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

TODO: Dedication.

Title: Improving Type Inference in the C# Language

Author: Tomáš Husák

Department: Department of Distributed and Dependable Systems

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

Abstract:

TODO: Abstract.

Keywords: Type Inference C# Roslyn

Contents

1. Introduction

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

1.1 Improving C# type system

A key feature of strongly typed languages is type safety, prohibiting operations on incompatible data, achieved by determining data types at compile time. The easiest way for a compiler to reason about types of variables in the code is by providing type annotations determining the data type that these variables hold. Figure 1.1 shows an usage of type annotations given by a programmer 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, the programmer has to write more code to annotate the variable declaration and object creation whose type has a long name, 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 the context of 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`

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

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

Figure 1.3: C# Type inference of generic methods.

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

A similar concept of generic methods was introduced in the Rust [20] programming language, which belongs to strongly type languages too. Figure 1.4 shows a definition of the generic method `GetField`, which is equivalent to the C# method mentioned in the previous example. There is an equivalent initialization of `name` variable declaration starting with the `let` keyword, where Rust type inference deduces the type argument `T` to be the `&str` type utilizing the type information from the `name` variable declaration.

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

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

Figure 1.4: Rust Type inference of generic methods.

Although Rust is younger than C# and has a different type system, it managed to make type inference more powerful in the context of strongly typed languages to significantly save type annotations typing. The first goal of this thesis is to investigate if the same 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 [23] 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 [22], where it can deduce types of all variables in an entirely untyped code. The power of type inference is caused by properties of the type system, which, in comparison with the C# type system, doesn't use type inheritance or overloading. Despite the differences, Hindley-Millner type inference can be modified to work with other type systems like Rust or C#, causing limited use cases where it can be applied. Observing the influence of differences between these type systems on type inference will help to understand a limitation of possible type inference improvement in C#.

1.2 Implementation

The first part of the thesis explores limitations of C# type inference and proposes an improvement. The first goal of the second part tests the improvement by implementing it in an official C# compiler, Roslyn [16], which is an open-source project managed by Microsoft. The prototype is used to explore potential implementation issues which the improvement can cause, and that helps to adjust the improvement to be potentially enabled in the C# compiler.

Although the compiler is managed by the company, it has an open-source development, which makes contributions from interested people possible to be merged into the production. Although it is sufficient to make a *pull request* containing a fix for solving compiler issues to be merged, language design improvements, similar to what the thesis proposes, 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 [16], where everyone can add his ideas or comment on others' ideas. It is preferred to use a predefined template [18] 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 [17] changes, and other possible alternatives. The second goal of this part is to create the improvement as the language proposal, which would be presented to the team in order to have the potential to be a part of the current C# language. The process of language proposal ends with LDT accepting or declining it.

1.3 Summary

We summarize goals of this thesis in the following list:

G1. Explore possibilities of type inference in strongly typed languages

G2. Improve C# type inference based on previous analysis

G3. Implement the prototype in Roslyn

G4. Create an proposal containing the improvement

2. Related work

The introduction 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. A primary source of inspiration for the improvement is Hindley-Milner type inference, which is explored in more detail with references to its modification in Rust and C# programming languages. In the end, the chapter mentions relevant C# language issues presented on the GitHub repository, which is used later to prioritize the improvement features to make it more likely to be discussed at Language Design Meetings (LDM) held by LDT.

2.1 C# programming language

Since type inference is a complicated process touching many areas of the C# language, this section 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.

2.1.1 Type system

C# data types are defined in the C# type system, which also defines relations between them. The most fundamental relation is type inheritance, where every type inherits another type, forming a tree with `System.Object` as a root node that doesn't inherit any type. Types are divided into value and reference types, shown in Figure 2.1, where an arrow means *is inherited by* relation. Value types consist of built-in numeric types referred to as *simple types*, and enumerations referred to as *enum types*, structures referred to as *struct types*, and nullable types. Compared to reference types, value types are implicitly sealed, meaning that they can't be inherited by other types. Reference types consist of interfaces, classes, arrays, and

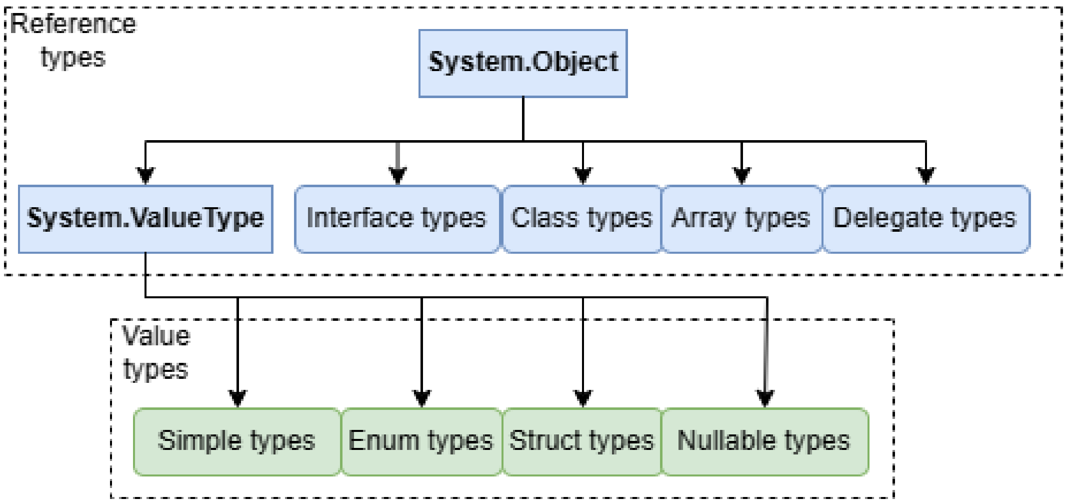

Figure 2.1: The C# data types schema adjusted from a C# blog[10].

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 [12], it allows to assign `null` value to nullable types, which are equivalents of the rest of value types prohibiting it. Because assigning `null` value is referred to as a billion-dollar mistake, C# 8.0 [12], introduced optional settings warning about assigning null values and created nullable reference types, which, together with nullable types, explicitly allows `null` assignment as a way of interaction with legacy code not using the feature.

A big part of the type system is C# *generics*, allowing the parameterization of types and methods by arbitrary types. A specific generic method or type is *constructed* by providing required type arguments, where *construction* means replacing all occurrences of type parameters with the type arguments. Since type argument can be arbitrary type, the type parameter is considered to be the most general type in the type system, `System.Object`. Assuming additional API from the type parameter is achieved by restricting a set of types, which can replace the type parameter, enabling a specific interface of this set. The restriction is described by type constraints, which can be applied to type parameters. There are several kinds of constraints that can be combined together, forcing the type argument to fulfill all of them. Figure 2.2 shows only two of them, and the rest can be found in the C# documentation [14]. 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.

2.1.2 Relevant constructs

This section mentions unrelated C# constructs where type inference occurs or influences the type inference algorithm. Their internals are then considered in the following chapters regarding the design of the improvement.

Dynamic

Introduction 1.1 mentioned that strongly typed languages require knowing data types at compile time to prohibit incompatible operations on them. In the context of C#, data means values of expressions that are transformed by operations defined on their types. It turned out that operations on expressions of unknown type at compile time became crucial for interoperability with other dynamic-typed languages whose types of expressions are known at runtime. To make the interoperability easier, C# introduced the `dynamic` type that can be used as an ordinary type, which avoids the checks and causes *dynamic binding*. *Binding* is a process of resolving referenced operations based on the type and value of the expression. The majority of the C# binding happens statically at compile time. Expressions containing a value of the `dynamic` type are dynamic bound at runtime, bypassing the static binding of the compiler. This behavior can lead to possible bugs regarding invalid operations on the dynamic data types, which will be reported during runtime. Figure 2.3 shows a declaration of the `a` variable of the dynamic type. Dynamic binding occurs in the `a.Foo()` expression, where the `Foo()` operation is not checked during compilation. An error is reported at runtime when the actual type of the `a` variable is determined to be `string`, which doesn't define `Foo()` operation. Despite the dynamic binding, a compiler can still little check certain kinds of expressions containing values of dynamic types to reveal possible errors at compile time. An example of such checking is the `Bar()` method call, where the compiler can check the first argument, whose type is known at compile time as the type of the parameter. An appropriate error occurs during the compilation because string value has the `string` type, and it is passed as the texttp1 parameter, which has the `int` type.

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

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

Figure 2.3: C# dynamic type.

Anonymous function

C# allows to define a function without a name, called *anonymous function*. The function is represented as an expression that can be called or stored in a variable. There are three types of anonymous function. The first type is *anonymous method* shown in Figure 2.4 where it is stored in the `a` variable. The `b` variable contains the second type called *explicit typed anonymous function*. The third variable `c` contains the last type called *implicit typed anonymous function*. As can be seen, all of them have inferred return types based on return expression inside their bodies. The most interesting type is the last one, where even parameter types are inferred based on a surrounding context and which is especially threatened by *method type inference* algorithm mentioned in Type inference section 2.1.3.

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

Figure 2.4: C# anonymous functions.

Object creation expression and initializer

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

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

Figure 2.5: C# collection initializer.

2.1.3 Type inference

C# type inference occurs in many contexts. However, this section mentions only these related to our improvement described in the following chapters.

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.

Operator new()

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

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

Figure 2.7: Operator new().

Method type inference

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

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

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

Figure 2.8: Method type inference.

Since one of the thesis's improvements is adjusting the algorithm, this section presents its description. The thesis doesn't show the complete algorithm described in the C# specification [13] since it is complex, and some sections are unimportant for the following chapters. The simplified algorithm is divided into four separate figures. Before describing the algorithm, the section presents definitions used by the algorithm.

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*.*

`{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.

The pseudocode used to describe the algorithm uses custom helper functions explained in 2.1. Figure 2.9 shows the initial phases of the algorithm. The method type inference process starts with receiving arguments of a method call and the method's signature, which type parameters have to be deduced. The algorithm has two phases, where the first phase initializes initial bounds' sets of type variables(inferred type arguments), and the second phase repeats until all type variables are fixed or fail if there is insufficient information to deduce them. Each type variable has three types of bounds. The exact bound consists of types, which have to be identical to the type variable, meaning that they can be converted to each other. The lower bound contains types that have to be convertible to the type variable, and the upper bound is opposite to it.

```
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 $\rightarrow$
    if (e.isAnonymousFunc)
      InferExplicitParamterType(e, $T$[e.idx])
    elif (e.getType() is Type u)
      switch (u) {
        p[e.idx].isValParam $\rightarrow$ InferLowerBound(u, $T$[e.idx])
        p[e.idx].isRefParam || p[e.idx].isOutParam $\rightarrow$
          InferExact(u, $T$[e.idx])
        p[e.idx].isInParam && e.isInArg $\rightarrow$
          InferLowerBound(u, $T$[e.idx])
      }
  )
fn SecondPhase():
  while (true):
    $X_{indep}$ = $X$.filter(x $\rightarrow$
      F[x.idx] == null && $X$.any(x $\rightarrow$ dependsOn(x, y)))
    $X_{dep}$ = $X$.filter(x $\rightarrow$
      F[x.idx] == null && $X$.any(y $\rightarrow$
        dependsOn(y, x) && ($B_{lower}$+$B_{upper}$+$B_{exact}$).isNotEmpty))
    switch {
      $X_{indep}$.isNotEmpty $\rightarrow$ $X_{indep}$.foreach(x $\rightarrow$ Fix(x))
      $X_{dep}$.isNotEmpty && $X_{indep}$.isEmpty $\rightarrow$ $X_{dep}$.foreach(x $\rightarrow$ Fix(x))
      ($X_{indep}$+$X_{dep}$).isEmpty $\rightarrow$
        return if (F.any(x $\rightarrow$ x == null)) Fail() else Success(F)
      default $\rightarrow$ $E$.filter(e $\rightarrow$
          $X$.any(x $\rightarrow$
            F[x.idx] == null && $T$[e.idx].outTypes.contains(x))
          && !X.any(x $\rightarrow$
            F[x.idx] == null && $T$[e.idx].inTypes.contains(x))
        ).foreach(e $\rightarrow$ InferOutputType(e, $T$[e.idx]))
    }
```

Figure 2.9: Phases of Method Type Inference

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 the example where the compatibility between `List<int>` and `IList<T2>` is firstly checked, and then the `int` type is added as a lower bound of the `T2` type parameter. Type variables can have dependencies between themselves, so the first phase postpones matching output types of arguments' types with corresponding parameters' types because the output types can contain dependent type variables.

`SecondPhase()` happens iteratively, respecting the *depends on* relation. Each iteration has two goals. The first one is the fixation of at least one type variable. If there is no type variable to fix because either all type variables are fixed or there are no other type bounds that could be used for type variable deduction, the algorithm ends. The sets X_{indep} and X_{dep} refer to type variables, which can be fixed in the current iteration. Line 31 contains the ending condition of the algorithm when all type variables are fixed, or there is no way to infer the next ones. The second goal is to match postponed matching of output types of arguments' types with output types of corresponding parameters' types where all type variables in the output type of a parameter type don't depend on unfixed variable types contained in input types of the parameter type. The second goal is to match postponed matching of output types of arguments' types with output types of corresponding parameters' types where all type variables in the output type of a parameter type don't depend on unfixed variable types contained in input types of the parameter type. Line 32 describes this process using the pseudocode. This goal represents dealing with implicit anonymous functions mentioned in Figure 2.8 where `T3` depends on `T2`. The algorithm first infers the `T2` input type of the anonymous function, then infers the function's output type, which is then used to match the output type of `Func<T2, T3>` parameter type with the output type of the inferred anonymous function's type. The match yields the `int` upper bound of the `T3` type parameter.

```
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 2.10: *Explicit parameter type inference, Output type inference*

Figure 2.10 contains definitions of two helper functions used in the first and second phases. The `ExplicitParameterType()` function is used to match an argument type, which is an explicit typed anonymous function. This function has typed parameters, so the algorithm matches them with input types of the corresponding parameter type.

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

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

Figure 2.11: *Exact inference*, *Upper-bound inference*, *Lower-bound inference*

Figure 2.11 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.

An interesting fact about adding new bounds is that there is no need to check possible contradictions because they are checked in the type variable fixation. The second observation that will be important for the following chapters is the absence of unfixed type variables in bound sets, making the algorithm easier. 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.

The last part of this algorithm is type variable fixation, shown in Figure 2.12. 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 2.12: Fixing of type variables

Array type inference

The last mentioned type inference happens in array initializers when the array type should be deduced from the initializer list. Figure 2.13 shows an example of a situation when the type inference is used for determining the `object[]` type of the `myArray` array. The C# specification calls it *common type inference*, which finds the most specialized common type between given types. From one point of view, it is just adjusted the already mentioned method type inference algorithm where there is just one type variable, and all initializer items are lower bounds of that type variable.

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

Figure 2.13: Array type inference.

2.2 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.

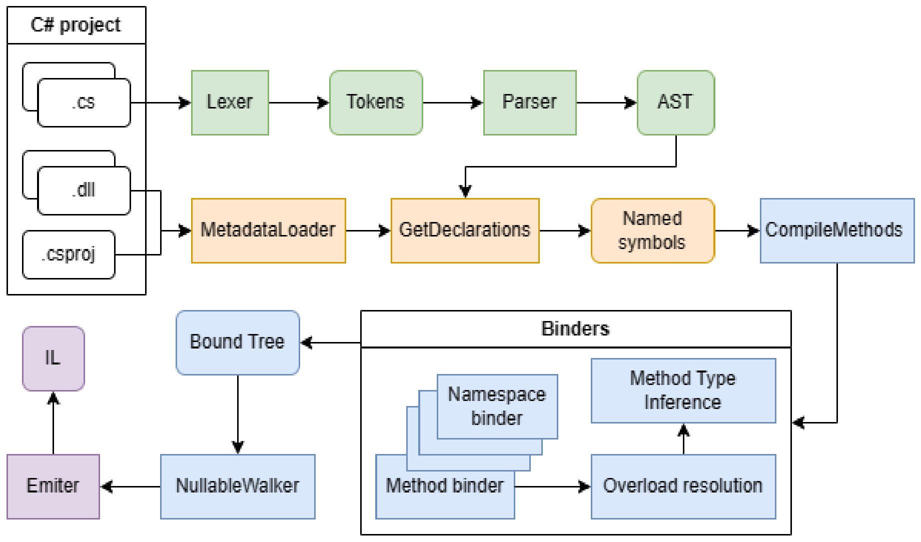

Figure 2.14: Roslyn architecture

This section presents Roslyn's architecture to better understand the context and restrictions that has to be cosidered to plug the improved type inference into the compiler.

Figure ?? shows the compilation pipeline, described in Microsoft documentation [19]. 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 ??) of the compiler checking syntax of C# sources.

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

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

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

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

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

2.3 Hindley-Millner type inference

Note: Intro

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

Note: Hindley-Millner type system

Hindley-Millner type system [22] is a type system for Lambda calculus capable of generic functions and types. We start with describing a set of expressions, which will be a target of type inference described in the video series [4]. Expression is either a variable (2.1), a function application (2.2), a lambda function (2.3), or a *let-in* clause (2.4).

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

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

$$
\begin{aligned}
mono\ \tau &= \alpha && (2.5)\\
&\mid C\ \tau_1, ..., \tau_n && (2.6)\\
poly\ \sigma &= \tau && (2.7)\\
&\mid \forall \alpha\ .\ \sigma && (2.8)
\end{aligned}
$$

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

Note: Set of rules

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

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

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

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

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

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

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

```
fn Infer(Γ, expr):
  switch(expr):
    expr isLike 'x' → return ({}, Instantiate(expr))
    expr isLike '$\lambda x \to e$' →
      $\beta$ = NewVar()
      ($S_1$, $\tau_1$) = Infer(Γ + x: $\beta$, e)
      return ($S_1$, $S_1\beta$ → $\tau_1$)
    expr isLike '$e_1 e_2$' →
      ($S_1$, $\tau_1$) = Infer(Γ, $e_1$)
      ($S_2$, $\tau_2$) = Infer($S_1$Γ, $e_2$)
      $\beta$ = NewVar()
      ($S_3$, $\tau_1$) = Unify($S_2\tau_1$, $\tau_2$ → $\beta$)
      return ($S_3S_2S_1$, $S_3\beta$)
    expr isLike 'let x = $e_1$ in $e_2$' →
      ($S_1$, $\tau_1$) = Infer(Γ, $e_1$)
      ($S_2$, $\tau_2$) = Infer(Γ + x: Generalize($S_1$Γ, $\tau_1$), $e_2$)
      return ($S_2S_1$, $\tau_2$)
```

Figure 2.15: *W* algorithm

Note: Describe type inference

With the definitions given above, we can start to talk about H-M type inference, which is able to find the type of every expression of a completely untyped program. There exist several algorithms for inferring most general type of the expression. We show the W algorithm in Figure 2.15 since it is closely related to C# and Rust type inference. Inputs are the context Γ and an expression which type we want to infer. The process consists of systematic traversing the expression from bottom to top and deducing the type of sub-expressions following the mentioned rules. The algorithm uses the `Instantiate` method to replace quantified type variables in the expression with new type variables, the `Generelize` method to replace free type variables in the expression with quantified type variables, and the `Unify` method, also known as *Unification* in Logic. Unification is an algorithm finding a substitution whose application on the unifying types makes them identical. Outputs of this algorithm are the inferred type with a substitution used for the algorithm's internal state.

Note: Mention relation to Method type inference

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

Note: Restriction and possible extensions - subtyping, overloading

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

A basic principle of extending H-M type inference by subtyping is described in Parreaux's work [25], where instead of accumulating type equivalent constraints, we accumulate and propagate subtyping constraints. These subtyping constraints

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

Figure 2.16: Rust code example taken from [21].

assemble a set of types, which have to be inherited by the constrained type variable, or the variable has to inherit them.

Note: Restriction and possible extensions - overloading

Extending H-M type inference by supporting overloading mentions Andreas Stadelmeier and Martin Plumicke's work [24]. 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. As we can see, for each method overload participating in type inference, we have to make an alternative type inference containing constraints obtained only from the overloaded method, excluding other overloads. This algorithm can be improved by excluding overloads that can't be used in the method call to save the branching. However, in the worst case, it still takes exponential time to infer types.

2.4 Rust type inference

Note: Intro

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

Note: Type inference

We use code example 2.16 to show the significant difference between C# method type inference and global Rust type inference. We can see that it can infer a type argument of generic type `vec<T>`, referring to a resizeable array, despite of the type information determining a type of the type argument is given in the later statement.

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

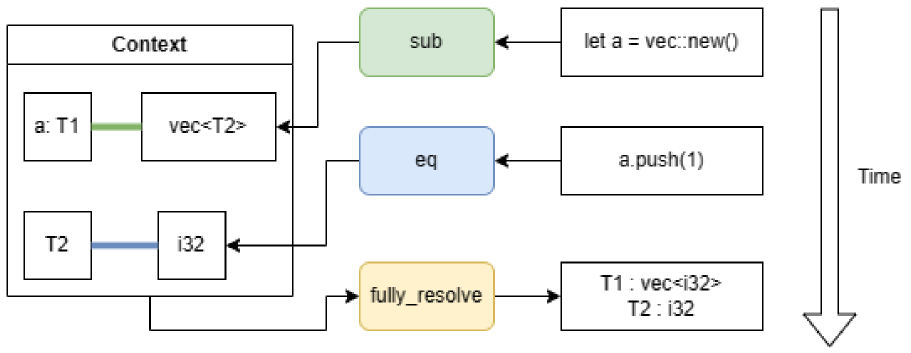

Figure 2.17: Rust type inference

the previous sections, where we extract the required bounds of unbound type variables by finding substitutions for type variables in order to make matching types containing the type variables equal. When there is enough information to resolve type variables, they are resolved by finding an appropriate type for the type variable with respect to collected types.

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

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

2.5 Github issues

Note: Intro

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

Note: Proposal

The process of designing a new language feature starts with publishing an idea into discussions [15], where the C# community can comment on it. The

```
var temp = new A();

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

Figure 2.18: Default type parameters.

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

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

Figure 2.19: Generic aliases.

idea contains a brief description of the feature, motivation, and design. Besides the idea, a new language feature requires a proposal, initially published as an issue, describing the feature in a way that can be later reviewed by the LDM committee. If the proposal sufficiently merits the discussions, it can be marked as a *champion* by a member of LDM for being discussed further by the team. There are several milestones in which the proposal can be. The most important for us is *AnyTime*, meaning that the proposal is not actively worked on and is open to the community to collaborate on it. At the time of writing, a member of LDM recommended championed issue [11] to be investigated since it contains many related discussions with proposed changes but still doesn't have a required proposal. When a proposal has sufficient quality to be discussed by LDM, a member invites the proposer to make a *Pull Request* where further collaboration continues. If LDM accepts the proposal, it is added to the *proposals* folder in the repository for being added into the C# specification, and its future implementation (in Roslyn) will be shipped with the next C# version.

Note: Discussions

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

Note: Discussion - Default type parameters

The first discussion [2] mentions *Default type parameters* introducing default type arguments, which are used when explicit type arguments are not used. Figure 2.18 shows a potential design of this feature where construing generic type `A` doesn't need a type argument since it uses `int` type as a default value.

Note: Discussion - Generic aliases

The next discussion [3] mentions *Generic aliases* allowing to specify default values similar to the goal of the previous discussion by defining an alias to that type with option generic parameters. We can see an example of usage in Figure2.19, where we predefine the first type argument of the `Dictionary` class to be the `string` type, which simplifies the usage of that type in scenarios where we often use dictionaries with keys of the `string` type.

Note: Discussion - Named type arguments

Discussion [5] mentions *Named type parameters*, which are similar to named parameters of methods. The basic thought of this idea is being able to specify a type parameter for which we provide a type argument by name. In Figure 2.20, we can see a generic method `F` with two type parameters. With the current type inference, we have to always specify type arguments in method calls of `F`. We can

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

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

Figure 2.20: Named type parameters.

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

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

Figure 2.21: Using char as inferred type argument.

tell the compiler specific type parameters for which we provide type arguments, `U` in this case, using the named type arguments and letting the compiler infer the rest of the type parameters.

Note: Discussion - Using char as inferred type argument

Comments of the mentioned championed issue [11] propose several keywords that can be used in a type argument list for skipping type arguments, which can be inferred by the compiler and just providing the remaining ones. In example 2.21, we can see the `var` keyword for skipping the first type argument since it can be inferred from the argument list, and we just specify the second type argument, which can't be inferred by the compiler. The comments propose other options for keywords like underscore or whitespaces.

Note: Discussion - Inference based on target

Discussion [7] proposes *Target-typed inference*, where type inference uses type information of the target assigned by the return value. We can see the usage in Figure 2.22, where type inference determines that the return type has to be `int` type and uses that to deduce the type argument `T`.

Note: Discussion - Type inference based on type constraints

The next idea of improving type inference is given by discussion [8] , where type inference utilizes type information obtained from type constraints. A simple example of that can be seen in Figure 2.23, where `T1` can be deduced by using `T1`'s constraint and inferred type of `T2` forming inferred type `List<int>`.

Note: Discussion - Type inference of method return type

Discussion [9] mentions type inference of method return type known from Kotlin language. We can see the usage in the following Figure 2.24, where the return type of method `Add` is inferred to be `int` based on the type of the return

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

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

Figure 2.22: Target-typed inference.

```
var temp = Foo(1);

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

Figure 2.23: Type inference based on type constraints.

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

Figure 2.24: Type inference of method return type.

expression.

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

Issue [6] proposes a way to compact type argument lists of identifiers containing inner identifiers with argument lists. We can see an idea demonstrated in example 2.25, where the argument list of `A<T1>` type and the `Foo<T2>` method are merged, and the type arguments are split by a semicolon.

Note: Discussion - Constructor type inference

The last discussion [1], which we mention here, regards *Constructor type inference* enabling type inference for object creation expressions. The type inference can be seen in Figure 2.26, where the `T` type parameter of the `C<T>` generic type can be deduced by using type information from its constructor.

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

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

Figure 2.25: Specifying type arguments in method calls (Realocation).

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

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

Figure 2.26: Constructor type inference.

3. Problem analysis

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

TODO: Describe the set of related issues.

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

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

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

4. Solution

TODO: Describe process of making proposal and the prototype.

TODO: Describe partial method type inference.

TODO: Describe constructor type inference.

TODO: Describe generic adjusted algorithm for type inference.

TODO: Describe decisions of proposed change design.

TODO: Describe changed parts of C# standard.

5. Evaluation

TODO: Describe achieved type inference. Mention interesting capabilities.

TODO: Note about the performance.

TODO: Links to csharplang discussions.

6. Future improvements

TODO: Mention next steps which can be done.

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

Conclusion

TODO: Describe issue selection.

TODO: Describe proposed changes in the lang.

TODO: Describe the prototype and proposal.

TODO: Mention csharplang discussions.

TODO: Mention observed future improvements.

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

List of Figures

List of Tables

List of Abbreviations

LDT	Language Design Team
LDM	Language Design Meetings
AST	Abstract Syntax Tree

A. Attachments