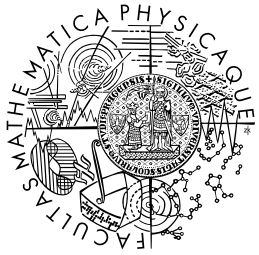

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

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

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 should have sufficient potential to be discussed by the Language Design Team (LDT) accepting new C# language features.

1.1 Improving C# type system

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. Figure 1.1 shows an usage of type annotations written in the C# programming language. The type declaration of the `people` variable guarantees that the following attempt to concatenate the `"Tom"` string to that variable will be reported as an error at compile time since the operation is not defined for a pair of the `List<string>` type and `string` type.

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

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

On the other hand, types can have long names, forcing the programmer to write more code to annotate the variable declaration or object creation, as we can see in the example. This disadvantage of strongly typed languages can be removed by *type inference* when a missing type annotation can be deduced using the context. Taking the example shown above, one of the `List<string>` type annotations could be removed since the type of `people` variable declaration can be deduced from its initializing value or the type of object creation can be deduced from the type of the assigning variable. There is an example of C# type inference in Figure 1.2, where the `var` keyword is used to trigger type inference determining a type of `people` variable to be the `List<string>` type.

```
var people = new List<string>();
```

Figure 1.2: Type inference in the C# programming language.

The power of type inference varies in strongly typed languages. An example of the difference can be seen in type arguments deduction of generic methods. In C#, a generic method is a method that is parametrized by types besides common parameters, as can be found in Figure 1.3. There is a generic method `GetField` enabling to return a value of `o`'s field with the `fieldName` name. The type of

```
T GetField<T>(object o, string fieldName) { ... }

object person = ...
string name = GetField<string>(person, "name");
```

Figure 1.3: C# Type inference of generic methods.

returned value is generic parameter `T` since it depends on the type of object's field. The `name` variable is initialized by using the method to retrieve a `person`'s name, which is supposed to be a string. There is a redundancy in that statement since the type argument list of the `GetField` method could be removed, and `T` could be deduced from the type of `name` variable, which has to be compatible with the return type. However, the current version of C# type inference fails to deduce it.

A similar concept of generic methods was introduced in the Rust [23] programming language, which belongs to strongly type languages too. Figure 1.4 shows a definition of the generic method `GetField`, which is equivalent to the C# method mentioned in the previous example. There is an equivalent initialization of `name` variable declaration starting with the `let` keyword, where Rust type inference deduces the type argument `T` to be the `&str` type utilizing the type information from the `name` variable declaration.

```
fn GetField<T>(o: &object, fieldName: &str) -> T { ... }

let person: &object = ...;
let name: &str = GetField(person, "name");
```

Figure 1.4: Rust Type inference of generic methods.

Although Rust is younger than C# and has a different type system, it managed to make type inference more powerful in the context of strongly typed languages to significantly save type annotations typing. The first goal of this thesis is to investigate if the similar level of type inference can be achieved in C# and improve C# type inference to be used in more scenarios saving type annotations typing.

The investigation explores type system requirements and type inference differences to achieve a desired level of type inference by formalizing Rust and C# type inference. These formalizations can be partially identified as a part of the existing Hindley-Millner [25] type inference formalization, which helps to reason about the inference in these languages. Traditional Hindley-Millner type inference is defined in the Hindley-Millner type system [24], 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 the differences, 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. Observing the influence of differences between these type systems on type inference will help to understand a limitation of possible type inference improvement in C#.

1.2 Implementation

The first part of the thesis explores limitations of C# type inference and proposes an improvement. The first goal of the second part tests the improvement by implementing it in an official C# compiler, Roslyn [19], which is an open-source project managed by Microsoft. The prototype is used to explore potential implementation issues which the improvement can cause, and that helps to adjust the improvement to be potentially enabled in the C# compiler.

Although the compiler is managed by the company, it has an open-source development, which makes contributions from interested people possible to be merged into the production. Although it is sufficient to make a *pull request* containing a fix for solving compiler issues to be merged, language design improvements, similar to what the thesis will propose, require a special process of validating the actual benefit. The process starts by proposing new C# features in public discussions of the C# language repository [19], where everyone can add his/her ideas or comment on others' ideas. It is preferred to use a predefined template [21] for describing the idea proposing the feature in order to make the idea more likely to be discussed by the team responsible for accepting new language features. The template includes motivation, detailed description, needed C# language specification [20] changes, and other possible alternatives. The second goal of this part is to create the improvement as the language proposal, which would be presented to the team in order to have the potential to be a part of the current C# language. The process of language proposal ends with LDT accepting or declining it.

1.3 Summary

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. Implement the prototype in Roslyn

G4. Create an proposal containing the improvement

2. C# programming language

Introduction 1.1 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. Since type inference is a complicated process touching many areas of the C# language, it firstly sorts these areas into separated groups described in necessary detail to understand all parts of the current type inference. These areas concern the C# type system, including generics and language constructs where the type inference occurs or interacts with.

2.1 Type system

C# data types are defined in the C# type system, which also defines relations between them. The most fundamental relation is type inheritance, where every type inherits another type, forming a tree with `System.Object` as a root node that doesn't inherit any type. Types are divided into value and reference types, shown in Figure 2.1, where an arrow means *is inherited by* relation. Value types consist of built-in numeric types referred to as *simple types*, and enumerations referred to as *enum types*, structures referred to as *struct types*, and nullable types. Compared to reference types, value types are implicitly sealed, meaning that they can't be inherited by other types. Reference types consist of interfaces, classes, arrays, and delegates. An interface introduces a new relation to the type system by defining a list of methods, called a contract, which has to be implemented by a type that implements the interface. The relation forms an acyclic graph, meaning a type can implement multiple interfaces, but the implementation relations can't form a cycle. Delegates represent typed pointers to methods describing its signature, including generic parameters, parameters, and a return type.

The type system implicitly allows to assign `null`, indicating an invalid value, to reference types. Since C# 2.0 [13], it allows to assign `null` value to nullable

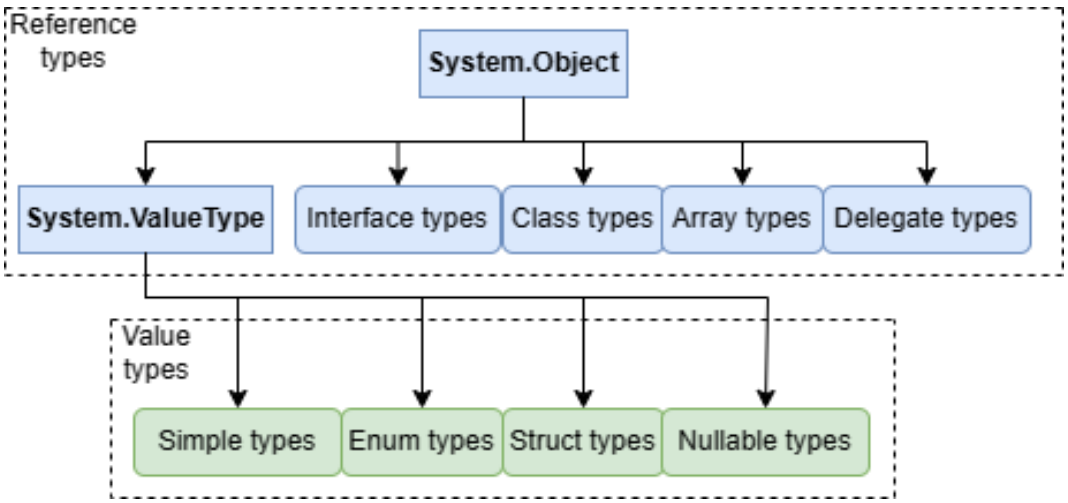

Figure 2.1: The C# data types schema adjusted from a C# blog[11].

types, which are equivalents of the rest of value types prohibiting it. Because assigning `null` value is referred to as a billion-dollar mistake, C# 8.0 [13], introduced optional settings warning about assigning null values and created nullable reference types, which, together with nullable types, explicitly allows `null` assignment as a way of interaction with legacy code not using the feature.

A big part of the type system is C# *generics*, allowing the parameterization of types and methods by arbitrary types. A specific generic method or type is *constructed* by providing required type arguments, where *construction* means replacing all occurrences of type parameters with the type arguments. Since type argument can be arbitrary type, the type parameter is considered to be the most general type in the type system, `System.Object`. Assuming additional API from the type parameter is achieved by restricting a set of types, which can replace the type parameter, enabling a specific interface of this set. The restriction is described by type constraints, which can be applied to type parameters. There are several kinds of constraints that can be combined together, forcing the type argument to fulfill all of them. Figure 2.2 shows only two of them, and the rest can be found in the C# documentation [15]. There is a definition of the `PrioritySorter` generic class with the `TItem` type parameter containing two constraints that the type argument has to hold. The `class` constraint allows only reference types. The `IPriorityGetter` constraint allows only types that implement the interface.

```
class PrioritySorter<TItem> where TItem : class, IPriorityGetter
{ ... }
```

Figure 2.2: C# type constraints.

Constructed methods and types are new entities that don't have any special relations between themselves implied from the construction. However, C# generic interfaces can utilize a concept of type variance to introduce additional relations between constructed types. Initially, type parameters are *invariant*, meaning an obligation to use the same type arguments as initially required. A type parameter can be specified to be *covariant*, by prepending the type parameter declaration with the `in` keyword, allowing to use more derived type than initially required. 0pposite *contravariance* uses the `out` keyword, allowing to use more general type than initially required.

The last relevant feature of the type system is method overloading, which allows definitions of multiple methods with the same name, return type, and count of type parameters having different types of parameters. Further chapters will mention the feature as an obstacle in designing efficient type inference.

2.2 Relevant constructs

Many unrelated C# constructs can use type inference or can influence the type inference algorithm. There are the most relevant whose internals are then considered in the following chapters regarding the design of the improvement.

Dynamic

Introduction 1.1 mentioned that strongly typed languages require knowing data types at compile time to prohibit incompatible operations on them. In the context of C#, data means values of expressions that are transformed by operations defined on their types. It turned out that operations on expressions of unknown type at compile time became crucial for interoperability with other dynamic-typed languages whose types of expressions are known at runtime. To make the interoperability easier, C# introduced the `dynamic` type that can be used as an ordinary type, which avoids the checks and causes *dynamic binding*. *Binding* is a process of resolving referenced operations based on the type and value of the expression. The majority of the C# binding happens statically at compile time. Expressions containing a value of the `dynamic` type are dynamic bound at runtime, bypassing the static binding of the compiler. This behavior can lead to possible bugs regarding invalid operations on the dynamic data types, which will be reported during runtime. Figure 2.3 shows a declaration of the `a` variable of the dynamic type. Dynamic binding occurs in the `a.Foo()` expression, where the `Foo()` operation is not checked during compilation. An error is reported at runtime when the actual type of the `a` variable is determined to be `string`, which doesn't define `Foo()` operation. Despite the dynamic binding, a compiler can still little check certain kinds of expressions containing values of dynamic types to reveal possible errors at compile time. An example of such checking is the `Bar()` method call, where the compiler can check the first argument, whose type is known at compile time as the type of the parameter. An appropriate error occurs during the compilation because string value has the `string` type, and it is passed as the `p1` parameter, which has the `int` type.

```
dynamic a = "string";
a.Foo();
Bar("string", 1, a); \\ Compilation error reported

void Bar<T>(int p1, T p2, long p3) {...}
```

Figure 2.3: C# dynamic type.

Anonymous function

C# allows to define a function without a name, called *anonymous function*. The function is represented as an expression that can be called or stored in a variable. There are three types of anonymous function. The first type is *anonymous method* shown in Figure 2.4 where it is stored in the `a` variable. The `b` variable contains the second type called *explicit typed anonymous function*. The third variable `c` contains the last type called *implicit typed anonymous function*. As can be seen, all of them have inferred return types based on return expression inside their bodies. The most interesting type is the last one, where even parameter types are inferred based on a surrounding context and which is especially threatened by *method type inference* algorithm mentioned in method type inference section 2.4.

```
Func<int, int> a = delegate(int p1) { return p1 + 1; };
Func<int, int> b = (int p1) => { return p1 + 1; };
Func<int, int> c = (p1) => { return p1 + 1; };
```

Figure 2.4: C# anonymous functions.

Object creation expression and initializer

Initializers are used as a shortcut during an object instantiation. The simplest one is *object initializer* allowing to assign values to the object's fields pleasantly instead of assigning them separately after the initialization. The second type of initializers regards arrays and collections. *Array initializers* are used to create fixed-size arrays with predefined content. Figure 2.5 shows the `arrayInit` variable initialized by an array of `int` with two items using the initializer. Under the hood, each item in the initializer is assigned to the corresponding index of the array after the array creation. *Collection initializers* are similar to array initializers defined on collections, which are created by implementing `ICollection<T>` interface. One of the interface's declaring methods is `void Add<T>(T)` with adding semantics. 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 an initializer uses an indexer to store referred values on predefined positions, which is used in the second statement where the `indexerInit` variable is initialized by a dictionary object using indexers in its initializer list.

```
var arrayInit = new int[] { 1, 2 };
var indexerInit = new Dictionary<string, int>() {
    ["a"] = 1, ["b"] = 2
};
```

Figure 2.5: C# collection initializer.

2.3 Type inference

C# type inference occurs in many contexts. However, the proposed improvement will be inspired by and influence only a few of them. These contexts are presented below.

2.3.1 Keyword `var`

One of the simplest type inference occurrences regards the `var` keyword used in a variable declaration. It lets the compiler decide the type of variable based on the type of initializing value, which implies that it can't use the keyword in declarations without initializing the value. Figure 2.6 shows the usage, where the type of the `a` variable is determined to be `string` since it is initialized by a string value.

```
var a = "str";
```

Figure 2.6: Keyword `var`.

2.3.2 Operator `new()`

There is also an opposite way of deducting types from a target to a source. An example is the `new()` operator, which can be called with arbitrary arguments and represents object creation of a type that is determined by a type of the target. An example of these situations can be seen in Figure 2.7 where the target-typed `new(1)` operator allows to skip the specification of creating type in the object creation expression since the `myList` variable type gives it. After the type inference, the operator represents the new `new List<int>(1)` object creation expression.

```
List<int> myList = new(1);
```

Figure 2.7: Operator `new()`.

2.3.3 Method type inference

Method type inference is the most complex C# type inference used during generic method call binding when type arguments are not given. Figure 2.8 shows a situation when the method type inference deduces `System.String`, `System.Int32` and `System.Int32` as type arguments of the `Foo` method. There is a multi-step process that the type inference has to do to be able to infer it. Regarding the `T1` type parameter, the inference has to find a common type between the `(long)1` argument and the `(int)1` argument. Regarding the `T2` type parameter, the type inference has to go into type arguments of the generic type of the `p3` parameter and the `myList` argument, check if the types are compatible, and then match the `T2` type parameter against the `int` type argument of the `List<int>`. The `T3` type parameter is the most challenging since it occurs as a return type of the delegate. The type inference has first to infer types of input parameters of this delegate to be able to infer the implicit anonymous function's return type. Then, it can match the inferred return type with the `T3` type parameter, resulting in the `System.Int32` type.

```
List<int> myList = ...
Foo((long)1, (int)1, myList, (p1) => p1 + 1);

Foo<T1, T2, T3>(T1 p1, T1 p2, IList<T2> p3, Func<T2, T3> p4) {...}
```

Figure 2.8: Method type inference.

The method type inference algorithm is detailly described in separate section 2.4 since it is a complex algorithm, and the proposed improvement will be based on that.

2.3.4 Array type inference

The last mentioned type inference happens in array initializers when the array type should be deduced from the initializer list. Figure 2.9 shows an example of a situation when the type inference is used for determining the `object[]` type of the `myArray` array. The C# specification calls it *common type inference*, which finds the most specialized common type between given types. From one point of view, it is just adjusted the method type inference algorithm where there is just one type variable, and all initializer items are lower bounds of that type variable.

```
var myArray = new[] {new object(), "string"};
```

Figure 2.9: Array type inference.

2.4 Method type inference algorithm

Since one of the thesis's improvements is adjusting the algorithm, this section presents its description. The thesis doesn't show the complete algorithm described in the C# specification [14] since it is complex, and some parts are unimportant for the following chapters. The simplified algorithm is divided into four subsections. The algorithm uses several definitions presented below.

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*.*

The pseudocode describing the algorithm uses custom helper functions explained in Table 2.1.

2.4.1 Algorithm phases

Figure 2.10 shows the initial phases of the algorithm. 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. The algorithm has two phases,

{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.

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 fail if there is insufficient information to deduce them. 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.

`FirstPhase()` iterates over provided arguments and matches their types with types of corresponding parameters. This matching has two goals. The first is to check the compatibility of matched types, and the second is to collect the mentioned bounds associated with type variables contained in parameters' types. This matching has many rules, followed by helping functions mentioned later in this section. The matching represents dealing with the `T2` type parameter mentioned in Figure 2.8 where the compatibility between `List<int>` and `IList<T2>` is firstly checked, and then the `int` type is added as a lower bound of the `T2` type parameter. Type variables can have dependencies between themselves, so the first phase postpones matching output types of arguments' types with corresponding parameters' types because the output types can contain dependent type variables.

`SecondPhase()` happens iteratively, respecting the *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 that could be used for type variable deduction, the algorithm ends. The sets X_{indep} and X_{dep} refer to type variables, which can be fixed in the current iteration. Line 31 contains the ending condition of the algorithm when all type variables are fixed, or there is no way to infer the next ones. The second goal is to match postponed matching of output types of arguments' types with output types of corresponding parameters' types where all type variables

```
Input: method call M(E_1,...E_x) and
       its signature T_e M<X_1,...,X_n>(T_1 p_1,...,T_x p_x)
Output: inferred X_1,...X_n
  B_lower = B_upper = B_exact = F = []
  FirstPhase()
  SecondPhase()

fn FirstPhase():
  E.foreach(e →
    if (e.isAnonymousFunc)
      InferExplicitParamterType(e, T[e.idx])
    elif (e.getType() is Type u)
      switch (u) {
        p[e.idx].isValParam → InferLowerBound(u, T[e.idx])
        p[e.idx].isRefParam || p[e.idx].isOutParam →
          InferExact(u, T[e.idx])
        p[e.idx].isInParam && e.isInArg →
          InferLowerBound(u, T[e.idx])
      }
  )

fn SecondPhase():
  while (true):
    X_indep = X.filter(x →
      F[x.idx] == null && X.any(x → dependsOn(x, y)))
    X_dep = X.filter(x →
      F[x.idx] == null && X.any(y →
        dependsOn(y, x) && (B_lower+B_upper+B_exact).isNotEmpty))
    switch {
      X_indep.isNotEmpty → X_indep.foreach(x → Fix(x))
      X_dep.isNotEmpty && X_indep.isEmpty → X_dep.foreach(x → Fix(x))
      (X_indep+X_dep).isEmpty →
        return if (F.any(x → x == null)) Fail() else Success(F)
      default → E.filter(e →
          X.any(x →
            F[x.idx] == null && T[e.idx].outTypes.contains(x))
          && !X.any(x →
            F[x.idx] == null && T[e.idx].inTypes.contains(x))
        ).foreach(e → InferOutputType(e, T[e.idx]))
    }
```

Figure 2.10: Phases of Method Type Inference

in the output type of a parameter type don't depend on unfixed variable types contained in input types of the parameter type. The second goal is to match postponed matching of output types of arguments' types with output types of corresponding parameters' types where all type variables in the output type of a parameter type don't depend on unfixed variable types contained in input types

of the parameter type. Line 32 describes this process using the pseudocode. This goal represents dealing with implicit anonymous functions mentioned in Figure 2.8 where `T3` depends on `T2`. The algorithm first infers the `T2` input type of the anonymous function, then infers the function's output type, which is then used to match the output type of `Func<T2, T3>` parameter type with the output type of the inferred anonymous function's type. The match yields the `int` upper bound of the `T3` type parameter.

2.4.2 Function type inference

Figure 2.11 contains definitions of two helper functions used in the first and second phases. The `ExplicitParameterType()` function is used to match an argument type, which is an explicit typed anonymous function. This function has typed parameters, so the algorithm matches them with input types of the corresponding parameter type.

The `InferOutputType()` function is used in the second phase when postponed matching of output types happens. Because potential type variables contained in the return types don't depend on any unfixed type variables, the algorithm can match them. There are two situations where the output type is matched. The first situation regards anonymous functions, where the algorithm first infers the return type represented by the `InferReturnType()` function and then matches it with the output type of the corresponding parameter. The second situation regards method groups, which are firstly resolved by `OverloadResolution()`. The return type of the resolved method is matched with the output type of the corresponding parameter.

```
1  fn InferExplicitParameterType(Argument E, Type T):
2    if (E.isExplicitTypedAnonymousFunc && T.isDelegateOrExprTreeType
3      && E.paramTypes.size == T.paramTypes.size)
4        E.paramTypes.zip(T.paramTypes)
5                    .foreach((e, t) → InferExact(e, t))
6  fn InferOutputType(Argument E, Type T):
7    switch(E) {
8      E.isAnonymousFunc && T.isDelegateOrExprTreeType →
9        InferLower(InferReturnType(E), T.returnType)
10     E.isMethodGroup && T.isDelegateOrExprTreeType → {
11         E_resolved= OverloadResolution(E, T.parameterTypes)
12         if (E_resolved.size == 1)
13           InferLower(E_resolved[0].returnType, T.returnType)
14       }
15     E.isExpression && E.getType() is Type u → InferLower(u, T)
16   }
```

Figure 2.11: *Explicit parameter type inference, Output type inference*

2.4.3 Collecting Type bounds

```
1  fn InferExact(Type U, Type V):
2    if (Type t = X.find(x → V == x && F[x.idx] == null))
3      B_exact[t.idx].add(U)
4    switch(V) {
5      V isLike 'V_1[..]' && U isLike 'U_1[..]' && V.rank == U.rank →
6        InferExact(U_1, V_1)
7      V isLike 'V_1?' && U isLike 'U_1' → InferExact(V_1, U_1)
8      V isLike 'C<V_1, ..., V_e>' && U isLike 'C<U_1, ..., U_e>' →
9        V.typeArgs.zip(U.typeArgs)
10                 .foreach((v, u) → InferExact(v, u))
11   }
12
13 fn InferLower(Type U, Type V):
14   if (Type t = X.find(x → V == x && F[x.idx] == null))
15     B_lower[t.idx].add(U)
16   switch { .... }
17
18 fn InferUpper(Type U, Type V):
19   if (Type t = X.find(x → V == x && F[x.idx] == null))
20     B_upper[t.idx].add(U)
21   switch { ... }
```

Figure 2.12: *Exact inference*, *Upper-bound inference*, *Lower-bound inference*

Figure 2.12 shows three functions that add new bounds to type variables' bound sets. Basically, all of them have similar behavior. It traverses the given `U` type contained in the argument type with the `V` type contained in the corresponding parameter type using the conditions contained in the `switch` statement and adds the `U` type to the type variable's bound when the `V` type is a type variable. Since the branching conditions in `InferLower` and `InferUpper` are similar to those in `InferExact` and unimportant for the proposed improvement, the thesis omits it.

Observation 1. *There is no need to check possible contradictions because they are checked in the type variable fixation.*

Observation 2. *Bound sets don't contain unfixed type variables, which makes the algorithm simpler and which will be important for the following chapters. The reason for that can be noticed in the design of functions API, where the left parameter always contains an expression or type from the argument, and the right parameter contains a type from the inferring method parameter. Since the three last-mentioned functions add only the type from the left parameter to bounds, there can't be any type variables because argument types don't contain type variables.*

2.4.4 Fixation

The last part of this algorithm is type variable fixation, shown in Figure 2.13. Initially, a set of candidates for the type variable is constructed by collecting all

its bounds. Then, it goes through each bound and removes the candidates who do not satisfy the bound's restriction. If there is more than one candidate left, it tries 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. This process can be seen in the initial example 2.8, where the `T1` type variable contains `long` and `int` in its lower bound set. At the start of this process, both types are candidates. However, `int` is removed because it doesn't have an implicit conversion to `long`.

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

Figure 2.13: Fixing of type variables

An important observation of the method type inference is an obligation of infering all type arguments of the method. If the compiler is not able to infer all type arguments, an user has to specify all type arguments. C# currently doesn't offer a way how to hint just ambigious type arguments.

3. Related work

This chapter follows by mentioning related work regarding the current implementation of the C# compiler and formalizing C# type inference using Hindley-Milner type inference, which is explored in more detail with references to its modification in Rust and C# programming languages. This knowledge will be utilized as a primary source of inspiration for the improvement. In the end, it mentions relevant C# language issues presented on the GitHub repository, which will be used later to prioritize the improvement features to make it more likely to be discussed at Language Design Meetings (LDM) held by LDT.

3.1 Roslyn

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. Presented Roslyn's architecture will help to better understand the context and restrictions that has to be cosidered to plug the improved type inference into the compiler. Figure 3.1 is used to explain the compilation pipeline [22] which consists of four phases highlighted with different colors.

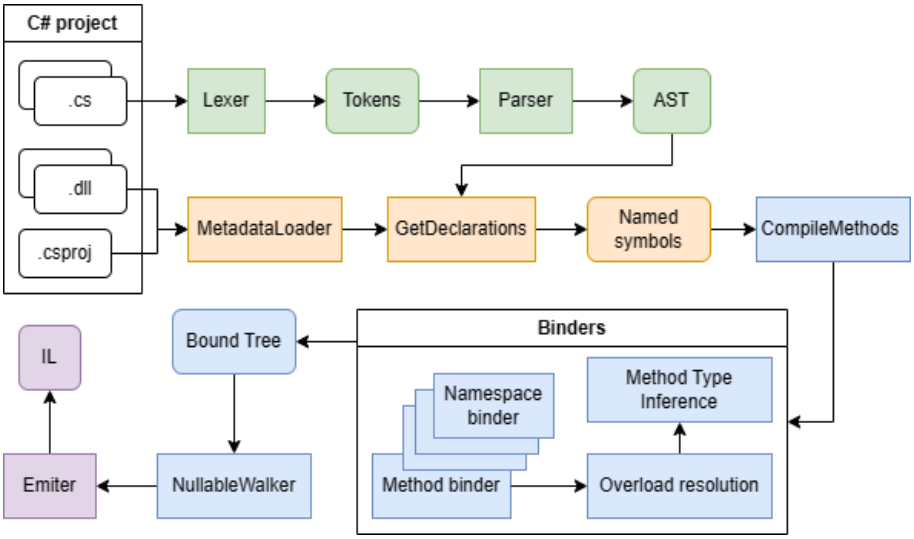

Figure 3.1: Roslyn architecture

3.1.1 Parsing C# sources phase

The pipeline starts with loading the `.csproj` file with related C# sources (`.cs`) and referenced libraries (`.dll`). C# sources are passed to the lexer, creating tokens used by the parser forming Abstract Syntax Tree (AST). AST construction is the first phase (green boxes in Figure 3.1) of the compiler checking syntax of C# sources.

3.1.2 Loading named symbols

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. Libraries, stored in `.dll` format are parsed by `MetadataLoader`, creating the same named symbols as those received from C# sources.

3.1.3 Binding phase

The third phase (represented by blue boxes), 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 is described in detail in the previous section 2.4.

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. The process of finding referred symbols is called *LookUp*. 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. An important 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 which would determine the type of the variable.

The preceding step before method type inference is overload resolution, part of `MethodCallExpression` or `ObjectCreationExpression` binding. As mentioned previously, method overloading allows to define multiple methods with the same name differing in parameters. So, when the compiler decides which method should be called, it has 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.

Method type inference can occur for the second time if previously mentioned

nullability analysis is turned on. 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.

3.1.4 Emiting code phase

The last phase, marked by purple, emits Common Intermediate Language (CIL) code targeting the .NET virtual machine. The code is later loaded and executed by .NET runtime.

3.2 Hindley-Millner type inference

C# method type inference is a restricted Hindley-Millner type inference which is able to work in C# type system. Since type inference in other languages like Rust or Haskell is based on the same principle, a high-level overview of Hindley-Millner type inference is presented together with its type system to formalize the C# type inference, compare it with Rust type inference formalization and propose possible extensions of current C# type inference based on these observations.

Hindley-Millner type system [24] is a type system for *lambda calculus* capable of generic functions and types. Lambda calculus contains four types of expressions given below which are described in the video series [4]. Expression is either a variable (3.1), a function application (3.2), a lambda function (3.3), or a *let-in* clause (3.4).

$$e = x \tag{3.1}$$
$$\mid e_1 e_2 \tag{3.2}$$
$$\mid \lambda x \rightarrow e \tag{3.3}$$
$$\mid \textbf{let } x = e_1 \textbf{ in } e_2 \tag{3.4}$$

The above-mentioned expressions have one of two kinds of types. *Mono* type is a type variable(3.5) or a function application(3.6) 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 kind is *Poly* type, which is arbitrary type with possible preceding $\forall$ operator 3.8, bounding its type variables.

$$mono\ \tau = \alpha \tag{3.5}$$
$$\mid C\ \tau_1, ..., \tau_n \tag{3.6}$$
$$poly\ \sigma = \tau \tag{3.7}$$
$$\mid \forall \alpha\ .\ \sigma \tag{3.8}$$

A context(Γ) contains bindings of an expression to its type which are described by pairs of expression and its type using the $x : \tau$ syntax. An assumption is than

described as a typing judgment shown in the $\Gamma \vdash x : \tau$ syntax meaning "In the given context Γ, x has the τ type".

The H-M deduction system gives the following inference rules, allowing to deduce the type of an expression based on the assumption given in the context. The syntax of a rule corresponds with what can be judged below the line based on assumptions given above the line. The rules can be divided into two kinds. The first four rules give a manual on what types can be expected by applying the mentioned expressions of lambda calculus. The two last rules allow to convert Poly types to Mono types and vice-versa.

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}[Variable]$$

$$\frac{\Gamma \vdash e_0 : \tau_a \to \tau_b \quad \Gamma \vdash e_1 : \tau_a}{\Gamma \vdash e_0 e_1 : \tau_b}[Function\ application]$$

$$\frac{\Gamma, x : \tau_a \vdash e : \tau_b}{\Gamma \vdash \lambda x \to e : \tau_a \to \tau_b}[Function\ abstraction]$$

$$\frac{\Gamma \to e_0 : \sigma \quad \Gamma, x : \sigma \vdash e_1 : \tau}{\Gamma \vdash \ \textbf{let}\ x = e_0\ \textbf{in}\ e_1 : \tau}[Let\ clause]$$

$$\frac{\Gamma \vdash e : \sigma_a \quad \sigma_a \sqsubseteq \sigma_b}{\Gamma \vdash e : \sigma_b}[Instantiate]$$

$$\frac{\Gamma \vdash e : \sigma \quad \alpha \notin Free(\Gamma)}{\Gamma \vdash e : \forall \alpha\ .\ \sigma}[Generalize]$$

H-M type inference is able to find the type of every expression of a completely untyped program using only these type rules. Although, there exist several algorithms for the inference 3.2 shows only the W algorithm since it is closely related to C# and Rust type inference. Inputs are the context Γ and an expression which type has to be inferred. The process consists of systematic traversing the expression from bottom to top and deducing the type of sub-expressions following the mentioned rules. The algorithm contains the `Instantiate` method which replaces quantified type variables in the expression with new type variables, the `Generelize` method replacing 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 of type variables 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.

Relations between this algorithm and C#, Rust type inference will be discussed in detail later, although there can noticed that the unification part of this algorithm is the same as method type inference mentioned in the C# section 2.4 where the substitution represents inferred type arguments of the method which parameter types were unified with corresponding argument types.

```
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 3.2: W algorithm

3.2.1 H-M etensions

H-M type system, 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 [27], where instead of accumulating type equivalent constraints, it accumulates and propagates subtyping constraints. These subtyping constraints consist of a set of types, which have to be inherited by the constrained type variable, or the variable has to inherit them.

Extending H-M type inference by supporting overloading is mentioned in Andreas Stadelmeier and Martin Plumicke's work [26]. 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 accumulating these sets, all combinations of items in OR-sets are generated and solved by type inference. For each method overload participating in type inference, it makes an type inference containing constraints obtained only from the overloaded method, excluding constraints obtained from 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.

3.3 Rust type inference

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 is now described in a high-level perspective to get

inspiration for the proposed C# improvement.

```
let mut a  = Vec::new();
a.push(1);
```

Figure 3.3: Rust type inference example.

Figure 3.3 shows a significant difference between C# type inference and global Rust type inference. There is the `a` variable declaraction initialized by the `Vec<T>` generic type which type argument is going to be inferred. The second statement calls the `push` method on the `a` variable, which is also generic and takes an argument of the `T` type. Since `1` value is passed into this call, `T` is inferred to be the `i32` type and the type argument of the creating vector becomes `i32`. An interesting behavior regarding the type inference is that it infers the type of creating object used in the first statement using information obtained from the second statement.

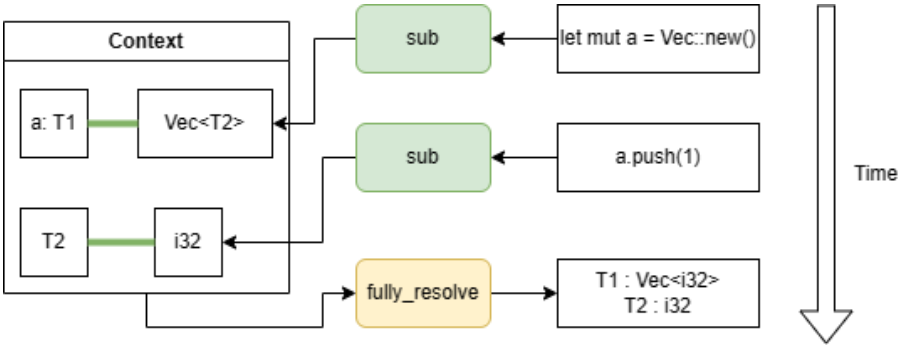

Figure 3.4: Rust type inference

This global type inference is possible thanks to a type inference context which is shared across multiple statements. Figure 3.4 shows a basic principle how the the context works. 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. The figure demonstrates it by adding a new type variable `T1` representing a type of the `a` variable. It uses the `sub` function which adds the `Vec<T2>` subtyping constaint to the `T1` variable represented by a thick green line. This constraint was obtained from a type of the initializing value `Vec::new()` which contains an unspecified type argument represented by a new `T2` type variable. Since, the initializing type can't be fully resolved because the constraint contains the `T2` unbound type variable it has to postpone the resolution. It passes the context to binding the next statement, where it collects another constraint about the `T2` type argument of the initializing value. The `sub` function is similar to the unification seen in the previous section 3.2, where it extracts 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 `T1` and `T2` type variables, they are resolved by finding an appropriate type for the type variable with respect to collected bounds. In the given example, the bounds contain only one type, so the type variables are resolved to them.

The mentioned sharing of context enables type inference to be backward, meaning that based on future type information, it is possible to infer already

collected 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#, it has to be considered language differences making type inference computation cost and difficulty relative to their features. As an example, the overloading can mentioned causing exponetial time of type inference. Since Rust doesn't have overloading, the type inference can be more powerful without significant slowdown, which is not the case of C# as will be shown in the following chapters.

3.4 Github issues

The last section describes a process of proposing new C# language features and mentions already existing ideas, presented in the C# language repository, which will be an inspiration to the thesis's proposed improvement.

The process of designing a new language feature starts with publishing an idea into discussions [17], 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 Github 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 LDT for being discussed further in LDM. The state of a proposal is described by several milestones. The most important for the thesis is the *AnyTime* milestone, 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 member of LDT recommended championed issue [12] regarding *partial type inference* to be investigated since it contains many related discussions with proposed changes but still doesn't have a required proposal specification which would allow to be discussed by the team. When a proposal has sufficient quality to be discussed by LDT, a member invites the proposer to make a *pull request* where further collaboration continues. If LDT 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.

The recommended issue doesn't contain a specific idea of the improvemenet rather the scope of the improvement. The improvement suggests *partial type inference* which would allow to hint ambigious type arguments which can't be inferred by a compiler instead of currently specifying the whole type argument list. Since it doesn't have any concerete way how to achive it, there are presented related discussions directly or indirectly mentioned in the issue which partially

suggest a possible solution.

3.4.1 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 3.5 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.

```
var temp = new A();

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

Figure 3.5: Default type parameters.

3.4.2 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. Figure3.6 shows an example where there is the `StringDictionary` generic alias specifying the first type argument of the `Dictionary` class to be the `string` type, which simplifies the usage of the `Dictionary` type in scenarios where there are often used dictionaries with keys of the `string` type.

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

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

Figure 3.6: Generic aliases.

3.4.3 Named type parameters

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 an user provides a type argument by name. Figure 3.7 shows a generic method `F` with two type parameters. The current type inference forces to specify all type arguments in the `F` method call since it is not able to infer the `U` type. Named type parameters offer a way how to tell the compiler specific type parameters for which an user provides type arguments, `U` in this case, and letting the compiler infer the rest of the type parameters.

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

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

Figure 3.7: Named type parameters.

3.4.4 Inferred type argument

Comments of the mentioned championed issue [12] 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. Figure 3.8 shows the `var` keyword for skipping the first type argument since it can be inferred from the argument list, and an user just specifies the second type argument, which can't be inferred by the compiler. The comments propose other options for keywords like underscore or whitespaces.

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

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

Figure 3.8: Using char as inferred type argument.

3.4.5 Target-typed inference

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 3.9, where type inference determines that the return type has to be `int` type and uses that to deduce the type argument `T`.

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

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

Figure 3.9: Target-typed inference.

3.4.6 Type inference based on constrains

The next idea of improving type inference is given by the discussion [8], where type inference utilizes type information obtained from type constraints. A simple example of that can be seen in Figure 3.10, where `T1` can be deduced by using `T1`'s constraint and inferred type of `T2` forming inferred type `List<int>`.

```
var temp = Foo(1);

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

Figure 3.10: Type inference based on type constraints.

3.4.7 Inferred method return type

Discussion [9] mentions type inference of method return type known from the Kotlin programming language. There is the usage in the following Figure 3.11, where the return type of the `Add` method is inferred to be `int` based on the type of the return expression.

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

Figure 3.11: Type inference of method return type.

3.4.8 Realocation

An issue [6] proposes a way to compact type argument lists of identifiers containing inner identifiers with argument lists. The idea is demonstrated in the example 3.12, where the argument list of `A<T1>` type and the `Foo<T2>` method are merged, and the type arguments are split by a semicolon.

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

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

Figure 3.12: Specifying type arguments in method calls (Realocation).

3.4.9 Constructor type inference

The last discussion [1], which is mentioned here, regards *constructor type inference* enabling type inference for object creation expressions. The type inference can be seen in Figure 3.13, where the `T` type parameter of the `C<T>` generic type can be deduced by using type information from its constructor.

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

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

Figure 3.13: Constructor type inference.

4. Problem analysis

The previous section 3.4 introduced the championed issue recommended by LDT. Since the description of the issue is not well defined, the work will continue to set the scope of the issue, which will bound the proposed improvement of this thesis. In that scope, it will identify a concrete motivation that should be solved by the improvement and which would be a real-world missing feature, making the proposal promising to become a potential future extension of C# language. Based on that motivation and information obtained from the previous sections, it will determine requirements that should be fulfilled by the improvement. The requirements will help to validate the thesis's goals regarding the improvement.

4.1 Scope

The previous section 2.3.3, regarding method type inference, shows that type inference is a complicated process, where even the current C# method type inference is difficult to understand. Hence, the thesis will choose a small part of C# where it will improve and introduce the type inference and will be possible to reason about and implement in the scope of this text. The second reason for choosing a minor change is that introducing a completely new type inference in C# would rather have an experimental result, which would have a smaller chance of getting into production. This consequence is different from the intention of this work. However, some more extensive changes in the type inference will also be mentioned to outline possible obstacles to introducing them in the C#.

The thesis will focus on the already-mentioned *partial type inference* proposal 3.4, which was recommended by a member of LDT and has a chance to be discussed in LDM and potentially accepted. Analysis of this improvement will contain a consideration of existing ideas, their consequences on C#, and their difficulties in implementing them in Roslyn. Additionally, the work will describe the relation to the Hindley-Millner formalization to express the strength of the type inference in a formalized way, which can be further used to compare it with other kinds of type inference in different programming languages and which decides the theoretical boundaries of the C# type inference.

4.2 Motivation

Partial type inference focuses on hinting to the compiler ambiguous type arguments of generic type or method in situations where it can't deduce them. In the context of C#, method type inference is the only type inference that infers type arguments. Even though method type inference is a complex algorithm, it has several weaknesses. The following three real-world examples demonstrate common issues with method type inference, which the thesis will try to solve.

4.2.1 Weakness - Target-typing

The first weakness regards target typing, which was mentioned in the previous chapter 3.4.5. Suppose a hypothetical situation when a user queries an item from a database whose column is a point of interest. Figure 4.1 shows an example of code that uses the `fetch` method defined on a database type. The `data` variable represents data fetched from a database. Since a concrete form of data is unknown, the data has the type of `object` containing an internal representation of fetched data with the columns stored as fields. The `GetField` method enables to read the variable's field of the given name with the supposed type given as a type argument. Suppose the fetched object contains the "name" field containing a string value. Now, a user wants to store the value in the `name` variable, which is explicitly typed. Even though the return type of the `GetField` method is known from the variable declaration, which also is the `TReturn` type argument of the method, the user still has to specify the type argument in the call. Generally, this problem consists of all type inferences, which depend on the target type. The target can be an argument of another method call or an assigning field. If the method type inference considers the target type, the user will not have to specify the `string` type argument in the `GetField` call.

```
object data = database.fetch();
string name = data.GetField<string>("name");
...
static class Extensions {
    static
    TReturn GetField<TReturn>(this object inst, string fieldName)
    {...}
}
```

Figure 4.1: Target-typed inference.

4.2.2 Weakness - Constraints-based inference

```
Test<TestCaseBase<MyData>, MyData>(new MyData());
...
void Test<T, U>(U data) where T : TestCaseBase<U> {...}
```

Figure 4.2: Constraints-based inference.

The second weakness is noticeable in more advanced generic APIs, like testing frameworks, using type constraints containing the type parameters. Figure 4.2 shows a scenario of a simple test framework that defines the `Test` method parameterized by a type of input data and test case represented as type parameters `U` and `V`, respectively. The provided type argument representing the test case has to inherit the `TestCaseBase` base implementation, which is a generic type parametrized by a type of input data. This constraint gives type information about the `T` type parameter, which is related to the type of input data. However, the user has to specify type arguments in the `Test` call since the type inference

doesn't consider this source of type information. If the compiler considers the constraint, the type arguments will be inferred, saving the type annotations.

4.2.3 Weakness - All or nothing principle

There are also situations where even strong type inference is not enough. Figure 4.3 shows a situation where the `log` method is parametrized by two type parameters that are obtained in the parameter types and hence inferable by the compiler. However, the `log` method call still has to specify type arguments because the `null` argument doesn't have concrete type information. In this case, the user always has to specify the second type argument, but the compiler can infer the first type argument. The thesis refers to this problem as *all or nothing* principle, which regards the obligation to specify all type arguments or none of them.

```
log<Message, Appendix>(new Message(...), null);
...
void log<T, U>(T message, U appendix) {...}
```

Figure 4.3: Uninferable type argument.

4.2.4 Solution - Improved method type inference

The first and the second weaknesses motivate us to extend the method type inference in order to consider a wider context for obtaining type information for the type arguments. This potential improvement is a problem for the compiler's back compatibility which was mentioned in the C# discussion [10]. New compiler versions should be back-compatible so that a new version does not change the behavior of the code compiled by the older version.

Figure 4.4 shows the breaking change when method type inference starts to consider target types. Before the improvement, the `M` method call is resolved to the non-generic version of this method because type inference can't infer the `T` type argument. After the improvement, the type inference infers `T` to be the `int` type, which is more specific to the type of `1` argument than the `long` type. So now, the `M` method call refers to the generic version of this method and executes different code without any warning or error.

```
int name = M(1);
...
T M<T>(int p1) {...}
int M(long p2) {...}
```

Figure 4.4: Breaking change: Target-typed inference.

Figure 4.5 shows a similar situation when the method type inference starts to consider type parameter constraints. Before the improvement, the `M` method call refers to the non-generic version of the method since the type inference can't infer the type argument of a generic version. After the improvement, the generic

version is inferred to have the `int` type argument and becomes to be more suitable for the overload resolution. So, the code behavior changed again because of compiling it with a different compiler's version.

```
M(1);
...
void M<T>(int p1) where T : List<int> {...}
void M(long p2) {...}
```

Figure 4.5: Breaking change: Constraints-based inference.

Besides the breaking change, the potential method type inference improvement to use a bigger context still doesn't solve our third weakness demonstrating a type parameter, which doesn't appear in parameter types, return type, and the type parameters' constraints. These obstacles give the reason for introducing a way to hint just ambiguous type arguments to the compiler.

4.2.5 Solution - Partial method type inference

Partial type inference can reduce the first two weaknesses. Type arguments, which the method type inference can't infer, can be hinted in order to avoid specifying the whole type argument list. Let's now ignore why the underscore character is used and how inferred type variables are determined in the following example. The reasons behind that will be mentioned later. Figure 4.6 shows the usage of partial type inference applied in the second presented example regarding method type inference weaknesses. Although the first type argument of the `Test` method call must still be provided, the second argument is omitted by using the underscore character to determine an inferred type argument. The reduction of the first weakness is to isolate the insufficient type inference of type arguments that are directly influenced by it and infer the rest.

```
test<TestCaseDefault<MyData>, _>(new MyData());

void test<T, U>(U data) where T : TestCaseDefault<U> {...}
```

Figure 4.6: Partial type inference: Reducing method type inference weakness.

The third motivation example confirms that partial type inference is not just a fix for missing type inference features but is needed when type arguments can't be inferred at all. Figure 4.7 demonstrates a usage of partial type inference where it omits the first type argument since it can be deduced from the first argument type and specifies the ambiguous type that can't be deduced.

```
log<_, Appendix>(new Message(...), null);

void log<T, U>(T message, U appendix) {...}
```

Figure 4.7: Partial type inference: Solving the *all or nothing* problem.

4.2.6 Solution - Constructor type inference

Partial type inference doesn't regard only method type inference. It can also be introduced in other places. One of the places that seems to be good for that is object creation expression. Except for the already mentioned `new()` operator, no other type inference infers type arguments of a construing generic type. The usage of the type inference is limited since the `new()` operator requires a target type to infer the construing type. Figure 4.8 shows an example of the limitation, where the `new()` operator can't be used since the `IWrapper` target type is not the `Wrapper<int>` construing type. Hence, the user has to specify the whole type with the `int` type argument, despite the fact that it could be inferred using method type inference.

```
IWrapper a = new Wrapper<int>(1);

class Wrapper<T> : IWrapper { public Wrapper(T item) {...} }
```

Figure 4.8: C# wrapper class.

Generally, object creation can be considered a special case of a method call with a side effect(creating the object), which already has method type inference. Figure 4.9 shows a workaround using the `Create` method, delegating the creation to the constructor call. Since the method call type arguments can be inferred, it allows the use of method type inference for inferring type arguments of construing type. However, this solution has disadvantages like the necessary boiler-plate and prohibition of using initializers.

```
IWrapper a = Create(1);

static Wrapper<T> Create<T>(T item) => new Wrapper<T>(item);
```

Figure 4.9: Workaround of constructor type inference.

A possible solution would be to use method type inference in object creation expression. Although this solution would be simple to implement, class type parameters are more likely not to be used in constructor parameter types, which makes the method type inference useless. Besides that, options for inferring type arguments of construing type are not limited by not introducing breaking changes since there is no type inference at all. So, there is a possibility of introducing an even stronger type inference, which could be one day introduced in the method type inference when there would be a way to make breaking changes in the new compiler version. Figure 4.10 shows an example of such a generic class whose all

```
class Algorithm<TData, TLogger> where TLogger : Logger<TData>
{ public Algorithm(TData data) { ... }}
```

Figure 4.10: Use case using type paraemter constraints.

type parameters are not used in the constructor. Because of that, extending the potential method type inference to be used in object creation expressions would be useless since the `TLogger` can't be inferred only from parameter types.

Introducing improved type inference based on method type inference would solve the mentioned issues. Figure 4.11 shows a potential usage of that type inference in the first case regarding the Wrapper class where an underscore is used to represent inferred type argument. The inference uses the parameter type of the constructor to infer the `T` parameter type which is `int`.

```
IWrapper a = new Wrapper<_>(1);

class Wrapper<T> : IWrapper { public Wrapper(T item) {...} }
```

Figure 4.11: Constructor type inference: Wrapper.

Figure 4.12 shows a potential extension of the type inference. The first statement of initializing the `alg` variable uses type inference, leveraging `TLogger`'s constraint to determine its type. The second statement demonstrates the possibility of having a nested underscore, which allows to more specify the type argument.

```
var alg = new Algorithm<_, _>(new MyData());
var algWithSpecialLogger =
    new Algorithm<_ , SpecialLogger<_>>(new MyData());

class Algorithm<TData, TLogger>
    where TLogger : Logger<TData>
{ public Algorithm(TData data) { ... }}
```

Figure 4.12: Constructor type inference: stronger method type inference.

From now on, thesis calls *constructor type inference* for introducing such a type inference.

4.3 Requirements

This section mentions requirements that have to be fulfilled by the improvement to be likely discussed by LDT.

Back compatibility is one of the most important requirements for new language features. The improvement shouldn't introduce a breaking change. However, this requirement is sometimes too strict for improvements, which would be very beneficial, and its breaking change would appear in cases that seem to be rare in the code. These improvements can break back compatibility by providing additional warnings or errors alerting a user of possible code behavior changes. Figure 4.13 shows an introduced breaking change when record classes were added into the C# language. Before the change, the `B` identifier referred to a method without parameters and returned the `record` type. After the change, the `B` identifier refers to a new record type declaration. There is an example where the breaking change can appear when there is a type with the `record` name. These situations are uncommon, and the improvement benefit was big enough to be added to the language. The possible breaking change is notified to the user by a compilation error.

```
class record {}
class A {
    record B(){...}
}
```

Figure 4.13: C# record class breaking change.

Convenience is a key requirement to make the improvement useful. Regarding partial type inference, the improvement should propose a convenient way to skip ambiguous type arguments. The way should also be possible to use in different places where skipping type arguments could yield an advantage, like type variable declaration or casting to a different type. Constructor type inference should be advanced enough to cover the mentioned examples in the previous section.

Extensibility would make the improvement open for new features that can be needed in future language versions. The improvement should consider possible future improvements and not be a blocker for them.

Performance is a critical part of Roslyn and which is one of the main goals of this project. The time complexity added by the thesis's improvement shouldn't be too big in order to not slow the compilation process.

5. Language feature design

The language feature design will be based on existing ideas of the partial type inference mentioned in the previous chapter 3.4 and will be validated by the given requirements. Besides the language feature design, the last section mentions type inference improvements, whose implementation and proposal is not in the scope of this text, altought their possible implementation is sketched for being inspiration of future improvements.

The mentioned ideas can be categorized into three groups. The first group consists of *Target-typed inference* 3.4.5 and *Type inference based on constraints* 3.4.6, which improve method type inference algorithm. The second group consists of *Default type parameters* 3.4.1, *Generic aliases* 3.4.2, *Named type parameters* 3.4.3, *Inferred type argument* 3.4.4, and *Relocation* 3.4.8, which regards partial type inference. The third group consists of *Constructor type inference* 3.4.9 and *Inferred method return type* 3.4.7, which introduce type inference in new C# constructs. Inferred method return type is not in the scope of this work, although it is mentioned in the last section as a future improvement that will be difficult to implement. The first group is discussed in relation to constructor type inference, where potentially applied method type inference can be improved without introducing breaking changes. The second group is discussed in relation to partial type inference, which is an objective of the championed issue.

5.1 Partial method type inference

Discussions regarding partial type inference can be further divided into two groups. The first group provides hints in type parameter declarations, which are done by default type parameters. Since it has to be done in the declaration, partial type inference wouldn't work with already existing code without adjusting it. For this reason, the work excludes the idea.

The second group provides hints through the usage. Generic aliases don't work for methods, and also, it doesn't seem to be useful when the inferred type arguments don't represent some common specialization of generic type. An example of the specialization is `StringDictionary<TValue>` mentioned in the previous chapter 3.4.2.

Named type parameters are excluded since providing that would be an uncommon new feature that has no equivalent in other well-known languages like Java, C++, Kotlin, and Rust. We believe it would be confusing to introduce it to the users since it is a controversial change.

Relocation doesn't solve the problem of specifying all type arguments. It just compacts type argument lists into one.

The last discussion regards using inferred type argument, which will be a core of the proposed design for the following reasons. It is already used in different languages, like a star in Kolin and Java or an underscore in Rust and F#. So, it is more common and intentional than previously mentioned ideas. It introduces no or at least minimal syntax changes into the language, which makes the usage simple, and it solves the problem of specifying all type arguments.

5.1.1 Syntax

The choice of the syntax is based on six use cases where the expression of inferred type argument can be used or could be used in the future. Table 5.1 shows examples of these usages, which identify necessary syntax requirements.

Generic method call	`Foo<{SYNTAX}>(arg1, arg2,...)`
Object creation expression	`new Bar<{SYNTAX}>(arg1, arg2,...)`
Variable declaration	`Bar<{SYNTAX}> temp = ...`
Array type	`{SYNTAX}[]`
Inferred type	`{SYNTAX}`
Inferred nullable type	`{SYNTAX}?`

Table 5.1: Use cases containing syntax for inferred type argument.

Generic method call use case represents a situation where the syntax is used during a generic method call.

Object creation expression represents the usage in the construction of generic type.

Variable declaration represents the potential usage in the variable type declaration, which is not in the scope of this work. However, it is a natural continuation of partial type inference extension where the inferred type arguments are determined by the surrounding context.

Array type has a different syntax of type argument list specifying a type of the contained elements. The type of the elements can also be inferred, and the proposed syntax has to offer a convenient way to express an array type with the inferred element type.

Inferred type doesn't have to be contained only in the type argument list. The previous chapter presented the `var` keyword used in the variable declaration, whose type is determined by the type of initializing value. C# also offers a discard pattern [16] represented by an underscore, which is commonly used as a placeholder for variables that are not intentionally used in the code. Since type argument inference relates to that, the syntax should be at least aligned with already existing related syntax. The alignment will help the syntax to be naturally used without using language documentation.

Inferred nullable type can be used to specify nullability of the inferred type which will be useful in scenarious where non-nullable and nullable code is mixed together.

There are several syntax variants that can be used for inferred types. The work presents the most relevant variants that appeared in the mentioned ideas and comments on the advantages and disadvantages for each variant.

Diamond operator

Syntax consists of a pair of two angle brackets `<>`. It is used as an empty type argument list determining that the type name or method name is generic.

Figure 5.1 demonstrates the usage in generic method calls of `Foo1` and `Foo2`. The declarations of these methods are not important for the purpose of showing the pros and cons. Calling the `Foo1` method with an empty argument list doesn't

make much sense since method type inference is enabled by default without using angle brackets. Calling the `Foo2` method with nested usage of the diamond operator allows limited partial type inference when the diamond operator is used inside the type argument list. The `Bar<>` type partially hints to the compiler the generic type name without specifying the type arguments. However, the usage is problematic when there are multiple generic types with the same name since it can cause ambiguity between them. This problem is described in the next example.

```
Foo1<>(arg1, arg2, arg3);
Foo2<Bar<>, int>(arg1, arg2, arg3);
```

Figure 5.1: Diamond operator - generic method call.

Figure 5.2 shows the usage in object creation expression. The first statement containing the `Bar<>` name has the advantage of expressing the will to infer the type's type arguments, which is necessary in comparison to the previous example since the object creation doesn't offer type inference. If the type inference were potentially turned on by default, it would introduce a breaking change. However, there is also a disadvantage of the usage since it doesn't specify the arity of the generic type. If there were multiple generic types differing in arity, it would complicate the already mentioned overload resolution phase since all constructors from these generic types would have to be considered. The process would be computationally demanding. A possible solution would be the following restriction. Usually, there is not more than one generic type with the same name. So when there is just one type of that name, the diamond operator would be allowed to use it since the name determined one specific generic type. In the example, `Baz<>` would refer to the `Baz<T1, T2>` generic type since there is no other generic type with the same name, causing the ambiguity.

```
new Bar<>(...);
new Baz<>(...);

class Bar { ... }
class Bar<T1> { ... }
class Bar<T1, T2> { ... }

class Baz<T1,T2> { ... }
```

Figure 5.2: Diamond operator - object creation expression.

The third example is shown in Figure 5.3, where the generic `Wrapper` class is used to specify the wrapping generic type. However, it doesn't offer to specify any type arguments, which is limiting when the compiler can't infer all of them.

```
Wrapper<> temp = ...
```

Figure 5.3: Diamond operator - variable declaration.

Figure 5.4 shows the operator as an indicator of the inferred element of the array type. Since there isn't a known popular language similar to C#, which

would contain a similar construct, the usage is considered to be unintentional for most of the users.

```
<>[] temp = ...
```

Figure 5.4: Diamond operator - array type.

Similar conclusions are made in the last example in Figure 5.5 where the `<>` operator has the same functionality as the already existing `var` keyword.

```
<> temp = ...
```

Figure 5.5: Diamond operator - inferred type.

The `?` nullable operator can be appended to the diamond operator. In general, each syntax introducing a placeholder which is not a whitespace is suitable for appending the `?` operator.

Whitespace separated by commas

The syntax would simly skip inferred type argument by whitespace which would allow to specify just the ambiguous type arguments. It would also determine the arity of type, which is important for the type name look-up phase of the compiler. A natural choice for the separator is commas since it is widely used. Another advantage is that this syntax is already used when working with C# reflaction. An example is the following expression `typeof(Dictionary<,>)`, which returns a class describing the `Dictionary` type from the standard library.

Figure 5.6 shows the usage when the `Foo1` generic method is called by skipping the first and the last type argument, which is inferred by the compiler. However, the syntax seems to be messy when it is used in generic methods with many generic type parameters, as can be seen in the `Foo2` method call. Similar thoughts regard the second and third use case.

```
Foo1<, string, List<>, >(arg1, arg2, arg3);
Foo2<,,,int,,>(arg1, arg2, arg3);
```

Figure 5.6: Whitespace - generic method call.

The usage with array type seems to be unintentional and would cause changes in the compiler parser. Figure 5.7 shows the syntax to express the inferred type of array's element.

```
[] temp = ...
Foo<, [], >(arg1, arg2)
```

Figure 5.7: Whitespace - array type.

The last use case regarding using the whitespace as an inferred type would probably cause problems with determining declarations of variables. Figure 5.8

```
temp = ...
```

Figure 5.8: Whitespace - inferred type.

shows a situation where the compiler can't determine if it is a variable declaration or a variable assignment. Although it could be solved by investigating the surrounding context, we consider that the code comprehension would get worse.

Since there is no placeholder which will be prepended to the nullable operator, the syntax doesn't work well in the last scenario regarding the nullable operator.

Underscores separated by commas

The syntax is commonly used in other programming languages like F# or Haskell to represent inferred type arguments, or as a placeholder for discarding variables that are intentionally not used. This is considered as a big advantage. A disadvantage is the introduction of breaking change because C# allows the underscore as a type identifier. However, that seems to be rather uncommon in the code. It seems to be less messy than the previous syntax when a generic type contains many type parameters, as shown in Figure 5.9.

```
new Bar<_, _, List<_>, _,_>(arg1, arg2, arg3);
```

Figure 5.9: Underscore - object creation expression.

The usage with array type also seems to work even if it can still be considered uncommon. Figure 5.10 shows the usage where we think that the syntax intentionally expresses the inferred type of array's elements.

```
_[] temp = ...
```

Figure 5.10: Underscore - array type.

The last use case is shown in Figure 5.11 where it clashes with the already existing `var` keyword.

```
_ temp = ...
```

Figure 5.11: Underscore - variable declaration.

`var` keywords separated by commas

The `var` keyword representing the placeholder is another natural option of the syntax. The big advantage of the syntax is the already used the `var` keyword in the variable declaration, whose meaning is coherent with inferred type arguments. However, it starts to raise the question if it brings the advantage of saving keystrokes. Figure 5.12 shows a usage of the syntax, which is considered to be unnecessarily long.

```
Foo<var, string, List<var>, int>(arg1, arg2, arg3);
```

Figure 5.12: `var` - generic method call.

Something else separated by commas

A different placeholder doesn't make a lot of sense because it needs to assign new meaning to that character in comparison with an underscore, the `var` keyword, the `<>` operator, or `<,,,>` syntax. An asterisk can be considered. However, it can remind a pointer in the context of unmanaged C# code.

Conslusion

The thesis chooses the underscore as a placeholder for inferred type argument since the meaning of this character is related to the intention. It also seems to be the shortest and synoptical way to skip inferred type arguments. The possible breaking change is not an obstacle in this situation since a similar decision was made for the `var` keyword, and the situation where it can occur seems to be rare. Problems with the potential future extension where the underscore can represent inferred type in the variable declaration would be prohibited to not mix it with the `var` keyword. Although the diamond operator is not very useful in a generic method call, it makes sense in object creation expression. The usage and analysis of that is covered by later in this work.

5.1.2 Method and typename lookup

The previous section presented the proposed syntax for skipping inferred type arguments using an underscore as a placeholder. This section continues with determining what the expression containing the syntax means.

Since an underscore character is a valid type identifier in C# and there is 1:1 mapping of inferred type arguments to these placeholders, determining the referred generic method containing the proposed syntax is almost unchanged. The change is in overload resolution where if the generic method is *partially inferred*, meaning it contains the syntax, the type inference has to be done to determine the type arguments of that method.

An underscore itself can be a nullable or non-nullable type. If the inferred type argument has to be a nullable type, the metioned `?` operator can be appended to the underscore. Figure 5.13 shows an example of using the nullable operator. The call of `Foo` generic method has three type arguments where the first one is inferred by the compiler and which has to be nullable. The second type argument is the `List` type, containing an inferred nullable type argument as well. The third type argument doesn't require the nullable type, so the inferred type can be either nullable or non-nullable.

```
Foo<_?, List<_?>, _>(arg1, arg2, arg3);
```

Figure 5.13: Inferring nullable type argument.

The type name lookup is also almost unchanged. If there is an underscore referring to inferred type argument, it simply ignores the binding of this identifier.

The underscores contained in the type arguments are threatened in the changed type inference algorithm mentioned in the following section.

5.2 Method type inference algorithm change

The thesis extends method type inference by introducing a new type variable bound, which represents inferred type arguments contained in the type argument list. Firstly, if a generic method call doesn't contain a type argument list, the method type inference is unchanged. The change is when the generic method is partially inferred. Figure 5.14 shows an example of a partially inferred method `Foo` containing two type parameters. The former algorithm identifies the type arguments as type variables for which it tries to find a unique type. The algorithm can be extended to represent placeholders for inferred type arguments as type variables, too. Using the example, all three underscores would be represented as three unique type variables besides those representing `T1` and `T2` type parameters. However, this extension also has to be respected by the order of type variables fixing and inferring. A reason can be seen in the `T2` type variable. The `Dictionary<_, _>` can be considered as a bound for the type variable, which has to be respected. However, this bound contains other type variables which are not yet known. So, the algorithm has to first infer the type variables contained in that bound and then infer the `T2` type variable. The second observation for this extension is a way of inferring the bounds. Since type variable bounds can contain other type variables, it is necessary to propagate the relation between the bounds at the time of adding new bounds. An example of this can be demonstrated using the given Figure 5.14. The `Dictionary<_, _>` is a bound of the `T2` type variable. The second bound is a type of the `p2` argument. This type gives us bounds for type variables contained in the former bound, which has to be propagated.

```
Foo<_, Dictionary<_, _>>(arg1, new Dictionary<int, int>());

void Foo<T1, T2>(T1 p1, T2 p2)
```

Figure 5.14: Partially inferred method call.

5.2.1 New definitions

The proposed algorithm uses two new definitions of dependendencies which are given below.

Definition 5 (Shape dependence)**.** *An unfixed type variable X_i* shape-depends directly *on an unfixed type variable X_e if X_e represents inferred_type_argument and it is contained in shape bound of the type variable X_i. X_e shape-depends on X_i if X_e shape-depends directly on X_i or if X_i shape-depends directly on X_v and X_v shape-depends on X_e. Thus shape-depends on is the transitive but not reflexive closure of shape-depends directly on.*

Definition 6 (Type dependence)**.** *An unfixed type variable X_i type-depends directly on an unfixed type variable X_e if X_e occurs in any bound of type variable*

X_i. X_e type-depends on X_i if X_e type-depends directly on X_i or if X_i type-depends directly on X_v and X_v type-depends on X_e. Thus type-depends on is the transitive but not reflexive closure of type-depends directly on.

5.2.2 Algorithm phases

The required change of the algorithm is presented as an algorithm divided into three sections, which is based on the former method type inference. Figure 5.15 shows the beginning, the first, and second phases. The first step is to identify all type variables, which would be an objective of the type inference. This is done by the `getAllTypeVariables` function, which replaces the underscore placeholders in the provided type arguments (If the type arguments were provided) with new type variables and joins them with type variables representing type parameters of the method. Besides the already known three types of bound, the algorithm adds *shape-bound* representing a type argument given in the type argument list. The reason for a new type of bound is the following. When a user provides the type argument, the algorithm should infer the exact same type(not containing any unfixed type variables). None of the already introduced type bounds offers this feature. An example of this need is described by a potential scenario when there is the `IList<_>` type as a type argument. When the compiler treats nullability, it wants the hinted type parameter to be non-nullable(not `IList<_>?`). It can happen, that other bounds would infer the nullable version, and although `IList<_>` can be converted to `IList<_>?`, it is not the user's intention. However, the exact bound would allow this.

`FirstPhase` collects shape bounds from the provided type argument list before the initial collection of bounds from the argument list. The referring `InferShapeBound` is described later with the rest of inferring methods.

`SecondPhase` now respects newly added dependencies, which forces to infer type variables in the correct order. If there are no type variables that are independent, the algorithm relaxes the condition for type variable fixation to break the possible circular dependency, which still has a chance to be resolved. In comparison to the former algorithm, the relaxation still has to respect *shape-depends on* relation. The reason for that is to prohibit inferred type from being different from the provided "shape" in the type argument list. The next condition is that at least one bound can't contain an unfixed type variable. This requirement ensures at least one candidate, which could be the inferred type argument.

```
1  Input: method call M<S_1,...S_n>(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,...X_{n+l}
4  B_lower = B_upper = B_exact = B_shape =F = []
5  TV = getAllTypeVariables(X, S)
6  FirstPhase()
7  SecondPhase()
8  fn FirstPhase():
9    S.foreach(s -> InferShapeBound(s, T[s.idx]))
10   /*Continuation as the former method type inference*/
11 fn SecondPhase():
12   while (true):
13     TV_indep = TV.filter(x →
14       F[x.idx] == null && TV.any(x →
15         dependsOn(x, y) && shapeDependsOn(x, y)
16         && typeDependsOn(x, y)
17       )
18     )
19     TV_dep = TV.filter(x →
20       F[x.idx] == null && TV.any(y →
21         (dependsOn(y, x) || shapeDependsOn(y, x)
22           || typeDependsOn(y, x))
23         && !TV.any(t → shapeDependsOn(x, t))
24         && (B_lower+B_upper+B_exact+B_shape).any(b →
25             !b.containsUnfixedTypeVariable
26         )
27       )
28     )
29     /*Continuation as the former method type inference*/
```

Figure 5.15: Phases of new Method Type Inference

5.2.3 Collecting Type bounds

Figure 5.16 shows three adjusted inferences, adding new bounds, and presents a new `InferShape` inference. The change is in propagating nested type bounds between type variable bounds. This is done by the `Propagate` function, which is invoked after a new bound is added. It iterates over all bounds of the type variable and makes additional type inferences for each bound containing an unfixed type variable checked by the `containsUnfixedTypeVariable` property. This step will ensure that the unfixed type variable will receive a bound which is associated with it. Using mentioned example 5.14, this phase propagates the `int` type bounds contained in the `Dictionary<int, int>` type of provided argument to the underscores representing type variables in the `Dictionary<_ ,_>` type argument. Since the design of the algorithm always adds type bounds received from the left argument of algorithm's functions to the type variable obtained from the right argument of algorithm's functions, the inference has to be done for each transposition of the pair of the added bound and an already collected type variable

bound.

```
1  fn Propagate(Type U, int typeVariable) {
2    setOf(B_shape[typeVariable],
3      B_lower[typeVariable],
4      B_upper[typeVariable],
5      B_exact[typeVariable]
6    ).foreach(b →
7      if (b.containsUnfixedTypeVariable) InferHelper(U, b)
8      if (U.containsUnfixedTypeVariable) InferHelper(b, U)
9    )
10 }
11
12 fn InferExact(Type U, Type V):
13   if (TypeVariable t = TV.find(x → V == x && F[x.idx] == null) &&
14     !B_exact[t.idx].contains(U)) {
15     B_exact[t.idx].add(U)
16     Propagate(U, t.idx)
17   }
18   /*Continuation as the former method type inference*/
19 fn InferLower(Type U, Type V):
20   if (TypeVariable t = TV.find(x → V == x && F[x.idx] == null) &&
21     !B_lower[t.idx].contains(U)) {
22     B_lower[t.idx].add(U)
23     Propagate(U, t.idx)
24   }
25   /*Continuation as the former method type inference*/
26 fn InferUpper(Type U, Type V):
27   if (TypeVariable t = TV.find(x → V == x && F[x.idx] == null) &&
28     !B_upper[t.idx].contains(U)) {
29     B_upper[t.idx].add(U)
30     Propagate(U, t.idx)
31   }
32   /*Continuation as the former method type inference*/
33 fn InferShape(Type U, Type V):
34   if (TypeVariable t = TV.find(x → V == x && F[x.idx] == null)) {
35     B_shape[t.idx] = U
36     Propagate(U, t.idx)
37   }
```

Figure 5.16: *Exact inference*, *Upper-bound inference*, *Lower-bound inference*, *Shape-bound inference*

Table 5.2 shows which inference is called based on the inputs. For example, if the `U` represents the added lower bound and `b` is an exact bound, the algorithm calls the `InferUpper` function. The intuition behind the table is to respect the relation between the bounds of a type variable. So, for example, all bounds of lower bounds are lower bounds for exact, upper, and shape bounds of that type variable and are exact bounds for each other.

	Lower	Upper	Exact	Shape
Lower	InferExact	InferUpper	InferUpper	InferLower
Upper	InferLower	InferExact	InferLower	InferLower
Exact	InferLower	InferUpper	InferExact	InferExact
Shape	InferLower	InferUpper	InferExact	InfeExact

Table 5.2: Matrix of `InferHelper` function.

5.2.4 Fixation

Figure 5.17 shows the last part of the changed algorithm, type variable fixation. The set of candidates is changed to respect the shape-bound ability to express the exact form of the inferred type argument. So, if the type variable contains a shape bound, the candidate list contains only this type, and other bounds are used to check if the candidate doesn't contradict the collected bounds. There are two notes regarding this step. If there is a shape bound, it doesn't contain any unfixed type variables because of the condition in the second phase. Hence, it will be a valid type argument. It can happen that some bounds will contain unfixed type variables. In this case, these bounds are removed from the checking and candidates set.

```
1  fn Fix(TypeVariable x):
2    U_candidates =
3    if (B_shape[x.idx] != null)
4      setOf(B_shape[x.idx])
5    else
6      (B_lower[x.idx] + B_upper[x.idx] + B_exact[x.idx]).filter(b →
7          !b.containsUnfixedTypeVariable
8      )
9    /*Continuation as the former method type inference*/
```

Figure 5.17: Fixing of type variables

Observation 3. *The last observation is to ensure that the propagation will end. Since the algorithm doesn't add the same bound multiple times, the cycle can't occur.*

5.3 Partial constructor type inference

Constructor type inference, which will be a core part of the second part of the improvement, was mentioned in the discussion [1]. The design of the partial type inference regarding underscore placeholders will be used again in order to be coherent with method type inference. Problems regarding using generic aliases, default type parameters, or named type parameters persist, so either constructor type inference will not be based on them. However, since the inference is not limited by introducing breaking changes because there is no type inference at all, the type inference can be stronger. The stronger type inference regards target-typed inference, which will be useful because object creation is usually associated

with assigning it to a target. It will also employ type inference based on type constraints, which will provide a wider context for type inference. Besides this functionality, the text will mention two additional features that will extend the context of type inference and will make the syntax less boilerplate, although they are controversial for LDT. For this reason, the features will be presented as a voluntary extension that can be removed from the core design. The first feature will regard using type information from initializers, which would be valuable when creating collections or objects with object initializer syntax. The second feature will use the diamond operator to turn on the constructor type inference in cases when the compiler can surely determine the referring generic type without knowledge about the arity.

5.3.1 Syntax

The syntax is similar to partial method type inference. Figure 5.18 shows a simple usage where the underscore representes the inferred type argument. This argument is deduced using a type received from the `t` variable declarations which determines the `int` type to be the inferred type argument.

```
List<int> t = new List<_>();
```

Figure 5.18: Syntax of constructor type inference.

Similar to the partial method type inferece, the `?` nullability operator is allowed to determine the type nullability.

5.3.2 Class lookup

Class lookup is done in the same way as in partial method type inference. The binding of the underscore identifier is skipped. The placeholder represents an inferred type, which has to be resolved during type inference. The inferred arguments are allowed only in an argument list of the type containing the called constructor. So, if there is a generic type containing a nested class that is being created, the type arguments of the generic class have to be provided. This limitation is shown in Figure 5.19, where the first statement is invalid since it contains inferred type arguments in the `GenericWrapper` identifier, which doesn't contain the called constructor. However, the second statement is valid because inferred type arguments are only in the `CreatingClass` identifier, which contains the called constructor.

```
new GenericWrapper<_>.CreatingClass<_>(arg1,...); // Not allowed
new GenericWrapper<int>.CreatingClass<_>(arg1,...); // Allowed
```

Figure 5.19: Allowed inferred type arguments.

5.3.3 Argument binding

The mentioned target-typed inference complicates the binding of the arguments. When a target is an assigned variable or a return statement, the type of the target

is given by the declaration of the variable or method's return type. However, when the target is an argument of the other method, the type doesn't have to be known yet if the method is generic. Figure 5.20 shows a scenario when, at the time of binding the `Foo` creation expression, the target type is unknown because the binding order is from the methods' arguments to the method call. Without the target type, `Foo`'s type argument can't be inferred. The `Bar` creation expression should be bound first, which would infer the type of the first parameter, which could be used as a target type of the `Foo` creation expression.

```
new Bar<_>(Foo<_>(), 1);

class Bar<T>{
    public Bar(Foo<T> arg1, T arg2){}
}
class Foo<T>{}
```

Figure 5.20: Target as an argument.

This process was mentioned regarding the target-typed `new()` operator. Roslyn solves the operator in the following way. The operator is bound as an unconverted bound element which needs to be converted in the future. When the right overload of the method is chosen, Roslyn has to generate the necessary conversions of these arguments, which don't have an identical type as the corresponding parameter. At that time, the operator is converted by using an already-known target type. The result of the conversion is the binding of the creation expression, which type is the target type, and arguments are contained in the type operator's argument list.

The improvement is inspired by that. When there is an object creation expression as an argument, and the constructor type inference fails, a similar unconverted bound element is created. After the overload resolution, when the types of parameters are known, and the necessary conversion is started to be created, the object creation expression is tried to be bound again with the already known target type. However, the compiler has to be careful here. It has to try to bind the object creation expression without a target type only once to prevent exponential time of binding.

Figure 5.21 shows a scenario where if the compiler does not do that, it will cause an exponential time of binding the expression. Since the constructor of the `Bar` type is overloaded, the overload resolution has to check these two overloads. Each of these checks includes binding of arguments, type inference, and checking of the applicability of bound arguments on the inferred constructor. Since the binding of arguments contains binding of the similar object creation expression, it will make the same overload resolution containing all overloads of the constructor. The target type can't be provided here because the type argument is still unknown. When the binding of this argument is unsuccessful, the argument will be represented as an unconverted bound element and will be bound later after the resolution. This failure will happen for each overload resolution, which wastes time because it does the same computation multiple times. After the outer object creation expression is inferred using the type of the second argument, it will convert the inner object creation expression by providing the target type, which

will resolve the valid object creation. This repeating failure is not a problem for the `new()` operator since it is cheap to express it automatically as an unconverted bound expression. However, the following observation offers a way to avoid it. If the argument can't be bound without the target type in checking one overload, it also can't be bound without the target type in the other overloads. So, the improvement remembers if the argument containing the object creation expression was already bound without the target type and failed. If this is true, it automatically skips the binding and returns an unconverted bound element, which will prevent obvious failure and save time.

```
new Bar<_>(new Bar<_>(null, null), 1);

class Bar<T>{
    public Bar(Foo<T> arg1, T arg2){}
    public Bar(Foo<T> arg1, List<T> arg2){}
}
```

Figure 5.21: Potential exponential time of binding.

5.3.4 Constructor type inference algorithm

The previously mentioned algorithm for inferring method type arguments is unchanged except for collecting new bounds in the first phase of the algorithm. If a target type is provided, an upper bound type inference is made from it to the partially inferred type containing the constructor. If the inferring type contains `where` clause containing type parameter constraints, for each of the type constraints which represented either inheritance constraints or interface implementation constraints is performed lower bound type inference from it to the corresponding type parameter.

```
IBar<int> t = Bar<_, _>();


interface IBar<T> {}

class Bar<T1, T2> : IBar<T1> where T2 : object {}
```

Figure 5.22: Example of type inferences.

Figure 5.22 shows an example of constructor type inference, where both of the type arguments of the `Bar` type are inferred. The algorithm will contain four type variables. The first two variables represent type arguments, and the second two variables represent the underscores. In the first phase, lower bound type inference is made from the `IBar<int>` type to the `Bar<_,_>` type, which will yield in the `int` lower bound of the type variable representing the first underscore. Then, the type constraint of `T2` type parameters causes another lower bound type inference from the `object` type to the type variable representing the `T2` type parameter. Shape inferences continue to relate type variables representing underscores with

type variables representing type parameters. Then, the fixation is made, and it results in `Bar<int, object>` inferred type.

5.3.5 Initiliazers extension

Initializers can be part of the creating expression, which will allow us to use it as another source of type information. As the thesis mentions in the previous chapter, initializers are syntax sugar. In the case of an object initializer, it represents a field assignment. If the field declaration contains a type parameter, the type of initializer element can be used to deduce the type parameter. Figure 5.23 shows an example of an initializer when the `Bar`'s type argument `int` can be deduced using type information from the initializer. Since the `Field` declaration type is the `T` type parameter and the `int` value is assigned to that field in the initializer, the compiler can deduce that the type argument is `int`. The improvement allows this inference by performing lower bound type inference for each item in the initializer. The inference is made from the type of assigning expression to the type of the field's type declaration.

```
new Bar<int>{ Field = 1 };

class Bar<T> {
    public T Field;
}
```

Figure 5.23: Object initializer.

A similar deduction can be made with array initializers where the element's type is inferred. The improvement will allow it by performing lower bound type inference for each item in the initializer. The inference is made from the type of assigning expression to the inferred type of array's element.

However, collection initializers have been shown to be a little tricky. Since it is a syntactical sugar for calling the special `Add` method for each element, the method can be overloaded. The previous section regarding Hindley-Millner type inference mentions that overloading can cause an exponential time of the type inference computation because it has to reason about each overload separately from the rest of the inferred bounds. To prevent slow compilation because of this issue, the improvement will allow the use of the information for the initializers in the case where the `Add` method is not overloaded. This limitation seems to be reasonable since a lot of collections from the standard library don't overload the method. When there is no overload, the issue with time complexity disappears. When the condition above is true, the first phase of the type inference algorithm additionally makes the lower bound inference for each item in the initializer. The inference is made from the type of the expression to the type of method parameter.

The last initializer regards indexers. Since indexers can be considered as a special case of methods that can be overloaded as well, the same process is made.

5.3.6 Diamond operator extension

Since constructor type inference is invoked when the creating type contains inferred type arguments in its type argument list, it can feel like a boilerplate when all type arguments are inferable. Figure 5.24 shows an example of creating the `Dictionary` generic type whose all type arguments can be inferred using the target type. Although C# allows the definition of generic types with the same name and different arity, it is not very common. The standard library doesn't contain another type of `Dictionary` with a different arity, so specifying the arity in the example is redundant in this case.

```
IDictionary<int, string> t = new Dictionary<_,_>();
```

Figure 5.24: Reduncant specification of arity.

For these use cases, where the compiler can confidently choose a generic type without looking at the arity, the diamond operator can be used to turn on the type inference. Figure 5.25 shows the usage of the diamond operator with the `Dictionary` type. Supposing that in the rest of the program, there is no other definition of `Dictionary` type with a different arity, the compiler can assume that the referring type is the `Dictionary` type from the standard library.

```
IDictionary<int, string> t = new Dictionary<>();
```

Figure 5.25: Diamond operator.

5.4 Partial time inferece during dynamic member invocation

The previous chapter mentioned that despite the dynamic binding, the compiler can still check an expression containing the dynamic value. The compiler checks three relevant expressions containing the dynamic value as an argument. It is a static method invocation, instance method invocation whose receiver is not a dynamic value, and object creation expression. For each candidate of this kind, a modified parameter list and argument list are created to be checked. If there is no candidate for which the test succeeded, a compile-time error occurs.

The modified parameter list is created in the following way. If the candidate is a generic method and type arguments were provided, it substitutes them in the parameter list. Then, all parameters that include a type parameter are elided with corresponding arguments. The resulting set of parameters and arguments are checked.

The improvement adjusts the modified parameter list of partially inferred methods by substituting only these type arguments, which don't contain any inferred type arguments. The same process is made in object creation expression, where the substitution is made on the type containing the constructor candidate.

For practical reasons, it also announces a warning about using partial inference in late-binding, which doesn't support it since it is handled by the runtime. However, there are situations where the runtime is able to infer type arguments

even without partially hinting. Figure 5.26 shows an example where the `Foo` method is inferred by runtime because inferred type arguments are inferrable in the runtime. These situations are valid, and hence, the compiler should just warn about them.

```
dynamic t = ...;
Foo<_, _>(t, 1);

void Foo<T>(T arg1, T arg2) {}
```

Figure 5.26: Runtime type inference.

5.5 Other type inference improvements

Besides the proposed improvement described above, the work touched on surrounding areas of type inference during the initial problem exploration. The chapter gives a couple of thoughts about touched areas since they were also investigated and can be a future extension of the C# programming language.

5.5.1 Shared type inference context

C# currently has a local type inference preventing advanced type inferences. The next big improvement of the type inference would be to provide global type inference using inferring contexts similar to Rust's. However, since it would bring a breaking change, it would require an additional tool that would patch the old code to work in the same way compiled with the new type inference. However, this refactoring can be challenging, so at least it should warn the user about possible behavior changes in each place where it can occur.

5.5.2 Inferring return value of methods

The work already mentioned the possibility of inferring the return value type of the method, which can be beneficial for writing a simple method whose name indicates the return value, helping a reader to understand the code. An example of the function can be `ToString`, which indicates that the return value is the `string` type.

This language feature can be seen in the Kotlin programming language, which allows the definition of a method without a return type if the method is exactly one expression. The Kotlin language is a strongly typed language developed by JetBrains. Kotlin's main target is JVM, and its goal is to be an alternative to Java with excellent interoperability between these languages to use them almost interchangeably in one project. Because of that, it has a similar type system as Java, which is not far away from C#. Figure 5.27 shows an example of the `MyClass` definition containing the `toMyString` method definition. The signature consists of the `fun` keyword, the name, and the equal sign followed by one expression. The method's return type is inferred based on the expression's return value.

```
class MyClass {
    fun toMyString() = "TEXT"
}
```

Figure 5.27: Kotlin return type inference.

The main obstacle regarding inferred return type in the method's signatures is an order of compilation and possible multithreaded compilation. The Roslyn compiler first finds all definitions in the program, which allows it to compile methods separately in different threads. This advantage is because the method's content consists of only types and methods defined in the program, whose signatures are already known thanks to the previous phase. So, the compiler knows the exact return types of the used functions, which allows it to do a type check.

Kotlin divides the method compilation into two groups, as it is described in the video [18]. The first group contains methods without a return type, which is compiled first in a single thread. This will allow us to obtain all signatures for the methods in the program. Obviously, if these methods are recursions, the compiler can't infer the return type, and an error occurs. The second group contains methods with a return type, which can be compiled in multiple threads since all method signatures are already known.

Although this implementation would be possible in Roslyn as well, it would require a large number of changes. Instead of that, C# could allow inferring return in a local function, which is a function defined inside a method body and can be used only inside the method. There are two benefits of this. The main architecture of the Roslyn compiler doesn't have to be changed since the method is compiled by one thread, and the local functions can't be used outside. Local functions are usually tied with the method implementation, and the return type is not contained in public API, so a reader shouldn't need to know the return type till the time when he/she wants to explore the inner implementation, which gives him/her a context to deduce the return type.

6. Solution

6.1 Proposal

6.2 Implementation

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.

7. Evaluation

TODO: Describe achieved type inference. Mention interesting capabilities.

TODO: Note about the performance.

TODO: Links to csharplang discussions.

8. 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 inference breaking change. https://github.com/dotnet/roslyn/pull/7850, . [Online; accessed 2023-12-3].

[11] C# data types. https://www.tutorialsteacher.com/csharp/csharp-data-types, . [Online; accessed 2023-09-22].

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

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

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

[15] C type constraints. https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/constraints-on-type-parameters, . [Online; accessed 2023-11-21].

[16] C discard patterns. https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/functional/discards, . [Online; accessed 2023-12-9].

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

[18] Overview of kotlin compiler. https://www.youtube.com/watch?v=db19VFLZqJM, . [Online; accessed 2023-12-19].

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

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

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

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

[23] Rust programming language. https://en.wikipedia.org/wiki/Rust_(programming_language), . [Online; accessed 2023-11-12].

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

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

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

[27] 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

LDT Language Design Team

LDM Language Design Meetings

AST Abstract Syntax Tree

CIL Common Intermediate Language

A. Attachments