UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Bachelor thesis

Jacob Aleksandar Siegumfeldt (`dzb977`) and Laust Kjæp Dengsøe (`frh993`)

Optimizing Futhark's Type Checker

Supervisor: Troels Henriksen

Handed in: June 10, 2025

Abstract

Contents

1 Introduction

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, a more suitable approach is to generate every constraint that can be inferred from a given expression and then solve these constraints afterwards. In other words, our focus is on making 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.

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 monotypes, 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*. However, formally we define a substitution $S : \mathcal{T} \to \mathcal{T}$ that maps types to types:

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

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 $\texttt{int} \to \alpha$.

2.1.4 Unification

We say that a type substitution S *unifies* the constraint $\tau_1 \equiv \tau_2$ if $S(\tau_1)$ is *syntactically* equal[1] 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. Given two substitutions S and S', we write $S \circ S'$ to denote composition of the two substitutions such that $(S \circ S')(\tau) = S(S'(\tau))$.

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 $S = S' \circ \rho$, where $\circ$ denotes function composition as usual such that, in general, $(S \circ S')(\tau) = S(S'(\tau))$. 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 $\alpha \equiv \beta$.

[1] 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.

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 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 $\gamma \equiv \texttt{string}$ (where γ is a type variable) since we cannot unify the types `int` and `string`. We'll come back to some of the other related reasons MGUs in section 2.1.6 .

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

2.1.5 A Unification Algorithm

To determine if a constraint c is solvable and, if it is, provide a unifying substitution, we define the unification algorithm shown below.

```
unify $\tau_1$ $\tau_2$ =
  case ($\tau_1$, $\tau_2$) of
    ($\tau_a \to \tau_b$, $\tau_c \to \tau_d$) →
      $S$ = unify $\tau_a$ $\tau_c$
      $S'$ = unify $S(\tau_b)$ $S(\tau_b)$
      return $S' \circ S$
    ($\alpha, \tau$) → bind $\alpha$ $\tau$
    ($\tau, \alpha$) → bind $\alpha$ $\tau$
    ($\tau, \tau'$) →
      if $\tau = \tau'$ then
        return []
      else
        fail "types do not unify"

bind $\alpha$ $\tau$ =
  if $\alpha = \tau$ then
    return []
  else if $\alpha \notin \tau$ then
    return [$\alpha \mapsto \tau$]
  else
    fail "occurs check fails"
```

Here, $\alpha \notin \tau$ means that α does not *occur* in τ, referred to as the 'occurs check' in much of the literature **[cite?]**. 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 $\emptyset$ = return []
unifySet $\{\tau_1 \equiv \tau_2\} \cup C'$ =
  $S$ = unify $\tau_1$ $\tau_2$
  return (unifySet $S(C')$) $\circ$ $S$
```

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.

2.1.6 What About Polymorphism?

In our small type system, we haven't included *type schemes* σ, that is, types that contain type variables that are universally quantified [1]. Type schemes make it possible to have types like $\forall\alpha\forall\beta.\alpha \to \beta$ where we can replace each universally quantified type variable with arbitrary monomorphic types. For instance, the identity function could be described as having the type $\forall\alpha.\alpha \to \alpha$ since we can give an input of any type to it and have it return this input which, naturally, must be of the same type.

In the context of type schemes, we also introduce the process of *instantiation*: If we conclude that some expression e has type σ, then we may replace this type with the type σ' if σ is *more general* than σ', typically denoted as $\sigma \sqsubseteq \sigma'$. **SOME MORE ABOUT WHAT THIS MEANS, AND RELATING IT TO PRINCIPAL TYPES AND MGUS**

2.2 Union-Find for Constraint-Solving

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 they occur 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 a *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 arbitrarily chosen 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 runtime 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 the proper heuristics [8, p. 527], therefore we will only describe this way of implementing the data structure. When using forests, *trees* represents sets. *Nodes* corresponds to elements, and point to a parent. 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. An example of this heuristic can be seen in figure 1.

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

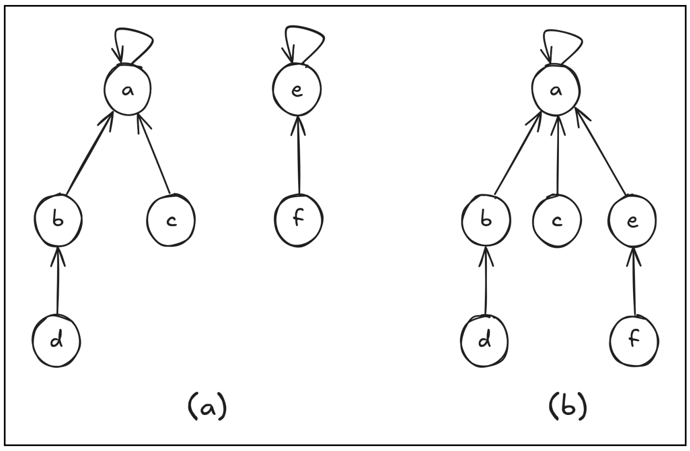

Figure 1: An example of the *union-by-weight* heuristic. **(a)** shows two trees, the one on the left has weight four, and the one on the right has weight 2. **(b)** shows the resulting tree after calling `Union(a,e)`.

the path to the root two times. The first time, it discovers what node is the root. The second time, it changes the parent pointer of the node being visited to the root. Thereby, the path to the root has been *compressed*, since the node is now pointing directly at the root. Figure 2 shows an example of this heuristic.

2.2.3 Implementation of Union-Find

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 has 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, and $x.w$ is the weight of the tree. This weight is only relevant for roots of trees, not for all nodes.

```
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

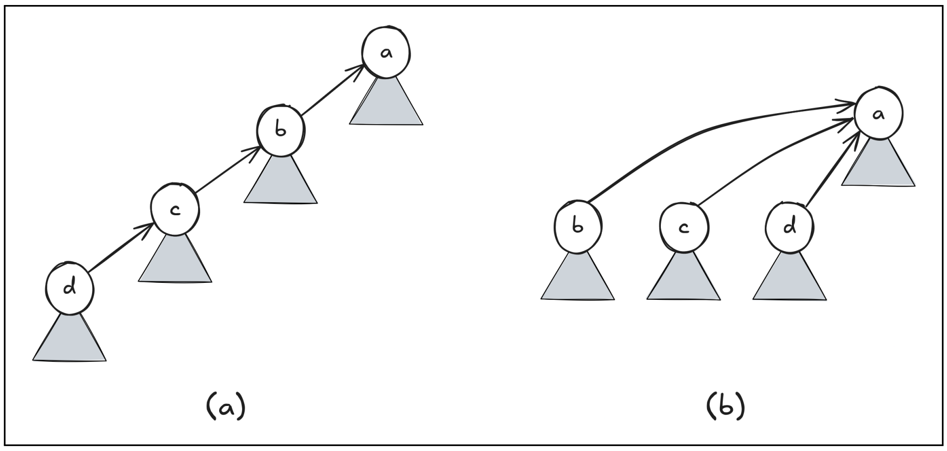

Figure 2: An example of the *path compression* heuristic. The arrow pointer at the roots have been omitted for simplicity. **(a)** shows a tree before `Find(d)` is called. The triangles represent subtrees with the shown nodes as roots. **(b)** shows the resulting tree after calling `Find(d)`.

given node is a root, and if not recurcively find the root and make it the parent. Lastly, it should return the root. This is shown in listing 2.

```
1: procedure Find($x$)
2:     if $x.p \neq x$ 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.

The union of two trees has to ensure that the smaller tree, 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 is an optimal implementation of the union-find data structure.

2.2.4 Runtimes of Operations

The three operations introduced previously form the basis of an implementation of the union find data structure. A single `Make` operation trivially takes constant time, $O(1)$. The analysis of the two remaining operations are not as trivial. A `Find` operation has an amortized cost of $O(\alpha(n))$, where $\alpha(n)$ is an extremely slowly growing function, when *union-by-weight* and*path compression* is

```
procedure Union($x, y$)
    $xRoot$ = Find($x$)
    $yRoot$ = Find($y$)
    if $xRoot.w > yRoot.w$ then
        $yRoot.p = xRoot$
        $xRoot.w = xRoot.w + yRoot.w$
    else
        $xRoot.p = yRoot$
        $yRoot.w = yRoot.w + xRoot.w$
    end if
end procedure
```

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

used [8, p. 539]. The runtime of the `Union` operation only using *union-by-weight* is $\Theta(m \log n)$, where m is the number of `Make` operations, and n is the number of `Find` operations [9]. However, when path compression is also used, this runtime reaches the asymptotically optimal runtime of $\Theta(m\alpha(n))$, [9] [8, ch. 19]. Thus, to achieve an asymptotically optimal implementation of the union-find data structure, one must use disjoint-set forests in combination with the heuristics described in section 2.2.2.

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 using the union-find data structure is provided.

```
unify $\tau_1$ $\tau_2$ =
    case ($\tau_1$, $\tau_2$) of
        ($t_1$,$t_2$) | ($t_1$ has type $\alpha$ **and** $t_2$ does not have type $\alpha$) →
            let $t$ = find $t_1$
            case $t.t$ of
                free →
                    if $\alpha \in t_2$ then error "Occurs check failed"
                    $t.t$ = $t_2$
                ty | ty == $t_2$ → return
                ty → unify $t.t$ $t_2$

        ($t_2$,$t_1$) | ($t_1$ has type $\alpha$ **and** $t_2$ does not have type $\alpha$) →
            unify $t_1$ $t_2$

        ($t_1$,$t_2$) | ($t_1$ has type $\alpha$ **and** $t_2$ has $\alpha$) →
            let $t_1$ = find $t_1$
            let $t_2$ = find $t_2$
            if $t_1$ == $t_2$ then return
            case ($t_1.t$,$t_2.t$) of
                (free,$t$) | $t \neq$ free → unify $t_1$ $t_2.t$
                ($t$,free) | $t \neq$ free → unify $t_2$ $t_1.t$
                (free,free) → continue -- wrongful syntax?
                _ → unify $t_1.t$ $t_2.t$
            union $t_1$ $t_2$

        ($t_1 \to t_2$,$t_3 \to t_4$) →
            unify $t_1$ $t_3$
            unify $t_2$ $t_4$

        ($i$,$i$) | ($i$ has type int) → return
        (_,_) → error "Type mismatch"
```

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

As with the algorithm in listing **??**, this algorithm is called on each given constraint, where τ_1 is the type that must be unified with τ_2. The different types are defined in section 2.1.1.

In the beginning of the constraint-solving process using union-find, i.e. before any solving, each type variable is initialized as a tree with a single root, which in this context is called an equivalence class. The root contains information, such as the weight of the tree, $r.w$, the pointer to the parent node, $r.p$, and the type assigned to the equivalence class, $r.t$, for a given root r. Each root is initialized with $r.w = 1$, $r.p = r$ and $r.t = \texttt{free}$.

The field $r.t$ is crucial as it contains the type information for the given equivalence class. It is intitialized to `free`, since all type variables are free, until they are bound to a type. Such a binding happens on line 8, where the type of the representative of an equivalence class is set to a concrete type. Before this binding, the `Find` operation is used on line 4 to find the representative of the equivalence class, thereby ensuring that the binding applies for all members of the equivalence class, that node t_1 is part of.

The implementation of `Find` can be done in a manner similar to the one provided in listing 2. It is also during this binding process (among others), that the imperative nature of the union-find data structure can be observed, as the binding on line 8 requires a modification of the state of the node. This is an important revelation, as it has an influence on the implementation of the data structure in Haskell, as we shall see in section 3.2.

Union happens when the types to be unified are both type variables, i.e. the case on line 15. As before, the representatives of the equivalence classes, that τ_1 and τ_2 are part of, are retrieved using `Find`. If the representatives are the same, then the type variables are already part of the same equivalence class, and thus no further action is needed, as per line 18. However, if the representatives are different, the match case on lines 19-23 ensures that the unification is valid and that types are bound if necessary, before the equivalence classes are finally unioned on line 24. The implementation of the `Union` operation should be similar to the one provided in listing 3.

The provided pseudocode thus shows an efficient implementation of a unification algorithm using union-find as the data structure. However, in practice this algorithm must be extended to handle the features that most programming languages, including Futhark, require, as the following section will explore.

2.3 Practical Complications

SOMETHING ABOUT OVERLOADED TYPES, LEVELS OF TYPE VARIABLES, AND LIFTEDNESS

3 Implementation

3.1 Overview

This section covers the optimizations we implemented in Futhark's type checker, which is written in Haskell. At first, 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. Next, 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.

3.2 Union-Find in Haskell

Haskell is a purely functional programming language, and thus it does not allow side effects. This means, that it can be challenging to effieciently implement certain algorithms that are imperative by nature in Haskell [10]. The union-find data structure intrduced in section 2.2.3 relies on mutability when performing certain operations, such as path compression where the parent pointer of a node is mutated in memory. This mutation of memory is what allows the implementation of the heuristics described in section 2.2.2. These heuristics are necessary to achieve the optimal implementation of the union-find data structure.

To achieve mutability in Haskell, one can use the `ST` monad[2], as it allows the use of the `STRef`, which is a mutable reference within a state thread. The `STRef` can be used as a variable of type `a` on the state thread, `s`, like so `STRef s a`. Our implementation of the data structure can be found in the `UnionFind.hs` module. By using the `STRef`, the following representation of nodes can be achieved.

```haskell
newtype Node s a = Node (STRef s (Link s a))

data Link s a
  = Repr (STRef s (ReprInfo a))
  | Link (Node s a)
```

A given node contains a reference to a `Link`, which can either be a representative or a link to another `Node` (described as a *parent pointer* in section 2.2.3). A representative contains the necessary information, about set, that it is representing. This information is also stored in `ReprInfo`, since it must also be mutable, it is thus an `STRef`. The only required information, that it must contain is the weight of the set. However it is permissable to store other information that is relevant in the context of its implementation (as we do in section 3.3.4).

[2] https://hackage.haskell.org/package/base-4.21.0.0/docs/Control-Monad-ST.html

The `find` operation of the union-find data structure is implemented in a way, that is similar to the pseudocode provided in listing 2. To implement it in Haskell, one must utilize the `readSTRef` and `writeSTRef` functions. The former function is needed to find the representative, `Repr`, of the given node's set, since this information is found by reading from the `STRef` and traversing the `Link`s, until a `Repr` is found. The latter function is used to do path compression, since that requires overwriting the `STRef` of the `Link`s in the traversed nodes.

`Union` is implemented similarly to the pseudocode in Listing 3. As the `find` operation, it also ultilizes the `readSTRef` and `writeSTRef` functions. `readSTRef` is used to obtain the `Repr` in the `Link`s of the roots returned by `find`. Afterwards it must be used again to obtain the weights found in `ReprInfo`. `writeSTRef` is used to write the summed weight to the new root and to overwrite the `Repr` with a `Link` in the node, that is no longer root.

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[3] and *rank* constraints[4] are not handled in this module.

3.3.1 The `SolveM` monad *(MÅSKE OVERFLØDIG)*

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

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

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

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

[4]As described in Schenck, Hinnerskov, Henriksen, *et al.* [11].

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 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. 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* (or *ad hoc polymorphic*), that is, it is defined for multiple (though not arbitrary) types [12, p. 192], `x`, `y`, and the return type of `f` could be any numeric type supported in Futhark[5]. We'll come back to this issue and how it's solved when we discuss how constraints are actually solved **REMEMBER**, but this is an example of a program which will generate flexible but constrained type variables, as mentioned in section 3.3.2.

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:

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

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 paramater are placed in a `TyVarNode` as a `Repr` (desribed in section 3.2), meaning that they will each represent their own equivalence class. However, a representative should also contain information regarding the class, which is stored in the `STRef s (ReprInfo a)` mentioned in section 3.2. In our implementation this `ReprInfo` contains the following information:

1. The *weight* of the equivalence class inorder to uphold the *union-by-weight* heuristic described in section 2.2.2.
2. The *key*, which is the name (called `TyVar` in the previous section) 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 is the representative is, i.e. 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 τ can be divided into three cases:

1. If τ is a solved type variable, we substitute it with the type we've assigned to it;
2. if τ 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 τ itself.

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

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

If these constraints are processed from left to right, the last constraint will be converted to the constraint $\texttt{int} \equiv \texttt{int} \rightarrow \texttt{int}$. 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 $\texttt{int} \equiv \texttt{int} \rightarrow \texttt{int}$ is unsolvable, that is, there is no substitution S that would make $S(\texttt{int})$ equal to $S(\texttt{int} \rightarrow \texttt{int})$.

THIS PART MIGHT BELONG IN THE *EVALUATION* SECTION: This is also a part of the code that benefits a lot from having path compression: Consider, for instance, the constraint set

$$\{\alpha_i \equiv \alpha_{i+1} \mid 0 \leq i \leq n-1\} \cup \left\{\alpha_j \equiv \texttt{int} \mid 0 \leq j \leq n\right\}$$

Assuming that each constraint solely involving type variables is solved by letting the left-hand side of the constraint point to the right-hand side – which is more or less what actually happens in the constraint solver – we might end up with a chain of length n of type variables before we begin processing the constraints on the form $\alpha_j \equiv \texttt{int}$. Then, for each type variable α_j in the constraints on the form $\alpha_j \equiv \texttt{int}$, we must take $n-j$ "steps" through the chain to resolve the type of α_j, and since $\Sigma_{j=0}^{n} j = \Theta(n^2)$ **(this might need to be explained in another manner)**, normalization would take time $\Theta(n^2)$ in this case. Contrastingly, with path compression, normalization is done much more efficiently since, for each type variable α_j we visit as we step through the chain, α_j will end up pointing directly to α_n.

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.

DESCRIBE WHAT HAPPENS WHEN A CONSTRAINT CONTAINS TYPE VARIABLES

4 Evaluation

4.1 Testing

4.2 Benchmarking

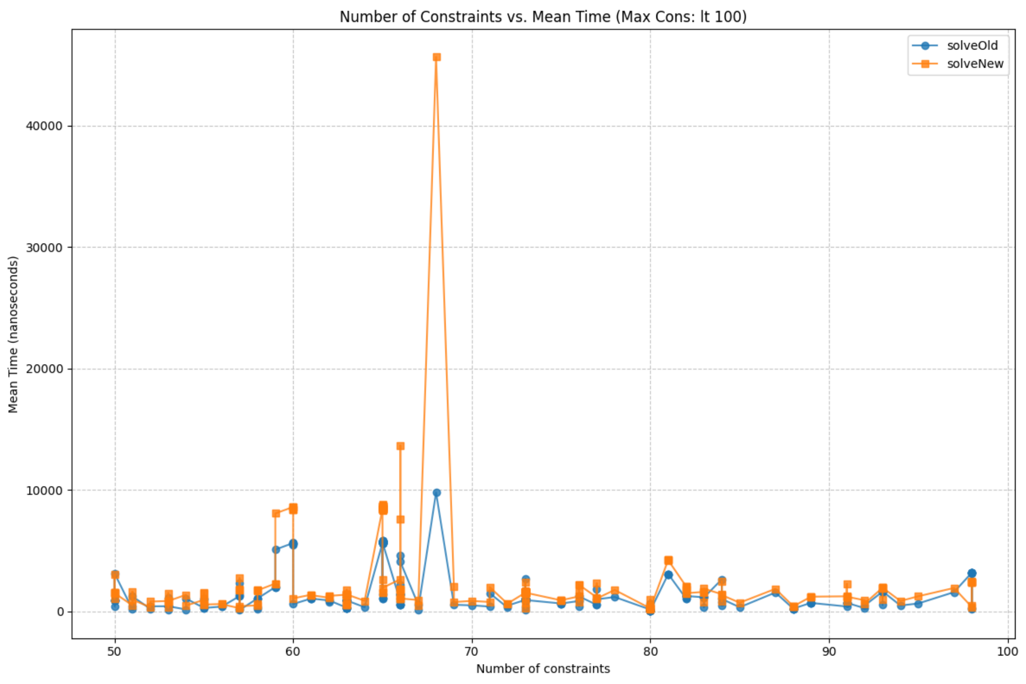

Figure 3: Enter Caption

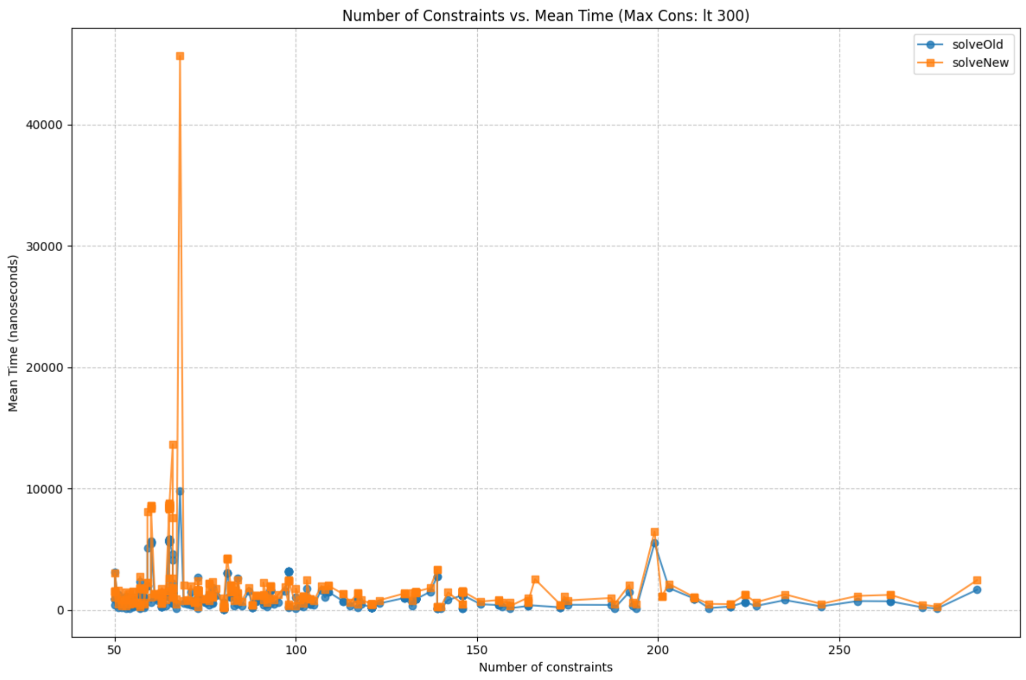

Figure 4: Enter Caption

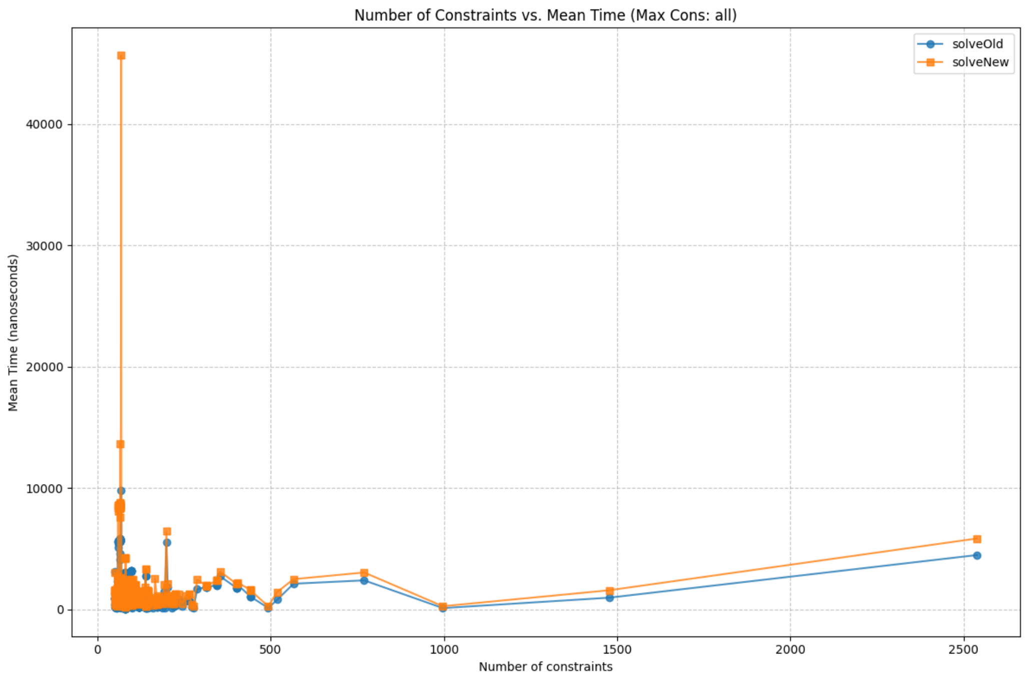

Figure 5: Enter Caption

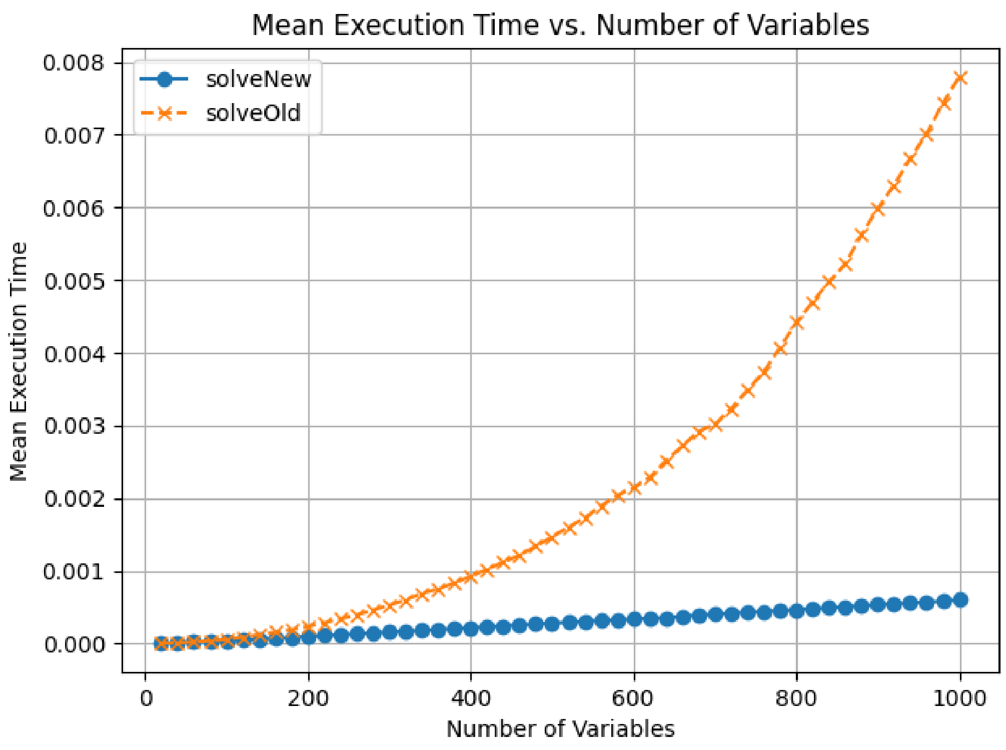

Figure 6: Enter Caption

4.3 Profiling

To try to gain some information about which parts of the original and updated version of `TySolve.hs`, respectively, acted as performance bottlenecks, we've also tried to profile some different Futhark programs.

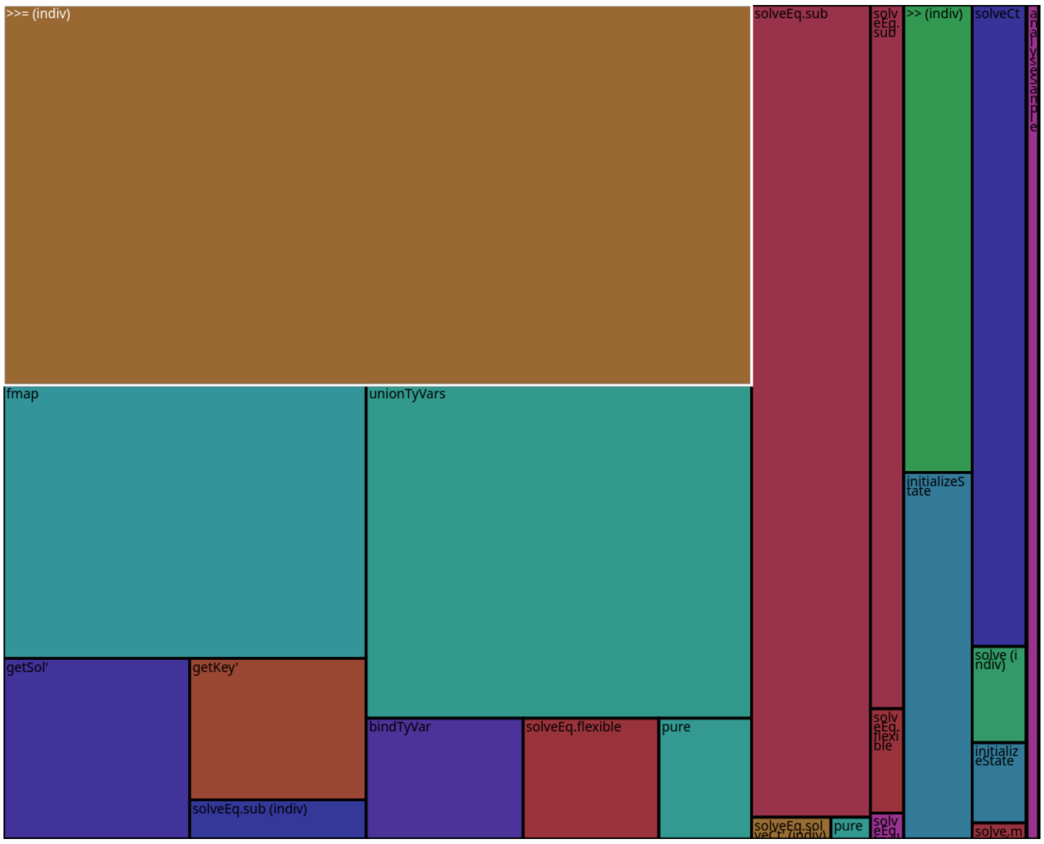

Figure 7: Profiling of `hashcat.fut (Block 2/4)` benchmark

4.4 Discussion

- General improvements of the code
- Optimize occurs check
- No unit tests with sum types
- Unit tests mostly focus on constraints involving type variables; should probably involve all types of constraints
- Could avoid using `lift` to lift `ST` into the `SolveM` monad
- More efficient lookups of type variables (hash table?)
- Heuristic(s) for choosing order of constraints to be processed?

5 Conclusions and Future Work

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*, Fourth edition. Cambridge, Massachusetts London: The MIT Press, 2022, isbn: 978-0-262-04630-5 978-0-262-36750-9.

[9] R. E. Tarjan and J. Van Leeuwen, “Worst-case Analysis of Set Union Algorithms,” *Journal of the ACM*, vol. 31, no. 2, pp. 245–281, Mar. 1984, issn: 0004-5411, 1557-735X. doi: 10.1145/62.2160. [Online]. Available: https://dl.acm.org/doi/10.1145/62.2160 (visited on 05/26/2025).

[10] C. G. Ponder, P. McGeer, and A. P.-C. Ng, "Are applicative languages inefficient?" en, *ACM SIGPLAN Notices*, vol. 23, no. 6, pp. 135–139, Jun. 1988, issn: 0362-1340, 1558-1160. doi: 10.1145/44546.44559. [Online]. Available: https://dl.acm.org/doi/10.1145/44546.44559 (visited on 06/05/2025).

[11] 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).

[12] 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.