safe and unsafe code. Instead, a semantic approach in the style of Milner was adopted over

this more familiar syntactic approach [93]. In particular, this allows terms to be observed as having a type, even when they use unsafe features. The specific property achieved in $\lambda_{\texttt{Rust}}$ is that, provided unsafe code is confined to libraries which respect their verification conditions, the program is safe to execute (i.e. will not get stuck). Underpinning this development is *Iris* — a framework for high-order concurrent separation logic [71–73]. This enables, for example, a notion of *borrow propositions* which correspond with borrowing in Rust. Several notable Rust libraries using unsafe code were ported to $\lambda_{\texttt{Rust}}$ and verified as correct, including: `Arc`, `Rc`, `Cell`, `RefCell`, `Mutex`, `RwLock` and more. Finally, given the size of Rust, some language features were omitted, including traits and certain relaxed forms of atomic access (used in libraries such as `Arc` for efficiency). More specifically, a key assumption was that the language is sequentially consistent when, in fact, certain libraries (such as `Arc`) employ relaxed-memory operations. Later work adapted RustBelt to account for relaxed memory operations and, in the process, uncovered a previously unknown data race in `Arc` [39].

Compared with the work presented here, however, there are some differences from RustBelt. First and foremost, $\lambda_{\texttt{Rust}}$ does not follow the source-level syntax of Rust (unlike `FR`):

> "*Crucially, $\lambda_{\texttt{Rust}}$ incorporates Rust's notions of borrowing, lifetimes, and lifetime inclusion—which are fundamental to Rust's ownership discipline—in a manner inspired by Rust's Mid-level Intermediate Representation (MIR).*" [70]

Operating on an intermediate representation allows various simplifications compared with formalising at the source-level. For example, all control-flow is represented in $\lambda_{\texttt{Rust}}$ using continuations, while local variables at the source-level are represented using heap locations in $\lambda_{\texttt{Rust}}$ (thereby avoiding the need to distinguish between the stack and heap). For example, consider the following Rust program:

```
fn option_as_mut<'a>(x: &'a mut Option<i32>) ->Option<&'amut i32> {
    match *x {
      None => None,
      Some(ref mut t) => Some(t)
}  }
```

The above program is represented in $\lambda_{\texttt{Rust}}$ as follows [70]:

```
funrec option_as_mut(x) ret ret :=
  let r = new(2) in
  letcont k() := delete(1;x); jump ret(r) in
  let y = *x in case *y of
  - r :== (); jump k()
  - r :== y.1; jump k()
```

The purpose of $\lambda_{\texttt{Rust}}$ is also quite different and the emphasis is on a comprehensive treatment of important Rust features. Indeed, during the work itself, a bug in Rust's standard library was uncovered and fixed. The downside, however, of such a thorough treatment is that the formalisation cannot easily be digested by researchers or practitioners, either to understand the concepts of lifetimes and borrowing, or to understand the proof (which itself is around 17.5KLOC of Coq).

Another relevant work is that of Wang *et al.* who presented a formal, executable operational semantics for Rust called KRust [132]. This was defined in $\mathbb{K}$ — a rewrite-based executable semantic framework particularly suited at developing operational semantics [112]. A large subset of Rust was defined in this way and validated against 157 tests from the official Rust test suite. We note, however, that this work differs considerably from that presented here as it covers only the *executable semantics* of Rust, not the rules for type and borrow checking.

The unpublished work-in-progress of Weiss *et al.* presents a system called *Oxide* which bears some striking similarities with that presented here [133]. For example, *places* in Oxide are essentially the same as lvals in FR, whilst *shapes* give something comparable to the compound values and paths in FR. Furthermore, Oxide was also inspired by Featherweight Java to produce a relatively lean formalisation of Rust. In fact, it includes a far larger subset of Rust than the FR core (perhaps making it more *middleweight* than *featherweight* in a manner somewhat reminiscent of Middleweight Java versus Featherweight Java [16]). However, there are also differences between FR and Oxide. For example, Oxide doesn't model boxes explicitly and, as a result, has no clear means to model heap allocated memory. Likewise, the judgments used in Oxide do not model undefined types explicitly as in FR but, rather, require a separate environment for holding declared types. In addition, the proof obtained relies on an operational semantics instrumented with additional (unnecessary) runtime checks, and a subsequent lemma is used to establish they can be safely erased. Another difference is that, unlike FR, Oxide has yet to be validated against `rustc` and its relative size may render this somewhat prohibitive. Oxide and FR both treat lifetimes in a similar fashion (i.e. lexically), though a notion of *weakening* brings Oxide closer to the non-lexical lifetimes found in Rust 2018. More specifically, variables can be dropped from the typing environment at arbitrary points allowing borrows to expire early, whilst still catching cases where dropped variables were in fact live. We note that such an approach is also directly applicable to FR.

Reed provides a preliminary (though unpublished) formalisation of Rust called "Patina" [110]. This shares some similarities with our work. For example, it employs a flow-sensitive type system for characterising borrow checking which operates over a "shadow" heap (roughly akin to our typing environment). However, there are also significant differences. The scope of Patina is significantly larger than that presented here and attempts to incorporate, for example, complex reasoning about partial borrows. Likewise, Patina is concerned with detailed aspects of exactly how and when memory is released. As such, the statements of progress and preservation are formidable in their complexity. Furthermore, their soundness is not established and, instead, are presented as conjectures with an argument that they would, if proven, have *"established soundness for Patina"*.

There has also been a growing interest in exploiting Rust's safety guarantees to improve program verification tools. For example, Matsushita *et al.* exploit Rust's uniqueness guarantees to aid verification of pointer manipulating programs [90]. Their tool, RustHorn, translates Rust programs into *Constrained Horn Clauses (CHC)* (which can then be discharged by a specialised CHC solver). More specifically, the translation operates on a formalisation of Rust inspired by $\lambda_{\texttt{Rust}}$ called the *Calculus of Ownership and Reference (COR)*. Since COR resembles Rust's Mid-Level Representation (MIR) their tool translates directly from the MIR emitted by `rustc`, thereby allowing RustHorn to leverage the guarantees provided by the borrow checker. Likewise, Astrauskas *et al.* argue that formal verification of systems software has been notoriously difficult due to the complex specifications needed for reasoning about pointers and aliasing [6]. To this end, they leverage Rust's type system to simplify the specification and verification of systems software. In particular, they developed a specification language for Rust which is embedded using annotations and statically checked using Viper [95]. The SMACK verifier which translates LLVM IR to Boogie/Z3 [12, 40] was also extended to Rust [10]. This was used in developing RedLeaf, an operating system written in Rust that targets firmware [96]. Firmware is a critical component sitting underneath traditional operating systems, where flaws enable complete access to the machine. Here formal verification is easily justified and, in Redleaf, pre/post-conditions are again given as Rust annotations and, in this case, checked statically using SMACK. The CRUST tool [128] enables unsafe code to be checked using the C Bounded Model Checker (CMBC) [77]. This employs a custom C code generator for `rustc`, and correctly identified bugs arising during development of Rust's standard library. Finally

the widely-used symbolic execution tool, Klee [27], was also extended for Rust allowing assertions to be checked statically [83, 84].

Dewey *et al.* focus on fuzz testing the Rust type checker [41]. Like us, a key challenge faced is that of generating well typed programs (and, also, "almost typed" programs). Their approach was to leverage the existing power of Constraint Logic Programming tools (e.g. Prolog) which allow the encoding of constraints and the enumeration of satisfying solutions. Using this approach, they fuzz tested the Rust compiler using over 900M automatically generated programs. However, we note, they did not attempt to exhaust particular spaces of programs but, rather, simply allowed the testing process to continue up to some time limit. Nevertheless, they identified 18 bugs in the Rust type checker (most of which were confirmed by the Rust developers).

Levy *et al.* report on experiences developing an Embedded OS in Rust [80]. They argued that *"At first examination, Rust seems perfectly suited for this task"*. Unfortunately, they were hindered by ownership in Rust preventing otherwise safe resource sharing. For example, an interrupt handler could not retain a mutable borrow of a shared resource (e.g. a network stack). Such situations are not safe in general. However, in their particular setting this was safe due to guarantees provided by the OS and, to workaround, they instead relied on `unsafe` code. In subsequent work, they further reduced this `unsafe` code to a single trusted primitive, `TakeCell` [78, 79]. This is similar to `Cell` but instead of copying values out as `Cell` does (which can introduce overhead), it provides a mechanism for code to execute "within" the cell with, effectively, zero overhead. As such, it provides a form of mutual exclusion.

Jespersen *et al.* describe a library for implementing session types in Rust which was an adaptation of communication patterns in Servo [66]. Session types require a linear usage of channels which naturally fits with the ownership in Rust and, as such, afforded some safety guarantees. Finally, it is interesting to note that Rust is the primary language used to develop Mozilla's experimental rendering engine, Servo, and accounts for some 800KLOC. Anderson *et al.* examined how the use of Rust here addresses many common security issues [4]. For example, use of uninitialised memory has led to problems in Firefox. They argue many aspects of Rust (e.g. good interoperation with C) make it well suited here, but found situations where its ownership model was problematic, such as for data structures which do not assume a single owner *"in order to provide multiple traversal APIs without favoring the performance of one over the other"*.

An interesting question explored by Jung *et al.* is that of deciding what compiler optimisations should be permitted in unsafe code [69]. This is a thorny issue because, within unsafe code, the usual guarantees provided by Rust may not hold. For example, in unsafe code, multiple mutable borrows of the same location can exist. The proposed system, *Stacked Borrows*, provides an operational semantics for memory accesses in Rust. This introduces a strong notion of *undefined behaviour* such that a compiler is permitted to ignore the possibility of such programs when applying optimisations (roughly in line with how C compilers handle undefined behaviour [91]). Indeed, much previous work has focused on the issues arising with unsafe code and the problems associated with checking unsafe code. To this end, Qin *et al.* conducted an empirical evaluation into the usage of unsafe code in Rust [109]. They found, amongst other things, that: (1) unsafe code was used extensively in real-world Rust code and, generally speaking, was encapsulated from library users; and (2) that many memory safety issues were caused by incorrect reasoning about the scope of lifetimes.

7.2 Linearity and Uniqueness

The literature on linearity and uniqueness [2, 23, 25, 38, 101, 130, 131] was a significant influence on Rust. Roughly, a linear variable must be used exactly once, whilst a unique variable is the only referent to a particular object. In some sense linearity is the dual of uniqueness, which Harrington

described thusly: *in linear logic, "linear" means "will not be duplicated" whereas in uniqueness typing, "unique" means "has not been duplicated"* [58]. Linearity dates back to Girard [51] and Wadler [130], whilst uniqueness was perhaps first encountered in Clean [11]. Both systems allowed, amongst other things, safe memory deallocation without garbage collection. One noticeable difference is seen when reading elements from an array: a linear variable can perform only one array read (since that constitutes a use), whilst a unique variable can make arbitrary many reads (as they don't violate the uniqueness invariant). Linear systems work around this by adopting a programming style where linear variables are *explicitly threaded* through programs. We can illustrate this with the following Rust code:

```
fn get(arr : [Item;2], i : usize) -> (i32,[Item;2]) { return (arr[i].v,arr); }
```

Here, `get()` is given ownership of `arr` by the caller but also *gives it back*. Thus, we can thread the array through calls to `get()` as follows:

```
...
let (first,arr) = get(arr,0);
let (second,_) = get(arr,1);
...
```

Since this code yields ownership to `get()` but then regains it, subsequent invocations with the same array are permitted.

Despite its benefits, researchers recognised early on that strict linearity is often too much. Wadler, for example, permitted multiple immutable references through a specialised `let!` construct [130]. Likewise, Kobayashi exploited evaluation order and static analysis to relax linearity on variables which do not "escape" their local context [76]. Fähndrich and DeLine lessened the distinction between linear and non-linear types through their *adoption* and *focus* constructs [46]. Here, adoption allows multiple aliases for a linear type within a limited scope and bears striking similarities with borrowing in Rust. They described it thusly: *"our approach to invalidating aliases is to tie the lifetime of the aliases to the lifetime of the adopter"* (the adopter being the scope of the borrow). Finally, their focus construct does roughly the opposite by enabling a non-linear type to be temporarily viewed as linear, provided other aliases to it cannot be witnessed during the focus. Cogent provides an interesting and practical instantiation of these ideas which aims to "*significantly reducing the cost of formal verification for important classes of systems code*" [100]. For example, Cogent employs Wadler's `let!` construct to eliminate (in many cases) the need to explicitly thread linear variables through functions (as discussed above). Likewise, specific support is included for accessing fields of (linear) records — namely, after such an access, the record type is updated so as to prohibit accessing the same field, whilst still permitting access to other fields.

Much of the early work on linearity/uniqueness considers functional settings without assignment. For an imperative language, uniqueness requires some notion of a *destructive read*. That is, after a unique variable is read, its contents are set to `null` (or equivalent). This is necessary because type checking for such languages is typically performed in a *flow insensitive* fashion and, thus, cannot update the typing environment for statements downstream. Such an approach is less than ideal since it effectively means uniqueness is checked at runtime (i.e. because subsequent uses generate `NullPointerException`s). This contrasts with the approach taken in Rust's borrow checker which is *flow sensitive*. This permits an alternative to destructive reads whereby the borrow checker actually *rejects* programs which break the uniqueness invariant. Boyland's *alias burying* approach is particularly relevant here [23]. His system was designed to be used on existing languages without changing their semantics, and employed annotations on fields and variables to specify uniqueness.

Most importantly, from our perspective, is that it employed a flow-sensitive static analysis to ensure variables were not used in a way that breaks the uniqueness invariant. Finally, his system also supported a limited form of borrowing in annotated instance methods that simply prevents object receivers from being stored in the heap.

More recently, the Mezzo language attempts to incorporate linearity into an ML-like language (which, in fact, compiles down to OCaml) [7]. This language has a particular focus on eliminating race conditions and takes a permissions-based approach. Roughly speaking, permissions are akin to type tags which, in conjunction with structural typing, enable a form of typestate programming [120, 121]. A key fundamental is the ability to specify a *duplicable* data type (i.e. one which is immutable and may be copied) versus a *mutable* (i.e. linear) data type. To make this work, a flow-sensitive type checker determines the set of permissions at each program point and, for greater expressiveness, the *adoption* and *abandonment* mechanism enables borrowing and the construction of cyclic structures. In a similar vein, Linear Haskell provides a backwards-compatible integration of linear typing in Haskell with an aim to enabling safe in-place updates of compound structures (e.g. mutable arrays) [15]. This was achieved by introducing the notion of a *linear function* which provides guarantees about how its arguments are used. For soundness, the underlying meta-theory employed so-called *multiplicities* in a style reminiscent of fractional permissions [24]. However, again, such multiplicities must be inferred *a priori* using a separate (flow sensitive) type inference algorithm. Finally, the work of Filliâtre *et al.* captures the underlying type system of Why3 and shares some similarities with that presented here [18, 48]. Their system aims to retain the simplicity of Hoare logic in the presence of aliasing using a form of linearity. For example, static region identifiers are embedded in types for reasoning about aliasing in a similar fashion to how lifetimes are used here. Likewise, the type system employs a flow-sensitive treatment of effects. This includes, amongst other things, support for reasoning about variable (non)liveness using a special "reset" effect which is roughly analogous to the undefined types, $\lfloor \mathsf{T} \rfloor$, used here for the same reason (recall §3.4).

7.2.1 Balloons. The evolution of C++ was also likely a key influence on the design of Rust. Smart pointers have been part of C++ for around twenty years (at the time of writing) [92]. The most infamous of these is `auto_ptr` which acts like a unique pointer in many ways. An `auto_ptr` should be the only reference to a given object and is responsible for its deallocation. Upon assignment an `auto_ptr` transfers ownership to the destination variable and sets itself to `null`. This was achieved through clever use of operator overloading and copy constructors. However, problems arose because `auto_ptr` supported copy semantics which, given its purpose, does not make sense and was easily misused by inexperienced programmers. Support for move semantics came later in C++11 with the addition of *move constructors* and *rvalue references*. Following this, `auto_ptr` was deprecated and replaced with `unique_ptr` which has no copy constructor and supports only move semantics. To enable flexibility, `unique_ptr` allows the programmer access to the underlying pointer via `get()`, thereby essentially providing an ad-hoc mechanism for borrowing.

Almeida was an early proponent of enforcing strong encapsulation arguing that the *"pervading possibility of sharing state is what makes it difficult to reason about programs in procedural or object-oriented languages"* [3]. His goal was to *"make the ability of sharing state a first class property of data types"* and his approach bears ressemblance to the issues being grappled with in C++. Specifically, he developed the concept of a *balloon* which, roughly speaking, represents an aggregate of objects with a specific root object acting as a gateway to those inside. The key invariant is that internal objects are guaranteed not to be referenced by objects external to the balloon. Of relevance here is the requirement that a balloon root be uniquely referenced by other objects to prevent against "accidental" sharing amongst different objects. As such, a balloon reference is similar to a unique reference though, curiously, the uniqueness requirement did not extend to

local variables (presumably for reasons of flexibility). The presentation of Almeida distinguished reference assignment (e.g. `x.v :- b`) from copy assignment (e.g. `x.v := b`). As such, reference assignment to a field was a compile-time error when the right-hand side had balloon type (i.e. as it would break uniqueness). However, copy assignment was permitted for balloons with a semantics that required a deep copy (though, unfortunately, how this could be implemented efficiently was not addressed).

The general idea of balloons has since been developed in a variety of ways [36, 52, 56, 117, 118]. The work of Gordon *et al.* is notable amongst these as having been tested in the context of a large industrial project [52]. Their system introduces `isolated` references (a.k.a balloons) with `readable` (i.e. read-only) and `immutable` references. An `isolated` reference and its reachable cluster are guaranteed externally unique for all heap and stack variables. A mechanism for implicit conversion of `isolated` references to other reference types is provided for usability. For example, an `isolated` reference can be implicitly converted to an `immutable` reference. To support this without destructive reads, the authors employ a flow-sensitive type system. The following illustrates:

```
isolated Shape iso = ...;
immutable Shape imm = iso;
```

Here, the assignment has a flow-sensitive effect on variable `iso` meaning it can no longer be treated as `isolated` (as, otherwise, it could then be modified). One subtlety is that the system is flow-sensitive for local variables *but not for fields*, the latter being conservatively handled using destructive reads (i.e. as reasoning flow-sensitively about heap data is challenging). A key contribution of their work is the observation that one can safely *recover* isolation in certain conditions. This is similar, in some sense, to the recovery of ownership after a borrow in Rust and, indeed, the authors comment that *"recovering isolation is reminiscent of borrowing"*. Finally, we note their system is surprisingly powerful and can, for example, safely describe the construction of cyclic immutable object structures.

Clarke and Wrigstad combine ownership and uniqueness to form what they call "external uniqueness" [38]. Their approach arrives at something similar to Almeida's balloons, where references are categorised as either external or internal. A `unique` external reference is the only external reference to that object, whilst internal references are considered "innocuous" and permitted regardless. One of the primary motivations here is the treatment of `this` which, without such consideration, cannot be internally assigned at any point which (amongst other things) prohibits back-links in linked data structures (e.g. doubly-linked lists). For example, under Almeida's formulation parent links are not permitted to the root of a balloon. To define the concept of inside versus outside, the authors employ ownership types (see below) to give more fine-grained control (compared with Almeida's balloons where inside/outside is a binary notion). Regarding movement of unique references, they adopt destructive reads (though acknowledge other options, such as alias burying). Their system also supports borrowing within specific lexical scopes where, for the duration of the borrow, the borrowed variable is effectively frozen.

7.2.2 Ownership. Ownership types and related systems attempt — in a similar fashion to Balloons — to provide strong guarantees about when and where aliasing is permitted between objects [2, 21, 29, 35, 37, 44, 62, 82, 85, 98, 99, 101, 106, 118, 136]. They have found use in areas such as: parallel and concurrent systems [21, 22], specification languages [13, 94], real-time systems [5, 107], and more. In general, ownership systems focus on mechanisms which restrict the shape of the object graph for the purposes of enforcing strong encapsulation. For example, in the *owners-as-dominators* protocol (e.g. [21, 35, 37, 106]) no reference to an owned object can be external to its owner and, hence, strong encapsulated is achieved. For example:

```
public class LinkedList {
  private @Owned Link next;
  ...
}
```

Here, the `@Owned` qualifier above indicates that every `LinkedList` instance *owns* the `Link` object referred to by `next` — so, the only references to the `Link` object are from the `LinkedList` itself. The `@Owned` qualifier also applies transitively, meaning that objects owned by a `Link` reachable from a `LinkedList` are also owned by it. Thus, we can be sure that the objects owned (either directly or indirectly) by two distinct `LinkedList` objects, l_1 and l_2, are disjoint. However, we cannot say anything about objects which are not owned by l_1 and l_2 and, hence, their reachable object graphs may overlap. Finally, ownership systems are not generally concerned with memory management. For example, they do not generally enforce a notion of uniqueness though, of course, hybrid systems do exists (e.g. [38]). Perhaps one similarity is that ownership is typically applied *transitively* which, in Rust, means that chains of owned objects can be deallocated together.

7.3 Regions

A significant body of work exists on restricting aliasing for memory management, based around the concept of a memory *region* [9, 28, 32, 33, 45, 54, 55, 59, 60, 68, 103, 108, 124, 125, 127, 129]. Early work focused around the use of regions for improving performance of Standard ML [126]. The essential idea was to stratify the store into a stack of nested regions and, using a static analysis, automatically infer in which region the result of a given expression should be held. Tofte and Talpin implemented such an approach in ML Kit and subsequently proved correctness of its core [127]. However, the setting here is somewhat different than that found in Rust. For example, regions were inferred entirely by the system, rather than being denoted explicitly by the programmer (as in Rust) [125]. Likewise, no distinction is made between stack- and heap-allocated data (something which is fundamental to Rust) and, instead, we have something more akin to a stack of heaps [28].

Several works have extended the system of Tofte and Talpin. For example, Aiken *et al.* relax the requirement that regions must follow lexical scope and enable regions to be freed in some situations earlier than was possible before (roughly similar to *non-lexical lifetimes* in Rust) [1]. Henglein *et al.* developed a simple imperative language of region operations sufficiently expressive to encode that of Aiken *et al.* [60]. Most notably, this supports first-class region variables which offer considerable expressive power in terms of when a region is deallocated. To make this work, reference counting is employed to decide when a given region can be safely deallocated and a flow-sensitive type system is used to track region variables. Likewise, Walker and Watkins introduce linear types in the system of Tofte and Talpin, arguing that they provide a complementary approach [131]. Of relevance here is that, by introducing regions as first-class (linear) entities, regions can be passed as arguments to provide a notion of region polymorphism similar to that in Rust. Calcagno simplified the proof of Tofte & Talpin by stratifying into high- and low-level views [28]. Hallenberg *et al.* integrated Cheney's *stop and copy* garbage collector with the region-based system of ML Kit [55]. They provided empirical evidence that programs optimised for regions perform better without garbage collection but, otherwise, garbage collection reduces memory footprint. Elsman later strengthened this system to eliminate unused dangling references which are hazardous during garbage collection [45]. Similarly, Qian and Hendren developed an adaptive region-based allocator for Java with an aim to avoiding static escape analysis [108]. Instead, they use on-the-fly detection of objects which do not escape their enclosing method and manage them in local regions.

Cyclone provides another interesting early work on the use of regions in an imperative setting which shares similarities with Rust [49, 50, 53, 54, 61, 67, 123]. Cyclone was designed to provide a type-safe alternative to C which, in particular, "*let programmers control data representation and memory management without sacrificing type-safety*" [54, 67]. A key objective was to prevent dereferences of dangling pointers (amongst other things) through region-based memory management. Like Rust, references to stack-allocated data are permitted in Cyclone, whilst regions ensure references do not outlive the data to which they refer. Memory allocation in Cyclone is done on a *per region* basis. That is, one creates a region and dynamically allocates into it. Invoked methods may also allocate into regions passed by parameter. A global "heap" region is provided, but memory allocated into this is never freed (though the authors argue that garbage collection could be used here). In contrast, Rust does not support dynamic allocation *into* a lifetime. Instead, there is a single heap for dynamically allocated memory and ownership (along with borrowing) is used to ensure eventual deallocation. As such, Rust requires a flow-sensitive approach to borrow checking, whilst Cyclone initially adopted a more standard (flow insensitive) approach.

The experiences gained with Cyclone also enable useful reflections on the design of Rust. Cyclone originally followed Tofte&Talpin in providing LIFO-style regions (i.e. which are allocated/deallocated strictly as a stack). Unfortunately, this proved overly restrictive as the developers observed "*LIFO arenas suffer from several well-known limitations that we encountered repeatedly. In particular, they are not suited to computations such as server and event loops.*" [61]. The fundamental problem is that, under Tofte&Talpin, one cannot deallocate data early (i.e. before its containing region is deallocated) *even when that data is no longer used.* This is excaberated with programs (e.g. servers, interpreters, etc) containing infinite loops with loop carried data. Since it is loop carried, such data must be allocated outside the loop — *meaning it is never deallocated.* To counter this, the team subsequently extended Cyclone with linear references which are, in many ways, similar to those found in Rust [49, 123]. This extension additionally required a flow-sensitive analysis to enforce movement semantics for linear references. An interesting feature was first-class support for an explicit "swap" operation to move linear references around in situations the flow analysis cannot reason about. Overall, Hicks *et al.* "*found we can use unique pointers to provide a more flexible form of arenas that avoids the LIFO lifetime restriction*". We note that boxes enable a similar level of flexibility in Rust and represent a departure from Tofte&Talpin style regions. In particular, a boxed value can be passed arbitrarily up and down the stack and be deallocated at points determined dynamically. For example, they can be used to efficiently carry data between iterations of an infinite loop before being safely deallocated on any subsequent iteration the programmer chooses.

Another related work is that of Deterministic Parallel Java (DPJ) [19, 68, 129]. DPJ is focused on simplifying parallel programming by providing guaranteed deterministic semantics for imperative / object-oriented languages. Of relevance here is the use of a type and effect system which allows the programmer to divide the heap up into regions and detail exactly which regions are read/written by a given method. Regions can be declared as part of a class declaration and then further subdivided within. Regions can be referred to directly by name, or through region paths relative to the given class. The ability to employ wildcards in region paths is particularly notable, as it enables fine-grained control over methods which recurse linked data structures.

The Real-Time Java Specification (RTSJ) aims to allow hard and soft real-time processes to run alongside each other in the same JVM [20]. To alleviate the issues of unpredictable performance resulting from the garbage collector, the RTSJ introduces a notion of *scoped memory*. These allow programmers to explicitly create and destroy memory regions and, essentially, bypass the garbage collector. Unfortunately the abstractions are very low level and, worse still, mistakes in usage result only in runtime errors [57]. Indeed, Beebee and Rinard found it "close to impossible" to develop error-free programs under the specification [14]. As a result a number of works have explored

various approaches to statically checking for dangling references, etc [5, 107, 135]. For example, the work of Potanin *et al.* employed a simple and effective scheme which equates Java packages with memory scopes [107]. On the other hand, the work of Chin *et al.* attempts to automatically infer the necessary region annotations [32].

8 CONCLUSION

Rust is a new systems language that takes an interesting approach to memory management. Most existing languages either rely on garbage collection (as in Java/C#/Haskell) or require that dynamically allocated memory be manually deallocated (as in C/C++). Rust is perhaps unusual in not following either of these paths. Instead, through judicious use of reference lifetimes and borrowing, Rust is able to automatically reclaim dynamically allocated memory.[20] Whilst the ideas underpinning this (namely, linearity/uniqueness and regions) are not new and have been extensively studied in the literature, Rust brings them together in a coherent fashion. As such, we find that Rust makes an interesting real-world example to study.

In this paper, we have presented a lightweight calculus which captures the salient aspects of reference lifetimes and borrowing in a succinct form. Our calculus is (effectively) a subset of Rust and supports copy- and move-semantics, mutable borrowing, reborrowing, partial moves, and lifetimes. At the same time the calculus is otherwise minimal, making it relatively easy to understand and digest. The calculus employs a flow-sensitive type system to encode the rules of type and borrow checking. For this, we have established a key result, namely that type and borrow safe programs do not get stuck and preserve the borrowing invariant. We have employed lightweight mechanisation to support our proof and, in particular, have model checked over 500B input programs. Whilst this does not constitute a full mechanical proof, we note the high likelihood of errors being revealed by small inputs [111] following the small scope hypothesis [65]. Furthermore, we have fuzz tested the Rust compiler, `rustc`, using over 2B input programs and compared results. This uncovered one previously known issue in `rustc` and several other possible issues. Furthermore, it identified areas where FR can be extended to give a more accurate model. We have explored several extensions to our calculus in §6, including for control flow and tuples.

Finally, several interesting areas for future work present themselves. For example, investigating whether Non-Lexical Lifetimes can be supported in FR (e.g. via the approach used in Oxide). Likewise, adding interesting language features not currently explored (e.g. structs with lifetime polymorphism, traits, etc).

Acknowledgements. The author would like to thank Nicholas D. Matsakis and Nicholas Cameron from Mozilla for helpful comments on earlier drafts. Their considerable experience with the Rust language has helped ensure this paper is as accurate as possible given the changing nature of Rust. The author would also like to thank Nathan Chong and the various anonymous reviewers of earlier drafts of this paper. They have certainly helped to improve this paper significantly, and the author is indebted to their care and consideration.

REFERENCES

[1] A. Aiken, M. Fähndrich, and R. Levien. 1995. Better Static Memory Management: Improving Region-Based Analysis of Higher-Order Languages. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. 174–185.

[20] Perhaps surprisingly, Rust does not provide a "no leak" guarantee as, for example, reference counting cycles can still be constructed manually.

[2] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. 2002. Alias Annotations for Program Understanding. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, 311–330.
[3] Paulo Sérgio Almeida. 1997. Balloon Types: Controlling Sharing of State in Data Types. In *Proceedings of the European Confereince on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 32–59.
[4] Brian Anderson, Lars Bergstrom, Manish Goregaokar, Josh Matthews, Keegan McAllister, Jack Moffitt, and Simon Sapin. 2016. Engineering the Servo Web Browser Engine Using Rust. In *Proceedings of the International Conference of Software Engineering (ICSE)*. ACM Press, 81–89.
[5] Chris Andreae, James Noble, Yvonne Coady, Celina Gibbs, Jan Vitek, and Tian Zhao. 2006. STARS: Scoped Types and Aspects for Real-Time Systems. In *Proceedings of the European Confereince on Object-Oriented Programming (ECOOP)*. 1–44.
[6] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM Press, Article 147.
[7] Thibaut Balabonski, François Pottier, and Jonathan Protzenko. 2016. The Design and Formalization of Mezzo, a Permission-Based Programming Language. *ACM Transactions on Programming Languages and Systems* 38, 4 (2016), 14:1–14:94.
[8] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamari, and Leonid Ryzhyk. 2017. System Programming in Rust: Beyond Safety. *Operating Systems Review* 51, 1 (2017), 94–99.
[9] A. Banerjee, N. Heintze, and J. G. Riecke. 1999. Region Analysis and the Polymorphic Lambda Calculus. In *Proceedings of the ACM/IEEE Symposium on Logic In Computer Science (LICS)*. IEEE Computer Society Press, 88–97.
[10] Marek Baranowski, Shaobo He, and Zvonimir Rakamarić. 2018. Verifying Rust Programs with SMACK. In *Automated Technology for Verification and Analysis*. Springer-Verlag, 528–535.
[11] Erik Barendsen and Sjaak Smetsers. 1996. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science* 6, 6 (1996), 579–612.
[12] M. Barnett, B. Evan Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. 2006. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Proceedings of the Formal Methods for Components and Objects (FMCO)*. 364–387.
[13] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. 2004. Verification of Object-Oriented Programs with Invariants. *Journal of Object Technology* 3, 6 (2004), 27–56.
[14] William S. Beebee, Jr. and Martin Rinard. 2001. An Implementation of Scoped Memory for Real-Time Java. In *1st International Workshop on Embedded Software (EMSOFT)*. Springer-Verlag, 289–305.
[15] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proceedings of the ACM on Programing Languages* 2, POPL (2018), 5:1–5:29.
[16] G. M. Bierman and M. J. Parkinson. 2003. Effects and effect inference for a core Java calculus. *Electronic Notes in Computer Science* 82, 8 (2003), 1–26.
[17] Jim Blandy and Jason Ordendorff. 2018. *Programming Rust.* O'Reilly.
[18] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. 2011. Why3: Shepherd Your Herd of Provers. In *Proceedings of the Workshop on Intermediate Verification Languages (BOOGIE)*.
[19] Robert Bocchino, Vikram Adve, Sarita Adve, and Marc Snir. 2009. Parallel Programming Must Be Deterministic by Default. In *Proceedings of the Workshop on Hot Topics in Parallelism (HotPar)*.
[20] Gregory Bollella, Ben Brosgol, Peter Dibble, Steve Furr, James Gosling, David Hardin, and Mark Turnbull. 2000. *The Real-Time Specification for Java.* Addison-Wesley. xxiii + 195 pages.
[21] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. 2002. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM Press, 211–230.
[22] Chandrasekhar Boyapati and Martin Rinard. 2001. A Parameterized Type System for Race-Free Java Programs. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 56–69.
[23] John Boyland. 2001. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience* 31, 6 (May 2001), 533–553.
[24] John Boyland. 2003. Checking Interference with Fractional Permissions. In *Proceedings of the Static Analysis Symposium (SAS) (LNCS)*, Vol. 2694. Springer-Verlag, 55–72.
[25] John Boyland. 2003. Connecting Effects and Uniqueness with Adoption. In *Proceedings of the Workshop Aliasing, Capabilities and Ownership (IWACO) (UU-CS)*. Utrecht University, 42 – 57.
[26] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. 2012. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the International Symposium on Software*

Testing and Analysis (ISSTA). ACM Press, 133–143.

[27] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the Conference on Operating Systems Design and Implementation (OSDI)*. 209–224.

[28] Cristiano Calcagno. 2001. Stratified Operational Semantics for Safety and Correctness of Region Calculus. In *Proceedings of the ACM symposium on the Principles Of Programming Languages (POPL)*. ACM Press, 155–165.

[29] Nicholas Robert Cameron, James Noble, and Tobias Wrigstad. 2010. Tribal ownership. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 618–633.

[30] Kartik Chandra and Rastislav Bodík. 2018. Bonsai: synthesis-based reasoning for type systems. *Proceedings of the ACM on Programing Languages* 2, POPL (2018), 62:1–62:34.

[31] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Praun, and V. Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 519–538.

[32] Wei-Ngan Chin, Florin Craciun, Shengchao Qin, and Martin C. Rinard. 2004. Region Inference for an Object-Oriented Language. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 243–254.

[33] Morten V. Christiansen, Fritz Henglein, Henning Niss, and Per Velschow. 1998. *Safe Region-Based Memory Management for Objects*. Technical Report. DIKU, University of Copenhagen.

[34] Koen Claessen, Jonas Duregård, and Michal H. Palka. 2015. Generating constrained random data with uniform distribution. *Journal of Functional Programming* 25 (2015), e8.

[35] Dave Clarke and Sophia Drossopoulou. 2002. Ownership, Encapsulation, and the Disjointness of Type and Effect. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 292–310.

[36] Dave Clarke, James Noble, and Tobias Wrigstad. 2012. *Aliasing in Object-oriented Programming*. LNCS, Vol. 7850. Springer.

[37] David Clarke, John Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 48–64.

[38] David Clarke and Tobias Wrigstad. 2003. External Uniqueness is Unique Enough. In *Proceedings of the European Confereince on Object-Oriented Programming (ECOOP)*. 176–200.

[39] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt meets relaxed memory. In *Proceedings of the ACM symposium on the Principles Of Programming Languages (POPL)*. Article 34.

[40] L. de Moura and N. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 337–340.

[41] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust Typechecker Using CLP (T). In *Proceedings of the Conference on Automated Software Engineering (ASE)*. IEEE Computer Society Press, 482–493.

[42] Ivaylo Donchev and Emilia Todorova. 2015. Implementation of Binary Search Trees Via Smart Pointers. *International Journal of Advanced Computer Science and Applications (IJACSA)* 6, 3 (2015).

[43] Jonas Duregård, Patrik Jansson, and Meng Wang. 2012. Feat: functional enumeration of algebraic types. In *Proceedings of the ACM Symposium on Haskell*. ACM Press, 61–72.

[44] C. Dymnikov, D. J. Pearce, and A. Potanin. 2013. OwnKit: Inferring Modularly Checkable Ownership Annotations for Java. In *Proceedings of the Australasian Software Engineering Conference (ASWEC)*. 181–190.

[45] Martin Elsman. 2003. Garbage Collection Safety for Region-based Memory Management. In *Proceedings of the Workshop on Types in Languages Design and Implementation (TLDI)*. ACM Press, 123–134.

[46] Manuel Fähndrich and Rob DeLine. 2002. Adoption and Focus: Practical Linear Types for Imperative Programming. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. 13–24.

[47] Josselin Feist, Laurent Mounier, and Marie-Laure Potet. 2014. Statically detecting use after free on binary code. *J. Computer Virology and Hacking Techniques* 10, 3 (2014), 211–217.

[48] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. 2016. *A Pragmatic Type System for Deductive Verification*. Technical Report. Laboratoire de Recherche en Informatique, Inria Sacla.

[49] Matthew Fluet, Greg Morrisett, and Amal J. Ahmed. 2006. Linear Regions Are All You Need. In *Proceedings of the European Symposium on Programming (ESOP) (LNCS)*, Vol. 3924. Springer-Verlag, 7–21.

[50] Prodromos Gerakios, Nikolaos Papaspyrou, and Konstantinos Sagonas. 2010. Race-free and memory-safe multithreading: design and implementation in cyclone. In *Proceedings of the Workshop on Types in Languages Design and Implementation (TLDI)*. ACM Press, 15–26.

[51] Jean-Yves Girard. 1987. Linear Logic. *Theoretical Computer Science* 50 (1987), 1–102.

[52] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and reference immutability for safe parallelism. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems,*

Languages and Applications (OOPSLA). ACM Press, 21–40.

[53] Dan Grossman. 2003. Type-safe multithreading in cyclone. In *Proceedings of the Workshop on Types in Languages Design and Implementation (TLDI)*. ACM Press, 13–25.

[54] Dan Grossman, J. Gregory Morrisett, Trevor Jim, Michael W. Hicks, Yanling Wang, and James Cheney. 2002. Region-Based Memory Management in Cyclone. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. ACM, 282–293.

[55] Niels Hallenberg, Martin Elsman, and Mads Tofte. 2002. Combining Region Inference and Garbage Collection. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 141–152.

[56] Philipp Haller and Martin Odersky. 2010. Capabilities for uniqueness and borrowing. In *Proceedings of the European Confereince on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 354–378.

[57] Hamza Hamza and Steve Counsell. 2012. Region-Based RTSJ Memory Management: State of the art. *Science of Computer Programming* 77, 5 (2012), 644–659.

[58] Dana Harrington. 2006. Uniqueness logic. *Theoretical Computer Science* 354, 1 (2006), 24–41.

[59] Simon Helsen and Peter Thiemann. 2000. Syntactic Type Soundness for the Region Calculus. In *Workshop on Higher Order Operational Techniques in Semantics (HOOTS) (Electronic Notes in Computer Science)*, Vol. 41(3). Elsevier, 1–20.

[60] Fritz Henglein, Henning Makholm, and Henning Niss. 2001. A Direct Approach to Control-Flow Sensitive Region-Based Memory Management. In *Proceedings of the Symposium on Principles and Practice of Declarative Programming (PPDP)*. 175–186.

[61] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. 2004. Experience with Safe Manual Memory-Management in Cyclone. In *Proceedings of the International Symposium on Memory Management (ISMM)*. ACM Press, 73–84.

[62] John Hogg. 1991. Islands: Aliasing Protection in Object-Oriented Languages. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 271–285.

[63] A. Igarashi, B. Pierce, and P. Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems* 23, 3 (May 2001), 396–459.

[64] Nicholas Jacek, Meng-Chieh Chiu, Benjamin M. Marlin, and Eliot Moss. 2016. Assessing the limits of program-specific garbage collection performance. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 584–598.

[65] Daniel Jackson and Craig Damon. 1996. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. *IEEE Transactions on Software Engineering* 22, 7 (1996), 484–495.

[66] Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. 2015. Session types for Rust. In *Proceedings of the Workshop on Generic Programming (WGP)*. 13–22.

[67] 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 USENIX technical Conference*. 275–288.

[68] Robert L. Bocchino Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A type and effect system for Deterministic Parallel Java. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 97–116.

[69] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2020. Stacked Borrows: An Aliasing Model for Rust. In *Proceedings of the ACM symposium on the Principles Of Programming Languages (POPL)*. Article 41.

[70] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the Foundations of the Rust Programming Language. In *Proceedings of the ACM symposium on the Principles Of Programming Languages (POPL)*. ACM Press, 66:1–66:34.

[71] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *Proceedings of the ACM International Conference on Functional Programming (ICFP)*. ACM Press, 256–269.

[72] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20.

[73] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *Proceedings of the European Confereince on Object-Oriented Programming (ECOOP)*, Vol. 74. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 17:1–17:29.

[74] Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. 2012. Run your research: on the effectiveness of lightweight mechanization. In *Proceedings of the ACM symposium on the Principles Of Programming Languages (POPL)*. ACM Press, 285–296.

[75] D. E. Knuth. 1981. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Second Edition, Addison-Wesley, Reading.

[76] Naoki Kobayashi. 1999. Quasi-linear types. In *Proceedings of the ACM symposium on the Principles Of Programming Languages (POPL)*. ACM Press, 29–42.
[77] Daniel Kroening and Michael Tautschnig. 2014. CBMC – C Bounded Model Checker. In *Proceedings of the conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer-Verlag, 389–391.
[78] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the Symposium on Operating System Principles (SOSP)*. ACM Press, 234–251.
[79] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. The Case for Writing a Kernel in Rust. In *Proceedings of the Asia-Pacific Workshop on Systems (APSYS)*. ACM, 1:1–1:7.
[80] Amit A. Levy, Michael P. Andersen, Bradford Campbell, David E. Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. 2015. Ownership is theft: experiences building an embedded OS in Rust. In *Proceedings of the Workshop on Programming Languages and Operating Systems*. 21–26.
[81] Ondrej Lhoták and Kwok-Chiang Andrew Chung. 2011. Points-to analysis with efficient strong updates. In *Proceedings of the ACM symposium on the Principles Of Programming Languages (POPL)*. ACM Press, 3–16.
[82] Paley Li, Nicholas Cameron, and James Noble. 2012. Sheep Cloning with Ownership Types. In *Proceedings of the Workshop on Foundations of Object-Oriented Languages (FOOL)*.
[83] M. Lindner, J. Aparicius, and P. Lindgren. 2018. No Panic! Verification of Rust Programs by Symbolic Execution. In *International Conference on Industrial Informatics (INDIN)*. 108–114.
[84] M. Lindner, N. Fitinghoff, J. Eriksson, and P. Lindgren. 2019. Verification of Safety Functions Implemented in Rust - a Symbolic Execution based approach. In *International Conference on Industrial Informatics (INDIN)*, Vol. 1. 432–439.
[85] Yi Lu and John Potter. 2006. On Ownership and Accessibility. In *Proceedings of the European Confereince on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 99–123.
[86] Nicholas D. Matsakis. 2012. Parallel Closures: A New Twist on an Old Idea. In *Proceedings of the Workshop on Hot Topics in Parallelism (HotPar)*.
[87] Nicholas D. Matsakis and Thomas R. Gross. 2009. Programming with Intervals. In *LCPC*. Springer-Verlag, 203–217.
[88] Nicholas D. Matsakis and Thomas R. Gross. 2010. Reflective Parallel Programming: Extensible and High-Level Control of Runtime, Compiler, and Application Interaction. In *Proceedings of the Workshop on Hot Topics in Parallelism (HotPar)*.
[89] Nicholas D. Matsakis and Thomas R. Gross. 2010. A time-aware type system for data-race protection and guaranteed initialization. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM Press, 634–651.
[90] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2020. RustHorn: CHC-Based Verification for Rust Programs. In *Programming Languages and Systems*. Springer-Verlag, 484–514.
[91] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. 2019. Exploring C semantics and pointer provenance. In *Proceedings of the ACM symposium on the Principles Of Programming Languages (POPL)*. 67:1–67:32.
[92] Scott Meyers. 1994. *Effective C++*. Addison-Wesley.
[93] R. Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* 17 (1978), 348–375.
[94] P. Müller. 2002. *Modular Specification and Verification of Object-Oriented Programs*. LNCS, Vol. 2262.
[95] P. Müller, M. Schwerhoff, and A. J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 41–62.
[96] Vikram Narayanan, Marek S. Baranowski, Leonid Ryzhyk, Zvonimir Rakamarić, and Anton Burtsev. 2019. RedLeaf: Towards An Operating System for Safe and Verified Firmware. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*. ACM Press, 37–44.
[97] Flemming Nielson, Hanne R. Nielson, and Chris L. Hankin. 1999. *Principles of Program Analysis*. Springer-Verlag.
[98] James Noble, David G. Clarke, and John Potter. 1999. Object Ownership for Dynamic Alias Protection. In *Proceedings of the IEEE Conference on Technology of Object-Oriented Languages and Systems*. IEEE Computer Society Press, 176–187.
[99] James Noble, Jan Vitek, and John Potter. 1998. Flexible Alias Protection. In *Proceedings of the European Confereince on Object-Oriented Programming (ECOOP) (LNCS)*, Vol. 1445. Springer-Verlag, 158–185.
[100] Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby C. Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. 2016. Refinement through restraint: bringing down the cost of verification. In *Proceedings of the ACM International Conference on Functional Programming (ICFP)*. ACM Press, 89–102.
[101] Johan Östlund, Tobias Wrigstad, Dave Clarke, and Beatrice Åkerblom. 2008. Ownership, Uniqueness and Immutability. In *TOOLS Europe 2008*.
[102] D. J. Pearce. 2005. *Some directed graph algorithms and their application to pointer analysis*. Ph.D. Dissertation. Imperial College, London.

[103] Quan Phan and Gerda Janssens. 2007. Static Region Analysis for Mercury. In *Proceedings of the ACM International Conference on Logic Programming (ICLP) (LNCS)*, Vol. 4670. Springer-Verlag, 317–332.
[104] B. C. Pierce. 2002. *Types and Programming Languages.* MIT Press.
[105] Robert Pollack. 1998. How to believe a machine-checked proof. In *Twenty Five Years of Constructive Type Theory*, G. Sambin and J. Smith (Eds.). Oxford University Press.
[106] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. 2006. Generic Ownership. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA).* ACM Press.
[107] Alex Potanin, James Noble, Tian Zhao, and Jan Vitek. 2005. A High Integrity Profile for Memory Safe Programming in Real-time Java. In *Proceedings of the Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES).*
[108] Feng Qian and Laurie Hendren. 2002. An Adaptive, Region-based Allocator for Java. In *Proceedings of the International Symposium on Memory Management (ISMM).* ACM Press, 127–138.
[109] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI).* ACM Press, 763–779.
[110] Erik Reed. 2015. *Patina: A Formalization of the Rust Programming Language.* Technical Report.
[111] Michael Roberson, Melanie Harries, Paul T. Darga, and Chandrasekhar Boyapati. 2008. Efficient software model checking of soundness of type systems. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA).* ACM Press, 493–504.
[112] Grigore Rosu and Traian-Florin Serbanuta. 2010. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434.
[113] Rust Team. [n.d.]. Rust Homepage. www.rust-lang.org. Retrieved 2016-05-01.
[114] Rust Team. [n.d.]. The Rust Programming Language. `doc.rust-lang.org/book/`. Retrieved 2016-05-01.
[115] Rust Team. [n.d.]. The Rustonomicon - The Dark Arts of Advanced and Unsafe Rust Programming. `doc.rust-lang.org/nomicon/`. Retrieved 2020-31-03.
[116] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the USENIX technical Conference.* 309–318.
[117] Marco Servetto, D. J. Pearce, and Lindsay Groves. 2013. Balloon Types for Safe Parallelisation over Arbitrary Object Graphs. In *Proceedings of the Workshop on Determinism and Correctness in Parallel Programming (WODET).*
[118] Sriram Srinivasan and Alan Mycroft. 2008. Kilim: Isolation-Typed Actors for Java. In *Proceedings of the European Confereince on Object-Oriented Programming (ECOOP).* 104–128.
[119] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the Conference on Code Generation and Optimisation (CGO).* IEEE Computer Society Press, 46–55.
[120] R. Strom and S. Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Transactions on Software Engineering* 12, 1 (1986), 157–171.
[121] Robert E. Strom and Daniel M. Yellin. 1993. Extending Typestate Checking Using Conditional Liveness Analysis. *IEEE Transactions on Software Engineering* 19, 5 (1993), 478–485.
[122] David Svoboda and Lutz Wrage. 2014. Pointer Ownership Model. In *Proceedings of the Hawaii International Conference on System Sciences (HICSS).* IEEE Computer Society Press, 5090–5099.
[123] Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. 2006. Safe Manual Memory Management in Cyclone. *Science of Computer Programming* 62, 2 (2006), 122–144.
[124] J.-P Talpin and P. Jouvelot. 1992. Polymorphic type, region, and effect inference. *Journal of Functional Programming* 2, 3 (1992), 245–271.
[125] Mads Tofte and Lars Birkedal. 1998. A Region Inference Algorithm. *ACM Transactions on Programming Languages and Systems* 20, 4 (1998), 734–767.
[126] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. 2004. A Retrospective on Region-Based Memory Management. *Higher-Order and Symbolic Computation* 17, 3 (2004), 245–265.
[127] Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Information and Computation* 132, 2 (1997), 109–176.
[128] J. Toman, S. Pernsteiner, and E. Torlak. 2015. Crust: A Bounded Verifier for Rust. In *Proceedings of the Conference on Automated Software Engineering (ASE).* 75–80.
[129] Mohsen Vakilian, Danny Dig, Robert Bocchino, Jeffrey Overbey, Vikram Adve, and Ralph Johnson. 2009. Inferring Method Effect Summaries for Nested Heap Regions. In *Proceedings of the Conference on Automated Software Engineering (ASE).* 421–432.
[130] P. Wadler. 1990. Linear types can change the world!. In *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel.* 347–359.

[131] David Walker and Kevin Watkins. 2001. On Regions and Linear Types. In *Proceedings of the ACM International Conference on Functional Programming (ICFP)*. 181–192.

[132] F. Wang, F. Song, M. Zhang, X. Zhu, and J. Zhang. 2018. KRust: A Formal Executable Semantics of Rust. In *Proceedings of the Symposium on Theoretical Aspects of Software Engineering (TASE)*. 44–51.

[133] Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. 2019. Oxide: The Essence of Rust. arXiv:cs.PL/1903.00982

[134] A. K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (1994), 38–94.

[135] Tian Zhao, James Noble, and Jan Vitek. 2004. Scoped Types for Real-Time Java. In *Proceedings of the Real-Time Systems Symposium (RTSS)*. IEEE Computer Society Press, 241–251.

[136] Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. 2010. Ownership and Immutability in Generic Java. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 598–617.

9 APPENDIX

In the following, the notation $\mathtt{x} \triangleq [\mathtt{X}]$ denotes that a given variable $\mathtt{x}$ is known to have the form $\mathtt{X}$. For example, $\mathtt{t} \triangleq [\mathtt{v}]$ indicates that term $\mathtt{t}$ is a value. Also, $\langle\cdot\rangle^1$ matches a slot where we *don't care* about its contents. Note, lemma and theorem numbering is adjusted in some places to align with §4.

9.1 Preliminaries

A number of supplementary lemmas are needed for the main lemmas which follow, and we now establish these first. Observe we often refer simply to a "well-formed typing environment" where it is unnecessary to consider the lifetime which it is well-formed with respect to.

Lemma 9.1 (Safe Strengthening). *Let $\mathcal{S}$ be a program store; let Γ be a well-formed typing environment where $\mathcal{S} \sim \Gamma$; let $\mathtt{T}_1, \mathtt{T}_2$ be types where $\mathtt{T}_1 \sqsubseteq \mathtt{T}_2$; and, let $\mathtt{v}$ be a value. If $\mathcal{S} \vdash \mathtt{v} \sim \mathtt{T}_1$ then $\mathcal{S} \vdash \mathtt{v} \sim \mathtt{T}_2$.*

Proof. Straightforward by induction on the structure of $\mathtt{T}_1 \sqsubseteq \mathtt{T}_2$ according to Definition 3.7, whilst ignoring cases which don't apply (i.e. for undefined types). □

Lemma 9.2 (Transitive Strengthening). *Let $\widetilde{\mathtt{T}}_1$, $\widetilde{\mathtt{T}}_2$ and $\widetilde{\mathtt{T}}_3$ be partial types. If $\widetilde{\mathtt{T}}_1 \sqsubseteq \widetilde{\mathtt{T}}_2$ and $\widetilde{\mathtt{T}}_2 \sqsubseteq \widetilde{\mathtt{T}}_3$ then $\widetilde{\mathtt{T}}_1 \sqsubseteq \widetilde{\mathtt{T}}_3$.*

Proof. By structural induction on the structure of $\widetilde{\mathtt{T}}_2$ according to Figure 1:

- **Base Case** $\widetilde{\mathtt{T}}_2 \triangleq [\epsilon]$. Straightforward as $\widetilde{\mathtt{T}}_1 = \widetilde{\mathtt{T}}_2$ by W-Reflex.
- **Base Case** $\widetilde{\mathtt{T}}_2 \triangleq [\mathtt{int}]$. Straightforward as $\widetilde{\mathtt{T}}_1 = \widetilde{\mathtt{T}}_2$ by W-Reflex.
- **Base Case** $\widetilde{\mathtt{T}}_2 \triangleq [\&[\mathtt{mut}]\ \overline{\mathtt{v}}]$. By W-Bor, $\widetilde{\mathtt{T}}_1 \triangleq [\&[\mathtt{mut}]\ \overline{\mathtt{u}}]$ where $\overline{\mathtt{u}} \subseteq \overline{\mathtt{v}}$. If $\widetilde{\mathtt{T}}_3 \triangleq [\&[\mathtt{mut}]\ \overline{\mathtt{w}}]$ then $\overline{\mathtt{v}} \subseteq \overline{\mathtt{w}}$ by W-Bor and $\widetilde{\mathtt{T}}_1 \sqsubseteq \widetilde{\mathtt{T}}_3$ follows. Otherwise, $\widetilde{\mathtt{T}}_3 \triangleq [\lfloor \mathtt{T}_4 \rfloor]$ where $\widetilde{\mathtt{T}}_2 \sqsubseteq \mathtt{T}_4$ by W-UndefB. Then, $\mathtt{T}_4 \triangleq [\&[\mathtt{mut}]\ \overline{\mathtt{w}}]$ where $\overline{\mathtt{v}} \subseteq \overline{\mathtt{w}}$ by W-Bor and $\widetilde{\mathtt{T}}_1 \sqsubseteq \widetilde{\mathtt{T}}_3$ follows as before.
- **Inductive Case** $\widetilde{\mathtt{T}}_2 \triangleq [\Box\widetilde{\mathtt{T}}_\mathtt{b}]$. By W-Box, $\widetilde{\mathtt{T}}_1 \triangleq [\Box\widetilde{\mathtt{T}}_\mathtt{a}]$ where $\widetilde{\mathtt{T}}_\mathtt{a} \sqsubseteq \widetilde{\mathtt{T}}_\mathtt{b}$. If $\widetilde{\mathtt{T}}_3 \triangleq [\Box\widetilde{\mathtt{T}}_\mathtt{c}]$ then $\widetilde{\mathtt{T}}_\mathtt{b} \sqsubseteq \widetilde{\mathtt{T}}_\mathtt{c}$ by W-Box and $\widetilde{\mathtt{T}}_1 \sqsubseteq \widetilde{\mathtt{T}}_3$ follows (since $\widetilde{\mathtt{T}}_\mathtt{a} \sqsubseteq \widetilde{\mathtt{T}}_\mathtt{c}$ by inductive hypothesis). Otherwise, $\widetilde{\mathtt{T}}_3 \triangleq [\lfloor\Box\mathtt{T}_\mathtt{c}\rfloor]$ where $\widetilde{\mathtt{T}}_\mathtt{b} \sqsubseteq \lfloor\mathtt{T}_\mathtt{c}\rfloor$ by W-UndefC and $\widetilde{\mathtt{T}}_1 \sqsubseteq \widetilde{\mathtt{T}}_3$ follows (since $\widetilde{\mathtt{T}}_\mathtt{a} \sqsubseteq \lfloor\mathtt{T}_\mathtt{c}\rfloor$ by inductive hypothesis).
- **Inductive Case** $\widetilde{\mathtt{T}}_2 \triangleq [\lfloor\mathtt{T}_\mathtt{b}\rfloor]$. By W-UndefA, $\widetilde{\mathtt{T}}_3 \triangleq [\lfloor\mathtt{T}_\mathtt{c}\rfloor]$ where $\mathtt{T}_\mathtt{b} \sqsubseteq \mathtt{T}_\mathtt{c}$. If $\widetilde{\mathtt{T}}_1 \triangleq [\lfloor\mathtt{T}_\mathtt{a}\rfloor]$ then $\mathtt{T}_\mathtt{a} \sqsubseteq \mathtt{T}_\mathtt{b}$ by W-UndefA and $\widetilde{\mathtt{T}}_1 \sqsubseteq \widetilde{\mathtt{T}}_3$ follows (since $\mathtt{T}_\mathtt{a} \sqsubseteq \mathtt{T}_\mathtt{c}$ by inductive hypothesis). Otherwise, $\widetilde{\mathtt{T}}_1 \triangleq [\mathtt{T}_\mathtt{a}]$ where $\mathtt{T}_\mathtt{a} \sqsubseteq \mathtt{T}_\mathtt{b}$ by W-UndefB and $\widetilde{\mathtt{T}}_1 \sqsubseteq \widetilde{\mathtt{T}}_3$ follows as before. □

Lemma 9.3 (Location). *Let $\mathcal{S}$ be a program store; let Γ be a well-formed typing environment where $\mathcal{S} \sim \Gamma$; let $\widetilde{\mathtt{T}}$ be a partial type; let $\mathtt{m}$ be a lifetime; and, let $\mathtt{w}$ be an lval. If $\Gamma \vdash \mathtt{w} : \langle\widetilde{\mathtt{T}}\rangle^\mathtt{m}$, then $\mathtt{loc}(\mathcal{S}, \mathtt{w}) = \ell_\mathtt{w}$ for some location $\ell_\mathtt{w}$ where $\mathcal{S}(\ell_\mathtt{w}) = \langle\mathtt{v}^\perp\rangle^\mathtt{m}$ and $\mathcal{S} \vdash \mathtt{v}^\perp \sim \widetilde{\mathtt{T}}$.*

Proof. By structural induction on the structure of $\mathtt{w}$ according to Figure 1:

- **Base Case** $\mathtt{w} \triangleq [\mathtt{x}]$. Straightforward as $\mathtt{loc}(\mathcal{S}, \mathtt{x}) = \ell_\mathtt{x}$ by Def 3.1 and $\Gamma(\mathtt{x}) = \langle\widetilde{\mathtt{T}}\rangle^\mathtt{m}$ by Def 3.11. Then, $\mathcal{S} \vdash \mathtt{v}^\perp \sim \widetilde{\mathtt{T}}$ follows from $\mathcal{S} \sim \Gamma$.
- **Inductive Case** $\mathtt{w} \triangleq [*\mathtt{u}]$ and $\Gamma \vdash \mathtt{u} : \langle\Box\widetilde{\mathtt{T}}\rangle^\mathtt{m}$. By inductive hypothesis, $\mathtt{loc}(\Gamma, \mathtt{u}) = \ell_\mathtt{u}$ where $\mathcal{S}(\ell_\mathtt{u}) = \langle\ell_\mathtt{w}^\bullet\rangle^\mathtt{m}$ and $\mathcal{S} \vdash \ell_\mathtt{w}^\bullet \sim \Box\widetilde{\mathtt{T}}$. Then, $\mathcal{S} \vdash \mathtt{v}^\perp \sim \widetilde{\mathtt{T}}$ follows by Def 4.4.
- **Inductive Case** $\mathtt{w} \triangleq [*\mathtt{u}]$ and $\Gamma \vdash \mathtt{u} : \langle\&[\mathtt{mut}]\ \overline{\mathtt{q}}\rangle^\mathtt{m}$. By inductive hypothesis, $\mathtt{loc}(\Gamma, \mathtt{u}) = \ell_\mathtt{u}$ where $\mathcal{S}(\ell_\mathtt{u}) = \langle\ell_\mathtt{w}^\circ\rangle^\mathtt{m}$ and $\mathcal{S} \vdash \ell_\mathtt{w}^\circ \sim \&[\mathtt{mut}]\ \overline{\mathtt{q}}$. By T-LvBor, we have $\overline{\Gamma \vdash \mathtt{q} : \mathtt{T}_\mathtt{i}}$ and by V-Borrow, $\exists_\mathtt{i}.\mathtt{loc}(\Gamma, \mathtt{q}_\mathtt{i}) = \ell_\mathtt{w}$ and, hence, $\mathcal{S} \vdash \mathtt{v}^\perp \sim \mathtt{T}_\mathtt{i}$. Then, $\mathcal{S} \vdash \mathtt{v}^\perp \sim \bigsqcup\mathtt{T}_\mathtt{i}$ follows by Lemma 9.1.

□

The location lemma establishes an important connection between a well-typed lval, and its corresponding location in the program store. Specifically, that: (1) a well-typed lval always corresponds to a valid location (i.e. some component of it is not undefined, etc); (2) that the slot type abstracts the actual slot value in the program store.

Corollary 9.4 (Read Preservation). *Let $\mathcal{S}$ be a program store; let Γ be a well-formed typing environment where $\mathcal{S} \sim \Gamma$; let $\mathtt{T}$ be a type; let $\mathtt{m}$ be a lifetime; and, let $\mathtt{w}$ be an lval. If $\Gamma \vdash \mathtt{w} : \langle \mathtt{T} \rangle^{\mathtt{m}}$, then $\mathtt{read}(\mathcal{S}, \mathtt{w}) = \langle \mathtt{v} \rangle^{\mathtt{m}}$ follows for some value $\mathtt{v}$ where $\mathcal{S} \vdash \mathtt{v} \sim \mathtt{T}$.*

Proof. Follows trivially from Lemma 9.3 and Def 3.2. □

We next consider an important lemma which we refer to as *drop preservation*. Specifically, this establishes that dropping a given set of variables (e.g. those declared in a block) preserves a safe abstraction between the runtime environment and typing environment:

Lemma 9.5 (Drop Preservation). *Let $\mathcal{S}$ be a program store; let Γ be a well-formed typing environment with respect to a lifetime $\mathtt{l}$ where $\mathcal{S} \sim \Gamma$. Then, $\mathtt{drop}(\mathcal{S}, \mathtt{l}) \sim \mathtt{drop}(\Gamma, \mathtt{l})$.*

Proof. Straightforward since $\mathcal{S} \sim \Gamma$. □

We now establish a similar result covering the subsequent assignment of variables after a drop (e.g. as happens in an assignment statement).

Lemma 9.6 (Update Preservation). *Let $\mathcal{S}$ be a program store; let Γ be a well-formed typing environment; let $\widetilde{\mathtt{T}}_1, \widetilde{\mathtt{T}}_2$ be partial types; let $\mathtt{m}$ be a lifetime; let $\mathtt{v}_1^{\perp}, \mathtt{v}_2^{\perp}$ be partial values where $\mathcal{S} \vdash \mathtt{v}_1^{\perp} \sim \widetilde{\mathtt{T}}_1$ and $\mathcal{S} \vdash \mathtt{v}_2^{\perp} \sim \widetilde{\mathtt{T}}_2$; let $\mathtt{m}$ be a lifetime; and, let $\mathtt{w}$ be an lval where $\Gamma \vdash \mathtt{w} : \langle \widetilde{\mathtt{T}}_1 \rangle^{\mathtt{m}}$. If $\mathcal{S} \sim \Gamma$, then $\mathtt{write}(\mathtt{drop}(\mathcal{S}, \{\mathtt{v}_2^{\perp}\}), \mathtt{w}, \mathtt{v}_1^{\perp}) \sim \mathtt{write}^{\mathtt{0}}(\Gamma, \mathtt{w}, \widetilde{\mathtt{T}}_2)$.*

Proof. By inspection of Def 3.4, $\mathtt{drop}(\mathcal{S}, \{\mathtt{v}_2^{\perp}\})$ can only remove heap locations. Thus, we have $\mathtt{write}(\mathtt{drop}(\mathcal{S}, \{\mathtt{v}_2^{\perp}\}), \mathtt{w}, \perp) \sim \mathtt{write}^{\mathtt{0}}(\Gamma, \mathtt{w}, \lfloor \mathtt{T} \rfloor)$ and the rest follows. □

Lemma 9.7 (Value Typing). *Let Γ_1 be a well-formed typing environment with respect to a lifetime $\mathtt{l}$; let Γ_2 be a typing environment; let σ be a store typing; let $\mathtt{T}$ be a type; and, let $\mathtt{v}$ be a value. If $\Gamma_1 \vdash \langle\, \mathtt{v} : \mathtt{T} \,\rangle_{\sigma}^{\mathtt{l}} \dashv \Gamma_2$ then $\Gamma_1 = \Gamma_2$.*

Proof. Straightforward by inspection of T-Const, T-MutBorrow and T-ImmBorrow (the only rules for typing values). □

9.2 Borrow Invariance Lemma

The borrow invariance lemma ensures, roughly speaking, that the well-formedness of environments is preserved by the typing rules (recall Definition 4.8, page 28). The following corresponds to Lemma 4.9 (page 29):

Lemma 4.9. *Let $\mathcal{S}_1 \triangleright \mathtt{t}$ be a valid state; let σ be a store typing where $\mathcal{S}_1 \triangleright \mathtt{t} \vdash \sigma$; let Γ_1 be a well-formed typing environment with respect to a lifetime $\mathtt{l}$ where $\mathcal{S}_1 \sim \Gamma_1$ and Γ_2 be an arbitrary typing environment; let $\mathtt{t}$ be a term; and, let $\mathtt{T}$ be a type. If $\Gamma_1 \vdash \langle\, \mathtt{t} : \mathtt{T} \,\rangle_{\sigma}^{\mathtt{l}} \dashv \Gamma_2$, then $\Gamma_2[\gamma \mapsto \langle \mathtt{T} \rangle^{\mathtt{l}}]$ is well formed with respect to $\mathtt{l}$ for arbitrary $\gamma \in$ fresh.*

Proof. By structural induction on the structure of $\mathtt{t}$ according to Figure 1:

- **Base Case** $\mathtt{t} \triangleq [\mathtt{v}]$. By T-Const, $\Gamma_1 = \Gamma_2$. If $\mathtt{v} \not\triangleq [\ell^{\circ}]$ then follows trivially as adding $\gamma \mapsto \langle \mathtt{T} \rangle^{\mathtt{l}}$ to Γ_1 cannot invalidate well-formedness (where $\sigma \vdash \mathtt{v} : \mathtt{T}$). Otherwise $\mathtt{v} \triangleq [\ell^{\circ}]$ and follows by Def 4.5.
- **Base Case** $\mathtt{t} \triangleq [\hat{\mathtt{w}}]$. By T-Copy, we have $\mathtt{copy}(\mathtt{T})$ and $\Gamma_1 = \Gamma_2$. Hence, adding $\gamma \mapsto \langle \mathtt{T} \rangle^{\mathtt{l}}$ to Γ_1 cannot invalidate well-formedness (i.e since $\mathtt{T}$ already exists in Γ_1 without invalidating it).

- **Base Case** $\mathtt{t} \triangleq [\mathtt{w}]$. By T-Move, we have $\Gamma_2 = \mathtt{move}(\Gamma_1, \mathtt{w})$. By inspection of Def 3.18, $\mathtt{move}(\Gamma_1, \mathtt{w})$ replaces exactly one occurrence of $\mathtt{T}$ with $[\mathtt{T}]$. Since $\neg\mathtt{writeProhibited}(\Gamma_1, \mathtt{w})$ this cannot invalidate well-formedness, hence Γ_2 is well formed. Since $\mathtt{T}$ existed safely in Γ_1, it follows that $\Gamma_2[\gamma \mapsto \langle \mathtt{T} \rangle^{\mathtt{l}}]$ is well formed.
- **Base Case** $\mathtt{t} \triangleq [\&[\mathtt{mut}]\ \mathtt{w}]$. By both T-ImmBorrow and T-MutBorrow, $\Gamma_1 = \Gamma_2$ and $\Gamma \vdash \mathtt{w} : \langle \mathtt{T} \rangle^{\mathtt{m}}$. The latter implies $\mathtt{w}$ identifies valid existing state, hence $\Gamma_1[\gamma \mapsto \langle \&[\mathtt{mut}]\ \mathtt{w} \rangle^{\mathtt{l}}]$ is well formed.
- **Inductive Case** $\mathtt{t}_1 \triangleq [\mathtt{box}\ \mathtt{t}_2]$. By T-Box, $\Gamma_1 \vdash \langle\ \mathtt{t}_2 : \mathtt{T}_2\ \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2$ and, by inductive hypothesis, $\Gamma_2[\gamma \mapsto \langle \mathtt{T}_2 \rangle^{\mathtt{l}}]$ is well-formed. Hence, it follows that $\Gamma_2[\gamma \mapsto \langle \Box\mathtt{T}_2 \rangle^{\mathtt{l}}]$ is well formed.
- **Inductive Case** $\mathtt{t}_1 \triangleq [\mathtt{let}\ \mathtt{mut}\ \mathtt{x} = \mathtt{t}_2]$. By T-Declare, $\Gamma_1 \vdash \langle\ \mathtt{t}_2 : \mathtt{T}_2\ \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma$ and, by inductive hypothesis, $\Gamma[\gamma \mapsto \langle \mathtt{T}_2 \rangle^{\mathtt{l}}]$ is well-formed. By T-Declare, $\mathtt{x} \notin \mathbf{dom}(\Gamma_1)$ and $\Gamma_2 = \Gamma[\mathtt{x} \mapsto \langle \mathtt{T} \rangle^{\mathtt{l}}]$. By inspection, $\mathtt{x} \notin \mathbf{dom}(\Gamma)$ and well-formedness of $\Gamma_2[\gamma \mapsto \langle \epsilon \rangle^{\mathtt{l}}]$ follows trivially.
- **Inductive Case** $\mathtt{t}_1 \triangleq [\mathtt{w} = \mathtt{t}_2]$. By T-Assign, $\Gamma_1 \vdash \langle\ \mathtt{t}_2 : \mathtt{T}_2\ \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma$ and, by inductive hypothesis, $\Gamma[\gamma \mapsto \langle \mathtt{T}_2 \rangle^{\mathtt{l}}]$ is well-formed. By T-Assign, we have $\Gamma_1 \vdash \mathtt{w} : \langle \mathtt{T}_1 \rangle^{\mathtt{m}}$ where $\Gamma \vdash \mathtt{T}_2 \geq \mathtt{m}$ and $\Gamma_2 = \mathtt{write}^{\emptyset}(\Gamma, \mathtt{w}, \mathtt{T}_2)$. Then, follows that $\Gamma_2[\gamma \mapsto \langle \epsilon \rangle^{\mathtt{l}}]$ is well formed.
- **Inductive Case** $\mathtt{t} \triangleq [\{\overline{\mathtt{t}}\}^{\mathtt{m}}]$. By inductive hypothesis and T-Seq, $\Gamma_1 \vdash \langle\ \overline{\mathtt{t}} : \mathtt{T}\ \rangle^{\mathtt{m}}_{\sigma} \dashv \Gamma$ for fresh $\mathtt{l} \geq \mathtt{m}$ and where $\Gamma[\gamma \mapsto \langle \mathtt{T} \rangle^{\mathtt{m}}]$ is well-formed. By T-Block, we have $\Gamma_2 = \mathtt{drop}(\Gamma, \mathtt{m})$ and, hence, Γ_2 is well formed since only locations declared in lifetime $\mathtt{m}$ are removed. Furthermore, since $\Gamma \vdash \mathtt{T} \geq \mathtt{l}$, follows $\mathtt{T}$ is not invalidated (i.e. was not a borrow to a dropped location). Then, by inspection of Defintion 3.20, follows $\Gamma_2[\gamma \mapsto \langle \mathtt{T} \rangle^{\mathtt{l}}]$ is well formed. □

Here, the inductive case for blocks is perhaps the most involved we now consider it further. Since dropping only removes locations, we need only ensure this doesn't create dangling references (i.e. as it does not introduce mutable borrows which might break the first borrow invariant). This follows as the well-formedness of Γ_2 ensures no borrow $\mathtt{y}$ has a lifetime $\mathtt{l}$ which outlives the lifetime $\mathtt{m}$ of its referent (i.e. $\mathtt{l} \geq \mathtt{m}$ does not hold).

4.3 Progress Lemma

The progress lemma states, roughly speaking, that a well-typed term will reduce at least one step (this corresponds to Lemma 4.10 on page 29). For a given term $\mathtt{t}$, we have two cases. Either $\mathtt{t}$ is already a value or a term within $\mathtt{t}$ reduces by one step to $\mathtt{t}'$:

Lemma 4.10 (Progress). *Let $\mathcal{S}_1 \triangleright \mathtt{t}_1$ be a valid state; let σ be a store typing where $\mathcal{S}_1 \triangleright \mathtt{t}_1 \vdash \sigma$; let Γ_1 be a well-formed typing environment with respect to a lifetime $\mathtt{l}$ where $\mathcal{S}_1 \sim \Gamma_1$; let Γ_2 be a typing environment; and, let $\mathtt{T}$ be a type. If $\Gamma_1 \vdash \langle\ \mathtt{t}_1 : \mathtt{T}\ \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2$ then either $\mathtt{t}_1 \in \mathtt{Value}$ or $\langle\ \mathcal{S}_1 \triangleright \mathtt{t}_1 \longrightarrow \mathcal{S}_2 \triangleright \mathtt{t}_2\ \rangle^{\mathtt{l}}$ for some state $\mathcal{S}_2 \triangleright \mathtt{t}_2$.*

Proof. By structural induction on the possible forms of $\mathtt{t}_1$ according to Figure 1:

- **Base Case** $\mathtt{t}_1 \triangleq [\hat{\mathtt{w}}]$. By T-Copy, $\Gamma_1 \vdash \mathtt{w} : \langle \mathtt{T} \rangle^{\mathtt{m}}$. By Lemma 9.3, $\mathtt{read}(\mathcal{S}_1, \mathtt{w})$ is defined and, hence, R-Copy applies.
- **Base Case** $\mathtt{t}_1 \triangleq [\mathtt{w}]$. By T-Move, $\Gamma_1 \vdash \mathtt{w} : \langle \mathtt{T} \rangle^{\mathtt{m}}$. Hence, both $\mathtt{read}(\mathcal{S}_1, \mathtt{w})$ and $\mathtt{write}(\mathcal{S}_1, \mathtt{w}, \bot)$ are defined by Lemma 9.3 and R-Move applies.
- **Base Case** $\mathtt{t}_1 \triangleq [\&[\mathtt{mut}]\ \mathtt{w}]$. By either T-ImmBorrow or T-MutBorrow, $\Gamma_1 \vdash \mathtt{w} : \langle \mathtt{T} \rangle^{\mathtt{m}}$ and by Lemma 9.3 $\mathtt{loc}(\mathcal{S}_1, \mathtt{w})$ is defined. Hence, R-Borrow applies.
- **Base Case** $\mathtt{t}_1 \triangleq [\mathtt{box}\ \mathtt{v}]$. Straightforward as R-Box applies for fresh location ℓ_n.
- **Base Case** $\mathtt{t}_1 \triangleq [\mathtt{let}\ \mathtt{mut}\ \mathtt{x} = \mathtt{v}]$. By T-Declare $\mathtt{x} \notin \mathbf{dom}(\Gamma_1)$. Then, since $\mathcal{S}_1 \sim \Gamma_1$ we have $(\mathbf{dom}(\mathcal{S}_1) - \mathcal{L}) = \Theta(\mathbf{dom}(\Gamma_1))$ by Def 4.7. By construction, $\ell_{\mathtt{x}} \notin \mathcal{L}$ (i.e. it's not a heap location), hence $\ell_{\mathtt{x}} \notin \mathbf{dom}(\mathcal{S}_1)$ and R-Declare applies.

- **Base Case** $\mathtt{t}_1 \triangleq [\mathtt{w} = \mathtt{v}]$. By T-Assign, $\Gamma_1 \vdash \mathtt{w} : \langle \widetilde{\mathtt{T}} \rangle^{\mathtt{m}}$ and $\mathtt{read}(\mathcal{S}_1, \mathtt{w}) = \langle \mathtt{v}' \rangle^{\mathtt{m}}$ by Corollary 9.4. Since $\mathcal{S}_1 \sim \Gamma_1$, follows that $\mathcal{S} = \mathtt{drop}(\mathcal{S}_1, \{\mathtt{v}'\})$ is defined. It remains to show $\mathtt{write}(\mathcal{S}, \mathtt{w}, \mathtt{v})$ is defined. This won't hold only if $\mathtt{loc}(\mathcal{S}, \mathtt{w})$ traversed a location dropped between $\mathcal{S}_1$ and $\mathcal{S}$. By inspection of Def 3.4, such a location must be a heap location involved in a cycle with itself (e.g. $\{\ell_{\mathtt{x}} \mapsto \langle \square\square\&\mathtt{x} \rangle^1\}$ for "$*\mathtt{x} = \mathtt{v}$"). Since $\mathcal{S}_1 \sim \Gamma_1$, any cycle could only involve a borrow. Since $\neg\mathtt{writeProhibited}(\Gamma_1, \mathtt{w})$ by T-Assign, R-Assign applies.
- **Base Case** $\mathtt{t}_1 \triangleq [\{\mathtt{v}\}^{\mathtt{m}}]$. By inspection of Def 3.4, $\mathtt{drop}(\mathcal{S}_1, \mathtt{m})$ is defined. Hence, R-BlockB applies.
- **Inductive Case** $\mathtt{t}_1 \triangleq [\mathtt{box}\ \mathtt{t}_2]$ where $\mathtt{t}_2 \not\triangleq [\mathtt{v}]$. Follows by inductive hypothesis.
- **Inductive Case** $\mathtt{t}_1 \triangleq [\mathtt{let\ mut\ x} = \mathtt{t}_2]$ where $\mathtt{t}_2 \not\triangleq [\mathtt{v}]$. Follows by inductive hypothesis.
- **Inductive Case** $\mathtt{t}_1 \triangleq [\mathtt{w} = \mathtt{t}_2]$ where $\mathtt{t}_2 \not\triangleq [\mathtt{v}]$. Follows by inductive hypothesis.
- **Inductive Case** $\mathtt{t} \triangleq [\{\overline{\mathtt{t}}\}^{\mathtt{m}}]$ where $\overline{\mathtt{t}} \not\triangleq [\mathtt{v}]$. Follows by inductive hypothesis. □

4.4 Preservation Lemma

The preservation lemma states, roughly speaking, that after a well-typed statement has reduced the typing environment remains a safe abstraction of the runtime environment. However, compared with the progress lemma, the preservation lemma is more involved and requires several supporting lemmas. The first of these ensures aliasing amongst heap locations is prohibited (recall Definition 4.3 from page 27):

Lemma 9.8 (Alias Preservation). *Let $\mathcal{S}_1 \triangleright \mathtt{t}$ be a valid state and $\mathcal{S}_2 \triangleright \mathtt{v}$ a terminal state; let σ be a store typing where $\mathcal{S}_1 \triangleright \mathtt{t} \vdash \sigma$; let Γ_1 be a well-formed typing environment with respect to a lifetime $\mathtt{l}$ where $\mathcal{S}_1 \sim \Gamma_1$; let Γ_2 be a typing environment; and, let $\mathtt{T}$ be a type. If $\Gamma_1 \vdash \langle\ \mathtt{t} : \mathtt{T}\ \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2$ and $\langle\ \mathcal{S}_1 \triangleright \mathtt{t} \longrightarrow \mathcal{S}_2 \triangleright \mathtt{v}\ \rangle^1$ then $\mathcal{S}_2 \triangleright \mathtt{v}$ remains valid.*

Proof. By case analysis on the structure of $\mathtt{t}$ for terms which can reduce to a value in one step:

- **Base Case** $\mathtt{t}_1 \triangleq [\hat{\mathtt{w}}]$. Straightforward. Observe $\mathcal{S}_1 = \mathcal{S}_2$ by R-Copy and $\Gamma_1 = \Gamma_2$ by T-Copy. Then, follows by T-Copy as $\mathtt{copy}(\mathtt{T})$ implies $\mathtt{v} \triangleq [\ell^{\bullet}]$ cannot hold.
- **Base Case** $\mathtt{t}_1 \triangleq [\mathtt{w}]$. By R-Move, $\mathcal{S}_2 = \mathtt{write}(\mathcal{S}_1, \mathtt{w}, \bot)$ for some $\ell_{\mathtt{w}} \in \mathbf{dom}(\mathcal{S}_1)$. Since $\mathcal{S}_1 \triangleright \mathtt{v}_2$ was valid, follows that $\mathcal{S}_2$ is valid. Suppose $\mathtt{v} \triangleq [\ell^{\bullet}]$ (since otherwise trivial). By inspection of Def 3.23, follows that $\mathtt{v}$ is not contained in $\mathcal{S}_2$. Thus, validity of $\mathcal{S}_2 \triangleright \mathtt{v}$ follows.
- **Base Case** $\mathtt{t}_1 \triangleq [\&[\mathtt{mut}]\ \mathtt{w}]$. Straightforward. Observe $\mathcal{S}_1 = \mathcal{S}_2$ by R-Borrow and $\Gamma_1 = \Gamma_2$ by both T-ImmBorrow and T-MutBorrow. Then, follows by R-Borrow as $\mathtt{v} \triangleq [\ell^{\circ}]$.
- **Base Case** $\mathtt{t}_1 \triangleq [\mathtt{box}\ \mathtt{v}_2]$. By R-Box, $\mathcal{S}_2 = \mathcal{S}_1[\ell_{\mathtt{n}} \mapsto \langle \mathtt{v}_2 \rangle^*]$ for some $\ell_{\mathtt{n}} \notin \mathbf{dom}(\mathcal{S}_1)$. Since $\mathcal{S}_1 \triangleright \mathtt{v}_2$ was valid, follows that $\mathcal{S}_2$ is valid. Finally, follows by R-Box as $\mathtt{v} = \ell^{\bullet}_{\mathtt{n}}$.
- **Base Case** $\mathtt{t}_1 \triangleq [\mathtt{let\ mut\ x} = \mathtt{v}_2]$. By R-Declare, $\mathcal{S}_2 = \mathcal{S}_1[\ell_{\mathtt{x}} \mapsto \langle \mathtt{v}_2 \rangle^*]$ where $\ell_{\mathtt{x}} \notin \mathbf{dom}(\mathcal{S}_1)$. Since $\mathcal{S}_1 \triangleright \mathtt{v}_2$ was valid, follows that $\mathcal{S}_2$ is valid. Finally, follows by R-Declare as $\mathtt{v} \triangleq [\epsilon]$.
- **Base Case** $\mathtt{t}_1 \triangleq [\mathtt{w} = \mathtt{v}_2]$. By R-Assign, $\mathcal{S}_2 = \mathtt{write}(\mathcal{S}_1, \mathtt{w}, \mathtt{v}_2)$ for some $\ell_{\mathtt{w}} \in \mathbf{dom}(\mathcal{S}_1)$. Since $\mathcal{S}_1 \triangleright \mathtt{v}_2$ was valid, follows that $\mathcal{S}_2$ is valid. Finally, follows by R-Assign as $\mathtt{v} \triangleq [\epsilon]$.
- **Base Case** $\mathtt{t} \triangleq [\{\mathtt{v}\}^{\mathtt{m}}]$. By R-BlockB, $\mathcal{S}_2 = \mathtt{drop}(\mathcal{S}_1, \mathtt{m})$. By T-Block, $\Gamma_2 \vdash \mathtt{T} \geq \mathtt{l}$ and, hence, validity of $\mathcal{S}_1 \triangleright \mathtt{t}$ implies validity of $\mathcal{S}_2 \triangleright \mathtt{v}$.

□

Lemma 9.9 (Value Preservation). *Let $\mathcal{S}_1 \triangleright \mathtt{t}$ be a valid state and $\mathcal{S}_2 \triangleright \mathtt{v}$ a terminal state; let σ be a store typing where $\mathcal{S}_1 \triangleright \mathtt{t} \vdash \sigma$; let Γ_1 be a well-formed typing environment with respect to a lifetime $\mathtt{l}$ where $\mathcal{S}_1 \sim \Gamma_1$; let Γ_2 be a typing environment; and, let $\mathtt{T}$ be a type. If $\Gamma_1 \vdash \langle\ \mathtt{t} : \mathtt{T}\ \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2$ and $\langle\ \mathcal{S}_1 \triangleright \mathtt{t} \longrightarrow \mathcal{S}_2 \triangleright \mathtt{v}\ \rangle^{1}$ then $\mathcal{S}_2 \vdash \mathtt{v} \sim \mathtt{T}$.*

Proof. By case analysis on the structure of $\mathtt{t}$ for terms which can reduce to a value in one step:

- **Base Case** $\mathtt{t}_1 \triangleq [\hat{\mathtt{w}}]$. By R-Copy $\mathtt{read}(\mathcal{S}_1, \mathtt{w}) = \langle \mathtt{v} \rangle^{\mathtt{n}}$ and by T-Copy $\Gamma_1 \vdash \mathtt{w} : \langle \mathtt{T} \rangle^{\mathtt{m}}$. Hence, follows by Corollary 9.4.
- **Base Case** $\mathtt{t}_1 \triangleq [\mathtt{w}]$. By R-Move $\mathtt{read}(\mathcal{S}_1, \mathtt{w}) = \langle \mathtt{v} \rangle^{\mathtt{n}}$ and by T-Move $\Gamma_1 \vdash \mathtt{w} : \langle \mathtt{T} \rangle^{\mathtt{m}}$. Hence, follows by Corollary 9.4.
- **Base Case** $\mathtt{t}_1 \triangleq [\&[\mathtt{mut}]\ \mathtt{w}]$. Follows trivially by Lemma 9.3.
- **Base Case** $\mathtt{t}_1 \triangleq [\mathtt{box}\ \mathtt{v}_2]$. By T-Const, $\sigma \vdash \mathtt{v}_2 : \mathtt{T}_2$. By R-Box, $\mathcal{S}_2 = \mathcal{S}_1[\ell_{\mathtt{n}} \mapsto \langle \mathtt{v}_2 \rangle^*]$ where $\mathtt{v} = \ell^{\bullet}_{\mathtt{n}}$. Hence, $\mathcal{S}_2 \vdash \ell^{\bullet}_{\mathtt{n}} \sim \square \mathtt{T}_2$ follows trivially from Def 4.4.
- **Base Case** $\mathtt{t}_1 \triangleq [\mathtt{let\ mut\ x} = \mathtt{v}_2]$. Straightforward since $\emptyset \vdash \epsilon \sim \epsilon$ by Def 4.4.
- **Base Case** $\mathtt{t}_1 \triangleq [\mathtt{w} = \mathtt{v}_2]$. Straightforward since $\emptyset \vdash \epsilon \sim \epsilon$ by Def 4.4.
- **Base Case** $\mathtt{t} \triangleq [\{\mathtt{v}\}^{\mathtt{m}}]$. Straightforward since $\sigma \vdash \mathtt{v} : \mathtt{T}$ by T-Const.

□

The next lemma ensures that, after a given transition, the typing environment remains a safe abstraction of the runtime program store.

Lemma 9.10 (Store Preservation). *Let $\mathcal{S}_1 \triangleright \mathtt{t}$ be a valid state and $\mathcal{S}_2 \triangleright \mathtt{v}$ a terminal state; let σ be a store typing where $\mathcal{S}_1 \triangleright \mathtt{t} \vdash \sigma$; let Γ_1 be a well-formed typing environment with respect to a lifetime $\mathtt{l}$ where $\mathcal{S}_1 \sim \Gamma_1$; let Γ_2 be a typing environment; and, let $\mathtt{T}$ be a type. If $\Gamma_1 \vdash \langle\ \mathtt{t} : \mathtt{T}\ \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2$ and $\langle\ \mathcal{S}_1 \triangleright \mathtt{t} \longrightarrow \mathcal{S}_2 \triangleright \mathtt{v}\ \rangle^{1}$ then $\mathcal{S}_2 \sim \Gamma_2$.*

Proof. By case analysis on the structure of $\mathtt{t}$ for terms which can reduce to a value in one step:

- **Base Case** $\mathtt{t}_1 \triangleq [\hat{\mathtt{w}}]$. Follows immediately as $\mathcal{S}_1 = \mathcal{S}_2$ by R-Copy and $\Gamma_1 = \Gamma_2$ by T-Copy.
- **Base Case** $\mathtt{t}_1 \triangleq [\mathtt{w}]$. By R-Move, $\mathcal{S}_2 = \mathtt{write}(\mathcal{S}_1, \mathtt{w}, \bot)$. By T-Move, we have $\Gamma_1 \vdash \mathtt{w} : \langle \mathtt{T}_{\mathtt{w}} \rangle^{\mathtt{m}}$ and $\Gamma_2 = \mathtt{move}(\Gamma_1, \mathtt{w})$ where $\neg \mathtt{writeProhibited}(\Gamma_1, \mathtt{w})$. By Def 4.4, $\mathcal{S} \vdash \bot \sim \lfloor \mathtt{T}_{\mathtt{w}} \rfloor$ and, hence, $\mathcal{S}_2 \sim \Gamma_2$ follows since $\mathtt{w}$ is not borrowed.
- **Base Case** $\mathtt{t}_1 \triangleq [\&[\mathtt{mut}]\ \mathtt{w}]$. Follows immediately as $\mathcal{S}_1 = \mathcal{S}_2$ by R-Borrow and $\Gamma_1 = \Gamma_2$ by both T-ImmBorrow and T-MutBorrow.
- **Base Case** $\mathtt{t}_1 \triangleq [\mathtt{box}\ \mathtt{v}_2]$. By R-Box, $\mathtt{v}_1 = \ell^{\bullet}_{\mathtt{n}}$ and $\mathcal{S}_2 = \mathcal{S}_1[\ell_{\mathtt{n}} \mapsto \langle \mathtt{v}_2 \rangle^*]$ for some $\ell_n \notin \mathbf{dom}(\mathcal{S}_1)$. Hence, $(\mathbf{dom}(\mathcal{S}_2) - \mathcal{L}) = (\mathbf{dom}(\mathcal{S}_1) - \mathcal{L})$. By T-Box, $\Gamma_1 \vdash \langle\ \mathtt{v}_2 : \mathtt{T}_2\ \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2$ and $\Gamma_1 = \Gamma_2$ follows by Lemma 9.7. Thus, $\mathcal{S}_2 \sim \Gamma_2$ follows directly from $\mathcal{S}_1 \sim \Gamma_1$.
- **Base Case** $\mathtt{t}_1 \triangleq [\mathtt{let\ mut\ x} = \mathtt{v}_2]$. By R-Declare, we have $\mathcal{S}_2 = \mathcal{S}_1[\ell_{\mathtt{x}} \mapsto \langle \mathtt{v}_2 \rangle^1]$. By T-Declare and Lemma 9.7, $\Gamma_1 \vdash \langle\ \mathtt{v}_2 : \mathtt{T}_2\ \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_1$ follows. Hence, $\Gamma_2 = \Gamma_1[\mathtt{x} \mapsto \langle \mathtt{T}_2 \rangle^1]$ by T-Declare. Observe, $\mathtt{x} \notin \mathbf{dom}(\Gamma_1)$ implies $\ell_{\mathtt{x}} \notin \mathbf{dom}(\mathcal{S}_1)$, hence $(\mathbf{dom}(\mathcal{S}_2) - \mathcal{L}) = \mathbf{dom}(\Gamma_2)$. Thus, $\mathcal{S}_2 \sim \Gamma_2$ follows.
- **Base Case** $\mathtt{t}_1 \triangleq [\mathtt{w} = \mathtt{v}_2]$. By R-Assign, $\mathcal{S} = \mathtt{drop}(\mathcal{S}_1, \{\mathtt{v}_1\})$ and $\mathcal{S}_2 = \mathtt{write}(\mathcal{S}, \mathtt{w}, \mathtt{v}_2)$ where $\mathtt{read}(\mathcal{S}_1, \mathtt{w}) = \langle \mathtt{v}_1 \rangle^{\mathtt{m}}$. By T-Assign, $\Gamma_1 \vdash \mathtt{w} : \mathtt{T}_1$ where $\mathcal{S}_1 \vdash \mathtt{v}_1 \sim \mathtt{T}_1$ by Lemma 9.3. By T-Assign and Lemma 9.7, $\Gamma_1 \vdash \langle\ \mathtt{v}_2 : \mathtt{T}_2\ \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_1$ follows and, hence, $\Gamma_2 = \mathtt{write}^{\emptyset}(\Gamma_1, \mathtt{w}, \mathtt{T}_2)$. Then, follows by Lemma 9.6 since $\mathcal{S}_1 \vdash \mathtt{v}_2 \sim \mathtt{T}_2$.

- **Base Case** $\mathtt{t} \triangleq [\{\mathtt{v}\}^{\mathtt{m}}]$. By R-BlockB, $\mathcal{S}_2 = \mathtt{drop}(\mathcal{S}_1, \mathtt{m})$. By T-Block and Lemma 9.7, we have $\Gamma_1 \vdash \langle\ \mathtt{v} : \mathtt{T}\ \rangle^{\mathtt{m}}_{\sigma} \dashv \Gamma_1$ follows. Hence, $\Gamma_2 = \mathtt{drop}(\Gamma_1, \mathtt{m})$. Then, $\mathcal{S}_2 \sim \Gamma_2$ by Lemma 9.5. □

We can now establish the preservation lemma for well-typed terms (this corresponds to Lemma 4.11, page 29). Recall that $\rightsquigarrow$ denotes a reduction involving zero or more steps:

Lemma 4.11 (Preservation). *Let $\mathcal{S}_1 \triangleright \mathtt{t}$ be a valid state and $\mathcal{S}_2 \triangleright \mathtt{v}$ a terminal state; let σ be a store typing where $\mathcal{S}_1 \triangleright \mathtt{t} \vdash \sigma$; let Γ_1 be a well-formed typing environment with respect to a lifetime $\mathtt{l}$ where $\mathcal{S}_1 \sim \Gamma_1$; let Γ_2 be a typing environment; and, let $\mathtt{T}$ be a type. If $\Gamma_1 \vdash \langle\ \mathtt{t} : \mathtt{T}\ \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2$ and $\langle\ \mathcal{S}_1 \triangleright \mathtt{t} \rightsquigarrow \mathcal{S}_2 \triangleright \mathtt{v}\ \rangle^{\mathtt{l}}$ then $\mathcal{S}_2 \triangleright \mathtt{v}$ remains valid where $\mathcal{S}_2 \sim \Gamma_2$ and $\mathcal{S}_2 \vdash \mathtt{v} \sim \mathtt{T}$.*

Proof. Follows by structural induction on the reduction relation ($\rightsquigarrow$) using Lemmas 9.8, 9.10 and 9.9 for the base cases. □

4.5 Type and Borrow Safety Theorem

We now bring all of the pieces together to establish the main type and borrow safety theorem (this corresponds to Theorem 4.12, page 30).

Theorem 4.12 (Type and Borrow Safety). *Let $\mathcal{S}_1 \triangleright \mathtt{t}$ be a valid state; let σ be a store typing where $\mathcal{S}_1 \triangleright \mathtt{t} \vdash \sigma$; let Γ_1 be a well-formed typing environment with respect to a lifetime $\mathtt{l}$ where $\mathcal{S}_1 \sim \Gamma_1$; let Γ_2 be a typing environment; and, let $\mathtt{T}$ be a type. If $\Gamma_1 \vdash \langle\ \mathtt{t} : \mathtt{T}\ \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2$, then $\langle\ \mathcal{S}_1 \triangleright \mathtt{t} \rightsquigarrow \mathcal{S}_2 \triangleright \mathtt{v}\ \rangle^{\mathtt{l}}$ for some terminal state $\mathcal{S}_2 \triangleright \mathtt{v}$.*

Proof. Follows trivially from Lemma 4.9, Lemma 4.10 and Lemma 4.11. □

4.5.1 Borrow Safety. Finally, we establish the borrow safety corollary (this corresponds to a variation of Corollary 4.14 from page 30 which has been strengthened for the calculus core).

Corollary 4.13 (Borrow Safety). *Let $\mathcal{S}_1 \triangleright \mathtt{t}_1$ be a valid state; let σ be a store typing where $\mathcal{S}_1 \triangleright \mathtt{t}_1 \vdash \sigma$; let Γ_1 be a well-formed borrow safe typing environment with respect to a lifetime $\mathtt{l}$ where $\mathcal{S}_1 \sim \Gamma_1$; let Γ_2 be a typing environment; and, let $\mathtt{T}$ be a type. If $\Gamma_1 \vdash \langle\ \mathtt{t}_1 : \mathtt{T}\ \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2$ then $\Gamma_2[\gamma \mapsto \langle \mathtt{T} \rangle^{\mathtt{l}}]$ is a well-formed and borrow safe typing environment for arbitrary $\gamma \in$ fresh.*

Proof. First observe $\Gamma_2[\gamma \mapsto \langle \mathtt{T} \rangle^{\mathtt{l}}]$ well-formed by Lemma 4.9. Then, follows trivially by inspection of typing rules for FR, noting: firstly, that mutable and immutable borrows can only be introduced by (respectively) T-MutBorrow and T-ImmBorrow; and, secondly, that mutable borrows cannot be copied. In other words, a mutable borrow for a given lval can only be introduced when no other borrow of that lval exists in the environment. □