Borrow checking Hylo

Dimi Racordon
Northeastern University
Boston, Massachusetts, USA
d.racordon@northeastern.edu

Dave Abrahams
Adobe
USA
dabrahams@adobe.com

Abstract

Hylo is a language for high-level systems programming that promises safety without loss of efficiency. It is based on mutable value semantics, a discipline that emphasizes the independence of values to support local reasoning. The result—in contrast with approaches based on sophisticated aliasing restrictions—is an efficient, expressive language with a simple type system and no need for lifetime annotations.

Safety guarantees in Hylo programs are verified by an abstract interpreter processing an intermediate representation, Hylo IR, that models lifetime properties with ghost instructions. Further, lifetime constraints are used to eliminate unnecessary memory allocations predictably.

Keywords: mutable value semantics, memory safety, borrowing, optimization, intermediate representation

ACM Reference Format:
Dimi Racordon and Dave Abrahams. 2023. Borrow checking Hylo. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 9 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The growth of our reliance on software has created an urgent need for better tools and practices to guarantee safety and correctness. Concerns over software defects have reached beyond industry and academia, sparking government action worldwide [6, 7].

This crisis is due in large part to the use of so-called unsafe programming languages, like C and C++, which are prone to serious security vulnerabilities. Despite their flaws, however, unsafe languages remain ubiquitous, even in new projects, because safer alternatives often impose prohibitive run-time costs in resource-critical applications. To address this issue, some newer languages have been designed to be safe by construction while avoiding the need for expensive run-time safety mechanisms.

Enter Hylo [1], a project that promises memory safety with zero-cost abstraction. Unlike languages with similar goals such as Rust [13], Hylo does not attempt to police references with a sophisticated type system; instead, it simply removes them from the user model. Variables cannot share mutable state, eliminating data races, and object lifetimes can be tracked trivially, eliminating temporal safety violations. As a result, Hylo is immune to most commonly reported software vulnerabilities [4].

The theoretical model of Hylo bears more similarities to a functional language with a polished syntax for the state monad than to a traditional imperative one. Its programming model, though, includes first-class part-wise in-place mutation, efficiently implemented via references [18]. Hylo's compiler also uses references to implement patterns that traditionally rely on aliasing. Hence, program analysis is needed to avoid the numerous hazards that come with the unrestricted use of references.

The analysis is implemented using an abstract interpreter that processes an intermediate representation (IR) resembling single-static assignment form [5]. In this IR, aliasing is modeled using automatically-generated *ghost instructions*, which only exist for analysis purposes and have no operational semantics [8].

This approach allows us to describe borrow checking—the process of upholding memory safety—in terms of the language's operational semantics, extended for ghost instructions. The resulting implementation is simple to extend and debug. Adding new instructions only involves teaching the interpreter about their effects on abstract states. Further, one can step through abstract interpretation to observe its behavior. That is in contrast to Rust's approach [17] of collecting and then solving a set of equational constraints, which operates on a more declarative description of the type system, farther from the operational semantics.

This paper reports the current status of the Hylo borrow checker. After a presentation of some key user-level features, we briefly introduce Hylo's intermediate representation, describe the implementation of our lifetime analysis, and comment on future challenges.

2 Hylo in a Nutshell

Support for mutable value semantics (MVS) [18]—a discipline that upholds the independence of values to support

IWACO'23, October 22–27, 2023, Cascais, Portugal

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

local reasoning—is central to Hylo. In its purest form, MVS removes references from the user model, meaning that all data structures must form disjoint topological trees. While this restriction is reminiscent of pure functional programming, a key difference is that MVS allows part-wise, in-place mutation, as commonly found in traditional imperative languages. For instance, in the following program, the second part of `s` is mutated in-place at line 3 without reassigning the whole pair:

```
1 public fun main() {
2   var s = (x: 4, y: 2)
3   &s.x += 6
4   print(s) // "(4, 8)"
5 }
```

Note that the ampersand (&) in line 3 is merely a marker required by Hylo on all mutated values, and does not denote the address-of operator as in C or Rust.

Linearity. Hylo enforces MVS using a form of linear type system [20]. Such systems require that all values be used *exactly* once, thus guaranteeing freedom from aliasing and resource leakage. Most are, however, extremely restrictive and have been widely criticized for being impractical. Fortunately, Hylo makes two observations to address this issue.

The first is that destruction need not be explicit, as the compiler can detect unused values and destroy them automatically.[1] The second is that reading or modifying a value is safe as long as such value cannot be modified concurrently. As a result, Hylo only requires that all values be *consumed* (returned from a function, moved to new storage, or deinitialized) exactly once. In the above program, for instance, `s` can be used several times because neither modifying it at line 3 nor reading it at line 4 requires consumption.

Local bindings. No value is copied implicitly in Hylo. Consequently, assigning a value to a mutable local variable is always a consuming operation, regardless of the value's type. For instance, the following program is illegal because it accesses the value of `s` after it has been relocated to `t`:

```
1 public fun main() {
2   var s = 4
3   var t = s
4   print(s + t) // error: 's' was consumed
5 }
```

This error can be fixed by explicitly copying `s`, thereby producing a new independent value, or by declaring `t` as a *subscript* of `s`. The two kinds of projection, **`let`** and **`inout`**, denote immutable and mutable access, respectively.

Per the principles of MVS, for all intents and purposes, all bindings can be thought of as independent values. However, to uphold this property, declaring one binding may cause another to become inaccessible.

```
1 public fun main() {
2   var s = (x: 4, y: 2)
3   inout t = &s.x
4   _ = t    // 't' is consumed here
5   print(s) // error: 's' is inaccessible
6   &t = 8   // 't' is reconstituted here
7 }
```

The rule that makes `s` inaccessible in line 5 exists because part of `s` is being accessed mutably via `t`—the values are interdependent and therefore cannot be observed together. The semantics are as though that part has been moved into `t` and will be moved back to `s` when `t`'s lifetime ends.

Hylo uses non-lexical lifetimes: the lifetime of a Hylo binding ends just after the binding's last use. Hence, swapping the two last statements above fixes the error by pushing the call to `print` outside of `t`'s lifetime.

While there is no universally-accepted definition of "reference", we make the key distinction that references allow the observation of non-local effects. Indeed, in a system without mutation or exposed addresses, it is indistinguishable whether data is accessed "by reference" or "by value". Although **`let`** and **`inout`** bindings may be implemented using references, they do not allow the observation of non-local effects, which justifies our claim that semantically, they are values.

Passing conventions. Function signatures explicitly define the ways each argument may be accessed. Four conventions are available, denoting corresponding capabilities and duties given to parameters [16]:

- **`let`** is the default and denotes a parameter that can only be read.
- **`inout`** denotes a parameter that can be modified.
- **`sink`** denotes a parameter that can be read or modified and *must* be consumed.
- **`set`** denotes a parameter that starts uninitialized and must be initialized by the function.

Again, for all intents and purposes, all parameters can be thought of as an independent values: modifying a **`sink`**, **`inout`**, or **`set`** parameter cannot affect later operations on another variable in scope, including closure captures.

The choice of a particular convention has an impact on the efficiency of an operation. For example, consider the following function:

```
1 typealias Point = {x: Int, y: Int}
2 fun inc(_ p: let Point) -> Point {
3   (p.x + 1, p.y + 1)
4 }
```

From its signature, we know that `inc` cannot consume its argument and must create a new value. Hence, using it

[1] Unused values are not dropped from the typing context as in affine type systems [19]. Instead, the compiler automatically inserts their destruction into the source program.

in a functional update causes wasteful object creation and assignment:

```
1 public fun main() {
2   var s = (x: 4, y: 2)
3   &s = inc(s) // inefficient update
4   print(s)
5 }
```

The assignment at line 3 causes `inc` to create a brand new pair to be assigned to `s`, rather than reusing the memory already allocated. A better choice in this situation would have been to take `p` with the **`sink`** convention, allowing `inc` to take ownership of `p`'s value, modify its parts, and return it. Better yet, one could take `p` with the **`inout`** convention and implement `inc` as a proper in-place operation.

```
1 fun inc_inplace(_ p: inout Point) {
2   &p.x += 1
3   &p.y += 1
4 }
```

Explicitly choosing the most efficient variant of an operation depending on the surrounding code is tedious, error-prone, and forces callers into a low-level programming style that obscures semantics. To avoid this problem, Hylo allows us to bundle the variants and let the compiler select the correct one:

```
1 extension Point {
2   public fun inc() -> Point {
3     let { (x: x + 1, y: y + 1) }
4     inout { &x += 1; &y += 1 }
5     sink { &x += 1; &y += 1; return self }
6   }
7 }
```

Given the above definition and a pair `s`, `s.inc()` will apply the **`let`** or **`sink`** variant depending on whether `s` is used thereafter. For instance, the compiler will select the first one if `s` is read once more after the call.

Projections. In functional programming, a *lens* [9] allows manipulation of records using a pair of functions $S \to V$ and $V \to S \to S$. The former, often called a *getter*, presents a view of a source. The latter, often called a *setter*, returns a source updated with the value of a view. For example, the source might be a hash table and the view a value in it.

A naive way to realize a lens is to simply define such a pair of functions. Unfortunately, this solution is often inefficient for implementing updates. For instance, incrementing a value in a hash table with such a lens would require the construction of a whole new copy of the hash table with the changed parts replaced! A far more efficient approach is to use references to implement an in-place update. In Rust, for example, `HashMap` has a method `get_mut` that returning a reference to the value associated with a given key. Of course, since references are not available in Hylo, another mechanism is needed.

Before we look at this mechanism, let us discuss another way to provide access to a part of a value that involves neither getters and setters nor references.

```
1  extension UInt32 {
2    public fun with_bit<T>(
3      _ i: Int, _ f: (inout Bool) -> T
4    ) inout -> T {
5      var b = (self & (1 << i)) != 0
6      f(&b)
7      if b { &self |= 1 << i }
8      else { &self &= ~(1 << i) }
9    }
10 }
11
12 public fun main() {
13   var s: UInt32 = 0
14   &s.with_bit(4, fun(_ b) { &b = true })
15   print(s) // "16"
16 }
```

This example illustrates the use of a 32-bit unsigned integer as an array of Booleans. `with_bit` takes an index and calls a lambda with the value at that index, passed **`inout`**. In other words, it provides access to a partial view of an integer using inversion of control instead of references.

Using this approach, one can expose notional parts that have no inline memory representation. In the above program, for example, `with_bit` provides access to each bit of an integer as a Boolean value, synthesizing it on demand. Doing the same with references would require a proxy object [12], introducing all sorts of wrinkles in the type system.

On the other hand, using higher-order functions for routine updates tends to hinder legibility. Fortunately, Hylo offers syntactic support for defining access through inversion of control. For example, the above program can be rewritten as follows:

```
1  extension UInt32 {
2    public subscript(_ i: Int): Bool {
3      sink { return (self & (1 << i)) != 0 }
4      let { yield (self & (1 << i)) != 0 }
5      inout {
6        var b = self[i]
7        yield &b
8        if b { &self |= 1 << i }
9        else { &self &= ~(1 << i) }
10     }
11   }
12 }
13
14 public fun main() {
15   var s: UInt32 = 0
16   &s[4] = true
17   print(s) // "16"
18 }
```

The subscript declared at line 2 *projects* a `Bool` out of a `UInt32`, enabling the more familiar programming style

displayed at line 16. The yielded value of a subscript—unlike a function return value—is logically bound to its sources, with lifetime and independence guarantees upheld just as with local **`let`** or **`inout`** bindings.

Similarly to methods, different variants of a subscript can be bundled together, allowing the compiler to automatically select the correct one depending on the situation. Here, for example, the **`inout`** variant is selected at line 16 because the projection is used mutably and `s` is used again later. If the projection was used immutably, the compiler would select the **`let`** variant if `s` is still used or the **`sink`** variant otherwise.

3 Hylo IR

This section introduces Hylo IR, a typed, static single assignment (SSA) instruction set designed to express and check the flow-sensitive typing rules of the Hylo programming language. Only the essential features are presented here. A more complete description can be found in the documentation of the Hylo's reference implementation (https://github.com/Hylo-lang/Hylo).

3.1 Anatomy of a module

A program is a collection of modules, including one called the "main" module. A module is a collection of functions and constants. A main module contains at least one function called `main` that acts as the program's entry point.

Control flow within a function is represented using unstructured branch statements. The instructions of a function are partitioned into *basic blocks* whose final instructions either branch to another block or exit the function. Such instructions are called *terminators* and can only occur at the end of a basic block. Basic blocks may accept arguments representing values that depend on control flow.

Hylo IR is defined in terms of a machine with an unbounded number of registers, which can be seen as the variables of the IR. Note that registers do not have a one-to-one correspondence with variables in the surface-language. Instead, they merely denote the result of an instruction.

Each instruction is associated with a site in the source program so that error diagnostics can be reported at the right location. These sites also serve to setup debug information in compiled programs.

3.2 Memory model

Hylo IR is openly inspired by LLVM IR [14] and designed to support similar code patterns. In particular, variables in the surface-language are represented by writing and reading to and from stack allocations rather than SSA registers. These allocations are then expected to be optimized away further down the pipeline.[2] This approach enables a straightforward encoding of Hylo's memory model, in which objects are represented by the contents of some memory region identified by a typed address. Instructions manipulate the state of a program by allocating and deallocating memory for objects or writing and reading values to and from their storage.

Flow-insensitive typing rules are checked by the front-end before IR is generated, so types in Hylo IR mostly serve to describe memory layouts, of which two kinds are distinguished: *transparent* and *opaque*. When an object has a transparent layout, the addresses of its parts can be computed at compile-time, thus allowing the compiler to track partial borrows and moves. Accessing a part stored in object with an opaque layout must be assumed to borrow or move the whole object. Opaque layouts serve to raise an ABI boundary between modules, allowing library authors to modify the internal representation of public types without requiring client code to be recompiled.

3.3 Semantics

The abstract target machine for Hylo IR can be loosely defined as a 4-tuple $\langle F, C, R, M\rangle$ denoting the state of a program at run-time, where:

- F is a set of functions.
- C is a program counter.
- L is a stack of tables mapping registers and basic block arguments to values and memory addresses.
- M is a table mapping memory addresses to values.[3]

Evaluation starts with F assigned to the set of functions in the program, C to the entry's first instruction, L to a stack of a single empty map, and M to an empty map. Executing an instruction updates the top of L with its results, thereby defining new registers, and sets C to the next instruction. Allocating, deallocating, or writing a value to memory updates M. Calling a function pushes a new map on L.

3.4 Ghost instructions

Consider a variable declaration of the form **`var`** `x = foo()`. In the IR, this statement is decomposed into four instructions:

```
1 %1: &Int = alloc_stack Int
2 %2: &Int = borrow [set] %1
3 call @foo() to %2
4 end_borrow %2
```

The first allocates storage for `x` on the stack, returning the address of the allocated memory. The second creates an access for initialization. The third calls `foo` and writes its result to the allocated storage, through the above-created access. The last instruction closes the access.

Both `borrow` and `end_borrow` are *ghost instructions* [8]. They have no effect on the state of the program and only

[2] Our compiler relies on out-of-box optimizations from LLVM. Thanks to Hylo's guarantees, it can produce hints about uniqueness and immutability more precise than those coming from typical C or C++ front-ends, allowing optimizations to apply more aggressively.

[3] Note that memory addresses cannot be stored, which is a direct consequence of the absence of references in Hylo's user model.

serve to model aliasing for the borrow checker, so they are ignored during machine code generation.

Hylo IR currently features five ghost instructions:

- `borrow` creates a non-consuming access;
- `end_borrow` ends a non-consuming access; and
- `consume` creates a consuming access;
- `access` creates either a consuming or non-consuming access, depending on contextual information;
- `mark_state` unconditionally marks an object in memory with an initialization state.

4 Borrow checking

Borrow checking in Hylo has four stages, as shown in Figure 1. Since the analysis is intraprocedural, multiple IR functions can be checked in parallel. The analysis also transforms the raw IR produced by the front-end to a refined form ready for optimization and machine code generation.

The first two stages mostly rely on inspection of def-use chains and control-flow graphs. The others are implemented as abstract interpreters.

4.1 Access reification

The precise semantics of an expression in Hylo may depend on whether the variables involved are used later in the function. In Section 2, for example, we discussed how method and subscript bundles could be used to let the compiler select the most efficient variant of an operation. This selection process is called *access reification*. Implementing access reification as an AST analysis would be challenging because it would have to take into account all constructs and sugars of the Hylo language. Since Hylo IR directly encodes the use of bindings, implementing it as an IR analysis is simpler but requires the front-end to encode the different possibilities in the IR.

The ghost instruction `access` is designed for this purpose. When a call to a bundle is lowered to IR, its arguments are represented by `access` instructions enumerating the possible ways to access their values. For example:

```
1  public fun main() {
2    var s: UInt32 = 0
3    var t = s[16]
4    print(t)
5  }
```

The fragment of the raw IR generated by the front-end for the declaration and initialization of `t` (line 3) is:

```
1  %1 : &Bool = alloc_stack Bool
2  %2 : &Int = alloc_stack Int
3  %3 : &i64 = inline_view %2, 0, 0
4  %4 : &i64 = borrow [set] %3
5  store i64(0x10), %4
6  %6 : &Int = borrow [let] %3
7  %7 : &Int32 = access [let, inout, sink] %0
8  %8 : &Bool = project_bundle @UInt32.[](%7, %6)
9  %9 : &Bool = access [let, inout, sink] %8
10 %10: &Bool = borrow [set] %1
11 %11: &Void = alloc_stack Void
12 %12: &Void = borrow [set] %11
13 call @Bool.take_value.set(%10, %9) to %12
```

Line 1 allocates storage for `t`. Lines 2 to 6 create the subscript's argument—the constant literal 16. Line 7 creates an access to `s`, assumed to be allocated at `%0`. Because the front-end does not know how this access will be used, it emits an `access` instruction requesting the capabilities declared by to the bundle's variants. Line 8 applies one of these variants. Finally, lines 9 to 13 initialize `t`'s storage by calling `@Bool.take_value.set`, which initializes a `Bool` to a value that is then consumed. Again, since the front-end does not know exactly how the projection will be used, it emits another `access` instruction at line 9, requesting either the **`let`**, **`inout`**, or **`sink`** capability.

Note that some accesses may lack a corresponding closing instruction in the IR at this stage. These will be inserted by last use analysis (see Section 4.2.)

`access` and `project_bundle` are called *abstract access definitions*, as they create an access to a value without any specific capability. The first stage of borrow checking replaces them with more precise instructions, called *concrete access definitions*.

The compiler collects the set of capabilities actually requested an abstract access definitions. Here, for example, because `@Bool.take_value.set` takes its second argument with the **`sink`** convention, the compiler deduces that `%9` only requires the **`sink`** capability. Next, it replaces the abstract definition with a concrete one, selecting the requested strongest capability using the order **`let`** < **`set`** < **`inout`** < **`sink`**. This process is repeated until all abstract access definitions have been replaced.

Access reification always identifies a principal choice eventually since there exists a total order on capabilities. Further, it converges because abstract access definitions are monotonically eliminated from a work list until all accesses are reified or removed for having no user.

4.2 Last use analysis

The next stage in the pipeline is to identify the useful lifetime of all bindings that appear in the source program. Translated in terms of the IR, the task consists of computing the live-range of an access definition, i.e., the set of instructions over which it is *live*, and inserting instructions closing the access at the boundary of this set.

A definition is live over an instruction if it is is used by that instruction or may be used by another instruction in the future. For concision, we do not describe live-range computation and refer the reader to Brandner et al.'s extension of Appel and Palsberg's algorithm [2, 3]. For the purpose of this paper, we simply assume there exists a function $R(d)$ that returns the live-range of any instruction d.

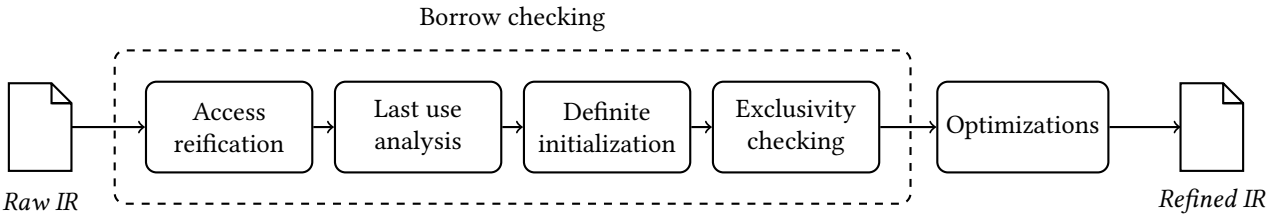

Figure 1. Architecture of the compiler back-end

A live-range r is a poset whose elements are instructions ordered by order of execution. For any pair $i_1, i_2 \in r$, $i_1 < i_2$ if and only if i_1 dominates i_2 [5]. In the fragment below, for instance, $R(\%x)$ contains the instructions at lines 3, 4, 6, and 9. The instruction at line 3 precedes that at line 6, but the instructions at lines 6 and 9 are unordered.

```
1  b0():
2    %w: &Int = project @foo()
3    %x: &Int = borrow [let] %w
4    cond_branch %1, b1, b2
5  b1():
6    %y: &Int = borrow [let] %x
7    branch b3
8  b2():
9    %z: &Int = borrow [let] %x
10   branch b3
```

Let d_1 be a definition conferring access to storage s, a definition d_2 is said to extend d_1, written $d_2 \sqsupset d_1$, if it depends on the lifetime of s. Intuitively, $d_2 \sqsupset d_1$ if d_2 invokes a subscript that depens on s or forms a **`let`** or **`inout`** binding that depends on s. For example, in the IR shown in Section 4.1, `%8`, a subscript call, extends both `%6` and `%7`, which represent the subscript's arguments.

The *extended* live-range $R^+(d)$ of a definition d is the live-range of d merged with the extended live-ranges of the definitions extending d. More formally:

$$R^+(d) = \bigcup_{\forall d' \in R(d), d' \sqsupset d} R^+(d')$$

Given these definitions, last use analysis is simply described as follows: for each access definition d, the compiler computes $r = R^+(d)$ and inserts an instruction closing the access opened by d after each maximal element of r.

4.3 Definite initialization

Definite initialization [10] is a common data-flow analysis ensuring that storage is initialized before it is read. Hylo extends this definition, admitting that storage may be partially initialized and that once initialized, storage—or part thereof—may become uninitialized. Moreover, to uphold linearity, Hylo requires that storage be *uninitialized* before it can be written. In other words, Hylo marks a clear distinction between assignment and initialization: the former modifies a value and relates to **`inout`** while the latter writes a value to storage and relates to **`set`**.

The implementation follows the usual recipe for forward data-flow analysis. Until a fixed-point is reached, the input of a basic block, merged by combining the outputs of its predecessors, is passed to a transfer function that models the effects of the block, producing a new output.

4.3.1 State representation.

The transfer function is implemented as an abstract interpreter. Its evaluation context describes a conservative assumption of the initialization state of all stack allocations and an abstract representation of the values in registers. It also stores the inputs and outputs of the analysis for each basic basic block.

Let A denote abstract memory addresses recursively defined as the minimal set such that:

- $a \in A$ if a is a basic block argument, register, or constant literal.
- $a.i \in A$ if $a \in A$ and $n \in \mathbb{N}$.

An address $a.i$ identifies the i-th part of the object stored at a. For example, if a pair `{Int, Bool}` is stored at address a, then $a.1$ identifies the storage of the Boolean part.

Let S_D denote object initialization states recursively defined as the minimal set such that:

- $\top \in S_D$, denoting a fully initialized object.
- $\bot \in S_D$, denoting a fully uninitialized object.
- $[s_1, \ldots, s_n]$ where $s_i \in S_D$, denoting an aggregate.

S_D forms a lattice whose supremum is $\top$ and infimum is $\bot$, and where the meet $\wedge$ of two elements is the conservative superposition of two initialization states. For example, $[\top, [\top, \bot]] \wedge [\top, \bot] = [\top, \bot]$. To make $\wedge$ total, we assume $[s_1, \ldots, s_n] \wedge [s_1, \ldots, s_m] = \bot$ when $n \neq m$, although this situation never occurs in practice.

The evaluation context of the interpreter is an abstract representation of the machine discussed in Section 3.3. It is defined as a 4-tuple $\langle I, C, L, M \rangle$ where:

- I is a set of basic blocks.
- C is a program counter.
- L is a table mapping registers and basic block arguments to $\mathcal{P}(A) \cup \{\top\}$, called local map.
- M is a table mapping A to S_D, called memory map.

The input of a function's entry is derived from the function's signature. The local map is initialized with an entry

for each parameter mapped onto a different abstract address. The memory map is initialized with an entry for each of these addresses mapped onto $\bot$ if the corresponding parameter is passed with **set** or $\top$ otherwise.

Merging the outputs of a basic block's predecessor is a another conservative superposition. Merging local maps ignores registers not defined in both contexts, unions abstract addresses, and forms the meet of object states. For example, if $L_1 = [\%x \mapsto \{a_1\}, \%y \mapsto \top]$ and $L_2 = [\%x \mapsto \{a_2\}]$, the resulting function becomes $[\%x \mapsto \{a_1, a_2\}]$. Merging memory maps forms the meet of all object states, assuming addresses not mentioned in both contexts refer to uninitialized storage. Program counters are not merged; I in the input of a basic block always points at its first instruction.

Perhaps more surprisingly, instruction blocks do need to be merged, as the analysis may transform the IR to insert implicit deinitialization of unused linear values and reify `move` instructions (see Section 4.3.4). A couple of properties are maintained to ensure termination: transformations cannot modify control-flow and can only cause the output to become more conservative.

4.3.2 Transfer function. The semantics of all IR instructions are transformed to operate on the abstract domain described in the previous section. Because space is limited, we only discuss **borrow** to present the general intuition.

borrow is a ghost instruction that creates an access with a specified capability on some existing storage. Its abstract semantics is given by the two following rules:

$$\frac{I[C] = \%x = \textbf{borrow}\ [k]\ \%y \qquad k \in \{\texttt{let}, \texttt{inout}\} \qquad \forall a \in L(\%y), M(a) = \top}{\langle I, C, L, M\rangle \longrightarrow \langle I, C+1, L[\%x \mapsto L(\%y)], M\rangle}$$

$$\frac{I[C] = \%x = \textbf{borrow}\ [\texttt{set}]\ \%y \qquad \forall a \in L(\%y), M(a) = \bot}{\langle I, C, L, M\rangle \longrightarrow \langle I, C+1, L[\%x \mapsto L(\%y)], M\rangle}$$

In English, `borrow` requires that its argument $\%y$ be a set of addresses, all referring to objects with the same initialization state. These objects must be initialized to borrow a **let** or **inout** capability, and deinitialized otherwise. In the conclusion of the rules, L is updated with the new access.

A failure to step denotes a violation of the flow-sensitive rules of the language, unless the interpreter can "fix" the IR by inserting implicit deinitialization (see Section 4.3.4).

4.3.3 Evaluation order. Evaluating a basic block requires knowing the state of all its uses. Further, in order to avoid overly conservative assumptions about the state of memory, a basic block should not be processed before its predecessors. These constraints create a conundrum in the presence of loops. For example, consider the following program:

```
1 public fun main() {
2   var s: Array<Int>
3   &s = [1, 1]
4   &s = [2, 2]
5   while Bool.random() { &s = [3, 3] }
6   print(s)
7 }
```

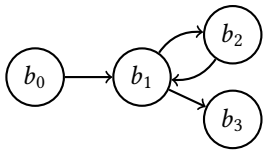

Figure 2. Control-flow graph of a simple loop

The control flow graph of this function is shown in Figure 2. `b0` is the function's entry. `b1` is the head of the loop, which evaluates its termination condition. `b2` is the loop body, and `b3` is the code following the loop.

Clearly, b_0 must be visited first and its output can be used to setup the input of b_1. Since definitions must dominate their uses—per the rules of SSA—b_2 cannot contain definitions used in b_1. However, b_2 may modify the state of memory allocated in b_0. Hence, in the absence of any knowledge about `b2`'s output, a conservative computation of b_1's input would consider `s`'s storage to be uninitialized.

To address this problem, we state that a basic block b can be visited if and only if all its predecessors have been visited *or* are dominated by b. Following this rule, we can now process b_1 after b_2 and are then free to process either b_2 or b_3.

4.3.4 Definite deinitialization. Live objects must be automatically deinitialized after their useful lifetime, as mentioned in Section 2. The interpreter ensures that storage is uninitialized before new **set** accesses are formed or stack memory is deallocated, inserting deinitialization—i.e., a call to the object's deinitializer—as necessary.

The picture is slightly more complicated for the IR form of assignment statements. In the surface language, `&x = y` can represent either assignment or initialization, depending `x`'s initialization state. Because the front-end cannot determine initialization states in general, it emits a `move` instruction representing both operations. For example, lines 2 to 4 in the example from the previous section are lowered to the following raw IR fragment, simplified to its essentials:

```
1 %1: &Array<Int> = alloc_stack Array<Int>
2 %2: &Array<Int> = alloc_stack Array<Int>
3 // initialization of %2 to '[1, 1]'
4 %4: Array<Int> = consume %2
5 move %4 to %1
6 %6: &Array<Int> = alloc_stack Array<Int>
7 // initialization of %6 to '[2, 2]'
8 %8: Array<Int> = consume %6
9 move %8 to %1
```

`%1` is the storage allocated for `s`. Lines 2 to 4 represent the evaluation of `[1, 1]`, about to be assigned to `s`. The `move` instruction that follows represents either assignment or initialization. As b_1 is processed, the interpreter will conclude that `s`'s storage is not initialized at this point and replace this `move` instruction with a call to `Array`'s move-initialization. The next sequence of instructions is similar. This time, however, the interpreter will conclude that `s`'s storage *is* initialized and replace the `move` at line 9 with a call to `Array`'s move-assignment.

Unfortunately, this trick is not enough to cover all ambiguous cases of assignment vs initialization. Imagine that lines 3 and 4 are removed from the original program, leaving `s` uninitialized before the loop. In that case, abstract interpretation will assume that b_2 initializes `s`'s storage and apply move-initialization. However, that is only valid for the first iteration of the loop!

One fix would be to track the initialization state of `s` at run-time to apply either assignment or initialization. That, however, would introduce a test in a seemingly branchless fragment, which would not align with Hylo's zero-cost abstraction philosophy. Another option would be to unroll the first iteration of the loop. But such a modification would invalidate control-flow information, making it harder to reason about the state of the abstract interpreter. Hylo opts for a third approach: we insert implicit deinitialization in b_2 to force its output to be consistent with that of b_1's other predecessor. As a result, `s` is definitely uninitialized at b_1's entry and move-initialization can apply properly.

An unfortunate consequence of this solution is that move-assignment can never occur on a variable in the body of a loop unless this variable is definitely initialized before the first iteration. In our particular example, it confines the lifetime of the array's out-of-line storage to the body of a loop, causing a new heap allocation at each iteration.

4.4 Exclusivity checking

The last stage of borrow checking relates to uniqueness [11]. Its purpose is to ensure that live mutating bindings may not overlap with other live and accessible bindings.

Just like definite initialization (see Section 4.3), exclusivity checking is realized as a forward data-flow analysis using an abstract interpreter as the transfer function. While abstract domains obviously differ, most other implementation details are actually shared between the two analyses.

4.4.1 State representation. Let S_E denote object ownership states recursively defined as the minimal set such that:

- $\square \in S_E$, denoting a unique object.
- $\triangle\{i_1, \ldots, i_n\} \in S_E$, denoting a borrowed object where i_i is an instruction or the token $\flat$, which represents an unknown borrower.
- $[s_1, \ldots, s_n]$ where $s_i \in S_E$, denoting an aggregate.

S_E forms a lattice whose supremum is $\square$ and infimum is $\triangle B$ where B is the set of all instructions with the unknown borrower $\flat$. Again, the meet $\wedge$ of two elements is the conservative superposition of two states.

The input of a function's entry is derived from the function's signature. The local map is initialized with an entry for each parameter mapped onto a different abstract address. The memory map is initialized with an entry for each of these addresses mapped onto $\triangle\{\flat\}$ if the corresponding parameter is passed with **`let`** or onto $\square$ otherwise.

4.4.2 Transfer function. Again, we focus on `borrow`, as it is one of the most interesting instructions. In the context of exclusivity checking, the abstract semantics of an **`inout`** access creation is given by the following (simplified) rule[4]:

$$\frac{\begin{array}{c} I[C] = \%x = \textbf{borrow}\ [\texttt{inout}]\ \%y \\ \forall a \in L(\%y), M(a) = \square \end{array}}{\begin{array}{c} \langle I, C, L, M\rangle \longrightarrow \\ \langle I, C+1, L[\%x \mapsto L(\%y)], M[a \mapsto \triangle\{C\} \mid a \in L(\%y)]\rangle \end{array}}$$

In English, `borrow` requires that its argument $\%y$ be a set of addresses, all identifying objects that are unique. In the conclusion of the rule, the local map is updated with the newly created access and the memory map is updated to reflect the loans.

5 Conclusion

We have presented the implementation of a borrow checker for Hylo, a programming language that adopts mutable value semantics to offer memory safety without loss of efficiency. Our approach leverages abstract interpretation to track object ownership, guarantee lifetime safety, and enforce uniqueness of mutable accesses.

Unlike techniques based on equational reasoning, abstract interpretation lets data-flow analyses be expressed in terms of the operational semantics of the program's intermediate representation. Specifically, the semantics of the interpreters carrying out definite initialization (Section 4.3) and exclusivity checking (Section 4.4) resemble traditional formal descriptions of small-step operational semantics. This approach has the theoretical advantage that analyses can be developed in lock-step with the formalization of the type system, and the engineering advantage that the behavior of the borrow checker is straightforward to debug.

Although we have not yet spent significant effort on diagnostic quality, we believe our approach should be particularly well-suited to generating informative error messages. A violation of the borrowing rules translates directly to a failure to take a step, at which point the evaluation context of the interpreter provides a perfect snapshot of the information

[4]We assume that reborrowing is not allowed for simplicity. The rule actually implemented checks whether the borrow is made on the currently live borrower of the access.

known to the borrow checker. In contrast, systems based on equational reasoning typically require this information to be recovered in some way.

We have not conducted serious benchmarks on the efficiency of our implementation, nor have we spent any time optimizing our compiler. Nonetheless, anecdotal evidence currently points at promising results as compilation times of small programs (~1000 lines of code) are on par with Rust.

Next steps on our roadmap include the continued development of Hylo's compiler, in particular to support existential types [15], and formalizing Hylo's type system. Our current implementation is open source and available at https://github.com/Hylo-lang/Hylo.

References

[1] Dave Abrahams, Sean Parent, Dimi Racordon, and David Sankel. 2022. The Val Object Model. *ISO C++ Committee: WG21* P2676R0 (2022).

[2] Andrew W. Appel and Jens Palsberg. 2002. *Modern Compiler Implementation in Java, 2nd edition.* Cambridge University Press.

[3] Florian Brandner, Benoit Boissinot, Alain Darte, Benoît Dupont de Dinechin, and Fabrice Rastello. 2011. *Computing Liveness Sets for SSA-Form Programs.* Technical Report. INRIA.

[4] CWE [n. d.]. 2022 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html. Accessed: 2023-07-04.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems* 13, 4 (1991), 451–490. https://doi.org/10.1145/115372.115320

[6] EO 2021. Improving the Nation's Cybersecurity, U.S. Executive Order 14028.

[7] EU 2022. European Proposal for a Regulation on cybersecurity requirements for products with digital elements - Cyber resilience Act.

[8] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. 2016. The spirit of ghost code. *Formal Methods in System Design* 48, 3 (2016), 152–174. https://doi.org/10.1007/s10703-016-0243-x

[9] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2005. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, Jens Palsberg and Martín Abadi (Eds.). ACM, 233–246. https://doi.org/10.1145/1040305.1040325

[10] Nicu G. Fruja. 2004. The Correctness of the Definite Assignment Analysis in C#. *Journal of Object Technoly* 3, 9 (2004), 29–52. https://doi.org/10.5381/jot.2004.3.9.a2

[11] Philipp Haller and Martin Odersky. 2010. Capabilities for Uniqueness and Borrowing. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6183)*, Theo D'Hondt (Ed.). Springer, 354–378. https://doi.org/10.1007/978-3-642-14107-2_17

[12] John Hunt. 2013. *Proxy.* Springer, Cham, 201–205. https://doi.org/10.1007/978-3-319-02192-8_25

[13] Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language.* No Starch Press. https://books.google.ch/books?id=lrgrDwAAQBAJ

[14] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO.* IEEE Computer Society, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[15] Konstantin Läufer. 1996. Type Classes with Existential Types. *J. Funct. Program.* 6, 3 (1996), 485–517. https://doi.org/10.1017/S0956796800001817

[16] Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. 2012. A type system for borrowing permissions. In *Symposium on Principles of Programming Languages, POPL*. 557–570. https://doi.org/10.1145/2103656.2103722

[17] Polonius [n. d.]. Polonius. https://rust-lang.github.io/polonius/. Accessed: 2023-07-04.

[18] Dimi Racordon, Denys Shabalin, Daniel Zheng, Dave Abrahams, and Brennan Saeta. 2022. Implementation Strategies for Mutable Value Semantics. *Journal of Object Technology* 21, 2 (2022), 2:1–11. https://doi.org/10.5381/jot.2022.21.2.a2

[19] Jesse A. Tov and Riccardo Pucella. 2011. Practical affine types. In *Symposium on Principles of Programming Languages, POPL 2011.* 447–458. https://doi.org/10.1145/1926385.1926436

[20] Philip Wadler. 1990. Linear Types can Change the World!. In *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*, Manfred Broy and Cliff B. Jones (Eds.). North-Holland, 561.