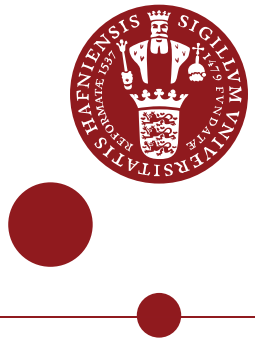
UNIVERSITY OF COPENHAGEN
DEPARTMENT OF COMPUTER SCIENCE



Bachelor project

Nikolaj Krarup, ltf688

Reference counting and memory management for Fasto

Block 3-4, 2025

Advisor: Andrzej Filinski

June 5, 2025

Contents

1 Introduction **1**

2 Background **2**
2.1 The Fasto language 2
2.2 Memory management 4
2.3 Reference counting 4
2.4 Tools and Technologies 4

3 Reference counting in MiniFasto **5**
3.1 Initial design 5
3.2 A-Normal form 9
3.3 Flattening an expression 11
3.4 Analysing A-Normal form 14

1 Introduction

Tough arrays are the main way to do things in Fasto, in its current state it has no garbage collection or memory management functions. Meaning that the heap will eventually run out of space in longer programs. The work done in this paper is an attempt to fix this problem, by extending the Fasto compiler to include garbage collection.

2 Background

2.1 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 handout is attached 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.

2.1.1 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` and `scan` apply functions to each element in an array, and therefore take 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 figure 2 for an example of a Fasto program making use of SOACs, ACs and lambda expressions. This program will read the length of an array, followed by its contents, then read one final integer which will be added to each element in the array, and finally it will be written to the standard output.

Example

```
fun [char] main() =
  let n = read(int) in
  let a = map(fn int (int i) => read(int), iota(n)) in
  let x = read(int) in
  let b = map(fn int (int y) => x + y, a) in
  write(b)
```

Figure 2: A Fasto program.

2.1.2 Code generation, præciser dette mere efter du ved hvad der er relavant når du impleenterer i FASTO. Rars??

The Fasto code generator function `compile`, located in `codegen.fs`, will convert a `TypedProg`, into an `Instruction list`. `TypedProg` and `Instruction` are discriminated unions 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, 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 3, 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.

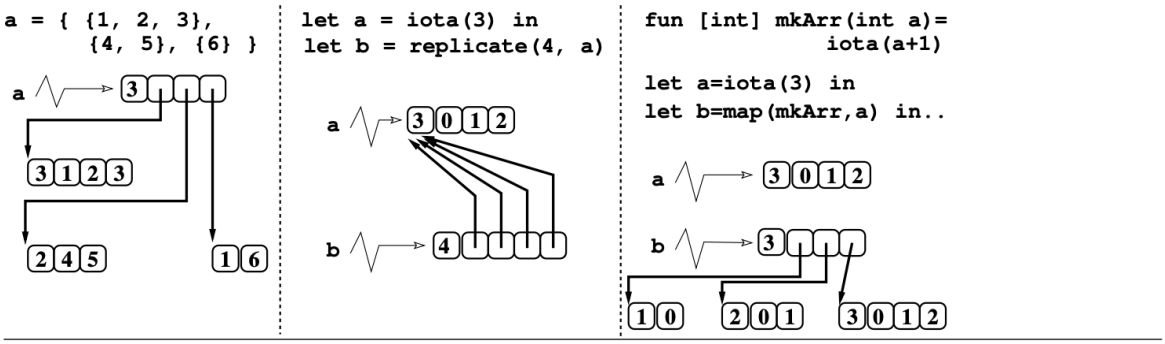


Figure 3: Array Layout.

For allocation of memory, the Fasto compiler uses a stack allocation approach, a more simplified version of the one described in (p. 73, Mogensen, 2022), as there is 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.

2.2 Memory management

2.3 Reference counting

2.4 Tools and Technologies

CLANG...

When running a compiled Fasto program, the RISC-V simulator RARS is used (Duncan et al., 2023). RARS is a simple simulator, created for educational purposes, as such it is not 100% up to industry standards as opposed to CLANG. This lead to some incompatibilities, which required some duct tape-like solutions to get CLANG compiled C code to run in RARS.

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. 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.

3.1 Initial design

As stated, 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, 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. The abstract syntax of the language can be seen in listing 1.

```
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 1: Definition of MiniFasto syntax

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 encapsulate the behaviour of `filter`, `reduce` and `scan`. 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 the Fasto src, but for the sake of simplicity MiniFasto uses F#'s build in `Map` type.

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

Listing 2: 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 3: MiniFasto interpreter

The functions closely follow the structure of the interpreter explained in chapter 4.3 in Mogensen, 2024. `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 1, 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`.

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. To simplify it further, it also has some additional pseudo-instructions. `CALL` and `RET` are used to streamline function calls and returns, similarly to how it's done in the intermediate language in chapter 6 of Mogensen, 2024. `CALL` has a list of registers containing the actual parameters' values, `RET` places the returned value in a special return register. As stated, Fasto uses the stack pointer to allocate arrays. 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.

```
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 4: Pseudo RISC-V types

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 5, 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 5: 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 6 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 6: MiniFasto original compiler

In the simulator, heap addresses point to integers, 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 7: MiniFasto RV simulator

The simulator, see listing 7, 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.

3.2 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.

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 (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.

3.2.1 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. 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)
```

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. Let's take a look at the same expression, 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 8: 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. 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.

Would a visual representation of minifasto vs anf tree be useful here?

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. The grammar has the restriction that we cannot do:

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

This adds copy and constant propagation as part of the translation. 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.

3.3 Flattening an expression

Before we can flatten an expression to ANF, the syntax needs to be defined in F#, see listing 9. It looks very similar to the original seen in listing 1, 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 9: 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 8, 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' return 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 10.

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

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

Listing 10: 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 11.

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

Listing 11: 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 statements can be handled much like binary operations, as the number of operands is explicit. 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 1, a `Prog` consists of a `FunDec` list, which we want to flatten into an `AProg` and an `AFunDec` respectively. See listing 12.

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

Listing 12: 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.

3.4 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:

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

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

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

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.

3.4.1 Live sets

To insert these annotations, we will use chapter 8 and 10.3 from 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 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 in a single tree traversal, in a functional manner using a recursive function `analyse`, eliminating the need for fixed-point iteration. This section defines how to calculate live sets for each sub-tree in an ANF tree.

In 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.

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 Mogensen, 2024.

For a call to `analyse` on the tree s, we use three sets, each defines the following:

- $below(s)$, the information received from below, flows upward. The result of a recursive call to `analyse` on the body of the tree. Contains the set of variables live in the body of the tree s. For a leaf, it will be initialised as $above(s)$.
- $above(s)$, the information received from above, flows downward. A parameter to `analyse` given along with the tree s by the parent. Contains the set of variables that are live after the entire tree finishes executing. For example, if `analyse` is going down a sub-tree inside `C` of a tree `let x = C in body`, then it should contain the variables that are live in `body`. If s is part of the main Let chain of the program, it should be empty.
- $in(s)$. Computed locally. Combines information received from both above and below, representing variables live in the sub-tree s. This set is returned to the parent or external caller.

Thus, for a tree s of the form `let x = C in body`, calling `analyse(s, above(s))` returns `(s', in(s))`. The function performs recursive calls on the body of the tree to obtain $below(s)$, and potentially on nested sub-trees within C. Leaf nodes act as base cases for the recursion, receiving $above(s)$ as their $below(s)$ set, forming the foundation for bottom-up calculation.

To formally define live sets, we need two additional helper sets: $gen(s)$ and $kill(s)$. $gen(s)$ is the set of free variables in the computation C of the tree, or simply the atom itself for a leaf. $kill(s)$ is the variable bound by the Let-binding at the tree's root; empty for leaf nodes. Since our language is immutable and avoids shadowing, each variable appears exactly once in $kill$ at its definition site. The live sets can then be formally defined as the following:

$$below(s) = \begin{cases} in(\text{body}(s)) & \text{if } s \text{ is a Let sub-tree,} \\ above(s) & \text{if } s \text{ is a leaf.} \end{cases} \tag{5}$$

$$above(s) = \begin{cases} below(parent(s)), & s \text{ is the root of a nested sub-tree within C} \\ above(parent(s)), & s \text{ is the body of a Let.} \end{cases} \tag{6}$$

$$in(s) = gen(s) \cup (below(s) \setminus kill(s)) \tag{7}$$

The names reflect the direction of information flow clearly. Information flows upward via $below(s)$, computed by analysing the body of each Let-binding, the body's in is the parent's $below$. Although the body of a Let-binding has no explicit reference to its parent, the recursive nature of the call stack allows information to flow upward as calls return. Meanwhile, $above(s)$ 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 $above(s)$ set received from the parent, while calls to nested sub-trees within a computation pass along the parents $below(s)$ set.

Thus, the upward flow ($below$ sets) is computed dynamically during recursive returns. However, the downward flow ($above$ 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 computation of the Let-binding `let b`. It receives an $above(s)$ set containing $\{b, t\}$, which are live after s_1s sub-tree execution.

Without downward information flow (via the $above(s)$ 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 4. Note that (*) denotes the body of the lambda function, which is the nested sub-tree s_1. Also, `x` will currently be returned as part of the entire tree's live set; of course, this is wrong as it is not in scope in the main Let chain. This will be corrected later.

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.

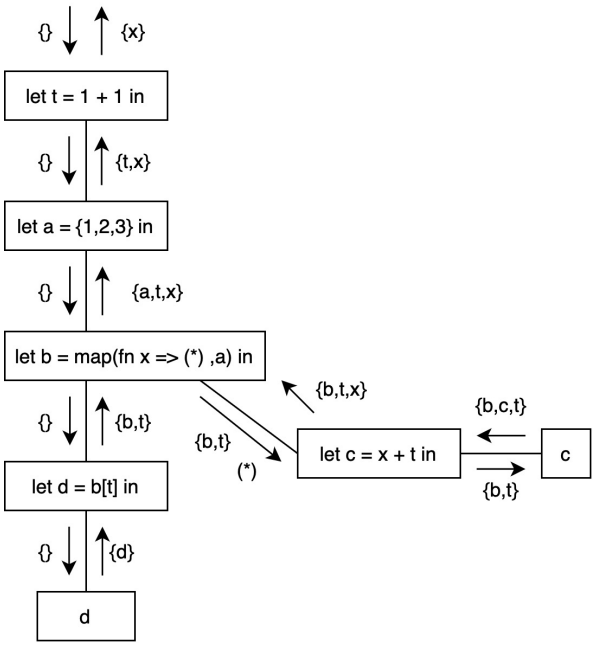


Figure 4: Example of information flow in ANF analysis

3.4.2 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 Mogensen, 2024, if a variable `x` is dead in the body of the tree `let x = C in body`, and C contains no side effects, then the root can be removed from the tree, simply giving us `body`. By applying this optimisation during the analysis, we have to rethink the formal definition (7), defining the set $in_{let}(s)$ for non-leaf trees of the form `let x = C in body`:

$$in_{let}(s) = \begin{cases} gen(s) \cup (below(s) \backslash kill(l)), & x \in below(s) \text{ or C has side effects} \\ below(s), & x \notin below(s) \text{ and C has no side effects} \end{cases} \tag{8}$$

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. $below(s)$.

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, lets 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 (7), (5) and (6), we have:

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

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 (**??**), this will not happen. `b` will be added to the live set $in(s_1)$, even though its value is never used. If instead, the set did not add the atom, giving us $in(s_1) = \{g\}$, then there would be a cascading effect, letting us remove the binding `let b`. Which in turn, as (8) 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 always be used.

This optimisation is only possible due to the fact that $a \notin below(s_2)$, i.e. s_2 is a dead binding, meaning that the value within `b` will never be used. Whereas in the example of Figure 4, the map is not a dead binding. This information needs to join the downward flow of the *above* sets. Therefore, we define the flag $use(s)$, 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 *above*, and is passed along down the tree where it's information will be used in the leaf, also similar to *above*.

Using this, we can formally extend (8), as the productions mirror what we want for a leaf (don't add *gen* or do), by creating the predicate $keep(s)$:

$$\begin{aligned} keep(s) = &(s \text{ is a leaf and } used(s)) \text{ or} \\ &(s \text{ is a Let and } (x \in below(s) \text{ or C has side effects })) \end{aligned} \tag{9}$$

This denotes weather or not the root of a tree s should remain. If it's a leaf, then we keep it if it's value is used. For a Let binding, we keep it if there are side effects, or the bound value is live in the body of the tree. Giving us the final definition:

$$in(s) = \begin{cases} gen(s) \cup (below(s) \backslash kill(l)), & keep(s) \\ below(s), & \text{not } keep(s) \end{cases} \tag{10}$$

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 $in(s)$ will correctly handle this, as in this case $b \in below(s)$, 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 $gen(s)$ to the set when $used(s)$ is false.

Using these definitions for the *in* sets of leaves and let bindings, it is still possible to perform the live set generation in a single traversal of the tree. To do so, the *used* flag and *above* 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 *in* and *below* sets, 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.

3.4.3 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
getVarsComp : AComp -> LiveSet
```

Listing 13: 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. Finally, `getVarsComp` is used to calculate the $gen(s)$ 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.

Now we can define the `analyse` function, see listing 14. 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) (liveAbove : LiveSet) (useFlag : bool) : ANorm, LiveSet =
  match a with
  | ValueA (N _) -> a, liveAbove
  | ValueA (V x) -> if useFlag then a, liveAbove.Add X else ValueA (N 0), liveAbove
  | LetA (id, tp, c, body) ->
      let body1, liveBelow = analyse body liveAbove
      let deadBinding = not (liveBelow.Contains id)
      if deadBinding && not (cIO c) then
        body1, liveBelow
      else
        let compLive = getVarsComp c
        let liveIn = Set.union compLive (liveBelow |> Set.remove id)
        LetA (id, tp, c, body1) , liveIn, liveIn
```

Listing 14: Analyse function for dead-binding elimination

The function will return the $in(s)$ set for the current tree along with an optimised version of said tree. It takes an ANF tree, the $above(s)$ set, and the use-flag $used(s)$. For a non-literal atom, we apply the definition (10), 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 $below(s)$, if $id \notin below(s)$ and there are no side effects, we can remove the binding. Otherwise, compute $gen(s)$ with `getVarsComp`, and calculate and return $in(s)$ along with the optimised ANF tree.

3.4.4 Annotating the tree, foundation

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 $above(s)$ and $below(s)$. Initially, we will start by just analysing flat computations, saving nested sub-trees a bit for now, so $above(s)$ 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 `C` of `let x = C in body`. A variable is used for the last time if it is present in $gen(s)$, but not in $below(s)$.

$$dead(s) = gen(s) \setminus below(s) \tag{11}$$

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 $below(s)$. 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. 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 15: 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 16.

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

Listing 16: 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 *above* 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) (liveAbove : LiveSet) (useFlag : bool) : ANorm, LiveSet =
  match a with
  ...
  | LetA (id, tp, c, body) ->
      let ttab1 = Map.add id tp ttab
      let body1, liveBelow = analyse body liveAbove ttab1 useFlag
      let deadBinding = not (liveBelow.Contains id)
      if deadBinding && not (cIO c) then
        body1, liveBelow
      else
      let compLive = getVarsComp c
      let dead = Set.difference compLive liveBelow
      let liveIn = Set.union compLive (liveBelow |> 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) , liveIn
    | _ -> failwith "not yet implemented"
```

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 *in* set. Some extra steps are required for the remaining computations, along with `IndexA` for multi-dimensional arrays.

3.4.5 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.

First, lets 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 17. The program is displayed on the left, the `liveBelow` set returned from a recursive call to the body is displayed in the middle, and the reference count of the initially allocated array is displayed on the right. When the analyser finds a variable in the tree's computation that is not present in `liveBelow`, it should decrement the variable, but it should also insert increments when a variable is copied.

```
   program                 liveBelow       {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; 1+2-1
3  let a3 = a2[0] in       {a2,a3}         3, copied again into a3; 2+1
4  let b  = f(a3,a2) in    {a3,b}          4?, copied a3 and a2, a2 last use?; 3+1
5  let c  = a3[b] in       {c}             3?, a3 last use; 4-1
6  c                       {}              3?
```

Listing 17: ANF example, incrementing.

First, the array is allocated with a reference count of 1, and a pointer is placed in `a1`. Next, the array's pointer is copied twice into the 2d array `a2`; however, since `a1` $\notin$ `liveBelow`, it also loses a reference as `a1` is dead and should be decremented. Now the only pointers to the array exist within `a2`. At line 3, an index computation is performed, copying the array pointer, ref count is 3. At line 4, a function is called, copying `a2` and `a3` into the formal parameter names of the function. This creates a copy of `a3`, ref count +1. `a2` has its last use in the current scope, but still exists within the function, so ref count is 4, maybe. The variables copied to the function are no longer in the program's scope, so it is currently unclear how many references remain. Next `a3` is used for the last time; now the only references that may remain are the ones passed to the function.

The example illustrates how the number of array references can increase, leading to some design decisions that need to be made. Most notably, is the caller or the callee responsible for decrementing arrays given as arguments? As we attempt to decrement variables as soon as possible, waiting for the function to return before decrementing is not the way to go, as at runtime, all arguments wouldn't be decremented until the function returns. However, the caller should still maintain responsibility to increment the variables before passing them to the function, after which the responsibility should be passed to the callee. Thus, when entering the body of the Let binding containing the function call, we will assume that the callee has correctly decremented the arguments. So in the example, instead of `4?`, it should say `1`, and from line 5 and down it should be `0`, implying that the array will be freed after its last pointer dies at line 5. What about function return values? If the callee decrements them before returning, they will return an invalid pointer. The caller should be responsible for the returned values, decrementing them when they are no longer needed.

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 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, liveIn
...
```

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 `liveIn` set are returned.

As was clear from the example in Listing 17, 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), liveIn
...
```

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.

Notes:

There is one final, less obvious case where it is necessary to place increment annotations. When a leaf of a tree within a computation node contains a free variable

local vs free variable returned from the tree, if it is local we don't have to increment, a copied reference is returned and the original dies with the branch. This is enforced as the

local variable is not contained in the `liveAbove` set, cause we only increment in that case. Also we only have to increment in the case of a true use flag.

References

Duncan, K., et al. (2023). RARS: RISC-V Assembler and Runtime Simulator [Accessed: 2024-02-7].

Flanagan, C., Sabry, A., Duba, B. F., & Felleisen, M. (1993). The essence of compiling with continuations. *SIGPLAN Not.*, *28*(6), 237–247. https://doi.org/10.1145/173262.155113

James, C. (2022). *Anf conversion* [Accessed: 2025-04-9]. https://compiler.club/anf-conversion/

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

Mogensen, T. Æ. (2022). *Programming language design and implementation*. Springer Nature Switzerland AG.

Mogensen, T. Æ. (2024). *Introduction to compiler design* (3rd). Springer International Publishing. https://doi.org/10.1007/978-3-031-46460-7

Waterman, A., & Asanovi, K. (2019, December). *The risc-v instruction set manual, volume i: Unprivileged isa* (Document Version 20191213). RISC-V Foundation. https://riscv.org