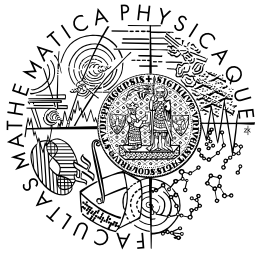

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Tomáš Husák

Improving Type Inference in the C# Language

Department of Distributed and Dependable Systems

Supervisor of the master thesis: Mgr. Pavel Ježek, Ph.D.
Study programme: Computer Science
Study branch: Software Systems

Prague 2024

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

TODO: Dedication.

Title: Improving Type Inference in the C# Language

Author: Tomáš Husák

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek, Ph.D., Department of Distributed and Dependable Systems

Abstract:

TODO: Abstract.

Keywords: Type Inference C# Roslyn

Contents

1. Introduction

Note: Tell what is C# and a goal of the thesis

C# is an object-oriented programming language developed by Microsoft. It belongs to the strongly typed languages helping programmers to possibly reveal bugs at compile time. The first part of this thesis focuses on exploring type systems of strongly typed languages and proposes an improvement to the C# type system. The second part concerns the implementation of the improvement in the current C# compiler and the creation of a proposal that will likely be discussed by the Language Design Meetings (LDM) accepting new C# language features.

1.1 Improving C# type system

Note: What is type inference in context of strongly typed languages

A key feature of strongly typed languages is type safety, prohibiting operations on incompatible data, achieved by determining data types at compile time. The easiest way for a compiler to reason about types of variables in the code is by providing type annotations determining the data type that these variables hold. We can see an example of a type annotation given by a programmer using an example 1.1 written in the C# programming language. The type declaration of the `people` variable guarantees that any possible chances to threaten it differently from `List<T>` will be reported at compile time to save the programmer time to debug it. On the other hand, the programmer has to write more code to annotate a variable declaration whose type has a long name, as we can see in the listing. This disadvantage of strongly typed languages can be removed by *Type inference* when a missing type annotation can be deduced using the context. Taking our example, we can notice redundancy of type annotation `List<String>` in the code. Since we do the initialization and type declaration in the same place, the declared variable `people` has to have the same type as the initializing value. The use of type inference can be seen in the `myFriend` variable declaration, where we used the `var` keyword triggering C# type inference to determine the variable's type being the type of initializing value, `System.String` in this case.

Note: Describe C# and Rust type inference in context of generics

Scenarios where type inference can deduce a type vary in strongly typed languages. An example can be seen in type arguments deduction of generic methods. In the context of C#, a generic method is a method that is parametrized by types besides common parameters, as you can find in code 1.2. Although the type in-

```
using System.Collections.Generic;

List<string> people = new List<string>() {"Joe", "Nick", "Mike"};
people += "Tom"; // Error reported during compilation
var myFriend = "Tom";
```

Figure 1.1: Type annotations in the C# programming language.

```
Foo("Tom");
int temp = Bar(); // Error reported during compilation

void Foo<T> (T arg) { ... }
T Bar<T>() { ... }
```

Figure 1.2: C# Type inference of generic methods.

```
fn main() {
    let elem : Option<u8> = foo();
}

fn foo<T>() -> Option<T> { return None; }
```

Figure 1.3: Rust Type inference of generic methods.

ference deduces type arguments of the first generic method `Foo`, it fails to deduce type arguments of `Bar` even though it could be possible in this case since we know the method's return type.

When we compare it with example 1.3, demonstrating similar functionality written in Rust language, which belongs to strongly type languages too, we can see that Rust's type inference uses the target type to deduce the type arguments.

Note: Introduce Hindley-Millner type system and type inference as a formalization

Type inference capabilities of C# and Rust can be formalized by Hindley-Millner type inference [22] used by these languages in a modified way. Traditional Hindley-Millner type inference is defined in the Hindley-Millner type system [21], where it can deduce types of all variables in an entirely untyped code. The power of type inference is caused by properties of the type system, which, in comparison with the C# type system, doesn't use type inheritance or overloading. Despite these barriers, Hindley-Millner type inference can be modified to work with other type systems like Rust or C#, causing limited use cases where it can be applied.

Note: Present goal of this part of thesis

The first part of the thesis aims to explore possible extension of C# type inference based on Rust's type inference observation and the theoretical background given by Hindley-Millner type inference, which these languages use with modifications.

1.2 Implementation

Note: Describe proposing a new feature

C# is an open-source project where the community can contribute by fixing issues of the compiler, proposing new language features, and elaborating on implementing them. Proposing new C# features is done in public discussions of the C# language repository [15], where everyone can add his ideas or comment on others' ideas. Although there is no required structure for how the idea should be described, LDM created a template [17] containing a base structure for proposing the feature in order to make the idea more likely to be discussed

by the team. The template includes motivation, detailed description, needed C# language specification [16] changes, and other possible alternatives.

Note: Roslyn

Language feature prototypes are implemented in feature branches of the Roslyn repository [19], which contains an open-source C# compiler developed by Microsoft and the community.

Note: Present goal of the second thesis part

The process of language proposal ends with LDM accepting or declining it. The second part of this thesis regards creating the proposal describing our improvement using the prepared template and implementing it in Roslyn's feature branch.

1.3 Summary

Note: Goals of this thesis

We summarize goals of this thesis in the following list:

G1. Explore possibilities of type inference in strongly typed languages

G2. Improve C# type inference based on previous analysis

G3. Create an proposal containing the improvement

G4. Implement the prototype in Roslyn

2. Related work

In the introduction, we presented the programming language C# and its possible improvement of type inference. This chapter continues by describing relevant sections of the C# language and its type inference algorithm to understand the possible barriers to implement improved type inference. As a primary source of inspiration for the improvement, we will explore Hindley-Milner type inference in more detail and describe its modification in Rust and C# programming languages. For the third goal of this thesis, we will mention relevant C# language issues presented on the GitHub repository, which we use later to prioritize the improvement features to make it more likely to be accepted by LDM.

2.1 C# programming language

Note: Explain purpose of this section

Although C# language features complement each other, we try to extract only relevant components for type inference in this section. We describe the type system, including C# generics and their possibilities. Then, we mention unrelated language constructs that influence the type inference, and we have to count on them in proposing improved type inference. At the end of this section, we list types of type inference in C# and describe them in necessary detail for the following chapters.

2.1.1 Type system

Note: Value/reference types

Note: Inheritance

As we mentioned in the introduction, each variable and expression returning a value has to have a type in the C# type system [10] called the Common Type System (CTS). Its fundamental characteristic is type inheritance, where every type directly or indirectly inherits a base type `System.Object`, as you can see in the picture 2.1. This chain of inheritance forms a tree, meaning that it is prohibited to inherit more than one type. Types are divided into value and reference types. Value types consist of built-in numeric types, structures (`struct`), and enumeration (`enum`). Compared to classes (`class`) and records (`record`) belonging to reference types, value types can't be inherited by other types. The last relevant member of reference types is interface (`interface`), which can extend multiple interfaces and be implemented by `class`, `record`, or `struct`.

Note: Nullability analysis

C# type inference infers, besides a type, its nullability, determining if it is possible to assign `null` value to that type. C# implicitly allows to assign `null` values to reference types indicating invalid value. Since C# 2.0 [12], it allows to assign `null` values to nullable value types, which are generic wrappers around value types. Because assigning `null` value is referred to as a billion-dollar mistake, C# 8.0 introduces optional settings prohibiting it and created nullable reference

types explicitly allowing `null` assignment as a way of interaction with legacy code not using the feature.

Note: Generic types and methods

An essential part of the type system is C# Generics, allowing parametrization of types and methods by other types. An example of a generic class is `System.Collections.Generic.List<T>` representing resizeable mutable array where `T` represents arbitrary data type, which we want to have a collection of. Providing a type argument for `T`, we create a new type where the type argument replaces the usages of `T`.

Note: Generic constraints

Because a type parameter can be arbitrary, C# treats it as a `System.Object`, which is insufficient in cases where the type parameter should provide special behaviour distinct from `System.Object`. This requirement is achieved by type constraints, which restrict a set of types that can be passed to the parameter. Several types of restrictions can be applied to type parameters in order to enable more actions on values of the restricted type parameters. We can see examples of type constraints in the following code 2.2, where we use implementation restriction forcing the `T` to implement an interface providing API for comparing values with the same type. The second restriction forces the type argument to have a default constructor. Another restriction concerns an obligation that the type will be a value type or the type has to be non-nullable.

Note: Variance and contra-variance

The last feature of generics influencing type inference is the concept of type variance. Initially, type parameters are invariant, meaning an obligation to assign a generic type to another generic type having the same types of type parameters.

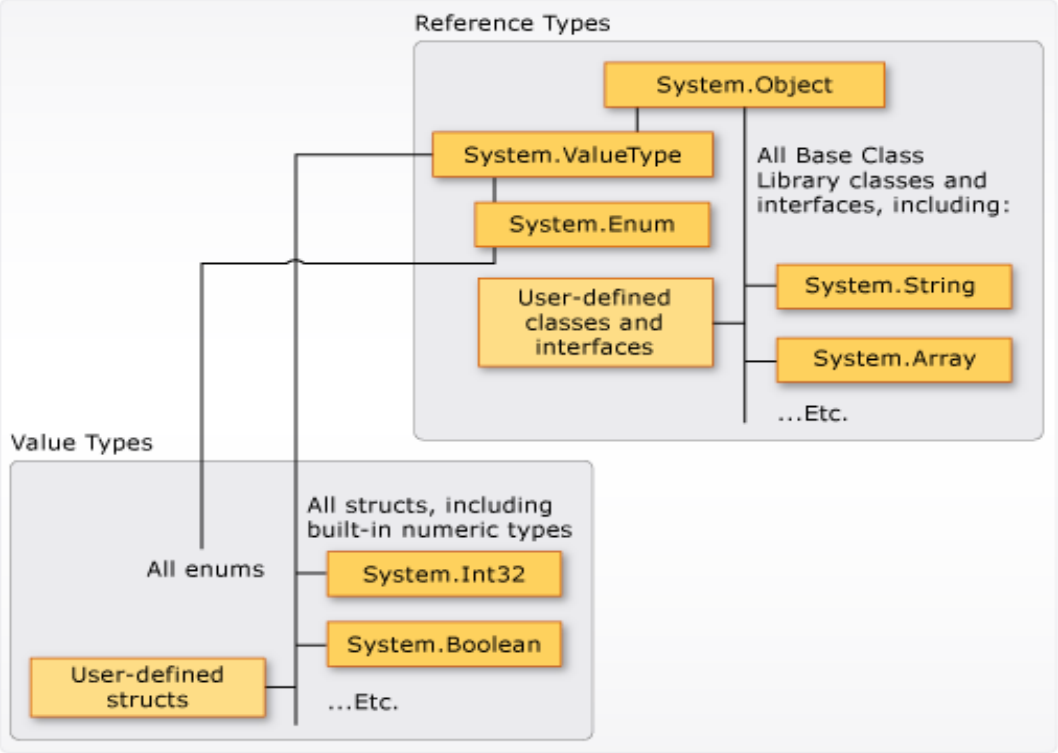

Figure 2.1: The C# type system [10].

```
class MyList<T> : where T : IComparable<T>, new()
{
    private T[] myBuffer;

    int CompareOnIndicies(int idx1, int idx2) {
        return myBuffer[idx1].CompareTo(myBufffer[idx2])
    }
}
```

Figure 2.2: C# type constraints.

Generic interfaces introduce additional modifiers (`in`, and `out`) of type parameters, which allow to assign a type with the more specialized type argument to a type with the less specialized type argument or vice-versa respectively.

Note: Overloading

We end this subsection by presenting method overloading. C# allows the definition of multiple methods with the same and count of type arguments having different parameters. We will see in the following chapters why this feature is one of the barriers to implement strong and efficient type inference.

2.1.2 Relevant constructs

For this and the following section, we use the code example 2.3 to better demonstrate the constructs and type inference.

Note: Dynamic

C# type inference mostly happens at compile time, with one exception. We previously mentioned that C# requires knowing the types of all variables and expressions during compilation. It turned out that the possibility of expressing type, which is unknown at compile time, became crucial for interoperability with other dynamic typed languages. To make the interoperability easier, C# introduced a dynamic keyword that can be used as an ordinary type, which causes late binding of all expressions containing the dynamic value. We can see an usage of the `dynamic` keyword on line 6 in the figure 2.3 where we can notice the late binding on line 7 which doesn't cause a compile time error even the referring value doesn't define any method with the name `Foo`. Internally, the type is `System.Object`, however, the compiler skips its binding and postpones it to the runtime. Although the mentioned error on line 7 occurs at runtime, the compiler still attempts to check certain expressions containing dynamic values to reveal possible errors at compile time as we can see on line 8 where an error regarding of passing `System.String` to `System.Int` occurs during compilation.

Note: Implicit typed lambdas

The next language construct influencing type inference is an anonymous function, also known as Lambda, which, instead of declaring a dedicated method with a signature and a body, allows to specify only the body with untyped parameters on places where a function delegate is required. We can see an example of Lambda expression on line 4 in the figure 2.3 where we pass it as the fourth argument. Type inference infers its signature based on the surrounding context.

```
1  var myArray = new [] {new object(), "string"};
2  List<int> myList = new();
3
4  Helper.Method2((long)1, (int)1, myList, (p1) => p1 + 1);
5
6  dynamic dynamicValue = "string";
7  dynamicValue.Foo();
8  Helper.Method1("string", myArray, dynamicValue);
9
10 class Helper {
11  static void Method1<T>(int p1, IList<T> p3) {...}
12  static void
13  Method2<T1, T2, T3>(T1 p1, T1 p2, IList<T2> p3, Func<T2, T3> p4)
14  {...}
15 }
```

Figure 2.3: C# code example.

Note: Object creation expression and initializers

The last language feature which will take part in the improved type inference are initializers used as a shortcut during an object instantiation. The most simple one is an object initializer that allows to assign values to the object's fields in a pleasant way instead of assigning them separately after the initialization. The second type of initializers regard arrays and collections. Array initializers are used to create fixed arrays with predefined content as we can see on the first line of our example 2.3 where we create an array of `System.Object` with two items. Under the hood, each of the items in the initializer is assigned to the corresponding index of the array after the array creation. Collection initializers are similar to an array initializer defined on collections which are determined by implementing `ICollection<T>` interface. One of the interface's declaring methods is `void Add<T>(T)` with adding semantic. Each type implementing this interface is allowed to use an initializer list in the same manner as an array initializer. It's just a sugar code hiding to call the 'Add' method for each item in the initializer list. The last type of initializer uses indexer to store referred values on predefined positions.

2.1.3 Type inference

Note: Introduce kinds of type inferences

Note: Method type inference

C# type inference occurs in many contexts. However, we mention only these related to our improvement described in the following sections. One of the most simple occurrences regards the `var` keyword, used in the variable declaration, as seen on the first line in code 2.3. It lets the compiler decide the type of variable based on the type of initializing value, which implies that we can't use the keyword in declarations without initializing the value.

The most interesting and complex context is the method type inference used during generic method call binding when type arguments are not given. We

`{Parameter}.isValParam`	Checks if the parameter is passed by value.
`{Parameter}.isRefParam`	Checks if the parameter is passed by reference.
`{Parameter}.isOutParam`	Checks if the parameter has `out` modifier.
`{Parameter}.isInParam`	Checks if the parameter has `in` modifier.
`{Argument}.isInArg`	Checks if the argument has `in` modifier.
`{Type}.outTypes`	Returns *Output* types of type.
`{Type}.inTypes`	Returns *Input* types of type.
`{Type} isLike '{Pattern}'`	Checks if the type matches the pattern.
`{Type}.isDelegateOrExprTreeType`	Checks if the type is Delegate or Expression Tree type.

Table 2.1: Description of used properties.

can see a situation when the method type inference deduces `System.String`, `System.Int32` and `System.Int32` as type arguments of the `MyMethod` method on line 4 in our code example 2.3. We can notice several tasks that the type inference has to be capable of. Regarding the `T1` type parameter, the inference has to find a common type between the first and the second type parameters. Regarding the `T2` type parameter, the type inference has to go into type arguments of the generic type of parameter and the argument, check if the types are compatible, and then match the type parameter against the type argument of the third parameter. The most challenging is Lamdas, whose return type has to be inferred after all lambda argument types are inferred.

As we can see, the method type inference is a complex process containing many steps. Since one of our improvements is adjusting the algorithm, we present its complete description. The algorithm is divided into four sections to explain its functionality better. Before we show the algorithm, we have to present definitions that are used by the algorithm referred in the C# specification [13].

Definition 1 (Fixed type variables, bounds)**.** *We call inferred type parameters* type variables *which are at the beginning of the algorithm unknown,* unfixed*. During the algorithm, they start to be restricted by sets of type* bounds*. The type variable becomes* fixed *when the its actual type is determined using its* bounds*.*

Definition 2 (Method group)**.** *A* method group *is a set of overloaded methods resulting from a member lookup.*

Definition 3 (Input/Output types)**.** *If `E` is a method group or anonymous function and `T` is a delegate or expression tree type, then return type of `T` is an* output type *of `E`. If `E` is a method group or implicitly typed anonymous function, then all the parameter types of `T` are* input types *of `E`.*

Definition 4 (Dependence)**.** *An unfixed type variable X_i* depends directly on *an unfixed type variable X_e if for some argument `E` X_e occurs in an input type of `E` and X_i occurs in an output type of `E`. X_i* depends on *X_e is the transitive but not reflexive closure of* depends directly on*.*

Note: Algorithm description

The method type inference process starts with receiving arguments of a method call and the method's signature, which type parameters have to be deduced, as we can see in the first Figure 2.4. Our representation of the algorithm uses many helper functions which description is given in Table 2.1. The algorithm

has two phases where the first phase initializes initial bounds' sets of type variables(inferred type arguments), and the second phase repeats until all type variables are fixed or fails if there is insufficient information to deduce it. Each type variable has three types of bounds. The exact bound consists of types, which have to be identical to the type variable, meaning that they can be converted to each other. The lower bound contains types that have to be convertible to the type variable, and the upper bound is opposite to it.

```
1  Input: method call M(E_1,...E_x) and
2         its signature T_e M<X_1,...,X_n>(T_1 p_1,...,T_x p_x)
3  Output: inferred X_1,...X_n
4  B_lower = B_upper = B_exact = F = []
5  FirstPhase()
6  SecondPhase()
7  fn FirstPhase():
8    E.foreach(e →
9      if (e.isAnonymousFunc)
10       InferExplicitParamterType(e, T[e.idx])
11     elif (e.getType() is Type u)
12       switch (u) {
13         p[e.idx].isValParam → InferLowerBound(u, T[e.idx])
14         p[e.idx].isRefParam || p[e.idx].isOutParam →
15           InferExact(u, T[e.idx])
16         p[e.idx].isInParam && e.isInArg →
17           InferLowerBound(u, T[e.idx])
18       }
19   )
20 fn SecondPhase():
21   while (true):
22     X_indep = X.filter(x →
23       F[x.idx] == null && X.any(x → dependsOn(x, y)))
24     X_dep = X.filter(x →
25       F[x.idx] == null && X.any(y →
26         dependsOn(y, x) && (B_lower+B_upper+B_exact).isNotEmpty))
27     switch {
28       X_indep.isNotEmpty → X_indep.foreach(x → Fix(x))
29       X_dep.isNotEmpty && X_indep.isEmpty → X_dep.foreach(x → Fix(x))
30       (X_indep+X_dep).isEmpty →
31         return if (F.any(x → x == null)) Fail() else Success(F)
32       default → E.filter(e →
33           X.any(x →
34             F[x.idx] == null && T[e.idx].outTypes.contains(x))
35           && !X.any(x →
36             F[x.idx] == null && T[e.idx].inTypes.contains(x))
37         ).foreach(e → InferOutputType(e, T[e.idx]))
38     }
```

Figure 2.4: Phases of Method Type Inference

Note: The first phase description

`FirstPhase()` iterates over provided arguments and chooses the right set where to add the type of the argument by calling helper functions based on several conditions.

Note: The second phase description

`SecondPhase()` happens iteratively, respecting *depends on* relation. Each iteration has two goals. The first one is the fixation of at least one type variable. If there is no type variable to fix because either all type variables are fixed or there are no other type bounds which could be used for type variable deduction, the phases and algorithm ends. The sets X_{indep} and X_{dep} refer to type variables, which can be fixed in the current iteration. Line 31 contains a case ending the algorithm when all type variables are fixed, or there is no way to infer the next ones. The second goal checks for output types, dependent on input types where the last unfixed types were fixed in this iteration, and infers them. We can see respecting the order of inferring the return type of anonymous functions at line 32, where the return type of lambdas is inferred if all type variables contained in the parameter list are fixed.

Note: Describe infer output type

Figure 2.5 contains definitions of three inferences used in the first and second phases. `InferOutputType()` infers return types of parameter types, which are delegates or expression trees. We can notice two situations where the first one takes care of arguments, which are Lambdas, and the second deals with method groups with possible type arguments. We can see two undefined methods used to achieve the inference above. `InferReturnType()` utilizes already known types of lambdas argument to infer its return type. `OverloadResolution()` tries resolving a method group to one method in order to infer the return type of that method.

Note: Describe infer explicit parameter type

`ExplicitParamterType()` method is used when an argument is explicitly typed anonymous function where the given parameter types are used to get information about type variables contained in a parameter of delegate or expression tree type.

Note: Describe infer exact

`InferExact()` is one of the three inferences which adds new bounds to type variables' bound sets. Basically, it finds arguments' types with corresponding parameter types and adds these types to the bounds when they match type variables as we can see at line 19 where we choose the bound based on several kinds of match.

Note: Describe infer lower, upper

Figure 2.6 shows remaining two inferences, adding types to lower and upper bounds. We can see various conditions testing corresponding types between arguments and parameters. An interesting fact about adding new bounds is that there is no need to check possible contradictions, which are checked in type variable fixation. The second point that will be important for us in the following sections is the absence of unfixed type variables in bound sets, making the algorithm easier for implementation.

Note: Fix

The last part of this algorithm is type variable fixation, shown in Figure

```
1  fn InferOutputType(Argument E, Type T):
2    switch(E) {
3      E.isAnonymousFunc && T.isDelegateOrExprTreeType →
4        InferLower(InferReturnType(E), T.returnType)
5      E.isMethodGroup && T.isDelegateOrExprTreeType → {
6          E_resolved= OverloadResolution(E, T.parameterTypes)
7          if (E_resolved.size == 1)
8            InferLower(E_resolved[0].returnType, T.returnType)
9        }
10     E.isExpression && E.getType() is Type u → InferLower(u, T)
11   }
12 fn InferExplicitParameterType(Argument E, Type T):
13   if (E.isExplicitTypedAnonymousFunc && T.isDelegateOrExprTreeType
14     && E.paramTypes.size == T.paramTypes.size)
15       E.paramTypes.zip(T.paramTypes)
16                   .foreach((e, t) → InferExact(e, t))
17 fn InferExact(Type U, Type V):
18   if (X.any(x → V == x && F[x.idx] == null)) B_exact[i].add(U)
19   switch(V) {
20     V isLike 'V1[..]' && U isLike 'U1[..]' && V.rank == U.rank →
21       InferExact(U1, V1)
22     V isLike 'V1?' && U isLike 'U1' → InferExact(V1, U1)
23     V isLike 'C<V1, ..., Ve>' && U isLike 'C<U1, ..., Ue>' →
24       V.typeArgs.zip(U.typeArgs)
25                 .foreach((v, u) → InferExact(v, u))
26   }
```

Figure 2.5: *Output type inference, Explicit parameter type inference, Exact inference*

```
fn InferLower(Type U, Type V):
  if (X.any(x → V == x && F[x.idx] == null)) B_lower[i].add(U)
  switch {
    V isLike 'V1?' && U isLike 'U1?' → InferLower(V1, U1)
    V isLike 'V1[..]' && U isLike 'U1[..]' && V.rank == U.rank →
      (!U1.isRefType) ? InferExact(U1, V1) : InferLower(U1, V1)
    V isLike 'IEnumerable<V1>' || 'ICollection<V1>' || 'IList<V1>'
      || 'IReadOnlyList<V1>' || 'IReadOnlyCollection<V1>'
      && U isLike 'U1[]' →
        (!U1.isRefType) ? InferExact(U1, V1) : InferLower(U1, V1)
    V isLike 'C<V1, ..., Ve>' →
      temp = ([U] + U.inheritedTypes + U.implementedInterfaces)
        .filter(x → x isLike 'C<U1, ..., Ue>')
      if (temp.size == 1) →
        V.typeArgs.zip(temp[0]).foreach((v, u) →
          switch(v) {
            v.isCovariant → InferLower(v, u)
            v.isContravariant → InferUpper(v, u)
            v.isInvariant → InferExact(v, u)
          }
        )
  }
fn InferUpper(Type U, Type V):
  if (X.any(x → V == x && F[x.idx] == null)) B_upper[i].add(U)
  switch {
    V isLike 'V1?' && U isLike 'U1?' →
      (!U1.isRefType) ? InferExact(U1, V1) : InferUpper(U1, V1)
    V isLike 'V1[..]' && U isLike 'U1[..]' && V.rank == U.rank →
      (!U1.isRefType) ? InferExact(U1, V1) : InferUpper(U1, V1)
    U isLike 'IEnumerable<U1>' || 'ICollection<U1>' || 'IList<U1>'
      || 'IReadOnlyList<U1>' || 'IReadOnlyCollection<U1>'
      && V isLike 'V1[]' →
        (!U1.isRefType) ? InferExact(U1, V1) : InferLower(U1, V1)
    U isLike 'C<U1, ..., Ue>' →
      temp = ([V] + V.inheritedTypes + V.implementedInterfaces)
        .filter(x → x isLike 'C<V1, ..., Ve>')
      if (temp.size == 1) →
        U.typeArgs.zip(temp[0]).foreach((u, v) →
          switch(u) {
            u.isCovariant → InferUpper(u, v)
            u.isContravariant → InferLower(u, v)
            u.isInvariant → InferExact(u, v)
          }
        )
  }
```

Figure 2.6: *Upper-bound inference*, *Lower-bound inference*

```
fn Fix(TypeParameter x):
  U_candidates = B_lower[x.idx] + B_upper[x.idx] + B_exact[x.idx]
  B_exact[x.idx].foreach{b →
    U_candidates.removeAll(u -> !b.isIdenticalTo(u))}
  B_lower[x.idx].foreach{b →
    U_candidates.removeAll(u -> !hasImplicitConversion(b, u))}
  B_upper[x.idx].foreach{b →
    U_candidates.removeAll(u -> !hasImplicitConversion(u, b))}
  temp = U_candidates.filter(x → U_candidates.all(y →
    hasImplicitConversion(y, x)))
  if (temp.size == 1) F[i] = temp[0] else Fail()
```

Figure 2.7: Fixing of type variables

2.7. At the beginning, a set of candidates for the type variable is constructed by collecting all its bounds. Then, we go through each bounds and remove the candidates which do not satisfy the bound's restriction. If there is more than one candidate left, we try to find a unique type that is identical to all left candidates. The fixation is successful if the candidate is found. The type variable is fixed to that type.

Note: Array type inference

The third type inference happens in array initializers when the type of the array should be deduced from the initializer list. We can see an example of a situation when the type inference is used for determining `myArray` type on the first line in our code example 2.3. The most specialized common type is just adjusted already mentioned type inference algorithm where there is just one type parameter, and all initializer items are lower bounds of that type variable.

Note: Target-typed inference

The last kind of type inference, which we mention, regards inference based on target type. An example of these situations can be seen on the second line of our example 2.3 where we use target-typed `new()` operator, allowing us to skip creating type, which is provided by the target type, variable type, in this case.

2.2 Roslyn

Note: Intro

The implementation of C# type inference can be found in the Roslyn compiler, as open-source C# and VisualBasic compiler developed at the GitHub repository. In this section, we present Roslyn's architecture to better understand the context and restrictions that we must consider to plug the improved type inference into the compiler.

Note: Overview of compilation pipeline, the first phase

The compilation pipeline, described in Microsoft documentation [18], starts with loading the `.csproj` file with related C# sources (`.cs`) and referenced libraries (`.dll`), as we can see in Figure 2.8. C# sources are passed to the lexer, creating tokens used by the parser forming Abstract Syntax Tree (AST). Constructing AST is the first phase (green color boxes of Figure 2.8) of the compiler

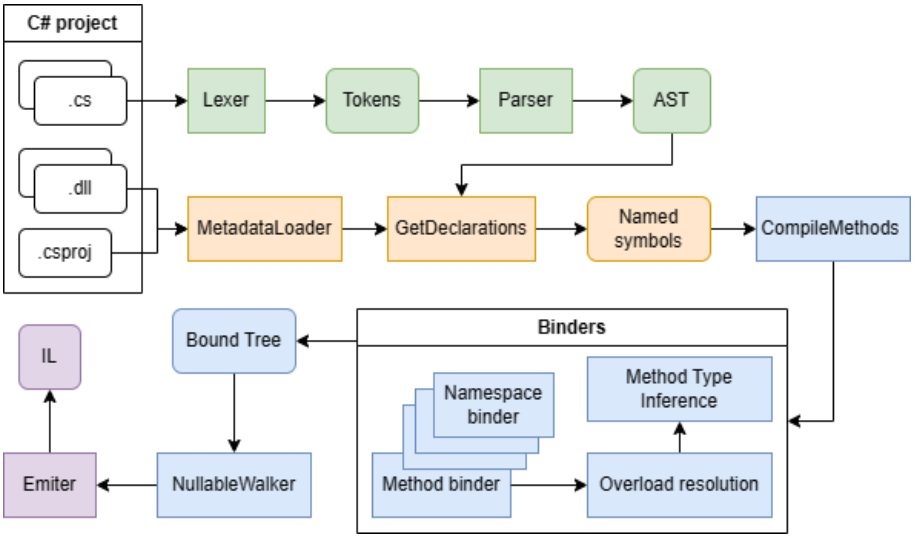

Figure 2.8: Roslyn architecture

checking syntax of C# sources.

Note: Describe the second phase

The second phase, marked by orange, forms *Named symbols* exposed by public API representing defined namespaces, classes, methods, etc., in the C# project. The declarations are received from C# sources by traversing AST and seeking for the particular syntax. Stored in `.dll` format, Libraries are parsed by `MetadataLoader`, creating the same Named symbols as those received from C# sources.

Note: Describe the third phase

The third phase, also called the binding phase, matches identifiers in the code with received Named symbols from the previous phase. Because the processing of a method body is not dependent on other method bodies since the code only uses already known declarations, Roslyn makes this phase concurrent. The result of the phase is a *Bound tree* where all identifiers refer to the Named symbols. A method binding itself is a complicated procedure consisting of many subtasks such as *Overload resolution* or mentioned Method type inference, which algorithm we described in detail in the previous section.

Note: Binder

The binding is divided into a chain of binders, taking care of smaller code scopes. One purpose of the binders is the ability to resolve an identifier to the Named symbol if the referred symbol lies in their scope. If they can't find the symbol, they ask the preceding binder. This process is called *LookUp*, determining which Named symbols are accessible in the current position. Examples of binders are `NamespaceBinder` resolving defined top-level entities in the namespace scope, `ClassBinder` resolving defined class members, or `MethodBinder` binding method bodies. The last mentioned binder sequentially iterates body statements and matches identifiers with their declarations. Statement and expression binding are important steps that are related to type inference. The first observation is that statement binding doesn't involve binding of the following statements, which can be referred to as backward binding. The consequence is that C# is not able to infer types in the backward direction. An example can be the usage of the `var` keyword in variable declarations, which has to be used always with initializing

value. If C# would allow backward binding, we could initialize the variable later in one of the following statements.

Note: OverloadResolution, MethodTypeInferrer

The preceding step before Method type inference is Overload resolution, part of `MethodCallExpression` or `ObjectCreationExpression` binding. As we mentioned previously, method overloading allows to define multiple methods with the same name differing in parameters. So, when deciding which method should be called, we have to resolve the right version of the method by following language rules for method resolution. This step involves binding the method call arguments first and then deciding which parameter list of the method group fits the argument list the best. If the method group is generic and the expression doesn't specify any type arguments, Method type inference is invoked to determine the type arguments of the method before the selection of the best candidate for the call. When the right overload with inferred type arguments is chosen, unbinded method arguments requiring target type (for example already mentioned target-typed `new()` operator) are binded using corresponding parameter type.

Note: NullableWalker

Method type inference can occur for the second time if we turn on the nullability analysis mentioned previously. Nullability analysis is a kind of Flow analysis that uses a Bound tree to check and rewrite already created bound nodes according to nullability. Because overloading and method type inference are nullable-sensitive, the whole binding process is repeated, respecting the nullability and reusing results from the previous binding. The required changes are stored during the analysis, and the Bound tree is rewritten by the changes at the end of the analysis.

2.3 Hindley-Millner type inference

Note: Intro

C# method type inference is a restricted Hindley-Millner type inference in order to work in its C# type system. Since type inferences in other languages like Rust or Haskell are based on the same principle, we present a high-level overview of Hindley-Millner type inference and its type system to reason about possible extensions of current C# type inference.

Note: Hindley-Millner type system

Hindley-Millner type system [21] is a type system for Lambda calculus capable of generic functions and types. We start with describing a set of expressions, which will be a target of type inference described in the video series [4]. Expression is either a variable (2.1), a function application (2.2), a lambda function (2.3), or a *let-in* clause (2.4).

$$
\begin{aligned}
e &= x && (2.1)\\
&\mid e_1 e_2 && (2.2)\\
&\mid \lambda x \rightarrow e && (2.3)\\
&\mid \textbf{let } x = e_1 \textbf{ in } e_2 && (2.4)
\end{aligned}
$$

The above-mentioned expressions have one of two different types. *Mono* type is presented either as a type variable(2.5) or as a function application(2.6) where C is an item from an arbitrary set of functions containing at least $\rightarrow$ symbol taking two type parameters which represents a lambda function type. The second group is *Poly* types, which assemble from a type possible preceding $\forall$ operator 2.8, bounding its type variables.

$$mono\ \tau = \alpha \tag{2.5}$$
$$\mid C\ \tau_1, ..., \tau_n \tag{2.6}$$
$$poly\ \sigma = \tau \tag{2.7}$$
$$\mid \forall\alpha\ .\ \sigma \tag{2.8}$$

To be able to reason about more complex type expressions, a context(Γ) gives us assumptions of a current scope where we evaluate the type of expression. It contains pairs of expressions and their types from which we can make typing judgments (using $\vdash$ operator).

Note: Set of rules

The H-M deduction system gives us the following inference rules, allowing us to deduce the type of an expression based on the assumption given in the context. The syntax of the rule corresponds with what we can judge (the right side of $\vdash$) from the left side of $\vdash$ below the line based on assumptions given above the line. The rules can be divided into two kinds. The first four rules give us a manual on what types we can expect by applying the mentioned expressions of Lambda calculus. The two last rules allow us to specify Poly types to Mono types and vice-versa.

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}[Var]$$
$$\frac{\Gamma \vdash e_0 : \tau_a \rightarrow \tau_b \quad \Gamma \vdash e_1 : \tau_a}{\Gamma \vdash e_0 e_1 : \tau_b}[App]$$
$$\frac{\Gamma, x : \tau_a \vdash e : \tau_b}{\Gamma \vdash \lambda x \rightarrow e : \tau_a \rightarrow \tau_b}[Abs]$$
$$\frac{\Gamma \rightarrow e_0 : \sigma \quad \Gamma, x : \sigma \vdash e_1 : \tau}{\Gamma \vdash \ \textbf{let}\ x = e_0\ \textbf{in}\ e_1 : \tau}[Let]$$
$$\frac{\Gamma \vdash e : \sigma_a \quad \sigma_a \sqsubseteq \sigma_b}{\Gamma \vdash e : \sigma_b}[Inst]$$
$$\frac{\Gamma \vdash e : \sigma \quad \alpha \notin Free(\Gamma)}{\Gamma \vdash e : \forall\alpha\ .\ \sigma}[Gen]$$

Note: Describe type inference

With the definitions given above, we can start to talk about H-M type inference, which is able to find the type of every expression of a completely untyped program. There exist several algorithms for inferring most general type of the expression. We show the W algorithm in Figure 2.9 since it is closely related to C# and Rust type inference. Inputs are the context Γ and an expression which type we want to infer. The process consists of systematic traversing the expression from bottom to top and deducing the type of sub-expressions following the

```
1  fn Infer(Γ, expr):
2    switch(expr):
3      expr isLike 'x' → return ({}, Instantiate(expr))
4      expr isLike 'λx → e' →
5        β = NewVar()
6        (S1, τ1) = Infer(Γ + x: β, e)
7        return (S1, S1β → τ1)
8      expr isLike 'e1e2' →
9        (S1, τ1) = Infer(Γ, e1)
10       (S2, τ2) = Infer(S1Γ, e2)
11       β = NewVar()
12       (S3, τ1) = Unify(S2τ1, τ2 → β)
13       return (S3S2S1, S3β)
14     expr isLike 'let x = e1 in e2' →
15       (S1, τ1) = Infer(Γ, e1)
16       (S2, τ2) = Infer(Γ + x: Generalize(S1Γ, τ1), e2)
17       return (S2S1, τ2)
```

Figure 2.9: W algorithm

mentioned rules. The algorithm uses the `Instantiate` method to replace quantified type variables in the expression with new type variables, the `Generelize` method to replace free type variables in the expression with quantified type variables, and the `Unify` method, also known as *Unification* in Logic. Unification is an algorithm finding a substitution whose application on the unifying types makes them identical. Outputs of this algorithm are the inferred type with a substitution used for the algorithm's internal state.

Note: Mention relation to Method type inference

Relations between this algorithm and C#, Rust type inference will be discussed in detail later, although we can notice that the unification part of this algorithm is the same as Method type inference mentioned in the C# section where the substitution represents inferred type arguments of the method which parameters were unified with matching arguments.

Note: Restriction and possible extensions - subtyping, overloading

H-M type system, as we presented, doesn't allow subtyping known from Rust or overloading known from C#.

A basic principle of extending H-M type inference by subtyping is described in Parreaux's work [24], where instead of accumulating type equivalent constraints, we accumulate and propagate subtyping constraints. These subtyping constraints assemble a set of types, which have to be inherited by the constrained type variable, or the variable has to inherit them.

Note: Restriction and possible extensions - overloading

Extending H-M type inference by supporting overloading mentions Andreas Stadelmeier and Martin Plumicke's work [23]. An important thought behind this paper is to accumulate two types of type variable constraint sets. Constraints observed from a method call are added into one *AND-set*. When the method call has multiple overloads, the AND-sets are added to the *OR-set*. After accumu-

```
fn main() {
    let mut things = vec![];
    things.push("thing");
}
```

Figure 2.10: Rust code example taken from [20].

lating these sets, all combinations of items in OR-sets are generated and solved by type inference. As we can see, for each method overload participating in type inference, we have to make an alternative type inference containing constraints obtained only from the overloaded method, excluding other overloads. This algorithm can be improved by excluding overloads that can't be used in the method call to save the branching. However, in the worst case, it still takes exponential time to infer types.

2.4 Rust type inference

Note: Intro

Rust is a strongly typed programming language developed by Mozzila and an open community created for performance and memory safety without garbage collection. Besides its specific features like traits or variable regions, it also has advanced type inference, which we now describe in a high-level perspective to get inspiration for our proposed improvement.

Note: Type inference

We use code example 2.10 to show the significant difference between C# method type inference and global Rust type inference. We can see that it can infer a type argument of generic type `vec<T>`, referring to a resizeable array, despite of the type information determining a type of the type argument is given in the later statement.

This is possible thanking to type inference context, which is shared across multiple statements. We show a basic principle of the context in Figure 2.6. It starts with an empty context. As the compiler traverses a method body, it adds new type variables that have to be solved and constrains them by the types which they are interacting with. In the figure, it constrains a type variable T1, required to infer the type variable of variable a by an initializing type. However, the initializing type can't be fully resolved because the constraint contains an unbound type variable. So, it passes the context to binding the next statement, where it collects another constraint about the type argument of the initializing value. The collection of the constraints is similar to the Unification seen in the previous sections, where we extract the required bounds of unbound type variables by finding substitutions for type variables in order to make matching types containing the type variables equal. When there is enough information to resolve type variables, they are resolved by finding an appropriate type for the type variable with respect to collected types.

Note: Properties of the type inference. More source of type info, Probing, Snapshots, No overloading

The mentioned sharing of context enables type inference to be a back forward, meaning that based on future type information, it is possible to infer already col-

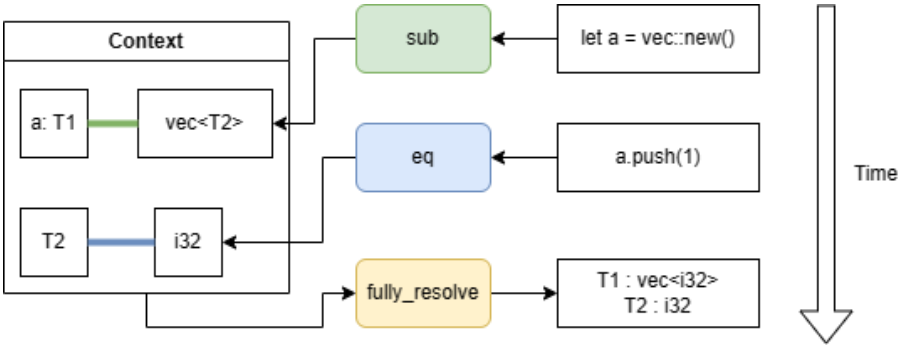

Figure 2.11: Rust type inference

lected unbounded type variables. Besides sharing the context, there are other inference features that are missing in C# type inference and would be valuable. The first of them is type inference in object creation expressions, which doesn't exist in C#. The next regards collecting type constraints, which are obtained from a wider context than C# uses. For example, If a generic method containing a type variable in the return type is used as an assignment of an already typed variable, the type variable is constrained by the type of the target. Other features regard the inner implementation of type inference, which offers probing to constrain a type variable without influencing the context. There is a possibility of a snapshot that records all changes and can be used for backtracking and finding the right inferred type arguments. Although Rust type inference is more advanced in comparison with C#, we have to still consider language differences making type inference computation cost and difficulty relative to their features. As an example we can mention overloading causing already mentioned computation cost. Since Rust doesn't have overloading, the type inference can be more powerful without significant slowdown, which is not the case of C#.

2.5 Github issues

Note: Intro

The last subsection mentions the C# developer's environment, where everybody is free to share his/her thoughts about possible improvements or issues with C# language or Roslyn compiler. We base our solution on issues mentioned in this section and recommended future possible features by C# developers in the already mentioned the C# language and Roslyn repository.

Note: Proposal

The process of designing a new language feature starts with publishing an idea into discussions [14], where the C# community can comment on it. The idea contains a brief description of the feature, motivation, and design. Besides the idea, a new language feature requires a proposal, initially published as an issue, describing the feature in a way that can be later reviewed by the LDM committee. If the proposal sufficiently merits the discussions, it can be marked as a *champion* by a member of LDM for being discussed further by the team. There are several milestones in which the proposal can be. The most important for us is *AnyTime*, meaning that the proposal is not actively worked on and is open to the community to collaborate on it. At the time of writing, a

```
var temp = new A();

class A<T = int> {}
```

Figure 2.12: Default type parameters.

```
var temp = new StringDictionary<int>();

using StringDictionary<TValue> = Dictionary<String, TValue>;
```

Figure 2.13: Generic aliases.

member of LDM recommended championed issue [11] to be investigated since it contains many related discussions with proposed changes but still doesn't have a required proposal. When a proposal has sufficient quality to be discussed by LDM, a member invites the proposer to make a *Pull Request* where further collaboration continues. If LDM accepts the proposal, it is added to the *proposals* folder in the repository for being added into the C# specification, and its future implementation (in Roslyn) will be shipped with the next C# version.

Note: Discussions

Regarding the mentioned issue, we present related discussions directly or indirectly referred from it.

Note: Discussion - Default type parameters

The first discussion [2] mentions *Default type parameters* introducing default type arguments, which are used when explicit type arguments are not used. Figure 2.12 shows a potential design of this feature where construing generic type `A` doesn't need a type argument since it uses `int` type as a default value.

Note: Discussion - Generic aliases

The next discussion [3] mentions *Generic aliases* allowing to specify default values similar to the goal of the previous discussion by defining an alias to that type with option generic parameters. We can see an example of usage in Figure2.13, where we predefine the first type argument of the `Dictionary` class to be the `string` type, which simplifies the usage of that type in scenarios where we often use dictionaries with keys of the `string` type.

Note: Discussion - Named type arguments

Discussion [5] mentions *Named type parameters*, which are similar to named parameters of methods. The basic thought of this idea is being able to specify a type parameter for which we provide a type argument by name. In Figure 2.14, we can see a generic method `F` with two type parameters. With the current type inference, we have to always specify type arguments in method calls of `F`. We can tell the compiler specific type parameters for which we provide type arguments, `U` in this case, using the named type arguments and letting the compiler infer the rest of the type parameters.

Note: Discussion - Using char as inferred type argument

Comments of the mentioned championed issue [11] propose several keywords that can be used in a type argument list for skipping type arguments, which can be inferred by the compiler and just providing the remaining ones. In example 2.15, we can see the `var` keyword for skipping the first type argument since it can

```
var x = F<U:short>(1);

U F<T, U>(T t) { ... }
```

Figure 2.14: Named type parameters.

```
Foo<var, int>("string");

TResult Foo<T, TResult>(T p1){ ... }
```

Figure 2.15: Using char as inferred type argument.

be inferred from the argument list, and we just specify the second type argument, which can't be inferred by the compiler. The comments propose other options for keywords like underscore or whitespaces.

Note: Discussion - Inference based on target

Discussion [7] proposes *Target-typed inference*, where type inference uses type information of the target assigned by the return value. We can see the usage in Figure 2.16, where type inference determines that the return type has to be `int` type and uses that to deduce the type argument `T`.

Note: Discussion - Type inference based on type constraints

The next idea of improving type inference is given by discussion [8] , where type inference utilizes type information obtained from type constraints. A simple example of that can be seen in Figure 2.17, where `T1` can be deduced by using `T1`'s constraint and inferred type of `T2` forming inferred type `List<int>`.

Note: Discussion - Type inference of method return type

Discussion [9] mentions type inference of method return type known from Kotlin language. We can see the usage in the following Figure 2.18, where the return type of method `Add` is inferred to be `int` based on the type of the return expression.

Note: Discussion - Specifying type arguments in method calls (Realocation)

Issue [6] proposes a way to compact type argument lists of identifiers containing inner identifiers with argument lists. We can see an idea demonstrated in example 2.19, where the argument list of `A<T1>` type and the `Foo<T2>` method are merged, and the type arguments are split by a semicolon.

Note: Discussion - Constructor type inference

The last discussion [1], which we mention here, regards *Constructor type infer-*

```
object row = ...
int id = row.Field("id")

static class ObjectEx {
    T Field<T>(this object target, string fieldName)
    { ... }
}
```

Figure 2.16: Target-typed inference.

```
var temp = Foo(1);

T1 Foo<T1,T2>(T2 item) where T1 : List<T2> {}
```

Figure 2.17: Type inference based on type constraints.

```
public static Add(int x, int y ) => x + y;
```

Figure 2.18: Type inference of method return type.

ence enabling type inference for object creation expressions. The type inference can be seen in Figure 2.20, where the `T` type parameter of the `C<T>` generic type can be deduced by using type information from its constructor.

```
A.Foo<int;string>();

static class A<T1> {
    public static void Foo<T2>(){}
}
```

Figure 2.19: Specifying type arguments in method calls (Realocation).

```
var temp = new C<_>(1); // T = int

class C<T> { public C(T p1) {}}
```

Figure 2.20: Constructor type inference.

3. Problem analysis

TODO: Describe outputs of this work(Proposal and prototype). Why these outputs are necessary.

TODO: Describe the set of related issues.

TODO: Describe the selection and scope of this work based on the issues and other factors.

TODO: Describe problems of C# lang architecture which prohibits some advanced aspects of type inference.

TODO: Describe goals of the work and explain benefits of proposed changes.

4. Solution

TODO: Describe process of making proposal and the prototype.

TODO: Describe partial method type inference.

TODO: Describe constructor type inference.

TODO: Describe generic adjusted algorithm for type inference.

TODO: Describe decisions of proposed change design.

TODO: Describe changed parts of C# standard.

5. Evaluation

TODO: Describe achieved type inference. Mention interesting capabilities.

TODO: Note about the performance.

TODO: Links to csharplang discussions.

6. Future improvements

TODO: Mention next steps which can be done.

TODO: Discuss which steps would not be the right way(used observed difficulties).

Conclusion

TODO: Describe issue selection.

TODO: Describe proposed changes in the lang.

TODO: Describe the prototype and proposal.

TODO: Mention csharplang discussions.

TODO: Mention observed future improvements.

Bibliography

[1] Constructor type inference. https://github.com/dotnet/csharplang/discussions/281, . [Online; accessed 2023-11-4].

[2] Default type parameters. https://github.com/dotnet/csharplang/discussions/278, . [Online; accessed 2023-11-4].

[3] Generic aliases. https://github.com/dotnet/csharplang/issues/1239, . [Online; accessed 2023-11-4].

[4] Video series about Hindley-Millner type inference. https://www.youtube.com/@adam-jones/videos, . [Online; accessed 2023-10-21].

[5] Named type parameters. https://github.com/dotnet/csharplang/discussions/280, . [Online; accessed 2023-11-4].

[6] Specifying type arguments in method calls (reallocation). https://github.com/dotnet/roslyn/issues/8214, . [Online; accessed 2023-11-4].

[7] Return type inference. https://github.com/dotnet/csharplang/discussions/92, . [Online; accessed 2023-11-4].

[8] Type inference based on type constraints. https://github.com/dotnet/roslyn/issues/5023, . [Online; accessed 2023-11-4].

[9] Type inference of method return type. https://github.com/dotnet/csharplang/discussions/6452, . [Online; accessed 2023-11-4].

[10] C# type system. https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/, . [Online; accessed 2023-09-22].

[11] C# proposed champion. https://github.com/dotnet/csharplang/issues/1349, . [Online; accessed 2023-11-2].

[12] C# version history. https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-version-history, . [Online; accessed 2023-10-8].

[13] C# type inference algorithm. https://github.com/dotnet/csharpstandard/blob/draft-v8/standard/expressions.md, . [Online; accessed 2023-10-14].

[14] C# language discussions. https://github.com/dotnet/csharplang/discussions, . [Online; accessed 2023-11-14].

[15] csharplang repository. https://github.com/dotnet/csharplang, . [Online; accessed 2023-09-22].

[16] C# specification. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/readme, . [Online; accessed 2023-09-22].

[17] Proposal template. https://github.com/dotnet/csharplang/blob/main/proposals/proposal-template.md, . [Online; accessed 2023-09-30].

[18] Roslyn architecture. https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/compiler-api-model, . [Online; accessed 2023-10-21].

[19] Roslyn repository. https://github.com/dotnet/roslyn, . [Online; accessed 2023-09-22].

[20] Rust type inference. https://doc.rust-lang.org/rust-by-example/types/inference.html, . [Online; accessed 2023-09-22].

[21] Hidley-milner type system. https://en.wikipedia.org/wiki/Hindley%E2%80%93Milner_type_system, . [Online; accessed 2023-09-22].

[22] Hidley-Milner type inference. https://www.youtube.com/watch?v=B39eBvapmHY, . [Online; accessed 2023-09-22].

[23] Andreas Stadelmeier and Martin Plumicke. Adding overloading to java type inference.

[24] Lionel Parreaux. The simple essence of algebraic subtyping: Principal type inference with subtyping made easy, 2020. [25th ACM SIGPLAN International Conference on Functional Programming - ICFP 2020].

List of Figures

List of Tables

List of Abbreviations

LDM Language Design Meetings

CTS Common Type System

AST Abstract Syntax Tree

A. Attachments