UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Bachelor thesis

Jacob A. Siegumfeldt, Laust K. Dengsøe

Optimizing Futhark's Type Checker

Advisor: Troels Henriksen

Handed in: June 10, 2025

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 better 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 solving type constraint, we start by considering a small language, similar to the λ-typed calculus, with its (monomorphic) 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.

2.1.2 Type Constraints

Formally, we write a *type constraint* as

$$\tau_1 \doteq \tau_2$$

where τ_1 and τ_2 are both monomorphic 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*. A type substitution σ is a (finite) map from type variables to types. However, formally we define substitution over types in general in the following way:

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

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

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

Note that all substitutions are performed *simultaneously*: For instance, given the substitution $\sigma = [\alpha \mapsto \texttt{int}, \beta \mapsto \texttt{int} \to \alpha]$, applying σ 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 σ *unifies* the constraint $\tau_1 \doteq \tau_2$ if $\sigma(\tau_1)$ is *syntactically* equal to $\sigma(\tau_2)$. Extending the notion of solvability to constraint sets, a type substitution σ solves a constraint set C if σ solves every type constraint

in C; we call such a substitution σ a *solution* or *unifier* for C. Given two substitutions σ and σ', we write $\sigma \circ \sigma'$ to denote composition of the two substitutions such that $(\sigma \circ \sigma')(\tau) = \sigma(\sigma'(\tau))$.

As there might be multiple unifiers for a constraint set [1], 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 $\sigma \in \mathcal{U}(C)$ there exists a type substitution σ' such that $\sigma = \sigma' \circ \rho$, where $\circ$ denotes function composition as usual such that, in general, $(\sigma \circ \sigma')(\tau) = \sigma(\sigma'(\tau))$. 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 $\alpha \doteq \beta$.

The reason why we're interested in a *most general* unifier of some constraint (set) is because we don't want to limit what type a type variable might represent. The constraint $\alpha \doteq \beta$ could also be solved with the substitution $\sigma = [\alpha \mapsto \texttt{int}, \beta \mapsto \texttt{int}]$ 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 $\alpha \doteq \gamma$ and $\gamma \doteq \texttt{string}$ (where γ is a type variable) since we cannot unify the types `int` and `string`.

Summarizing, to solve a set of constraints C we must find a substitution σ 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 set of constraints C is solvable and, if it is, find an MGU for C, we define the following unification algorithm:

```
unify ((τ₁, τ₂) : C) =
  case (τ₁, τ₂) of
    (τₐ → τ_b, τ_c → τ_d) →
```

2.2 Union-Find for Constraint-Solving

In offline constraint-solving, the goal is to divide the given type variables into their respective equivalence classes depending on the given constraints, thereby obtaining a solution. The underlying data structure of the constraint solving algorithm is thus important, since each type variable might be accessed many times based on how often they occur in the constraints. The union-find data structure

is therefore an efficient data structure for constraint-solving algorithms, since it allows for fast lookups of nodes when implemented optimally [2, p. 538]. It is particularly useful for applications that require the grouping of multiple distinct elements into disjoint sets [2, 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 an imperative data structure that relies on disjoint sets to represent sets that are disjoint, meaning that two distinct sets cannot contain the same elements. The data structure supports three basic operations: creating a set, union of two sets, and finding the set that a given element is part of. Each set has a representative or root, which can sometimes just be any arbitrary element in the given set, however in some cases (as in ours) a particular element is picked based on some property or heuristic.

The operation of making a set simply creates a set given some element. This element is made the root of the newly created set. Given some element, the find operation returns a pointer to the root of the set that the element is part of. The union operation is the most "complex" operation, as it first finds the roots of the sets, that the two elements to be joined, say x and y, are part of. Then a new set, S, is created containing all the elements in both S_x and S_y yielding $S = S_x \bigcup S_y$. As mentioned previously, the root of S can be arbitrarily chosen among the elements or chosen by some property or heuristic.

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 [2, p. 527], therefore we will only describe this way of implementing the data structure. When using forests, each tree represents a set. 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. [2, 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.

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. 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.

2.2.2 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 [2, p. 530].

The first operation of making a set is very simple, and could be implemented as follows. In the below, $x.p$ 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
```

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 recurcively find the root and make it the parent. Lastly, it should return the root. This is shown below.

```
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
```

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 the following.

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

10: **end if**
11: **end procedure**

These three operations form the basis 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, [2, p. 539]. When path compression is not used, but union-by-weight is, the runtime of the Union operation is $\Theta(m \log n)$, where m is the number of Make operations, and n is the number of Find operations, [3]. However, when using path compression this runtime reaches the asymtotically optimal runtime of $\Theta(m\alpha(n))$, [3] [2, ch. 19].

2.2.3 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. When using union-find for constraint-solving, each type variable has a corresponding node.

2.2.4 Union-Find in Haskell

2.3 Practical Complications

3 Implementation

4 Evaluation

4.1 Testing

4.2 Profiling

4.3 Benchmarking

4.4 Discussion

5 Conclusion and Future Work

References

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

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, eng, Fourth edition. Cambridge, Massachusetts London: The MIT Press, 2022, isbn: 978-0-262-04630-5 978-0-262-36750-9.

[3] R. E. Tarjan and J. Van Leeuwen, "Worst-case Analysis of Set Union Algorithms," en, *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).