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 would like to give my thanks to my supervisor, Mgr. Pavel Ježek, Ph.D., and consultant, Rikki Gibson, for their time and advice. I would like also to thank my girlfriend, Monika, for her support during writing this thesis.

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: C# is a strongly typed language utilizing type inference to save type annotations written by a programmer. However, the type inference is not as strong as other programming languages like Rust or Haskell. This thesis aims to propose the C# language improvement which would improve the current type inference and would be likely accepted by the C# language design team. For this achievement, we analyzed Rust's type inference and observed necessary language requirements and type inference restrictions based on Hindley-Millner's formalization of type inference. These observations were used to propose a language improvement consisting of two parts. The first part is a C# specification change describing the improvement by adjusting formal C# specification. This part was presented to the language design team which resulted in a positive reaction where the team decided to keep moving forward with the proposal to make it available in the future C# language version. The second part is an implementation of the improvement in the official C# language compiler, Roslyn. The implementation is tested by using the original compiler tests and new tests testing the proposal functionality.

Keywords: Type Inference C# Roslyn

Contents

1 Introduction 4
1.1 Improving C# Type System . . . 4
1.2 Implementation . . . 6
1.3 Summary . . . 6

2 C# Programming Language 7
2.1 Type System . . . 7
2.2 Relevant Constructs . . . 8
2.3 Type Inference . . . 10
2.3.1 Keyword `var` . . . 10
2.3.2 Operator `new()` . . . 11
2.3.3 Method Type Inference . . . 11
2.3.4 Array Type Inference . . . 12
2.4 Method Type Inference Algorithm . . . 12
2.4.1 Algorithm Phases . . . 12
2.4.2 Function Type Inference . . . 15
2.4.3 Collecting Type Bounds . . . 16
2.4.4 Fixation . . . 17

3 Related Work 18
3.1 Roslyn . . . 18
3.1.1 Parsing C# Sources Phase . . . 18
3.1.2 Loading Named Symbols . . . 19
3.1.3 Binding Phase . . . 19
3.1.4 Emiting Code Phase . . . 20
3.2 Hindley-Millner Type Inference . . . 20
3.2.1 H-M Extensions . . . 22
3.3 Rust Type Inference . . . 22
3.4 Language Design GitHub Issues . . . 24
3.4.1 Design Process and Championed Issue . . . 24
3.4.2 Topic: Default Type Parameters . . . 25
3.4.3 Topic: Generic Aliases . . . 25
3.4.4 Topic: Named Type Parameters . . . 25
3.4.5 Topic: Representing Inferred Type Argument . . . 26
3.4.6 Topic: Target-typed Inference . . . 26
3.4.7 Topic: Type Inference Based on Constrains . . . 26
3.4.8 Topic: Inferred Method Return Type . . . 27
3.4.9 Topic: Realocation . . . 27
3.4.10 Topic: Constructor Type Inference . . . 27

4 Problem Analysis 28
4.1 Scope . . . 28
4.2 Motivation . . . 28
4.2.1 Weakness – Target-typing . . . 28
4.2.2 Weakness – Constraints-based Inference . . . 29

4.2.3 Weakness – All or Nothing Principle 29
4.2.4 Solution – Improved Method Type Inference 30
4.2.5 Solution – Partial Method Type Inference 31
4.2.6 Solution – Constructor Type Inference 32
4.3 Requirements 33
4.4 Summary 34

5 Language Feature Design **35**
5.1 Partial Method Type Inference 35
5.1.1 Syntax 36
5.1.2 Method and Typename Lookup 41
5.2 Method Type Inference Algorithm Change 42
5.2.1 New Definitions 42
5.2.2 Algorithm Phases 43
5.2.3 Collecting Type Bounds 44
5.2.4 Fixation 46
5.3 Partial Constructor Type Inference 46
5.3.1 Syntax 47
5.3.2 Typename Lookup 47
5.3.3 Argument Binding 48
5.3.4 Type Inference Algorithm 49
5.3.5 Initiliazer Extension 50
5.3.6 Diamond Operator Extension 51
5.4 Partial Type Inference During Dynamic
Member Invocation 52
5.5 Other Type Inference Improvements 52
5.5.1 Shared Type Inference Context 52
5.5.2 Inferring Return Value of Methods 53

6 Solution **55**
6.1 Proposal 55
6.1.1 Creation Process 55
6.1.2 Content 55
6.2 Implementation 56
6.2.1 Development Environment 56
6.2.2 Repository Overview 59
6.2.3 Test Suite 61
6.2.4 Parsing Inferred Type Arguments 61
6.2.5 TypeInferrer 62
6.2.6 Binding Partial-inferred Method 62
6.2.7 Binding Partial-inferred Object Construction 62
6.2.8 Nullable Walker 65

7 Evaluation **66**
7.1 Tests 66
7.2 Demo Examples 66
7.2.1 Basics 66
7.2.2 Advanced Examples 68
7.3 LDM Meeting Summary 71

8 Future Work **72**

Conclusion **73**

Bibliography **74**

A Attachments **77**

A.1 Build and run Demo 77

A.2 Build your applications 77

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 a 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 a 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 [32] programming language, which belongs to strongly typed 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 [13] 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 [14], where it can deduce types of all variables in an entirely untyped code. The power of type inference is caused by the 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 the official C# compiler, Roslyn [31], which is an open-source project managed by Microsoft. The prototype is used to explore potential implementation issues which the improvement can cause, and to help 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 [35], where everyone can add his/her ideas or comment on others' ideas. It is preferred to use a predefined template [26] 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 [5] changes, and other possible alternatives. The process of language proposal ends with LDT accepting or declining it. 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.

1.3 Summary

We summarize the 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 a 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*, 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 [9], it allows to assign the `null` value to nullable

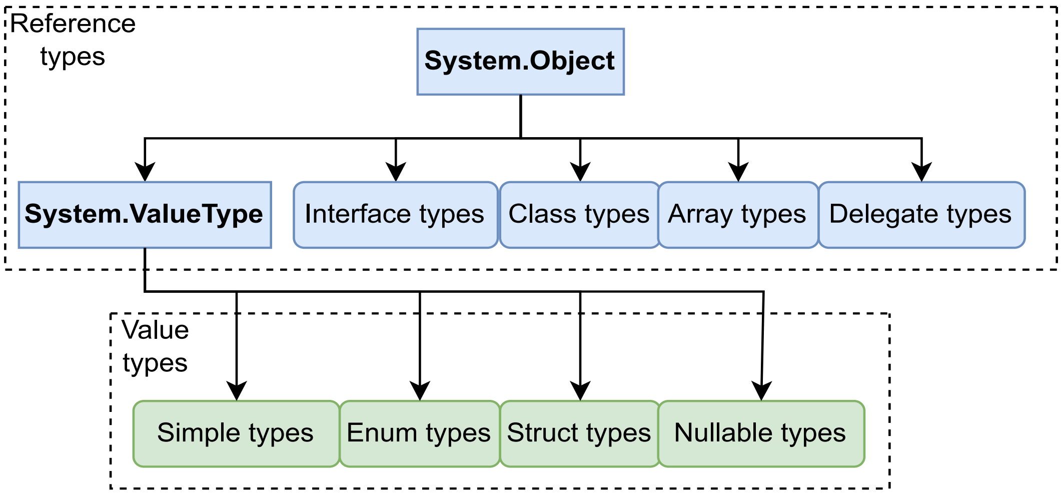

Figure 2.1: The C# data types schema adjusted from a C# blog [2].

types, which are equivalents of the rest of value types prohibiting it. Because assigning the `null` value is referred to as a billion-dollar mistake [36], C# 8.0 [9], 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 the type argument can be an 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 [6]. 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 the `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 the `"text"` value has the `string` type, and it is passed as the `p1` parameter, which has the `int` type.

```
void Bar<T>(int p1, T p2, long p3) { ... }
...
dynamic a = "string";
a.Foo();
Bar("text", 1, a); // Compilation error reported
```

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 the 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 the 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 a 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 the `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 an 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 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 the 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 the 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 `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.

```
Foo<T1, T2, T3>(T1 p1, T1 p2, IList<T2> p3, Func<T2, T3> p4)
{ ... }
...
List<int> myList = ...
Foo((long)1, (int)1, myList, (p1) => p1 + 1);
```

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 element type of the constructing array. The C# specification calls it *common type inference*, which finds the most specialized common type between given types. In this case, it is the `object` type. 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.

```
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 method type inference algorithm, this section presents its description. The thesis doesn't show the complete algorithm described in the C# specification [7] 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, whose 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 (also called *inferred type arguments*), and the second phase repeats until all type variables are fixed or fails 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

```
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

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. 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 the 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 anonymous function type 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.

```
fn InferExplicitParameterType(Argument E, Type T):
  if (E.isExplicitTypedAnonymousFunc && T.isDelegateOrExprTreeType
    && E.paramTypes.size == T.paramTypes.size)
      E.paramTypes.zip(T.paramTypes)
                  .foreach((e, t) → InferExact(e, t))

fn InferOutputType(Argument E, Type T):
  switch(E) {
    E.isAnonymousFunc && $T$.isDelegateOrExprTreeType →
      InferLower(InferReturnType(E), T.returnType)
    E.isMethodGroup && T.isDelegateOrExprTreeType → {
        $E_{resolved}$= OverloadResolution(E, T.parameterTypes)
        if ($E_{resolved}$.size == 1)
          InferLower($E_{resolved}$[0].returnType, T.returnType)
      }
    E.isExpression && E.getType() is Type u → InferLower(u, T)
  }
```

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

2.4.3 Collecting Type Bounds

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.

```
fn InferExact(Type U, Type V):
  if (Type t = X.find(x → V == x && F[x.idx] == null))
    $B_{exact}$[t.idx].add(U)
  switch(V) {
    V isLike '$V_1$[..]' && U isLike '$U_1$[..]' && V.rank == U.rank →
      InferExact($U_1$, $V_1$)
    V isLike '$V_1$?' && U isLike '$U_1$' → InferExact($V_1$, $U_1$)
    V isLike 'C<$V_1, ..., V_e$>' && U isLike 'C<$U_1, ..., U_e$>' →
      V.typeArgs.zip(U.typeArgs)
                .foreach((v, u) → InferExact(v, u))
  }

fn InferLower(Type U, Type V):
  if (Type t = X.find(x → V == x && F[x.idx] == null))
    $B_{lower}[t.idx]$.add(U)
  switch { .... }

fn InferUpper(Type U, Type V):
  if (Type t = X.find(x → V == x && F[x.idx] == null))
    $B_{upper}[t.idx]$.add(U)
  switch { ... }
```

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

Observation 1. *There is no need to check possible contradictions between already collected bounds and the currently adding bound because It is 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`.

```
fn Fix(TypeParameter x):
  $U_{candidates}$ = $B_{lower}$[x.idx] + $B_{upper}$[x.idx] + $B_{exact}$[x.idx]
  $B_{exact}$[x.idx].foreach{b $\rightarrow$
    $U_{candidates}$.removeAll(u -> !b.isIdenticalTo(u))}
  $B_{lower}$[x.idx].foreach{b $\rightarrow$
    $U_{candidates}$.removeAll(u -> !hasImplicitConversion(b, u))}
  $B_{upper}$[x.idx].foreach{b $\rightarrow$
    $U_{candidates}$.removeAll(u -> !hasImplicitConversion(u, b))}
  temp = $U_{candidates}$.filter(x $\rightarrow$ $U_{candidates}$.all(y $\rightarrow$
    hasImplicitConversion(y, x)))
  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, a 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 Language Design Team (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 [31]. 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 [28] which consists of four phases highlighted with different colors.

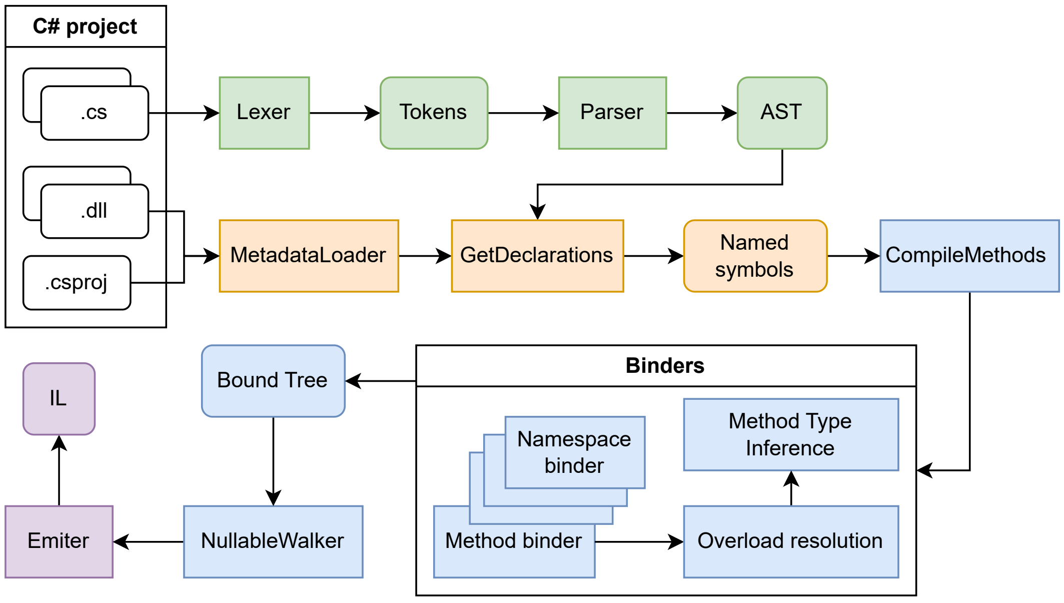

Figure 3.1: Roslyn architecture

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 the 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 the `.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, whose 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. The 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 the 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 the 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, the 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, unbound method arguments requiring the target type (for example already mentioned the target-typed `new()` operator) are bound using the corresponding parameter type.

The method type inference can occur for the second time if the previously mentioned nullability analysis is turned on. The nullability analysis is a kind of a flow analysis that uses a bound tree to check and rewrite already created bound nodes according to nullability. Because overloading and the 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 [14] 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 [39]. An expression is either a variable (3.1), a function application (3.2), a lambda function (3.3), or a *let-in* clause (3.4).

$$\begin{aligned} e &= x && (3.1)\\ &\mid e_1 e_2 && (3.2)\\ &\mid \lambda x \rightarrow e && (3.3)\\ &\mid \textbf{let } x = e_1 \textbf{ in } e_2 && (3.4) \end{aligned}$$

The above-mentioned expressions have one of two kinds of types. The *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 the $\rightarrow$ symbol taking two type parameters which represents a lambda function type. The second kind is the *Poly* type, which is an arbitrary type with possible preceding the $\forall$ operator 3.8, bounding its type variables.

$$\begin{aligned} mono\ \tau &= \alpha && (3.5)\\ &\mid C\ \tau_1, ..., \tau_n && (3.6)\\ poly\ \sigma &= \tau && (3.7)\\ &\mid \forall \alpha\ .\ \sigma && (3.8) \end{aligned}$$

A context(represented by the Γ symbol) contains bindings of an expression to its type which are described by pairs of an 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.

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

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

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

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

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

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

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 Figure 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 whose 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.

We can notice that the unification part of this algorithm is similar to the method type inference mentioned in the C# section 2.4 where the substitution represents inferred type arguments of the method whose parameter types were unified with corresponding argument types. The same concept of the unification is also used in Rust type inference.

```
fn Infer($\Gamma$, expr):
  switch(expr):
    expr isLike 'x' $\rightarrow$ return ({}, Instantiate(expr))
    expr isLike '$\lambda x \rightarrow e$' $\rightarrow$
      $\beta$ = NewVar()
      ($S_1$, $\tau_1$) = Infer($\Gamma$ + x: $\beta$, e)
      return ($S_1$, $S_1\beta$ $\rightarrow$ $\tau_1$)
    expr isLike '$e_1 e_2$' $\rightarrow$
      ($S_1$, $\tau_1$) = Infer($\Gamma$, $e_1$)
      ($S_2$, $\tau_2$) = Infer($S_1\Gamma$, $e_2$)
      $\beta$ = NewVar()
      ($S_3$, $\tau_1$) = Unify($S_2\tau_1$, $\tau_2$ $\rightarrow$ $\beta$)
      return ($S_3 S_2 S_1$, $S_3\beta$)
    expr isLike 'let x = $e_1$ in $e_2$' $\rightarrow$
      ($S_1$, $\tau_1$) = Infer($\Gamma$, $e_1$)
      ($S_2$, $\tau_2$) = Infer($\Gamma$ + x: Generalize($S_1\Gamma$, $\tau_1$), $e_2$)
      return ($S_2 S_1$, $\tau_2$)
```

Figure 3.2: W algorithm

3.2.1 H-M Extensions

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 [20], 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 [1]. 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 Mozilla and an open community created for performance and memory safety without a garbage collection. Besides its specific features like traits or variable regions, it also has advanced type inference [33], which is now described in a high-level perspective to get an inspiration for the proposed C# improvement.

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 whose 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 the `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.

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

Figure 3.3: Rust type inference example.

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 of how the context is gradually filled by type variables' constraints deduced from the statements. 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

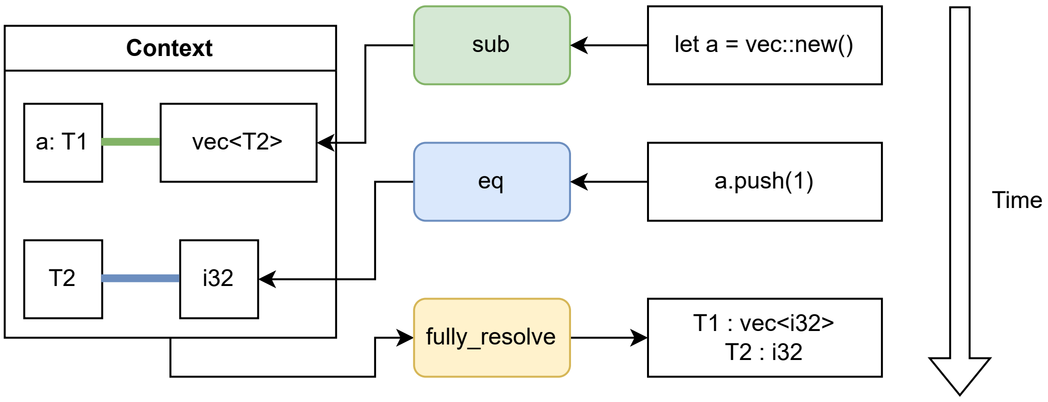

Figure 3.4: Rust type inference

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 implementation detail 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 mentioned overloading can cause 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 Language Design GitHub Issues

Some ideas for type inference improvements have already been introduced in discussions in the C# language repository [35], which will be used as inspiration for the thesis's proposed improvement. However, before describing the related ideas, a process of proposing new C# language features is mentioned to better understand how the ideas and final language changes are proceeded.

3.4.1 Design Process and Championed Issue

The process of designing a new language feature starts with publishing an idea into discussions [3], 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 a championed issue [4] 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 discussion topics directly or indirectly mentioned in the issue which partially suggest a possible solution.

3.4.2 Topic: Default Type Parameters

One of the discussion [11] 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 the `int` type as a default value.

```
class A<T = int> {}
...
var temp = new A();
```

Figure 3.5: Default type parameters.

3.4.3 Topic: Generic Aliases

Another discussion [12] 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. Figure 3.6 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.

```
using StringDictionary<TValue> = Dictionary<String, TValue>;
...
var temp = new StringDictionary<int>();
```

Figure 3.6: Generic aliases.

3.4.4 Topic: Named Type Parameters

The discussion [17] 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 a user provides a type argument by the 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 a user provides type arguments, U in this case, and letting the compiler infer the rest of the type parameters.

```
U F<T, U>(T t) { ... }
...
var x = F<U:short>(1);
```

Figure 3.7: Named type parameters.

3.4.5 Topic: Representing Inferred Type Argument

Comments of the mentioned championed issue [4] 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 a user just specifies the second type argument, which can't be inferred by the compiler. The comments propose other options for keywords like nothing, underscore, or whitespaces.

```
TResult Foo<T, TResult>(T p1) { ... }
...
Foo<var, int>("string");
```

Figure 3.8: Using char as inferred type argument.

3.4.6 Topic: Target-typed Inference

The discussion [27] 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 the `int` type and uses that to deduce the type argument `T`.

```
T Field<T>(this object target, string fieldName) { ... }
...
object row = ...
int id = Field(row, "id")
```

Figure 3.9: Target-typed inference.

3.4.7 Topic: Type Inference Based on Constrains

The next idea of improving type inference is given by the discussion [37], 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 the inferred type of `T2` forming the inferred type `List<int>`.

```
T1 Foo<T1,T2>(T2 item) where T1 : List<T2> {}
...
var temp = Foo(1);
```

Figure 3.10: Type inference based on type constraints.

3.4.8 Topic: Inferred Method Return Type

The discussion [38] mentions type inference of the 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.

3.4.9 Topic: Realocation

The issue [34] 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 the `A<T1>` type and the `Foo<T2>` method are merged, and the type arguments are split by a semicolon.

```
static class A<T1> {
  public static void Foo<T2>() {}
}
...
A.Foo<int;string>();
```

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

3.4.10 Topic: Constructor Type Inference

The discussion [10], 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.

```
class C<T> { public C(T p1) {} }
...
var temp = new C<_>(1); // T = int
```

Figure 3.13: Constructor type inference.

4. Problem Analysis

The previous section 3.4.1 introduced the championed issue, accompanied by several ideas for the improvement. Since the description of the issue is not well defined, the thesis 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 the method type inference 3.4.1, 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 thesis. 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.

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#, the method type inference is the only type inference that infers type arguments. Even though the method type inference is a complex algorithm, it has several weaknesses. The following three real-world examples demonstrate common issues with the method type inference, which the thesis will try to solve.

4.2.1 Weakness – Target-typing

The first weakness regards the target typing, which was mentioned in the previous chapter 3.4.6. 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 is also 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 a parameter 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.

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

Figure 4.1: No target-typed inference.

4.2.2 Weakness – 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 a 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

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

Figure 4.2: No constraints-based inference.

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 because the constraint gives a lower bound of the first type argument and the second type argument can be inferred from the first argument of the method.

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.

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

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 backward compatibility which was mentioned in the C# discussion [8]. New compiler versions should be backward 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 the method type inference starts to consider target types.

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

Figure 4.4: Breaking change: Target-typed inference.

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.

Figure 4.5 shows a similar situation when the method type inference starts to consider type parameter constraints.

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

Figure 4.5: Breaking change: Constraints-based inference.

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.

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, the 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

The partial method 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 the partial method type inference applied in the second presented example regarding method type inference weaknesses. Although the first type argument of the `DoTest` 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 to infer the rest.

```
void DoTest<T, U>(U data) where T : TestCaseDefault<U> { ... }
...
DoTest<TestCaseDefault<MyData>, _>(new MyData());
```

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

The third motivation example confirms that the partial method 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 the partial method 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.

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

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

4.2.6 Solution – Constructor Type Inference

The partial type inference doesn't regard only the partial 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 type with the `int` type argument, despite the fact that it could be inferred using the method type inference algorithm adjusted to be used in the object creation expression binding.

```
class Wrapper<T> : IWrapper { public Wrapper(T item) { ... } }
...
IWrapper a = new Wrapper<int>(1);
```

Figure 4.8: C# wrapper class.

Generally, the object creation can be considered a special case of a method call with a side effect(creating the object), which already has the 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

```
static Wrapper<T> Create<T>(T item) => new Wrapper<T>(item);
...
IWrapper a = Create(1);
```

Figure 4.9: Workaround of constructor type inference.

be inferred, it allows the use of the method type inference for inferring type arguments of construing type. However, this solution has disadvantages like the necessary boiler-plate and a prohibition of using initializers.

A possible solution would be to use the 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 the construing type are not limited by not introducing breaking changes since there is no type inference at all now. 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 type parameters are not used in the constructor.

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

Figure 4.10: Use case using type parameter constraints.

Because of that, extending the potential method type inference algorithm 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 the method type inference algorithm 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 an inferred type argument. The inference uses the parameter type of the constructor to infer the T parameter type which is int.

```
class Wrapper<T> : IWrapper { public Wrapper(T item) { ... } }
...
IWrapper a = new Wrapper<_>(1);
```

Figure 4.11: Constructor type inference: Wrapper.

Figure 4.12 shows a potential extension of the type inference. There is the Algorithm class definition containing two type parameters representing the type of data and logger used by the class. The first statement of initializing the alg variable uses type inference, leveraging the TLogger's constraint to determine its type. Now imagine that there is the SpecialLogger class that is intended to be used as a logger. The second statement demonstrates the possibility of having a nested underscore, which allows to specify the type of logger without providing its type argument.

```
class Algorithm<TData, TLogger> where TLogger : Logger<TData>
{ public Algorithm(TData data) { ... } }
...
var alg = new Algorithm<_, _>(new MyData());
var algWithSpecialLogger =
  new Algorithm<_ , SpecialLogger<_>>(new MyData());
```

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

The following requirements should be fulfilled by the solution to make it more likely to be discussed by LDT.

Backward 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 backward 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 type named record. 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 the 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 the type variable declaration or casting to a different type. The 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.

4.4 Summary

Three possible improvements of type inference were introduced based on the mentioned motivation, where was identified main weaknesses of the current type inference. Then, several requirements were created to achieve the fourth goal of the thesis, which is to make the proposal likely to be discussed by LDT. Based on the requirements, the first improvement introducing breaking change was excluded. The remaining two improvements seem to fulfill the requirements, and they will be further proceeded in the following section with focusing on not breaking any mentioned requirements.

5. Language Feature Design

The previous chapter determined two improvement directions that would be beneficial to explore. This chapter continues with exploring these directions consisting of the partial method type inference and the constructor type inference. The goal of this exploration is to precisely design the improvements that are used afterward as the content of the mentioned proposal for LDM and implemented in Roslyn.

We take the language change suggestions, mentioned in the Langauge Design GitHub Issues section 3.4, as an inspiration for the improvements and divide them into three following groups. Suggestions in the first group, given below, regard improving the method type inference algorithm and are used in the constructor type inference mentioned later.

1. *Target-typed inference* mentioned in Section 3.4.6
2. *Type inference based on constraints* mentioned in Section 3.4.7

The following second group consists of suggestions describing general partial type inference which relate to both improvements.

1. *Default type parameters* mentioned in Section 3.4.2
2. *Generic aliases* mentioned in Section 3.4.3
3. *Named type parameters* mentioned in Section 3.4.4
4. *Inferred type argument* mentioned in Section 3.4.5
5. *Relocation* mentioned in Section 3.4.9

The third group contains suggestions introducing type inference in new C# constructs. The first suggestion describes the second improvement which is explored further and the second suggestion is mentioned in the last section of this chapter as a potential future improvement that will be difficult to implement and which is not in the scope of this thesis.

1. *Constructor type inference* mentioned in Section 3.4.10
2. *Inferred method return type* mentioned in Section 3.4.8

5.1 Partial Method Type Inference

The partial method type inference occurs in the second group where various techniques are used to allow specify only some type arguments in the generic method call. These techniques are discussed below.

The disadvantage of *default type parameters* 3.4.2 is that it requires to provide the defaults in the type parameter declarations. It is problematic since the old code has to be changed to allow skipping the type arguments which have defaults. For this reason, the thesis excludes this suggestion.

Generic aliases 3.4.3 don't work for methods. There could be something like a generic alias for a method using a similar syntax as for the type which would allow to give defaults for some type parameters. However, it would still need additional declarations to provide these aliases and we don't think that it is flexible enough since it requires a new declaration for each method call scenario, where is beneficial to omit a type argument.

Named type parameters 3.4.4 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.

The suggestion of representing *inferred type arguments* 3.4.5 looks promising. It is already used in different languages, like a star symbol in Kolin and Java or an underscore in Rust and F#. So, it is more common and intentional than other mentioned suggestions. 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. This suggestion is a core of the partial method type inference improvement.

Relocation 3.4.9 doesn't solve the problem of specifying all type arguments. It just compacts type argument lists into one.

5.1.1 Syntax

The syntax has to be chosen to allow a user to represent type arguments that have to be inferred by the compiler. The choice of the syntax should be based on the following conditions:

1. the usage should be intuitive for a user,
2. a future extension of type inference should reuse the syntax, and
3. it shouldn't introduce the breaking change.

We identified five use cases that use the syntax in different situations to help reasoning about advantages and disadvantages of different syntax alternatives which are presented in the mentioned GitHub issues and discussions. Table 5.1 shows the use cases and their examples in the current C# language version.

Use case	**Example**
Type argumens of generic method	`Foo<Type1, ...>(...)`
Type arguments of generic type	`new Bar<Type1, ...>(...)`
Variable declaration	`int temp = ...`
Array type	`int[]`
Nullable type	`int?`

Table 5.1: Use cases helping to choose the syntax.

Type argumens of generic method use case represents a situation where a user wants to skip type arguments which can be inferred by the compiler and wants to specify just the ambiguous type arguments.

Type arguments of generic type use case represents a situation where a user wants to infer either all type arguments of the generic type or to specify ambiguous type arguments and let the compiler infer the rest. The syntax doesn't have to be bound to the shown object creation expression example, but it can be also used on other potential places like a variable declaration, type casting, or a type argument of a generic method. The description of the syntax behavior in other places than an object creation expression and a generic method call is not in the scope of this thesis. However, the syntax is considered since it is a natural future continuation to use type inference on new places in the language.

Variable declaration use case represents a situation where a user wants to infer a type of variable that is declared. This use case already has the `var` syntax and reminds the secondary ability of the `var` keyword representing an inferred type of the declared variable. Because of that, the new syntax shouldn't be allowed to be used in this situation.

Array type use case represents a situation where a user should be able to use the syntax to represent an array type whose element type should be inferred by the compiler.

Nullable type use case represents a situation where a user should be able to hint to the compiler a nullability of a type that should be inferred.

We continue with presenting syntax alternatives found in the mentioned GitHub discussions and comparing their advantages and disadvantages in the use cases mentioned above.

Diamond Operator

The diamond operator syntax consists of a pair of two angle brackets `<>`.

In the context of the first use case, the syntax is used after the generic method name instead of the type argument list. The semantics of the operator is to determine that the method name is generic so the compiler will consider only generic methods during overload resolution. Figure 5.1 shows an example of the syntax used in the `Foo` method call. Since it doesn't allow to skip inferable type arguments, we don't see it as a good candidate for the feature.

```
Foo<>(arg1, ...);
```

Figure 5.1: Type argumens of generic method use case.

In the context of the second use case, the syntax is used after the generic type name instead of the type argument list. The semantics of the operator is to determine that the type is generic so the compiler will look up only generic types. Figure 5.2 shows the usage in the object creation expression where the type arguments of the `Bar` type should be inferred by the compiler. As we can

```
class Bar { ... }
class Bar<T1> { ... }
class Bar<T1, T2> { ... }
...
new Bar<>(...);
```

Figure 5.2: Type arguments of generic type – ambiguity.

notice, it is not clear, where the compiler should find constructors for overload resolution since the Bar<> syntax can represent multiple types differing in type parameter's arity. Probing all options for the constructors seems complicated because it requires a significant change in the Roslyn implementation and it will influence the compile time because of the increased overloading resolution complexity.

On the other hand, it is common to have just one type without multiple variations of type parameter's arity in the current name scope. Figure 5.3 shows an example where an object representing a dictionary is created. The compiler can check if there is just one generic type with this name in the current scope. If it is, the problem with ambiguity disappears and the compiler can try to infer the type arguments. We think that it is a beneficial usage since it allows to turn on type inference of object creation expression. Extending the current overload resolution during object creation expression binding to consider generic types when the type doesn't contain the type argument list can't be used instead of it since it would cause a breaking change. The syntax can be used in other places too, however, we don't see any practical usage for that.

```
new Dictionary<>(...);
```

Figure 5.3: Diamond operator – object creation expression.

The third use case is not considered since there is already the var keyword.

In the context of the fourth use case, the syntax determines that a user wants to infer an element of an array. Figure 5.4 shows an example, where there is the temp variable declaration of the array type whose element should be inferred. We think, that the meaning of this syntax is not clear to a user in comparison to the previous use case where it reminds the type argument list.

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

Figure 5.4: Diamond operator – array type.

In the context of the fifth use case, the syntax should represent a nullable type which is inferred by the compiler. Figure 5.5 shows an example, where a user wants to infer the type of the declared temp variable and hints to the compiler the nullability of that type. We don't think that the usage is intuitive for that and we exclude it.

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

Figure 5.5: Diamond operator – inferred nullable type.

We think that the diamond operator would be beneficial only in the second use case where it turns on the type inference of object creation expression. The symbol already means a generic type with one type argument in the typeof(Bar<>) expression which has a conflict with the new semantics in the context of object creation expression. However, we think that the context of reflection and object creation expression are distinct enough to use the symbol for a different meaning as we can see in other language constructs. We think that the advantage of the syntax in this use case is that it still reminds the type argument list and it is not verbose.

Commas Separating Nothing

In the context of the first use case, the syntax omits type arguments that should be inferred by the compiler and specifies just ambiguous type arguments using commas as a separator. Figure 5.6 shows an example where the syntax is used to call the `Foo` method with three type arguments. The first and third type argument are inferred by the compiler and the second type argument is specified by a user as the `int` type. There are the following advantages of this syntax. It

```
Foo<, int,>(arg1, ...);
```

Figure 5.6: Whitespace – generic method call.

determines the arity of the called generic method. The syntax is already used in the `typeof()` operator where commas are used to determine the arity of the reflected type. It allows specifying just ambiguous types. Although the syntax is a good candidate, we think that skipping the type arguments by nothing makes the code comprehension of the code worse since it makes it difficult to determine the position of the type arguments at first sight.

The syntax has the same use and meaning in the second use case.

In the context of the fourth use case, the syntax omits the type of a array. Figure 5.7 shows the syntax where the first statement means a declaration of the `temp` variable which has a type of array whose element is inferred by the compiler. The second statement uses the syntax to hint to the compiler that the second type argument should be an array type. We think that the usage in this use case is unintuitive.

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

Figure 5.7: Whitespace – array type.

In the fifth use case, the syntax makes it difficult to append the `?` mark since it doesn't have any placeholder. There is the same problem as mentioned above.

Although the syntax is a good candidate in the first and second use cases, it is not suitable for the array types and nullable types. We consider this disadvantage important and exclude the syntax.

Commas Separating Underscores

The syntax uses an underscore to represent a type that has to be inferred by the compiler. The syntax is commonly used in other programming languages like F# or Haskell to represent inferred type arguments which is considered as an advantage. Although it has different semantics in C# where it can also mean a discarded variable. We don't think it makes the usage of that unintuitive or confusing in the following use cases.

The usage of the syntax in the first use case is similar to the previous syntax where omitting the type arguments is done by using the underscore. Figure 5.8 shows the same `Foo` method using underscores to represent the first and third type argument as an inferred argument. We think that the underscore placeholder

```
Foo<_, int, _>(arg1, ...);
```

Figure 5.8: Underscore – generic method.

helps to identify type argument positions in comparison with the previous syntax variant.

The syntax has the same use and meaning in the second use case.

In the context of the fourth use case, the syntax seems to be more intuitive than in the previous syntax variant. Figure 5.9 shows the declaration of the `temp` variable whose type is an array with the inferred element's type.

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

Figure 5.9: Underscore – array type.

The fifth use case is shown in Figure 5.10 where it uses the _? syntax to express that the inferred type of the temp variable has to be nullable.

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

Figure 5.10: Underscore – nullable.

A disadvantage is the introduction of breaking change because C# allows the underscore as a type identifier. However, we don't think that it is a blocker since a similar change was done with the discards and the potential change behavior is easily discoverable.

Commas Separating `var` Keywords

The `var` keyword representing the placeholder is another natural option of the syntax. The usage and meaning of the syntax is similar to the previous syntax variant except for replacing the underscore with the `var` keyword. Figure 5.11 shows the usage of `var` in the generic method call where the first and third type argument has to be inferred by the compiler. An advantage is the already used

```
Foo<var, int, var>(arg1, ...);
```

Figure 5.11: `var` – generic method.

`var` keyword in a variable declaration where the secondary meaning of it is that it represents an inferred type. On the other hand, it also means that a variable declaration which can be confusing. Another disadvantage is the syntax size which seems to be too long for this purpose. We exclude this variant based on these several issues.

Commas Separating "Something Else"

A different placeholder for the inferred type arguments doesn't make a lot of sense because we think that it would be less intuitive than already mentioned syntax variants.

Conlusion

The thesis chooses the underscore as a placeholder for the inferred type argument since the meaning of this character is related to the intention. Table 5.2 reuses Table 5.1 to show the usage of the underscore placeholder. The first two examples use the underscore placeholder in the type argument list. We can notice that the underscore placeholder can be nested and it can be used as an inferred type of array's element. The question mark can be also used with the underscore to represent a nullable type which has to be inferred by the compiler. We prohibit to use the underscore placeholder in the variable declaration since the `var` keyword should be used for this purpose.

Use case	**Example**
Type argumens of generic method	`Foo<_, Baz<_, _[]>, _?>(...)`
Type arguments of generic type	`new Bar<_, Baz<_, _[]>, _?>(...)`
Variable declaration	Not applicable
Array type	`_[]`
Nullable type	`_?`

Table 5.2: Use cases utilizing the underscore placeholder.

We think that the underscore is the shortest and synoptical way to skip inferred type arguments based on the exploration of syntax alternatives given above. 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. Although the diamond operator is not very useful in a generic method call, it makes sense in an object creation expression. The potential usage and analysis of that is covered later in this thesis.

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 for the compiler.

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. A change is in the 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 can be represent 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 5.12 shows an example of using the nullable operator. The call of the `Foo` generic method has three type arguments where the first 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 5.12: Inferring nullable type argument.

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

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

5.2 Method Type Inference Algorithm Change

The thesis extends the method type inference 2.4 by introducing new type variables, 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 5.13 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<_, _>` is considered as a bound for the type variable, which has to be respected. However, this bound contains other type variables which doesn't have to be known yet. So, the algorithm has to first fix the type variables contained in that bound and then infer the `T2` type variable. The second observation for this extension is a way of collecting 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 5.13. The `Dictionary<_, _>` is a bound of the `T2` type variable. Its second bound is a type of the `p2` argument. This type gives us bounds for type variables contained in the `Dictionary<_, _>` bound, which have to be propagated.

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

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

Figure 5.13: Partially inferred method call.

5.2.1 New Definitions

The proposed algorithm uses three new definitions which are given below.

Definition 5 (Inferred type argument)**.** *Inferred type argument is a type argument which is inferred by the compiler and is represented as the _ symbol in a type argument list.*

Definition 6 (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 X_e is contained in shape bound of the type variable X_i. X_i* shape-depends on *X_e if X_i* shape-depends directly on *X_e 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 7 (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_i* type-depends on *X_e if X_i* type-depends directly on *X_e 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 5.14 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. 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 `string?` type as a type argument. When the compiler treats nullability, it should infer the nullable type(not `string`). Imagine that the provided type argument would be added as an exact bound. It can happen that the type variable will contain another exact bound, which will be the non-nullable version of the type. The current implementation will infer the `string` type although it should fail since these types are not equivalent in nullable-aware code. The behavior of the exact bound can't be changed because of breaking changes, so we have to add a new bound to reflect this need.

`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. Additionally to allow type variable fixing, at least one bound mustn't not contain an unfixed type variable. This requirement ensures at least one type candidate, which can be the inferred type argument.

```
Input: method call M<$S_1$,...$S_n$>($E_1$,...$E_x$) and
       its signature $T_e$ M<$X_1$,...,$X_n$>($T_1$ $p_1$,...,$T_x$ $p_x$)
Output: inferred $X_1$,...$X_n$,...$X_{n+l}$
$B_{lower}$ = $B_{upper}$ = $B_{exact}$ = $B_{shape}$ =F = []
$TV$ = getAllTypeVariables(X, S)
FirstPhase()
SecondPhase()

fn FirstPhase():
  S.foreach(s -> InferShapeBound(s, T[s.idx]))
  ...here continue as the former method type inference

fn SecondPhase():
  while (true):
    $TV_{indep}$ = $TV$.filter(x $\rightarrow$
      F[x.idx] == null && $TV$.any(x $\rightarrow$
        dependsOn(x, y) && shapeDependsOn(x, y)
        && typeDependsOn(x, y)
      )
    )
    $TV_{dep}$ = $TV$.filter(x $\rightarrow$
      F[x.idx] == null && $TV$.any(y $\rightarrow$
        (dependsOn(y, x) || shapeDependsOn(y, x)
          || typeDependsOn(y, x))
        && !$TV$.any(t $\rightarrow$ shapeDependsOn(x, t))
        && ($B_{lower}$+$B_{upper}$+$B_{exact}$+$B_{shape}$).any(b $\rightarrow$
            !b.containsUnfixedTypeVariable
        )
      )
    )
    ...here continue as the former method type inference
```

Figure 5.14: Phases of new Method Type Inference

5.2.3 Collecting Type Bounds

Figure 5.15 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 all bounds which are associated with it. Using mentioned example 5.13, this phase propagates the `int` type bounds contained in the `Dictionary<int, int>` type of the 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.

```
fn Propagate(Type U, int typeVariable) {
  setOf($B_{shape}$[typeVariable],
    $B_{lower}$[typeVariable],
    $B_{upper}$[typeVariable],
    $B_{exact}$[typeVariable]
  ).foreach(b $\rightarrow$
    if (b.containsUnfixedTypeVariable) InferHelper(U, b)
    if (U.containsUnfixedTypeVariable) InferHelper(b, U)
  )
}

fn InferExact(Type U, Type V):
  if (TypeVariable t = TV.find(x $\rightarrow$ V == x && F[x.idx] == null) &&
    !$B_{exact}$[t.idx].contains(U)) {
    $B_{exact}$[t.idx].add(U)
    Propagate(U, t.idx)
  }
  ...here continue as the former method type inference

fn InferLower(Type U, Type V):
  if (TypeVariable t = TV.find(x $\rightarrow$ V == x && F[x.idx] == null) &&
    !$B_{lower}$[t.idx].contains(U)) {
    $B_{lower}$[t.idx].add(U)
    Propagate(U, t.idx)
  }
  ...here continue as the former method type inference

fn InferUpper(Type U, Type V):
  if (TypeVariable t = TV.find(x $\rightarrow$ V == x && F[x.idx] == null) &&
    !$B_{upper}$[t.idx].contains(U)) {
    $B_{upper}$[t.idx].add(U)
    Propagate(U, t.idx)
  }
  ...here continue as the former method type inference

fn InferShape(Type U, Type V):
  if (TypeVariable t = TV.find(x $\rightarrow$ V == x && F[x.idx] == null)) {
    $B_{shape}$[t.idx] = U
    Propagate(U, t.idx)
  }
```

Figure 5.15: *Exact inference*, *Upper-bound inference*, *Lower-bound inference*, *Shape-bound inference*

Table 5.3 shows which inference is called based on the `InferHelper`'s 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 the 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 5.3: Matrix of `InferHelper` function.

5.2.4 Fixation

Figure 5.16 shows the last part of the changed algorithm, the 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.

```
fn Fix(TypeVariable x):
  $U_{candidates}$ =
  if ($B_{shape}$[x.idx] != null)
    setOf($B_{shape}$[x.idx])
  else
    ($B_{lower}$[x.idx] + $B_{upper}$[x.idx] + $B_{exact}$[x.idx]).filter(b →
        !b.containsUnfixedTypeVariable
    )
  ...here continue as the former method type inference
```

Figure 5.16: Fixing of type variables

Observation 3. *We want to make sure 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

The second improvement explored by the thesis is the *constructor type inference* mentioned in the discussion [10]. In addition to the constructor type inference, we present partial constructor type inference using underscore placeholders to represent inferred type arguments in the same way as in the partial method type inference.

Since the potential suggestions from GitHub discussions can be applied to the partial constructor type inference, we discuss them again in relation to this improvement. Problems regarding using *default type parameters*, *generic aliases*, *named type parameters*, or *relocation* mentioned in the partial method type inference persist, so either the partial constructor type inference is not 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 can be done by using *target-typed inference* and *type inference based on constraint* contained in the first group of divided suggestions presented in the introduction of this chapter. Target-typed inference is useful because an object creation is usually associated with assigning it to a target. Type inference based on type constraints provides a wider context for type inference allowing to infer the type arguments based on `where` clauses in the type declaration of the created object.

Besides this functionality, the text mentions two additional features that extends the context of type inference and makes the syntax less boilerplate. After a discussion with a member of LDT, the additional features are considered to be controversial. For this reason, the features is presented as a voluntary extension that can be removed from the core design. The first feature regards using type information from initializers, which is valuable when creating collections or objects with the object initializer syntax. The second feature uses 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 choice of syntax is the same as in the partial method type inference. Figure 5.17 shows a simple usage where the underscore represents the inferred type argument. This argument is deduced using a type received from the `t` variable declaration which determines the `int` type to be the inferred type argument.

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

Figure 5.17: Syntax of the partial constructor type inference.

Similar to the partial method type inference, the `?` operator is allowed to append to determine the type nullability of the inferred type argument.

5.3.2 Typename Lookup

The typename lookup is done in the same way as in the partial method type inference. The binding of the underscore identifier is skipped. The underscore represents an inferred type, which has to be resolved during type inference. The inferred arguments are allowed only in the 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 5.18, 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 5.18: Allowed inferred type arguments.

5.3.3 Argument Binding

The target-typed inference mentioned in Section 3.4.6 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 during the argument's binding yet if the method is generic. Figure 5.19 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 constructor's arguments to the constructor 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.

```
class Bar<T> {
    public Bar(Foo<T> arg1, T arg2) {}
}
class Foo<T> {}
...
new Bar<_>(new Foo<_>(), 1);
```

Figure 5.19: Target as an argument.

The need to postpone the argument binding relates to Section 2.3.2 regarding the target-typed `new()` operator. Roslyn binds 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 the already-known target type. The result of the conversion is the binding of the object creation expression, which type is the target type. Its 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 argument is not needed for the inference success(the corresponding parameter doesn't contain a type parameter), a similar unconverted bound element is created. Otherwise the argument is bound without the target type. After the overload resolution, when the types of parameters are known, and the necessary conversion is started to be created, the unconverted object creation expression is tried to be bound with the already known target type. The compiler has to be careful here. It has to bind the object creation expression with and without a target type no more than once to prevent exponential time of binding.

Figure 5.20 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

```
class Bar<T>{
    public Bar(Foo<T> arg1, T arg2){}
    public Bar(Foo<T> arg1, List<T> arg2){}
}
...
new Bar<_>(new Bar<_>(null, null), 1);
```

Figure 5.20: Potential exponential time of binding.

`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 finally resolve the object creation expression. This repeating failure would not be problem for the `new()` operator since it is cheap to express it automatically as an unconverted bound expression. However, it is a problem for binding the object creation expression which can take long time.

The following observation offers a way to avoid it. The arguments can be bound before the overload resolution without target types. The arguments' bindings, which will fail due to missing target type information will be bound after the overload resolution when a final used overload will be choose. This strategy binds each expression no more than twice.

5.3.4 Type Inference Algorithm

The previously mentioned partial method type algorithm 5.2 is unchanged except for collecting new bounds in the first phase of the algorithm. If a target type is provided, the upper bound type inference is made from it to the partially inferred type containing the constructor. If the inferring type contains the `where` clause containing type parameter constraints, for each of the type constraints which represent either inheritance constraints or interface implementation constraints is performed the lower bound type inference from it to the corresponding type parameter.

Figure 5.21 shows an example of the partial constructor type inference, where both of the type arguments of the `Bar` type are inferred. The algorithm will create four type variables. The first two variables will represent type arguments, and the second two variables will represent the underscores. In the first phase,

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

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

Figure 5.21: Example of type inferences.

the 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 the `T2` type parameter causes another lower bound type inference from the `object` type to the type variable representing the `T2` type parameter. Shape inference will relate type variables representing underscores with type variables representing type parameters. Then, the fixation will be made, and it will result in the `Bar<int, object>` inferred type.

5.3.5 Initiliazer Extension

Initializers can be a part of the object creation expression, which allows us to use it as another source of type information. As the thesis mentions in the previous chapter, initializers are a 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 5.22 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 would allow this inference by performing the lower bound type inference for each item in the initializer. The inference would be made from the type of assigning expression to the type of the field's type declaration.

```
class Bar<T> {
    public T Field;
}
...
new Bar<int>{ Field = 1 };
```

Figure 5.22: Object initializer.

A similar deduction can be made with array initializers where the element's type is inferred. The improvement would allow it by performing the lower bound type inference for each item in the initializer list. The inference would be 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 syntax 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 would 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 the time complexity disappears. So when the `Add` method is not overloaded, the first phase of the type inference algorithm would additionally make the lower bound inference for each item in the initializer. The inference would be made from the type of the expression to the type of method parameter.

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

5.3.6 Diamond Operator Extension

Since the partial 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 5.23 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, there are a lot of situations where the type-name represents only one type in the current scope. In the given example, the current scope contains only one generic definition for the `Directory` typename, so specifying the arity in the example is redundant in this case.

```
System.Collections.Generic;

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

Figure 5.23: Redundant 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 5.24 shows the usage of the diamond operator with the `Dictionary` type. Supposing that there is no other definition of `Dictionary` type with a different arity in the current name scope, the compiler can assume that the referring type is the `System.Collections.Generic.Dictionary<,>` type from the standard library.

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

Figure 5.24: Diamond operator.

5.4 Partial Type Inference During Dynamic Member Invocation

Inferred type arguments can appear in an expression containing a dynamic value which is still checked by the compiler in the limitted way mentioned in the C# Programming Langauge chapter 2.2. We have to adjust this check to reflect the described improvements. The compiler checks three relevant expression types 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 for applicability. If there is no candidate for which the test succeeds, 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 for the applicability.

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. It also announces a warning about using partial type inference in late-binding, which is not supported since it is handled by the runtime. There are situations where the runtime is able to infer type arguments even with used underscores in the type argument list. Figure 5.25 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.

```
void Foo<T1, T2>(T1 arg1, T2 arg2) {}
...
dynamic t = ...
Foo<_, _>(t, 1);
```

Figure 5.25: Runtime type inference.

5.5 Other Type Inference Improvements

Besides the proposed improvement described above, we 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 inference. The next big improvement of the type inference would be to provide global type inference using inferring contexts similar to Rust. 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. 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 thesis 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 5.26 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 5.26: Kotlin return type inference.

Although C# can infer the return type of an anonymous function, it can't infer the return type of a method definition. The main obstacle 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 architecture has two consequences.

The first consequence is that the compiler can start the methods' bodies compilation in parallel since a 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.

The second consequence is that it can't infer the method's return type because it requires the compiled method body. This is the difference between methods and anonymous functions which are expressions. So, the compiler can infer the return type of the anonymous function by compiling its body in the same phase, which is not the case in the method definition.

Kotlin divides the method compilation into two groups, as it is described in the video [19]. The first group contains methods without a return type, which is compiled first in a single thread. This will allow to obtain all signatures for these 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 it. 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 the 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 by himself/herself.

6. Solution

The solution consists of a proposal describing the language feature given in the previous chapter 5 and an implementation of the prototype in a separate Roslyn branch.

6.1 Proposal

The final version of the proposal can be found in the attachments as the `Attachments/Proposal/partial-type-inference.md` file.

6.1.1 Creation Process

The proposal had three stages of development. The first version of the document was created in author's personal repository [21], where it was reviewed by a member of Roslyn's development team. The review was in the form of a pull request [24] where the member suggested several changes on how to structure the proposal and how to refer to the original C# standard documentation and pointed out possible improvements that would be beneficial to investigate.

After the revisions were made, the member recommended to post it as a discussion [22], which was the first time when a wider community could comment on the proposal. Besides several upvotes received from anonymous readers, another member of Roslyn's development team started to give his recommendations on how to adjust the document. The main change of the improvement was to erase most of the examples taken from the tests made together with the prototype and replace them with more references to the original C# documentation.

The third version of the improvement was published as the next discussion [23], where it received even more emoticons as likes or hearts, which was a good sign of progress. At that time, the discussion contained just answers to the questions raised by the member of the Roslyn team, which clarified the intention of the improvement.

After this step, the third stage was made by publishing the proposal as a pull request [25]. This step was done after the recommendation from the team member. The pull request was continued by another round of clarifications, recommendations, and revisions from three members of Roslyn's team.

The current stage of the proposal at the time of writing is that the pull request is still open, waiting for the next requirements from the LDT. Meanwhile, there was the LDM meeting regarding our proposal, whose result we present later in this thesis. We continue with the description of the proposal's content.

6.1.2 Content

The whole document has two styles of describing the feature. The first style explains the intention of the improvement and necessary relations, which helps to understand it. The second style used in the detailed design section is rather a patch of C# standard documentation, which enables improvement. So, the text doesn't contain fluent sentences but fragments of the documentation that need to be changed. The proposal consists of five parts:

1. The first part gives a quick overview of the proposed change, summarizing it in a few sentences.
2. The second part describes the motivation why it should be done. The text and the used examples are similar to those in section 4.2.
3. The third and largest part contains a detailed design of the improvement. There is a description of grammar change, where it explains a new underscore contextual keyword in the type argument list. It is followed by the change of binding method invocation and object creation expressions. This part describes the mentioned core design of the improvement with the changed method type inference algorithm. The design ends with extending compile-time checking dynamic member invocation, which is explained in section 5.4.
4. The fourth part comments on the reason for not doing other alternatives contained in the discussions. It also mentions two possible extensions of the improvement using the diamond operator 5.3.6 and initializers 5.3.5 in the type inference context.
5. The last part suggests other potential improvements.

6.2 Implementation

The already mentioned proposal is tested by the implementation described in this section. The goal of the implementation is to observe the consequences of the proposed feature in a practical way, which can be tried by the C# community. The goal will be achieved by contributing to the the `final/PartialTypeInference` branch [30] of the forked Roslyn project on GitHub, which is public. The branch contains one commit, with the
`[PartialTypeInference] Add partial type inference` name, containing all our changes. The copy of the branch can be also found in the attachments in the `Attachments/Source/roslyn` folder. Since the proposal is in the state of probing the benefits, the implementation is rather a proof of concept than a ready-to-production code.

6.2.1 Development Environment

Since the compiler is a complicated program consisting of many parts, the creators provide a guide [29] describing a common workflow, including building the compiler, writing tests, and deployment.

We started cloning the Roslyn repository [31] and opening it in an IDE. Several IDEs can be used. We recommend Visual Studio 2022 [40], which was used to implement the proposal. The IDE was chosen since the Roslyn folks have recommended it, and it has provided helpful static code analysis and a debugger, which is almost necessary when a programmer is unfamiliar with the code base.

However, hardware requirements for opening the repository were quite high regarding memory consumption, as it required around 20 GB of RAM for a smooth experience with browsing and launching the code. So we recommend Visual Studio Code [41] in cases when the hardware resources are limited as compensation for worse code analysis.

However, the IDE is not required to launch or deploy the compiler. The correct version of the .NET SDK specified in `global.json` placed in the project root is needed to build the project. We used SDK `8.0.1`, which can be downloaded from Microsoft's original websites [18]. We also use the Windows operating system, although the deployment should also be possible on Linux or MacOS.

Since building large programs like compilers can be complicated, the root folder provides three scripts that take care of it. The `Restore.cmd` script is run to download the required packages. The `Build.cmd` script is used to build the compiler, and the `Test.cmd` script is used to run tests of the compiler. The scripts have multiple options that modify the workflow. We will mention a few of them that were used during the implementation.

Testing

Launching the compiler tests is done by a sequence of Windows command console commands, as shown in Figure 6.1. Since the repository also contains an implementation of a C# extension to Visual Studio, tests are divided into several groups that test the independent parts of the compiler. In our case, we used the `-testCompilerOnly` flag to run basic compiler tests. The tests will be a baseline, which has to be passed by the implementation of the proposal. The result of `Build.cmd` command is a built compiler placed in the `roslyn/artifacts` folder. Test results is stored in the `roslyn/artifacts/TestResults` folder as `.xml` files and `.html` files for displaying in the browser.

```
Restore.cmd
Build.cmd
Test.cmd -testCompilerOnly
```

Figure 6.1: Running C# compiler tests on Windows.

The results of the tests, when we started the implementation, can be seen in the attachments in the `Attachments/TestResults/Tests-Master` folder, where two tests regarding Visual Basic were not passed. Since our contribution is the C# compiler, we ignore it.

Deployment

The standard deployment of a new compiler version is bundling it with a new version of the SDK, which would require compiling the SDK from sources. However, there is a possibility of injecting a custom version of the compiler into an already installed SDK. This option was created to temporarily hot-fix compiler problems delivered with an SDK. Since the implementation is a proof of concept, this option will be sufficient to check the main functionality of the proposal directly in the user's code.

The injection is done by referencing a special package generated by the `Build.cmd` script in a project file of a target C# project using partial type inference. The Nuget cache containing .NET packages needs to be modified before the package generation. The Nuget cache is usually placed in the `C://Users/%user%/.nuget` folder on Windows, where it is required to delete the version of the `Microsoft.CSharp.Net.Compiler.ToolSet` package, which will be generated by the `Build.cmd -publish` command. The command generates the `roslyn/artifacts/publish/Shipping` folder containing the `Microsoft.CSharp.Net.Compiler.ToolSet` package, which needs to be referenced. Additionally, the project file has to define a folder containing the package that will be used to restore the packages in the Nuget cache. The last required action is specifying a language version, which has to be set to `preview`, enabling the partial method type inference and the partial constructor type inference. Figure 6.2 shows a modified project file of the demo application, which can be found in the `Attachments/Demo` folder. The `PropertyGroup` element contains a definition of the language version and a folder for package restoration. The `ItemGroup` element contains a definition of the package reference injecting the locally built compiler.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <LangVersion>preview</LangVersion>
        <RestoreSources>
      E:\roslyn\artifacts\packages\Debug\Shipping
    </RestoreSources>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference
      Include="Microsoft.Net.Compilers.Toolset"
      Version="4.9.0-dev"
    />
  </ItemGroup>
</Project>
```

Figure 6.2: `Demo.csproj`

Then, a common `dotnet build` command can be used to compile the project using a custom compiler version instead of the bundled one. An easy way to check the currently used compiler is by putting the `#error version` pragma anywhere in the compiled code. Figure 6.3 shows the usage of the pragma in the demo application.

When the project is compiled again, it will raise a compilation error announcing the current package version, as shown in Figure 6.4. We can see that the code was compiled by the `4.9.0-dev` compiler version, and the language version was set to `preview`.

```
using System;

#error version

...
```

Figure 6.3: Begining of `Program.cs`.

```
E:\Demo>dotnet build
MSBuild version 17.8.3+195e7f5a3 for .NET
  Determining projects to restore...
  All projects are up-to-date for restore.
E:\Demo\Program.cs(4,8): error CS1029:
  #error: ’version’ [E:\Demo\Demo.csproj]
E:\Demo\Program.cs(4,8): error CS8304:
  Compiler version: ’4.9.0-dev (<developer build>)’.
  Language version: preview [E:\Demo\Demo.csproj]

Build FAILED.

E:\Demo\Program.cs(4,8): error CS1029:
  #error: ’version’ [E:\Demo\Demo.csproj]
E:\Demo\Program.cs(4,8): error CS8304:
  Compiler version: ’4.9.0-dev (<developer build>)’.
  Language version: preview. [E:\Demo\Demo.csproj]
    0 Warning(s)
    2 Error(s)

Time Elapsed 00:00:09.19
```

Figure 6.4: Output of building `Demo.csproj`

6.2.2 Repository Overview

Roslyn’s repository contains several projects besides the C# compiler, like the Visual Basic compiler or Visual Studio extension. Figure 6.5 shows relevant files and folders of the `roslyn` directory, which have been adjusted or are used by the implementation. The root directory contains already mentioned helper scripts for building and deploying the compiler, the `src` folder containing the code, and the `artifacts` folder generated by the build system. It also contains the `Roslyn.sln` solution, which can be opened in Visual Studio, enabling a better user experience during code browsing.

The artifacts contain the `TestResults` folder, which contains the already-mentioned test reports, and the `bin` folder, which contains the compiled code and packages. The package required to inject the compiled compiler can be found in the nested `Shipping` folder.

There are three important folders nested in the `src` folder. The `Test` folder contains compiler tests that can be run using the `testCompilerOnly` flag. The `Core` folder contains a common Visual Basic and C# compiler code. It provides a public API used to compile and analyze code. The `CSharp/Portable` folder contains the C# compiler sources, modified by the proposal's implementation.

Describing all parts of C# compiler sources is out of the scope of this thesis since it is an extensive program, and most of the parts are not influenced by the implementation. So, the thesis mentions only a small part of the compiler, which is necessary to understand implementation internals. The compiler's source code is divided into multiple folders where most of the changes are in the `Binder` folder. The folder contains logic for binding AST to *bound tree* mentioned in the Roslyn section 3.1. The implementation of the type inference algorithm can be found in the `Semantics/OverloadResolution` folder, together with the whole mechanism for overload resolution of methods and constructors. The `BoundTree` folder contains an XML description of automatically generated nodes representing *bound tree*. The `Errors` folder contains definitions of all C# errors and a list of features that can be disabled and which contain an item for enabling partial type inference. The `FlowAnalysis` contains several control flow analyzers, such as a nullability analyzer that checks the nullability state of variables. The `Generated` folder contains generated nodes of *bound tree*, and the `Symbols` folder contains nodes of the AST tree.

```
+roslyn
  +src
    +Compilers
      +Core
      +CSharp
       +Portable
         +Binder
           +Semantics
             +Conversions
             +OverloadResolution
        +BoundTree
        +Errors
        +FlowAnalysis
        +Generated
        +Symbols
          +Source
      +Test
  +artifacts
    +bin
    +packages/Debug
      +Shipping
    +TestResults
  -global.json
  -Roslyn.sln
  -Build.cmd
  -Restore.cmd
  -Test.cmd
```

Figure 6.5: Simplified view of the `roslyn` directory content.

6.2.3 Test Suite

Roslyn tests use the xUnit framework [42], providing wide API for testing application functionalities in various scenarios. We used this framework for our own tests in the `PartialTypeInferenceTests.cs` file. An additional helper method, `TestCallSites`, was made to save repeating code, which initializes the compilation, compiles the provided source code as a string, and verifies diagnostics representing compiler warnings and errors. Since we are interested in information regarding inferred type arguments, the source code is augmented by comments representing asserts which the augmented symbols have to hold.

Figure 6.6 shows an example of a test case that tests a scenario where an inferred type argument syntax hides a typename having the _ identifier. We can see that the `new F1<_>(1)` expression is tested by the following comment describing the desired resolved call. The second argument of the `TestCallSites` defines which symbols we assert. In this case, all object creation expressions are asserted. The last argument describes expected diagnostics, which the compiler should announce. This example requires a warning regarding notifying a user about possible conflicting class names with the _ contextual keyword.

```
[Fact]
public void PartialConstructorTypeInference_UnderscoreClass()
{
  TestCallSites("""
class P
{
  static void M()
  {
    new F1<_>(1); //-P.F1<int>..ctor(int)
  }

  class F1<T> { public F1(T p) {} }
}

class _ {}
""",
    Symbols.ObjectCreation,
    ImmutableArray.Create(
      Diagnostic(ErrorCode.WRN_UnderscoreNamedDisallowed, "_")
        .WithLocation(11, 7)
    )
  );
}
```

Figure 6.6: Example of a test.

6.2.4 Parsing Inferred Type Arguments

We didn't change the Roslyn parser. Instead of that, we created the `SourceInferredTypeSymbol` symbol representing the inferred type argument and postponed the _ type argument recognition to the binding phase. In this phase, we changed the `BindIdentifier` method in `Binder_Expressions.cs`, which looks up the type symbol represented by the provided identifier. The change is propagated into subsequent calls, which handle various cases of identifier binding, such as generic type names or fully qualified names. If the method is called during the binding type argument list of a method invocation or object creation expression, we change the mode, which enables the binding of the _ identifier as an inferred type argument symbol.

6.2.5 TypeInferrer

Replacing the `MethodTypeInference.cs` file with the `TypeInference.cs` was the biggest source code update in the code. Since the old algorithm for type inference was primarily focused on generic method type inference, we rewrote it in a more generic way using a concept of type variables and the relations between them and the type symbols. Additionally, we used the `#region` syntax to divide the source code into multiple parts referencing corresponding parts of the already mentioned type inference algorithm. We also added a public API, which adapts the old method type inference API to the new one. This helped us to not change the other interacting parts of the codebase too much. There is the `InferMethod` static method, which has a signature similar to the previous one and is an entry point for the partial method type inference. We also added the `InferConstructor` static method as an entry point for the partial constructor type inference.

6.2.6 Binding Partial-inferred Method

The partial-inferred method binding is contained in the `Binder_Invocation.cs` and the `OverloadResolution.cs` files. The changes in the binder were mostly about an error recovery and raising potential warnings. An example of that is a warning raised during dynamic method invocation binding where if the inferred type argument is used, we notify the user about a potential error in runtime since runtime doesn't support the partial type inference. The change in the overload resolution was to analyze the method group with used inferred type arguments and to use the mentioned API of the type inferrer.

6.2.7 Binding Partial-inferred Object Construction

Binding partial-inferred object construction was the most tricky in the implementation since there was no preparation for the type inference as in the previous case. Moreover getting type information from the target is complicated since the binding order is reversed to the binding direction.

The first step was to change the `OverloadResolution.cs` file to infer the type defining the constructor candidate when it contains inferred type arguments. Then, we had to substitute the constructor signature with inferred type arguments and choose the constructor that best fits.

During the implementation, the following problem occurred. Figure 6.7 shows a definition of the `M` class containing two constructors. The second constructor's body contains an expression creating an object of the same type. The inferred type argument of this expression should be the T type parameter of the class. However, to be able to do that, alpha renaming has to be applied to the inferring type arguments since the type of the p1 parameter is the T type symbol, and the type variable representing the inferring type argument is also the T type symbol. Without this modification the type inference failed since the algorithm thought that it could not find the exact type symbol for that type argument. From the type inference perspective, these two symbols are different, so we had to distinguish these cases by replacing the type parameters represented by type variables with an alpha-renamed type symbol representing the same symbol. The alpha-renamed symbol is different from the one received from the argument list. This prevented the mentioned problem, and the expression was properly bound. Note that this problem also occurs in the binding of method invocation when there is a recursive generic method in which a recursion call is inferred and uses the same type of arguments.

```
class M<T>
{
    M(T p1, int p2) {}

    M(T p1)
    {
        new M<_>(p1, 1);
    }
}
```

Figure 6.7: Problem regarding alpha renaming.

The second step was to add additional type information received from the type parameter constraints of the containing type.

The last step was to enable a target-typed inference. We used a similar way as in the `new()` operator. We created a new `BoundUnconvertedInferredClassCreationExpression` representing an object creation expression containing the inferred type. This expression is bound lately when a target type is known in conversions defined in the `Conversions.cs` file. Whenever an expression is assigned, these conversions are used to check if it is possible to convert the expression to the target and generate appropriate conversion if necessary. In our case, the appropriate conversion is to invoke the binding of the object creation expression again with the already known target type, which is used in the type inference. Also, the expression doesn't have to be assigned. For these situations, the `BindToNaturalType` method is called to invoke the conversions in these scenarios to make additional expression processing. In our case, it is binding the expression without the target type. The conversions are a bottleneck for this process, and every assignment or a statement uses it to invoke the additional necessary operation on expressions.

The previous paragraph is not completely true in the binding order of the expression without a target. At the time of creating the unconverted inferred class creation expression, we bind the expression without the target and save the diagnostics obtained during that. This prevents us from binding the same expression without targeting multiple times, as we can see in Figure 6.8. There is the `F(new M<_>(1))`, which calls a generic method with an inferred object creation expression. At the time of binding the object creation expression, we bind it without the target but still represent it as an unconverted inferred class creation expression. In the time of binding the `F` method, the overload resolution invokes the inference for each overload. Since the inference is influenced by a type of argument, we have to use the type of that expression. In this case, we use an already bound object creation expression without a target, which enables type inference to infer the type argument of the `F` function call. Also, there is

```
F(new M<_>(1))

static void F<T>(M<T> p1);
static void F<T>(M<T> p1, T p2);

class M<T>
{
    M(T p1) {}
}
```

Figure 6.8: Preventing multiple binding.

an opposite scenario. Figure 6.9 shows an example where the first argument needs the target type to infer the expression. The argument doesn't influence the type inference algorithm since the corresponding parameter doesn't contain the inferred type argument. In this case, after the overload of the `F` method is chosen, there is a phase of applying conversions from the arguments to the parameters invoked by the `CoerceArguments` method. This phase binds the first argument with the already known type of the parameter as a target type. We prohibit this behavior when the object creation expression is used in the type inference by generating a different type of conversion for that. Also, in this phase, we add the previously saved diagnostics depending on whether we bound the expression without the target. See `ObjectCreationConvertionWithTarget` and `ObjectCreationConvertionWithoutTarget` conversions, which were created for this purpose.

```
F(new M<_>(), 1)

static void F<T>(M<int> p1);
static void F<T>(M<int> p1, T p2);

class M<T>
{
    M() {}
}
```

Figure 6.9: Target-typed binding.

As we can see, binding an expression with the target type has complicated implementation in Roslyn.

6.2.8 Nullable Walker

The last part of the code changes is in the `NullableWalker.cs` file, which provides a nullability analysis pass. The basic principle of the analysis is traversing the control-flow graph and rebinding the nodes of *bound tree*, if necessary, according to the nullability. The rebound nodes are rewritten at the end of the pass.

An important change was to modify the analysis of the bound object creation expression symbol. Based on the information stored in the previous binding, we know if it was target-typed. If it was target-typed, we postpone the binding in the same way as the `new()` operator. The partial constructor type inference is added in the process of rebinding the constructor, where we use saved data from the previous binding to repeat it in the context of nullability. After the expression binding, we rewrite the containing type if it was changed during the type inference by storing the update to the end of the nullability analysis pass.

7. Evaluation

The implementation and proposal are evaluated using the following metrics.

7.1 Tests

We run the same suite case before and after our changes. The results can be found in the `/Attachments/TestResults` folder containing the `Test-Master` folder and the `Test-Feature` folder corresponding to test results before and after the change. These folders contain the results from the `-testCompilerOnly` suite case run whose tests can be found the the `Test` folder mentioned in Section 6.2.2. All tests pass when they are run separately. However, we noticed that when we use the testing framework to run `-testCompilerOnly` suite case, two of our tests don't pass as can be seen in the `WorkItem_4_x64_test_results.html` test results summary. We didn't manage to identify the reason for that, but we think that it is related to the test framework complexity of how it runs the tests. This failure needs further investigation in the context of how the test framework executes the tests. Except for this failure and unimportant small changes in the compiler tests regarding adding new error messages caused by our changes, all tests that passed before the improvement also passed after the improvement. This result ensures that the improvement doesn't introduce any significant regression in the compiler tests.

7.2 Demo Examples

The basic capabilities of the improvement are demonstrated in the demo application contained in the `Attachments/Demo` attachments folder. We also show advanced features of the improvement which can be found in the tests and which were presented to LDT.

7.2.1 Basics

Figure 7.1 shows an example of the partial method type inference where the underscore is used to skip the first type argument of the `M1` method call because it can be inferred from the first argument. In this case, the inferred type argument is `int`.

```
void M1<T1, T2>(T1 p1) { ... }
...
M1<_, string>(1);
```

Figure 7.1: RunExample1 – Top level inferred type argument

Figure 7.2 shows the usage of the underscore in the first type argument of the `M2` method call where it is an inferred type argument of the `IList<T>` type. The usage allows to influence the original method type inference algorithm which would infer the type of the first argument. The nested inferred type argument is `int`.

```
void M2<T1>(T1 p1) { ... }
...
M2<IList<_>>(new List<int>());
```

Figure 7.2: RunExample2 – Nested inferred type argument

Figure 7.3 shows an example of the partial constructor type inference where the first type argument of the `C1<T1>` object creation expression is inferred based on the first argument. The process of the type inference is the same as in the first example. The inferred type argument is `int`.

```
class C1<T1> {
  public C1(T1 p1) { ... }
}
...
new C1<_>(1);
```

Figure 7.3: RunExample3 – Top level inferred type argument

Figure 7.4 shows the usage of the nested inferred type in the `C2<T1>` object creation expression. The process of type inference is the same as in the second example. The nested inferred type argument is `int`.

```
class C2<T1> {
  public C2(T1 p1) { ... }
}
...
new C2<IList<_>>(new List<int>());
```

Figure 7.4: RunExample4 – Nested inferred type argument

Figure 7.5 shows an example where type inference of the `C3<T1>` object creation expression uses the target type to infer the type argument of the `C3<T1>` type. The target type is the `C3<int>` type obtained from the `a` variable declaration. The inferred type argument is the `int` type.

```
class C3<T1> {
  public C3() { ... }
}
...
C3<int> a = new C3<_>();
```

Figure 7.5: RunExample5 – Target-typed inference

Figure 7.6 shows an example of type inference based on the type constraints where the first type argument of the `C4<T1, T2>` object creation expression is inferred based on the second type argument. Since the first type argument has to inherit the `List<T2>` type, it gives us a default for this type argument. So the first type argument is inferred to the `List<int>` type.

```
class C4<T1, T2> where T1 : List<T2> {
    public C4() { ... }
}

new C4<_, int>();
```

Figure 7.6: RunExample6 – Type inference based on constraints

7.2.2 Advanced Examples

Figure 7.7 shows the `F4` generic method containing three function parameters. The parameters and arguments are assembled in the way that is required to infer `T1` to infer `T2`, which is required to infer `T3`, which is required to infer `T1`. The call site uses the partial method type inference to hint to the compiler one of the type arguments, which will break this circular dependency and enable the inference of the rest of the type parameters.

```
void F4<T1, T2, T3>(
  Func<T1, T2> p12,
  Func<T2, T3> p23,
  Func<T3, T1> p31)
{}
...
F4<_, _, string>(x => x + 1, y => y.ToString(), z => z.Length);
```

Figure 7.7: Circular type inference dependency of type parameters

Figure 7.8 shows an advanced scenario containing a nested inferred type argument. There is the `F6` generic method call containing one type parameter. The `temp` variable and the type argument's hint (`I2<_, A>`) are used to determine the inferred nested type argument, which is `int`. During the process, multiple rules for inference, such as inheritance and variance, are applied.

```
class A {}
class B : A {}
interface I2<T1, out T2> {}
class C2<T1, T2> : I2<T1, T2> {}
void F6<T1>(T1 p1) {}
...
F6<I2<_, A>>(new C2<int, B>());
```

Figure 7.8: Nested inferred type argument

Figure 7.9 shows an example where the nullability inference is a part of the type argument deduction. The `F10` call site has an inferred type argument with an appended question mark determining the nullability of the inferred type argument. The inferred type argument would be `string` if there wouldn't be the appended question mark. However, the question mark is used to hint to infer `string?`.

```
void F10<T>(T p1) {}
...
F10<_?>("");
```

Figure 7.9: Inferred type argument nullability hint

Figure 7.10 shows an example of constructor type inference, which uses a target type to infer type arguments of the new `new C2<_>()` expression. The target type is a type of the `C4` constructor parameter resulting in the inferred `int` type argument.

```
class C1<T> {}
class C2<T> : C1<T> {}
class C4
{
  public C4(C1<int> p1) {}
}
...
new C4(new C2<_>());
```

Figure 7.10: Target-typed object creation expression

Figure 7.11 shows an example of an advanced binding order of arguments. The top-level object creation needs the type of the constructor's argument to determine its type parameter. For this purpose, the nested object creation expression is bound without the target to provide this type of information.

```
class C1<T> {}
class C5<T> : C1<T>
{
    public C5(T p1) {}
}
...
new C5<_>(new C5<_>(1));
```

Figure 7.11: Advanced arguments binding – Without target type

Figure 7.12 is the advanced scenario of a previous use case where the first argument can't be bound because type inference failed during the attempt to bind it without a target type. The failure is caused by a lack of type information. However, the second argument provides enough type information to infer a type parameter of the `C3` class. After the bound, the new `new C2<_>()` expression is tried to bound again with the inferred target type. After that, both inferred type arguments are resolved to the `int` type.

```
class C1<T> {}
class C2<T> : C1<T> {}
class C3<T>
{
   public C3(C1<T> p1, T p2) {}
}
...
new C3<_>(new C2<_>(), 1);
```

Figure 7.12: Advanced arguments binding – With target type

Figure 7.13 shows the most advanced scenario where type the binding order and type inference uses several things to infer the `new C9<_,_,int,_>(1)` expression. We will focus on how the type parameters of the `C9` class are gradually inferred. The `T3` type parameter is inferred by using the provided hint in the type argument list, which is `int`. The `T1` type parameter is inferred by using a type of constructor's argument `p1`, which is `int`. The `T4` type parameter is inferred by using the type parameter's dependency using the `T3` type parameter. The type parameter is resolved to `int`. The `T2` type parameter is resolved by using the target type given by the `F1` method parameter `p1`. This type information is received in the second try of binding the argument using the target type. The `T1` type parameter of the `F1` method is determined based on its second parameter type, which is the `int` type.

```
class C1<T> {}
void F1<T>(C1<T> p1, T p2) {}

class C9<T1, T2, T3, T4> : C1<T2> where T4 : C1<T3>
{
    public C9(T1 p1) {}
}
...
F1(new C9<_,_,int,_>(1), 1);
```

Figure 7.13: Constructor type inference – All in one

Based on the previous examples, the solution can be considered robust enough to solve complicated examples.

7.3 LDM Meeting Summary

LDM discussed the proposal on 7th February 2024. Besides the overall summary of the proposal with additional questions, the LDT agreed to continue with moving the proposal forward. Although it is not going to present the change in the upcoming C# 13, it is planned to ship this feature with some modifications in the following releases. The summary [15] of the meeting and the following discussion [16] are published on the C# GitHub repository.

8. Future Work

Since LDM agreed to proceed with the partial type inference, the following work addresses future comments in the PR [25]. Simultaneously, there is the discussion [16] regarding the meeting where the feature is discussed. So, the continuation of this thesis is replying to potential questions there.

Besides that, there is a plan to have the next LDM meeting regarding further proceeding with the proposal, which will be cut into smaller pieces to let the LDM discuss it properly. The meeting will regard two items. The first item is top-level inferred type arguments in the method invocation and object creation expression. The second item is nested inferred type arguments in the method invocation and object creation expression.

It is likely that during the process of continuing discussions, the proposal will be divided into more pieces, which will be delivered across multiple releases.

Conclusion

The section 1.3 of the thesis's introduction sets four goals, which are gradually achieved by the thesis. It starts by exploring C# type inference by using the language specification and implementation in the Roslyn project. Then, the thesis compares C# type inference with Rust type inference, which offers more advanced type inference than C#. The type inference difference is explained using the theory background of Hindley-Milner type inference, which describes the limits of type inference in strongly typed languages.

Based on these observations, the thesis selects a subset of type inference improvements suggested in C# language discussions on GitHub and explores the motivation behind it to make it likely to add to the C# language. The motivation is described in the created proposal, which is the fourth goal and which describes the improvement in terms of changing the specification.

Together with the proposal, the implementation was made in the Roslyn project fork, with unit tests checking the functionality. This is the third and last goal of the thesis. The implementation ensured that basic compiler tests passed to reveal possible regression.

In addition to the mentioned goals, the thesis succeeded in presenting the proposal and implementation to the LDM responsible for approving language changes into C#. The committee agreed to continue with the change for further discussion and is generally inclined to ship it with future language releases.

Based on the goals achieved and additional success in the LDM meeting, which was not a part of the scope of the thesis, we claim that the thesis accomplished all the promised goals.

Bibliography

[1] Andreas Stadelmeier and Martin Plumicke. Adding overloading to Java type inference.

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

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

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

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

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

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

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

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

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

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

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

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

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

[15] LDM meeting summary. https://github.com/dotnet/csharplang/blob/main/meetings/2024/LDM-2024-02-07.md. [Online; accessed 2024-2-16].

[16] LDM meeting summary discussion. https://github.com/dotnet/csharplang/discussions/7938. [Online; accessed 2024-2-16].

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

[18] .NET SDK. https://dotnet.microsoft.com/en-us/download/dotnet/8.0. [Online; accessed 2023-11-28].

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

[20] 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].

[21] Personal repository. https://github.com/TomatorCZ/master-thesis. [Online; accessed 2023-12-29].

[22] Proposal discussion 1. https://github.com/dotnet/csharplang/discussions/7286. [Online; accessed 2023-12-29].

[23] Proposal discussion 2. https://github.com/dotnet/csharplang/discussions/7467. [Online; accessed 2023-12-29].

[24] Proposal pull request 1. https://github.com/TomatorCZ/master-thesis/pull/1. [Online; accessed 2023-12-29].

[25] Proposal pull request 2. https://github.com/dotnet/csharplang/pull/7582. [Online; accessed 2023-12-29].

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

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

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

[29] Roslyn development guide. https://github.com/dotnet/roslyn/blob/main/docs/contributing/Building%2C%20Debugging%2C%20and%20Testing%20on%20Windows.md. [Online; accessed 2023-11-28].

[30] Roslyn fork. https://github.com/TomatorCZ/roslyn/tree/final/PartialTypeInference. [Online; accessed 2024-2-23].

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

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

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

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

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

[36] The billion dollar mistake. https://medium.com/@PurpleGreenLemon/how-null-references-became-the-billion-dollar-mistake-bcf0c0cc72ef. [Online; accessed 2024-2-16].

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

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

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

[40] Visual Studio. https://visualstudio.microsoft.com/vs/. [Online; accessed 2023-11-28].

[41] Visual Studio Code. https://code.visualstudio.com/. [Online; accessed 2023-11-28].

[42] xUnit. https://xunit.net/. [Online; accessed 2024-2-4].

A. Attachments

Table A.1 describes the content of the attachments. Although the attachments contain most of the thesis outcomes, part of the outcome is placed on a public GitHub repository where further discussions regarding the change, made by C# community members and LDT, continue.

Folder	**Description**
`Bin/`	Nuget packages of compiler with proposed changes
`Demo/`	Ready-to-run demo using the packages
`Proposal/`	Laguage change proposal and LDM summary
`Source/roslyn/`	Source code of forked compiler with proposed changes in the `[PartialTypeInference] Add partial type inference` commit
`TestResults/`	Test results before and after changes

Table A.1: Attachments content

A.1 Build and run Demo

The thesis contains a demo that can be used as a playground for testing the change. The .NET SDK is needed to be able to compile and run the demo. We used version 8.0.100, which can be downloaded from Microsoft's offical websites

After the download, make sure that you don't have the `Microsoft.Net.Compilers.Toolset.4.9.0-dev` package in your nuget packages cache. The cache is usually placed in the `C:/Users/%user%/.nuget` folder. Then, navigate to the `Demo` folder and run `dotnet build` which builds the demo example.

A.2 Build your applications

You can try to build already existing C# projects by the modified compiler by adding the following code fragment A.1 to the `.csproj` file. Then, you can again use a common `dotnet build` to build your application.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <RestoreSources>
      path/to/Attachments/Bin/folder
    </RestoreSources>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference
      Include="Microsoft.Net.Compilers.Toolset"
      Version="4.9.0-dev"/>
  </ItemGroup>
</Project>
```

Figure A.1: `.csproj` project