UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Bachelor thesis

Jacob Aleksandar Siegumfeldt (dzb977) & Laust Kjæp Dengsøe (frh993)

Optimizing Futhark's Type Checker

Supervisor: Troels Henriksen

Handed in: June 10, 2025

Abstract

Having a systematic way of inferring the types of expressions is a well-known benefit of having a Hindley-Milner type system in a programming language, yet there exist various approaches to solving the type inference problem. Futhark uses an offline approach where all constraints of an expression are inferred first and then solved afterwards; however, the constraint solver is not currently implemented in an optimal way.

This thesis explores a common approach of solving type constraints that uses an underlying union-find data structure along with heuristics like path compression and union-by-weight to provide a good asymptotic running time, and we propose a new implementation of the constraint solver in Futhark using this approach. To evaluate this updated implementation, we benchmarked programs and compared our implementation with the existing implementation. Our initial results show that our implementation does not generally provide a better running time in practice, but despite this, we argue that we have laid the groundwork for doing type inference in a more efficient manner than the current approach.

Contents

1 Introduction

Originally, Damas and Milner [1] proposed the so-called *Algorithm W* as a way to infer types of expressions in the original Hindley-Milner type system. Algorithm W uses an online approach where the process of generating constraints from expressions is intertwined with the process of solving them. An alternative approach, which Futhark has also switched to use, is an offline approach where all constraints generated by an expression are inferred first and only solved afterwards. Futhark's current implementation, however, is much less efficient than it potentially could be.

In this thesis, we explore the theoretical foundation of solving constraints with an offline approach, as well as how it can be done in an efficient way using a union-find data structure with heuristics like path compression and union-by-weight. Our aim is then to provide an implementation that uses these optimizations to make the constraint solver more efficient.

As we will outline, our updated implementation does not currently yield a better performance, in general, than the original implementation. Despite this, our hope is that our contributions have laid the groundwork for solving type constraints in an efficient way.

All the code for our proposed implementation is based on the `automap` branch in the original Futhark repository[1]. Our proposed implementation, benchmarks, and tests can be found in the following GitHub repository:

`https://github.com/jacobgummer/futhark/tree/automap-tysolve-optim`

[1] `https://github.com/diku-dk/futhark/tree/automap`

2 Background

2.1 Solving Type Constraints

There are different ways of solving the type inference problem in programming languages with a Hindley-Milner type system. One approach is to intertwine the process of generating constraints and solving them, that is, an *online* approach. For this project, we take a different approach where every constraint that can be inferred from a given expression is generated first and then solved afterwards. In other words, our focus is on presenting an *offline* algorithm for solving type constraints.

2.1.1 Types

To build up the theoretical foundation for how type constraints can be solved, we start by considering a small language, similar to the Damas-Milner (or Hindley-Milner) type system originally proposed in Damas and Milner [1], with its types being defined by the grammar

$$\tau ::= \texttt{int} \mid \tau_1 \to \tau_2 \mid \alpha$$

`int` simply denotes the integer type, $\tau_1 \to \tau_2$ denotes function types, and α denotes *type variables* which are just placeholders for types. In the following sections, we let $\mathcal{T}$ be the set of all types, and we let α, β, and γ range over type variables.

2.1.2 Type Constraints

Formally, we define a *type constraint* c as

$$c ::= \tau_1 \equiv \tau_2$$

where τ_1 and τ_2 are both types, while a *constraint set* C is a set of such equalities. For our purposes, we do not concern ourselves with how constraints are generated/inferred, only that they are inferred from expressions and that they must be *solvable* (as explained in section 2.1.4) for an expression to be typable.

2.1.3 Type Substitutions

To be able to *solve* a constraint, we also introduce *type substitutions*. Informally, a type substitution S is a mapping from type variables to types. However, formally

we define a substitution $S : \mathcal{T} \to \mathcal{T}$ mapping types to types as follows:

$$
\begin{aligned}
S(\texttt{int}) &= \texttt{int} \\
S(\tau_1 \to \tau_2) &= S(\tau_1) \to S(\tau_2) \\
S(\alpha) &= \begin{cases} \tau & \text{if } (\alpha \mapsto \tau) \in S \\ \alpha & \text{otherwise} \end{cases}
\end{aligned}
$$

We can also extend the notion of substitution to constraints and sets of constraints in the following manner:

$$
\begin{aligned}
S(\tau_1 \equiv \tau_2) &= S(\tau_1) \equiv S(\tau_2) \\
S(C) &= \{S(c) \mid c \in C\}
\end{aligned}
$$

Note that all substitutions are performed *simultaneously*: For instance, given the substitution $S = [\alpha \mapsto \texttt{int}, \beta \mapsto \texttt{int} \to \alpha]$, applying S to β doesn't yield the type $\texttt{int} \to \texttt{int}$ but $\texttt{int} \to \alpha$.

Further, given two substitutions S and S', we write $S \circ S'$ to denote composition of the two substitutions such that $(S \circ S')(\tau) = S(S'(\tau))$.

2.1.4 Unification

We say that a type substitution S *unifies*, or *solves*, the constraint $\tau_1 \equiv \tau_2$ if $S(\tau_1)$ is *syntactically*[2] equal to $S(\tau_2)$, as originally described in Robinson [4]. Extending the notion of solvability to constraint sets, a type substitution S solves a constraint set C if S solves every type constraint in C; we call such a substitution S a *solution* or *unifier* for C.

As there might be multiple unifiers for a constraint set [5], we define $\mathcal{U}(C)$ to be the set of all unifiers for C. Furthermore, a type substitution ρ is called a *most general unifier* (MGU) of a set of type constraints C if $\rho \in \mathcal{U}(C)$ and for every $S \in \mathcal{U}(C)$ there exists a type substitution S' such that $S = S' \circ \rho$. An MGU ρ can thus be interpreted as the simplest substitution that solves a type constraint (or a set of them) since any other unifier is simply a refinement of ρ. Note that there can also be multiple MGUs: For instance, if both α and β are type variables, $\rho = [\alpha \mapsto \beta]$ and $\rho' = [\beta \mapsto \alpha]$ are both MGUs of the constraint set $\{\alpha \equiv \beta\}$.

The are different reasons for why we're interested in a *most general* unifier of some constraint (set). Evidently, we don't want to limit what type a type variable might represent. The constraint $\alpha \equiv \beta$ could also be solved with the substitution $S = [\alpha \mapsto \texttt{int}, \beta \mapsto \texttt{int}]$ but with this substitution, we have constrained these

[2] There also exists a notion of *equational unification* where one is interested in finding a substitution that makes types equal modulo some equational theory E [2]. This is relevant for sum types, as Schenck [3, p. 21] describes, but we won't cover it in this section.

type variables to be of type `int` even though there is no need to do so. It might even make it impossible to solve a set of constraints if, say, the set also contained the constraints $\alpha \equiv \gamma$ and $\gamma \equiv \texttt{int} \rightarrow \texttt{int}$ since we cannot unify the types `int` and $\texttt{int} \rightarrow \texttt{int}$.

In type systems that include *type schemes*, that is, type systems that thus allow (parametric) polymorphism, finding an MGU of a constraint set also relates to finding a *principal type scheme* [1], that is, the most "general" type, of an expression. We won't cover the details of what this means since the unification process, in itself, is the same with or without type schemes, but with type schemes, MGUs are even more crucial. As an example, consider the Haskell expression

```
let id = \x -> x in (id "foo", id True)
```

which should generate a tuple with type `(String, Bool)`. If the constraints generated by this expression are not solved carefully, that is, if each constraint isn't solved with an MGU, the type checker might infer `id` to have a type that makes it impossible to apply both strings and booleans (or arguments of any other type) to it. This is an example of *let-polymorphism* where we need to infer a principal type of the first expression in `let`-expressions to make sure that it can be used polymorphically, but it also applies to expressions in general when the underlying type system includes type schemes – and to infer a principal type of an expression, we also need a most general unifier of the constraints generated by the expression.

Summarizing what we've covered so far, to solve a set of constraints C we must find a substitution S that solves C, and an expression e that produces the constraint set C is typable if and only if there exists a solution for C.

2.1.5 A Unification Algorithm

In order to find a (most general) unifier, that is, a unifying substitution S, of a constraint – if one exists – we'd like to have a (partial) function $\texttt{unify} : \tau \times \tau \rightharpoonup S$ that can achieve this. Such a function can be described in an algorithmic manner as outlined below:

```
1  unify τ1 τ2 =
2    case (τ1, τ2) of
3      (τa → τb, τc → τd) →
4        S = unify τa τc
5        S′ = unify S(τb) S(τd)
6        return S′ ∘ S
7      (α, τ) → bind α τ
8      (τ, α) → bind α τ
```

```
    (τ, τ′) →
      if τ = τ′ then
        return []
      else
        fail "types do not unify"

bind α τ =
  if α = τ then
    return []
  else if α ∉ τ then
    return [α ↦ τ]
  else
    fail "occurs check fails"
```

Here, $\alpha \notin \tau$ means that α does not *occur* in τ (often referred to as the 'occurs check' in the literature). To be precise, in this type system, a type variable α occurs in a type τ if and only if τ is a function type on the form $\tau' \to \tau''$ and either 1) α is equal to either τ' or τ'', or 2) α occurs in either τ' or τ'' [6].

In the two cases where either of the types is a type variable α, the check of whether α occurs in τ is necessary since we might otherwise create an infinite type [7] which we aren't able to represent with the simple type system we're basing the unification algorithm on.

The idea is then to call this algorithm for every constraint c in a constraint set C and gradually build up a composition of substitutions. The following algorithm more accurately captures the idea:

```
unifySet ∅ = return []
unifySet {τ₁ ≡ τ₂} ∪ C′ =
  S = unify τ₁ τ₂
  return (unifySet S(C′)) ∘ S
```

It's important that, for every constraint we solve which results in some substitution S, we must apply S to the rest of the constraint set C' since we might have obtained new information that affects how the remaining constraints need to be handled.

Even though this is a valid way to solve type constraints, it should also be evident why it isn't an optimal approach: We have to continually compose and apply substitutions which can be very expensive computationally. In the next subsection, we will thus explore a different approach that uses an underlying data structure well-suited for the unification problem.

2.2 Constraint-Solving Using Union-Find

In offline constraint-solving, the underlying data structure of the constraint solving algorithm is important, since each type variable might be accessed many times based on how often it occurs in the given constraints. As we shall see, the union-find data structure is an efficient data structure for constraint-solving algorithms since it allows for fast lookups of nodes when implemented optimally [8, p. 538]. It is particularly useful for applications that require the grouping of multiple distinct elements into disjoint sets [8, p. 520] as it is the case for type variables being divided into disjoint equivalence classes.

2.2.1 The Union-Find Data Structure

Union-find is a data structure that manages a collection of disjoint sets, meaning that a unique *element* can be part of exactly one *set*. Each set has an associated *representative* which is an element of the given set. The data structure supports three basic operations: `Make(x)`, `Find(x)` and `Union(x,y)`.

`Make(x)` creates a set where `x` is the representative.

`Find(x)` returns the representative of the set that `x` is part of.

`Union(x,y)` is the most "complex" operation as it first finds the representatives of the sets that `x` and `y` are part of, S_x and S_y, by calling `Find(x)` and `Find(y)`. Then, a new set, S, is created containing all the elements in both S_x and S_y yielding $S = S_x \cup S_y$. The representative of S can be chosen arbitrarily among the elements in S or chosen by some property or heuristic.

These are the most basic operations that must be supported by a union-find data structure. However, as we shall see, the specific implementation is also important as it has a significant impact on the time complexity of the operations.

2.2.2 Disjoint-Set Forests and Heuristics

Known implementations of the union-find data structure uses linked-lists or disjoint-set forests. The latter allows for a faster runtime by using two essential heuristics [8, p. 527]; therefore, we will only describe this way of implementing the data structure. When using the disjoint-set forest representation, each *tree* represents a set. *Nodes* corresponds to elements in the sets, and each node also points to a node. A representative of a set is the *root* of a tree, and it is thus its own parent.

Now, the operation of making a set creates a new tree with one single node pointing to itself. The finding of a set follows the parent pointers of the given element until the root is reached. Finally, the union of two sets changes the parent of one of the roots to point to the other root. This implementation is no better

than the linked list implementation. However, by introducing two new heuristics, namely *union-by-weight* and *path compression*, an optimal implementation of the union-find data structure can be achieved. [8, sec. 19.3]

Union-by-weight is a heuristic, that makes the smallest tree point to the larger tree during union. To adhere to this heuristic, each root must keep track of the number of nodes in its tree/set, which we will refer to as the *weight* of the tree. An example of this heuristic can be seen in figure 1.

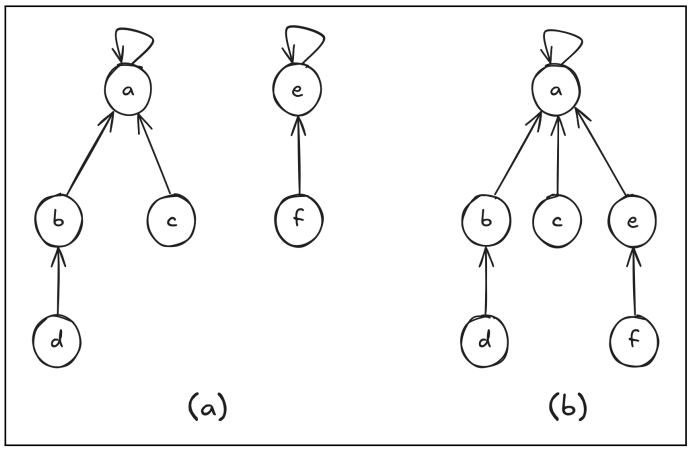


Figure 1: An example of the *union-by-weight* heuristic. **(a)** shows two trees, the one on the left has weight four, and the one on the right has weight 2. **(b)** shows the resulting tree after calling `Union(a,e)`.

Path compression is the heuristic ensuring that the path to the root is as short as possible after visiting a node. This means that the find operation must traverse the path to the root two times. In the first pass, it discovers what node is the root, and in the second pass, it changes the parent pointers of the nodes being traversed to point to the root. Thereby, the path to the root has been *compressed*, since the traversed nodes are now pointing directly at the root. Figure 2 shows an example of this heuristic.

2.2.3 Implementation of Union-Find Operations

In the previous section, the union-find data structure has been described, and the heuristics a union-find data structure must abide by to achieve an optimal implementation have also been described. The following will show how such a data structure could be implemented by slightly modifying the pseudocode provided by [8, p. 530].

The first operation of making a set is very simple and could be implemented as in listing 1. Here, $x.p$ denotes the parent pointer contained in a given node,

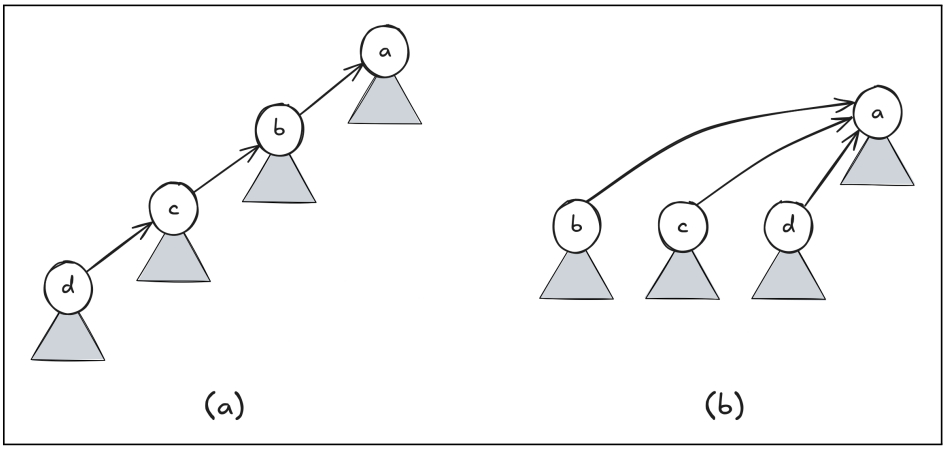


Figure 2: An example of the *path compression* heuristic. The parent pointer of the roots have been omitted for simplicity. **(a)** shows a tree before `Find(d)` is called. The triangles represent subtrees with the shown nodes as roots. **(b)** shows the resulting tree after calling `Find(d)`.

and $x.w$ is the weight of the tree.

```
1: procedure Make(x)
2:     x.p = x
3:     x.w = 1
4: end procedure
```

Listing 1: Make algorithm for union-find data structure.

Finding the root given a node in a tree can be done by simply checking if the given node is a root, and if not recursively find the root and make it the parent of the input node. Lastly, it should return the root. The pseudocode for this is shown in listing 2.

The union of two trees has to ensure that the smaller tree, that is, the tree with the smallest weight, is set to point to the larger tree, as in listing 3.
As the next section will show, the pseudocode provided in this section provides a very good asymptotic running time for the union-find operations.

2.2.4 Time Complexity of Union-Find Operations

The three operations introduced previously form the basis of an implementation of the union-find data structure. A single `Make` operation trivially takes time $O(1)$, that is, constant time. The running times of the two other operations

```
1: procedure Find(x)
2:     if x.p ≠ x then
3:         x.p = Find(x.p)
4:     end if
5:     return x.p
6: end procedure
```

Listing 2: Find algorithm for union-find data structure.

```
1: procedure Union(x, y)
2:     x_r = Find(x)
3:     y_r = Find(y)
4:     if x_r.w > y_r.w then
5:         y_r.p = x_r
6:         x_r.w = x_r.w + y_r.w
7:     else
8:         x_r.p = y_r
9:         y_r.w = y_r.w + x_r.w
10:    end if
11: end procedure
```

Listing 3: Union algorithm for union-find data structure.

are not as simple, though: With the union-by-weight heuristic alone, the running time becomes $O(m \lg n)$ for a sequence of m operations in total of which n are calls to `Make` [8, p. 530]. Further, by only applying the path compression heuristic, the worst-case running time becomes $\Theta(n + f \cdot (1 + \log_{2+f/n} n))$ [8, p. 530]. However, combining both heuristics yields a worst-case running time of $O(mA(n))$ where $A(n)$ is a function that grows *very* slowly. In a strictly mathematical sense, $O(mA(n))$ is a superlinear running time, but in any conceivable application of a disjoint-set data structure, including unification, $A(n) \leq 4$ [8, p. 530], meaning that using both heuristics guarantees a running time that is practically linear in the total number of union-find operations. Furthermore, this gives an amortized running time of $O(A(n))$ for each `Find` operation performed [8, p. 5.39], which, again, is practically constant time.

2.2.5 Union-Find for Unification

As hinted in the beginning of this section, the union-find data structure is an efficient data structure for offline constraint-solving. Section 2.1.5 provided pseudocode for a unification algorithm that returned the substitutions necessary to

solve the given constraints. In Listing 4, a similar unification algorithm, now using the union-find data structure, is provided:

```
unify τ1 τ2 =
    τ1' = findType τ1
    τ2' = findType τ2
    case (τ1',τ2') of
        (α,β) → if α ≠ β then unionTyVars α β
        (α,τ) → bindUF α τ
        (τ,α) → bindUF α τ
        (τa → τb,τc → τd) →
            unify τa τc
            unify τb τd
        (int,int) → do nothing
        otherwise → fail "types do not unify"
```

Listing 4: Unification algorithm using union-find data structure.

As with the function outlined in section 2.1.5, `unify` is called on each constraint in the constraint set. However, before the first call to `unify`, we go through each type variable in the constraint set and make a new tree for each of them in the underlying union-find data structure. In the following, we'll also sometimes use the term *equivalence class* to denote that every node in the same tree represents the same thing. Conceptually, in this context, one can picture an equivalence class as a set of types with at most one non-type variable in it, and if an equivalence class contains a non-type variable, this is the (sole) type associated with the equivalence class; otherwise, an arbitrary type variable in the equivalence class can be chosen to be the representative and, hence, the type associated with this equivalence class.

The root in each tree then contains information such as the weight of the tree as well as a pointer to the parent node. Each root is initialized with its weight set to 1, pointing to itself, and letting the type of the equivalence class represented by this tree simply be the type variable that this root node represents.

Now, for each type in a given constraint, the first thing we do is to call `findType` on it: The `findType` function should work such that, when given a non-type variable as input, it works like the identity function and returns the type itself. If it is, instead, some type variable α, it should return the type associated with the equivalence class that α is part of. Furthermore, to help obtain the asymptotic running time described in the previous subsection, `findType` must perform path compression whenever it is given a type variable as input.

After this, the different cases are similar to the ones in the pseudocode outlined in section 2.1.4. A noteworthy first difference is that this now has a case for when τ_1' and τ_2' are both type variables since, if they're not the same type variable, we need to join their equivalence classes. To do this this in an optimal way, we need `unionTyVars` to use the union-by-weight heuristic, too.

If, instead, exactly one of the types in the constraint is a type variable, we must call `bindUF`: The goal for `bindUF` is to ensure that any future call to `findType` with α as its input returns τ. `bindUF` must also perform the occurs check and it is thus crucial that if τ is a function type, that is, a composite type, every inner type variable α' in τ must be substituted with `findType` α'. An example of how it might go wrong otherwise could be to have the small constraint set $\{\beta \equiv \alpha, \alpha \equiv \beta \to \texttt{int}\}$: After processing the first constraint, we'll have an equivalence class with both α and β in it, and if its corresponding tree has α's node as root, we'll have $\tau_1' = \alpha$ and $\tau_2' = \beta \to \texttt{int}$ when processing the second constraint. If we don't substitute β with α inside τ_2', we'll miss that the occurs check should fail since we're essentially going to create a cyclic substitution otherwise.

The three final cases are then practically identical to how they're handled in the unification algorithm in section 2.1.5, with the only real difference being that this function doesn't output anything itself. This also means that, to find out what a type variable should (possibly) be substituted with when we're done processing the constraints, one would need to look in the union-find data structure and resolve the type this way instead, essentially by calling `findType` α for each type variable α in the constraint set.

Given the favorable asymptotic bounds described in the previous subsection, this pseudocode thus describes an efficient implementation of a unification algorithm by using an underlying union-find data structure with the union-by-weight and path compression heuristics.

2.3 Practical Complications

In practice, many languages include features that introduce complications to how constraints should be solved, some of which we find worth briefly mentioning since they also play a role in Futhark's constraint solving process.

2.3.1 Overloading

Many languages include *overloaded* (or, *ad hoc polymorphic*) operators, that is, operators that are defined for multiple (though not arbitrary) types [9, p. 192]. Consider, for instance, the small Futhark program

```
def f x y = x + y
```

What type should `f` be inferred to have in this case? Since the `+`-operator is overloaded, `x`, `y`, and the return type of `f` could be any numeric type supported in Futhark.[3] In this case, we could just default `f` to have type `i32 -> i32 -> i32`; in other cases, however, it might not be sensible to default the type of some variable or formal parameter, for instance, if you're using it as if it's a record without specifying which type of record it must be. In a more general context, we handle this complication by constraining which types a type variable can be substituted with whenever this type variable represents a type used in an expression with an overloaded operator. With the concrete example above, during the unification process, we must make sure that `x` and `y` can only be substituted with numeric types.

2.3.2 Levels and Scope Violations

When programmers have the ability to annotate expressions with types in their programs, we must also keep track of which scope a given type variable was introduced in. To understand why, consider the Futhark program

```
def f x = let g 't (y: t) = [x, y] in g
```

Here, we define a function in a `let`-expression whose argument must have some type `t`. But since `g` returns an array containing its argument as well as `f`'s argument `x`, `x` and `y` must be of the same type, yet where `x` is specified in the program, the type parameter `t` hasn't been introduced yet. Conclusively, the type parameter `t` is bound at a deeper scope than the type variable representing `x`, and the program is therefore ill-typed.[4]

The way to identify such an error is, as mentioned above, to associate with each type variable and type parameter an integer representing the *level* (or *depth*) of the scope in which it was introduced. Whenever a type variable is constrained to have the same type as some type parameter appearing in the program, we must check if the type parameter has a higher level than the type variable and, if that is the case, report a scope violation. This can be handled in different ways in the unification process, either by "allowing" a type variable to be substituted with a type parameter no matter what and then checking afterwards if an erroneous substitution has been made, or just by checking continuously.

2.3.3 Liftedness

Finally, it's also worth mentioning the concept of *liftedness* constraints in Futhark: When a programmer uses a type parameter to specify the type of some expres-

[3] https://futhark.readthedocs.io/en/stable/language-reference.html#x-binop-y.

[4] Note that the program wouldn't be ill-typed if the explicit `t` parameter hadn't been used.

sion, they can also specify whether a type (parameter) is *unlifted* (no constraints), *size-lifted* (possibly contains existential sizes) or *fully lifted* (possibly contains functional types). In the part of the type checker that we're focusing on in this project, it only has the implication that whenever a constraint involves two type variables that are completely unconstrained in terms of what types they may be substituted with but where their liftedness constraints differ, both type variables should now have the least "strict" constraint among the two: Informally, the order of strictness is `unlifted` < `size-lifted` < `fully lifted`. Further, when we're done solving constraints, we must also return a list of all the type variables that occur in the constraint set but shouldn't be substituted with anything, along with the liftedness constraint they've been inferred to have.

3 Implementation

This section covers the optimizations we implemented in Futhark's type checker, which is written in the programming language Haskell. At first, we'll examine the `ST` monad and how it can be used to implement algorithms and/or data structures that benefit from having some kind of updatable state. Next, we'll go through how a union-find data structure with path compression and union-by-weight, an inherently imperative data structure, can be implemented in Haskell, a purely functional programming language. Finally, we'll cover how we've updated the code for solving type constraints to utilize the benefits of having an efficient union-find data structure.

3.1 The ST Monad

In purely functional languages like Haskell, values are *immutable*: Whenever a variable, array element, record field, or something similar has been initialized to some value, it cannot be changed afterwards [9, p. 122]. A common challenge in such languages is thus to implement algorithms or data structures in which mutability seems to play a crucial role. In an attempt to address and solve this problem (in Haskell), the `ST` monad was introduced and described by Launchbury and Peyton Jones [10].

3.1.1 States and References

The "`ST`" stands for "state transformer" which is another way of writing "a stateful computation"; essentially, it is a computation that takes one state and transforms it into another [10]. A value of type `ST s a` is then a computation which transforms a state "indexed" by some (abstract) type `s` – typically referred to as the "state thread" – and delivers a value of type `a` [10].

But what is meant, conceptually, by a "state", then? As Launchbury and Peyton Jones [10] outline, "part of every state is a finite mapping from *references* to values" (along with other components), and a reference can be thought of as the name identifying a variable, that is, an updatable location in the state that is capable of holding some value. Hence, we would also like some kind of way to create new references, read existing ones, and write new values into them; this is achieved in the actual implementation [11] with the following three operations:

```
newSTRef   :: a -> ST s (STRef s a)
readSTRef  :: STRef s a -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()
```

The `newSTRef` function is responsible for creating a new reference in the current state "thread", `readSTRef` is used to read the value store in a reference, and

`writeSTRef` is used to write a new value at the location the reference points to. Notice, again, that they are all parametrized by a state thread `s` which, in some sense, identifies a specific state thread.

3.1.2 Encapsulation

Two natural questions arise now: 1) how do we use state transformers as part of a larger program that doesn't manipulate state at all, and 2) how do we ensure that a reference from one thread is not used in another thread? A possible answer to both question lies in constructing a function, `runST`, that takes a state transformer and returns the final result computed by the state transformer. As an initial attempt, we could try giving this function the following type:

```
runST :: ∀s,a. ST s a -> a
```

The idea here would be to let `runST` take a state transformer as input, build an initially empty state, apply the state transformer to the empty state, and return the result while discarding the final state. However, with this type, we do not solve the problem implied in question 2. For instance:

```
let v = runST (newSTRef True) in runST (readSTRef v)
```

Allowing an expression like this would be bad because we are allocating a reference in one thread and then reading it in another thread, and reads in one thread are not sequenced with respect to writes in the other [10]; hence, this would introduce nondeterministic behavior. Instead, whenever we call `runST`, what we really want is to not make any assumptions about the initial state, that is, what has already been allocated in the initial state: `runST` "should work regardless of what initial state it is given" [10]. To express this on the type level, `runST` is specified to have the following type:[5]

```
runST :: ∀a. (∀s. ST s a) -> a
```

Letting `runST` have this type, the let-expression above is now ill-typed: Firstly, `readSTRef v` has type `ST s Bool` and this cannot be generalized to `∀s. ST s Bool` [10] so the `in`-part of the expression is ill-typed. Secondly, a reference won't even be able to escape the thread in which it was created in the first place: The call `runST (newSTRef True)` would be ill-typed as well since the return type of `newSTRef` includes a "specific" `s` whereas `runST` demands that the `s` in the argument must be arbitrary. As such, it is guaranteed that references created in a thread are safely encapsulated from the outside world. Moreover, `runST` answers

[5] This is an example of rank-2 polymorphism; we won't cover the details of what this implies but it makes it easier to describe how to make `runST` fulfill the requirements we need it to.

the first question proposed at the beginning of this section since we can use it to "escape" the state transformer and retrieve the result it computes – as long as this result doesn't contain any references (or, in general, is a result produced by using one of the three functions for manipulating references).

3.1.3 Implementation

To actually implement state transformers, Launchbury and Peyton Jones [10] propose an implementation framework where

- the *state* of each encapsulated state thread is represented by a collection of objects allocated on the heap,
- a *reference* is represented by the address of an object in heap-allocated storage (similar to the concept of pointers in a language like C),
- a *read operation* returns the current content(s) of the object whose reference is given, and
- a *write operation* overwrites the contents of the specified object.

In the actual implementation[6], the `ST` type has been defined (essentially) as follows:

```
newtype ST s a = ST (State# s -> (# State# s, a #))
```

As Launchbury and Peyton Jones [10] describe, one should think of a value of type `State# s`, for some type `s`, that is, a primitive state, as a "token" that represents the state of the heap: The implementation doesn't need to actually inspect a primitive state value, it's just passed to and returned from every primitive state transformer operation to maintain the idea of having a single "state thread".

It's also worth noting that by making `ST` a *monad*, we can use the bind function (>>=) to chain states together in the sense that the result and updated state of some stateful computation can be given as input to another stateful computation; without this functionality, it would be hard to imitate updatable states in a purely functional language like Haskell.

Conclusively, when implementing algorithms or data structures in which in-place updates are crucial, the `ST` monad is ideal since it can be used in a manner that doesn't result in impure code and since it makes it possible to have in-place updates directly in memory.

[6] https://hackage.haskell.org/package/ghc-internal-9.1201.0/docs/src/GHC.Internal.ST.html#ST

3.2 Union-Find in Haskell

The union-find data structure introduced in section 2.2.3 relies on some kind of mutability when performing certain operations, such as the `Find` operation where path compression will change the parent pointers of nodes in the tree, and the `Union` operation which must change the weight associated with a given root node when the union-by-weight heuristic is used.

To implement these operations and heuristics in Haskell, an obvious way would thus be to use the `ST` monad and how it can be used to have in-place (memory) updates with `STRef`.

3.2.1 Representing Nodes in the Disjoint-Set Forest

One way of representing nodes in a disjoint-set forest in a generic manner is outlined below:

```
newtype Node s a = Node (STRef s (NodeInfo s a))

data NodeInfo s a
  = Link (Node s a)
  | Repr ReprInfo

data ReprInfo a = ReprInfo { weight :: Int, info :: a }
```

With this representation, every node is, at its core, a reference to a value with type `NodeInfo`. A value of type `NodeInfo` is then either linked (that is, points) to another node, or it's a `Repr` containing a value of type `ReprInfo`. Values of type `ReprInfo` then have a `weight` field to make it possible to use the union-by-weight heuristic, and a field called `info` which can be used to contain any kind of information relevant to the concrete application of the data structure.

It's worth noting that this representation differs in some notable ways from how we presented nodes in section 2.2.3: With the representation outlined above, nodes do not have a uniform representation since not every node has an associated weight and parent pointer. Instead, there is just a single node in each equivalence class describing the entire equivalence class, including its current weight/size, but this is also all we need: The concept of having an equivalence class is based on every element in it being (or, at least, representing) *the same thing* so we only *need* one element to "carry the burden" of containing the relevant information about the class.

3.2.2 Implementing `find`

The `find` operation can then be implemented in a reminiscent but not completely equivalent way with regards to how we described it in section 2.2.3. Concretely, it looks like this:

```
find :: Node s a -> ST s (Node s a, ReprInfo)
find node@(Node node_info_ref) = do
  node_info <- readSTRef node_info_ref
  case node_info of
    -- Input node is representative.
    Repr repr_info -> pure (node, repr_info)

    -- Input node points to another node.
    Link parent -> do
      a@(repr, _) <- find parent
      when (repr /= parent) $
        -- Performing path compression.
        writeSTRef node_info_ref $ Link repr
      pure a
```

As expected, it takes a `Node` as input, but since a node is a wrapper for a reference, we need to first read the contents pointed to by this reference, using `readSTRef`. This content may then be a `Link` to a parent node, and if that's the case, we recursively call `find` with this parent to find the representative. Also, whenever this parent isn't also the representative, we perform path compression by overwriting the content pointed to by the input node's reference with a `Link` directly to the representative node.

Other operations might also want to easily extract the information from any of the fields in the representative so we also make sure to return the `ReprInfo` encapsulated in the root node along with the node itself.

3.2.3 Implementing `union`

As in Listing 3, the implementation of `union` starts by finding the representatives of the input nodes with the `find` operation. Then, to avoid spending unnecessary time reading and writing references, we check if the nodes are already in the same equivalence class. If that's not the case, the node representing the class with the least number of elements among the two is set to point to the other node. Concretely, it looks like this:

```
if w1 >= w2
  then do
```

```
    writeSTRef root_info_ref2 $ Link root1
    writeSTRef root_info_ref1 $ Repr $ ReprInfo (w1 + w2) info'
  else do
    writeSTRef root_info_ref1 $ Link root2
    writeSTRef root_info_ref2 $ Repr $ ReprInfo (w1 + w2) info'
```

`w1` and `w2` refer to the weight/size of the input nodes' respective trees, and `root_info_refN` is the reference to input node `N`'s root: The corresponding value of this reference can be overwritten with a value constructed with the `Link` constructor when it must point to a new root, or with a new `ReprInfo` when it must hold new information about the (joined) equivalence class. Also, `info'` is the information that should be associated with the resulting equivalence class; this could just come from either of the roots, or it could be new information provided as input to the `union` function.

3.3 Solving Type Constraints

Now, we move on to describe the module where the different constraints are actually solved, namely in `TySolve.hs`. This module is only responsible for solving *type* constraints, that is, *size* constraints[7] and *rank* constraints[8] are not handled in this module.

3.3.1 The `SolveM` Monad

As described in the previous section, we need the `ST` monad to be able to perform path compression and union-by-weight in an efficient way. Additionally, instead of explicitly passing around the mapping from type variable names to their nodes in the disjoint-set forest, we've used the `ReaderT` monad (transformer). Finally, we'd also like to be able to handle type errors in a proper way which the `ExceptT` monad (transformer) is well-suited for. Thus, we end up with a monadic stack on the form

```
newtype SolveM s a = SolveM { runSolveM :: ExceptT TypeError
↪  (ReaderT (SolverState s) (ST s)) a }
```

The `SolverState s` is simply a wrapper for the mapping from type variable names to their corresponding node described in section 3.2 and is parametrized by the state thread `s` because every node in the mapping is also parametrized by this state thread.

[7] https://futhark.readthedocs.io/en/stable/language-reference.html#size-types

[8] As described in Schenck, Hinnerskov, Henriksen, *et al.* [12].

3.3.2 Type Variable Solutions

When we're done unifying constraints, we have to be sure that the final solution is sound in the sense that, if we conclude that a type variable should be substituted with some specific type, it must also actually be possible for this type variable to be resolved to this type. To do this, during the unification process, we distinguish between three types of type variables: 1) solved type variables, 2) unsolved but possibly constrained type variables, and 3) type *parameters* which are a type variables that appear in an explicit type annotation in the source code from which the constraints were generated.

When a type variable is either solved or is a type parameter, we'll refer to it as being *rigid* to accentuate that it cannot be assigned another type (anymore). Contrastingly, we'll call a still unsolved type variable *flexible* although it might be constrained in terms of *how* it must be solved.

3.3.3 The `Solution` map

Ultimately, what we want to achieve (if the constraints can be solved) is a substitution that solves the constraint set. In the code, we represent this as

```
type Solution = M.Map TyVar (Either [PrimType] (TypeBase ()
↪  NoUniqueness))
```

Similar to the disjoint-set forest, this is a mapping from type variable names (`TyVar`) to the type they should be instantiated with. More precisely, the `Either` monad is used to distinguish between whether a type has numerous possible (primitive) types it can be instantiated with or whether it must be instantiated with a specific type. As we mentioned in section 2.3, the reason a type can be ambiguous is because of overloading; we'll come back to this issue and how it's solved when we discuss how constraints are actually solved, but an expression that uses an overloaded operator will generate type variables that can only be substituted with a specific range of types – that is, it will generate flexible but constrained type variables.

A takeaway from this is also that this isn't a substitution in the exact same sense that we discussed in section 2 and, hence, it isn't necessarily an MGU either: It is only (or, at least, should be) an MGU if every unique type variable occurring in the constraint set can be resolved to a non-ambiguous type.

3.3.4 Representing Type Variables

In order to start solving constraints, we must first store the type variables and parameter in the union-find data structure. Therefore, each type variable and parameter are placed in a `TyVarNode` as a `Repr` (described in section 3.2), meaning

that they will each represent their own equivalence class. However, a representative should also contain information about the class and we store this information in fields added to the `Repr` constructor. Concretely, in our implementation, the `Repr` constructor contains the following fields:

1. The *weight* of the equivalence class in order to utilize the *union-by-weight* heuristic described in section 2.2.2.

2. The *key*, which is the name (called `TyVar` in the previous subsection) of the type variable or parameter that is the representative of the equivalence class.

3. The *solution* of the equivalence class. This field contains the information about what kind of type variable the the representative is, that is, if the type variable is *solved*, *unsolved* or a type parameter as described in section 3.3.2.

Now, after putting each type variable or parameter in its own equivalence class along with the proper information about the representative, we have initialized the union-find data structure, and we can thus begin the process of solving the constraints.

3.3.5 Normalization of Types

When we process a constraint, the first thing we do is to *normalize* each type occurring in the constraint. Normalizing a type t can be divided into three cases:

1. If t is a solved type variable, we substitute it with the type we've assigned to it;

2. if t is an unsolved type variable (or parameter), we find the *key* (that is, the name) of the representative type variable of the equivalence class, and substitute it with this type variable;

3. otherwise, we just return t itself.

Sticking to the notation used in section 2, an example where this might be relevant could be the constraint set

$$\{\alpha \equiv \texttt{int}, \beta \equiv \texttt{int} \to \texttt{int}, \alpha \equiv \beta\}$$

If these constraints are processed from left to right, the last constraint will be converted to the constraint $\texttt{int} \equiv \texttt{int} \to \texttt{int}$. By doing this, we avoid an attempt to re-solve a type variable in a possibly illegitimate way. In this concrete

example, it means that we discover early in the process that α and β are not unifiable since the constraint $\texttt{int} \equiv \texttt{int} \to \texttt{int}$ is unsolvable, that is, there exists no substitution S that would make $S(\texttt{int})$ equal to $S(\texttt{int} \to \texttt{int})$.

In the first two cases, it's also important that normalization involves path compression, similar to how we described `findType` in Listing 4 should work, to get the favorable asymptotic running time described in section 2.2.4.

3.3.6 Solving the Constraints

If a given constraint doesn't involve any type variables, we must check if it is unifiable and if it emits any new constraints that must hold. For instance, since Futhark is a functional language, arrays must be *homogeneous*, that is, all of its elements must be of the same type, so when we encounter a constraint involving two array types, a new constraint involving the element types is emitted.

However, if a given constraint *does* involve type variables, we must consider several cases to solve it. The goal is to either bind a type variable to a concrete type (that is, a type that isn't a type variable), union the equivalence classes of two type variables if they're both flexible, or conclude that they're non-unifiable if they're both rigid and non-equal.

In the case where the constraint involves exactly one type variable α and if α is flexible, we must bind the type variable to the given type t. However, before performing this binding, we need to substitute any type variable occurring in the type with the type that this variable is bound to (if it's bound to anything). We have to do so because we have to perform the so-called occurs check mentioned in section 2.1.4, and if we don't fully substitute the type variables occurring in t, the occurs check might miss that we're about to create a cyclical type.

However, if the occurs check doesn't fail, we naively update the type of the α to t before checking if the binding is valid (a type error will be thrown if not). If the type variable is rigid, however, we must instead check if the type that the type variable is bound to is unifiable with the type specified by the constraint, as we did when unifying two types without type variables.

The same approach is taken when the given constraint contains two type variables, one of which is rigid. Here, we bind the flexible type variable to either the type of the rigid type variable (if it's solved) or the type variable itself (if it's a parameter). If the given type variables are both rigid but not equal, the constraint cannot be solved. Finally, if the two type variables are equal, the constraint is trivially solvable by doing nothing.

If the constraint contains two flexible type variables (after normalization), we must union them. In this case, we naively union the equivalence classes of the type variables before checking if it is valid to do so. For instance, even though they're both flexible, one or both of them could have constraints on how they

must be solved, potentially making them non-unifiable.

When all constraints have been solved as described above, we iterate through the original list of type variables in the `solveTyVar` function. During each iteration, we check for ambiguities which arise when a type variable initialized to be flexible but not *free* remains *unsolved*. A type variable is free when it can be bound to any type. Contrastingly, a non-free flexible type variable is classified as unsolved as described in section 3.3.2 but constrained in terms of how it must be solved. For instance, a type variable might be constrained to only be able to be unified with a record that includes a given set of fields, meaning that it is flexible but not free since it can only be unified with a record type. But if it is never unified with a record type, its type is (too) ambiguous. More precisely, we throw a type error whenever a flexible type variable constrained to be either a sum type or a record hasn't been assigned a solution in the end. If a type variable which must be unified with a primitive type hasn't been assigned a type in the end, this isn't considered an error since later parts of the type checking process can choose a type to default it to.

When we're done solving constraints, we also do a *scope* and a *liftedness* check for each free, flexible type variable. As we described in section 2.3, the scope check ensures that no type variables have been unified with a type parameter bound at a deeper scope than where the type variable is bound, whereas the liftedness check ensures that the liftedness constraint of free type variables is set to the least restrictive liftedness constraints of any free type variable they have been unified with.

Finally, we extract the type of every type variable by looking in the union-find data structure. If the `solution` found in the representative is a value of type `Solved t`, we make a mapping from the name of the given type variable to the type `t` that it's been assigned. If the representative instead represents either a flexible type variable or a type parameter, we must check if the name of the given type variable and the key of the root is the same: If they're the same, that is, it shouldn't be substituted with anything, this variable shouldn't be included in the final `Solution` but we still make sure to note that it's unconstrained since it might be relevant to the other parts of the type checker. If they're not the same, the type variable must be substituted with the type variable with the same name as the key found in the root.

4 Evaluation

This section aims to evaluate our new implementation of the constraint solver described in the previous section. We evaluate its correctness by making and running unit tests but also using predefined tests. The performance is evaluated by benchmarking on two suites of benchmarks that we have created and converted from existing Futhark programs. Lastly, we attempt to identify bottlenecks in our implementation using profiling.

During the initial development of our implementation, we relied primarily on testing to ensure the correctness as it is difficult and nonsensical to benchmark an implementation that does not work as intended. Therefore, we created some very simple tests that served as milestones for our implementation, and we gradually added more whenever a milestone was reached until we were fairly certain that our implementation was mostly correct.

Now, having a mostly correct implementation, we began profiling the implementation to identify the parts of our code that used the most amount of time to execute. For instance, we found out early in the process that we had implemented the occurs check in a very inefficient way which we then updated to make it quicker.

Next, when we had removed the obvious bottlenecks, we wanted to evaluate the execution time of our implementation against the old implementation to see how we compared. For this purpose, we used benchmarking where, in the beginning, we simply used `hyperfine`[9] to run the `futhark check` command on a Futhark program. However, later on we developed the aforementioned suite of benchmarks.

This sums up the workflow that we used to develop the implementation, that is, iterating through the stages of developing, testing, profiling, and benchmarking. The remainder of this section will describe each of these stages and the final results thereof in depth. All graphs presented in this section can be found in an enlarged version in appendix A.2.

4.1 Testing

To evaluate the correctness of our implementation, we used unit tests in combination with a suite of tests (that we didn't make ourselves) of the entire type checker. The unit tests can be found in the `TySolveTests` module, and the suite of tests can be found in the `tests` directory at the root of our repository.

The unit tests cover simple edge cases, such as discovering infinite types, scope violations, differently sized tuples, records, and more. These were devel-

[9] https://github.com/sharkdp/hyperfine

oped to test some of the very basic types and scenarios arising from the type system. The unit tests can be run using the following command: `cabal run unit -- -p Unsized`. Our implementation passes all the unit tests we've made. However, it's worth mentioning that we were unable to test sum types in our unit tests since the parsing of constraints does not support sum types. But as we shall see, some tests containing sum types were still resolved correctly.

The second approach was using the tests provided in the aforementioned `tests` directory. We ran these tests using the command: `futhark test -t tests/`. This suite tests a wide variety of programs using more than 2000 Futhark programs as test cases. Our implementation was hereby also tested on sum types as some of these programs use them. Working from an experimental branch, that is, the `automap` branch, was sometimes challenging as we on rare occasions ran into bugs that were not only present in our implementation. These bugs were fixed, however, the test suite contains 20 programs that fail even with the original implementation from the `automap` branch. We did not attempt to solve these since they were related to other parts of the type checker, and, as such, our implementation also fails the same 20 test cases.

However, even with both these approaches, we can not be completely sure that our implementation is correct since there might be cases that are not covered by these tests. But also because the `TypeChecker.Terms.Unsized` module calls the `solve` function of our implementation and uses the results. This means that even if we were to return an incorrect solution, it is not certain that the incorrect part of the solution will be used by the type checker.

Although, we still find that the described testing is sufficient to evaluate the correctness of our implementation, and determine that it is sufficiently correct compared to the old implementation as it passes the same tests as the old implementation.

4.2 Benchmarking

The purpose of benchmarking our implementation was to evaluate whether it has improved the performance of the type constraint solver of the type checker. As described in section 2.2.4, the union-find data structure can provide a practically linear (in the number of type variables) running time implemented with the heuristics described in section 2.2.2. Therefore, we were interested in benchmarking the *execution time* of our implementation to discover if this new data structure would improve the execution time.

Many approaches can be taken to benchmark this metric but we chose two different approaches. The first approach was to convert a set of real-world Futhark programs to a set of benchmarks to be evaluated. We chose this approach since it was a way to see the effect of our implementation on the runtime on actual

programs written in Futhark.

The second approach we chose was to create a list of benchmarks that simulated a scenario where we hypothesized that the path compression heuristic should be particularly advantageous to the execution time of our implementation. This is also a scenario where we believed that the previous implementation should have a significantly worse performance. This approach was chosen in addition to the other in order to evaluate whether our implementation would perform better than the old in this scenario as that is what we expected.

For both of these approaches, we wanted to evaluate the impact of the heuristics on the execution time of our implementation. Consequently, we tested this by running the benchmarks using our implementation, firstly with both heuristics enabled, secondly with path compression enabled and union-by-weight disabled, and lastly with both heuristics disabled.

4.2.1 Running the Benchmarks

To run all the benchmarks, we used a laptop with an Intel Core i7-13700H and 16GB of RAM running Fedora Workstation 42. The implementation of the benchmarks in Haskell can be found in the `TySolveBenchmarks` module and was done using the package `criterion`[10]. This `criterion` package is a reputable tool that ensures that the measurements of the execution time are more accurate by utilizing regression analysis and cross-validation to distinguish the actual data from noise [13].

In order to benchmark our implementation and the old, we put each implementation in their own module. Our implementation was placed in the `TySolve` module[11], and the old implementation was placed in the `TySolveOld` module[12]. It is the `solve` function that we are benchmarking since that is the exported function called by the remaining parts of the Futhark compiler when constraints are to be solved. To distinguish between the old and the new `solve` function, we created two new `solve` functions in the `TySolveBenchmarks` module, `solveOld` and `solveNew`. These functions are wrappers as they simply pass on given arguments to their respective `solve` function.
Now, a benchmark consists of the following three elements:

1. A list of constraints.
2. A map with mappings of type parameters to their respective information.
3. A map with mappings of type variables to their respective information.

[10] https://hackage.haskell.org/package/criterion
[11] https://github.com/jacobgummer/futhark, commit `c948040`.
[12] https://github.com/jacobgummer/futhark, commit `c9a9261`.

By passing these three elements as arguments, the implementations can be evaluated by calling the one (or both) of the aforementioned `solve` functions. Before such a benchmark was run, we had to place it in the `TySolveBenchmarks` module. Next, to run the benchmark and save the generated data to a `CSV` file alongside a report formatted in `HTML`, we ran the following command:

```
cabal bench
↪  --benchmark-options="--csv results.csv --output report.html"
```

4.2.2 Benchmarking of Futhark Programs

As mentioned in the beginning of section 4.2, this approach to benchmarking seek to evaluate the effect on the "real-life execution time" of our implementation. To achieve this, we need a reasonably sized suite of Futhark programs to convert to benchmarks as these programs will serve as a representation of average real-life programs.

While the synthetic benchmarks described in further detail in the next section seek to evaluate a synthetic scenario, these benchmarks seek to provide a more realistic measure of the performance impact of our implementation.

Converting Futhark Programs to Benchmarks

To obtain the aforementioned suite of Futhark programs, we had to look no further than at the `futhark-benchmarks` repository[13] included as a submodule in the `futhark` repository.

Inside the `futhark-benchmarks` repository is 124[14] Futhark programs in the form of `.fut` files. To extract the constraints, type variables, and type parameters from a Futhark program, the following command can be run:

```
FUTHARK_LOG_TYSOLVE=0 futhark check <Futhark program>.fut
```

The environment variable `FUTHARK_LOG_TYSOLVE` enables logging of the `TySolve` module during the (type)`check` of the given Futhark program. During this `check`, the `solve` function is called many times since its output is used for many purposes, including to check the type of each function in the given program individually. Thus, the logging consists of the following ***block*** of output repeated for each call to the `TySolve.solve` function:

[13] `https://github.com/diku-dk/futhark-benchmarks`, commit `7d97653`.

[14] Excluding the `.fut` files found in (sub)directories named `lib` to avoid duplicate benchmarks, as these files are primarily imports from other repositories and are often repeated.

```
# TySolve.solve
## constraints
"t\8322" ~ "a"
## typarams
["(\"a\",(0,Unlifted,NoLoc))"]
## tyvars
[("t\8322",(6,TyVarFree NoLoc Lifted))]
## solution
([], [("t\8322",Right "a")])
```

Figure 3: An example of a *block* from the output generated during a type check, if logging is enabled.

Each block is a problem that can be converted to a benchmark since it contains the three elements mentioned in section 4.2.1. The `solution` field is thus ignored since it is not needed to create a benchmark.

Any Futhark program generates *a lot*[15] of blocks, since a block is logged each time the `solve` function is called during a type check. However, 832 of these calls are actually done to type check the built-in `Prelude`[16]. Since the `Prelude` is implicitly imported in all Futhark programs, and thus introduce a constant amount of work, we decided to exclude the first 832 blocks generated by each file.

Now, excluding the first 832 blocks of each file, more than 6000 blocks were generated by the 124 files. Since each block is converted to a benchmark, this would result in an equal number of benchmarks. Running this many benchmarks would take an infeasible amount of time and thus, we decided to exclude some of these blocks. To ensure that the included blocks yielded the most insightful results, we wanted to include the blocks with the highest complexity in the sense that they result in the highest execution time, and because blocks with higher complexity hopefully increase the possibility of revealing significant differences in performance between the old and new implementation.

However, predicting the complexity of a given block is difficult to do systematically without doing a deeper analysis of the constraints, type parameters, and variables in the given block. We decided to use the number of constraints as a proxy for the complexity of a given block, that is, the more constraints the higher the complexity. This choice is founded on the fact that each constraint represents a relationship between two types that must be resolved. A higher number of con-

[15] As an example it is called more than 1000 times for the `rodinia/myocyte/myocyte.fut` file, generating a block of output for each call.

[16] https://futhark-lang.org/docs/prelude/

straints must therefore increase the number of potentially complex relationships to resolve, thereby also increasing the execution time of the constraint solver.

As we shall see, this proxy is by no means perfect as there can still exist blocks with a low number of constraints and high complexity and vice versa. But excluding such edge cases, this proxy still guides us towards the problems that are more likely to challenge the efficiency of the implementations and thus reveal insightful results.

Having chosen the number of constraints as the proxy for the complexity of a block, the question as to how many constraints a given block must contain in order to be sufficiently complex was still open. However, we decided to base this threshold on the (in)feasibility of running the remaining number of blocks as benchmarks. We was found that a threshold of 30 constraints resulted in 598[17] remaining blocks from 105 different files which is a feasible number of benchmarks to run[18].

These blocks then had to be converted into benchmarks. To achieve this, a new `Generated.AllFutBenchmarks` module was created. Each file also got its own submodule named after the path of the file, for example the `rodinia/myo-cyte/myocyte.fut` got a submodule called `Generated.AllFutBenchmarks-.Rodinia.Myocyte.Myocyte`. All the blocks generated by the given file were placed inside this submodule.

However, before placing a block in its submodule, it must first be converted to a benchmark as it is not as simple as taking the output from a block and directly placing it in the submodule. As described in 4.2.1, a benchmark requires a list of constraints, a map with type parameters, and a map with type variables. This information is available in the block, as seen in figure 3 but it must first be formatted correctly. As an example, backslashes in type variables cannot be used in benchmarks[19], instead they must be replaced with underscores.

When formatting is completed, the information from the block is finally ready to be inserted into its corresponding submodule. All the submodules use the same template. An example of the block in figure 3 using the template is shown below.

```
module Generated.AllFutBenchmarks.Example (benchmarkDataList)
 ↪  where
-- Imports ...
(~) :: TypeBase () NoUniqueness -> TypeBase () NoUniqueness ->
 ↪  CtTy ()
```

[17]The actual number was 631, but we had to exclude 33 blocks due to errors. Two entire files were also excluded due to errors. We expect that these errors are caused by the lack of support in the `SyntaxTests` module for parsing strings to sum types.

[18]It takes approximately 40 minutes to run the 598 benchmarks.

[19]The conversion and parsing of a string to the `CtType` type is done in the `SyntaxTests` module which requires the strings to be formatted in a certain way.

```
t1 ~ t2 = CtEq (Reason mempty) t1 t2

type BenchmarkCaseData = ([CtTy ()], TyParams, TyVars ())

benchmarkDataList :: [BenchmarkCaseData]
benchmarkDataList =
  [   (
    [
    "t_8322" ~ "a_1",
    ],
    M.fromList [("a_1",(0,Unlifted,NoLoc))],
    M.fromList [("t_8322",(0,TyVarFree NoLoc Unlifted))]
  ), -- Remaining blocks, if any.
  ]
```

Finally, when all blocks have been converted to benchmarks, they are imported into one complete list in the `Generated.AllFutBenchmarks` module. This list is then imported to the `TySolveBenchmarks` module which iterates through each benchmark in the list when the benchmarks are run as described in section 4.2.1.

Results

As briefly mentioned in the beginning of this section, our implementation was evaluated against both the old implementation but also against our own implementation with different combinations of heuristics enabled. This suite of benchmarking was thus run 4 times, yielding 4 different datasets to be analyzed. The implementation used to generate the datasets are summarized below along with their shorthand name used in the graphs in this section. More detailed information about the datasets, such as the average mean execution time and more, will be referenced in the following and can be found in appendix A.1.

Implementation used to generate dataset	Shorthand name
Old implementation	solveOld
New implementation	solveNew
New implementation without union-by-weight	solveNew w/o ubw
New implementation with no heuristics	solveNew w/o heuristics

The first graph shown in figure 4 does not seem to show a correlation between the number of constraints and the mean execution time (MET) to solve these constraints. It can also be seen that the vast majority of the benchmarks have between 30 and 400 constraints. The relative performance graph shows that

the `solveNew` implementation seems to have a higher MET than the `solveOld` implementation on average. This is backed up by the fact that the mean of the relative performance is −0.22 across all benchmarks, and also that the average MET of the `solveOld` implementation was 16.4% lower than that of `solveNew`.

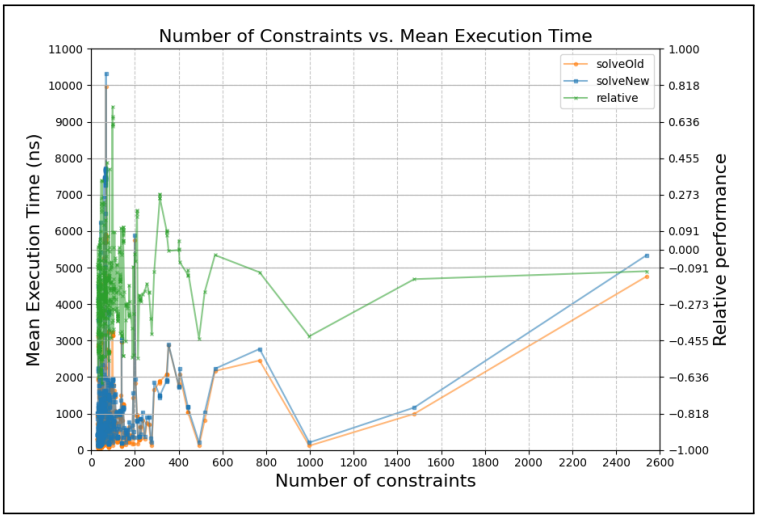


Figure 4: A graph of the number of constraints against the mean execution time (in nanoseconds) to solve the given number of constraints for both `solveOld` and `solveNew`. The green graph shows the number of constraints against the relative performance of the two solvers.

As the previous graph did not reveal a clear correlation between the number of constraints and the mean execution time, we decided to focus more on the relative performance between implementations. Figure 5 shows that `solveOld` had a lower MET in 535 of the the total 598 benchmarks, which further underlines the fact that `solveNew` is slower than `solveOld` in this suite of benchmarks.

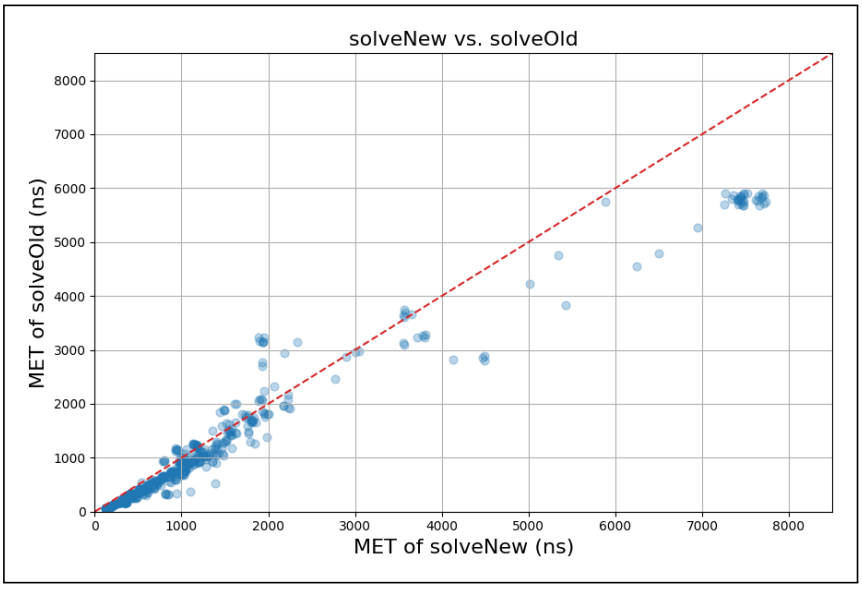
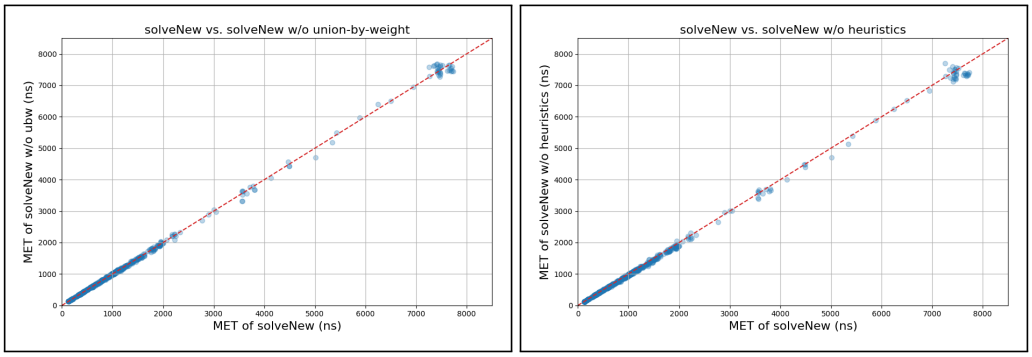


Figure 5: A graph showing the relative performance between `solveOld` and `solveNew`. The diagonal red line is the line of equality where $y = x$. If a data point is above the red line, `solveNew` had a lower (that is, better) MET than `solveOld` in the given benchmark, and vice versa. There are 63 data points above the red line and 535 below.

In figure 6, we wanted to examine the effect of the heuristics of the union-find data structure on the MET of `solveNew`. Therefore we constructed two graphs: figure 6a shows the relative performance between `solveNew` and `solveNew` without *union-by-weight*, and figure 6b shows the relative performance between `solveNew` and `solveNew` without heuristics.

Figure 6a shows that `solveNew` with or without union-by-weight perform almost equally as the data points are all very close to the diagonal line. This can also be inferred by looking at their average MET where `solveNew w/o ubw` has an average MET of 1333.25 ns that is less than 0.5% lower than `solveNew`.

However, when disabling path compression also, it seems that the performance improves a little since the number of benchmarks where `solveNew w/o heuristics` are faster than `solveNew` increases from 331 when still using union-by-weight to 535 when using no heuristics. This is furthermore observed in the average MET that decreases by 2.18% from 1333.25 ns to 1304.12 ns when compared to `solveNew`.



(a) There are 267 data points above the red line, and 331 below.

(b) There are 63 data points above the red line, and 535 below.

Figure 6: The diagonal red line is the line of equality, where $y = x$. If a data point is below the red line, the dataset represented on the y-axis had a lower MET than the dataset represented on the x-axis in the given benchmark.

4.2.3 Benchmarking of Synthetic Scenario

In this approach to benchmarking, we wanted to evaluate whether our new implementation would have a lower execution time in a constructed synthetic scenario where we believe that it should benefit from the path compression of the union-find data structure. This approach is interesting as it will reveal whether our implementation is efficient enough to outperform the old implementation in a scenario that is well-suited for the union-find data structure using path compression.

The scenario consists of the following the constraint set.

$$\{\alpha_i \equiv \alpha_{i+1} \mid 0 \leq i \leq n-1\} \cup \{\alpha_n \equiv \texttt{int}\}$$

Assuming that each constraint solely involving type variables is solved by letting the node representing the left-hand type variable point to the right-hand type variable – which is practically what happens in the old implementation – we might end up with a chain of length n of type variables before we reach the last constraint $\alpha_j \equiv \texttt{int}$. The last constraint then essentially constrains every type variable in the chain to be substituted with `int`. This means that, when we must eventually determine what each type variable should be substituted with, we must take $n - i$ "steps" through the chain to resolve the type of $\alpha_i, 0 \leq i \leq n$, and since $\Sigma_{i=0}^{n} i = \Theta(n^2)$, resolving the type of every type variable in the chain would take time $\Theta(n^2)$. Contrastingly, with path compression and union-by-weight, resolving the type of a type variable will have an amortized $O(A(n))$ cost (with n again being the number of times a new node is created in the union-find

structure) which is less than or equal to 4 for any conceivable number of n, as we argued in section 2.2.4, and, hence, practically constant.

As previously, we also wanted to evaluate whether our new implementation would perform worse without using any of the heuristics described in section 2.2.2. This should be particularly interesting as we hypothesize that the path compression heuristic should improve the execution time in this synthetic scenario.

Constructing the Scenario as a Benchmark

To construct the scenario, we had to find a way to generate a list of constraints of length n as described above. For that purpose we created the function `generateConstraints` and placed it in the `TySolveBenchmarks` module. This function starts by generating a list of n free type variables. Next, the list of constraints needed to create the 'chain' of type variables is created. Lastly, one single constraint, $a_n \equiv$ `int`, is concatenated to this list. As described above, we hypothesized that this scenario should scale much better with the new implementation.

We generated 50 benchmarks, with the number of variables (n) starting at 20 and increasing by steps of 20 up to 1000. As we shall see, this range of benchmarks is sufficient to reveal the differences in the performances of the implementations. As in section 4.2.2, the metrics and shorthand names of datasets generated by this suite of benchmarks can be found in appendix A.1.

Results

Figure 7 presents the MET for the `solveNew`, `solveOld`, `solveNew W/o ubw` and `solveNew w/o heuristics` implementations across the aforementioned range of type variables (n). As anticipated, the `solveNew` implementation shows the lowest execution time, indicating that path compression reduces the MET. This is also futher confirmed by the `solveNew w/o ubw` implementation, as that seems to be almost equal with `solveNew`. The MET of these appears to grow slowly with n, indicating that the time complexity is linear.

The `solveOld` implementation shows a noticeably higher MET than `solveNew`. Its execution time increases at a much faster rate as n grows. Looking at the graph it seems to have the hypothesized $\Theta(n^2)$ running time.

As we expected, `solveNew w/o heuristics` performs the worst when comparing the four implementations. Its MET is higher than that of `solveOld` throughout all the benchmarks. It seems that they scale exactly the same, however, `solveNew w/o heuristics` performs worse by a constant factor. This might indicate that `solveNew w/o heuristics` has to do the same amount of traversals as `solveOld` but it uses a constant amount of time more on each traversal than `solveOld`.

The graph suggests that using the heuristics described in section 2.3 provides a great optimization that improves the time complexity of our implementation in this synthetic scenario.

Figure 7: A graph of the number of type variables in the 'chain' against the MET of the `solveNew`, `solveOld`, `solveNew w/o ubw` and `solveNew w/o heuristics` implementations.

4.3 Profiling

To try to gain information about which parts of our new implementation of the `TySolve` module acted as performance bottlenecks, we have profiled some benchmarks, since this allows us to profile the `TySolve` module in isolation. This is preferable, as the runtime of `TySolve` is easily overshadowed by the runtime of other modules, making it harder to identify what parts of `TySolve` are inefficient. We only focused on profiling the *execution time* of our implementation and not the *memory usage* as it is the execution time that we aimed to improve.

Profiling in Haskell with the GHC compiler[20] is done by compiling with profiling enabled and enabling the `+RTS -p` flags. This generates a profiling report in the form a `.prof` file with a *Cost Centre Module*. A *cost* is the CPU time (and memory) used by the Haskell code to evaluate a given expression. GHC will create a call tree of costs which allows one to identify the *inherited* percentage of the total costs as shown below[21].

[20] https://downloads.haskell.org/ghc/latest/docs/users_guide/profiling.html

[21] The `MODULE` and `SRC` columns have been omitted for brevity.

```
                                          individual      inherited
COST CENTRE      no.    entries  %time %alloc   %time %alloc
solve            1628     31401    0.8    1.0    98.6   98.3
solveCt          1648   2041065    0.5    0.5     6.1   10.8
```

The sample shows that the `solve` function calls the `solveCt` function. The cost of the `solve` function *individually* is 0.8% of the total time, but all its children's costs, that is, the *inherited* cost, takes up 98.6% of the total time which is expected since the `solve` function is the exported function that runs the constraint solver.

However, it can be difficult to analyze such a `.prof` file by hand as it can contain thousands of lines. Therefore we used `profiteur`[22] to vizualize the `.prof` file.

We chose to profile the two slowest benchmarks from the suite of benchmarks converted from Futhark programs as described in section 4.2.2, since this would allows us to examine what parts of our implementation use the most time to evaluate. These were block 1 and 3 from the `accelerate/hashcat/hashcat.fut` and `accelerate/julia/julia.fut` files, respectively.

Figure 8 visualises the profiling of the benchmark from block 1 of the `hashcat.fut` program. It shows that the `solve` function inherits a substantial amount of costs from calls to `solveEq.sub` (20.4%time and 19.8%time[23]), `unionTyVars` (19.4%) and the monadic bind operation >>= (19%time), totaling 78.6%time from these operations alone.

This informs us that an optimization or less use of these operations, would improve the execution time of this benchmark. The visualization also allows us to see what calls inside these functions are costly by examining the children costs. As an example the `unionTyVars` inherits 7.4%time of the total 19%time from the `unionTyVars.pts` function.

[22] https://github.com/jaspervdj/profiteur

[23] The reason why `solveEq.sub` appears twice in the visualization might be because the GHC compiler chose to inline calls to this function.

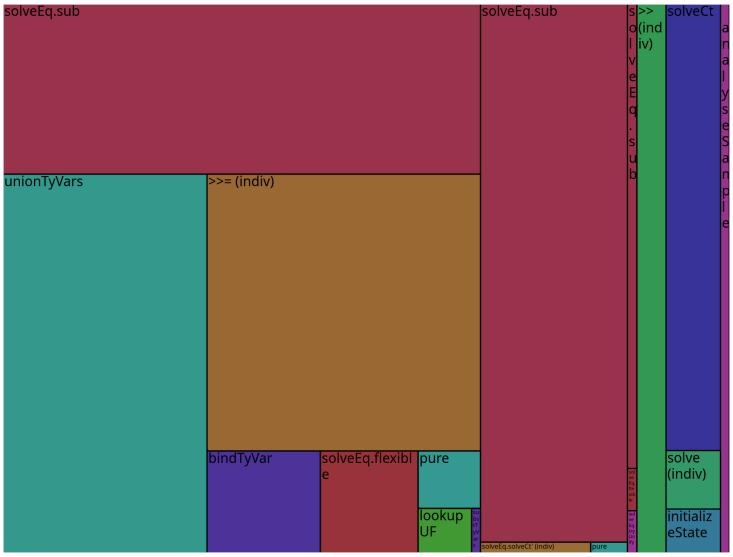


Figure 8: Visualisation of the profiling of the benchmark `accelerate/hashcat/hashcat.fut (Block 1/5)` containing 68 constraints.

The second visualization of the profiling of block 3 of the `accelerate/julia/julia.fut` file is shown in figure 9. It shows that a significant amount of cost comes from `solveEq.sub` (14%time and 11%time), `bindTyVar` (9.5%time), `getSolution` (9%time), and the monadic bind operation >>= (19.4%time). This shows that an optimization of the `solveEq.sub` function would improve the runtime of both profiled benchmarks. Interestingly, no calls are made to `unionTyVars` function which used 19%time in the previous profiling. However, this might be expected if the list of constraints in the block contains no constraints that results in the union of two type variables.

Figure 9: Visualisation of the profiling of the benchmark `accelerate/julia/julia.fut (Block 3/8)` containing 65 constraints.

4.4 Discussion and Possible Improvements

In terms of improving our implementation, we would definitely add more unit tests to ensure that it remains correct. We already have a reasonable amount of unit tests but there could be more, and, moreover, there could be more tests that don't necessarily involve type variables or type parameters to ensure that these types of constraint are also handled in an correct manner.

Our project was also focused on improving the running time of the constraint solver, so, naturally, we weren't focusing that much on making sure that the error messages provided to the user when a type error occurs are actually helpful. Hence, although it's a completely different scope, ensuring that error messages are correct and helpful would also be worth looking into in future work with the constraint solver.

The data gained from the suite of benchmarks converted from Futhark programs provided two important insights. The first being that our new implementation is measurably slower than the previous. We believe that the overhead introduced by the many reads and writes to `STRefs` in our implementation could be responsible for part of heightened execution time. This is backed up by the profiling that we have done, since its shows that functions such as `solveEq.sub` use a significant amount of time to evaluate (up to 40%). This particular function is called on every type in the constraints, and if the type is a type variable with a corresponding mapping to its node, the solution of the given node is looked up causing a read (and potentially a write, due to path compression) from an `STRef`.

The second important insight from the benchmarking of converted Futhark programs is that using *path compression* and *union-by-weight* in the implementation of our union-find data structure does not have the expected impact on the performance of our implementation; if anything, the impact is slightly negative resulting in a higher execution time compared to the implementation using these heuristics. We believe that some of this negative impact stems from the additional reads and writes of the `STRefs` that path compression, in particular, introduces, although more work should be done to identify more precisely what causes our implementation to be slower in general.

In contrast, the data from the benchmarking of the synthetic scenario suggest that the heuristics do indeed provide a substantial performance improvement in this specific scenario since it scales in a way more preferable manner than the old implementation.

The question of whether to use the heuristics or not naturally arises now. We argue that the performance degradation of using heuristics is so small that it is worth implementing them since the performance improvement in the synthetic scenario shows that the heuristics work. This performance improvement *alone*, should not be the reason for using heuristics but we argue that there is no reason to believe that removing the heuristics would provide a better asymptotic running time; the analyses we made in section 2.2.2 should justify this.

On the contrary, we believe that the improved performance seen by disabling the heuristics is a symptom of a structurally inefficient implementation of them. In addition, by using heuristics we can guarantee the theoretical $O(A(n))$ amortized time complexity for union-find operations.

As a final note, apart from improving the current logic used in our implementation, there are other things worth looking into to optimize the constraint solver even further.

First of all, a more efficient way of finding the corresponding node of a type variable in the union-find data structure could help in a better asymptotic running time, at the very least. A possible solution would be to replace the `Map` structure, which has $O(\lg n)$ lookup time, with a hash table, which in the best case has $O(1)$ (amortized) lookup time [14]. Existing Haskell implementations of hash tables, like *Data.Hashtable.Class* [14], also use the `ST` monad to achieve this efficient time used for lookups (among other operations, like insertion and deletion), so it wouldn't be difficult to introduce this to our implementation.

Second of all, the occurs check is potentially quite slow, at least when dealing with composite types that might contain type variables that should be substituted with types that then should be substituted with other types and so on. To make this check more efficient, Kiselyov [15] propose postponing this check by utilizing the levels of type variables to make unification of two free type variables take constant time. However, to actually implement this, one would need to as-

sign levels to composite types and not just type variables, meaning that it would require a substantial modification of the current, underlying type system used in Futhark. The benefits would most likely be significant but it would probably also require a project similar in size, or even bigger, compared to ours.

5 Conclusions and Future Work

We have described the theory underlying the process of solving constraints, how a union-find data structure and heuristics like path compression and union-by-weight provide a good asymptotic running time of unification, and how we updated Futhark's current constraint solver to utilize this data structure in hopes of making it more efficient.

Theoretically, we explored how to unify, that is, solve type constraints from a small type system we introduced. One approach to the problem of unification was based on continually applying and composing substitutions, but since this approach is inherently computationally expensive, we looked at how a union-find data structure using the aforementioned heuristics could be used to make unification have a good asymptotic running time. However, we noted that the operations provided by the union-find data structure are inherently difficult to implement in purely functional languages like Haskell. To address this, we took a look at the `ST` monad which can be used to have in-place updates in memory and, hence, make it possible to make an efficient implementation of the union-find data structure. With this, we thus explored how to actually implement the data structure in Haskell as well as how it was used in our updated implementation of Futhark's constraint solver.

To evaluate the correctness of our implementation, we constructed a small suite of unit tests that we had to (and did) pass. In addition to these tests, we utilized an already existing suite of Futhark programs as test cases. Our implementation passed the same amount of tests as the previous implementation, indicating that our new implementation is as correct as the existing implementation.

Our evaluation of the performance of our proposed implementation showed that it, in general, performs worse than the current implementation, demonstrating that there is room for improvements. We argued that a better implementation of the heuristics of the union-find data structure in combination with a postponed occurs check, and a more efficient replacement for the current `Map` data structure, such as a hash table, should significantly improve the performance of our implementation.

In conclusion, we have contributed with a large suite of benchmarks, a thorough theoretical walkthrough of how to do efficient constraint-solving using a purely functional programming language such as Haskell, but, perhaps most importantly, we have contributed with the foundation of a more efficient implementation of a constraint solver than the current.

References

[1] L. Damas and R. Milner, "Principal type-schemes for functional programs," in *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '82, New York, NY, USA: Association for Computing Machinery, Jan. 1982, pp. 207–212, isbn: 978-0-89791-065-1. doi: 10.1145/582153.582176. [Online]. Available: https://dl.acm.org/doi/10.1145/582153.582176 (visited on 05/13/2025).

[2] F. Baader, W. Snyder, P. Narendran, M. Schmidt-Schauss, and K. Schulz, "Unification Theory," in *Handbook of Automated Reasoning*, Elsevier, 2001, pp. 445–533, isbn: 978-0-444-50813-3. doi: 10.1016/B978-044450813-3/50010-2. [Online]. Available: https://www.cs.bu.edu/fac/snyder/publications/UnifChapter.pdf (visited on 06/04/2025).

[3] R. Schenck, "Sum types in Futhark," M.S. thesis, University of Copenhagen, Copenhagen, Denmark, Dec. 2019. [Online]. Available: https://futhark-lang.org/student-projects/robert-msc-thesis.pdf (visited on 06/05/2025).

[4] J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," *J. ACM*, vol. 12, no. 1, pp. 23–41, Jan. 1965, issn: 0004-5411. doi: 10.1145/321250.321253. [Online]. Available: https://dl.acm.org/doi/10.1145/321250.321253 (visited on 06/04/2025).

[5] J. Hoffmann, "Lectures 5–7: Type Inference," [Online]. Available: https://www.cs.cmu.edu/~janh/courses/ra19/assets/pdf/lect03.pdf.

[6] I. Grant, "The Hindley-Milner Type Inference Algorithm," Jan. 2011. [Online]. Available: https://steshaw.org/hm/hindley-milner.pdf (visited on 06/04/2025).

[7] S. Diehl, *Write You a Haskell*. [Online]. Available: https://smunix.github.io/dev.stephendiehl.com/fun/006_hindley_milner.html (visited on 06/04/2025).

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th. Cambridge, Massachusetts London: The MIT Press, 2022, isbn: 978-0-262-04630-5.

[9] T. Ægidius Mogensen, *Programming Language Design and Implementation*, 1st ed. 2022. Springer Cham, Nov. 2022, isbn: 978-3-031-11806-7. doi: 10.1007/978-3-031-11806-7.

[10] J. Launchbury and S. L. Peyton Jones, "Lazy functional state threads," in *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, ser. PLDI '94, New York, NY, USA: Association for Computing Machinery, Jun. 1994, pp. 24–35, isbn: 978-0-89791-662-2. doi: 10.1145/178243.178246. [Online]. Available: https://dl.acm.org/doi/10.1145/178243.178246 (visited on 06/07/2025).

[11] *Data.STRef*. [Online]. Available: https://hackage.haskell.org/package/base-4.21.0.0/docs/Data-STRef.html (visited on 06/08/2025).

[12] R. Schenck, N. H. Hinnerskov, T. Henriksen, M. Madsen, and M. Elsman, "AUTOMAP: Inferring Rank-Polymorphic Function Applications with Integer Linear Programming," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 1787–1813, Oct. 2024, issn: 2475-1421. doi: 10.1145/3689774. [Online]. Available: https://dl.acm.org/doi/10.1145/3689774 (visited on 05/14/2025).

[13] M. A. T. Handley and G. Hutton, "AutoBench: Comparing the time performance of Haskell programs," en, in *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, St. Louis MO USA: ACM, Sep. 2018, pp. 26–37, isbn: 978-1-4503-5835-4. doi: 10.1145/3242744.3242749. [Online]. Available: https://dl.acm.org/doi/10.1145/3242744.3242749 (visited on 06/07/2025).

[14] *Data.Hashtable.Class*. [Online]. Available: https://hackage.haskell.org/package/hashtables-1.4.2/docs/Data-HashTable-Class.html (visited on 06/09/2025).

[15] O. Kiselyov, *Efficient and Insightful Generalization*. [Online]. Available: https://okmij.org/ftp/ML/generalization.html (visited on 06/09/2025).

A Appendix

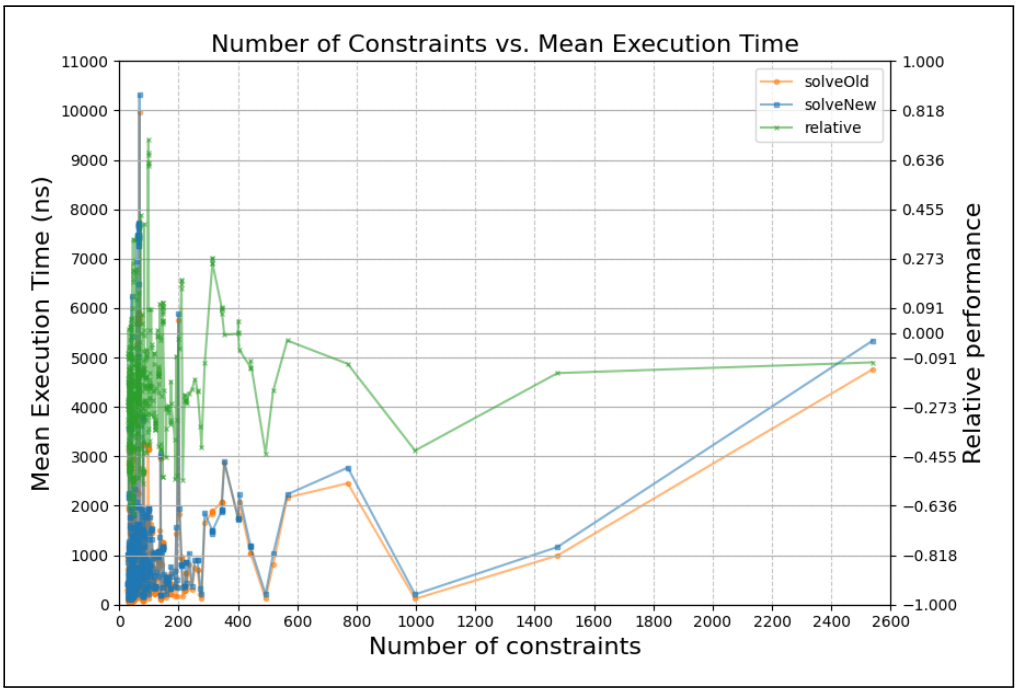
A.1 Metrics From Datasets Generated During Benchmarking

Datasets generated during Benchmarking of Futhark Programs				
Origin	**Shorthand name**	**Mean**	**Median**	**Standard Deviation**
Old implementation	`solveOld`	1119.31	651.24	1427.19
New implementation	`solveNew`	1338.78	862.17	1738.25
New implementation without union-by-weight	`solveNew w/o ubw`	1333.25	853.21	1732.29
New implementation with no heuristics	`solveNew w/o heuristics`	1304.12	843.34	1708.98
Relative performance between solveOld and solveNew	`relative`	-0.2214	-0.2322	0.1975
Datasets generated during Benchmarking of Synthetic Scenario				
Origin	**Shorthand name**	**Mean**	**Median**	**Standard Deviation**
Old implementation	`solveOld`	2191.62	1381.88	2202.46
New implementation	`solveNew`	239.02	239.71	145.54
New implementation with no heuristics	`solveNew w/o heuristics`	2837.69	2074.04	2582.98

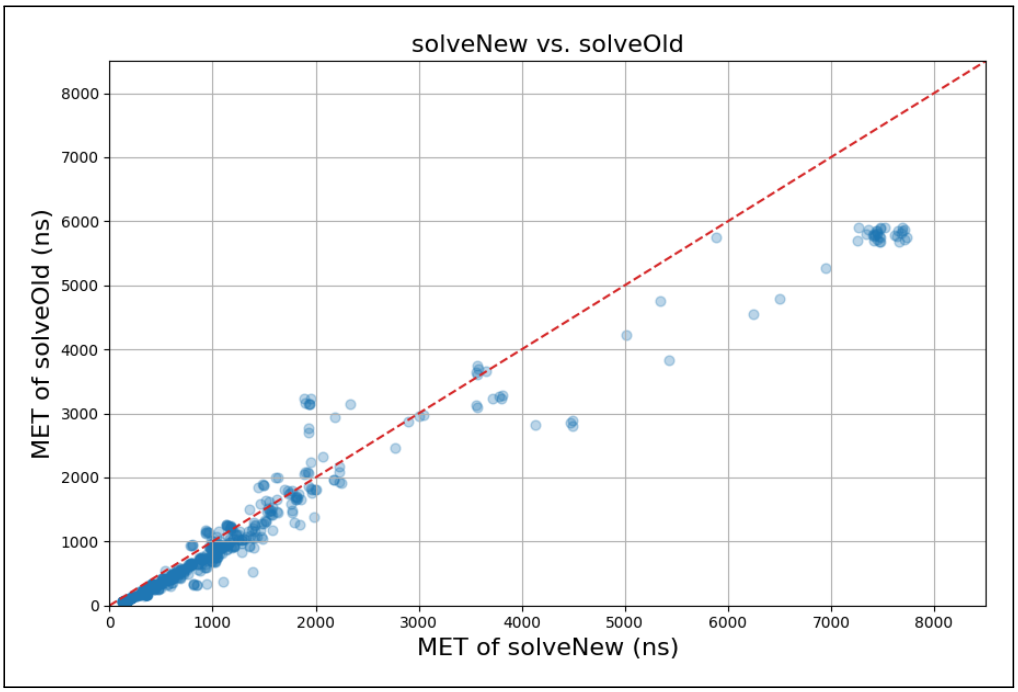
All metrics are calculated from the Mean Execution Time, and the time unit is thus nanoseconds, except for the `relative` dataset as it represents the relative performance between `solveOld` and `solveNew`.
All metrics are rounded to nearest 2 decimals, except for the `relative` dataset which is rounded to nearest 4 decimals.

A.2 Enlarged Graphs

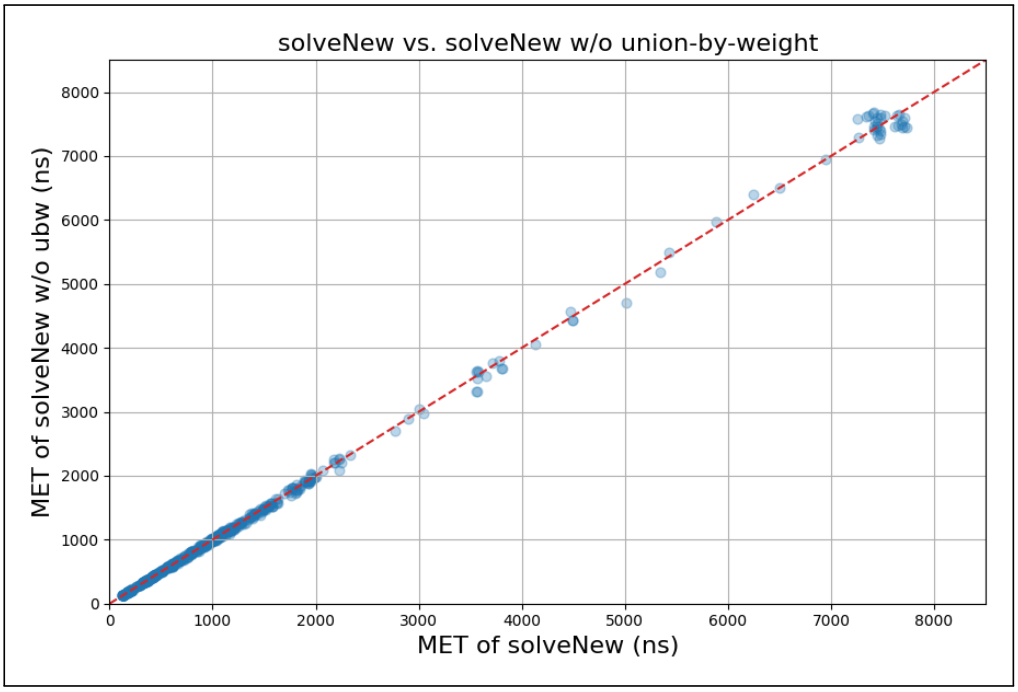
A.2.1 Figure 4



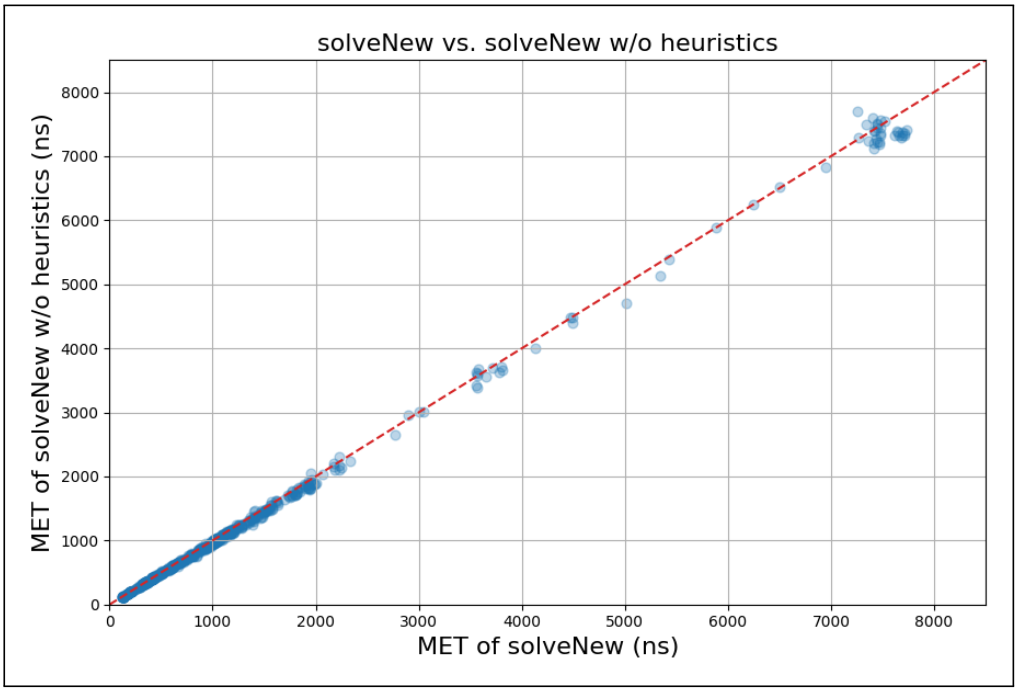
A.2.2 Figure 5



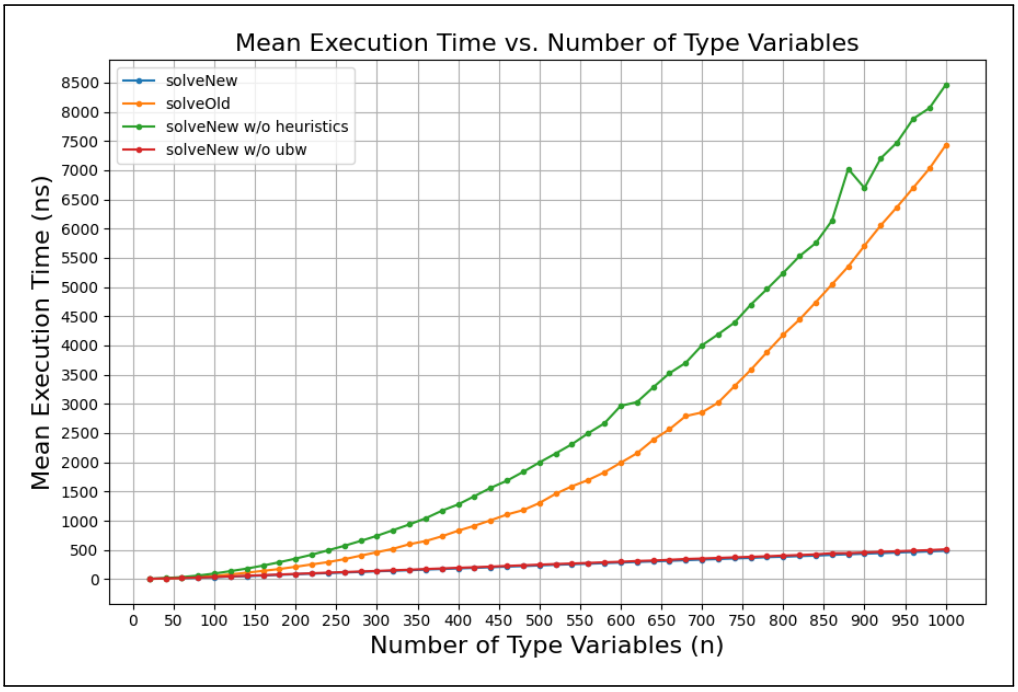
A.2.3 Figure 6a



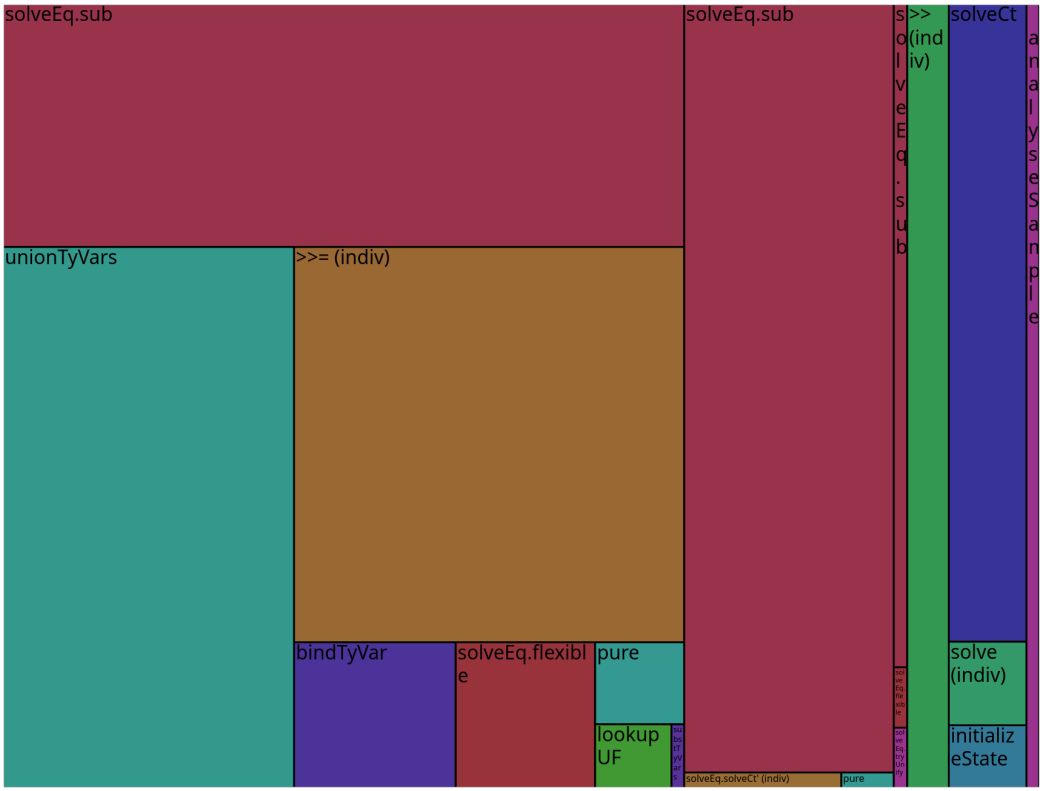
A.2.4 Figure 6b



A.2.5 Figure 7


Mean Execution Time vs. Number of Type Variables
solveNew
solveOld
solveNew w/o heuristics
solveNew w/o ubw
Mean Execution Time (ns)
Number of Type Variables (n)
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
5500
6000
6500
7000
7500
8000
8500
0
50
100
150
200
250
300
350
400
450
500
550
600
650
700
750
800
850
900
950
1000

A.2.6 Figure 8


solveEq.sub
solveEq.sub
solveEq.sub
>> (indiv)
solveCt
analyseSample
unionTyVars
>>= (indiv)
bindTyVar
solveEq.flexible
pure
lookup UF
solveEq.solveCt' (indiv)
pure
solve (indiv)
initializeState

A.2.7 Figure 9


>>= (indiv)
solveEq.sub
getSolution
solveCt
analyseSample
bindTyVar
solveEq.flexible
solveTyVar
solveEq.tryUnify
pure
lookupUF
solveEq.sub
solveEq.solveCt' (indiv)
>> (indiv)
solveTyVar
solve (indiv)
solveEq.tryUnify
solveEq (indiv)
solveEq.sub
initializeState

A.3 AI declaration

Deklaration for anvendelse af generative AI-værktøjer (studerende)

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

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

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

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

ChatGPT https://chatgpt.com/
Claude.ai https://claude.ai/login?returnTo=/?
Google Gemini https://gemini.google.com/app

Beskriv hvordan generativ AI er anvendt i opgaven:

1) *Formål (hvad har du/I brugt værktøjet til)*
   Vi har brugt GAI til kilde- og litteratursøgning, men også som en slags ekstra vejleder til f.eks. at forklare kode eller teori for os. Vi har også i mindre grad brugt det til at give forslag til strukturering af vores afsnit i rapporten.

2) *Arbejdsfase (hvornår i arbejdsprocessen har du/I brugt GAI)*
   Vi har brugt GAI i mere eller mindre alle dele af arbejdsprocessen, dvs. både ift. litteratursøgning, forklaring af eksisterende kode samt rapportskrivning (som nævnt ovenfor ved at få hjælp til strukturering af afsnit).

3) *Hvad gjorde du/I med outputtet (herunder også, om du/I har redigeret outputtet og arbejdet videre med det)*
   Vi har læst outputtet, og i nogle tilfælde har vi yderligere udspurgt den om noget, f.eks. for at få en mere dybdegående forklaring af noget.

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