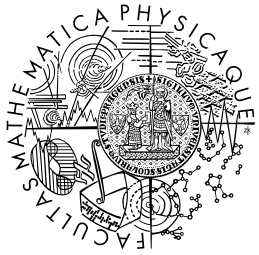

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

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

Introduction

Note: Describe what is type inference.

Statically typed languages have many advantages like revealing bugs in compilation time or high performance. To achieve these benefits, the languages demand type annotations from a programmer. These type annotations define types of program data during runtime protecting operations on incompatible data. Because code usually contains a lot of variables whose type has to be known during compilation time, type inference was introduced to eliminate type annotations that can be deduced from a context. Type inference tries to deduce a type of variables using a context, where the variables are used. As an example of the context, we can take a variable which is passed as an argument to a function. The type of the parameter has to be compatible with the type of that variable.

Note: Describe type inference in C#.

C# is a statically typed language whose type system, besides common primitives and classes known from other languages, contains generics. Generics are used for parametrizing types or methods in order to create reusable code(e.g. containers). C# generics parametrizes types or methods by other types. The main feature of the C# type inference is getting rid of type arguments of a generic method in cases, where the arguments can be deduced from a context. Despite type inference being a very useful feature, possible scenarios where it can be applied are restrictive in comparison with other languages.

Note: Compare it with type inference in Rust or Haskell as an example of Hindley-Millner type inference.

As an example of advanced type inference, we can mention Rust language [5]. Although the type system has differences from C# type system, the type inference is done across multiple statements which is much more powerful than the former one. One of those reasons regards specifics of the type system, which enables to use Hindley-Millner type inference [7]. Traditional Hindley-Milner type inference is defined in Hindley-Millner type system [6] which has different characteristics from C# or Rust. Although the most powerful is in that type system, it can be adjusted to work in type systems in already mentioned languages.

Note: Describe C# specification

C# type inference is a variant of Hindley-Millner type inference and the algorithm can be found in the language specification [3]. The specification consists of all language features described independently on the compiler implementation. As the language evolves, the specification also changes. These changes are done publicly on a Github repository [2] to offer participation in creating the specification for the outside community.

Note: Describe Roslyn.

The current implementation of C# type inference is contained in Roslyn [4]. Roslyn is an open-source C# and VB compiler developed by Microsoft and the community. Since it is open source, it is possible to investigate and participate in implementing new features to the C# language.

Note: Mention CSharplang repo, community, and describe a process of accepting lang changes.

A common process to make a new possible language feature to be merged into Roslyn is making a proposal and a prototype. A proposal is a description of

the new feature consisting of motivation, detailed description, needed language specification changes, and other possible alternatives. The proposal is published in Github discussions of mentioned repositories where the community can share their opinion. If the proposal is promising enough, the language committee will choose it for further discussion. The feature is added to the language if Language Design Team (LDM) accepts the proposal. After that, the implementation of that feature can be merged into Roslyn. During the proposal, a prototype is usually done to demonstrate the feature in wider usage and to explore possible barriers in the implementation.

Note: Goal of this thesis

The goal of this thesis is to create a proposal regarding improving C# type inference in order to offer more power type deductions as we can see in other similar languages. To make the proposal more likely to be accepted by the LDM, a prototype is created to estimate the level of implementation difficulty in the production C# compiler.

Note: Give an overview of chapters.

The first chapter describes the Roslyn compiler together with a theoretical background of type inference. The second chapter describes the scope of the language improvements based on community preferences. Then, It describes difficulties regarding the architecture of C# language and the compiler implementation. Based on this knowledge, we propose an improvement. At the end of this chapter, we propose the improvement and benefits which the improvement brings. The third section consists of the architecture design of implementation together with a new type inference algorithm and changes made in the specification. The fourth chapter contains an evaluation of the implemented improvement. The last chapter discusses possible future features as a continuation and other possible interactions with already existing language features.

1. 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 it's 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.

1.1 C# type inference

1.1.1 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 resizeable 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.

1.1.2 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.

1.1.3 Type inference

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

TODO: Method type inference

1.2 Roslyn

TODO: Overview of compilation pipeline

TODO: Binder

TODO: OverloadResolution

TODO: MethodTypeInferrer

TODO: NullableWalker

TODO: Dynamic biding vs. runtime binding

1.3 Hindley-Millner type inference

TODO: Hindley-Millner type system

TODO: Set of rules

TODO: Restriction and possible extensions

1.4 Rust type inference

TODO: Rust type system

TODO: Type inference context

TODO: Type inference across multiple statement

TODO: Constructor type inference

1.5 Github issues

TODO: Mention related Github issues and csharplang repo.

TODO: Roslyn and csharplang repo

TODO: Proposal champions

TODO: Related issues

2. 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.

3. 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.

4. Evaluation

TODO: Describe achieved type inference. Mention interesting capabilities.

TODO: Note about the performance.

TODO: Links to csharplang discussions.

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

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

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

[7] 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

CIL Common Intermediate Language

A. Attachments