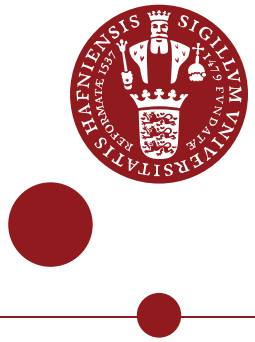
UNIVERSITY OF COPENHAGEN
DEPARTMENT OF COMPUTER SCIENCE



Bachelor project

Nikolaj Krarup, ltf688

Reference counting and memory management for Fasto

Block 3-4, 2025

Advisor: Andrzej Filinski

June 10, 2025

Abstract

This bachelor's project attempts to implement memory management through reference counting for the functional programming language Fasto. Fasto relies heavily on arrays but has no built-in garbage collection; when the heap is full, programs crash. A simplified subset of Fasto called MiniFasto was created as a prototyping environment, allowing rapid development and analysis of different approaches.

The core strategy involves transforming MiniFasto programs into an intermediate representation known as A-Normal Form (ANF), which significantly simplifies the analysis of variable liveness. Flattening nested expressions into linear sequences of explicit let-bindings makes it straightforward to determine when variables are no longer needed (dead) and their reference count should be updated. Proper definitions of live sets in the ANF structure were defined, and a recursive analyser was developed to insert increment and decrement annotations as precisely as possible, based on said live sets.

The correctness of the implementation is based on a thorough analysis of the design and rigorous testing. Porting the reference counting to Fasto remains theoretical. However, the groundwork laid in this project provides a clear path toward practical integration into the complete Fasto compiler.

Contents

1 Background

1.1 The Fasto language

(Note: figures and theory from this section are taken from the group assignment handout given to students of the IPS course in 2024. The full handout is included in the digital appendix.)
The Fasto programming language was created as an educational tool for the course "Implementation of Programming Languages (IPS)" at the University of Copenhagen. The implementation of the language was to be finished by the students of the course. The unfinished source code for Fasto given in the course contains an interpreter as well as a compiler written in `F#`. The work in this paper builds upon the resulting Fasto compiler developed during the course.

Fasto is a simple, first-order functional language that supports recursive definitions and multidimensional arrays. Other than arrays, Fasto has the three simple types (`int`, `bool`, `char`). See figure 1 for the full syntax of the Fasto language.

Prog → *FunDecs*

FunDecs → `fun` *Fun FunDecs*
FunDecs → `fun` *Fun*
Fun → *Type* ID `(` *Params* `)` `=` *Exp*
Fun → *Type* ID `(` `)` `=` *Exp*

Params → *Type* ID `,` *Params*
Params → *Type* ID

Type → `int`
Type → `char`
Type → `bool`
Type → `[` *Type* `]`

Exp → ID
Exp → ID `[` *Exp* `]`
Exp → NUM
Exp → `true`
Exp → `false`
Exp → CHARLIT
Exp → STRINGLIT
Exp → `{` *Exps* `}`

Exp → *Exp* `+` *Exp*
Exp → *Exp* `-` *Exp*
Exp → *Exp* `*` *Exp*

(... continued on the right)

Exp → *Exp* `/` *Exp*
Exp → *Exp* `==` *Exp*
Exp → *Exp* `<` *Exp*
Exp → `~` *Exp*
Exp → `not` *Exp*
Exp → *Exp* `&&` *Exp*
Exp → *Exp* `||` *Exp*
Exp → `(` *Exp* `)`
Exp → `if` *Exp* `then` *Exp* `else` *Exp*
Exp → `let` ID `=` *Exp* `in` *Exp*
Exp → ID `(` *Exps* `)`
Exp → ID `(` `)`
Exp → `read` `(` *Type* `)`
Exp → `write` `(` *Exp* `)`

Exp → `iota` `(` *Exp* `)`
Exp → `length` `(` *Exp* `)`
Exp → `replicate` `(` *Exp* `,` *Exp* `)`
Exp → `map` `(` *FunArg* `,` *Exp* `)`
Exp → `filter` `(` *FunArg* `,` *Exp* `)`
Exp → `reduce` `(` *FunArg* `,` *Exp* `,` *Exp* `)`
Exp → `scan` `(` *FunArg* `,` *Exp* `,` *Exp* `)`

Exps → *Exp* `,` *Exps*
Exps → *Exp*

FunArg → ID
FunArg → `fn` *Type* `(` `)` `=>` *Exp*
FunArg → `fn` *Type* `(` *Params* `)` `=>` *Exp*

Figure 1: Syntax of the FASTO Language.

The four lexical atoms in the syntax are (**ID**) variable and function names, (**NUM**), numbers, (**CHARLIT**), chars, and (**STRINGLIT**) strings.

A Fasto program is a list of function declarations, one of which must be called `main` and have no parameters. The execution of a Fasto program will always start with calling this

function. The result of the main function is disregarded in the compiled program, output is only produced from explicit calls to the built-in function `write`.

Tough Fasto is a functional language, it still contains two built-in functions (`read`, `write`) containing side effects (I/O). These functions are polymorphic, meaning that their types are not expressible in Fasto, and as such, they are represented with specific `Read` and `Write` nodes in the abstract syntax.

1.1.1 Fasto Arrays

Fasto has an array type constructor `[]`, to create types like `[bool]` or `[[int]]`, the former being an array of boolean values and the latter being a 2D array of integers (an array of arrays of integers). The sub-arrays of a multidimensional array need not have identical lengths.

Arrays can be created using array literals or array constructor functions (ACs) such as `iota` and `replicate`. They can be modified and collapsed using second-order array combinators (SOACs) such as `map` and `reduce`. The SOACs `map`, `reduce`, `scan` and `filter` apply functions to each element in an array, taking a function as a parameter. This can either be a user-defined function or a lambda expression (anonymous function). The syntax for lambda-expressions in Fasto can be found in the last two lines in the right column of figure 1. See Listing 1 for an example of a Fasto program making use of SOACs, ACs and lambda expressions. This program will read an integer, create an array with the AC `iota`, mutate it using two SOACs, writing both results to the standard output.

```
fun int write_int(int x) = write(x)
fun [int] write_int_arr([int] x) = map(write_int, x)

fun int main() =
  let N = read(int) in
  let z = iota(N) in
  let w = write_int_arr(map(fn int (int x) => x + 2, z)) in
  let nl = write("\n") in
  write_int(reduce(fn int (int x, int y)=>x+y, 0, w))
```

Listing 1: A Fasto program.

1.1.2 Code generation

The Fasto code generator function `compile`, located in `codegen.fs`, will convert a `TypedProg`, into an `Instruction list`. `TypedProg` is a `FunDec` list. The body of a `FunDec` holds an `Exp` which, like `Instruction`, is a discriminated union defined using the `F# type` keyword. `TypedProg` is defined in the `AbSyn.fs` file and follows the rules of figure 1. `Instruction` is defined in `RiscV.fs`, and is an abstract syntax for RISC-V32 (Waterman and Asanovi 2019). The code generator also contains functions for compiling expressions, let-declarations, function arguments and functions, which are used when compiling the program. There are many other files, functions and types, such as a lexer, a parser, a register allocator, a type checker, etc. However, these will not be further explained as this paper's main focus is on the code generation, i.e. translating a `TypedProg` into an `Instruction list`. Currently, this translation is done directly, meaning there is no intermediate code generation. The main

program `Fasto.fs` is the driver of the compiler, to compile a Fasto program to RISC-V code, it does the following: running the lexer and the parser, validates the abstract syntax using the type checker, (optionally) rearranges it in the optimizer, and finally compiles it to RISC-V code.

In the resulting RISC-V code, arrays have the structure as seen in figure 2, here we consider the word size to be 4 bytes. So a Fasto array is a pointer to a memory address, where the array is located as a continuous memory block. The array has a one-word header which contains the length of the outer dimension of the array. Furthermore, arrays are word aligned, so the address of the header is located on a memory address divisible by 4. So even if the array consists of a 1-byte type, and the last element is not on a word-aligned address, additional memory is allocated so subsequent memory allocation will occur on a word-aligned address. Arrays of the type `[bool]` and `[char]` will take up $4 + len$ bytes, whereas multidimensional arrays and `[int]` arrays will take up $(1 + len) \cdot 4$ bytes. The additional 4 bytes are due to the header.

Multidimensional arrays consists of memory addresses to the sub-arrays. These might point to the same array, if an AC such as replicate is used.

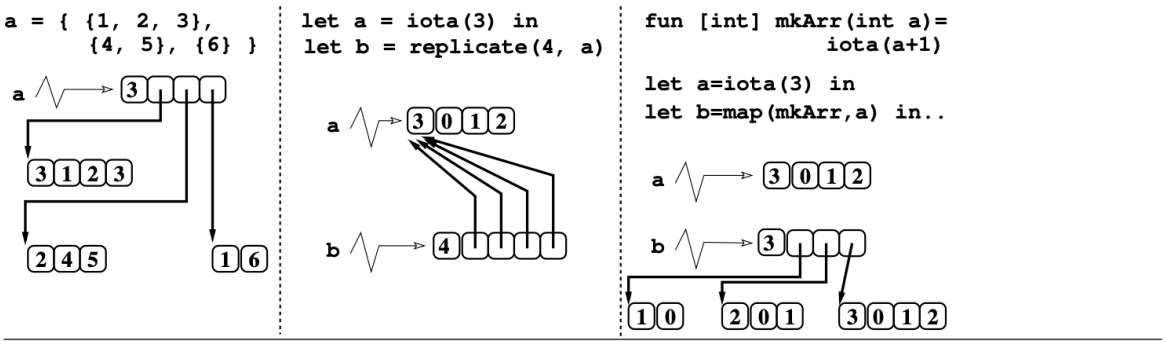


Figure 2: Array Layout.

For allocation of memory (arrays) in the heap, the Fasto compiler uses a stack-like allocation approach, a more simplified version of the one described in (Torben Ægidius Mogensen 2022, p. 73), with no way to free memory. When the code generator needs to allocate an array, it will call the `dynalloc` function, which generates code for moving the heap pointer by a certain number of words and places the old heap pointer address into a given register; this register will contain the address of the allocated array. As such, the heap pointer will keep increasing until no more memory is left, at which point the program crashes.

1.2 Memory management

Efficient memory management is crucial for programs that dynamically allocate data on the heap. In the current Fasto compiler, memory is allocated by simply moving a heap pointer with no way to free memory, so the heap grows until it is exhausted and the program eventually crashes. In general, managing heap allocation raises issues of fragmentation and object lifetime. Fragmentation occurs when freed blocks are scattered such that available memory splits into many small pieces. Object lifetime refers to the period during which a data object is in use; once an object is no longer needed (no live references), its space should be reclaimed to avoid leaks. This thesis focuses on the latter.

There are two broad strategies for memory management: manual and automatic. Manual memory management leaves allocation and deallocation to the programmer (e.g., using `malloc/free` in C). This gives explicit control but is error-prone. Automatic memory management relieves the programmer of this task, automatically freeing a block when the program can no longer access it. This approach eliminates explicit `free` calls, at the cost of additional runtime complexity. An approach to this is reference counting, which is described next. (Torben Ægidius Mogensen 2022, Chapter 4)

1.3 Reference counting

Reference counting is a form of automatic memory management that attaches a counter to each heap-allocated object. The counter tracks how many active references point to the object. Every time a reference to the object is created or copied, the counter is incremented; every time a reference goes out of scope or is reassigned, the counter is decremented. If the counter ever drops to zero, meaning no references remain, the object is immediately deallocated. This ensures that unused objects do not stay on the heap. The reference counting insertion used in this project works by analysing program variables liveness and inserting counter increments and decrements at appropriate points, so that each objects count reflects the number of live pointers.

While simple in principle, reference counting has well-known pitfalls and trade-offs. For example, it can be a challenge to ensure the correctness of count updates. A mistake in incrementing or decrementing can either leak memory or free an object too soon. There is also a runtime cost to maintaining reference counts, such as freeing an object. If the object holds pointers to other heap objects (like Fasto's multi-dimensional arrays), the system must recursively decrement those referents counters as well, potentially triggering a cascade of deallocations. These overheads make RC generally slower than stack allocation or explicit freeing. On the other hand, reference counting distributes deallocation work across the programs execution (freeing objects immediately when they become unreachable) rather than pausing execution for periodic garbage collection, which can be an advantage for responsiveness. (Torben Ægidius Mogensen 2022, Chapter 4.7)

1.4 Tools and Technologies

This project leverages several tools and technologies for implementation and testing. The target architecture is RISC-V-32, a modern RISC instruction set architecture. We generate RISC-V assembly as the output of the Fasto compiler. To run the generated code, we use RARS, the RISC-V Assembler and Runtime Simulator (Duncan et al. 2023). RARS is an educational sandbox for RISC-V, capable of assembling and simulating RISC-V programs with useful debugging features.

We implement the dynamic garbage collection functions in C and compile them to RISC-V assembly using the Clang compiler, linking them with the Fasto-generated code. Some minor incompatibilities between Clangs output and the RARS simulator were resolved with simple post-processing scripts. Finally, since the Fasto compiler itself is written in F#, we used F# to integrate all components. The analysis passes (ANF transformation, liveness analysis, and RC annotations) were implemented in F#. Bash was used for a build script to automate the compiling and cleaning of the C code, described in the next section.

2 Preparing Fasto

To be able to implement reference counting in Fasto, we need a way to free memory. As stated, memory is currently allocated by simply moving the heap pointer. This straightforward stack-like approach is insufficient for reference counting, as freeing becomes tricky. As such, we implement dynamic memory management for Fasto. Since Fasto's tool chain ultimately emits 32-bit RISC-V for the RARS simulator, we create the library in plain C and compile it to RISC-V using Clang. We also need a small run-time system to get Clang's generated machine code to work with RARS. The code for all this can be found in `Fasto-memory/memory`. It consists of the following:

- `mem.c, mem.h`. An Alloc/Free library using a free list, similarly to how it is done in chapter 4.5 in (Torben Ægidius Mogensen 2022).
- `lib.s`. Two handwritten RISC-V functions, performing RARS system calls:

  `allocateHeap:` Allocates the heap with `sbrk` system call.

  `printInt:` Call Fasto's `p.putint` function, performing a system call for printing integers. Only for debugging purposes, facilitating print statements from the C environment, without adding libraries.

- `cleanupRV.fsx`. An F# post-processing script, to adapt the generated assembly from Clang to work with RARS
- `memTests.c`. A small testing suite for the Alloc/Free library.

At first call, the allocate function will perform the system call using the function from `lib.s`, to allocate the heap. It is initialised as a single big block, which is split into smaller blocks upon later calls, using a free list. The block header is the size of the block and a pointer to the next block. On later calls, it walks the free list, splits blocks when useful, and returns a pointer past the header, where Fasto expects the array-length word to be. Currently, the header has no reference count, as reference counting never actually reached Fasto. When freeing an array, we add it to the front of the free list.

This library should be added at the bottom of compiled Fasto programs. To do so, Clang's compiled code must be compatible with RARS. Therefore, we have the `cleanupRV.fsx` script, that removes any directives unknown to RARS, and appends `lib.s` to the compiled assembly. It is a bit brittle currently and should be extended to handle all directives. Now we can compile the `mem.c` with Clang, clean it up and add it to compiled Fasto programs. To make this easier, we have created the script `Fasto-memory/bin/compilecleanc.sh`. We can call this on a `file.c` to compile with Clang, and clean with the F# script, outputting the files `file.s` and `fileclean.s`.

To get Fasto to use this library, we have to replace the old `dynalloc` function in `codeGen.fs` with a new `allocate` function. This function will load the type of the array elements and the size of the array into the argument registers `x10` and `x11`, before adding a `JAL` instruction to the allocate function defined in `mem.c`. Finally, a `MV` instruction moves the returned value from `x10` into the desired register. Before calling a function from the Clang compiled code, we should save registers `x28 - x31`, since these are caller-saved in Clang, but not in the Fasto compiler. Now we can replace calls to `dynalloc` with calls to `allocate` in the `codeGen.fs` file.

To test, we have done this for the `iota` code generation. We can then compile the `iota.fo` test, with Fasto, compile and clean `mem.c` with the cleaning script, merge the files and run the program with RARS:

```
$ ./bin/fasto.sh -o ./tests/iota.fo
$ ./bin/compilecleanc.sh mem.c
$ cat ./memory/memclean.s » ./tests/iota.asm
$ ./bin/rars.sh ./tests/iota.asm
-----------------------------------------------
Wrote clean file to  ./bin/../memory/memclean.s
0 1 2 3 4 5 6 %
```

`mem.c` also has a small debug mode, which was used to perform additional testing within C, using Clang instead of RARS. Here we can simply use `malloc` instead of system calls to allocate te heap. The test files can be found in `memTests.c`. To run the tests, do the following:

```
$ cd memory
$ Clang mem.c memTests.c -o memtests
$ ./memtests
-----------------------------------------------
Running allocator tests...
..
All tests passed successfully.
```

3 Reference counting in MiniFasto

Adding reference counting to Fasto is a significant and complex task due to the extensive and interconnected structure of its source code. So, instead of attempting to implement reference counting directly in the main source code of Fasto, a different approach is taken, namely making a simplified subset of Fasto, directly in F#, removing unnecessary components unrelated to memory management. The simplified Fasto version, MiniFasto, will serve as a lightweight sandbox environment, facilitating quick prototyping, iterative development, and fast observation of results. The implementation can be found in the file `Fasto-memory/miniFasto/miniFasto.fs`.

3.1 Initial design

Programs of the language will be written directly in F#, so there is no need for lexing and parsing. Next is the abstract syntax of the language, the selection of which is crucial for MiniFasto to be useful in this project. It has to mimic the essence of the Fasto language and include each attribute that can affect memory management. We have chosen the following subset:

$$
\begin{aligned}
Value \rightarrow \quad & \texttt{NUM} \mid \texttt{ID} \\
Exp \rightarrow \quad & Value \mid \{Exp, ..., Exp\} \mid Exp + Exp \mid \\
& \texttt{ID}[Exp] \mid \text{len}(Exp) \mid \text{let } \texttt{ID} = Exp \text{ in } Exp \mid \\
& \texttt{ID}(Exp, ..., Exp) \mid \text{if } Exp \text{ then } Exp \text{ else } Exp \mid \\
& \text{map}(\text{fn } \texttt{ID} \Rightarrow Exp, Exp)
\end{aligned}
$$

The full abstract syntax of the language can be seen in listing 2. It's a subset of Fasto's syntax (from `AbSyn.fs`), carefully selected to capture the essential edge cases relevant for analysis and compilation, such as array computations, function calls, conditional branches and SOACs. For example, by adding the plus computation to the syntax, we have covered all the arithmetic expressions, since minus and times are no different than plus with regards to memory usage. For the same reason, the `char` and `bool` types have also been left out, even though they differ a bit from `int` in terms of their size in memory; they don't make much of a difference in the context of reference counting. An assumption about SOACs is also made, that `map` is enough to illustrate the behaviour of `filter`, `reduce` and `scan`. Furthermore, `map` can only use locally defined lambda functions, because the complexities of lambda functions are important to consider since they can reference variables from the surrounding scope, whereas named functions should be more straightforward. The AC's such as `replicate` and `iota` have been completely left out, as MiniFasto already has array literals, and the dynamic length won't make much difference. The focused design ensures minimal unnecessary bloat, while remaining expressive enough to model the core constructs that we should consider in order to implement memory management in Fasto. Like real Fasto, a program is a list of function declarations, which should include a `main` function with no parameters.

MiniFasto has minimal error handling, as it will be used carefully to experiment with ideas. So, there is no actual type checker, but types will be checked where necessary when interpreting and compiling.

Another necessary component is a symbol table. It is used to look up identifiers to get useful information while compiling/interpreting. Fasto makes use of the `SymTab` defined in

```
type fname = string
type vname = string

type Type =
    Int
  | Array of Type

type Value =
    IntVal   of int
  | ArrayVal of Value list * Type (* Type corresponds to element type *)

type Param = Param of vname * Type

type Exp =
    Constant  of Value
  | Var       of vname
  | ArrayLit  of Exp list * Type
  | Plus      of Exp * Exp
  | Let       of vname * Exp * Exp (* Let "a" = exp in exp *)
  | Index     of vname * Exp * Type
  | Length    of Exp
  | If        of Exp * Exp * Exp (* If exp != 0 Then exp Else exp *)
  | Apply     of fname * Exp list
  | Map       of Param * Exp * Exp * Type * Type

type FunDec = FunDec of fname * Type * Param list * Exp
type Prog = FunDec list
```

Listing 2: Definition of MiniFasto syntax

the Fasto src, but for the sake of simplicity MiniFasto uses F#'s build in `Map` type.

```
type VarTableI = Map<vname,Value> (* Interpretor tables *)
type FunTableI = Map<fname,FunDec>
type VarTableC = Map<vname,reg>   (* Compiler vartable *)
```

Listing 3: Definition of symbol tables

There are three specific types of symbol tables, the first two are for the interpreter, binding strings to values and functions, a *vtable* and a *ftable*. The last one is for the compiler, a *vtable* binding variable names to registers.

3.1.1 Interpreter

The interpreter is quite simple, following the same structure as Fasto, with much of the code taken directly from there. The only purpose of the interpreter is testing, to verify the correctness of compiled programs. It's made up of the function signatures seen in listings 4.

```
bindParamsI : Param list -> Value list -> VarTableI
evalExp     : Exp -> VarTableI -> FunTableI -> Value
callFun     : FunDec -> Value list -> FunTableI -> Value
evalProg    : Prog -> Value
```

Listing 4: MiniFasto interpreter

The functions closely follow the structure of the interpreter explained in (Torben Ægidius Mogensen 2024, chapter 4.3). `evalProg` initialises the `FunTable`, searches for the `main` function, and then calls `evalExp` on the function body's expression with the *ftable* and an empty *vtable*. `evalExp` works recursively. It matches the expression with the different `Exp` cases from listing 2, and calculates accordingly. Since we have an expression-based language, it will always return a value. For function calls, it uses the function `callFun`, which calls `bindParamsI` to create a function argument *vtable*, and then evaluates the function's expression with `evalExp`. Arrays in the interpreter are represented using F#'s built-in `list` type, utilising F#'s garbage collector, consequently never running out of memory.

3.1.2 Compiler

Fasto's compiler translates to RISC-V32, so to mimic this, MiniFasto has pseudo-RISC-V instructions, which are a more compact subset of Fasto's discriminated union `Instruction`. Like with the `Exp` syntax, it's kept minimal in order to keep things simple, so only the necessary instructions are added, see Listing 5. To simplify it further, it also has some additional

```
type reg  = string
type imm  = int
type addr = string

type PseudoRV =
  | LABEL of addr
  | LI    of reg*imm       | MV  of reg*reg
  | ADDI  of reg*reg*imm   | ADD of reg*reg*reg
  | LW    of reg*reg*imm   | SW  of reg*reg*imm
  | BEQ   of reg*reg*addr  | J   of addr
  | CALL  of addr*reg list | RET
  | ALLOC of reg*reg

type RVProg = Map<string, reg list * PseudoRV list>
```

Listing 5: Pseudo RISC-V types

pseudo-instructions. `CALL` and `RET` are used to streamline function calls and returns, similarly to how it's done in the intermediate language in (Torben Ægidius Mogensen 2024, chapter 6). `CALL` has a list of registers containing the actual parameters' values, `RET` places the returned value in a special return register. As stated, arrays are allocated in Fasto by bumping the heap pointer. Still, since the pseudo-RISC-V instructions will be simulated in F#, the `ALLOC` pseudo instruction is also included; the second of its registers should contain the size that is wished to be allocated, placing the address of the allocated array in the first. An `RVProg` is an F# `Map`, where a function name has a list of registers (arguments) and an instruction

list (function body). Registers are purely symbolic, so things such as register spilling are not needed. Two special registers are defined: `x_0`, which always contains 0, and `x_ret`, where function return values will be placed. A global mutable counter variable initialised as 1 is used and incremented by naming functions to create fresh and unique register and label names.

Fasto has no intermediate code generation; likewise, in the first iteration of MiniFasto, expressions were translated directly to `PseudoRV`. This was later scrapped in favour of adding an intermediate step, but its key concepts in code generation still served as a useful springboard in creating the final compilation pipeline. The original compiler consists of the functions seen in listing 6, where `compileExp` closely follows the structure of `evalExp` in listing 4. However, instead of returning a value, it returns a list of instructions which loads the expression's value into a specified register. Most of the implementation is very close to the original Fasto compiler, with a few tweaks to match the `PseudoRV` instructions rather than Fasto's `Instruction` type.

```
compileExp  : Exp -> VarTableC -> reg -> PseudoRV list
compileFun  : FunDec -> string * (reg list * PseudoRV list)
compileProg : Prog -> RVProg
```

Listing 6: MiniFasto original compiler

3.1.3 Simulator

To run `PseudoRV` instructions, a simulator is needed. Rather than translating the instructions into actual RISC-V code, to run in RARS like Fasto, a small `PseudoRV` simulator was created. The types in listing 7 are used to replicate the data structures of registers and the heap. The symbolic registers are located in a map, with no upper bound. The heap is only used for arrays; they are in a map with a corresponding address.

```
type Registers = Map<reg, int>
type Heap = Map<int, array<int> >
```

Listing 7: MiniFasto original compiler

In the simulator, heap addresses point to words, rather than bytes, as MiniFasto doesn't support `bool` or `char` types. An address is an integer of the form `offsetSize*object# + offset`, such that the heap is represented as the mapping `(object# -> array)`. The `offsetSize` is 1000, meaning the maximum array length is 999. So the heap address 1002 would correspond to the third element in the second array in the heap map. This way, the heap closely mimics an actual heap with integer addresses, but abstracted for easier designing and better readability.

```
simulate : Heap -> RVProg -> int * Heap
    simulateFun  : reg list * PseudoRV list -> int list -> int
        simulateInst  : PseudoRV -> addr option
        simulateBlock : PseudoRV list -> int
```

Listing 8: MiniFasto RV simulator

The simulator, see listing 8, consists of a single function `simulate`, which takes an initial heap, an `RVProg`, and returns the result along with a new heap. It has a mutable heap object, initialised as the heap argument, and a local recursive function `simulateFun`. `simulateFun` simulates a function from a `RVProg` using a corresponding list of argument. It has a mutable register bank, so each function is called with fresh registers. These are initialised by adding the formal parameters (registers) and loading the actual arguments from the integer list into these. It also has two local functions: `simulateInst` and `simulateBlock`. `simulateInst` simulates a single instruction, mutating the heap and registers appropriately and returning an `addr option` indicating whether a jump is needed. For the pseudo instruction `CALL`, `simulateFun` is called recursively, and for `ALLOC`, a new array is added to the mutable heap object. `simulateBlock` will recursively go through the instruction list, calling `simulateInst`. If `Some addr` is returned, it will continue from the label instead of the next instruction in the list. A special label indicates that the simulator has arrived at the return statement `RET`, prompting the function to return the value stored in the return register.

3.1.4 Drawbacks

The initial design has some drawbacks, which motivated the scrapping of the initial compiler in favour of an intermediate representation of expressions. Even though MiniFasto has a somewhat simple syntax, due to the recursive nature of `Exp`'s grammar, it can be deeply nested, which makes it hard to analyse. For example, this is a valid MiniFasto expression:

```
f(a[0] + g(h(a[2])), b)
```

Listing 9: MiniFasto expression, example 1

If this expression contains the last use of the variable `a`, we need to pinpoint where exactly that is. To do so, the analyser must understand the full expression tree, and look for all the uses of `a` which may appear multiple times and maybe even in branches (not in this example). This is quite complex to analyse. Adding conditionals to the mix, complicates things even further:

```
let x = foo( if bar(z[0] - 1)
        then { g(a[2] + h(b)), c + 5 }
        else { d[ k(2) ], e * f(3) }
        ) in
x
```

Listing 10: MiniFasto expression, example 2

The function `foo` is called with an If-expression with a guard containing a function call, and whose branches each build an array literal containing indices, arithmetic, and nested function calls. To decide where the variables are last used, the analysis must inspect the entire nested tree and reason about evaluation order in both branches. Because the conditional is itself nested in an expression, the argument of `foo`, it is hard to see locally where its joint point lies.

Furthermore, the current syntax makes it hard to represent the results of an analysis. With such nested syntax, there are no logical points to insert liveness annotations.

4 A-Normal form

As the core of this project is implementing reference counting in Fasto, the first critical step is detecting the last use of variable names. The approach that is used in this project is to add an intermediate step in the compilation. The source language is first compiled into a structured, functional intermediate language known as ANF (A-Normal form). This is then analysed and optimised, before being translated into machine code. The full pipeline of flattening, analysing and generating code can be found in the file `Fasto-memory/miniFasto/ANorm.fs`.

ANF originates from the work done by (Flanagan et al. 1993), as they rethought transformations in continuation-passing style (CPS). Classical CPS-based compilers transform source programs into a form where all computations are made explicit through continuation parameters. However, this naïve CPS transformation tends to bloat the size of the intermediate program, requiring additional compaction passes and specialised treatment of continuations in code generation. Flanagan et al. found that the later compaction and code generation stages invert the CPS translation. Based on this, they argued that CPS transformations were irrelevant, as the same result can be achieved with a simpler source-to-source transformation that simulates the compaction phase. This transformation is known as A-Normal form.

ANF enforces strict syntactic rules that ensure that every intermediate computation is explicitly named with a let-binding, ensuring that every sub-expression is a direct value or reference. In MiniFasto, the version of ANF used is based on the core ideas of (Flanagan et al. 1993), but tailored to fit the Fasto language. The syntax is defined below.

$$
\begin{aligned}
A \rightarrow &\quad V \mid \text{let } x = C \text{ in } A && (1)\\
V \rightarrow &\quad n \mid x && (2)\\
C \rightarrow &\quad f(V, \ldots, V) \mid V + V \mid \{V, \ldots, V\} \mid V[V] \mid \\
&\quad \text{if } V \text{ then } A \text{ else } A \mid \text{map}(\text{fn } x \Rightarrow A, V) && (3)
\end{aligned}
$$

Where n is an integer, x is a variable name, f is a function name, and $(\text{fn } x \Rightarrow A)$ is a lambda function.

In (1), we have the ANF term, which is the top-level expression that adheres to the A-Normal form structure. It is quite similar to a subset of the Fasto syntax, including let bindings, which are the primary building blocks of programs, used to sequence computations and atomic values as the leaf nodes, defined in (2).

The computations defined in (3) are also a smaller subset of Fasto expressions, more specifically, it's the remaining `Exp` types from the MiniFasto syntax. It should be noted that the grammar does not include any of Fasto's I/O functions. However, we will still analyse it as tough these exist, as it's essential to consider possible side effects.

4.1 ANF motivation

So why translate to ANF? As stated, we want to detect the last use of variable names for the sake of reference counting. The complexities of doing this in MiniFasto's `Exp` syntax were previously highlighted. Let's take a look at the expression in Listing 9, translated to ANF form:

```
let t1 = a[0] in
let t2 = a[2] in
let t3 = h(t2) in
let t4 = g(t3) in
let t5 = t1 + t4 in
let t6 = f(t5, b) in
t6
```

Listing 11: ANF example

The deeply nested expression has been "flattened" into a linear tree by naming each intermediate computation. This is one of the constraints of the non-recursive computation term (3), it ensures that the tree consists of non-nested computations chained together by let-bindings. Now it is quite clear that the last use of `a` appears in the computation of `t2`.

Another advantage of the ANF tree consisting of let-computation nodes is that it enforces a structured control flow that always merges. In the original Fasto syntax, even though an If expression must eventually produce a value, the branches themselves do not always merge back into a shared continuation. This lack of an explicit join point makes it harder to reason about last uses, exemplified in Listing 10. However, in ANF, branches are isolated in computation nodes and immediately wrapped by a Let, which guarantees a clear and local merge point in the syntax tree. This creates a flat and linear chain of Lets, where calculating live sets for each node in the chain is possible without needing information outside of the current node's subtree. The same logic applies to loops (SOACs), originally a `map` could appear anywhere, but in ANF it is constrained to a Let binding.

Finally, the language is quite close to 3-address code, which maps closely to machine code. This simplifies the compilation process, especially when reference counting is added to the equation.

Outside of reference counting, there are other valuable qualities to ANF. Since complex expressions are flattened, it is trivial to identify copy patterns like `let x = y`. We can utilise this by removing syntax like the following from the grammar:

$$\text{let } x = V \text{ in } A$$

We can perform copy and constant propagation as part of the translation, ensuring that we never output it by having a more restrictive grammar. This optimises the code while translating it, eliminating the need for this to be done in a second pass through the code, as in the original Fasto language. In the Fasto source code, there remains an issue in which the copy propagation optimisation pass will create incorrect programs due to shadowing. This is not a problem in ANF as copy propagation happens during translation, keeping each variable name in a *vtab* with a corresponding atomic value, instead of an indirect reference through an identifier that may be shadowed. Furthermore, shadowing will not even exist in the resulting ANF program, as we create fresh names for each variable.

Another relatively easy optimisation in ANF is dead-binding elimination. Since the tree is a linear let-chain, it is easy to see whether a variable name is used after it is declared. And since live sets are already needed to analyse the program, all the information is already available. So, dead-binding elimination can be done in the same single pass that the program is analysed. This will be discussed in more detail later.

4.2 Flattening an expression

Before we can flatten an expression to ANF, the syntax needs to be defined in F#, see listing 12. It looks very similar to the original seen in listing 2, except that `Exp` have been split into the two different types `ANorm` and `AComp`, to follow the grammar of (1) and (3). Furthermore, we do not have typed values in ANF syntax. Instead, types are attached to let bindings. This design reflects the structure of ANF, which is centred around naming intermediate computations with let expressions. Since all non-trivial expressions are lifted into separate bindings, assigning types at the binding site is sufficient and more convenient than assigning types to each value node.

```
type AVal =
  | N of int
  | V of vname

type AComp =
  | ApplyA of fname * AVal list
  | AddA of AVal * AVal
  | ArrLitA of AVal list * Type
  | IndexA of AVal * AVal * Type
  | LenA of AVal
  | IfA of AVal * ANorm * ANorm
  | MapA of Param * ANorm * AVal * Type * Type

and ANorm =
  | ValueA of AVal
  | LetA of vname * Type * AComp * ANorm

type AFunDec = AFunDec of fname * Type * Param list * ANorm
type AProg = AFunDec list
```

Listing 12: ANF in MiniFasto

To transform an `Exp` into an `ANorm`, simply using a recursive function with the same structure as `compileExp` to transform each sub-expression would be insufficient. Each intermediate result will be bound with a fresh variable name, and the use of this variable is going to be nested deeper inside the tree. For example, in listing 11, the `t5` node uses `t1` and `t4`, which are bound at a more shallow level in the tree. To construct such a tree with fresh names, continuation functions will be used, similarly to the ANF transformation algorithms done in Scheme by (Might 2008) and in OCaml by (James 2022).

Other than an `Exp` and a continuation function, symbol tables are needed. Like in `compileExp` and `evalExp`, we need a table containing mappings of source-language (SL) variable names to their corresponding `AVal`. Furthermore, to create typed Let-nodes in the target-language (TL), the symbol tables should also include types, so a type is added to the *vtab*, and an *ftab* contains the functions' types.

Along with the typed symbol tables, a helper function `getTypeExp` will be used to add types to the *vtab* when a Let binding is encountered in the SL, as well as adding a type to TL Lets for if-else statements as this type cannot be read from the syntax or any symbol tables. One final essential tool is a naming function that generates fresh variable names. This

function uses the same approach as the compiler, with a global mutable variable. See listing 13.

```
type VarTableA = Map<vname, AVal * Type>
type FunTableA = Map<fname, Type list, Type>

getTypeExp  : Exp -> VarTableA -> FunTableA -> Type
newVar      : string -> string
```

Listing 13: ANF symbol tables and helper functions

With these tools, the flattening function can take shape; a portion of its source code will be covered in this section. Its continuation function is defined as `(AVal -> ANorm)`. `flat` will wrap each sub-expression in a Let binding with a fresh name, producing atomic values that are handed over to the continuation function. The continuation then dictates how the rest of the program will be encoded, placing the values in their correct position in the surrounding ANF. See listing 14.

```
flat : Exp -> VarTableA -> FunTableA -> (AVal -> ANorm) -> ANorm
```

Listing 14: flat function

For atomic expressions, simply feed the atom to the continuation.

```
let rec flat (e : Exp) (vtab : VarTableA) (ftab : FunTableA) (k : AVal -> ANorm): ANorm =
    match e with
    | Constant (IntVal n) -> k (N n)
    | Var id -> lookupVal id vtab |> k
```

For binary operators, call recursively on each expression with a local continuation, collecting the atomic values and placing them in an intermediate computation node, wrapped in a Let binding. The outer continuation is then applied to the bound value to determine what happens next.

```
    | Plus(e1, e2) ->
      flat e1 vtab ftab (fun l ->
          flat e2 vtab ftab (fun r ->
              let t = newVar "t"
              LetA (t, Int, AddA (l, r), k (V t))))
```

For SL Let binding expressions, the bound expression is flattened to produce an atom, which, in a continuation, is added to the *vtab*, before flattening the body of the Let. When an atom is reached in the body, it is fed to the continuation.

```
    | Let(id, e1, e2) ->
      flat e1 vtab ftab (fun v1 ->
          let idtp = getTypeExp e1 vtab ftab
          let vtab1 = bindVar id (v1,idtp) vtab
          flat e2 vtab1 ftab (fun v2 -> k v2))
```

We never create a `LetA` node that binds `id`, in the flattening of a Let expression, instead it is added to the *vtab*, such that every occurrence of `id` will be immediately replaced with the atom `v1`. This is done to ensure copy and constant propagation. If `e1` was just the value `x`, then every use of `id` will be replaced by `x` (copy propagation), and the same goes for literals (constant propagation). If `e1` is a compound expression, the necessary Let chain is still generated, but the final Let for `id` is not added, as it is redundant.

For array literals and function applications, it's not possible to follow the structure of binary operations, as the number of expressions can be arbitrary. Instead, we define a recursive helper function for flattening a list of expressions.

```
flatl : Exp list -> VarTableA -> FunTableA -> (AVal list -> ANorm) -> ANorm
```

The function will initially receive a base accumulator function `(AVal list -> ANorm)`, that places a list of values into a corresponding computation wrapped in a Let, the symbol tables, and a list of expressions. Flattening of the list `e::rst` then proceeds recursively. The function will call `flat` on each expression `e` along with the continuation context that the atom `v`, will be added to the accumulator in a recursive call on the remaining list:

```
flatl rst vtab ftab (fun vs -> acc (v::vs))
```

This way, the full list of atoms will eventually be handed to the base of the accumulator. As seen below, for function calls. Array literals follow a similar structure.

```
| Apply (f, es) ->
  let letBase (vs : AVal list) =
    let tp = lookupFRet f ftab
    let t = newVar "fRes"
    let applComp = ApplyA (f, vs)
    LetA (t, tp, applComp, k (V t))
  flatl es vtab ftab letBase
```

If expressions can be handled much like binary operations, as the number of operands is fixed. First, the guard-expression is flattened, within its continuation context, each branch is flattened with its own separate continuation: `(fun v -> ValueA v)`, stopping when a value is reached. These are repackaged with the guard-atom into the computation node: `(IfA of AVal * ANorm * ANorm)`. We can optimise the code during the flattening by checking whether the guard is a literal; if that is the case, we can safely remove the branch and only flatten one of the expressions.

Map follows a similar structure, flattening the lambda's body inside the continuation context of the flattening of the applied array, halting when a value is reached. Before flattening the body however, we need to add the lambda's formal parameter to the *vtab*. The atomic array value and the flat function body is then packed into the Map computation node, wrapped in a Let, of course.

Now, we can flatten an expression `exp` as seen below, assuming there are no function calls, and that the expression is not a function body, i.e. containing no formal parameter names.

```
flat exp Map.empty Map.empty (fun v -> ValueA v)
```

In order to flatten expressions containing function calls and formal parameter names, the symbol tables must first be properly initialised. To do so, access to the entire program is

needed. As seen in listing 2, a `Prog` consists of a `FunDec` list, which we want to flatten into an `AProg` and an `AFunDec` respectively. See listing 15.

```
anfFun  : FunDec -> FunTableA -> AFunDec
anfProg : Prog -> AProg
```

Listing 15: Flattening functions for programs and function declarations

`anfProg` will go trough each `FunDec`, in order to create the *ftab*. Then the `FunDec` list is mapped with `anfFun`, passing along the *ftab* as well. `anfFun` will create fresh variable names for each formal parameter, a mapping of the SL parameter name to the TL name will be collected in a *vtab*. The two symbol tables are passed along with the expression to `flat`. The resulting ANF tree and fresh parameter names are repacked into an `AFunDec`, which is returned.

5 Analysing A-Normal form

After flattening an expression into an ANF tree, we have achieved an intermediate program structure that facilitates a simpler analysis. The goal of the translation was to detect last uses of variable names at compile time, for the purpose of reference counting. To do so, we extend our the ANF term grammar (1) with two explicit annotations: **inc** and **dec**, representing increments and decrements in the reference count, respectively. The grammar now takes the form:

$$A \to \quad V \mid \text{let } x = C \text{ in } A \mid \text{dec } x \text{ in } A \mid \text{inc } x \text{ in } A \tag{4}$$

Even though the ANF grammar states that atoms cannot be explicitly copied, there are still ways for array pointers to be indirectly copied. For example, when an array is passed to a function call `f(arr)`, or an index assignment is made for a multidimensional array `let x = arr[i]`. So, simply extending the grammar with an annotation such as

$$A \to \text{drop } x \text{ in } A$$

would be insufficient, as we also need to be able to increment the reference counters.

The correctness criteria for reference counting in this project consist of the following:

- No memory must be accessed after it has been deallocated.
- All allocated memory must be deallocated by the end of the program, excluding any returned array value.

While it would be trivial to satisfy this by placing all inc and dec operations at the end of the program, the purpose of the analysis is to determine how early deallocation can safely occur. This focus on early deallocation is what motivates the analysis. The insertion of inc and dec instructions at precise points in the program is therefore a central and distinctive aspect of this project.

5.1 Live sets

To insert these annotations, we will use chapter 8 and 10.3 from (Torben Ægidius Mogensen 2024) as a foundation for the liveness theory. However, since our ANF language differs in structure from the 3-address code used in the book, we adapt the analysis accordingly. These differences highlight some of the benefits of the more restrictive, functional grammar.

Live sets will be used to detect the last use of variable names. Similarly to how it is defined in definition 8.1 in (Torben Ægidius Mogensen 2024), A variable is *live* in an ANF tree if the value it contains might conceivably be used in any of the tree's computations. On the other hand, a variable is *dead* if its value will never be used in any of the tree's computations.

Calculating these sets in ANF is straightforward since ANF follows a flat and linear sequence of computations. Each intermediate computation is bound before use, removing nested expressions and making it clear exactly what variables are used when. Furthermore, the language contains no jumps; branches (`IfA`) and loops (`MapA`) are handled as their own self-contained ANF trees, confined to computation nodes. These are connected directly to the surrounding sequence of let-bindings, providing a predictable control flow. Thus, live sets can be calculated and used in a single tree traversal, eliminating the need for fixed-point

iteration. We do this functionally with a recursive function `analyse`. This section defines how to calculate live sets for each sub-tree in an ANF tree.

In (Torben Ægidius Mogensen 2024), live sets are calculated for each instruction i. Instead of a list of instructions, we have a tree of Let-bindings, each Let binding contains its own sub-tree (the body of the Let) and possibly more if we have an If or Map computation that also contains trees. Each sub-tree can contain an arbitrary number of branches, making it complicated to uniquely identify each sub-tree's location in a simple way. Instead, we refer to a sub-tree using the abstract identifier s. A tree s can be a leaf with a single value v, referred to as $s.v$. Alternatively, it can be a Let node of the form `let x = def in body`, where $s.x$ refers to `x`, $s.def$ refers the computation tree `def` and $s.body$ refers to the sub-tree `body`. $s.def$ is the computation c of the Let binding. It can either be a flat computation tree, such as `f(v1,..,vn)` or `v1 + v2`, or contain sub-trees s with their own let chains, such as If and Map computations. As such, s, v and c refers to (1), (2) and (3) respectively.

We will take a functional approach to calculating these live sets, and as such won't be using the standard definitions of *in* and *out*, since the ANF trees will be traversed through recursion, rather than the sequential analysis of the imperative language in (Torben Ægidius Mogensen 2024). Instead, for a tree s, we have:

- $uses(s)$, the variables live in the tree s. Return value of `analyse`.
- $usesC(c)$, the variables live in the computation c.
- $after(s)$, the variables that are live after the tree s. Parameter for `analyse`. Can be non-empty for nested trees confined to a c, empty for trees that are part of the main Let chain.

$uses(s)$ encapsulates all the live variables in s, which are needed to detect last use of variables. To generate $uses(s)$ with a call to `analyse` on the tree s, we need information from both directions:

- $uses(s.body)$, the information received from below, flows upward. The result of a recursive call to `analyse` on the tree's body. Contains the set of variables live in the body of the tree s. For a leaf, it will be initialised as $after(s)$.
- $after(s)$, the information received from above, flows downward. A parameter to `analyse` given along with the tree s by the parent. Recomputed upon entering nested sub-trees.
- $usesC(s.def)$, computed locally. The set of variables in `def`, only defined for flat computations.

Thus, calling `analyse(s, after(s))` returns `(s', uses(s))`. The function performs recursive calls on the body of the tree to obtain $uses(s.body)$, and potentially on nested sub-trees within `def`. Leaf nodes act as base cases for the recursion, receiving $after(s)$ and using it as the foundation for bottom-up calculation. For a tree s with flat Let binding, i.e. children ≤ 1, the returned live set $uses(s)$ is defined as:

$$uses(s) = \begin{cases} usesC(s.def) \cup (uses(s.body) \setminus \{s.x\}) & s \text{ is a } \texttt{let x} \text{ tree} \\ after(s) \cup \{s.v\} & s \text{ is a leaf value } v \end{cases} \tag{5}$$

Note that $\{s.v\}$ is empty if v is a literal. The parameter passed down when analysing the body remains unchanged when there is a single child.

$$after(s.body) = after(s) \tag{6}$$

For a tree s with $1 <$ children, we need to combine information from multiple sub-trees. Suppose that s is a tree where $s.def$ is of the form `If V then A1 else A2` where $s.guard$, $s.then$ and $s.else$ refers to each component respectively:

$$uses(s) = uses(s.guard) \cup uses(s.then) \cup uses(s.else) \tag{7}$$

Here, the $s.body$'s live set is used to initialise each of the two branches' bottom-up calculations, by passing it down to each branch, $after(s.body)$ remains identical to (6).

$$\begin{aligned} after(s.then) &= uses(s.body) \setminus \{s.x\} \\ after(s.else) &= uses(s.body) \setminus \{s.x\} \end{aligned} \tag{8}$$

Similarly, for a tree where $s.def$ is of the form `Map (fn arg => A, V)`, where $s.arg$, $s.loop$ and $s.arr$ refers to each component respectively, we have:

$$uses(s) = (uses(s.loop) \cup \{s.arr\}) \setminus \{s.arg\} \tag{9}$$

$$after(s.loop) = uses(s.body) \setminus \{s.x\} \tag{10}$$

The formal parameter of the lambda function should be removed before returning, as it is out of scope in the parent tree to s. It should be noted that even though this generates the correct live set for the trees, a slightly different approach is actually implemented, as loops require some extra thought for reference counting. This will be gone over in a later section.

With these formal definitions, the bi-directional information flow is established. Information flows upward to the parent s via $uses(s.body)$, computed by analysing the body of each Let-binding. Although the body of a Let-binding has no explicit reference to its parent, the call stack allows information to flow upward as calls return. Meanwhile, $after(s)$ flows downward, used to initialise the bottom-up calculation of live sets, ensuring correct propagation of variables live beyond branching sub-trees. Calls on bodies of Let-bindings pass along the same $after(s)$ set received from the parent, while calls to nested sub-trees within a computation pass along the parents $uses(s.body)$ set.

Thus, the upward flow ($uses$ sets) is computed dynamically during recursive returns. However, the downward flow ($after$ sets) is only updated when entering nested sub-trees within computations. As an example, let's take a look at the nested sub-tree s_1, in this program:

```
let t = 1 + 1 in
let a = {1,2,3} in
let b = map(fn x =>
            let c = x + t in    <- s1
            c, a) in
let d = b[t] in
d
```

In this example, s_1 is the root of the nested sub-tree within the `def` of the Let-binding `let b`. It receives an $after(s)$ set containing $\{t\}$ as defined in (10), which are the variables live after s_1 finishes executing.

Without downward information flow (via the $after(s)$ set), the analysis would incorrectly determine that `t` is last used within s_1. For arrays, this would cause premature freeing of memory. The flow of information is illustrated in the graph in figure 3. Note that (*) denotes

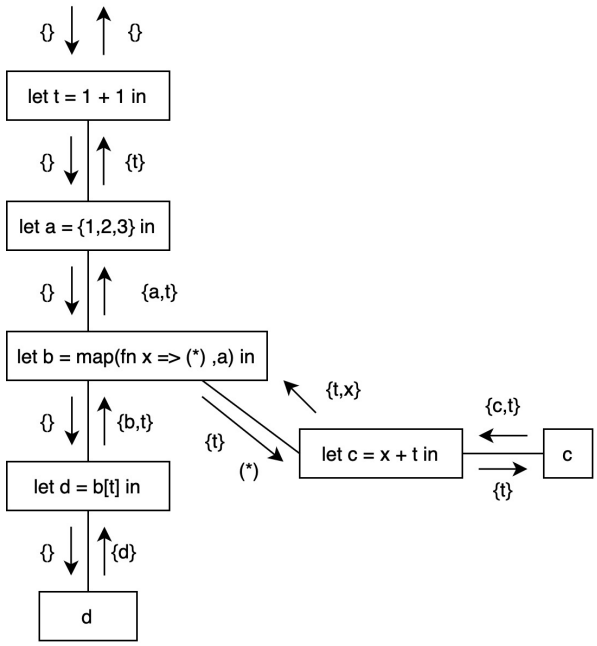


Figure 3: Example of information flow in ANF analysis

the body of the lambda function, which is the nested sub-tree s_1.

Using the functional flow of information, we can go down and up through the tree while only visiting every sub-tree once. All the information we need will be available at this visit, meaning we can analyse, annotate and optimise each sub-tree in one traversal.

5.2 Dead binding elimination

While traversing the tree, we can make use of the liveness information to perform optimisations on the tree. As described in chapter 10.3 in (Torben Ægidius Mogensen 2024), if a variable `x` is dead in the body of the tree `let x = def in body`, and `def` contains no side effects, then the binding can be removed from the tree, simply giving us `body`. By applying this optimisation during the analysis, we have to rethink the formal definitions of $uses(s)$ for all non-leaf trees:

$$uses(s) = \begin{cases} \text{See (5), (7) or (9),} & s.x \in uses(s.body) \text{ or } \texttt{def} \text{ has side effects} \\ uses(s.body), & s.x \notin uses(s.body) \text{ and } \texttt{def} \text{ has no side effects} \end{cases} \tag{11}$$

Thus, if the binding is removed from the tree, we should just return to the parent the same set we received from below, i.e. $uses(s.body)$. Otherwise, use the previously established definitions. The set passed to the body $after(s.body)$ remains the same as (6), but no sets should be passed to any other branches, since the Let binding is removed.

As stated, the ANF grammar does not include any of Fasto's I/O functions, yet it is analysed as though they still exist to avoid eliminating side effects. Therefore, any function

calls cannot be eliminated, since it's unknown whether or not the function has any I/O. One might also view memory-accessing computations as functions with side effects, even though Fasto arrays are immutable, since memory access can fail. However, they will not be treated as such in this project.

In addition to dead binding elimination for let bindings, we can do something similar with leaf nodes. To illustrate this, let's look at an example, where s_1 is the atom `b` in a nested tree inside the `let a` tree s_2:

```
let x = ... in
let g = ... in
let a = if g then       <- s2
        let b = x + g in
        b               <- s1
    else
        let c = <I/O> in
        c
in
g
```

With the current definitions of (5) and (8), we have:

$$uses(s_1) = after(s_1) \cup \{b\} = \{b, g\}, \text{ where } after(s_1) = uses(s_2.body) = \{g\}.$$

Removing the s_2 binding from the tree should not be done due to the I/O of its Else branch. However, it is clear that we can optimise the program by finding a way to trivialise the If branch. One solution would be to replace the branch with the atom 0, so `b` is never added to the live set. With the current definition (5), this will not happen. `b` will be added to the live set $uses(s_1)$, even though its value is never used. If instead, the set did not add the atom, giving us $uses(s_1) = \{g\}$, then there would be a cascading effect, letting us remove the binding `let b`. Which in turn, as (11) defines, `x` would also not be added to the live sets, making it clear for the analysing algorithm that the binding `let x` can also be removed. The same logic also applies in situations where the leaf is not inside a nested tree, but we will always assume that the resulting variable in the main tree will be used.

This optimisation is only possible due to the fact that $a \notin uses(s_2.body)$, i.e. s_2 is a dead binding, meaning that the value within `b` will never be used. Whereas in the example of Figure 3, the map is not a dead binding. This information needs to join the downward flow of the $after$ sets. Therefore, we define the flag $used(s)$, which is true if the resulting value of s's tree will be used, i.e. is live in the grander Let chain that s is confined to, and false otherwise. This flag is reevaluated upon entering a nested sub-tree, just like $after$, and is passed along down the tree, where its information will be used in the leaf, also similar to $after$. With this flag, we can formally define the $uses(s)$ set for leaves to return as the following:

$$uses(s) = \begin{cases} after(s) \cup \{s.v\}, & used(s) \\ after(s), & \text{not } used(s) \end{cases} \tag{12}$$

It should be noted that in the previous example, the optimisation would also not be possible if `b` was defined outside the branch and still live after the nested tree finished executing. Of course, the branch could still return 0, but b still has to stay live, in order to prevent the cascading effect of removing the binding nodes of `x` and `b`. The definition of $uses(s)$ will

correctly handle this, as in this case $b \in after(s)$, so there is no need to add it to the live set, as it's already present. So even though the cascading optimisation cannot be performed, it's fine that we don't add v to the set when $used(s)$ is false.

Using these definitions for the *uses* sets of leaves and let bindings, it is still possible to perform the live set generation in a single tree traversal. To do so, the *used* flag and $after$ sets must be re-computed upon entering a nested sub-tree, and passed along until the atom is reached. From here, they will be used to create the base of the *uses* set, which will be built upon in the bottom-up calculation of each node's live sets. Allowing us to analyse the code and optimise it in one go.

5.3 Calculating live sets

We can follow the formal definitions to calculate the live sets and apply the optimisations. We need to define a live set type, along with two helper functions:

```
LiveSet     : Set<vname>
cIO         : AComp -> bool
isFlatComp  : AComp -> bool
getVarsComp : AComp -> LiveSet
```

Listing 16: Tools for live sets

`LiveSet` is simply a set of variable names. `cIO` will see if a computation node contains any possible I/O, by seeing if any `ApplyA` computations are present. For `IfA` and `MapA` computations, it will traverse the entire nested tree(s), stopping if a function call is found. Similarly, `isFlatComp` will check if the computation contains any sub-trees, as they should be handled differently. Finally, `getVarsComp` is used to calculate the $usesC(s.def)$ set. If C is a flat computation, it will simply return the variable names used; if C contains a nested sub-tree such as `IfA` or a `MapA`, it will also return all free variables within the computation. The reason they remain defined for `IfA` and `MapA`, even though they don't appear in (7) and (9), is that we will later need to find all free variables in a sub-tree, to properly analyse loops.

Now we can define the `analyse` function, see listing 17. Here, the live sets are only used for dead binding elimination and nothing else, but this will be extended later.

```
let rec analyse (a : ANorm) (usesAfter : LiveSet) (useFlag : bool) : ANorm, LiveSet =
  match a with
  | ValueA (N _) -> a, usesAfter
  | ValueA (V x) -> if useFlag then a, usesAfter.Add X else ValueA (N 0), usesAfter
  | LetA (id, tp, c, body) ->
      let body1, usesBody = analyse body usesAfter
      let deadBinding = not (usesBody.Contains id)
      if deadBinding && not (cIO c) then
        body1, usesBody
      elif isFlatComp c then
        let usesC = getVarsComp c
        let uses = Set.union usesC (usesBody |> Set.remove id)
        LetA (id, tp, c, body1), uses
      else failwith "branches not yet supported"
```

Listing 17: Analyse function for dead-binding elimination

The function will return the $uses(s)$ set as defined in (5), for the current tree along with an optimised version of said tree. It takes an ANF tree, the $after(s)$ set, and the use-flag $used(s)$. For a non-literal atom, we apply the definition (12), only adding `x` to the set if $used(s)$ is true. Otherwise, we don't need to change the set, and we can replace the atom with 0.

For a let binding, we call recursively on the body, receiving $uses(s.body)$, if $id \notin uses(s.body)$ and there are no side effects, we can remove the binding. Otherwise, compute $usesC(s.def)$ with `getVarsComp`, and calculate and return $uses(s)$ along with the optimised ANF tree.

5.4 Annotating the tree

Since we can calculate the live sets for each sub-tree in one traversal of the tree, we can also insert the annotations in the same sweep. All the information needed to insert these instructions is present in the sets $after(s)$ and $uses(s.body)$. Initially, we will start by just analysing flat computations, saving nested sub-trees a bit for now, so $after(s)$ will remain empty.

Let's start by extending `analyse` to insert decrements correctly, pretending that increments are not needed, that means we assume that the tree only contains flat computations, no function calls, and only one-dimensional arrays. We can then define the set of dead variables $dead(s)$ in a tree s as the variables that are used for the last time in the computation of the root, i.e. in the `def` of `let x = def in body`. A variable is used for the last time if it is present in $usesC(s.def)$, but not in $uses(s.body)$.

$$dead(s) = usesC(s.def) \setminus uses(s.body) \tag{13}$$

In other words, at runtime, the variables in $dead(s)$ will never be accessed again once the tree's body s has been entered, since they don't exist in $uses(s.body)$. By only decrementing dead variables, we can ensure that we satisfy the correctness criteria of never accessing freed memory. Furthermore, for flat computations, inserting decrement annotations for variables from $dead(s)$ right around the tree's body will ensure they are freed as early as possible. When `def` contains a tree (branch), this is not optimal, and the tree should instead be analysed as

well. Now we can extend the `analyse` function accordingly, but first, we have to extend the MiniFasto `ANorm` type to contain the annotations defined in (4):

```
and ANorm =
  | ValueA of AVal
  | LetA of vname * Type * AComp * ANorm
  | IncA of vname  * ANorm
  | DecA of vname * int * ANorm
```

Listing 18: ANF with annotations

Since Fasto arrays can be multidimensional, we also add an integer to `DecA`, which corresponds to the dimensions of the array that should be decremented. This is done such that the code generator will correctly decrement any possible sub-arrays if the parent array is freed, since the sub-arrays will lose a reference (pointer). But for now, as we assume that all arrays are one-dimensional, this will always be 1. Next, we define some extra tools for inserting these decrements, see listing 19.

```
TypeTable  : Map<vname,Type>
insertDecs : LiveSet -> TypeTable -> ANorm -> ANorm
insertIncs : LiveSet -> TypeTable -> ANorm -> ANorm
```

Listing 19: Tools for decrementing

The symbol table `TypeTable` is a Map of ANF variable names and their corresponding types. It is given to `insertDecs`, or `insertIncs` along with a `LiveSet` of variables to annotate and an ANF tree that the annotations will be wrapped around. It will look up the variables in the type table, only inserting annotations for arrays, and also adding the number of dimensions to decrement annotations. The symbol table will join the *used* flag and the $after$ set in the downward information flow as parameters to `analyse`, updating the table at each Let binding before passing it along to recursive calls. Now we can extend the case of Let binding in the body of `analyse`:

```
let rec analyse (a : ANorm) (usesAfter : LiveSet) (ttab : TypeTable)
                (useFlag : bool) : ANorm * LiveSet =
  match a with
  ...
  | LetA (id, tp, c, body) ->
      let ttab1 = Map.add id tp ttab
      let body1, usesBody = analyse body usesAfter ttab1 useFlag
      let deadBinding = not (usesBody.Contains id)
      if deadBinding && not (cIO c) then
        body1, usesBody
      elif isFlatComp c then
        let usesC = getVarsComp c
        let dead = Set.difference usesC usesBody
        let uses = Set.union usesC (usesBody |> Set.remove id)
        match c with
        | LenA(_) | AddA(_,_) | IndexA(_,_,_) -> //Only for index in 1d array
```

```
            let decbody = insertDecs dead ttab1 body1
            LetA (id, tp, c, decbody) , uses
          | _ -> failwith "not yet implemented"
        else failwith "branches not yet supported"
```

We define this straightforward approach only for flat computations that do not allow copying array references. Here, we can simply place decrement annotations around the body of the tree and return that along with the *uses* set. Some extra steps are required for the remaining computations, along with `IndexA` for multi-dimensional arrays.

5.4.1 Adding increments

Before we extend the `analyse` function, we should determine what steps are necessary to achieve a resulting ANF tree that will not break any of the correctness criteria, while still attempting to free the variables sooner rather than later. Let's change our earlier assumption that the ANF tree has no branches or multidimensional array, and assume that there are not branches, but multidimensional arrays can exist.

Now, let's define the scenarios in which array pointers can be copied. As stated, the syntax does not allow direct copying, but it can still happen indirectly. As we have a limited ANF syntax, we can quite easily define each transformation where a reference is copied. Most obvious is the three computations:

```
ApplyA (f, args) | ArrLitA (vals, t) | IndexA (arr, v, t)
```

Lets view an ANF program that makes use of all three of these computations, see Listing 20. The program is displayed on the left, the `usesBody` set returned from a recursive call to the

```
   program                 usesBody       {1,2,3} ref count

1  let a1 = {1,2,3} in     {a1}            1,   init in a1.
2  let a2 = {a1,a1} in     {a2}            2,   copied twice but a1 last use.
3  let a3 = a2[0] in       {a2,a3}         3,   copied again into a3.
4  let b  = f(a3,a2) in    {a3,b}          4/1, copied a3 and a2, a2 last use.
5  let c  = a3[b] in       {c}             0,   a3 last use.
6  c                       {}              0
```

Listing 20: ANF example, incrementing.

body is displayed in the middle, and the reference count (RC) of the initially allocated array, let's call it `arr1`, is displayed on the right. When the analyser finds a variable in the tree's computation that is not present in `usesBody`, it should decrement the variable, but it should also insert increments when a variable is copied.

Initially, the array `arr1` is allocated with a ref-count of 1, placing its pointer in `a1`. Defining `a2` copies the pointer twice (RC→3), but since `a1` is now dead (not present in `usesBody`), the array is decremented (RC→2). Next, at line 3, `a3` is defined using an index operation in the 2D-array `a2` copying `arr1`'s pointer, (RC→3). At line 4, there is a function call, `arr1` is passed along with the 2D-array. In the scope of `f`, one reference to `arr1`, and the final reference to the 2D-array exist, as such (RC→4) upon calling the function. Assuming that the function returns an integer, based on the result being used as such in line 5, only the reference in `a3` remains after the `f` returns. Inside `f`, the copied reference to `arr1` dies, and the 2D-array

containing two references to `arr1` is freed (RC→1). At line 5 the final reference to the array dies, (RC→0). `arr1` is freed.

The example illustrates how the number of array references can increase and decrease, highlighting some key design choices for function calls:

- Caller increments array arguments before a function call.
- Callee decrements its copies before returning.
- Caller maintains responsibility of the returned value.

Another thing to be learned from the example is that there are similarities in function calls and array literals, as they both need to increment variables and may require multiple increments. Lets try to view array literals as function calls:

```
{a1,a1} <-> mkArr(a1,a1)
```

Here, the function simply places the values in an array and returns. The function does not decrement or increment anything, cause it's not actually a function. Still, the same logic as for function calls applies, as the caller keeps responsibility for incrementing arguments and decrementing return values.

Since we treat array literals and functions as identical, let's compare them in the example above. At line 2 and line 4, the ref count is only incremented by one even though the number of references copied differs. This is because in line 2, `a1` is dead in the Let's body, whereas in line 5, `a3` remains alive. So, in line 2, it should actually be incremented twice and decremented once; an alternative would be to increment it once. So for function calls (and array literals), each variable `x`, passed as an argument to a function `n` times, should be incremented `n` times, minus one if it is dead in the body of the Let. We never have to add any decrements after function calls, as `n` cannot be negative. If it is only passed one time, and dead after, there is no need to increment or decrement, as the only live reference is handed to the caller, which now has responsibility for it.

Now we can extend the computation match case in the`analyse` function for flat computations, to correctly annotate the function calls and array literals.

```
...
match c with
| ApplyA (_,args) | ArrLitA (args,_) ->
  let incrLet =
    args
    |> List.fold (fun lst v -> match v with
                               | V vn -> vn::lst
                               | N _ -> lst) []
    |> List.fold (fun map arg ->
        map |> Map.change arg (function
          | Some n -> Some (n+1)
          | None -> Some 1)) Map.empty
    |> Map.map (fun arg n -> if dead.Contains arg then n-1 else n)
    |> Map.fold (fun lst arg n -> lst @ List.replicate n arg) []
    |> List.fold (fun A v -> insertIncs (Set.singleton v) ttab1 A) (LetA (id, tp, c, body1))
  incrLet, uses
...
```

To correctly increment `n` times, minus one if the variable is dead, we start by filtering out all literals. Then we create a map that contains the number of times each function argument is repeated. Then we subtract one from each of the dead variables. We turn it back into a list, where each argument is replicated, corresponding to the count in the map. An increment is placed around the Let binding for each variable name in the list to ensure we increment *before* function calls. Finally, the new tree and the `uses` set are returned.

As was clear from the example in Listing 20, index computations in multi-dimensional arrays also require an increment. This is very straightforward.

```
...
match c with
| IndexA(_,_,t) when t<>Int ->
  let incrBody = IncA(id, insertDecs dead ttab1 body1)
  LetA (id, tp, c, incrBody), uses
...
```

To avoid accessing the array and finding out what pointer is copied, we can simply increment the bound variable of the Let. This will contain a pointer to the correct array whose reference count should be incremented at runtime.

There is one final, less obvious case where it is necessary to place increment annotations. Let's reject the previous assumption that there are no branches in the tree, and instead assume that there are, and that our `analyse` function will get called recursively on any branches it encounters. This raises another case we need to handle. The variable in the leaf of a sub-tree within a computation node will be copied into the bound variable of the computation's Let. For example, in the following program:

```
1  let a = {1,2,3} in
2  let b = map(fn x => a,a) in
3  let c = if .. then
4              let d = {a,a,a} in
5              d
6          else
7              b
8  in
9  let e = {b,c} in
10 e
```

At the second line of the program, there is a nested sub-tree consisting of a single leaf `a`, inside the map computation. The value contained within `a` will be copied to each element in the new array `b`. The reference count of the array that `a` points to should therefore be increased 3 times (the length of `b`). To do so, we can place a single increment annotation around the leaf such that it gets incremented at each iteration of the loop at runtime. Similarly, with the if computation at line 3, the *else* branch returns the array `b` that should also be incremented. It should be noted, however, that if either the bound variable `c` or the copied variable `b` was not live in the body of `c`'s let (line 8-10), then the increment would not be necessary. This applies to the *then* branch, which returns a locally scoped array `d`, that is not live in the body of `let c`, and thus there is no need to increment or decrement as the reference to `d`'s array is passed along before `d` dies. Likewise, for the map computation at line 2, if `a` was not live in the tree's body, only three references to its array would exist in the body, all from within `b`, as `a` should be decremented. As such, a more precise approach would be to increment it twice,

but this is not straightforward to implement. This is because a single increment annotation in a map will increment the array at each iteration of the loop at runtime, exemplifying how reference counting is a mix of static (compile-time) and dynamic (run-time) information.

From the example, we can gather that the variables to be incremented are the leaves of nested sub-trees. Furthermore, whether or not they should be incremented depends on information received from above:

- Is the result of the sub-tree used? i.e. is the bound variable live in the body of the tree? Information obtained from the `useFlag`.
- Is the leaf variable x live in the body of the tree, whose sub-tree it is confined to? i.e. is $x \in$ `usesAfter`?

This gives us 4 different scenarios:

	`useFlag = true`	`useFlag = false`
$x \in$ `usesAfter`	Increment the array	Replace leaf with 0.
$x \notin$ `usesAfter`	add x to live set	Replace leaf with 0.

In the scenario where the result of the sub-tree is used, the array's ref count should be incremented if the leaf's variable is also present in the `usesAfter` set. If it is not present, we should add the leaf's variable to the live set before returning, to ensure it stays live in the sub-tree. If the `useFlag` is false, we can simply replace the leaf with 0, since the result is never used. In this scenario, the branch will be removed from the tree if no side effects are present, so this optimisation only occurs if the binding is not already optimised away. There is no need to change the `usesAfter` set before returning it, if it's not already present, it is fine as the leaf is changed to 0.

We can now change the `analyse` case for leaves, to match the logic we have defined:

```
match a with
..
| ValueA (V x) ->
  if useFlag then
    let uses = usesAfter.Add x (* May already be present *)
    match Map.tryFind x ttab with
    | Some tp when tp <> Int && usesAfter.Contains x -> IncA (x,a), uses
    | None -> failwith (sprintf "Unkown variable in ttab in analyse: %s" x)
    | _ -> a, uses
  else
    ValueA (N 0), usesAfter
..
```

5.4.2 Analysing branches

In MiniFasto's ANF language, trees can currently branch out of the main Let chain using If-else and Map computations. As the control flow of branches is underdetermined at compile time, they require well-thought-out annotations that will yield the correct reference count at run time. The copying of the leaf variable for these sub-trees has already been handled, but decrementing also requires extra steps. We can build upon much of the logic we have already established to analyse them properly. First, let's extend `analyse` for If-else computations,

going down the else branch of the predicate `isFlatComp c`. We separate these into two branches, so we don't call `getVarsComp` on computations that don't need them, as they would unnecessarily traverse all the sub-trees, giving us the structure:

```
let rec analyse (a : ANorm) (usesAfter : LiveSet) (ttab : TypeTable)
                (useFlag : bool) : ANorm * LiveSet =
  match a with
  ..
  | LetA (id, tp, c, body) ->
    ..
    if deadBinding && not (cIO c) then
      body1, usesBody
    elif isFlatComp c then
      let usesC = getVarsComp c
      let dead = Set.difference usesC usesBody
      let uses = Set.union usesC (usesBody |> Set.remove id)
      match c with
       <Handle flat computations>
    else
      match c with
        <Handle computations with sub-trees>
```

If-else branches have to be handled with extra care than other computations. For example, let's say that a variable `x` is used for the last time in the *then* branch of an if, but not referenced at all in the *else* branch. With the current logic of our `analyse` function, the variable will only be decremented in the *then* branch, as the analyser will only insert a decrement when a variable is encountered if it is not in the live set of the Let's body. But no decrement will be placed in the *else* branch. Thus, if the run-time environment never goes down the branch that contains the last use of the variable, it will never reach the decrement annotation, and the ref count of the array will be wrong. As such, inserting a decrement in *both* branches for any variable used for the last time in any branch is necessary. Let's look at the implementation:

```
  match c with
  | IfA (g,b1,b2) ->
    let b1body, usesB1 = analyse b1 (usesBody.Remove id) ttab1 (not deadBinding)
    let b2body, usesB2 = analyse b2 (usesBody.Remove id) ttab1 (not deadBinding)

    let b1dead = Set.difference usesB1 usesB2
    let b2dead = Set.difference usesB2 usesB1
    let b1dec = insertDecs b2dead ttab1 b1body
    let b2dec = insertDecs b1dead ttab1 b2body

    let decIf = IfA (g, b1dec, b2dec)
    let uses = Set.union usesB1 usesB2 |> Set.union (varsAVal g)
    LetA (id, tp, decIf, body1), uses
    ..
```

The analyser will start by recursively analysing the two branches. As defined in 8, the live set to be passed downward should be the `usesBody` set without the bound variable `id`, to ensure correct initialisation of the bottom-up calculation of the live sets in the branches. Furthermore, the `useFlag` is the negation of the bool `deadBinding`, stating whether the bound value *id* of the Let is dead in the tree's body. Next, we collect the variables the differences of

the two returned *uses* sets. The two set of `usesBody` used to initialise both branches' live sets will cancel each other out, leaving the decremented variables of each branch. A decrement annotation is then inserted right around the root of the branches, for each variable in the corresponding set. The branches are wrapped in the computation union, and returned along with the `uses` set, computed as defined in (7). Note that we never insert any decrements for the guard `g`, as this can never be an array, since we assume that the program is well typed.

We have the opposite issue for map computation: arrays are decremented too many times. If an array has its last reference in the body of the lambda function passed to Map, the analyser will place a decrement annotation inside the lambda body. At run time, since maps are handled imperatively, the decrement will be encountered at each iteration of the loop, decrementing the variable multiple times. Instead, we can place all the necessary decrements outside the lambda function's body, such that they are decremented the right number of times. To ensure nothing gets decremented, we can use `getVarsComp` to gather all the free variables of the lambda's body, and add it to the $after(s.loop)$ set.

But what about the lambda's parameter? In the case of the `ApplyA` computation, we increment the reference counters of the parameters, and leave the rest to the callee, maintaining responsibility only for the returned value. Here, we can avoid an unnecessary inc/dec pair, since the variable is locally scoped. The parameter is a borrowed reference. It's limited to the duration of an iteration, so ownership never leaves the loop. We are not decrementing any free variables in the loop, we can do the same for the parameter. However, if the variable is "returned" at the leaf of the lambda's body, it should be incremented. As we defined earlier, this only happens if the variable is a part of the $after$ set, which the parameter will never be because it is locally scoped. But since we add `getVarsComp` to $after(s.loop)$ before analysing the body, this is handled. This approach might seem unintuitive, but it's simpler to implement, and possibly removes a superfluous inc/dec pair.

```
match c with
  let usesC = getVarsComp c
  let ttab2 = ttab1 |> Map.add arg argtp
  let fbody1, usesLoop = analyse fbody (Set.union (usesBody |> Set.remove id) usesC)
                         ttab2 (not deadBinding)

  let dead = Set.difference usesC usesBody
  let deadMap = dead |> Set.remove arg
  let decbodyMap = insertDecs deadMap ttab2 body1
  let mapComp = MapA(Param(arg,argtp), fbody1, arr, arInT, arOutT)

  let uses = Set.union usesLoop (varsAVal arr)  |> Set.remove arg
  LetA(id, tp, mapComp, decbodyMap), uses
```

Before analysing the body of the function, we extend the symbol table to include the lambda's parameter. Then we call recursively on the body. We provide `(usesBody - id)` $\cup$ `usesC` as the $after$ set. `usesC` includes all free variables of the lambda's body, ensuring nothing gets decremented. As stated, the lambda's parameter is intentionally also a part of this set (assuming it is used), so it correctly gets incremented if it is the leaf of the tree. The `useFlag` is computed as it was for If computations. Next, we place decrements for each of the dead variables in the body, ensuring to remove the parameter, as it is out of scope, and as we didn't initially increment it. Finally, we piece the tree back together, returning it along with the *uses* set.

Now the analyser will correctly handle If-Else and Map computations, which in more broad terms means that we have understood how branches and loops should be handled in the Fasto language. It is fair to assume that we can apply the same logic to the remaining expressions from Fasto that are not included in MiniFasto. With more time, the approach to loops might have been optimised, as this implementation decrements a bit later than optimal in some cases.

5.4.3 Analysing functions

The `analyse` function will now correctly place annotations for all the locally defined variables in a tree. However, when it encounters function parameters, it will be unable to look them up in the type table. As such, we need to initialise the type table with the function parameters before calling the analyser.

Another issue that If-Else branches similarly had is that unused variables will not be decremented. This is problematic, as we defined in section 5.4.1 that the function is responsible for decrementing all of the parameters. We can take a similar approach to If branches, by adding decrements right in the beginning of the function body, for any parameters that are not present in the returned live set from the finished analysis.

Let's extend the `anfFun` function, defined in Listing 15. When we are creating the fresh variable names and making the variable table for the `flat` function, we can also create the type table for `analyse`. After flattening the function, we can also make a call to `analyse`, to insert annotations. Finally, we can insert decrements for any unused variables. `anfProg` remains unchanged, but now returns an annotated program.

6 From ANF to RISC-V

The next step is generating machine code from the ANF tree. The ANF syntax is structurally different from the original `Exp` syntax, but the underlying computations remain the same. Consequently, we can repurpose the existing `compileExp` function with minimal changes to produce a working RISC-V code generator for the ANF tree. We define the following functions:

```
genVal  : AVal -> VarTableC -> reg -> PseudoRV list
genComp : AComp -> VarTableC -> reg -> PseudoRV list
gen     : ANorm -> VarTableC -> reg -> PseudoRV list
genArgs : Param list -> VarTableC -> int -> reg list -> string
                                        -> (PseudoRV list * reg list * VarTableC)
genFun  : AFunDec -> (string * (reg list * PseudoRV list))
genProg : AProg -> RVProg
```

Listing 21: Gen functions

`genVal`, `genComp` and `gen`, generator code for each of the three different unions in the ANF syntax from Listing 12. The implementation of these if very close to the original compiler, the only thing to note is that when `gen` encounters one of the Inc/Dec annotations, it will place one of the newly added `PseudoRV` types:

```
type PseudoRV =
  ..
  | INC   of reg
  | DEC   of reg*imm
```

These are added to the program as pseudo instructions, to keep things simple.

`genFun` and `genProg` also works similarly to the old compiler, with the exception of the use of the new function `genArgs`, called from within `genFun`. This function creates the variable table, but also a short function prologue that moves the parameters into their correct registers to ensure recursion works properly.

With this, the finished pipeline becomes `flat -> analyse -> gen`. Flattening the Mini-Fasto program to ANF, placing annotations, and generating machine code. As touched upon before, reference counting is a mix of compile-time and run-time information. So, another important aspect is how the simulator should handle these pseudo RISC-V instructions. We can start by extending the heap map defined in Listing 7, to include a reference counter:

```
type Heap = Map<int,(int * array<int>)>
```

When the simulator allocates an array, it should be initialised to 1. When a `DEC` instruction is encountered, this count should be decremented, and if it reaches 0, the array should be freed. If an array gets freed, we also need to decrement each of the nested sub-arrays, as they each lose a reference. For this use we have the immediate in the instruction that corresponds to the dimensionality of the array. So we correctly cascade the decrementing effect for all possible sub-arrays and their possible sub-arrays, etc. Increment is very straightforward, simply adding 1 to the counter.

7 Testing

To validate the correctness of the reference counting in MiniFasto through extensive testing. For each program, we test 3 things:

1. Verify the correctness of the compiled and analysed program by comparing results with the MiniFasto interpreter.
2. Ensure no memory block is accessed after being deallocated.
3. Ensure that the heap is empty when the program finishes executing, after possibly freeing the returned value.

The simulator will raise a fail flag if any unallocated heap address has been accessed. And a small testing library will compare results with an interpreter, and check that the heap is empty. The testing library consists of the following functions

```
dimsToType  : int -> Type
rebuildCRes : int -> Heap -> int -> Value
compareVal  : Value -> Value -> bool
countArrs   : Value -> int
ppValue     : Value -> string
runTestA    : Prog -> string -> unit
```

In short, the testing function `runTestA` will run the program using the `flat -> analyse -> gen` pipeline, and with the interpreter. Since the simulator will return a single int and the mutated heap, the tester uses `rebuildCRes` to rebuild the result into a MiniFasto `Value`. Next, if the result was an array, we run a small program in the simulator to decrement the pointer in the heap. Then we can check if the heap is empty or not. Finally, we can compare the two results using the rebuilt `Value` with the interpreters `Value`.

A suite of test cases covering numerous tricky scenarios that stress the reference counting analysis was developed to ensure robust testing. These tests reflect many of the examples we reviewed during the earlier sections of the analyser function. The test suite includes over 50 MiniFasto programs that cover cases such as:

- **Repeated function arguments.** Numerous tests have been made that contain function calls with repeated arguments of arrays with varied dimensionality.
- **Last use of variables in If-Else branches.** Using a variable for the last time in only one of the branches, requires the other branch to still decrement the array.
- **Return of free variable in a branch.** If the leaf of a branch is a variable from the outer scope, live after the branch, and the branch is not a dead binding, the variable must be incremented.
- **Arrays of different dimensions.** Programs that include branches, index computations and function calls of arrays up to 4 dimensions have been tested.
- **Map with references to variables from the outer scope in its lambda function.** This scenario tests that the analysis correctly handles closures capturing references from an enclosing scope.

The full test suite can be found in the file `Fasto-memory/miniFasto/Program.fs`. To run the tests, navigate to the `miniFasto` folder and type `dotnet run`. All tests were executed successfully, giving correct results, an empty heap, and no failed memory access. This, coupled with the reasoning behind the implementation of the analyser, gives confidence that the implementation behaves correctly in practice. While a full formal proof of correctness is beyond the scope of this project, we have argued intuitively why the reference counting should be correct, giving substance to the intuition with empirical evidence.

What remains to be proven is the effectiveness of our analysis. Does it deallocate arrays at the earliest program point possible? The short answer is no, which should be clear from the previous sections. Chasing the perfect spot to place these annotations would need whole-program reasoning and possibly more work at run-time. Instead, this project keeps a simpler, single-pass algorithm that still follows the previously defined correctness criteria. Based on the design of the algorithm, it should, intuitively, perform exact reference counting for flat programs and inside branches that have explicit merge points, which is always the case in ANF. This is not easy to test, and therefore remains to be a statement based on intuition. The obvious problem occurs when variables flow through higher-order constructs such as the Map, where arrays may live longer than they should, since annotations are never placed within the loop. In short, reference counting was successfully implemented in MiniFasto, not deallocating as early as possible, but as early as is practical.

8 Conclusion

8.1 Summary and preliminary insights

The work carried out in MiniFasto provides a foundation for implementing reference counting in the full Fasto language. We designed and implemented a memory management system based on reference counting for a simplified subset of Fasto called MiniFasto, including a MiniFasto interpreter and compiler to pseudo-RISC-V. The core of our approach was to translate MiniFasto programs into A-Normal Form (ANF), flattening nested expressions into a linear chain of let-bindings. This greatly simplified variable liveness analysis: with ANF, it became straightforward to determine when values (especially arrays) are no longer needed and should have their reference count decremented. We formally defined the live sets $uses(s)$ and $after$ to show the bi-directional information flow of a recursive analyser that scans the ANF syntax tree to insert reference-count annotations as accurately as possible. The thesis presented the design and implementation of these components in detail, along with their integration into the MiniFasto toolchain. Testing was performed to validate the correctness of the implementation, demonstrating that MiniFasto programs run with the new reference counting system produce correct results with no memory leaks or premature deallocations. This empirical evidence, combined with a design analysis, gives confidence that the reference counting approach works correctly in practice.

Beyond achieving the initial goal, this project yielded several insights and lessons that extend beyond our initial expectations. Firstly, while the ANF transformation proved essential for clarity, we encountered practical challenges in implementing it for all language constructs. Converting arbitrary nested Fasto expressions into ANF required handling complex expression trees and control-flow structures using continuation functions. However, the effort made program analysis more straightforward. Analysing deeply nested or branched expressions directly is difficult, whereas once transformed into ANF, those same computations become a linear sequence with explicit merge points, making it much easier to pinpoint last uses of variables. This highlighted the critical importance of ANF in enabling precise memory management analysis.

Annotating the ANF tree highlighted the mix of static (compile-time) and dynamic (runtime) information needed to implement reference counting. For example, when dealing with higher-order array operations like map (loops) some variables had to be incremented at each iteration of a loop, while others had to wait for the loop to end. For conditional branches, additional, less trivial annotations were also needed. This ensured correctness, but for loops, we sometimes incremented more than needed, forcing additional decrements, and at times decremented later than optimally possible. Theoretically, this could be optimised with a deeper analysis, but implementing this precisely turned out to be non-trivial, since an annotation in a loop will always be executed at each iteration. Chasing an optimal placement of reference-count updates (freeing memory at the earliest possible point) would require more global program analysis or added runtime complexity, which was beyond our scope. Instead, we adopted a simpler single-pass analysis that meets the correctness criteria and frees memory as early as practical, even if not always at the theoretical earliest moment. These insights particularly regarding the trade-offs between static compile-time analysis and dynamic runtime counting were enlightening and went beyond what we anticipated at the projects outset. They highlight benefits and trade-offs of managing lifetimes in a functional language that is imperatively executed, and the value of a robust intermediate representation (ANF) in

navigating that complexity.

8.2 Next steps

Finally, we can outline the next steps to transition from the MiniFasto prototype to a fully integrated solution in the Fasto compiler.

First, the current Fasto compiler emits code that uses a simple incremental heap allocator without freeing memory; this must be replaced or modified to call our custom memory allocator written in C (compiled to RISC-V via CLANG). Furthermore, it should be extended to incorporate the generated Inc and Dec operations at the appropriate points. In practice, this means altering the compiler to emit calls to special increment and decrement functions when annotations are encountered in the program. These Inc/Dec functions could be written in C, extending `mem.c`. Furthermore, the heap data structures should include a reference count field the prototype allocator was designed for this, but the field is not yet in use in Fastos runtime.

Second, we must generalise the liveness analysis and annotation pipeline developed for MiniFasto to handle all Fasto language features. MiniFasto captured the essential semantics of Fasto, but the full language includes additional constructs that were not part of the subset. Intuitively, we expect that implementing these should be somewhat trivial if we apply the knowledge gained in this project. Still, the ANF transformation and analysis will need to be reviewed and extended to ensure it covers all of these constructs. For ease of implementation, we should keep the original syntax of Fasto rather than defining an entirely new ANF syntax like we did for MiniFasto. Instead, we can flatten the expressions, following the non-nested restrictions of ANF, but keeping the types the same.

Third, after updating the compiler and analysis, the integrated system must be thoroughly tested and optimised within the real Fasto environment. An in-depth testing suite of complex Fasto programs to ensure that the reference counting works correctly, deallocating memory safely and eliminating heap exhaustion crashes. By addressing these steps, we will hopefully move from a successful prototype to a practical implementation: the groundwork laid in this project has already provided a viable path toward integrating reference counting into the complete Fasto compiler.

8.3 Closing Remarks

In summary, this thesis has demonstrated a viable approach to automatic memory management in Fasto by using reference counting on a simplified language subset. We have implemented and validated the core mechanisms needed for reference counting from program transformation to runtime allocation and gained a deeper understanding of the challenges and design choices involved. With the insights and infrastructure developed here, Fasto can be extended to manage memory safely, turning what was once a crash-prone language (due to a lack of garbage collection) into one that proactively reclaims unused data. The next steps outlined will focus on scaling up our solution to Fastos full feature set, ultimately bringing a *hopefully* robust memory management to Fasto and preventing heap exhaustion in future programs.

References

Duncan, Kenneth et al. (2023). *RARS: RISC-V Assembler and Runtime Simulator*. https://github.com/TheThirdOne/rars. Accessed: 2024-02-7.

Flanagan, Cormac et al. (June 1993). "The essence of compiling with continuations". In: *SIGPLAN Not.* 28.6, pp. 237–247. ISSN: 0362-1340. DOI: 10.1145/173262.155113. URL: https://doi.org/10.1145/173262.155113.

James, Colin (2022). *ANF Conversion*. Accessed: 2025-04-9. URL: https://compiler.club/anf-conversion/.

Might, Matt (2008). *A-normalization: Writing compilers in continuation-passing style*. Accessed: 2025-04-9. URL: https://matt.might.net/articles/a-normalization/.

Mogensen, Torben Ægidius (2022). *Programming Language Design and Implementation*. Springer Nature Switzerland AG. ISBN: 978-3-031-11805-0.

— (2024). *Introduction to Compiler Design*. 3rd. Undergraduate Topics in Computer Science. Cham, Switzerland: Springer International Publishing. ISBN: 978-3-031-46459-1. DOI: 10.1007/978-3-031-46460-7.

Waterman, Andrew and Krste Asanovi (Dec. 2019). *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. Document Version 20191213. RISC-V Foundation. URL: https://riscv.org.

Deklaration for anvendelse af generative AI-værktøjer (studerende)

☒ **Jeg/vi har benyttet generativ AI som hjælpemiddel/værktøj** *(sæt kryds)*

☐ **Jeg/vi har <u>IKKE</u> benyttet generativ AI som hjælpemiddel/værktøj** *(sæt kryds)*

Hvis brug af generativ AI er tilladt til eksamen, men du ikke har benyttet det i din opgave, skal du blot krydse af, at du ikke har brugt GAI, og behøver ikke at udfylde resten.

Oplist, hvilke GAI-værktøjer der er benyttet, inkl. link til platformen (hvis muligt):

ChatGPT: http://chatgpt.com

Beskriv hvordan generativ AI er anvendt i opgaven:

1) *Formål (hvad har du/I brugt værktøjet til)*
   Eftersom projektet er skrevet individuelt, er AI blevet benyttet lidt som en sparringspartner til at diskutere ideer, spørge om faglig viden og til tider hjælp med debugging.
2) *Arbejdsfase (hvornår i arbejdsprocessen har du/I brugt GAI)*
   Inden implementering har jeg til tider spurgt den om ideen til mit design giver fin mening, eller om der er nogle indlysende fejl. Under implementering har jeg i nogle tilfælde spurgt den om hjælp til at finde kilden til fejl jeg ikke selv kunne spotte. Til at kunne teste min implementering fik jeg den også til at lave en pretty printer af ANF så jeg hurtigt kunne se hvordan min flat og analyse funktioner håndterede Fasto expressions.
3) *Hvad gjorde du/I med outputtet (herunder også, om du/I har redigeret outputtet og arbejdet videre med det)*
   Outputtet har for det meste forblevet på et diskussions niveau, og der er aldrig taget sat noget direkte ind i min kode eller rapport, men nærmere brugt som inspiration til mit arbejde, på nær nogle grammatiske tjek og pretty printeren i mine source code.

NB. GAI-genereret indhold brugt som kilde i opgaven kræver korrekt brug af citationstegn og kildehenvisning. Læs retningslinjer fra Københavns Universitetsbibliotek på KUnet.