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 ahead of execution, or dynamically at runtime. Programming languages make different choices regarding type system to accommodate their specific needs.

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 programs grow 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. In addition, having type information available statically also helps development in many other ways beyond type system, such as type-specialized optimization and improved development tools. 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 looking for a middle ground between them. One of these approaches 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 be responsible of completely checking only portions that are type-annotated in both static and dynamic manner. Gradual typing gives programmer fine grained control over whether type checking should occur, and guarantees the semantics to be consistent when adding or removing type annotations, therefore allowing gradual and smoother evolution of programs.

One crucial advantage of gradual typing is that it can be implemented by extending an existing statically or dynamically typed language: syntactically, the extension often results in a superset of the original language that allows more expressiveness in types, which means the effort of migrating existing code and language users to take advantage of gradual typing is minimized.

In this survey, we will walk through some research works that extends existing languages to support gradual types, discuss challenges and solutions about making these extensions, and look into related works and the future of gradual typing.

Table of Contents

Chapter 1: Introduction

Type system defines sets of rules about programs, which can be checked by a machine in a systematic way. With a type system, one can assign types to appropriate language concepts like variables, functions and classes to describe expected behaviors and how they interact with each other. Then type system takes the responsibility of ensuring that types are respected and type errors prevent erroneous part of a program from execution.

There are other benefits of type systems besides automated checking: types can be used to reveal optimization opportunities, can serve as simple documentation to programmers or provide hints to development tools to automate navigation, documentation lookup or auto-completion.

All these benefits make type system an important part of programming languages, and there are different ways of accommodating languages with type systems. The most noticeable difference is the process of checking programs again type rules, which is also known as type checking. It occurs either statically ahead of program execution, or dynamically at runtime. For a static type system, ill-typed programs are rejected by compiler and no executable can be produced until all type errors are fixed. For a dynamic type system, type information is examined at runtime and

type errors result in abortion of the program or exceptions being raised instead of causing segmentation faults or other worse consequences. Static and dynamic type system both have their advantages and disadvantages, and languages make different choices depending on their needs.

In this chapter, we start off by discussing static and dynamic type systems, which motivates research works that attempted to combine both within one system in order to take benefits from two, which in return gives birth to gradual typing.

1.1 Strength and Weakness of Static and Dynamic Typing

The difference between static and dynamic typing is the difference between whether type checking occurs ahead of execution or at runtime. This section discusses advantages and disadvantages of both.

In this section we use $T_0, T_1, \ldots$ for type variables. **num** and **str** are types of fixed point numbers and strings respectively. And the type notation for functions that takes as input $T_0, T_1, \ldots$ and returns a value of T_2 is $(T_0, T_1, \ldots) \to T_2$. We use $o : T$ to mean that a language object o is assigned a type T. For example $f : (\mathbf{num}, \mathbf{num}) \to \mathbf{num}$ is a function that takes two numbers as input and returns a number. In addition, tuple types are notated as $(T_0, T_1, \ldots)$

1.1.1 Static Type Systems

A static type system is preferred if robustness and performance is the main goal of a language. Because type checking occurs ahead of execution, ill-typed programs

are rejected before it can start, and when a program typechecks, one can therefore expect no type error to occur and no extra cost is paid for typechecking at runtime.

For a static type system to work, all variables, functions, etc. in a program will need to be assigned types. This is usually done by programmers or through the means of type inference, which is a technique that infers types using available type information. This is both an advantage and a disadvantage of a static type system: having type annotations improves readability and since programmers are required to keep the consistency between type and code, type also serves as simple, faithful documentation. But on the other hand, adding and maintaining type annotations can also considered a burden.

Having static known type information also helps in terms of performance in other ways. 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 ahead of execution, resulting in improved performance at runtime.

Besides extra effort of maintaining type annotations, 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 have to use an imprecise type and keep extra information at runtime, effectively creating a dynamic 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 implementation to solve a problem, which means the ability to write partial programs, make frequent changes to structure of variables and functions, but these features are not easily available for a statically typed language.

1.1.2 Dynamic Type System

Dynamic type system does typechecking at runtime, which is best suited for languages designed for scripting and fast prototyping. A shell script, for example, runs other executables and feeds one's output to another, effectively gluing programs together. In such a case, we have little to no knowledge about these executables ahead of time, making statically type checking impractical. And 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, it is also easy to spot weaknesses of dynamic type systems. 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, dynamic type system needs 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 at runtime.

Optimization could also be hindered 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.

1.2 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 typechecked statically, while allowing runtime type checking. While static type system does typechecking ahead of time and avoids runtime overhead, it lacks the ability of using runtime type

information; dynamic type system does typechecking at runtime, but it requires sophisticated work to reduce the amount of unnecessarily repeated runtime checks. In hope of bringing in benefits of both, several attempts are motivated to put these two type systems in one. In this section, we will discuss about these research 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. When a program starts running, type errors should already be eliminated and runtime overhead should be minimized to keep its performance close to that of a static type system.

- **Delay typechecking until needed at runtime.** This is the crucial feature of a dynamic type system, it allows ill-typed or partially written programs to execute, which makes it suitable in situations where static type information is limited or fast prototyping is preferred over robustness or runtime performance.

Aside from these two goals, type annotations can also be used for other purposes like optimization and providing hints to development tools. For a language with this new type system, it should also be possible.

1.2.2 A Introduction to Gradual Typing

Gradual typing is one possible approach that meets our design goal. It captures the fact that type information might be partially known or unknown ahead of execution by introducing a special type that indicates partial types. It attempts to typecheck programs like a static type system does, but delays typechecking on partial types until sufficient type information is available at runtime.

It is originally introduced as an extension to static type system. But it is also possible and practical to support gradual typing by extending a dynamic type system.

In the following sections, notation **dyn** will be used for special types mentioned above, which is also known as dynamic types.

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 $\mathbf{dyn} \times \mathbf{number}$ is consistent with type **dyn**; $\mathbf{boolean} \to \mathbf{dyn}$ is consistent with type **dyn** and $\mathbf{dyn} \to \mathbf{number}$. But $\mathbf{dyn} \to \mathbf{dyn}$ is never

consistent with $\mathbf{dyn} \times \mathbf{dyn}$, 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: $f : \mathbf{string} \to \mathbf{dyn}$. 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 sup-

port 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 TypeScript

JavaScript is a dynamically typed language widely used in web techniques.

One popular extension to it is TypeScript, in which variables are allowed to have types. TypeScript compiler then type checks the program and compile it to JavaScript in return.

One noticeable work of JavaScript extension is Safe TypeScript by Rastogi, Swamy, Fournet, Bierman & Vekris, which shares the same syntax with TypeScript but 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 often 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 improved to have less verbosity 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 Type-safety and encapsulation

2.1.4 RTTI and Performance

2.2 From Scheme to Typed Scheme

```
(define (creal x)
  (cond [(number? x) x]
        [else (car x)]))
```

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