



**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	2
1.1	Improving C# type system	2
1.2	Implementation	3
1.3	Summary	4
2	Related work	5
2.1	C# type inference	5
2.1.1	Type system	5
2.1.2	Other constructs	6
2.1.3	Type inference	7
2.2	Roslyn	7
2.3	Hindley-Millner type inference	7
2.4	Rust type inference	7
2.5	Github issues	7
3	Problem analysis	8
4	Solution	9
5	Evaluation	10
6	Future improvements	11
	Conclusion	12
	Bibliography	13
	List of Figures	14
	List of Tables	15
	List of Abbreviations	16
A	Attachments	17

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 an implementation of the improvement in the current C# compiler and creating a proposal that will be likely 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 which is achieved by determining types of data at compile time. The easiest way how a compiler reasons about types of variables in the code is providing type annotations determining data type which 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 `people` variable guarantees us that any possible changes to threat it in the different way from `List<T>` will be reported at compile time to save programmer time to debug it. On the other hand, the programmer has to write more code to annotate a variable declaration, whose type has 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 on the same place, the declared variable `people` has to have the same type as the initializing value. Use of type inference can be seen in the `myFriend` variable declaration where we used `var` keyword triggering C# type inference to determine variable's type being the type of initializing value which is `System.String` in this case.

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

Scenarios where type inference is able to 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#, generic method is a method which is parametrized by types besides common parameters as you can find in the code 1.2. Although the type inference deduces type arguments of the first generic method `Foo`, it fails

```
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.

to deduce type arguments of `Bar` even it could be possible in this case since we now the return type of the method.

When we compare it with an 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 [7] used by these languages in modified way. Traditional Hindley-Millner type inference is defined in Hindley-Millner type system [6] where it is able to 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 C# type system doesn't use type inheritance or overloading. Despite of these barriers, Hindley-Millner type inference can be modified to work with other type systems like Rust of C# causing limited use cases, where it can be applied.

Note: Present goal of this part of thesis

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

1.2 Implementation

Note: Describe proposing a new feature

C# is an open-source project where community can contribute by fixing issues of the compiler, propose new language features and elaborate on implementing them. Proposing new C# features is done in public discussions of C# language repository [2] where everyone can add his idea, or comment others' ideas. Although there is no required structure how the idea should be described, Language Design Team (LDM) created a template [4] containing a base structure for proposing the feature in order to make the idea more likely to be discussed by the team.

The template consists of motivation, detailed description, needed C# language specification [3] changes, and other possible alternatives.

Note: Roslyn

Language feature prototypes are implemented in feature branches of Roslyn repository [5], which contains an open-source C# compiler developed by Microsoft and community.

Note: Present goal of the second thesis part

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

1.3 Summary

Note: Goals of this thesis

We summarized 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

This chapter describes C# type inference together with its injection into the Roslyn compiler. Then we compare it with traditional Hindley-Milner type inference and its variance in Rust language. In the end, we present C# issues presented on the GitHub repository which we use later to prioritize the improvement to make it more likely to be accepted by LDM.

2.1 C# type inference

2.1.1 Type system

Note: Type system(struct, class, record, and interface) + Inheritance

Type inference is dependent on a type system. Since C# is a strongly typed language, each variable or expression returning a value has to have a type in the C# type system [1] called Common Type System (CTS). The main fundamental characteristic is type inheritance. Every type directly or indirectly inherits a base type `System.Object`. We can further divide types into value and reference types. Value types consist of built-in numeric types, structures (`struct`), and enumeration (`enum`). Reference types consist of classes(`class`), records (`record`), and interfaces (`interface`). Besides its own semantics during runtime, there are other implications during compile time. Built-in numeric types and structures directly inherit `System.ValueType` reference type. Enumerations directly inherit `System.Enum` reference type. In comparison with reference types, value types can't be inherited by other types. However, they can implement multiple interfaces. An interface can extend multiple interfaces and a class or record can extend other class or record. They can also implement multiple interfaces. It's prohibited to inherit more than one type.

Note: Overloading, nested types

Types can contain fields, methods, and other nested type definitions. A field is a data holder containing data of its defined type. A method consists of a body and a signature describing its name, parameters, and return type. Another important characteristic of CTS is overloading where a type can contain multiple methods with the same name differing in number or types of parameters.

Note: Nullability analysis

The type system allows to assign `null` value to a reference type meaning an invalid reference. This feature is usually referred to as a billion-dollar mistake. In order to fix it, late C# versions added a voluntary nullable analysis prohibiting assigning `null` to reference types. It offers nullable types by adding question mark `?` to a type identifier to be able to assign `null` to that type as an option to interact with older code that doesn't use it.

Note: Dynamic

C# language is one of the languages contained in the .NET ecosystem. One of the goals of this ecosystem is to provide easy interaction with other .NET-compliant languages. Although Common Intermediate Language (CIL) of .NET is strongly typed, there are projects enabling a compilation of dynamically typed

language to the CIL. C# `dynamic` keyword was introduced for interaction with these languages skipping type checking of values marked as `dynamic`. These values have a type of `object` and its method or type references are resolved during runtime.

Note: Generics(struct, class, method, and interface)

Generics are usually used to make code reusable by parametrizing types or methods by other entities than values. C# generics allow to parametrize types and methods by types. These type parameters can be then used as a normal type identifier. One example can be seen in `System.List<T>` class representing resizable mutable array where the functionality is the same for every type `T`. Providing type arguments to a generic type or method is called a construction of a generic type or method respectively.

Note: Generics(where clauses, invariance, variance, and contra-variance)

To keep type safety in generic code, C# treats unrestricted type parameters as a `object`. Several types of restrictions can be applied to type parameters in order to enable more actions on values of the restricted type parameters. These restrictions are checked by a compiler at the time of construction. Inheritance restriction is the most common one guaranteeing that the type argument will directly or indirectly inherit the given type. Another restriction concerns an obligation to have an empty constructor, or the type argument has to be a structure. Normally, type parameters are invariant meaning an obligation to assign a generic type to another generic type having the same types of type parameters. Generic interfaces introduce additional modifiers of type parameters. We can annotate a type parameter as a type variant meaning a possibility to assign a more specialized type to the less specialized type. There is also a contra-variance meaning the opposite thing.

2.1.2 Other constructs

Note: Implicitly typed lambdas

Anonymous functions also known as Lambdas are frequently used language features. Instead of declaring a dedicated method with a signature and a body, it allows to specify just the body with parameters on places, where a function delegate is required.

Note: initializers

Initializers are 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. Array initializers are 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. Collections are types 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.

2.1.3 Type inference

TODO: var, target-typed new, target-typed ternary operator, target-typed lambdas

TODO: Method type inference

2.2 Roslyn

TODO: Overview of compilation pipeline

TODO: Binder

TODO: OverloadResolution

TODO: MethodTypeInferer

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 csharp-lang repo.

TODO: Roslyn and csharp-lang 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 csharp-lang 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 csharp-lang 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] csharp-lang repository. <https://github.com/dotnet/csharp-lang>, . [Online; accessed 2023-09-22].
- [3] C# specification. <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/readme>, . [Online; accessed 2023-09-22].
- [4] Proposal template. <https://github.com/dotnet/csharp-lang/blob/main/proposals/proposal-template.md>, . [Online; accessed 2023-09-30].
- [5] Roslyn repository. <https://github.com/dotnet/roslyn>, . [Online; accessed 2023-09-22].
- [6] Hindley-milner type system. https://en.wikipedia.org/wiki/Hindley%E2%80%93Milner_type_system, . [Online; accessed 2023-09-22].
- [7] Hindley-Milner type inference. <https://www.youtube.com/watch?v=B39eBvapmHY>, . [Online; accessed 2023-09-22].

List of Figures

1.1	Type annotations in the C# programming language.	2
1.2	C# Type inference of generic methods.	3
1.3	Rust Type inference of generic methods.	3

List of Tables

List of Abbreviations

LDM Language Design Team

CTS Common Type System

CIL Common Intermediate Language

A. Attachments