Conservative Compiler Optimisations

For a Polarised Intermediate Representation

Master's thesis in Computer Science and Engineering

SAMUEL KYLETOFT
EDVIN LUNDQVIST STERNVIK

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden, 2025

Master's thesis 2025

Conservative Compiler Optimisations

For a Polarised Intermediate Representation

Samuel Kyletoft
Edvin Lundqvist Sternvik

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden, 2025

Conservative Compiler Optimisations
For a Polarised Intermediate Representation
Samuel Kyletoft
Edvin Lundqvist Sternvik

Supervisor:	András Kovács,	Department of Computer Science and Engineering
Examiner:	Alex Gerdes,	Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31-772 10 00

Typeset in Typst
Gothenburg, Sweden, 2025

Conservative Compiler Optimisations
For a Polarised Intermediate Representation
SAMUEL KYLETOFT
EDVIN LUNDQVIST STERNVIK
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Modern functional programming is built upon heavy abstraction that compilers are not always able to optimise away. A two-stage compiler has been proposed to solve this, but lacking any optimisation this remains clunky to work in. We specify a number of optimisations in a way that they can be guaranteed to always be applied so that code generated by the first stage can be made simpler without causing any unnecessary overhead.

Keywords: compiler, optimisation, staged compilation, functional programming

Acknowledgements

Thanks to Rachel for helping us figure out how on earth to write typing rules.

Thanks to CMD and Loke for translating the LaTeX template to Typst so we could avoid the nightmare that is LaTeX.

Thanks to Pingu for helping us figure out how on earth to write operational semantics.

Thanks to all our proof readers: Nor Pingu Führ, Loke Gustafsson, Marie Kyletoft and Erik Magnusson.

Lastly, thanks to our supervisor, András Kovács, for the help throughout.

Samuel Kyletoft and Edvin Lundqvist Sternvik, Gothenburg, 2025-11-05

Contents

List of Figures

List of Listings

1 Introduction

Modern programs are written in high level languages, far abstracted from the details of the actual hardware they run on. While this has made code easier to write, it has also opened up the possibility for code to be slower in many instances than it was in the days of assembly handwritten by experts.

In 1957 John Backus and his team at IBM introduced the FORTRAN programming language, along with the first compiler that could produce faster code than would typically be written by hand [1][1]. Since then programming languages have gotten higher level and compilers or interpreters can no longer efficiently generate machine code for modern dynamic languages. As an example of this we have the infamously slow Python interpreter generating code that cannot be optimised efficiently due to highly dynamic language semantics where almost everything can be rebound or overridden, or extended with native libraries at any time [2]. Languages with more restrictive semantics, such as Rust are still incredibly fast [3].

Optimisers typically work by normalising input and then applying many simple transformations on it to slowly transform the code to a faster equivalent. They work as black boxes, which leads to reliability issues when programmers depend on specific optimisations and small changes cause the optimisation to no longer fire, as evidenced by studies trying to resolve this issue [4].

Furthermore, optimisation is often performed heuristically. This means that poor, or just non-perfect, guesses can lead to performance degradation or pessimisation [5]. To combat this, some languages define certain optimisations to be a part of the semantics of the language, such as tail call optimisation (TCO) in Scheme [6] or certain forms of copy elision (return value optimisation/RVO) in C++17 [7]. This guarantees that the optimisations happen and can be relied upon in a specification compliant implementation.

New opportunities for different types of optimisers and guaranteed optimisations opened up when Kovács introduced a system for two-stage compiled languages with different type systems in each stage integrated in a two level type theory [8]. A major difference to previous work is that this system is polarised, meaning that there are separate value and

[1]Ignoring the possibility for time travellers or the Silurian Hypothesis.

computation types. In this paper we use the word computation to mean any function or continuation.

We have built upon this system, implemented a set of conservative optimisations and written formal rewrite rules that could be added to the specification of a language so that they can be relied upon regardless of optimisation or debugging configuration.

1.1 Context

This section provides a review of some relevant literature and concepts used throughout this thesis. It also establishes the context for the research presented in this report and illustrates the issues that it is attempting to address.

1.1.1 Closure Free Type Theory

In the previously mentioned 2022 paper, Kovács describes a new approach for two-stage compilation with different type systems in each stage. The first stage, *the meta-level*, is evaluated (*unstaged*) resulting in code written entirely in the second stage's type system, *the object level*, which can then be compiled to machine code using established methods. When viewed as a whole he calls the integrated system a *two-level type theory*. This can be compared to the more advanced macro systems of languages like the lisp family or Rust, except that macros get to use meta-level types at expansion time.

He then uses this approach in [9] where he develops a particular two-level type theory, *CFTT* (Closure Free Type Theory), which consists of a dependently typed meta-level and a first-order simply typed object level. Using this system, a developer can write programs utilising the more advanced meta-level type system in *meta-programs*, and through unstaging, generate code that is entirely in the first-order simply typed object level, *object-programs*. The key idea here is that a first-order simply typed language lacks closures and lambda functions with captured context; efficiently compiling this is a solved problem. If we can force any abstraction in the meta-level to evaluate down to a simpler object-level representation then we can guarantee, by construction, that we do not end up with any abstraction that we cannot generate efficient machine code for in the final binary.

In order to facilitate interaction between the meta-level and object level, the following primitives are used:

- Lifting: If `A` is a type, then ⇑`A` is the type of meta-programs that unstage into object-level programs of type `A`.
- Quoting: If `A` is a type, and `a: A`, then ⟨`a`⟩ is the meta-program that immediately returns `a`.
- Splicing: If `A` is a type, and `a:` ⇑`A`, then ∼`a: A` In other words, ∼`a` inserts the result of the meta-program `a` into the object-level program.

During unstaging, meta-programs are evaluated in the splices and the result is inserted into the object-level code. This is used in the paper to show how domain-specific optimisations can be moved into meta-programs in libraries, guaranteeing that the optimisations happen when needed and freeing up the compiler's optimiser to stick to general optimisations rather than having to implement complex, costly and unreliable ways to undo issues caused by common patterns.

To illustrate how this works, first consider the following simple implementation of the `map` function that doesn't utilise any meta-programming:

```
map : (A -> B) -> List A -> List B
map f lst =
  let go as = case as of
    Nil       -> Nil
    Cons a bs -> Cons (f a) (go bs)
  go lst
```

Using this definition of `map`, we can define a function that increments each element in a list by 10 like this:

```
add10 : List Int -> List Int
add10 lst = map (λx. x + 10) lst
```

When applied to a list of integers, this function will apply the lambda `λx. x + 10` on each element in the list. We can improve this by using meta-programming, as in the following implementation of `map` and `add10`:

```
map : (⇑A -> ⇑B) -> ⇑(List A) -> ⇑(List B)
map f lst =
  ⟨ let go as = case as of
      Nil       -> Nil
      Cons a bs -> Cons ∼(f ⟨a⟩) (go bs)
    go ∼lst ⟩


add10 : List Int -> List Int
add10 lst = ∼(map (λx. ⟨∼x + 10⟩) ⟨lst⟩)
```

During unstaging, the meta-programs are evaluated in the splices, so the `add10` function is converted to the following at compile time:

```
add10 : List Int -> List Int
add10 lst =
  let go as = case as of
    Nil       -> Nil
    Cons a bs -> Cons (a + 10) (go bs)
  go lst
```

In this version of `add10`, the `map` function is inlined, eliminating one function call. More importantly, the lambda application `(λx. x + 10) a` is reduced to `a + 10` which significantly improves performance for long lists.

The `map` example is a relatively simple demonstration of what library-based optimisations can achieve using the meta-programming capabilities of CFTT. In his paper, Kovács [9] gives two specific examples of more involved library-based optimisations: monads and monad transformers, and stream fusion. The issue with monads is that do-notation normally desugars to calls to monadic binding and (allocating) closures and that type classes are implemented with indirect (C++: *virtual*) function calls. GHC (The Glasgow Haskell Compiler) can inline much of it, but it remains fragile. By implementing monads and stream fusion using meta-programming in CFTT instead, a lot of overhead caused by these abstractions can be removed from programs by moving some computation to the unstaging step which is only performed during compilation. We are thereby guaranteed to end up with final object code without any runtime closures.

1.1.2 Continuation Passing Style (CPS)

Abstract Syntax Trees (AST) are typically represented as just that, trees, and code is evaluated in a depth first order. To make returns explicit there is an alternate representation called Continuation Passing Style (CPS). A continuation is a section of code that is called after the current section has terminated. This can be used to pass on where the code should continue instead of returning to the call site and assuming there is more code to be run there. Here all continuations are passed explicitly as an extra argument. This does not mean that the final code does not use an explicit stack or exclusively calls by function pointers. To lower to CPS, code is transformed as in the following example:

```
let f = λx.
  let a = g x
  let b = a + 2
  b
```

⇒

```
let f = λx cont.
  let f_after_a = λa.
    let b = a + 2
    cont b
  g x f_after_a
```

Listing 1 : The function `f` from the normal AST form (left) and transformed into the CPS representation (right).

On the left, we first evaluate `a` by calling `g x`, then compute `b`, then return it. On the right, we instead call `g` with an extra argument `f_after_a`, which is a continuation representing the rest of the computation (`let b = a + 2; cont b`). When `g` finishes, it calls `f_after_a` instead of returning to `f`.

Here continuations are represented with ordinary functions; using normal functions like this is called first class continuations. There could also be a dedicated continuation construct, a second class continuation, which allows for a more relaxed calling convention and more efficient code generation [10].

1.1.3 A-Normal Form

A-Normal Form (ANF), is a response to the complexities of working with CPS [11]. ANF is closer to the original AST but flattens expressions and makes evaluation order explicit, as seen below. Each binding is either a literal or a function call where all arguments are bindings. There are no complex expressions in ANF, just many bindings to intermediate values. As opposed to CPS, this allows for easy reordering of the let-bindings.

```
let f = λx y.
  g (x + 1) (h (y + 2))
```

$\Rightarrow$

```
let f = λx y.
  let v0 = 1
  let v1 = x + v0
  let v2 = 2
  let v3 = y + v2
  let v4 = h v3
  let v5 = g v1 v4
  v5
```

1.2 Our Contributions

Using CFTT (described in Section 1.1), programmers have the ability to use meta-programming to influence how their code is compiled. This way, they can specify many optimisations directly to suit their needs. Despite this, some optimisations should still remain in the compiler for the following reasons:

- To decrease boilerplate code for handling common patterns; it is often more convenient to write meta-programs that generate non-optimal code in exchange for saving the programmer from unnecessary complexity.

  For example, in staged monadic code, the handling of tail calls either requires complicated machinery in meta-programming, or the simplification of redundant pattern matching in the compiler. In such monadic code, every recursive call has to be wrapped in an "up", which converts an object-level `Maybe` expression to a meta-level `Maybe` value by generating an object-level pattern match. So if some optimal hand-written object code contains a tail call of the form `f x`, the code that we get from the corresponding staged version would contain

  ```
  match f x with
    Nothing -> Nothing
    Just x  -> Just x
  ```

  which destroys the tail call. For more details, see [9].

- Programmers should be able to write modular code, splitting it into multiple libraries and modules that are independent of other parts of the program. This can be problematic because the library-based optimisations are only capable of working locally on each module. Some optimisations, however, work better when applied to the entire program. Two such optimisations are dead-code elimination and common subexpression elimination.

While there are many existing compiler optimisation techniques, it is not clear how they interact with the new type system developed by Kovács [9]. This new setting for compiler optimisation has different priorities and goals than the usual setting, due to the fact that we expect programmers to specify their own domain-specific optimisations. To effectively achieve this, the compiler must optimise code in a predictable and consistent manner. This has been implemented and is available in our GitHub repository [12][2].

The intended compiler pipeline is therefore the following:

[2]Note that all types and constructors have been renamed in this paper for the sake of clarity.

$$\text{Meta-level code} \rightarrow \text{Object-level code} \rightarrow \text{ANF-IR}$$
$$\rightarrow \text{OPT-IR} \rightarrow \text{LLVM IR} \rightarrow \text{Machine code}$$

Though this paper only covers the optimisation stage from Object-level code to OPT-IR.

1.2.1 Goals

Achieving the best possible performance for the resulting code is not a goal for the project, as this should be up to the domain-specific optimisations in libraries. Modularity and user-customisable optimisation is also not a goal for the project. Instead, we aim to create a suite of code optimisations that can fit into the language specification so that they are always guaranteed to be used.

Including the optimisations directly in the language specification is essential for the programmers' ability to rely on the optimisations when using the language. However, it also creates restrictions on what type of optimisations are practical. Specifically, viable optimisation techniques should have the following properties:

- Performing the optimising transformations should be *fast* since they cannot be disabled. However, as the compilation will happen *ahead-of-time* the performance requirement is not as strict as for *just-in-time* compilers. Ideally, we should be able to pin down time complexities which can be analysed to find a good balance between the two extremes.
- The optimisations should be *predictable* and *understandable* so that they can be relied upon by programmers without deep knowledge of the compiler. The optimisations must therefore be simple and intuitive.
- The transformations should not increase code size.
- The optimisation stage should not pessimise code, in other words the code should not become slower after this stage.

1.2.2 Limitations

As this thesis is focused on compiler optimisations on an intermediate representation-level and making them predictable, we will not be implementing a parser or code generation.

Moreover, we will not be providing formal proofs of correctness or optimality of our transformations.

1.3 Optimisation Techniques

There are many existing optimisation techniques which may be suitable in our setting. Due to limited time and scope, we will cover the following optimisations:

- **Common Sub-expression Elimination (CSE)**

  Factors out repeated sub-expressions to an outer let binding. This means that the expression is only evaluated once. This is only for actual repeated values and does not involve outlining similar expressions to new functions parameterised over the differences. It also solves code duplication by removing duplicated functions that only differ by name and the names of its parameters. Further described in Section 3.2.

- **Dead Code Elimination**

  Finds code that is not used and removes it. For example, if some function is defined but never mentioned in the rest of the code it can safely be removed. Further described in Section 3.3.

- **Match Reduction**

  Removes unnecessary branching in certain cases. Further described in Section 3.4.

2
Intermediate Representation

We have three levels of intermediate representation (IR): the surface language, ANF-IR and OPT-IR. This chapter describes each IR and how to lower from the surface language to ANF-IR.

2.1 The Surface Language

Our optimisations are intended to be applied to object-level programs generated from unstaging meta-level programs, as described in [9]. Therefore, we use a simply typed language that only supports first-order functions.

The surface language is intended to be just large enough to facilitate our optimisations. As such, the language only supports integer and unit literals and some basic arithmetic operations, along with lambda abstraction, function application, product types, sum types and conditional branching. The surface language does not support recursive types, as they are not required to demonstrate our optimisations. This means that our language is not completely compatible with the CFTT object language.

A slightly truncated definition of the surface language AST is shown in Listing 2. The term `LetFun n t ps u v` is used to bind a computation `u` of type `t` with the parameters `ps` to the name `n` in the continuation `v`. Similarly, `LetVal n u v` binds the value `u` to the name `n` in the continuation `v`. `Lam n u` corresponds to the lambda abstraction $\lambda n \to u$ and `App f x` represents the function application $f\ x$. `Match c` $\mathtt{n_l}$ $\mathtt{e_l}$ $\mathtt{n_r}$ $\mathtt{e_r}$ branches on the term `c`, if it matches `Left a` then $\mathtt{e_l[n_l := a]}$ is evaluated, otherwise if `c` matches `Right a` then $\mathtt{e_r[n_r := a]}$ is evaluated. Bound computations and variables can be referenced with `Var` terms, using *de Bruijn levels*.

Since de Bruijn levels are used, let-bindings can be referenced unambiguously without requiring alpha renaming. It also makes the optimisations simpler, as discussed in Section 3.2.2.

The `LetFun` and `LetVal` constructors contain the names of the variables for the sake of debugging and pretty printing. They are not used for any other purpose as de Bruijn levels are used for that. In our pseudo-code examples we use them for readability, and include wildcards for names that are not relevant. This does not affect the language.

```
data Surface ty
  = Var ty Lvl
  | LetFun Name [ty] Surface Surface
  | LetVal Name ty Surface Surface
  | Lam Name ty ty Surface
  | App ty Surface Surface
  | Left ty Surface
  | Right ty Surface
  | Fst ty Surface
  | Snd ty Surface
  | Pair ty Surface Surface
  | Match Surface ty Name Surface Name Surface
  | Unit
  | IntLit Int
  | Add Surface Surface
  | Sub Surface Surface
  | ...
```

```
type Lvl   = Int
type Name  = String
type Arity = Int
```

Listing 2 : Surface language AST, written in Haskell-like syntax.

Listing 3 shows a short program written in both pseudo-code and our surface language AST. The example further illustrates how bound variables are labelled using de Bruijn levels. In general, a variable's de Bruijn level is equal to the number of bound variables in scope at the position where the variable is declared.

```
let a = 1
let f = λx0.
  let x1 = match x0 with
    Left y1 -> Right y1
    Right y2 -> Left y2
  let x2 = match x1 with
    Left y3 -> Right y3
    Right y4 -> Left y4
  x2
let b = a + 2
f (Left b)
```

(a)

```
LetVal "a" (IntLit 1)
$ LetFun "f" 1 (Lam "x0"
  $ LetVal "x1"
    (Match (Var 2)
      "y1" (Right (Var 4))
      "y2" (Left (Var 4))
    )
  $ LetVal "x2"
    (Match (Var 3)
      "y3" (Right (Var 5))
      "y4" (Left (Var 5))
    )
  $ Var 4
)
$ LetVal "b" (Add (Var 0) (IntLit 2))
$ App (Var 1) (Left (Var 2))
```

(b)

a	$\rightarrow$	0
f	$\rightarrow$	1
x0	$\rightarrow$	2
x1	$\rightarrow$	3
y1	$\rightarrow$	4
y2	$\rightarrow$	4
x2	$\rightarrow$	4
y3	$\rightarrow$	5
y4	$\rightarrow$	5
b	$\rightarrow$	2

(c)

Listing 3 : Example program in pseudo-code (a) and corresponding surface language AST (b). The de Bruijn levels for all bound variables are shown in (c).

2.1.1 Type System

Because we are targeting the object-level code we can restrict the language to first-order functions, which makes many of the transformations and optimisations simpler and faster. This is due to the fact that local function definitions in higher-order languages create closures, which have a runtime cost, and would therefore require us to be careful about space safety and computation duplication. There are also some optimisations that are only possible because of the restriction to first-order functions, like saturation (see Section 2.2.4).

This constraint does not significantly hinder the expressiveness of the language since most of the time higher order functions are used for abstractions, which can often be modelled in the meta-level language and then unstaged to first order functions [9]. Sometimes runtime closures are important, such as in cases where behaviour is chosen at runtime and this choice can be reasonably cached out to a single check up front or when trying to emulate object oriented method overloading. Our IRs could be extended with explicit type formers for closures to support these use cases. We leave this for future work.

Listing 4 describes the type system of the AST. It contains no special or unexpected features beyond separating function and value types. The only primitive types are integers and unit. Booleans are excluded and instead represented as the sum of two units, meaning that `true` = `Left () : unit + unit` and `false` = `Right () : unit + unit`[3]. Representing them this way allows us to exclude a separate control flow construct (if-statements) for them, making all control flow work via the pattern matching `Match` construct.

This type system allows partial application and currying. In later stages this is no longer allowed and all functions are fully eta-expanded and saturated. This is described in more detail in Section 2.2.4.

Advanced types are built up of sum types and product types. Both of these only take two arguments and more complex types in the surface language must be represented by nesting these constructs. For instance the Haskell type `data SumType = A | B | C` would be modelled as `(unit + (unit + unit)`, `data ProductType = Product Int Int Int` as `int × (int × int)` and combining sums and products:

```
data Combined
  = A Int
  | B Int Bool Int
  | C ()
```

would be modelled as `int + (int × ((unit + unit) × int) + unit)`.

[3]You might have noticed that this is flipped from the expected Haskell representation. This was just defaulted to due to influence from Rust which writes its error type as `Result<Ok, Err>`.

$$\text{ValType} ::= \texttt{int} \mid \texttt{unit} \mid \text{ValType} + \text{ValType} \mid \text{ValType} \times \text{ValType}$$

$$\text{FunType} ::= \text{ValType} \to \text{Type} \qquad \text{Type} ::= \text{ValType} \mid \text{FunType}$$

$$\frac{n \in \mathbb{Z}}{\Gamma \vdash n : \texttt{int}}\ \text{Integer literals} \qquad \frac{}{\Gamma \vdash \texttt{()} : \texttt{unit}}\ \text{Unit Literals}$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \texttt{Left}\ a : A + B}\ \text{Sum type left} \qquad \frac{\Gamma \vdash b : B}{\Gamma \vdash \texttt{Right}\ b : A + B}\ \text{Sum type right}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \times B}\ \text{Product types}$$

$$\frac{\Gamma \vdash x : A \times B}{\Gamma \vdash \texttt{Fst}\ x : A}\ \text{First projection} \qquad \frac{\Gamma \vdash x : A \times B}{\Gamma \vdash \texttt{Snd}\ x : B}\ \text{Second projection}$$

$$\frac{\Gamma, x : A \vdash y : B}{\Gamma \vdash \lambda x.y : A \to B}\ \text{Lambda definition}$$

$$\frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash x : A}{\Gamma \vdash fx : B}\ \text{Function application}$$

$$\frac{\Gamma \vdash y : A \quad \Gamma, x : A \vdash z : B \quad A : \text{ValType}}{\Gamma \vdash \texttt{LetVal}\ x = y\ \texttt{in}\ z : B}\ \text{Let value}$$

$$\frac{\Gamma \vdash y : A \to B \quad \Gamma, x : A \to B \vdash z : C \quad (A \to B) : \text{FunType}}{\Gamma \vdash \texttt{LetFun}\ x = y\ \texttt{in}\ z : C}\ \text{Let function}$$

$$\frac{\Gamma \vdash c : (A + B) \quad \Gamma, n_l : A \vdash e_l : C \quad \Gamma, n_r : B \vdash e_r : C}{\Gamma \vdash \texttt{Match}\ c\ \texttt{with}\ n_l \to e_l; n_r \to e_r : C}\ \text{Match statement}$$

Listing 4 : The type system of the surface language, boolean and integer operators excluded.

2.1.2 Type Inference

The AST in our interface has optional and untrusted type annotations (`ty = Maybe Type`). To deal with this, the first transformation applied is a type inference step, which is the only step in our pipeline that may fail. We do this to simplify the writing of test cases.

Because of how early test cases are written we assume no type information beyond the arity of functions. A unification step is required due to generics in sum types:

```
let f = λ(x: ? + ?).
  let l = Left 1
  let r = Right false
  match x with
    Left _  -> l
    Right _ -> r
```

Here the variables `l` and `r` cannot be typed on their own and assuming default types would either cause a type error here or allow examples we do not wish to allow (if the default is something like `any`):

```
let true = Left ()
let f = λ(x: unit + int).
  match x with
    Left _  -> 0
    Right x -> x

let bad = f true // Type checks
```

The issue here is that `true` has an implied type of being a boolean, which we, as previously mentioned, represent as `unit + unit`. Without annotations, either on the literal or on the binding, we cannot specify the type of `Right` and the literal `true` is accepted by a function taking a `unit + int`.

The inference is a Hindley-Milner style algorithm [13]. The AST with optional annotations is traversed in a depth first order, assigning a fresh type variable to each subexpression (or type literal for literal values). When all children of a node have been processed the free variables are unified through a Union-Find. If any type variables are free or are assigned to multiple types by the end we declare a type error, otherwise we traverse the tree again with a second pass and re-annotate each node with its final type.

This system is a way to fix unannotated input code and has limitations. The obvious example is that we do not have any way to create type constructors, meaning that types that might be meant to be recursive or infinite will be inferred to have the depth they have in the concrete code. Any infinitely recursive types (i.e. lists) must be annotated as such up front.

2.1.3 Operational Semantics

An operational semantics is a formal definition of the meaning of a language [14]. They can be written as small step semantics or big step semantics. In this thesis we use big step semantics. They function as a set of pattern matching rules. If a part of a program matches the part above the line the part below is true. The ⇓-arrow is read as “evaluates to”. As an example

$$\frac{\gamma \vdash x \Downarrow u \qquad \gamma \vdash y \Downarrow v}{\gamma \vdash \texttt{Add}\ x\ y \Downarrow u + v}$$

is read as “If in the context γ (gamma) x evaluates to u and in the context γ, y evaluates to v, the expression `Add` x y evaluates to $u + v$” where addition of semantic values is left to traditional mathematics.

The variables x, y and z are used for subexpressions in the language of value types, f and g for subexpressions in the language of function types while the variables u, v and w are used for evaluated values. The constructors *Inl* and *Inr* are used to represent *Left* and *Right* as semantic values.

When accessing variables in the context we use the syntax $(x \coloneqq u) \in \gamma$. This is read as the variable x is assigned to the value v in the context γ. When appending variables to a context we instead use the syntax $\gamma(x \coloneqq v)$, read as "the context γ appended with the variable x assigned to v".

Listing 5 defines the standard big-step operational semantics for the surface language in terms of the AST-constructors defined in Listing 2. The rules are conventional and follow expected patterns for functional constructs (functions, applications, pairs, and sums). Boolean and integer operators are omitted for brevity. γ is the normal variable context while δ is the captured context for closures.

$$\frac{(x \coloneqq v) \in \gamma}{\gamma \vdash \texttt{Var}\ x \Downarrow v} \qquad \gamma \vdash \texttt{Unit} \Downarrow () \qquad \gamma \vdash \texttt{IntLit}\ i \Downarrow i$$

$$\frac{\gamma \vdash e \Downarrow \lambda x.g\{\delta\} \quad \gamma(f \coloneqq \lambda x.g\{\delta\}) \vdash y \Downarrow v}{\gamma \vdash \texttt{LetFun}\ f = e \text{ in } y \Downarrow v} \qquad \frac{\gamma \vdash e \Downarrow u \quad \gamma(x \coloneqq u) \vdash y \Downarrow v}{\gamma \vdash \texttt{LetVal}\ x = e \text{ in } y \Downarrow v}$$

$$\gamma \vdash (\lambda x.e) \Downarrow (\lambda x.e)\{\gamma\} \qquad \frac{\gamma \vdash f \Downarrow (\lambda x.e)\{\delta\} \quad \gamma \vdash a \Downarrow u \quad \delta(x \coloneqq u) \vdash e \Downarrow v}{\gamma \vdash (f\ a) \Downarrow v}$$

$$\frac{\gamma \vdash x \Downarrow v}{\gamma \vdash \texttt{Left}\ x \Downarrow \text{Inl}\ v} \qquad \frac{\gamma \vdash x \Downarrow v}{\gamma \vdash \texttt{Right}\ x \Downarrow \text{Inr}\ v}$$

$$\frac{\gamma \vdash x \Downarrow (y, z)}{\gamma \vdash \texttt{Fst}\ x \Downarrow y} \qquad \frac{\gamma \vdash x \Downarrow (y, z)}{\gamma \vdash \texttt{Snd}\ x \Downarrow z} \qquad \frac{\gamma \vdash x \Downarrow u \quad \gamma \vdash y \Downarrow v}{\gamma \vdash \texttt{Pair}\ x\ y \Downarrow (u, v)}$$

$$\frac{\gamma \vdash c \Downarrow \texttt{Inl}\ x \quad \gamma(n_{\text{l}} \coloneqq x) \vdash e_{\text{l}} \Downarrow y}{\gamma \vdash \texttt{Match}\ c\ \texttt{with}\ n_{\text{l}} \to e_{\text{l}}; n_{\text{r}} \to e_{\text{r}} \Downarrow y} \qquad \frac{\gamma \vdash c \Downarrow \texttt{Inr}\ x \quad \gamma(n_{\text{r}} \coloneqq x) \vdash e_{\text{r}} \Downarrow y}{\gamma \vdash \texttt{Match}\ c\ \texttt{with}\ n_{\text{l}} \to e_{\text{l}}; n_{\text{r}} \to e_{\text{r}} \Downarrow y}$$

Listing 5 : Operational semantics of the surface language, boolean and integer operators excluded.

2.2 ANF-IR

Before we can perform the desired optimisations, we must first convert the AST described in Section 2.1 to a more suitable intermediate representation.

There are several potential intermediate representations which might be suitable for our use cases, two of which are described in section Section 1.1. We chose ANF for our project due to the following desirable properties [15]:

- ANF is more direct than CPS, so the programs and optimisations are easier to understand than the more complex CPS counterparts.
- Some transformations are harder or out of reach in CPS, for example CSE.
- ANF, unlike CPS, does not commit to a particular evaluation order, since the intermediate let-bindings from ANF are relatively simple to reorder at any point.

The ANF-IR, shown in Listing 6, has a similar structure to the surface language AST.

```
data ANF
  = ARet Lvl
  | ALam Name [Name] ANF ANF
  | AApp Name Lvl [Lvl] ANF
  | ALeft Name Lvl ANF
  | ARight Name Lvl ANF
  | AFst Name Lvl ANF
  | ASnd Name Lvl ANF
  | APair Name Lvl Lvl ANF
  | AMatch Lvl Name ANF Name ANF
  | AIntLit Name Int ANF
  | AUnit Name ANF
```

```
type Lvl   = Int
type Name  = String
type Arity = Int
```

Listing 6 : The ANF-IR, written in Haskell-like syntax.

The main difference is that sub-expressions are flattened to let-bindings, as shown in Section 1.1.3. This requires a slight modification to the way that expressions are represented in the IR. Namely, most of the constructors in the ANF-IR represents expressions as a list instead of the tree structure of the Surface AST. The exceptions to this is the `ALam` constructor, which stores both the function body and continuation, and the `AMatch` constructors which stores two branch continuations.

In order to reference an expression, the `ANF` constructors keep track of the corresponding de Bruijn level instead of storing the entire expression recursively. This is possible because all sub-expressions have their own let-binding, and can therefore be referenced using a de Bruijn level.

The other change between the Surface AST and ANF-IR is that the arithmetic and logical operations no longer have dedicated constructors. Instead, they are represented as normal function applications using the `AApp` constructor. To do this, all operations are assigned a negative integer value, which is used instead of de Bruijn level. For example, addition is identified by the constant `-1` and subtraction by `-2`. Negative numbers are used because they are simple to distinguish from de Bruijn levels, which are always non-negative.

The differences between the Surface AST and ANF-IR is illustrated in the following example which shows how the expression `(λx. x + 1) 2` is represented in the two formats:

Surface

```
App
  (Lam "x" (Add (Var 0) (IntLit 1)))
  (IntLit 2)
```

ANF

```
ALam "t0" ["x"] (
    AIntLit "t2" 1
    $ AApp "t3" (-1) [1, 2]
    $ ARet 3
  )
  $ AIntLit "t1" 2
  $ AApp "t2" 0 [1]
  $ ARet 2
```

Just like the Surface AST, the ANF-IR keeps track of the `Name`s for all let-bindings, which are only used for debugging information and are otherwise ignored in all algorithms.

2.2.1 AMatch conversion

Because `AMatch`es do not create a let-binding, extra care must be taken when converting these from surface language to ANF-IR. Some compilers will perform a commuting conversion which might duplicate parts of the program to cause a potentially exponential increase in code size [15]. For example, the expressions `e4` and `e5` are duplicated in the program shown in Listing 7 when using such a conversion (1).

Source:

```
let a =
  match e1 with
    Left _  -> e2
    Right _ -> e3
match a with
  Left _  -> e4
  Right _ -> e5
```

⇒

Conversion 1:

```
match e1 with
  Left _  ->
    match e2 with
      Left _  -> e4
      Right _ -> e5
  Right _ ->
    match e3 with
      Left _  -> e4
      Right _ -> e5
```

Conversion 2:

```
let t = λa.
  match a with
    Left _  -> e4
    Right _ -> e5
match e1 with
  Left _  -> t a2
  Right _ -> t a3
```

Listing 7 : Conversion of branching code using a naïve commuting conversion (1) and the conversion we use (2).

The solution we use is to put the rest of the program after the let-binding inside a lambda abstraction and call it in both branches, passing the result of the branch as the argument (Listing 7 conversion 2). This is similar to the conversion used in GHC, except that we do not have a special language construct for representing join points. The reason GHC handles join points explicitly is to make sure that they are tail-called and therefore compiled to simple jumps to labels, instead of calls to heap-allocated function closures [15]. Since our functions are already guaranteed to be compiled without closures (see Section 1.1) we can omit this complexity.

2.2.2 Type System

The type system of the ANF-IR is similar to the surface language. The differences are that the ANF-IR lacks partial application and that the operators have been relegated to library functions (implemented with negative values in the place of de Bruijn levels).

$$\text{ValType} ::= \texttt{int} \mid \texttt{unit} \mid \text{ValType} + \text{ValType} \mid \text{ValType} \times \text{ValType}$$

$$\text{FunType} ::= (\text{ValType}_0 \times \text{ValType}_1 \times \cdots \times \text{ValType}_\text{n}) \to \text{ValType}$$

$$\frac{\begin{array}{c} n \in \mathbb{Z} \\ \Gamma, x : \texttt{int} \vdash y : A \end{array}}{\Gamma \vdash \texttt{let } x \texttt{ = AIntLit } n \texttt{ in } y : \text{A}}\text{Integer literals} \qquad \frac{\Gamma, x : \texttt{unit} \vdash y : A}{\Gamma \vdash \texttt{let } x \texttt{ = AUnitLit in } y : \text{A}}\text{Unit literals}$$

$$\frac{\begin{array}{c} \Gamma \vdash a : A \\ \Gamma, x : (A + B) \vdash y : C \end{array}}{\Gamma \vdash \texttt{let } x \texttt{ = ALeft } a \texttt{ in } y : C}\text{Sum type left} \qquad \frac{\begin{array}{c} \Gamma \vdash b : B \\ \Gamma, x : (A + B) \vdash y : C \end{array}}{\Gamma \vdash \texttt{let } x \texttt{ = ARight } b \texttt{ in } y : C}\text{Sum type left}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B \quad \Gamma, x : (A \times B) \vdash y : C}{\Gamma \vdash \texttt{let } x \texttt{ = APair } a\ b \texttt{ in } y : C}\text{Product types}$$

$$\frac{\begin{array}{c} \Gamma \vdash y : A \times B \\ \Gamma, x : A \vdash z : C \end{array}}{\Gamma \vdash \texttt{let } x \texttt{ = AFst } y \texttt{ in } z : C}\text{First projection} \qquad \frac{\begin{array}{c} \Gamma \vdash y : A \times B \\ \Gamma, x : B \vdash z : C \end{array}}{\Gamma \vdash \texttt{let } x \texttt{ = ASnd } y \texttt{ in } z : C}\text{Second projection}$$

$$\frac{\begin{array}{c} \Gamma,\ x_0 : A_0,\ x_1 : A_1,\ \cdots\ x_\text{n} : A_\text{n} \vdash y : B \\ \Gamma,\ f : (A_0 \times A_1 \times \cdots \times A_\text{n}) \to B \vdash e : C \end{array}}{\Gamma \vdash \texttt{let } f \texttt{ = } \lambda\ x_0\ x_1 \cdots x_\text{n}.\ y \texttt{ in } e : C}\text{Lambda definition}$$

$$\frac{\begin{array}{c} \Gamma \vdash \text{f} : (\text{A}_0 \times \text{A}_1 \times \cdots \times \text{A}_\text{n}) \to \text{B} \\ \Gamma \vdash \text{x}_0 : \text{A}_0,\ \text{x}_1 : \text{A}_1,\ \cdots\ \text{x}_\text{n} : \text{A}_\text{n} \quad \Gamma, y : B \vdash z : C \end{array}}{\Gamma \vdash \texttt{let } y \texttt{ = } f\ x_0\ x_1 \cdots x_\text{n} \texttt{ in } z : C}\text{Function application}$$

$$\frac{\Gamma \vdash \text{c} : \text{A} + \text{B} \quad \Gamma,\ \text{n}_\text{l} : \text{A} \vdash \text{e}_\text{l} : \text{C} \quad \Gamma,\ \text{n}_\text{r} : \text{B} \vdash \text{e}_\text{r} : \text{C}}{\Gamma \vdash \texttt{AMatch}\ \text{c}\ \texttt{with}\ \text{n}_\text{l} \to \text{e}_\text{l}\ ;\ \text{n}_\text{r} \to \text{e}_\text{r} : C}\text{Match statement}$$

$$\frac{\Gamma \vdash x : A}{\Gamma \vdash \texttt{ARet } x : A}\text{Return value}$$

Listing 8 : The type system of the ANF-IR

2.2.3 Operational Semantics

The operational semantics of the ANF-IR are the essentially the same as the surface language, with the exception of function application where all calls must now be fully saturated. Note how every rule except for branches and returns are in the form of `let...in` expressions and how all sub-expressions (not continuations) are variable look ups rather than sub-evaluations.

Lambdas do not evaluate to boxed closures. Rather, as we cannot return a lambda expression the captured environment must be a subset of the current environment. This can therefore be implemented as the new environment being the prefix of the environment of so many elements as the de Bruijn level of the function variable. This is written with the $\gamma[..f]$ syntax, meaning the prefix of γ until f.

$$\frac{\gamma(x := \texttt{()}) \vdash a \Downarrow u}{\gamma \vdash \texttt{let}\ x\ \texttt{=}\ \texttt{AUnitLit}\ \texttt{in}\ a \Downarrow u} \qquad \frac{\gamma(x := n) \vdash a \Downarrow u}{\gamma \vdash \texttt{let}\ x\ \texttt{=}\ \texttt{AIntLit}\ n\ \texttt{in}\ a \Downarrow u}$$

$$\frac{(y := u) \in \gamma \quad \gamma(x := \texttt{Inl}\ u) \vdash a \Downarrow v}{\gamma \vdash \texttt{let}\ x\ \texttt{=}\ \texttt{ALeft}\ y\ \texttt{in}\ a \Downarrow v} \qquad \frac{(y := u) \in \gamma \quad \gamma(x := \texttt{Inr}\ u) \vdash a \Downarrow v}{\gamma \vdash \texttt{let}\ x\ \texttt{=}\ \texttt{ARight}\ y\ \texttt{in}\ a \Downarrow v}$$

$$\frac{((u, v) := y) \in \gamma \quad \gamma(x := u) \vdash a \Downarrow w}{\gamma \vdash \texttt{let}\ x\ \texttt{=}\ \texttt{AFst}\ y\ \texttt{in}\ a \Downarrow w} \qquad \frac{((u, v) := y) \in \gamma \quad \gamma(x := v) \vdash a \Downarrow w}{\gamma \vdash \texttt{let}\ x\ \texttt{=}\ \texttt{ASnd}\ y\ \texttt{in}\ a \Downarrow w}$$

$$\frac{(y := u) \in \gamma \quad (z := v) \in \gamma \quad \gamma(x := (u, v)) \vdash a \Downarrow w}{\gamma \vdash \texttt{let}\ x\ \texttt{=}\ \texttt{APair}\ y\ z\ \texttt{in}\ a \Downarrow w}$$

$$\frac{\gamma(f := (\lambda\ x_0\ x_1 \cdots x_n.\ a)) \vdash b \Downarrow u}{\gamma \vdash \texttt{let}\ f\ \texttt{=}\ \lambda\ x_0\ x_1 \cdots x_n.\ a\ \texttt{in}\ b \Downarrow u}$$

$$\frac{\begin{array}{c}(f := \lambda\ x_0\ x_1 \cdots x_n.\ b) \in \gamma \quad (y_0 := w_0,\ y_1 := w_1,\ \cdots y_n := w_n) \in \gamma \\ (\gamma[..f])(x_0 := w_0,\ x_1 := w_1,\ \cdots x_n := w_n) \vdash b \Downarrow u \quad \gamma(z := u) \vdash a \Downarrow v\end{array}}{\gamma \vdash \texttt{let}\ z\ \texttt{=}\ f\ y_0\ y_1 \cdots y_n\ \texttt{in}\ a \Downarrow v}$$

$$\frac{(x := \texttt{Inl}\ u) \in \gamma \quad \gamma(n_l := u) \vdash a \Downarrow v}{\gamma \vdash \texttt{AMatch}\ x\ \texttt{with}\ n_l \to a\ ;\ n_r \to b \Downarrow v} \qquad \frac{(x := \texttt{Inr}\ u) \in \gamma \quad \gamma(n_l := u) \vdash b \Downarrow v}{\gamma \vdash \texttt{AMatch}\ x\ \texttt{with}\ n_l \to a\ ;\ n_r \to b \Downarrow v}$$

$$\frac{(x := u) \in \gamma}{\gamma \vdash \texttt{ARet}\ x \Downarrow u}$$

Figure 1 : Operational semantics for the ANF-IR.

2.2.4 Saturation

In our IR we also make sure that all computations are *saturated.* This means that all computations are eta-expanded fully such that a function of arity n has exactly n parameters and is always applied to exactly n arguments.

Before:

```
let a = λx.
  let b = λy z. z
  b 1
let d = a 2
```

$\Rightarrow$

After:

```
let a = λx z.
  let b = λy z. z
  b 1 z
let d = λz. a 2 z
```

This makes computations more explicit in the code, since they will always be behind a lambda. It also has positive performance implications, since it enables the compiler to transform computations into simple function calls. Conversely, mismatches between the number of parameters a function takes and the number of arguments it is called with requires the creation of heap-allocated closures [16]. Another advantage of saturated function calls is that they can more easily be compiled to some backend with fixed arity functions, and register-based calling conventions can be used in machine code. The alternative would be to have “dynamic” argument passing, like in a *push/enter* evaluation model, where arguments can only be passed on the stack, and not in registers [17].

Ensuring saturation is not possible in general for higher-order functions. Consider for example the following simple program:

```
let id = λx. x
let f = λx y. x + y
(id f 1 2) + (id 3)
```

Here `id` is called with three arguments, but we cannot eta-expand it since it is also called with only one argument. Fortunately, as only first-order functions are allowed in the object-level language under consideration in this thesis (see Section 2.1), it is always possible to know their runtime arity, so saturation is not an issue for us.

ANF-IR conversion and saturation is done at the same time in a single pass, but we will be discussing them separately in the following sections.

2.2.5 Surface Language To ANF-IR Lowering

The conversion from `Surface` to `ANF` is done using a CPS-style algorithm which recursively traverses the AST. Since each surface language term might produce any number of new bindings, we do not know beforehand what de Bruijn level it will have in the ANF representation. We therefore process the remainder of the program in a continuation function which takes as arguments the level of the current term, and the next fresh level which will be used for the next term.

During the conversion, new let-bindings might be introduced if the program contains expressions with unnamed intermediate values. The ANF-IR lowering must therefore keep track of an "environment" that maps the old de Bruijn levels to the updated ones, adjusted for the newly inserted let-bindings.

Since de Bruijn levels are increasing monotonically by exactly 1 for each let-binding in scope, we can represent the environment as a list of de Bruijn levels and use list indexing as the mapping function. In other words, during the conversion from `Surface` to `ANF`, when encountering a new let-binding, this let-bindings updated de Bruijn level is appended to the environment list. It can later be retrieved by taking the element at index l from the environment, where l is the old de Bruijn level of the let-binding.

To perform saturation, we also keep track of a list of extra arguments that should be applied to the term that is being processed. This process is described in Section 2.2.6.

2.2.6 Saturation Conversion

When saturating functions we make use of the fact that the `LetFun` constructor keeps track of the arity of the computation. From this information we can immediately create an `ALam` with that number of arguments. When processing the body of the computation we keep track of these arguments. When encountering a function application, we push the new argument to the list of arguments, and when encountering a lambda we pop the topmost argument and update the environment accordingly by pushing it to the top of the environment. Finally, after processing the entire body, we apply any remaining arguments to the result.

Listing 9 shows an example of the entire saturation procedure for a simple computation. The first step introduces a lambda abstraction with two arguments (`t` and `u`), since `f` has arity 2. These arguments are pushed to the argument list and the body of the computation is processed recursively. In the second step a function application is encountered, so the argument `a` is pushed to the argument list. The third step processes a lambda term, so `a` is removed from the argument list and pushed to the environment list, effectively renaming all occurrences of `x` to `a`. Steps four and five continue much like steps three and two, respectively. Finally, the remaining arguments (`a` and `u`) are applied to the `(+)` function.

```
    let f = (λx. λy. (+) x) a              []                ...
1.  let f = λt u. (λx. λy. (+) x) a     t : u : []           ...
2.  let f = λt u. λx. λy. (+) x     a : t : u : []           ...
3.  let f = λt u. λy. (+) a             t : u : []           x→a : ...
4.  let f = λt u. (+) a                   u : []       y→t : x→a : ...
5.  let f = λt u. (+)                   a : u : []     y→t : x→a : ...
6.  let f = λt u. (+) a u                  []          y→t : x→a : ...

                  (a)                      (b)               (c)
```

Listing 9 : Step by step procedure (a) for saturating the computation `f` with arity 2, along with the argument stack (b) in each step of the saturation procedure, and (c) the environment mapping old de Bruijn levels to the updated levels.

2.2.7 ANF Conversion Implementation

A truncated and simplified version of the `Surface` to `ANF` lowering function is shown in Listing 10. The omitted constructors are handled similarly to the `IntLit` constructor. In this specific case, an `AIntLit` is constructed, and `cont` is used to construct the continuation. The first argument to `cont`, which represents the next available de Bruijn level in the continuation, is `nextLvl + 1`. This is due to the fact that a new let-binding is constructed to store the integer literal. The second argument to `cont` is `nextLvl`, which is the de Bruijn level of the newly defined let-binding.

The most involved case in the `anf` function is the `LetFun` case. In this case, the computation body is converted using the `anfTail` function, which is the same as the `anf` function, except that it does not take a continuation. Since `LetFun` represents a let-binding in the surface language, we must prepend `nextLvl` to the environment `env` so that it can be referenced in the continuation. After the body is converted, the continuation is converted, and an `ALam` is constructed.

The `Var` constructor is used for referencing a let-binding from the original surface language. Since new let-bindings are created during the ANF conversion pass, the original de Bruijn level `v` must be converted to the new level using the environment `env`. As described in Section 2.2.6, the referenced variable might be wrapped in a function application if it is a computation.

```
type Env = [Lvl]
type Arguments = [Lvl]
type NextLvl = Lvl
type Cont = NextLvl -> Lvl -> ANF

anf :: Env -> Args -> NextLvl -> Surface -> Cont -> ANF
anf ... env args nextLvl term cont = case term of
  Var _ v -> case args of
    [] -> cont nextLvl (lookup env v)
    _  -> AApp ... (lookup env v) args (cont (nextLvl +1) nextLvl)
  LetVal _ _ t u ->
    anf (nextLvl : env) [] nextLvl t $ \nextLvl' tLvl ->
      anf (valLvl : env) args nextLvl' u cont
  LetFun _ args comp letCont ->
    let arity = length args
  bodyParams = take arity [nextLvl + 1 ..]
  bodyNextLvl = nextLvl + 1 + arity
  bodyAnf = anfTail (nextLvl : env) bodyParams bodyNextLvl comp
  contAnf = anf (nextLvl : env) args (nextLvl + 1) letCont cont
     in ALam ... bodyAnf compAnf contAnf
  Lam _ _ _ body ->
    let (param : args') = args
     in anf (param : env) args' nextLvl body cont
  IntLit i ->
    AIntLit ... i (cont (nextLvl + 1) nextLvl)
  ...
```

Listing 10 : A simplified version of the anf lowering function. This version omits types and let-binding names. The `anfTail` function, which is mentioned in the code, is the same as the `anf` function, except that it does not take a continuation.

2.3 OPT-IR

The final IR used in this project is largely the same as the ANF-IR, except that `OLam` and `OMatch` are modified to refer to computations using a list of captured variables and a computation id. This IR is shown in Listing 11.

```
data OPT
  = ORet Lvl
  | OLam Name [Name] Comp OPT
  | OApp Name Lvl [Lvl] OPT
  | OLeft Name Lvl OPT
  | ORight Name Lvl OPT
  | OFst Name Lvl OPT
  | OSnd Name Lvl OPT
  | OPair Name Lvl Lvl OPT
  | OMatch Lvl Name Comp Name Comp
  | OIntLit Name Int OPT
  | OUnit Name OPT
```

```
type Captures = [Lvl]
type CompID = Int
type Comp = (Captures, CompID)
```

Listing 11 : The OPT-IR, written in Haskell-like syntax.

The reason why this IR exists is because the extra information for computations is required in the CSE optimisation pass, described in Section 3.2. This IR is the target

IR for the optimisations described in Section 3. In other words, the optimisations are performed during the conversion from the ANF-IR to the OPT-IR.

Performing the conversion from Surface AST to OPT-IR in two steps through the ANF-IR makes the conversion much simpler than it would have been if the entire conversion was performed directly from Surface AST to OPT-IR.

3

Optimisations

In this chapter we present the optimisation passes listed in Section 1.3 and describe our implementations and specifications of them.

3.1 Rewrite Rules

The optimisations are specified using the rewrite rules in Listing 12. Throughout the program, sections matching the expression on the left and fulfilling any side conditions are replaced with the expression to the right of the arrow. While they are implemented in the pass lowering from ANF-IR to OPT-IR, the specification is in terms of the surface language.

Dead value/computation elimination

$$\texttt{let x = t; u} \quad \longmapsto \quad \texttt{u} \qquad \text{if } \texttt{x} \notin \mathrm{FV}(\texttt{u})$$

Identity match reduction

```
match t with
  Left x  -> Left x         ⟼   t
  Right y -> Right y
```

Constant match reduction

```
match t with                               x ∉ FV(u)
  Left x  -> u         ⟼   u    if
  Right y -> u                             y ∉ FV(u)
```

Duplicate computation elimination

$$C[C'[\texttt{comp}]...[\texttt{comp}]] \quad \longmapsto \quad C[\texttt{let f = comp;}\ C'[\texttt{f}]...[\texttt{f}]] \quad \text{if } C' \text{ binds all in } \mathrm{FV}(\texttt{comp})$$

Listing 12 : Optimisation rewrite rules

In the rewrite rules, the meta-variables `x` and `y` represent arbitrary let-binding names, `u` and `t` represent arbitrary terms, and `comp` represents an arbitrary computation. A computation is a lambda, or a function application where an insufficient number of arguments are supplied to saturate the function. Where the same meta-variable is used multiple times in the same rule, it stands for multiple alpha-convertible terms or computations.

Furthermore, the side conditions use the notation FV(`u`) to refer to the set of free variables of the expression `u`. The notation C[`e`] is used for *contexts*, which refer to a section of the code corresponding to the expression `e`, and the context C[`e`]...[`e`] refers to multiple different sections of the code; all corresponding to the expression `e`. To illustrate how this is used in rewrite rules, consider the following example: $C[\texttt{t}]...[\texttt{t}] \longmapsto C[\texttt{u}]...[\texttt{u}]$, which can be read as "all occurrences of `t` in the code is replaced by `u`".

The dead value/computation elimination rule states that unused let-bindings are removed if they are not used in the rest of the program. The identity match reduction rule eliminates match statements when both branches simply reconstruct the conditional variable. Similarly, the constant match reduction rule removes match statements when both branches are identical up to alpha-conversion. Notably, this rewrite rule does not get triggered if the branches reference the unwrapped condition variable (`t`), as shown by the side-condition. For example, the following code would not get reduced even though the branches look identical:

```
match t with
  Left x -> x
  Right x -> x
```

The final rule, the duplicate computation elimination rule, removes unnecessary repetition in the code by merging multiple definitions of the same computation (`comp`) and placing it under a single let binding (`f`). This let-binding is then referenced in place of the original computation.

3.1.1 Rule Ordering

The rewrite rules behave differently depending on the order in which they are applied. For example, some computations might look very different originally but become identical only after performing some optimisations, like DCE. If duplicate computation elimination is applied before these other computations then it might miss an opportunity to simplify the code. The same argument could be made for the other rules.

Therefore, all sub-terms referenced in the rewrite rules in Listing 12 are optimised before the rewrite rule is performed for the entire term. For example, in the expression `let x = t; u`, all rewrite rules are evaluated on the `t` and `u` terms first, before the rewrite rules are applied to the whole expression.

3.2 Common Subexpression Elimination

CSE is performed by constructing a global table which maps computations to unique identifiers. These identifiers are then used to refer to computations, instead of storing the computations directly in the IR. Since the computations might contain free variables, the IR must also keep track of the captured variables at each reference to a computation.

This requires us to convert the program into a modified version of ANF-IR, where all constructors correspond exactly to the ANF-IR constructors, except for `OLam` and `OMatch` which have been modified to use the computation identifiers and capture lists. This IR was described in Section 2.3.

During conversion from the ANF-IR to the OPT-IR, whenever a computation body is encountered, it is converted to OPT-IR before the rest of the function. If an identical computation already exists in the table, its identifier is retrieved. Otherwise, the computation body is assigned a new unique identifier and inserted into the table.

3.2.1 De-duplication of Value Bindings

Our implementation of CSE only covers de-duplication of computations. An implementation could cover de-duplication of values as well, but in our garbage collected environment values might disappear after last use rather than at the end of a lexical scope. Therefore any de-duplication of values would lead to the risk of lifetime extensions and memory “leaks” [18]. Defining the semantics around this are possible, but non-obvious, as showcased by different approaches taken by languages such as Haskell (garbage collection and last use [19]) and Rust (manual memory management and lexical scopes [20]). Doing de-duplication of values would be simple, but the consequences of the life time extension are not always understandable or intuitive. In addition, if the programmer requires the longer lifetime, it is still possible for them to do CSE of it manually. We have therefore chosen to not do CSE of values at this time.

3.2.2 Let-binding Labelling

As previously mentioned in Section 2.1, bindings in ANF-IR are labelled using de Bruijn levels. The following example shows why this is problematic:

```
let f = λx. x         ⟹      let f = λx. 1        let f = λx. 0
let g = λy. y                 let g = λy. 2        let g = λy. 0

                            de Bruijn levels    de Bruijn indices
```

In this example, the functions `f` and `g` are identical, but look different when taking the de Bruijn indices into consideration, due to `f` existing in scope by the time `g` is defined, offsetting all new bindings by one. One idea might be to fix this using *de Bruijn indices* instead. Indeed, this fixes the previous example, but it breaks the following program:

```
let f = λx. x           ⟹      let f = λx. 1          let f = λx. 0
let g = λx. f x                 let g = λx. 0 2        let g = λx. 2 0
let h = λx. f x                 let h = λx. 0 3        let h = λx. 3 0

                              de Bruijn levels     de Bruijn indices
```

The solution is to use de Bruijn levels, but to restart labelling from 0 inside each function body. This way two equivalent functions become equal, but it is no longer possible to reference free variables inside a function body. To fix this, the computations must be annotated with a list of captures. The captured variables are labelled consecutively from 0, followed by the function arguments and then the let-bindings in the body.

The previous two programs are represented in the following way when using this labelling scheme:

```
                                   let f = λx. 0
let f = λx. 0                      let g = λ[f] x. 0 1
let g = λy. 0                      let h = λ[f] x. 0 1
```

Using this small modification to normal de Bruijn level labelling, computations can be directly compared to check if they are equal, irregardless of how the let-bindings are named. Using another method like alpha-renaming, we would need to use a much more complicated procedure for comparing computations.

The capture annotation procedure is similar to *closure conversion* [21], but the annotated computations are not used to create runtime closures. Instead, they are just used for checking equality of computations between scopes.

3.2.3 Capture List Ordering

In order to determine that two computations are equivalent and should be combined into a single entry in the computation table, we must check that they have the same arity, the same number of captured variables, and identical bodies after the OPT-IR conversion procedure. Notably, only the number of captured variables are relevant, not which variables are actually captured.

A natural implementation would be to store the captured variables as a set, ordered by the de Bruijn levels of the captured variables. This ordering does not work in practice however, as illustrated in Listing 13, where an extra entry must be created in the table for `h` just because it captures `a` and `b` in the reverse order from `f`.

```
      Source:                 OPT-IR converted:         Computation table:

let a = 1                    let a = 1                  comp0: {d, e} λx. d + e
let b = 2                    let b = 2                  comp1: {d, e} λx. e + d
let c = 3              ⇒     let c = 3
let f = λx. a + b            let f = comp0 {a, b}
let g = λx. b + c            let g = comp0 {b, c}
let h = λx. b + a            let h = comp1 {a, b}
...                          ...
```

Listing 13 : Example of a program where three functions are combined into two entries into computation table using a naïve implementation where the capture list is ordered by de Bruijn level. The variables in { .. } represents the list of captured variables.

The solution to this problem, which is used in Listing 14 is to order the capture list by appearance in the computation body. This way, the computation bodies for `f` and `h` become identical and the capture list for `h` is reversed instead. Since the capture list is not relevant for checking equality for computation bodies, all three functions are now merged into a single entry in the computation table.

Source:

```
let a = 1
let b = 2
let c = 3
let f = λx. a + b
let g = λx. b + c
let h = λx. b + a
...
```

⇒

OPT-IR converted:

```
let a = 1
let b = 2
let c = 3
let f = comp0 {a, b}
let g = comp0 {b, c}
let h = comp0 {b, a}
...
```

Computation table:

```
comp0: {d, e} λx. d + e
```

Listing 14 : The same example program as in Listing 14, except this time the capture list is ordered by appearance in the computation body.

3.2.4 Topological Sorting of Let-bindings

Before OPT-IR terms are compared for equality, they should be normalised by sorting the let-bindings. This ensures that the order in which let-bindings are defined does not influence the equality of two terms that are otherwise the same. For example, without sorting, the two programs in Listing 15 are not equal even though they are essentially the same program.

```
let a = 1
let b = 2
let c = b + b
let d = a + c
```

```
let b = 2
let c = b + b
let a = 1
let d = a + c
```

Listing 15 : Two programs that are identical except for the order of their let-bindings

As we have seen, the CSE optimisation pass relies on the ability to compare computations for equality. Consequently, this optimisation would occasionally behave inconsistently for two programs which have different orders of let-bindings. This makes CSE less robust and more difficult to reason about.

To fix this issue, we can define a consistent ordering of terms so that different programs become equal after sorting if they previously only differed in the order of their let-bindings. This ordering must be a *topological ordering*, as this ensures that let-bindings are defined before they are used.

A topological ordering over a graph $G = (V, E)$ is a total ordering of V such that for every directed edge $(u, v) \in E$, u appears before v in the ordering. In our case, the nodes are the let-bindings, and the directed edges are dependencies between them. Informally, a topological ordering will sort these nodes in a way such that the edges only go forward, as seen in Figure 2. In other words, the let-bindings will be sorted such that they are defined before being referenced for the first time.

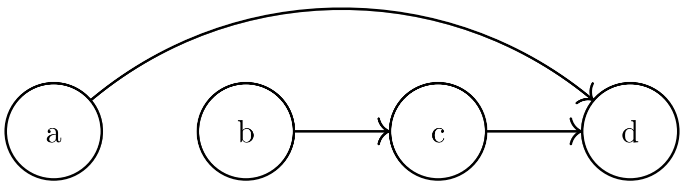


Figure 2 : Graph corresponding to the programs in Listing 15

In general, there are several topological orderings for any given graph. Any of these can be chosen, but it must be chosen consistently for different programs such that the same ordering is produced irrespective of the original order and names of let-bindings.

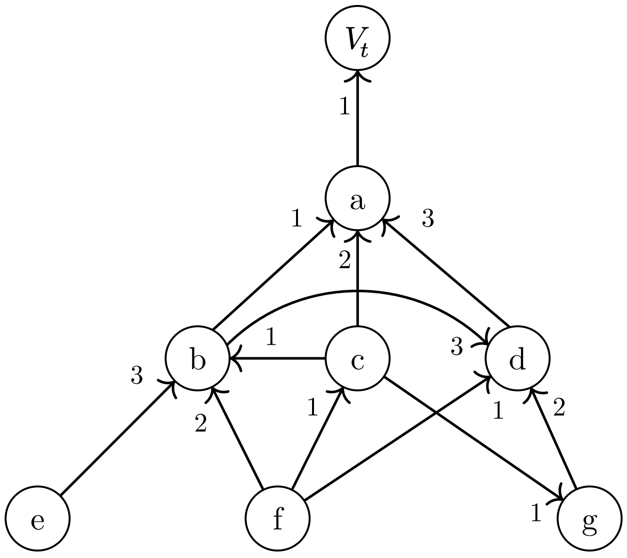
The capture list ordering described in section Section 3.2.3 updates the de Bruijn levels for the free-variable list based on the order of the let bindings in the computation body. Consequently, it must therefore be performed after the let-bindings have been sorted.

3.2.5 Topological sort algorithm

We represent the unsorted computation as a dependency graph $G = (V, E)$ where the nodes V represent the `OPT` statements of the computation, and the edges E are dependencies between them such that $(v, u) \in E$ if u depends on v. We also keep track of the order of the dependencies (edges) to each node u, such that it is consistent for all re-orderings of the let-bindings. Most statements only reference a single let-binding, so there is only one possible ordering for these. However, the `OApp`, `OPair` and `OMatch` statements might reference more than one let-binding, so there are several possible orderings possible, which all work equally well. We have chosen to order the dependencies in the same way that they are defined in the `OPT` constructors, from left to right. Figure 3 shows the dependency graph for an example program.

```
let f = 3
let c = f + 1
let e = 2
let b = c + f + e
let g = c + 4
let d = f + g + b
let a = b + c + d
a
```

(a)



(b)

Figure 3 : An example computation (a), and the corresponding dependency graph (b). Each node is labelled with the name of the corresponding let-binding, or V_t for the terminating node. The edges are labelled with its dependency order.

The algorithm for topologically sorting a computation is shown in Listing 16. It always traverses the nodes in the same way, if the same dependency graph, dependency ordering, and starting node is used. These properties are satisfied for any two computations which differ only in the ordering of let-bindings:

- Any two computations that become identical after being topologically sorted are represented by the same dependency graph, since they differ only in the order of the let-bindings; They consist of the same statements and dependencies between them, so their dependency graphs consist of the same nodes and edges.

- Because we have selected a consistent ordering of the dependencies that does not depend on the original order of the let-bindings, the dependency graphs will also have the same orderings of the edges.
- All computations in the OPT-IR consist of a sequence of let-bindings, followed by an `ORet` or an `OMatch` statement. This statement must remain the last statement of the computation after sorting. The corresponding node in the graph (the *Terminating node*, V_t) is therefore necessarily used as a starting point for the topological sort algorithm, which is is consistent between all possible orderings of the computation.

```
1  $S \leftarrow [\,]$              // The resulting topologically sorted program
2  $Q \leftarrow \{ V_t \}$          // Queue of nodes to process
3  while $Q$ is not empty
4      $u \leftarrow$ Pop first node from $Q$ with 0 remaining outgoing edges
5      Prepend $u$ to $S$
6      $D \leftarrow \{ v \mid (v, u) \in E \}$, in a consistent order
7      for each $v \in D$
8          Append $v$ to the end of $Q$ if $v \notin Q$
9          Remove $(v, u)$ from $E$
```

Listing 16 : Pseudocode for the topological sorting algorithm.

Step 4 in the algorithm requires that if Q is not empty, then there exists at least one node in Q with 0 remaining outgoing edges. This always holds if the graph does not contain any cycles. Since all valid computations are topologically sorted, we know that the original graph is cycle free and the topological sort algorithm will terminate with a valid topologically sorted computation.

To illustrate how the topological sort algorithm works, Figure 4 shows the contents of S and Q for each iteration of the algorithm for the example shown in Figure 3 a. The resulting topologically sorted computation is:

```
let e = 2
let f = 3
let c = f + 1
let g = c + 4
let b = c + f + e
let d = f + g + b
let a = b + c + d
a
```

1.	$\{ \}$	$\{ (R_t, 0) \}$
2.	$\{ R_t \}$	$\{ (a, 0) \}$
3.	$\{ a, R_t \}$	$\{ (b, 1), (c, 2), (d, 0) \}$
4.	$\{ d, a, R_t \}$	$\{ (b, 0), (c, 2), (f, 2), (g, 0) \}$
5.	$\{ b, d, a, R_t \}$	$\{ (c, 1), (f, 1), (g, 0), (e, 0) \}$
6.	$\{ g, b, d, a, R_t \}$	$\{ (c, 0), (f, 1), (e, 0) \}$
7.	$\{ c, g, b, d, a, R_t \}$	$\{ (f, 0), (e, 0) \}$
8.	$\{ f, c, g, b, d, a, R_t \}$	$\{ (e, 0) \}$
9.	$\{ e, f, c, g, b, d, a, R_t \}$	$\{ \}$
	(S)	(Q)

Figure 4 : The contents of S, Q after each iteration of the topological sort algorithm (Listing 16) for the example computation shown in Figure 3 a. For each entry in Q, the number of outgoing edges for that node is also shown.

3.2.6 CSE Implementation

The CSE optimisation is performed at the same time as the other optimisations described in this section. This is done in two essentially linear passes over the program. Our implementation is not actually linear but could be with a better choice of data structures, e.g. linear lists vs arrays or maps.

The first pass converts the program from the ANF-IR to the OPT-IR. However, after this conversion, the de Bruijn levels are not updated to use the labelling scheme described in Section 3.2.2. This "renaming" translation is done in the second pass, since it depends on the number of captured variables in each computation, which is only known after performing the first pass.

Listing 17 shows a simplified version of this optimisation procedure. The version shown in this listing only performs CSE and will be expanded on in the coming sections to show how it is modified to handle the other optimisations.

```
type NumCaptures = Int
type Comp = (NumCaptures, Arity, OPT)
type Table = HashMap Comp CompId
type NextLvl = Lvl
type OptEnv = [Lvl]
type OptState = State Table (OPT, CapturesAnf, Env)
opt :: OptEnv -> NextLvl -> ANF -> OptRes
opt optEnv nextLvl = \case
  ARet var ->
    return (ORet var, [var], optEnv)
  ALam name params body cont -> do
    let arity = length params
    (bodyId, bodyCptrs) <- optComp (nextLvl : optEnv) arity body
    let lamConstr = OLam name params (bodyCptrs, bodyId)
    optCont lamConstr bodyCptrs cont
  AApp name func args cont ->
     optCont (OApp name func args) (func : args) cont
  AMatch cond nameL contL nameR contR ->
    let splitData = (cond, nameL, contL, nameR, contR)
     in optMatch (nextLvl : optEnv) nextLvl splitData
  AIntLit name lit cont ->
    optCont (OIntLit name lit) [] cont
  ...
 where
    optCont :: (OPT -> OPT) -> [Lvl] -> ANF -> OptRes
    optCont constr captures cont = do
      (cseCont, cptrsCont, envCont) <-
  opt (nextLvl : optEnv) (nextLvl + 1) cont
      let cptrs = addCaptures cptrsCont captures
      let cptrs' = filter (/= length optEnv) cptrs
      return (constr cseCont, cptrs', envCont)
```

Listing 17 : A simplified version of the `opt` function, which only performs CSE while lowering from ANF-IR to OPT-IR. It uses the function `addCaptures` which adds entries to the capture list. The helper functions `optComp` and `optMatch` are shown in Listing 18 and Listing 19.

For each let-binding in the ANF term, the conversion procedure is executed recursively on the continuation. So, when converting the term `let t = ...; u`, it begins by converting `u` before converting the let-binding `t`. A list is maintained during this conversion, keeping track of the free variables of `u`. If `t` was a free variable in `u` it is then removed from this list since it is now no longer a free variable.

The function `optComp` shown in Listing 18 is used for converting computations. When converting the term `let f = λx. comp; u`, the continuation `u` is converted first, then the computation `comp` is both converted and renamed. From this we immediately get the free-variable list for `comp` which can be used to create or retrieve an entry in the computation table. Afterwards, both the free-variable lists are combined (after removing `f` if it was included in either list) to get the free-variable list for the whole term.

```haskell
type CompRes = State Table (CompId, Captures)
optComp :: OptEnv -> Arity -> ANF -> CompRes
optComp optEnv arity comp = do
  let optEnv' = reverse (take arity [0 ..]) ++ optEnv
  let argLvls = take arity [length optEnv ..]
  (cse, cptrs, optEnv'') <- opt optEnv' arity comp
  let cptrs' = filter (`notElem` argLvls) cptrs
  let cse' = optRename optEnv'' cptrs' cse
  compId <- addComp cse' cptrs' arity
  return (compId, cptrs')
```

Listing 18 : Simplified helper function for performing CSE an `ANF` computation, converting it into `OPT` and potentially updating the computation table. This function is used in the `opt` function defined in Listing 17.

This is used both for converting the computation body from an `ALam`, and in the `optMatch` function shown in Listing 19 for converting the branches in an `AMatch`.

```haskell
type MatchData = (Lvl, Name, ANF, Name, ANF)
optMatch :: OptEnv -> NextLvl -> MatchData -> OptRes
optMatch optEnv nextLvl (cond, nameL, contL, nameR, contR) = do
  let splitVarLvl = length optEnv - 1
  (lId, lCptrs) <- optComp optEnv 0 contL
  (rId, rCptrs) <- optComp optEnv 0 contL
  let split = OMatch cond nameL (lCptrs, lId) nameR (rCptrs, rId)
  let captures = addCaptures rCptrs (cond : lCptrs)
  let captures' = filter (/= splitVarLvl) captures
  return (split, captures', optEnv)
```

Listing 19 : Simplified helper function for performing CSE on an `AMatch`, converting it into `OPT` and potentially updating the computation table. This function is used in the `opt` function defined in Listing 17.

The `optRename` function used in Listing 18 performs the “renaming” translation described above. It is implemented as shown in Listing 20.

```haskell
optRename :: OptEnv -> Captures -> OPT -> OPT
optRename optEnv captures = mapLvls convertLvl
 where
  convertLvl lvl
    | lvl < 0 = lvl
    | lvl `elem` captures = fromJust $ elemIndex lvl captures
    | otherwise = lookup optEnv lvl + length captures
```

Listing 20 : The “renaming” translation performed as the second step of the CSE conversion. It uses the function `mapLvls` which maps over an `OPT`, applying some function to each `Lvl`. The implementation for this function is straightforward and is therefore omitted.

There are three distinct cases for each `lvl`:

- `lvl < 0` — As discussed in Section 2.2, this means that `lvl` represents a built-in function. It should therefore not be modified.

- `lvl `elem` captures` — `lvl` corresponds to a let-binding defined outside of the current scope. It should be updated to point to the appropriate element in the capture list.
- `otherwise` — `lvl` corresponds to a let-binding defined inside of the current scope. The environment created in Listing 17 is used to update the let-binding.

3.2.7 Lambda De-duplication

As previously mentioned, the `OLam` constructor only stores a capture list and computation id, which makes it trivial to compare two lambda bindings. This property is used during the CSE pass to eliminate duplicated lambda bindings that perform the same computation and binds the same variables. To do this, we keep track of the let-bound lambdas in scope at any point of the program during the conversion. When encountering a lambda we check if an identical lambda is already in scope. If so, all future references to this lambda is redirected to the one already in scope.

To clarify what this means, here is a simple example:

```
Duplicate lambdas          No duplicate lambdas         Computation table

let a = 1                  let a = 1                 comp0: {a} λx. x + a
let f = comp0 {a}          let f = comp0 {a}
let g = comp0 {a}    ⇒     let b = f 2
let b = f 2                let c = f 3
let c = g 3                ...
...
```

In order to perform this optimisation some modifications have to be made to the `opt` function shown in Listing 17. First, a new environment (`LamEnv`) is added, which maps de-duplicated lambda let-bindings to the first instance of an identical let-binding. Second, a computation map (`CompMap`) is added, which keeps track of which lambda let-bindings exists in the current scope. These are used in the `ARet` and `ALam` cases of the `opt` function as shown in Listing 21.

```
type LamEnv = [Lvl]
type CompMap = HashMap (CompId, Captures) Lvl
opt :: LamEnv -> OptEnv -> CompMap -> NextLvl -> ANF -> OptRes
opt lamEnv optEnv compMap nextLvl = \case
  ARet var ->
    let var' = lookup lamEnv var
     in return (ORet var', [var'], optEnv)
  ALam name params body cont -> do
    let arity = length params
    let lamEnv' = length lamEnv : lamEnv
    let optEnv' = nextLvl : optEnv
    (bodyId, bodyCptrs) <- optComp lamEnv' optEnv' compMap arity body
    case lookup (bodyId, bodyCptrs) compMap of
      (Just lamLvl) ->
  opt (lamLvl : lamEnv) optEnv' compMap nextLvl cont
      Nothing ->
  let compMap' = insert (bodyId, bodyCptrs) nextLvlAnf compMap
      lamConstr = OLam name params (bodyCptrs, bodyId)
   in optCont lamConstr bodyCptrs compMap' cont
  ...
  where
    optCont :: (OPT -> OPT) -> [Lvl] -> CompMap -> ANF -> OptRes
    optCont constr captures compMap' cont =
      ...
```

Listing 21 : Modification of the `opt` function shown in Listing 17 to perform lambda de-duplication.

3.2.8 Downstream Function Compilation

At some point, the computations in the global computation table must be compiled into the program. This is considered out of scope for this thesis however. We can consider the following three cases for each computation in the table:

1. The computation is never used in the program: The computation can be safely ignored, and will be removed in the dead code elimination pass (see Section 3.3).
2. The computation is only tail called: The computation can later be compiled into a local label that can be goto'd.
3. The computation is not only tail called: The computation must be *lambda lifted* [22] by moving the computation to top level scope and including the capture annotations into its parameter list.

Source: ⟹ Lambda lifted:

```
let f = λx.                    let g = λx y. x + y
  let g = λy. x + y            let f = λx.
  let a = g 1                    let a = g x 1
  a + 2                          a + 2
```

3.3 Dead Code Elimination

There is no point in keeping code that has no effect on the final result of the program; so called *dead code*. At best, such code will unnecessarily inflate the size of the executable and increase compile times. At worst, it could result in an arbitrarily large runtime performance degradation, caused by performing unnecessary computations and discarding the result.

While performing dead code elimination is guaranteed to have no effect on the result of well-behaved programs, it does not preserve termination behaviour. Specifically, non-terminating programs might become terminating after this optimisation pass, but not the other way around. Shown below is an example of an infinitely looping program which terminates after the dead code is eliminated.

Source:

```
let f = λx. f x
let d = f 0
123
```

$\Longrightarrow$

Post dead-code elimination:

```
123
```

3.3.1 Dead Code Elimination Implementation

The procedure described in Section 3.2.6 for the OPT-IR conversion keeps track of the free variables at each point in the program. This information can be used perform dead code elimination. Whenever a let-binding `let t = ...; u` is being converted, we check if `t` is in the free variable list of `u`. If so, this means that `t` is used in `u`, so we keep the let-binding. Otherwise, it is discarded.

To perform dead code elimination, the `optCont` function from Listing 17 is modified as shown in Listing 22.

```
opt :: OptEnv -> NextLvl -> ANF -> OptRes
opt optEnv nextLvl = \case
  ...
  where
    optCont :: (Opt -> Opt) -> [Lvl] -> ANF -> OptRes
    optCont constr captures cont = do
      let optEnv' = nextLvl : optEnv
      (cseCont, cptrsCont, envCont) <- opt optEnv' (nextLvl + 1) cont
      let nextAnfLvl = length optEnv
      if nextAnfLvl `elem` cptrsCont
  then let cptrs = addCaptures cptrsCont captures
           cptrs' = filter (/= nextAnfLvl) cptrs
        in return (constr cseCont, cptrs', envCont)
  else let envCont' = updateNewBindings (-1) optEnv' envCont
        in return (cseCont, cptrsCont, envCont')
```

Listing 22 : Updated version of the `optCont` function from Listing 17 that performs dead code elimination. The `updateNewBindings` function simply subtracts `1` from all the elements in the environment corresponding to let-bindings in the environment.

3.4 Match Reductions

Conditional branching is an essential feature of most programming languages. Occasionally however, some abstraction might introduce unnecessary pattern matches. For example, it might create two identical branches, or two branches that become identical after some optimisation passes. In these cases, the branches cause both code bloat and performance degradation. Match reductions are therefore a good candidate for an optimisation pass.

We have considered two types of match reductions in this thesis:

- Constant match reduction: A case match with two identical branches that do not depend on the unwrapped condition-variable can be reduced by replacing it with the branch computation. Optimisations are applied to the branches before comparing them, so they are reduced recursively. An example of this is shown below:

```
match c with
  Left x  -> match (Left 1) with
    Left y  -> 1 + 2 + 3
    Right y -> 1 + 2 + 3              ⇒              1 + 2 + 3
  Right x -> match (Right 1) with
    Left y  -> 1 + 2 + 3
    Right y -> 1 + 2 + 3
```

- Identity match reduction: If the two branches both return the re-wrapped match-variable then they can be reduced to directly returning the match-variable.

```
match c with
  Left x  -> Left x            ⇒                       c
  Right x -> Right x
```

3.4.1 Match Reduction Implementation

The representation described in Section 2.3 is used for the CSE optimisation pass because it allows checking for equality of computations. This property is also useful for the case-split reduction optimisation pass, which is performed at the same time as CSE and dead code elimination.

When encountering a match statement in the OPT-IR conversion described in Section 3.2.6, the two branches are processed as computations. We therefore get a list of free variables for each branch, which will include the unwrapped condition-variable if it is used in the branch body. Using this information, it is straightforward to check the two types of match reductions described in section Section 3.4. This optimisation is performed in the updated `optMatch` function shown in Listing 23.

```
optMatch :: OptEnv -> NextLvl -> MatchData -> OptRes
optMatch optEnv nextLvl (cond, nameL, contL, nameR, contR) = do
  let splitVarLvl = length optEnv - 1
  (lCse, lCptrs, lEnv) <- opt optEnv 0 contL
  (rCse, rCptrs, rEnv) <- opt optEnv 0 contR
  let lCse' = optRename lEnv lCptrs lCse
  let rCse' = optRename rEnv rCptrs rCse
  let lIdent = (OInl "" 0 $ ORet 1, [splitVarLvl])
  let rIdent = (OInr "" 0 $ ORet 1, [splitVarLvl])
  if notElem splitVarLvl lCptrs && (lCse', lCptrs) == (rCse', rCptrs)
    then let optEnv' = updateNewBindings nextLvl optEnv lEnv
       captures = filter (/= splitVarLvl) $ addCaptures rCptrs lCptrs
    in return (lCse, captures, optEnv')
    else if (lCse', lCptrs) == lIdent && (rCse', rCptrs) == rIdent
      then return (ORet cond, [cond], drop 1 optEnv)
      else do
    lId <- addComp lCse' lCptrs 0
    rId <- addComp rCse' rCptrs 0
    let split = OMatch cond nameL (lCptrs, lId) nameR (rCptrs, rId)
    let captures
      = filter (/= splitVarLvl)
      $ addCaptures rCptrs (cond : lCptrs)
    return (split, captures, optEnv)
```

Listing 23 : Updated version of the `optMatch` function from Listing 19 that performs match reductions.

4
Testing and Evaluation

To verify the integrity of every transformation pass, each IR has an interpreter corresponding to the operational semantics and a QuickCheck `Arbitrary` instance. This is used on every transformation pass to ensure that the evaluated result of valid code is never changed by any pass for terminating programs.

4.1 QuickCheck

QuickCheck is a Haskell library for automatic property based testing written by John Hughes and Koen Claessen in 2000 [23]. It allows the user to write a set of laws their code needs to follow and leave QuickCheck to automatically generate tests to try to find a counter example. If a counter example is found QuickCheck will also try to reduce it to the smallest possible case.

The test cases are generated through the provided `Arbitrary` type class. This is implemented on the basic standard library types but must be implemented manually for any user provided types. Such an implementation requires the `arbitrary :: Gen a` function and optionally the `shrink :: a -> [a]` function to try to automatically minimise failing test cases.

To test the correctness of a test-case, it is evaluated both before and after applying the optimisations, and the results are compared to check that they are identical.

4.1.1 Test Case Generation

The `Surface` instance is structured as follows: At the top level we take a size parameter. This is the target amount of nodes in the top level and we will generate `LetFun` and `LetVal` definitions until we run out. The size of expressions in `LetVal`s and function bodies in `LetFun` are also limited by it, but with a capped ceiling. To encourage a wider spread of complex types and not just the common and easily generated literals we pick a target type up front. If no variable of this type exists when the size limit is reached a minimal expression of the requested type is generated, reusing as many preexisting variables as possible in sub-expressions.

As unstructured random code usually does not generate valid base cases we do not allow for recursion in the generator as that would just lead to non termination. To alleviate the gap in input space this creates we have a number of handwritten test cases instead.

The language is exception-free for well-typed code. The only construct which should have an undefined result, division by zero, has been defined as zero. However, invalid code is still representable and because tests cannot run forever we have a recursion limit at interpretation time. The interpreter can therefore still crash.

We do not provide a `shrink` implementation as this would be a re-implementation of Dead Code Elimination, which is one of the passes we want to test, rendering the test pointless.

4.2 Properties

The automatic tests, or properties, take randomly generated code and make sure some specified property is consistent before and after a lowering step. They generally follow the pattern in Listing 24.

```haskell
prop_anfLowering_consistentValue :: Surface (Maybe TypeAnnotation) -> Property
prop_anfLowering_consistentValue t =
  case typeCheckSurface t of
    Nothing ->
      counterexample omittedErrorMessage
      $ property False
    Just checked ->
      let anf = anf checked
          term_result = interpretTerm checked
          anf_result = interpretAnf anf
      in counterexample omittedErrorMessage
          . property
          $ case (term_result, anf_result) of
            (Just tRes, Just aRes) -> compareValues tRes aRes
            (Nothing, _) -> True
            _ -> False
```

Listing 24 : Example property checking lowering from Surface to ANF-IR, checking that type correct Surface language code evaluates to the same thing even after being lowered to ANF-IR.

These properties exist for every step in the pipeline and generally check that types as described in Section 2.1.1 or evaluated result as described by the operational semantics in Section 2.1.3 stay the same. We additionally check that there are no duplicated computations after lowering to OPT-IR.

4.3 Manual Testing

In addition to the tests generated by QuickCheck we also have a handwritten test suite. This covers tests that were written to develop against and regression tests based on generated tests that failed previously.

The manual tests we wrote during development were useful to test the system after the optimisation passes had been implemented. While there are far fewer of them than what can be generated automatically, including these tests ensures that certain edge cases are always covered. Manual tests are therefore a good complement to the automated test-cases, which might find edge-cases that we have missed, but at the same time do not provide any guarantees that specific cases are tested.

Another advantage of the manual tests is that they allow us to test different properties of correctness more thoroughly. For instance, we include some test-cases for which we explicitly specify how the code should look after performing the optimisations. For these types of manual tests we can ensure that the optimisations correctly optimise the code by, for example, removing dead code and common subexpressions. This is a more thorough test than just checking that evaluation behaviour is preserved, as we do for the automated tests.

4.4 Performance Impact

None of our transformations have any way of producing slower code than was originally provided. This is because we have chosen simple optimisations that always reduce code size and remove unnecessary computation.

We cannot, however, produce any numbers on any potential performance improvement as there are no real world programs written in any language using Kovács type theory. To get any statistically significant performance metrics we would need to evaluate our optimisations on many different programs of various sizes that correspond to the types of code you might write in real programs. While our generated programs are useful for checking the correctness of our implementation, and could potentially give some insight into the performance impact of our optimisations, any tests using such generated programs would be highly unrepresentative.

5 Conclusion

We have specified and implemented some optimisations for a first-order simply typed object-level language using CFTT. The optimisations are performed on an ANF intermediate representation which is constructed from the surface language AST. To verify the correctness of our optimisation stages, we use QuickCheck to run it through both a suite of both manually specified and automatically generated tests.

5.1 Discussion

There are many optimisations that could potentially be included in the language specification. However, there is no use including an optimisation if that optimisation is too complicated for programmers to be able to effectively reason about how it will affect their code. Such optimisations would simply create clutter, and are better suited for traditional black-box optimisation passes. In summary, there is clearly a trade-off between using more complicated and powerful optimisation passes, versus using simpler but perhaps less effective ones.

Our implementation shows that a fast implementation is possible, even if ours is lacking due to sticking with default Haskell data structures. A serious implementation should be faster and stick to reasonable complexity requirements, $O(n)$, meeting our goal of being fast.

Restricting ourselves to simple types means that optimisation passes can be simpler and more straightforward. This makes them easier to define in the language specification, and easier to understand for users of the language. Limiting the language to simple types would normally be too great of a restriction, as this would prohibit programmers from using most of the common abstractions that are commonly relied upon in functional programming languages. However, we can largely avoid this as CFTT provides an alternative solution to that problem.

This makes CFTT an ideal choice for a programming language with explicitly defined optimisation passes in the language specification, since it uses a simply typed object level, while many language features are still available through meta-programming.

5.2 Comparison to GHC and the OCaml Compiler

Our solution does not exist in a vacuum. The two major functional languages, OCaml and Haskell, also have their own solutions to optimising code.

5.2.1 Flambda2 (OCaml internals)

Flambda2 is an IR to be used in the OCaml compiler introduced in 2023 to improve optimisation, particularly around inlining [24]. It is targeted at being a drop in replacement for Flambda 1 and tries to deal with patterns common in idiomatic OCaml. It is written with "Double-Barrelled" CPS, a form of continuation passing style that also supports exceptions [25] by passing a second continuation to each function to be called when an exception is raised. There are only five types of main nodes, plus an additional unreachable marker, in the Flambda2 IR [24]. Closures are explicit, allowing them to reliably inline functions without additional alias analysis.

Comparing to our IR we do not support any form of exception handling, inlining or even tail calls. Our lambdas are not explicit, but rather pass in all captures at each call site. As our type system lacks higher order functions this does not cause any issues with closures escaping the scope of the captured variables.

5.2.2 GHC Internals

GHC uses an IR that is similar to ANF but extended with something they call *join points* [15]. Join points exist to solve the problem of commutative conversions accidentally creating closures that need to be allocated. Inlining functions can often end up resulting in code like:

```
if (if e1 then e2 else e3) then e4 else e5
```

which after a commutative conversion looks like:

```
if e1
  then (if e2 then e4 else e5)
  else (if e3 then e4 else e5)
```

If `e4` and `e5` are small this is not a problem, but if they are large this can end up with significant code bloat. To solve this `e4` and `e5` can be factored out.

```
let { j4 () = e4; j5 () = e5 }
in if e1
    then (if e2 then j4 () else j5 ())
    else (if e3 then j4 () else j5 ())
```

As long as j4 and j5 are tail called they can be lowered to plain jumps rather than full function calls [15]. Keeping them as function calls in the IR opens up the risk of other optimisations changing them and turning them into normal, non-tail, calls, which would then require an allocation for the function closures. To prevent this the GHC IR has dedicated constructs for binding and calling join points.

Because this is not a part of the Haskell language but rather an implementation detail of the GHC compiler join points that were not generated by the optimiser must be inferred through a process called *contification* [15].

Because of fully saturated calls and non-allocating lambda expressions we do not need join points in our IR to avoid allocating continuations. This does not help us with stack reordering (when continuations have the same variables on the stack but in a different order), but we hope future work with an LLVM [26] backend would be able to inline any continuations and avoid stack manipulations.

5.3 Guarantees

The transformations in this paper are guaranteed to always apply when applicable. The only other guaranteed optimisations we can find are in C++ and Scheme, with (N)RVO and TCO.

5.3.1 Copy Elision in C++17

The C++ standard library is primarily built upon *value semantics* as described in [27], a language design paradigm where correctness is ensured and local reasoning provided not through purity, but through having mutable objects but no references, preferring to copy them when needed. As these copies can be expensive C++ allowed the copies to be elided in certain contexts (replacing them with out-pointers to construct them in-place) when this could not be observed.

C++17 went one step further and requires this optimisation rather than just allowing it.

```
struct Heavy { /* expensive contents to copy */ };
Heavy make_heavy() {
  Heavy a_new_heavy = { ... };
  return a_new_heavy;
}

Heavy my_heavy = make_heavy();
```

In the above snippet, when interpreted as C++14, there are three constructor calls, one to construct the `Heavy` and then two copies: one to the return expression and then another copying from the return to the new `my_heavy`. When interpreted as C++17 however, these copies must now be elided and only one constructor call is made and the `Heavy` is constructed in-place as `my_heavy` [7].

This is guaranteed by redefining the value categories of C++ in the definition of its abstract machine. The previous definition of a temporary object (C++: *prvalue*) was changed so that it only describes the recipe of the temporary object and the return object's storage must be allocated before the function is called. What used to say "Create an object, then copy it to the right place" now says "Create space for the final object, then call the function that creates it".

The elision is thereby guaranteed not through text saying that it can or must happen, but through a redefinition of the language semantics so that the temporary was always a reference to the final object.

5.3.2 Tail Call Elimination in Scheme

Pure functional languages cannot express loops as they by definition require mutable variables to, at the very least, be the loop counter. Iteration is therefore expressed through recursion instead.

This, however, leads to the issue of very long running loops leading to stack overflows when the program runs out of memory. However, if the old stack frame is never needed again it could be reused and the stack can remain at a constant size. This happens when the recursive call is the very last thing that happens before the function returns [28].

Scheme achieves its guarantees not through any complicated categorisation of values, but through a set of rewrite rules they believe will cover all cases, along with a shorter rationale and citation to a full paper on why tail call elimination works and achieves the savings they claim [6].

We believe our rewrite rules function the same as the ones in the Scheme standard.

5.4 Future Work

In this thesis, we have investigated some optimisations that seem to be well suited for our use cases, which were discussed in Section 3. These are, however, not the only optimisations that could potentially be relevant. Exploring some other optimisations could therefore be a topic for future work.

As mentioned in Section 3.2.1 we do not do any de-duplication of value bindings. Defining the semantics around lifetimes of bindings in a way that balances less work with the additional memory requirements is a good candidate for future work.

Another extension that could be made to this work is described in Section 2.1.1. Specifically, it is possible to extend the language with explicit type formers for closures.

Also, we have only implemented part of the entire compiler pipeline. In particular, we have implemented the middle-end, but are missing both the front- and back-end. A future project could be to implement these to get a complete compiler.

Further, we have not provided any formal proofs in this paper. As discussed in Section 4, our implementation of the optimisation passes have been tested extensively on both manual and automatically generated test-cases. While they give a certain degree of confidence in the correctness of the implementation, they are not infallible and do not test for every single possible edge-case one might encounter in real world usage.

For use in a real compiler, it would therefore be useful to provide proofs of correctness along with the implementation. This is outside the scope of our work, but in the future it might be worthwhile to recreate the implementation in a proof assistant such as Agda, along with appropriate proofs.

If real world programs start being written in our setting it would be highly useful to provide benchmarks on them.

6

Bibliography

[1] J. W. Backus *et al.*, “The FORTRAN automatic coding system,” in *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, 1957, pp. 188–198.

[2] R. Power and A. Rubinsteyn, “How fast can we make interpreted Python?,” *arXiv preprint arXiv:1306.6047*, 2013, [Online]. Available: https://arxiv.org/abs/1306.6047

[3] W. Bugden and A. Alahmar, “Rust: The Programming Language for Safety and Performance,” *arXiv preprint arXiv:2206.05503*, 2022, doi: 10.48550/arXiv.2206.05503.

[4] Z. Gong *et al.*, “An empirical study of the effect of source-level loop transformations on compiler stability,” vol. 2, no. OOPSLA, Oct. 2018, doi: 10.1145/3276496.

[5] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August, “Compiler optimization-space exploration,” in *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, 2003, pp. 204–215.

[6] S. S. Committee, “Revised^7 Report on the Algorithmic Language Scheme (R7RS-Small),” 2013. [Online]. Available: https://standards.scheme.org/official/r7rs.pdf

[7] R. Smith, “Guaranteed Copy Elision through Simplified Value Categories.” [Online]. Available: https://open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0135r0.html

[8] A. Kovács, “Staged compilation with two-level type theory,” *Proceedings of the ACM on Programming Languages*, vol. 6, no. ICFP, pp. 540–569, 2022.

[9] A. Kovács, “Closure-Free Functional Programming in a Two-Level Type Theory,” *Proceedings of the ACM on Programming Languages*, vol. 8, no. ICFP, pp. 659–692, 2024.

[10] P. Chambart, V. Laviron, G. Bury, N. Courant, and D. Pinto, “Flambda2 Ep. 1: Foundational Design Decisions.” Mar. 19, 2024.

[11] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen, “The essence of compiling with continuations (with retrospective),” in *20 Years of the ACM SIGPLAN Con-*

ference on Programming Language Design and Implementation 1979-1999, A Selection, K. S. McKinley, Ed., ACM, 1993, pp. 502–514. doi: 10.1145/989393.989443.

[12] S. Kyletoft and E. L. Sternvik, “masters-thesis.” [Online]. Available: https://github.com/SKyletoft/masters-thesis

[13] R. Hindley, “The Principal Type-Scheme of an Object in Combinatory Logic,” *Transactions of the American Mathematical Society*, vol. 146, pp. 29–60, 1969, Accessed: Jun. 04, 2025. [Online]. Available: http://www.jstor.org/stable/1995158

[14] G. Kahn, “Natural semantics,” in *STACS 87*, F. J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 22–39.

[15] L. Maurer, P. Downen, Z. M. Ariola, and S. L. P. Jones, “Compiling without continuations,” in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, A. Cohen and M. T. Vechev, Eds., ACM, 2017, pp. 482–494. doi: 10.1145/3062341.3062380.

[16] J. Breitner, “Call Arity,” *Comput. Lang. Syst. Struct.*, vol. 52, pp. 65–91, 2018, doi: 10.1016/J.CL.2017.03.001.

[17] S. Marlow and S. L. P. Jones, “Making a fast curry: push/enter vs. eval/apply for higher-order languages,” in *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*, C. Okasaki and K. Fisher, Eds., ACM, 2004, pp. 4–15. doi: 10.1145/1016850.1016856.

[18] O. Chitil, “Common subexpressions are uncommon in lazy functional languages,” in *Implementation of Functional Languages*, C. Clack, K. Hammond, and T. Davie, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 53–71.

[19] HaskellWiki contributors, “GHC Memory Management.” [Online]. Available: https://wiki.haskell.org/index.php?title=GHC/Memory_Management&oldid=65457

[20] The Rust Project Developers, “Destructors.” [Online]. Available: https://doc.rust-lang.org/reference/destructors.html

[21] M. Wand and P. Steckler, “Selective and Lightweight Closure Conversion,” in *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, H.-J. Boehm, B. Lang, and D. M. Yellin, Eds., ACM Press, 1994, pp. 435–445. doi: 10.1145/174675.178044.

[22] T. Johnsson, “Lambda Lifting: Transforming Programs to Recursive Equations,” in *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings*, J.-P. Jouannaud, Ed., in

Lecture Notes in Computer Science, vol. 201. Springer, 1985, pp. 190–203. doi: 10.1007/3-540-15975-4_37.

[23] K. Claessen and J. Hughes, “QuickCheck: a lightweight tool for random testing of Haskell programs,” in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, M. Odersky and P. Wadler, Eds., ACM, 2000, pp. 268–279. doi: 10.1145/351240.351266.

[24] M. S. V. L. P. Chambart and M. Shinwell, “Efficient OCaml Compilation with Flambda 2,” *OCaml. https://icfp23.sigplan.org/details/ocaml-2023-papers/8/Efficient-OCaml-compilation-with-Flambda-2*, 2023.

[25] H. Thielecke, “Comparing control constructs by double-barrelled CPS,” *Higher-Order and Symbolic Computation*, vol. 15, pp. 141–160, 2002.

[26] “The LLVM Compiler Infrastructure Project — llvm.org.” [Online]. Available: https://llvm.org/

[27] A. Stepanov and P. McJones, *Elements of Programming*, Authors' Edition. Palo Alto and Mountain View, USA: Semigroup Press, 2019, pp. 49–50. [Online]. Available: https://www.elementsofprogramming.com/

[28] W. D. Clinger, “Proper tail recursion and space efficiency,” in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, in PLDI '98. Montreal, Quebec, Canada: Association for Computing Machinery, 1998, pp. 174–185. doi: 10.1145/277650.277719.