ABSTRACT

Title of dissertation: ADDING GRADUAL TYPES TO EXISTING LANGUAGES: A SURVEY

Type system defines sets of rules, which is checked against programs to detect certain kind of errors and bugs. Type checking happens either statically at compile time, or dynamically at runtime. Programming languages make different choices about type system.

Dynamically typed languages stand out when it comes to scripting and fast prototyping: programs can be easily modified and executed without need of enforcing consistency and correctness throughout the program. But as the program grows over time to handle more complicated tasks, it becomes harder to maintain.

Statically typed languages are more reliable and efficient. Because type errors are caught ahead of execution, programs are free of these errors at runtime. Also due to type information are available statically, more opportunities of optimization can be discovered. However, a new feature or design change in such program could introduce great refactoring efforts to not just implementation but also a redesign of types.

It is only natural that we start to think about a middle ground between two. One prevailing approach is gradual typing: in such system, all type annotations are optional. Programmers can choose to give types to some values or procedures and

later remove them or vice versa. And the type system will only check portions that are type-annotated, therefore allows gradual and smoother evolution of programs.

Nonetheless, theoretical works on a gradual typing is far from sufficient to prove its usefulness: a practical language needs a user base for its community to grow, efforts are required to develop packages for commonly used data structures, algorithms, etc. Fortunately this problem can be solved by extending existing lanugages, thus taking advantage of their communities and large amount of libraries. Several works has done in this regard. And this survey walks through some of them, discussing challenges and solutions of extending a language to support gradual typing.

Table of Contents

Chapter 1: Introduction

Type system defines sets of rules about programs, which can be checked by a machine. Using a type system, we can assign types to various language concepts like variables, functions and classes and describe how they interact with each other. This is important for programming languages, as this allows rule violations to be detected by machines and prevents erroneous part of the program from execution.

The process of checking programs against type rules is called type checking. It happens either statically at compile time, or dynamically at runtime. They both have their advantages and disadvantages. For programming languages, different design choices are made to accommodate their needs.

1.1 Strength and Weakness of Both

In this section we will discuss pros and cons of statically and dynamically typed languages, which leads to the motivation of gradual typing - an approach that takes the benefits from two.

1.1.1 Dynamically Typed Languages

A dynamically typed language checks type at runtime, which is best suited for scripting and fast prototyping. For example, a shell script runs other executables and feeds one's output to another, effectively gluing programs together. In this case, we have little to no knowledge about the executables ahead of time, making statically type checking impractical. On the other hand, when it comes to prototyping, it is more important for programmers to execute the code and make modifications accordingly than to enforce correctness and consistency through whole program. In such scenario, it is more convenient to delay type checking until it is required at runtime.

However, a dynamically typed language is not without weakness. Despite that type checking does not happen ahead of execution, programmers usually have facts about behaviors of programs, which is often described in comments or through naming of variables and functions without a systematic way of verifying them. As program develops, it is easy to make changes and forget updating these descriptions accordingly, which forces future maintainers to go back and rediscover these facts, causing maintenance difficulties.

Furthermore, without type information statically available, dynamically typed languages have to maintain runtime information and rely on it to implement expected semantics. For example, if it is allowed for a language to condition on non-boolean values, its type has to be available at runtime in order to determine how to cast it into a boolean value.

Optimization opportunities could also be missed for a dynamically type language: array cloning can be implemented efficiently as memory copy instructions knowing the array in question contains only primitive values but no reference or pointers, whereas without prior knowledge like this, every element of the array has to be traversed and even recursively cloned.

1.1.2 Statically Typed Languages

Statically typed languages, on the other hand, are better alternatives when reliability and performance are important for a program. These languages type checks during compilation, and ill-typed programs are prevented from execution. With expected behavior described in terms of types, type checking not only ensures that certain kind of errors cannot happen at runtime, but also improves maintainability: it is required for programmers to keep the consistency between type and code, because programs will not compile otherwise.

Having static known type information also helps in terms of efficiency. For example, In many programming languages, arithmetic functions are polymorphic. It is allowed for **a** and **b** to have different numeric types in **a + b**, a cast will be inserted to ensure arithmetic primitives only deals with addition of compatible types: if we are adding integer **b** to a floating number **a**, then **b** will be casted so we can call addition primitive that expected floating numbers on both sides. A dynamic language would have to figure out types of **a** and **b** at runtime, make decision about whether casting is required or which primitive addition to call all at runtime. But with type information statically available, these decisions can be made at compile time, resulting in improved efficiency.

The disadvantage of a statically type language often lies in the lack of flexibility: if a program need to deal with data whose structure is unknown at compile time, while a dynamically typed language allows inspecting data of unknown type, it requires more work for a statically typed language to accept such a program: because it is impossible to make a precise type about unknown data, programmers will need extra bookkeeping at runtime to convince the type checker, effectively creating a type system of their own.

In addition, it is often less convenient for programmers to prototype in statically typed languages: it involves trial and error to find the best solution to a problem, which means the ability to write partial programs, make frequent changes to structure of variables and functions, but these features are not available for a statically typed language.

1.2 Combining benefits of two

One might have noticed that static type checking and dynamic one are not mutually exclusive: it is possible to have a program type checked statically, while allowing runtime type checking. Indeed, several attempts are motivated to put these two type systems in one, in hope of bringing in the benefit of both. In this section, we will discuss about these works.

1.2.1 Design Goal

Summarizing pros and cons of static and dynamic typed languages, we now have a picture of an ideal type system, it should have following features:

- **Detect and rule out errors ahead of program execution.** This is one main purpose of having a static type system.
- **Delay type checking until needed at runtime.** This allows ill-typed or partially written programs to execute. Note that this also implies that it should be possible for type checking at runtime for this type system.

Aside from these two goals within type system, it should be possible for a language equipped with such a type system to make use of the type information available. As this allows better optimization and type safety both statically and dynamically.

1.2.2 A Introduction to Gradual Typing

Gradual typing is one possible approach that meets our design goal. It is an extension to static type system. By introducing a special type, it allows static and dynamic typing to coexist in one type system.

1.2.2.1 Dynamic Type and Type Consistency

Gradual typing introduces a special type, which is usually called a "dynamic" type (we will use notation **Dyn**). Such type captures the idea that type information might be partially known or unknown at compile time.

With this dynamic type introduced, it is possible for type system to work even if type information is only partial. In such a case, the type system only rejects programs that have inconsistencies in known parts of types.

(TODO: type consistency rules here)

For example, type **Dyn** $\times$ **number** is consistent with type **Dyn**; **boolean** $\rightarrow$ **Dyn** is consistent with type **Dyn** and **Dyn** $\rightarrow$ **number**. But **Dyn** $\rightarrow$ **Dyn** is never consistent with **Dyn** $\times$ **Dyn**, because the former is a function type while the latter a pair.

This extension also allows programs without any type information to pass static type checking: when a type is expected for a term but no type information is available, the term is assumed to have a dynamic type. Therefore no inconsistency can be detected statically to reject the program.

1.2.2.2 Optional Type Annotation and "pay-as-you-go"

Gradual typing enables static and dynamic typing within one type system, but what more important is that it creates a middle ground in which programs can be partially typed: all type annotations are not required for a gradual type system, but when programmers do annotate terms with types, the type system will be responsible to make sure that type annotations are consistent within programs.

For example, one might wish to implement a function f that expects a string value as input. There are two approaches: we can either give f a type annotation:

f : **string** $\rightarrow$ **Dyn**. By doing so, type system takes the responsibility of making sure that the input type is indeed a string. Alternatively, we can inspect type of the input value at runtime ourselves.

A static type system allows us to do the former, but a return type must either be given explicitly or inferred. And a dynamic type system does the later, but we cannot statically detect any mistake like applying a number literal to f. However, a gradual type system allows us to give input value a type without worrying about giving a return type if not needed. More generally, programmer only pays for what they want from a type system.

1.2.2.3 Runtime Checks and Cast Insertion

When dynamic types are present, it is not always possible to type check the whole program statically: applying a variable x : **Dyn** to a function f : *number* $\rightarrow$ **Dyn** is consistent, but at runtime x might turn out to be a function, which becomes inconsistent with f. Therefore, gradual typing inserts type casts into the program to allow more type errors to be detected at runtime.

1.2.3 Alternatives

There are other alternatives to gradual typing.

Soft typing: introduces type inference for dynamically-typed languages.

Contracts: is a runtime system.

Liquid type: uses logic terms and verify them through a solver.

Refinement type: works similarly?

Dependent type

1.2.4 Discussion

Despite that there are various research works that combines benefit of static and dynamic type checking gradual typing remains one of the most popular approach: thanks to gradual guarantee, an existing language can be extended to support gradual typing while all existing programs written in it is still valid with no difference in semantics. In addition, the cost of gradual typing pays as programmers need.

However, it is not a straightforward work to support gradual typing for existing languages. In chapter 2, we will explore some research works that extends existing popular languages. Common challenges and solutions while extending an existing language is discussed in chapter 3. Open questions in chapter 4.

Chapter 2: Extending existing languages

It is not a trivial work to extend existing languages to support gradual typing, as it involves understanding not just language features, but also idiom and practice commonly used among its community to allow a smooth and non-intrusive experience.

In these chapter we will go through some works that extend existing programming language to support gradual typing.

2.1 From JavaScript to Safe TypeScript

JavaScript is a dynamically typed language widely used in web techniques. One noticeable work of JavaScript extension is Safe TypeScript by Rastogi, Swamy, Fournet, Bierman & Vekris. It features a sound type system and efficient runtime type information(RTTI)-based gradual typing.

2.1.1 Type Checking for Free

In JavaScript, one cannot declare a variables to have a specific type, which usually leads to the style in which function checks its argument type at runtime before proceeding with actual implementation:

```
function f(x) {
    if (typeof x !== 'string') {
        // throw error
    }
    return x + '!';
}
```

By extending the language with an optional type annotation, the code is cleaner in Safe TypeScript:

```
function f(x : string) {
    return x + '!';
}
```

Here the logic for checking whether x is a string is removed, instead, Safe TypeScript compiles it to insert code for type checking, and throws with error messages upon failure.

By introducing type annotations, programmers get not just dynamic type checking without much effort, but also enables static type checking. Consider the following code with f above in scope:

```
f('one'); // good
f(1); // bad
```

While manual type checking in function bodies throws when the bad call is invoked, Safe TypeScript is capable of capturing the error ahead of execution.

2.1.2 Nominal class and structural interfaces

NOTES

(probably showcasing)

Object as map

Object in the sense of OOP

2.1.3 RTTI and Performance

2.2 Typed Scheme

occurrence typing

simple macros

2.3 Reticulated Python

structural typing to match for Python's duck typing

CSharp (this being the only static language that wants interaction with dynamic ones)

syntactic improvement dynamic Object ("Expando Object")

2.4 C♯ 4.0

2.5 Gradualtalk

TODO

Chapter 3: Challenges and Solutions

3.1 Extension to Type Systems

(Need to be re-organized)

introducing a dynamic type (changes to type judgment)

subtyping (consistency and transitivity)

Or: improve type system precision

using predicates constraints as type level computation (requires a type system of sufficient expressiveness)

typing structural data

nominal - structural

3.2 Object-Oriented Programming

object identity

giving ‘this‘ / ‘self‘ special type treatment

inheritance (could merge with subtyping)

variable / member mutation

3.3 Implementation and Performance

cast insertion

Large array

runtime check overhead

language-specific challenges

Chapter 4: Related Work

(TODO) sound gradual typing is nominally alive and well

TypeScript implements ”occurrence typing” (see ”Type Guards and Differentiating Types” of advanced types) and Array as tuple

Chapter 5: Future Work

Chapter 6: Conclusion