Implementing a Language with Explicit Assignment Semantics

Dimitri Racordon
University of Geneva
Centre Universitaire d'Informatique
Switzerland
dimitri.racordon@unige.ch

Didier Buchs
University of Geneva
Centre Universitaire d'Informatique
Switzerland
didier.buchs@unige.ch

Abstract

Anzen is a multi-paradigm programming language that aims to provide explicit and controllable assignment semantics. It is based on the observation that abstractions over memory management and data representation, as commonly adopted by contemporary programming languages, often transpire relics of the underlying memory model and lead to confusing assignment semantics in the presence of aliases. In response, Anzen's goal is to offer a modern approach to programming, built on a sound and unambiguous semantics.

This paper describes the implementation of a compiler for Anzen. Our implementation transpiles sources to an intermediate language inspired by the LLVM IR, designed to ease further analysis on Anzen's statements. This intermediate representation is then consumed by a register-based virtual machine. We present the Anzen compiler's architecture, introduce its intermediate language and describe the latter's evaluation. Our work aims to set a reference implementation for future developments and extensions of the language.

CCS Concepts • **Software and its engineering → Interpreters**; **Runtime environments**; *Imperative languages*; *Data types and structures*.

Keywords intermediate language, virtual machine, imperative languages, assignment, memory management

ACM Reference Format:
Dimitri Racordon and Didier Buchs. 2019. Implementing a Language with Explicit Assignment Semantics. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '19), October 22, 2019, Athens, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3358504.3361227

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
VMIL '19, October 22, 2019, Athens, Greece
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6987-9/19/10.
https://doi.org/10.1145/3358504.3361227

1 Introduction

Programming languages have become an ubiquitous tool, applied in a vast array of industrial and academic domains. In order to improve usability needs in these numerous use cases, modern programming languages typically sit at a very high abstraction level, so as to mask the intricacies of their computational model. Although the notions on which programming languages abstraction can vary widely, it is now common to hide memory management and data representation. If the benefits of such a practice are undeniable, relics of the underlying memory model still transpire in most programming languages, leading to confusing assignment semantics.

Based on this observation, we developed Anzen [17], a multi-paradigm programming language that aims to dispel such a confusion. Rather than overloading a single assignment operator with different meanings, Anzen provides three distinct assignment primitives with unequivocal semantics. One creates aliases, another performs deep copies and the last deals with uniqueness. The language further supports high-level concepts, such as generic types and higher-order functions, and offers capability-based [3] memory and aliasing control mechanisms that are enforced by its runtime.

This paper describes the implementation of a compiler for Anzen. Its architecture is schematized in Figure 1. We identify two main components. The *front-end* consumes Anzen sources with a recursive descent parser, directly producing an abstract syntax tree (AST). Information about the concrete syntax (e.g. line and column numbers) is kept in the form of metadata, and used to provide context to the error messages. The AST then undergoes semantic analysis, which essentially handles name binding and type inference. The *back-end* transforms type checked ASTs into an intermediate language, called Anzen Intermediate Representation (AIR), which is this paper's main contribution. AIR is a low-level assembly-like instruction set reminiscent to LLVM IR [12], but that preserves Anzen's assignment operators, together with a handful of high-level concepts, such as higher-order functions. AIR exposes how references (a.k.a. pointers) are manipulated by every instruction explicitly. Consequently, it is easier to interpret than its high-level counterpart, and more suitable to perform language-specific analysis and code optimization. It is consumed by a register-based virtual machine [9], effectively interpreting Anzen programs.

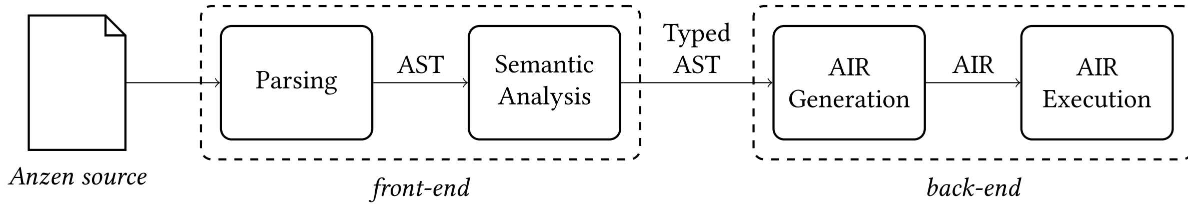

Figure 1. Architecture of the Anzen compiler.

2 Anzen in a Nutshell

This section briefly introduces Anzen, a multi-paradigm programming language, leaning towards imperative programming and object orientation. For spatial reasons, we leave out non-essential aspects and focus on core elements most closely impacting the language's interpretation. A more comprehensive presentation of Anzen, as well as a formal description of its semantics can be found here [17].

Data Types On the top of a handful of primitive types denoting numbers and boolean values, Anzen also supports aggregates of objects, called structures. Borrowing from object-oriented programming languages, methods, constructors and destructors can be declared along with a structure to compartmentalize behavior. Inside these functions, a reserved `self` reference allows access to the instance's internals.

Functions are first-class citizens in Anzen, and are therefore allowed to be assigned and passed as arguments to and returned from other functions. They may also capture identifiers from their declaration context, effectively forming function closures and thereby enabling partial application.

Assignment Semantics Anzen's most distinguishing characteristic lies in the use of three different assignment operators, each representing a particular assignment semantics:

- An aliasing operator `&-` assigns an alias on the object on its right to the reference on its left. Its semantics is the closest to what is generally understood as an assignment in languages that abstract over pointers, such as Python and Java.
- A copy operator `:=` assigns a *deep* copy of the object's value on its right to the reference on its left. If the left operand was already bound to an object, the copy operator mutates its value rather than reassigning the reference to a different one.
- A move operator `<-` moves the object on its right to the reference on its left. If the right operand was a reference, the move operator removes its binding, effectively leaving it unusable until it is reassigned. This corresponds to the affine assignment semantics [20] found in languages such as Rust and C++.

Figure 2 illustrates the three assignment semantics. Notice that we use the term "memory" rather than "stack" or "heap".

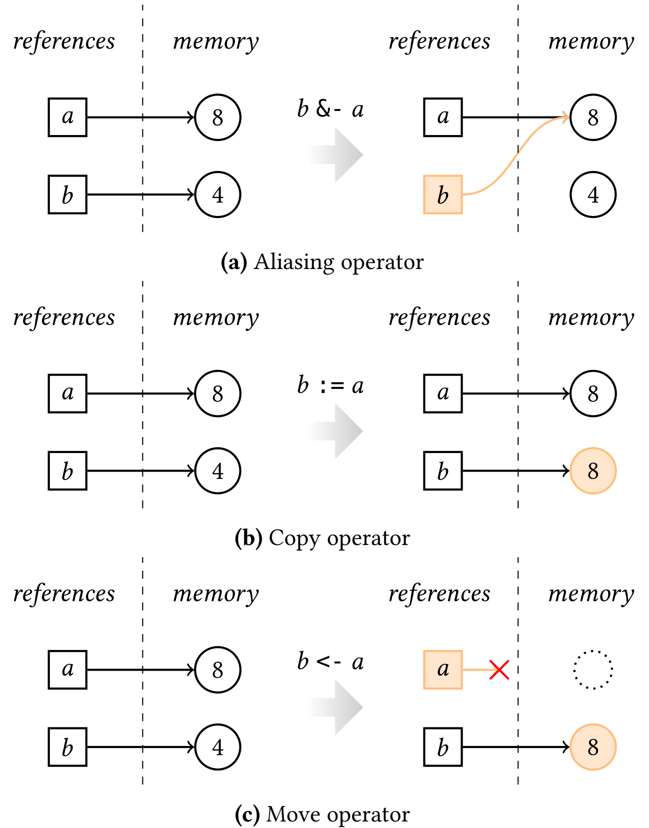

Figure 2. Effect of Anzen's assignment operators. Each illustration depicts the situations before and after a particular assignment, starting from a state where two variables a and b are bound to unrelated memory locations, holding the values 8 and 4 respectively. Changes are highlighted in color.

The reason is that we aim at making a clear distinction between the semantics of Anzen's operators and the actual underlying memory model. Whether a reference is a pointer to heap-allocated memory or a primitive value allocated on the stack should be irrelevant for the developer. That way she may focus solely on the semantics of her program, at an abstraction level that does not bother about the specifics of compilation, optimization and/or interpretation.

These three operators also serve to specify the parameter and return value policy in function calls. The use of an aliasing operator corresponds to a pass-by-alias, while the use of

a copy operator corresponds to a pass-by-value policy [15]. The move operator also treats objects as linear resources, and therefore mimics Rust's move semantics.

Parameter passing

```
1  fun f(list: @mut List<Int>) {
2    list.append(new_element <- 5)
3    print(line <- list)
4  }
5
6  var a: @mut <- List<Int>()
7  a.append(new_element <- 4)
8
9  // `a` is passed by value, therefore the
10 // mutation of the `list` parameter in
11 // `f`'s body have no side effects.
12 f(list := a)
13 // Prints "[4, 5]"
14
15 print(line <- a)
16 // Prints "[4]"
```

Aliasing Control Mechanisms In order to support the affine semantics of its move operator, Anzen has to keep track of reference uniqueness [8]. A reference is unique if there are no other references (i.e. aliases) on the object to which it is bound. In this case, it cannot appear as the right operand of a move assignment. Uniqueness is treated as a fractionable capability, reminiscent to Boyland's proposal [2]. The intuition is that it is initially obtained in full at allocation, but gets fractioned every time the reference or one of its alias appears as the right operand of an aliasing assignment. The loss of the uniqueness capability is however only temporary, as a reference can gather all fragments back once its aliases either go our of scope or are reassigned to another reference.

Uniqueness

```
1  var a, b, c: @mut Int
2  // `a`, `b` and `c` start unallocated
3  a := 10
4  b <- 20
5  // `a` and `b` get unique
6  c &- a
7  // `a` gets shared, `c` gets borrowed
8  c &- b
9  // `b` gets shared, `a` gets unique
10 b <- a
11 // `a` gets moved
```

On the top of uniqueness, Anzen also supports reference and object immutability. The former relates to a restriction on references. Simply put, variables declared with the keyword `var` can be reassigned (i.e. appear as the left the operand of an aliasing assignment), while those declared with the keyword `let` cannot. The latter relates to a restriction on values. Immutable objects cannot be *modified* (i.e. appear as the left operand of a copy or move assignment). Furthermore, object immutability is treated transitively, meaning that not only the fields of an immutable object cannot be reassigned, but their respective values are also considered immutable. The type qualifier `@cst` describes immutable values whereas `@mut` describes mutating ones. References to immutable (resp. mutable) objects are said *non-mutating* (resp. *mutating*).

Immutability

```
1  struct Point {
2    var x: @mut Int
3    var y: @mut Int
4  }
5
6  let p: @cst = Point(x <- 42, y <- 1337)
7  p <- Point(x <- 0, y <- 0)
8  // illegal since `p` is immutable
9  p.x <- 12
10 // illegal since `p.x` is immutable, by
11 // transitivity
12 p &- Point(x <- 3, y <- 14)
13 // illegal since `p` is not reassignable
```

Like uniqueness, object mutability is treated as a capability which can be fractioned by aliases. Non-mutating references may borrow an alias to an object referred to by a mutating reference, in which case the latter becomes non-mutating for the duration of the loan. The type system guarantees that mutating references cannot coexist with non-mutating references on the same object. However, unlike in most systems, there is no restriction on the number of mutating references. This allows mutable self-referential data structures to be expressed naturally, whereas such an exercise can prove challenging in more constrained type systems [13].

3 Semantic Analysis

The compiler's semantic analysis phase consists in the discovery of all expressions' types and the verification of reference immutability. It is performed directly on the AST.

3.1 Checking Reference Immutability

Reference immutability is checked with a variant of definite assignment analysis [7]. Following a particular non-reassignable variable declaration, the compiler scans all assignment where it appears as a left operand and verifies that it is guaranteed to be definitely unassigned (i.e. unassigned no matter what execution paths are followed). The analysis further guarantees that all members of an instance be definitely assigned at the end of its constructor. Therefore, enforcing their non-reassignability outside of constructors consists in checking that they never appear as left operand of an aliasing operator. The advantage of this approach is that

it does not require any additional bookkeeping for instance members to be passed along across function boundaries.

3.2 Extracting and Solving Type Constraints

Anzen does not require all declarations to be explicitly typed, as it is required in C for instance. Consequently, the compiler has to deduce omitted type annotations from the expressions themselves. Fortunately, Anzen's type system fits the traditional Hindley-Milner system for which numerous implementations have already been proposed (e.g. [10, 19]). Following the observations of Pierce and Turner [16], we only aim to infer types for local bindings, as well as for overloaded and generic symbols, while expecting functions' parameters to be explicitly typed. Therefore, our goal is not to perform the *complete* type inference of any arbitrary program. The compiler implements type inference by solving a large constraint system. In a first stage, all expressions are first associated with a unique type variable, before the the AST is scanned to extract explicit type annotations and type constraints. These constraints directly relate to the language's semantics, and are categorized into five groups:

- **Equality constraints** indicate that two type variables must be equal.
- **Conformance constraints** denote a relaxed notion of equality, that prescribes that one type be *compatible* with another.
- **Construction constraints** require that one type be the signature of a constructor for the other.
- Let τ and σ be two types and m a identifier, a **membership constraint** requires that τ be the type of a structure with a member m whose type is equal to σ.
- **Disjunction constraints** represent choices between different ways to type an expression.

Once built, the constraint system is solved to find type bindings for each type variable that has been introduced. This is achieved by breaking non-trivial constraints into groups of smaller ones, until only equality constraints remain. These simply correspond to unification, and are solved by either assigning type variables to their inferred type, or by ensuring that existing bindings match. A handful of heuristics are applied to select which constraints should be solved first, hopefully reducing the size of the search space by eliminating unsatisfiable branches as early as possible.

Polymorphic Types Constraints involving polymorphic (a.k.a. generic) types are rewritten by substituting each polymorphic parameter with a fresh variable. For instance, a polymorphic type of the form $\forall a, a \times a \to U$ is substituted with a monomorphic type of the form $\tau \times \tau \to U$, where τ is a fresh variable. If the remaining constraints provide enough context, these fresh variables are then unified with concrete types, effectively discovering how polymorphic types should be specialized. This approach is sound, because Anzen deliberately does not feature first-class polymorphic values [11].

Conformance Conformance constraints cannot be trivially solved if either of the types it involves is unknown (i.e. is an unbound type variable). However, information already inferred can be leveraged to refine a particular constraint. Assuming the type conformance relation forms a lattice, solving conformance constraints is equivalent to computing either the join or the meet of two types. In both cases the strategy is to form a disjunction of equality constraints for each known sub-type or super-type, respectively.

Construction and Membership Construction and membership constraints are rewritten as simple equality constraints, once the type owning the member or constructor involved has been inferred.

Disjunctions Disjunction constraints represent choices, such as the different overloads of a function. When the solver encounters a disjunction, it spawns as many sub-solvers as there are choices, which will all attempt to find a solution. Those that fail are discarded immediately, whereas others' solutions are collected and compared to choose the most specific one, with respect to the number of generic specializations and type coercions it involves. The compiler reports an ambiguous situation if it cannot decide between multiple solutions based on these criteria.

3.3 Reporting Type Errors

One challenge of type inference is to generate accurate error reports in case of failure. The complexity of the task stems from the difficulty to identify precisely the reason why a particular constraint might be unsatisfiable, in order to provide the user with relevant feedback. Moreover, while type inference is confluent for well-typed programs, the order in which constraints are solved may lead to different results in the presence of errors. Fortunately, by splitting the extraction of the constraint system and their solving into two separate steps, our compiler can avoid the left-to-right bias [19] and find the minimal subset of unsatisfiable constraints. Furthermore, each constraint is associated with a log that keeps a record of the reasons that brought it into existence. As a result, this information can be leveraged to trace back an unsatisfiable constraint to the exact point in the AST from which it originates. A generalization of this approach is presented in [21].

4 Anzen Intermediate Representation

AIR is a low-level instruction set similar to assembly code which preserves Anzen's assignment operators, together with a handful of high-level concepts such as structures and higher-order functions. Although the language is not supposed to be used for writing programs directly, it has a concrete syntax that can facilitate debugging. We use it to illustrate various examples in the remainder of this section.

4.1 Anatomy of an AIR unit

The Anzen compiler treats each source file as single compilation unit which is individually compiled and translated to AIR. Those units are linked together to form a single executable AIR program, effectively bringing the support for separate compilation. A unit is a collection of functions, that are themselves split into multiple blocks, each associated with a unique label. These are used to guide control flow. A function necessarily contains at least two blocks, labeled `entry#0` and `exit#0`, that denote its entry and exit points, respectively. Naturally, `entry#0` is always the first block and `exit#0` the last one. All instructions of a block are executed in order until a *terminator* instruction is reached. These are instructions that modify control flow (e.g. jumps and function returns). All blocks are assumed to finish with a terminator instruction, not followed by any other statement. Figure 3 depicts the anatomy of a typical AIR unit.

4.2 AIR Instructions

There are two kinds of instructions in AIR. *Non-assignable* instructions only consume references to update the program's state, but do not produce any result. On the contrary, *assignable* instructions yield a reference that has to be stored, and can later appear as an operand to another instruction.

The most significant difference between AIR and Anzen is that an expression cannot contain other sub-expressions. Instead, it should be decomposed into simpler instructions that store intermediate results into temporary registers. These registers are Static Single Assignment (SSA) variables [4] (i.e. variables that are assigned exactly once) which store references to actual values. In the words of the C language, a register is a pointer to a pointer to a value. The advantage of this approach is that an SSA register in AIR uniquely designates a single reference, explicitly exposing how it is manipulated. Consider for instance the following Anzen program, and its corresponding AIR code:

Anzen
```
1 let sum: @mut <- 19 + 22
2 sum <- sum + 1
```

AIR
```
1 %1 = make_ref Int
2 %2 = apply $__iadd, 19, 22
3 move %2, %1
4 %3 = apply $__iadd, %1, 1
5 move %3, %1
```

Lines 1 to 3 represent the declaration of the variable `sum`. At line 1, the instruction `make_ref` creates a new reference and assigns it to the register `%1`. Line 2 computes the addition of 19 and 22, and assigns the result to the register `%2`. Line 3 moves the result of this addition to the register representing the variable `sum`. Although the instruction corresponds to an assignment, notice that it does not reassign its target register `%1` (which would violate the SSA property). Instead, it *updates* the value of the reference it holds. Finally, line 3 and 4 represent the increment of the variable `sum`. Again, the expression in Anzen is decomposed so that the intermediate result is stored in a temporary register.

4.3 Assignments and Function Calls

AIR features one instruction for each assignment operator in Anzen. Note that their operands appear in reverse order. For instance "`move 42, %1`" reads as "move the value 42 into the reference held by the register `%1`".

Anzen reuses its assignment operators to specify parameter and return value policies in function calls. This principle is highlighted explicitly in AIR. Before a user function[1] is called, a reference is created for each of its parameters, and assigned to its respective argument with the appropriate semantics. Return values are treated similarly, using the reserved register `%0` as the target of assignments corresponding to return statements. Consider for instance the following Anzen function, and its corresponding AIR counterpart:

Anzen
```
1 fun factorial(x: Int) -> Int {
2   if x <= 1 {
3     return <- 1
4   } else {
5     return <- x * factorial(x <- x - 1)
6   }
7 }
```

AIR
```
1  fun $exfactorial_Fofi2i : Int -> Int {
2  entry#0:
3    %0 = make_ref Int
4    %2 = apply $__ile, %1, 1
5    branch %2, then#0, else#0
6  then#0:
7    move 1, %0
8    jump exit#0
9  else#0:
10   %3 = make_ref Int
11   %4 = apply $__isub, %1, 1
12   move %4, %3
13   %5 = apply $ex_factorial_Fofi2i, %3
14   %6 = apply $__imul, %1, %5
15   move %6, %0
16   jump exit#0
17 exit#0:
18   ret %0
19 }
```

[1] Built-in functions (e.g. the integer addition) refer directly to their operands, thus not requiring any explicit parameter assignments.

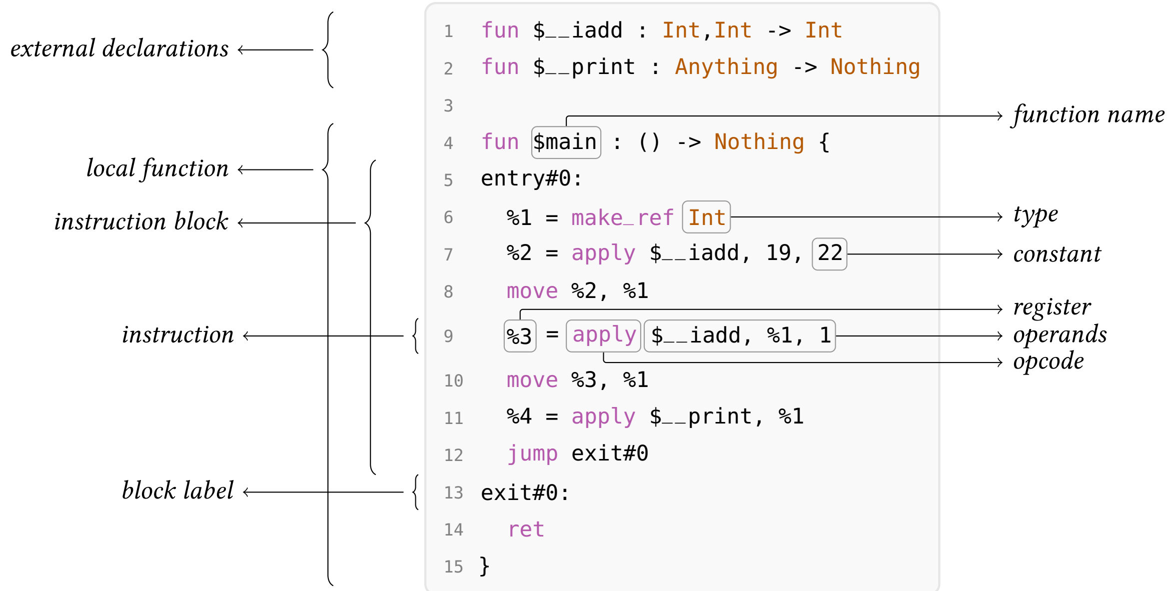

Figure 3. Anatomy of an AIR unit.

The reference creation at line 3 corresponds to the function's return reference, that is therefore assigned to the reserved return register `%0`. Within a function, parameters are also represented by registers, which by convention are numbered starting from `%1`. Thus, line 4 tests whether the function's argument is inferior or equal to one. If the test succeeds, the execution jumps to the block labeled `then#0`, assigns `1` to the return register and jumps to the exit block. If the tests fails, then the execution jumps to the block labeled `else#0` to recursively call the function. Line 10 creates a reference for its parameter, which gets assigned by move at line 12 with the argument `x`, subtracted by one. The recursive call occurs at line 13, and its result is multiplied by the argument `x`. Finally, line 15 assigns the result to the return register.

While Anzen supports function overloading and generic types, all functions in AIR are monomorphic. Hence, the name of each function in Anzen is mangled in AIR to guarantee its uniqueness across a single compilation unit. The mangling consists in encoding the scope in which the function is declared, as well as its signature. For instance in the above example, the name of the Anzen function `factorial` is mangled as `exfactorial_F0fi2i`, where `ex` denotes the module in which the function's defined, and `F0fi2i` denotes its signature. Type names are mangled similarly.

Two techniques are used to translate generic functions into AIR. The first consists of generating a monomorphized version of a generic function every time a particular specialization is needed, similar to how code for C++ templates is generated. In other words, each reference to a generic function instructs the compiler to *clone* its body, substituting generic types for specialized ones. One drawback of this approach is that it may produce very large AIR units in the presence of highly generic code. Furthermore and more importantly, monomorphization requires that the body of a called function be known during compilation, thus hindering the separate processing of each unit. To solve these issues, a second technique is to replace genericity with polymorphism, relying on boxing and virtual tables [6, Chapter 3] for functions and method dispatching (as described in Section 5). Although monomorphization is preferred for performance reasons [5], it can only occur under two conditions. First, the generic function must be declared in the same compilation unit, so that the compiler can have access to its body during AIR generation. Second, the monomorphization of a function must not lead to another monomorphization of itself, with a different type. The second constraint guarantees that there is a finite amount of code duplication.

4.4 Higher-Order Functions

As mentioned in Section 2, Anzen is a higher-order programming language. In AIR, functions names can appear as operands to other instructions, thereby making them first-class citizens. This approach is reminiscent to how function pointers can be used in C. However, it does not handle function environments, which are necessary to support function closures. To solve this problem, AIR features a particular instruction `partial_apply`, that computes the partial application of a function. As a result, a function closure can be represented as the partial application of a corresponding first-order function that accepts captured references as parameters. This process is commonly referred to as *closure conversion* or *defunctionalization* [18]. The resulting reference denotes a function that accepts the remaining parameters, and can be used as the first operand of an `apply` or

another `partial_apply` instruction, just as any other function reference. Consider for instance the following Anzen program, together with its AIR representation:

```
Anzen
1  fun inc(by x: Int) -> (n: Int) -> Int {
2    return <- fun(n: Int) -> Int {
3      return <- x + n
4    }
5  }
```

```
AIR
1  fun $exinc___L_Fni2i: Int,Int -> Int {
2  entry#0:
3    %0 = make_ref Int
4    %3 = apply $__iadd %1, %2
5    move %3, %0
6    jump exit#0
7  exit#0:
8    ret %0
9  }
10
11 fun $exinc_Fxi2Fni2i: Int -> Int -> Int {
12 entry#0:
13   %0 = make_ref Int -> Int
14   %2 = make_ref Int
15   bind %1, %2
16   %3 = partial_apply $exinc___L_Fni2i, %2
17   move %3, %0
18   jump exit#0
19 exit#0:
20   ret %0
21 }
```

Two AIR functions are generated. The first one corresponds to the anonymous function declared at line 2 of the Anzen program. Notice that it accepts two arguments, the first of which corresponding to the identifier `x` that it captures in its closure. The second one corresponds to the `inc` function in the Anzen program. Line 13 corresponds to the creation of a reference for the captured identifier, stored in `%2`. Note that captures are always performed by alias, which justifies the aliasing assignment at line 14. The reference in `%2` is used to partially apply `$exinc___L_Fni2i`, resulting in the creation of a function closure which is assigned by move to the return register at line 16.

4.5 Structure Instances and Methods

Structure instances are allocated with the `alloc` instruction which essentially creates an aggregate of references for each member of the instance. Constructors, destructors and other methods are not stored with the instance. Instead, they are defined as separate AIR functions that accept an additional parameter corresponding to `self`. Field names are forgotten in AIR. Consequently, members must be referred by an index denoting their position in the aggregate. By convention, members are indexed in the same order as they appear in Anzen's type definition. Access to particular instance member is achieved with the `extract` instruction. As methods are first-class citizen as well, assigning one to a variable has to create a partial application of its underlying AIR counterpart, in order to bind the `self` parameter. Consider for instance the following program, together with a part of its AIR representation. For the sake of legibility, function names are not mangled as they would be in an actual AIR unit:

```
Anzen
1  struct Counter {
2    let value
3    new() {
4      self.value <- 0
5    }
6    mutating fun next() -> Int {
7      self.value <- self.value + 1
8      return := self.value
9    }
10 }
11
12 let counter <- Counter()
13 let next &- counter.next
```

```
AIR
1  fun $Counter_next : (Counter) -> Int {
2  entry#0:
3    %0 = make_ref Int
4    %2 = extract %1, 0
5    %3 = apply $__iadd, %2, 1
6    move %3, %2
7    %4 = extract %1, 0
8    copy %4, %0
9    jump exit#0
10 exit#0:
11   ret %0
12 }
13
14 fun $main : () -> Nothing {
15 entry#0:
16   %1 = make_ref Counter
17   %2 = apply Counter_new
18   move %2, %1
19   %3 = make_ref () -> Int
20   %4 = partial_apply Counter_next, %1
21   // ...
22 }
```

The first function implements the `Counter.next` method. The `value` member is extracted at line 4, and incremented by one over the next two lines. It is finally copied at line 8

to the return register. The second function is a snippet of the main function. `Counter`'s constructor is applied at line 17, producing a new instance in register `%2`. It is reused as an operand to the partial application at line 20, producing a closure that binds the method to this particular instance. Consequently, this results in a function `() -> Int` that applies `Counter.next` for a particular instance of `self`.

5 AIR Interpreter

The Anzen compiler comprises an interpreter to run AIR programs, effectively providing a runtime system for Anzen's execution model. It is implemented as a register-based virtual machine [9]. While such interpreters are usually more complex than their stack-based counterparts, they can describe memory manipulations more faithfully. Our interpreter consists of an instruction pointer that keeps track of the next instruction to execute, a call stack that stores pointers to heap-allocated reference objects, and a frame pointer that is used to compute the runtime address of a particular AIR register. The stack is logically split into multiple *call frames*, each of them representing the execution context of a particular function. A call frame comprises all the locally available registers of the corresponding function, which are referred to as its *locals*, as well as a *return instruction pointer*, which indicates the next instruction to execute once the function returns. Each function application (i.e. the execution of an `apply` instruction) adds a new call frame onto the stack, and modifies the interpreter's frame pointer so that it always points to the return register (i.e. `%0`) of the current call frame. Consequently, the runtime address of a given AIR register can be computed as an offset to the frame pointer. Figure 4 depicts the anatomy of a call stack. The particular example that is illustrated is detailed later in this section.

5.1 Memory Model

Reference objects are four-words data structures that wrap a pointer to some heap-allocated memory, representing an actual payload (e.g. a number), together with a three-word value describing its current capabilities. These are used to enforce the aliasing control mechanisms discussed in Section 2. Note that the interpreter uses reference objects for all local values, including primitive ones. This is a departure from most virtual machines' implementations, which generally store simple values directly onto the stack. This strategy usually yields better performances, as it can avoid the cost of pointer indirection for data that fits into the stack. However, we take a different approach to simplify the implementation of Anzen's assignment operators, as explained later.

Values are boxed into containers, so as to represent them uniformly in memory. This allows to support the type erasure applied on generic functions that could not be monomorphized during AIR code generation (see Section 4.3). Such containers are defined as three-words structures. The first two words are used to store either a simple data (e.g. a number), or a pointer to a more complex value stored in heap-allocated memory (e.g. a large structure instance). The third word stores a pointer to its virtual table. The latter lets the interpreter retrieve the implementation of the functions related to a particular value type, in order to perform dynamic dispatch. In addition to the functions corresponding to user methods, all virtual tables define routines that allow copying and destroying values. These are defined internally for all data types, and are part of the runtime library.

5.2 Interpreting Assignments

Most of the complexity surrounding AIR's interpretation relates to the handling of assignments. Fortunately, the two-level indirection of our interpreter's memory model greatly eases the task. Aliasing assignments consist in copying the value of the pointer stored in its source's reference object. Copy assignments are carried out by reassigning copy of their source's value to their target's value, obtained by calling the copy routine defined in its virtual table. Move assignments are processed similarly, but additionally unset the pointer stored in their source.

Structure Instances Since structure instances are represented as aggregates of pointers to references, extracting a member boils down to a copy of the corresponding pointer to a local register, which as a result can be used like any other reference. The strategy is sound because of the two level of indirection. Indeed, as the *pointer* to the reference is copied rather than the reference object itself, modifications thereupon also affect the member's value.

Parameters and Return Values In the previous section, we saw that the passing of parameters and return values is performed by the means of regular assignments. This means that these references actually denote the reference objects representing the function's parameters and return register in the next call frame's locals. Consequently, their respective pointers can be simply copied into the next call frame.

Reference Capabilities As mentioned above, reference objects are associated with metadata describing their current capabilities. This metadata defines the state of the reference and whether it is mutating. Upon assignment, the interpreter first checks this information to verify that the assignment is legal and then updates it so that it reflects the situation once the assignment has been performed.

A reference can be in either of five states. The *unallocated* state denotes references that are not bound to any value container. The *unique* state denotes unaliased references. The *shared* state denotes unique references having fractioned their uniqueness. It is further associated with the number of fragments that have been formed. The *borrowed* state denotes references having borrowed a uniqueness fragment. It is further associated with a pointer to the lending reference.

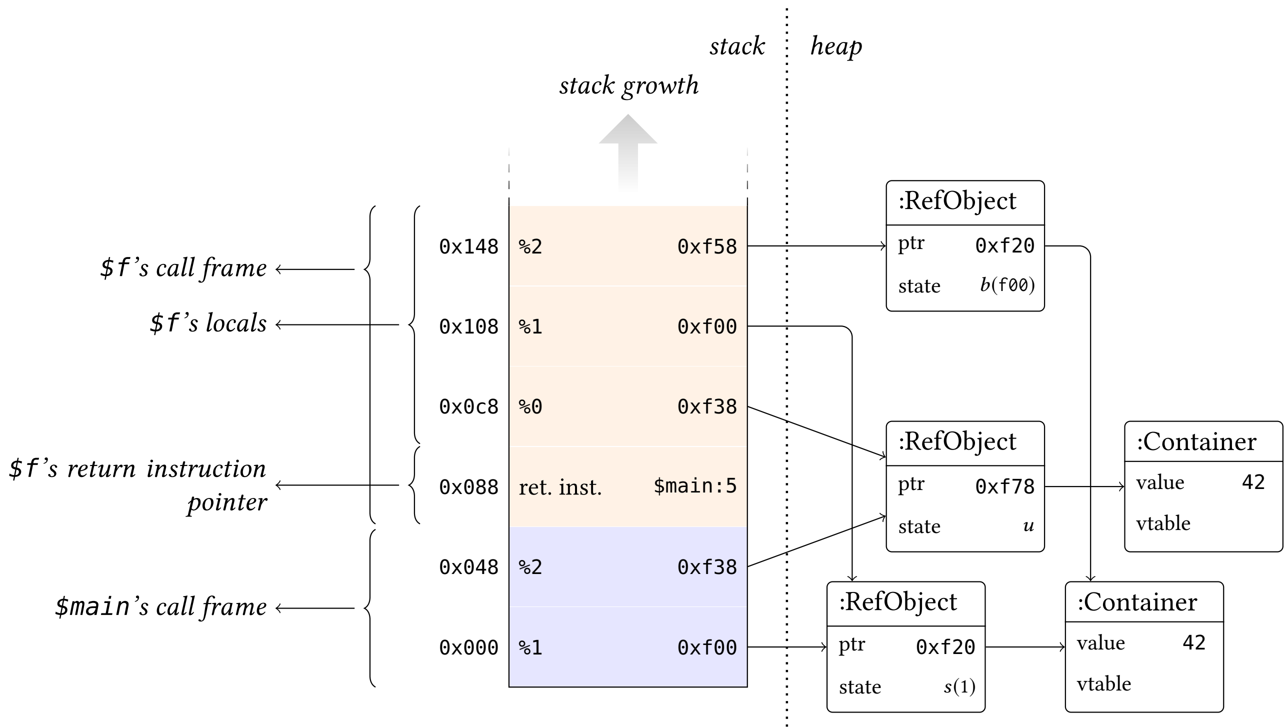

Figure 4. Snapshot of the interpreter's memory while executing an AIR program.

The *moved* state denotes moved references. The constraints enforced by each assignment operator and the transitions between each state are formally defined in [17].

5.3 Example

Consider the following AIR program:

```
1  fun $main : () -> Nothing {
2  entry#0:
3    %1 = make_ref Int
4    move 42, %1
5    %2 = apply $f, %1
6    jump exit#0
7  exit#0:
8    ret %0
9  }
10
11 fun $f : (Int) -> Int {
12 entry#0:
13   %0 = make_ref Int
14   %2 = make_ref Int
15   bind %1, %2
16   copy %2, %0
17   jump exit#0
18 exit#0:
19   ret %0
20 }
```

Figure 4 depicts a snapshot of the interpreter's memory before executing the `ret` instruction at line 19. Hexadecimal values (i.e. numbers starting with `0x`) represent memory addresses. For simplicity, we neglect mutability capabilities and focus on reference states, using the letters u, s and b to denote the unique, shared and borrowed states, respectively.

Line 3 creates a reference object stored at `0xf00` and assigns its pointer to the register `%1` in the `$main` function's call frame. It is assigned by move at line 4, leading to the creation of a value container stored at `0xf20`. Line 5 applies the function `$f`, associating `%0` (resp. `%1`) in `$f`'s call frame to `%2` (resp. `%1`) in `$main`'s call frame. In function `$f`, line 13 and 14 create the reference objects stored at `0xf38` and `0xf58`, respectively. `%2` is assigned by alias at line 15, resulting in a duplication of the pointer to the container stored at `0xf20`. Reference capabilities are updated to indicate that `%1` and `%2` are respectively shared and borrowed. Line 16 copies `%2`'s value into a new container, stored at the address `0xf78`, and assigned to the return register. The assignment also updates the latter's capability to specify that is is unique.

6 Related Work

Our implementation borrows heavily from the design of Apple's Swift compiler [1]. Swift too is translated to an intermediate representation, called Swift Intermediate Language (SIL), which is leveraged to perform various language-specific optimizations, before programs are fed to LLVM [12]

and eventually compiled into machine code. AIR shares some similarities with SIL. Both languages are SSA-form [4] IRs with constructs designed to ease the implementation of their high-level counterpart. Although far more modest in terms of supported features, AIR distinguishes itself by its support for different explicit assignment semantics.

Intermediate languages also appear in multiple other languages' compilers. Rust for instance performs static checks for its elaborate aliasing control mechanisms in a so-called mid-level intermediate representation [14]. However, Rust's intermediate language stays at a much higher-level than AIR and only expands Rust's construct into simpler instructions, better-suited for static pointer analysis.

7 Conclusion

We have presented an implementation of a compiler for the Anzen [17], a programming language with explicit and controllable memory assignment semantics. Our work aims to set a reference implementation for future developments and extensions of the Anzen programming language. All sources are available at https://github.com/anzen-lang/anzen.

Our implementation transpiles Anzen sources to an intermediate language called the Anzen Intermediate Representation (AIR), the main contribution of this paper. AIR is designed to ease the analysis and evaluation of Anzen's statements, by exposing how references are manipulated in simple instructions. We have also described a virtual machine to execute AIR programs, that uses two levels of indirection to represent references as first-class objects. Type capabilities [3] are associated with each reference to track uniqueness and immutability at runtime.

Future works include the implementation of language-specific optimizations. In particular, static knowledge on a given program could be leveraged to avoid indirection in situations where first-class references are not required (i.e. in the absence of aliasing). Determining such situations is straightforward, thanks to Anzen's explicit assignment operators. Immutability also offers useful hypotheses [8]. Another exciting perspective involves Just-in-Time (JIT) compilation. One promising lead is to leverage AIR's closeness with LLVM [12] to leverage the latter's JIT compilation support.

While we have described AIR and its interpretation in the context of Anzen, we believe this intermediate language to be suitable to express other imperative-oriented languages that advocate for precise memory management strategies. Its native support for higher order functions qualifies it to represent functional patterns as well.

References

[1] Apple. 2019. The Swift Programming Language. https://github.com/apple/swift. (2019).

[2] John Boyland. 2003. Checking Interference with Fractional Permissions. In *SAS 2003*. Springer, Berlin, Heidelberg, 55–72. https://doi.org/10.1007/3-540-44898-5_4

[3] John Boyland, James Noble, and William Retert. 2001. Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only. In *ECOOP 2001*. Springer, Berlin, Heidelberg, 2–27. https://doi.org/10.1007/3-540-45337-7_2

[4] 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

[5] Iulian Dragos and Martin Odersky. 2009. Compiling generics through user-directed type specialization. In *ICOOOLPS 2009*. ACM, New York, NY, USA, 42–47. https://doi.org/10.1145/1565824.1565830

[6] Karel Driesen. 1999. *Software and Hardware Techniques for Efficient Polymorphic Calls*. Ph.D. Dissertation. University of California, Santa Barbara, CA, USA.

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

[8] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and reference immutability for safe parallelism. In *OOPSLA 2012*. ACM, New York, NY, USA, 21–40. https://doi.org/10.1145/2384616.2384619

[9] David Gregg, Andrew Beatty, Kevin Casey, Brian Davis, and Andy Nisbet. 2005. The Case for Virtual Register Machines. *Science of Computer Programming* 57, 3 (2005), 319–338. https://doi.org/10.1016/j.scico.2004.08.005

[10] Bastiaan Heeren, Jurriaan Hage, and Doaitse Swierstra. 2002. *Generalizing Hindley-Milner Type Inference Algorithms*. Technical Report. Institute of Information and Computing Sciences, Utrecht University.

[11] Mark P. Jones. 1997. First-class Polymorphism with Type Inference. In *POPL 1997*. ACM, New York, NY, USA, 483–496. https://doi.org/10.1145/263699.263765

[12] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO 2004*. IEEE, San Jose, CA, USA, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[13] 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 *PLOS 2015*. ACM, New York, NY, USA, 21–26. https://doi.org/10.1145/2818302.2818306

[14] Niko Matsakis. 2019. Introducing MIR. https://blog.rust-lang.org/2016/04/19/MIR.html. (2019). Accessed: 2019-07-27.

[15] Martin Odersky, Dan Rabin, and Paul Hudak. 1993. Call by Name, Assignment, and the Lambda Calculus. In *POPL 1993*. ACM, New York, NY, USA, 43–56. https://doi.org/10.1145/158511.158521

[16] Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Transactions on Programming Languages and Systems* 22, 1 (2000), 1–44. https://doi.org/10.1145/345099.345100

[17] Dimitri Racordon and Didier Buchs. 2019. Explicit and Controllable Assignment Semantics. *CoRR* (2019).

[18] John C. Reynolds. 1998. Definitional Interpreters for Higher-Order Programming Languages. *Higher-Order and Symbolic Computation* 11, 4 (1998), 363–397. https://doi.org/10.1023/A:1010027404223

[19] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. 2006. Type Processing by Constraint Reasoning. In *APLAS 2006*. Springer, Berlin, Heidelberg, 1–25. https://doi.org/10.1007/11924661_1

[20] Jesse A. Tov and Riccardo Pucella. 2011. Practical Affine Types. In *POPL 2011*. ACM, New York, NY, USA, 447–458. https://doi.org/10.1145/1926385.1926436

[21] Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2017. SHErrLoc: A Static Holistic Error Locator. *ACM Transactions on Programming Languages and Systems* 39, 4 (2017), 18:1–18:47. https://doi.org/10.1145/3121137