FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Tomáš Husák

Improving Type Inference in the C# Language

Department of Distributed and Dependable Systems

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

Prague 2024

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

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

In .... date ....

Author's signature

TODO: Dedication.

Title: Improving Type Inference in the C# Language

Author: Tomáš Husák

Department: Department of Distributed and Dependable Systems

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

Abstract:

TODO: Abstract.

Keywords: Type Inference C# Roslyn

Contents

1. Introduction

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

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

1.1 Improving C# type system

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

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

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

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

```
using System.Collections.Generic;

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

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

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

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

Figure 1.2: C# Type inference of generic methods.

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

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

Figure 1.3: Rust Type inference of generic methods.

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

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

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

Type inference capabilities of C# and Rust can be formalized by Hindley-Millner type inference [9] used by these languages in a modified way. Traditional Hindley-Millner type inference is defined in the Hindley-Millner type system [8], where it can deduce types of all variables in an entirely untyped code. The power of type inference is caused by properties of the type system, which, in comparison with the C# type system, doesn't use type inheritance or overloading. Despite these barriers, Hindley-Millner type inference can be modified to work with other type systems like Rust or C#, causing limited use cases where it can be applied.

Note: Present goal of this part of thesis

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

1.2 Implementation

Note: Describe proposing a new feature

C# is an open-source project where the community can contribute by fixing issues of the compiler, proposing new language features, and elaborating on implementing them. Proposing new C# features is done in public discussions of the C# language repository [4], where everyone can add his ideas or comment on others' ideas. Although there is no required structure for how the idea should be described, LDM created a template [6] containing a base structure for proposing the feature in order to make the idea more likely to be discussed by the team. The template includes motivation, detailed description, needed C# language specification [5] changes, and other possible alternatives.

Note: Roslyn

Language feature prototypes are implemented in feature branches of the Roslyn repository [7], which contains an open-source C# compiler developed by Microsoft and the community.

Note: Present goal of the second thesis part

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

1.3 Summary

Note: Goals of this thesis

We summarize goals of this thesis in the following list:

G1. Explore possibilities of type inference in strongly typed languages

G2. Improve C# type inference based on previous analysis

G3. Create an proposal containing the improvement

G4. Implement the prototype in Roslyn

2. Related work

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

2.1 C# programming language

Note: Explain purpose of this section

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

2.1.1 Type system

Note: Value/reference types

Note: Inheritance

As we mentioned in the introduction, each variable and expression returning a value has to have a type in the C# type system [1] called the Common Type System (CTS). Its fundamental characteristic is type inheritance, where every type directly or indirectly inherits a base type `System.Object`, as you can see in the picture 2.1. This chain of inheritance forms a tree, meaning that it is prohibited to inherit more than one type. Types are divided into value and reference types. Value types consist of built-in numeric types, structures (`struct`), and enumeration (`enum`). Compared to classes (`class`) and records (`record`) belonging to reference types, value types can't be inherited by other types. The last relevant member of reference types is interface (`interface`), which can extend multiple interfaces and be implemented by `class`, `record`, or `struct`.

Note: Nullability analysis

C# type inference infers, besides a type, its nullability, determining if it is possible to assign `null` value to that type. C# implicitly allows to assign `null` values to reference types indicating invalid value. Since C# 2.0 [2], it allows to assign `null` values to nullable value types, which are generic wrappers around value types. Because assigning `null` value is referred to as a billion-dollar mistake, C# 8.0 introduces optional settings prohibiting it and created nullable reference types explicitly allowing `null` assignment as a way of interaction with legacy code not using the feature.

Note: Generic types and methods

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

Note: Generic constraints

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

Note: Variance and contra-variance

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

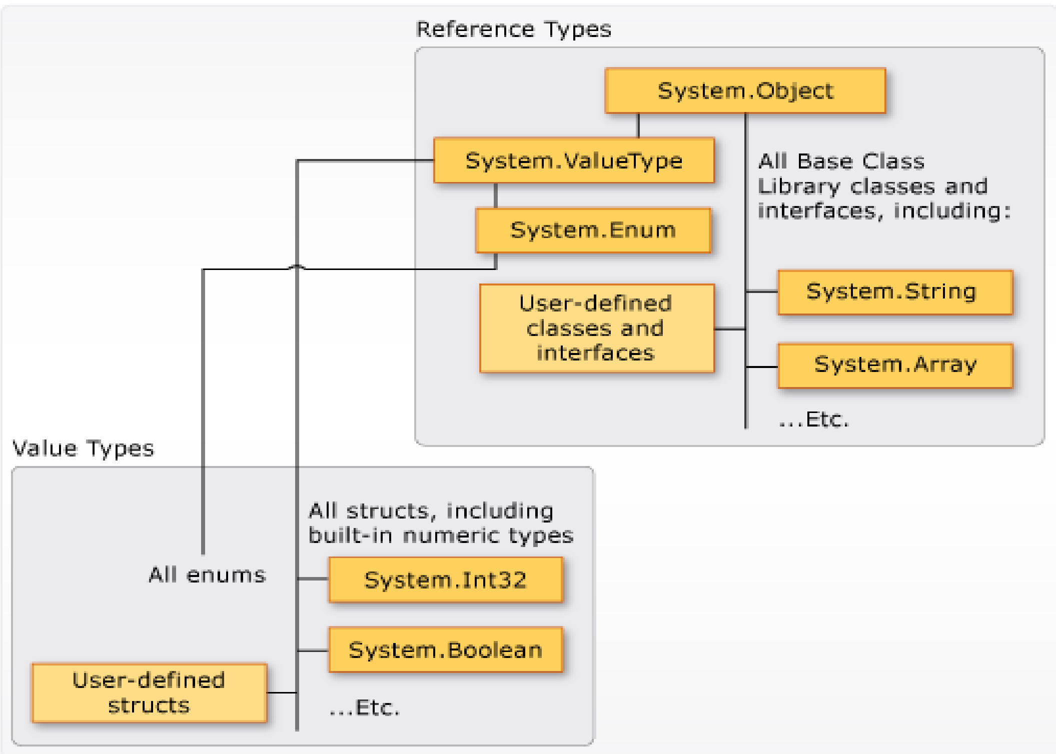

Figure 2.1: The C# type system [1].

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

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

Figure 2.2: C# type constraints.

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

Note: Overloading

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

2.1.2 Relevant constructs

Note: Dynamic

C# type inference mostly happens at compile time, with one exception. We previously mentioned that C# requires knowing the types of all variables and expressions during compilation. It turned out that the possibility of expressing type, which is unknown at compile time, became crucial for interoperability with other dynamic typed languages. To make the interoperability easier, C# introduced a dynamic keyword that can be used as an ordinary type, which causes late binding. Internally, the type is `System.Object`, however, the compiler skips its binding and postpones it to the runtime. This feature creates the exception when type inference happens at runtime, although the compiler still attempts to check expressions containing dynamic values to reveal possible errors. This process will be explained in the following section.

Question: Code example

Note: Implicit typed lambdas

The next language construct influencing type inference is an anonymous function, also known as Lambda, which, instead of declaring a dedicated method with a signature and a body, allows to specify only the body with untyped parameters on places where a function delegate is required. Type inference infers its signature based on the surrounding context.

Question: Code example

Note: Object creation expression and initializers

The last language feature which will take part in the improved type inference are initializers used as a shortcut during an object instantiation. The most simple one is an object initializer that allows to assign values to the object's fields in a pleasant way instead of assigning them separately after the initialization. The second type of initializers regard arrays and collections. Array initializers are

```
var myVariable = "string";
```

Figure 2.3: `var` keyword

```
Helper.MyMethod((long)1, 1, new List<int>(), (p1) => p1 + 1);

class Helper {
  static void
  MyMethod<T1, T2, T3>(T1 p1, T1 p2, IList<T2> p3, Func<T2,T3> p4)
  { ... }
}
```

Figure 2.4: C# Method type inference deduces the method type arguments.

used to create fixed arrays with predefined content. Under the hood, each of the items in the initializer is assigned to the corresponding index of the array after the array creation. Collection initializers are similar to an array initializer defined on collections which are determined by implementing `ICollection<T>` interface. One of the interface's declaring methods is `void Add<T>(T)` with adding semantic. Each type implementing this interface is allowed to use an initializer list in the same manner as an array initializer. It's just a sugar code hiding to call the 'Add' method for each item in the initializer list. The last type of initializer uses indexer to store referred values on predefined positions.

Question: Code example

2.1.3 Type inference

Note: Introduce kinds of type inferences

Note: Method type inference

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

The most interesting and complex context is the method type inference used during generic method call binding when type arguments are not given. We can see a situation when the method type inference deduces `System.String`, `System.Int32` and `System.Int32` as type arguments of the `MyMethod` method in the following code 2.4. We can notice several tasks that the type inference has to be capable of. Regarding the `T1` type parameter, the inference has to find a common type between the first and the second type parameters. Regarding the `T2` type parameter, the type inference has to go into type arguments of the generic type of parameter and the argument, check if the types are compatible, and then match the type parameter against the type argument of the third parameter. The most challenging is lamdas, whose return type has to be inferred after all lambda argument types are inferred.

As we can see, the method type inference is a complex process containing

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

Figure 2.5: C# Method type inference algorithm.

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

Figure 2.6: Target-typed `new()` operator.

many steps. Since one of our improvement is adjusting the algorithm, we present its complete description. The algorithm is divided into four sections to better explain its functionality. Before we show schema, we have to present definitions which are used by the algorithm.

Question: Create citations

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* bounds*. The type variable becomes* fixed *when the its actual type is determined using its* bounds*.*

Definition 2 (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 3 (Dependence)**.** *An unfixed type variable X_i* depends directly on *an unfixed type variable X_e if for some argument `E` X_e occurs in an input type of `E` and X_i occurs in an output type of `E`. X_i* depends on *X_e is the transitive but not reflexive closure of* depends directly on.

Note: Algorithm description

The algorithm runs in steps which are repeated until all of type arguments are deduced or fails when there are unsatisfied requirements or not enough information for determining all type arguments.

Note: Array type inference

The third type inference happens in array initializers when the type of the array should be deduced from the initializer list. We can see an example of situation, when the type inference is used for determining `myArray` type in the following code 2.5. The most specialized common type is just adjusted already mentioned type inference algorithm where is just one type parameter and all initializer items are lower bounds of that type variable.

Note: Target-typed inference

The fourth type inference regards inference based on target type. The example of these situations can be seen in the following example where we use target-typed `new()` operator allowing to skipping creating type which is provided by the target type, variable type in this case.

Note: Dynamic checking

The last group what we have to mention is rather type checking than type inference. As we mentioned in the previous section, `dynamic` keyword is used to enable late binding on the expression containing dynamic expression. Because of

Input: a method call $\mathtt{M(E_1, ..., E_x)}$ and its signature $\mathtt{T_e}$
$\mathtt{M<X_1, ..., X_n>(T_1 p_1, ..., T_x p_x)}$

Output: $X_1, ..., X_n$

$B_{lower}, B_{upper}, B_{exact}, F \leftarrow [\ ]$
`FirstPhase()`
`SecondPhase()`
fn `FirstPhase()`:
    foreach E_i **do**
        if $E_i \in AnonymousFuncs$ **then**
            `InferExplicitParameterType(`E_i, T_i`)`
        else if E_i *has a type* U **then**
            switch U **do** `/* At most one case is executed */`
                case `isValParam(`p_i`)` **do** `InferLowerBound` (U, T_i)
                case `isRefParam(`p_i`)` $\vee$ `isOutParam(`p_i`)` **do**
                    `InferExact(`U, T_i`)`
                case `isInParam(`p_i`)` $\wedge$ `isInArg(`E_i`)` **do**
                    `InferExact(`U, T_i`)`
                case `isInParam(`p_i`)` **do** `InferLowerBound(`U, T_i`)`

fn `SecondPhase()`:
    $X_{indep} \leftarrow \forall X_i : F[i] = null \wedge \nexists X_e : dependsOn(X_i, X_e)$
    $X_{dep} \leftarrow \forall X_i : F[i] = null \wedge (\exists X_e : dependsOn(X_e, X_i) \vee B_{lower}[i] \cup B_{exact}[i] \cup B_{upper}[i] \neq empty)$
    switch X_{indep}, X_{dep} **do**
        case $X_{indep} \neq empty$ **do**
            foreach $X_s \in X_{indep}$ **do** `Fix(`X_s`)`
        case $X_{indep} = empty \wedge X_{dependent} \neq empty$ **do**
            foreach $X_s \in X_{indep}$ **do** `Fix(`X_s`)`
        case $X_{indep} \cup X_{dep} = empty$ **do** `Failed()`
        otherwise do
            foreach $E_i : \exists X_e : F[e] = null \wedge X_e \in$ `outTypes(`T_i`)` $\wedge X_e \notin$ `inTypes(`T_i`)` **do** `InferOutputType(`E_i, T_i`)`
            `GoTo(`*14*`)`

Algorithm 1: Phases of Method Type Inference

```
fn InferOutputType(E, T):
    switch E do
        case E ∈ AnonymousFuncs ∧ T ∈ DelegateTypes ∪ ExprTreeTypes do
            InferLower(InferReturnType(E), T.ReturnType)
        case E ∈ MethGroups ∧ T ∈ DelegateTypes ∪ ExprTreeTypes do
            T_ret, T_1, ...T_n ← T.ReturnType, T.ParamTypes
            E_resolved ← OverloadResolution(E, T_1, ...T_n)
            if ‖E_resolved‖ = 1 then
                InferLower(E_resolved[1].ReturnType, ret)
        case E ∈ Expressions ∧ E has a type U do InferLower(U, T)
fn InferExplicitParameterType(E, T):
    if E ∈ ExplicitlyTypedAnonymousFuncs ∧ T ∈ DelegateTypes ∪ ExprTreeTypes ∧ ‖E.ParamTypes‖ = ‖T.ParamTypes‖ then
        foreach U_i, V_i : U_i ∈ E.ParamTypes ∧ V_i ∈ T.ParamTypes do
            InferExact(U_i, V_i)
fn InferExact(U, V):
    if ∃ i : V = X_i ∧ F[i] = null then B_exact[i].Add(U)
    switch V do
        case V = V_1[...] ∧ U = U_1[...] ∧ V.Rank = U.Rank do
            InferExact(U_1, V_1)
        case V = V_1? ∧ U = U_1 do InferExact(V_1, U_1)
        case V = C<V_1, ...V_e> ∧ U = C<U_1, ...U_e> do
            foreach V_i, U_i do InferExact(V_i, U_i)
```

Algorithm 2: *Output type inference*, *Explicit parameter type inference*, *Exact inference*

```
fn InferLower(U, V):
    if ∃ i : V = X_i ∧ F[i] = null then B_lower[i].Add(U)
    switch V, U do
        case V = V_1? ∧ U = U_1? do InferLower(V_1, U_1)
        case V = V_1[...] ∧ U = U_1[...] ∧ V.Rank = U.Rank do
            if !isRefType(U_1) then InferExact(U_1, V_1)
            else InferLower (U_1, V_1)
        case V ∈ {IEnumerable<V_1>, ICollection<V_1>,
         IReadOnlyCollection<V_1>, IReadOnlyCollection<V_1>,
         IList<V_1>} ∧ U = U_1[ ] do
            if isRefType(U_1) then InferExact(U_1, V_1)
            else InferLower (U_1, V_1)
        case V = C<V_1, ..., V_e> ∧ ∃! C<U_1, ..., U_e> : IdenticalTo(U,
         C<U_1, ..., U_e>) ∨ Inherits(U, C<U_1, ..., U_e>) ∨ Implements(U,
         C<U_1, ..., U_e>) do
            switch V_i do
                case isCovariant(V_i) do InferLower(U_i, V_i)
                case isContravariant(V_i) do InferUpper(U_i, V_i)
                case isInvariant(V_i) do InferExact(U_i, V_i)

fn InferUpper(U, V):
    if ∃ i : V = X_i ∧ F[i] = null then B_upper[i].Add(U)
    switch V, U do
        case V = V_1? ∧ U = U_1? do
            if !isRefType(U_1) then InferExact(U_1, V_1)
            else InferUpper (U_1, V_1)
        case V = V_1[...] ∧ U = U_1[...] ∧ V.Rank = U.Rank do
            if !isRefType(U_1) then InferExact(U_1, V_1)
            else InferUpper (U_1, V_1)
        case U ∈ {IEnumerable<U_1>, ICollection<U_1>,
         IReadOnlyCollection<U_1>, IReadOnlyCollection<U_1>,
         IList<U_1>} ∧ V = V_1[ ] do
            if isRefType(U_1) then InferExact(U_1, V_1)
            else InferLower (U_1, V_1)
        case U = C<U_1, ..., U_e> ∧ ∃! C<V_1, ..., V_e> : IdenticalTo(V,
         C<V_1, ..., V_e>) ∨ Inherits(V, C<V_1, ..., V_e>) ∨ Implements(V,
         C<V_1, ..., V_e>) do
            switch U_i do
                case isCovariant(U_i) do InferUpper(U_i, V_i)
                case isContravariant(U_i) do InferLower(U_i, V_i)
                case isInvariant(U_i) do InferExact(U_i, V_i)
```

Algorithm 3: *Upper-bound inference*, *Lower-bound inference*

fn $\texttt{Fix}(X_i)$:

$U_{candidates} \leftarrow B_{upper}[i] \cup B_{exact}[i] \cup B_{lower}[i]$

foreach $U \in B_{exact}[i]$ **do**

$\quad U_{candidates} \mathrel{-}= \{U_r : U_r \in U_{candidates} \wedge \texttt{!IdenticalTo}(U_r,\ U)\}$

foreach $U \in B_{lower}[i]$ **do**

$\quad U_{candidates} \mathrel{-}= \{U_r : U_r \in U_{candidates} \wedge \texttt{!hasImplicitConversion}(U,\ U_r)\}$

foreach $U \in B_{upper}[i]$ **do**

$\quad U_{candidates} \mathrel{-}= \{U_r : U_r \in U_{candidates} \wedge \texttt{!hasImplicitConversion}(U_r,\ U)\}$

if $\exists!\ V : V \in U_{candidates} \wedge \forall\ U_e \in U_{candidates} : \texttt{hasImplicitConversion}(U_e, V)$ **then** F[i] = V

else Failed()

Algorithm 4: Fixing of type variables

```
dynamic dynamicValue = null;
Helper.MyMethod(1, "string", dynamicValue);

class Helper {
 static
 void MyMethod<T1, T2>(T1 p1, T1 p2, T2 p3)
  { ... }
}
```

Figure 2.7: Type inference with dynamic argument.

late binding, the compiler can't check instance method calling at compile time to prevent type errors at runtime. However, there are situations, where the compiler can partial type check expressions like method calls. You can see a situation, where despite of late-binding there is an obvious error where we try to pass arguments with `System.Int32` and `System.String` types to parameters with the same type. Method type inference as we presented doesn't allow to infer `System.Object` when at least one bound doesn't contains that type causing type inference error at runtime. To prevent these situations, where error are not related to dynamic values, the compiler tries to infer the types ignoring the dynamic values. In case of type conflicts in the variable's bounds, the error is announced at compile time instead of runtime.

2.2 Roslyn

Note: Intro

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

TODO: Overview of compilation pipeline

TODO: Binder

TODO: OverloadResolution

TODO: MethodTypeInferrer

TODO: NullableWalker

TODO: Dynamic biding vs. runtime binding

2.3 Hindley-Millner type inference

TODO: Hindley-Millner type system

TODO: Set of rules

TODO: Restriction and possible extensions

2.4 Rust type inference

TODO: Rust type system

TODO: Type inference context

TODO: Type inference across multiple statement

TODO: Constructor type inference

2.5 Github issues

TODO: Mention related Github issues and csharplang repo.

TODO: Roslyn and csharplang repo

TODO: Proposal champions

TODO: Related issues

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

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

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

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

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

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

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

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

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

List of Figures

List of Tables

List of Abbreviations

LDM Language Design Team

CTS Common Type System

A. Attachments