ABSTRACT

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

Type system defines sets of rules, which is checked against programs to prevent 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.

Statically typed languages are 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 the type system, such as type-specialized optimization and improved development tools. While some believes static type systems assist code evolution in long run, a new feature or design change in codebase could introduce great effort to not just implementation but also adjusting type annotations accordingly, which might not be desirable for testing out new ideas on top of an existing large codebase.

Dynamically typed languages stand out when it comes to scripting and fast prototyping: executables from external sources can be used directly without worrying about type information and programs can be easily modified and executed without need of enforcing consistency and correctness throughout the program. But a program grows over time to handle more complicated tasks, the interaction between

different portions of the program becomes harder to track and maintain.

It is only natural that we start exploring the spectrum between them: type systems that enjoy benefits of both. Among one of these is gradual typing, which features a type system that provides optional type checking. In such a system, programmer can give types to only portion of the program. For the portion that has type information, static and dynamic typechecking will work together to ensure type consistency, whereas the untyped portion of the program is left unchecked, which might be desirable when performance or flexibility is concerned. By giving programmers control over when and where should typechecking occur, one does not commit to either static or dynamic typing while losing the benefit of the other.

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. Semantically, adding or removing type annotation does not change the result of programs aside from type errors, which grants gradual and smooth transition between static and dynamic type disciplines.

In this survey, we will walk through the relevant research literature on extending 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 Strengths and Weaknesses 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. **bool**, **num** and **str** are types of boolean values, 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 $(T_0, T_1, \ldots) \rightarrow T_2$. We use $o : T$ to mean that a language object o is assigned a type T. For example $f : (\mathbf{num}, \mathbf{num}) \rightarrow \mathbf{num}$ 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 Systems

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 the benefits of the 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 Goals

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 only be partially known 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.

In the following sections, we will introduce it 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 type system introduces a dynamic type on top of a static type system to capture the idea that type information might only be partially known ahead of execution.

To deal with dynamic types, we introduce type consistency to accommodate type judgments for gradual typing. Figure ? shows rules for type consistency relation.

(TODO: type consistency rules here, lambda calculus and with some ground types)

In general, two types are consistent when their type structures do not conflict with each other. For example, **bool** is consistent with type **dyn** despite that the latter is less precise; **num** $\rightarrow$ **dyn** is consistent with **dyn** $\rightarrow$ **num** and **dyn** $\rightarrow$ **dyn**, because all these types are functions that accepts one value and returns another and **num** and **dyn** are consistent with each other. However **dyn** $\rightarrow$ **dyn** is never consistent with **bool**, because the former is a function type while the latter a boolean value, whose type structures do not match.

By introducing dynamic type, gradual typing provides typechecking ahead of execution that accepts both statically and dynamically typed programs: when no dynamic type is involved, typechecking proceeds in the exactly same way as that of a static type system. There might be no type annotation in dynamically typed programs, in which case types of terms are default to **dyn**, which allows then to typecheck, without raising type errors.

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

So far we have seen that gradual type system introduces a dynamic type, which allows accepting both statically and dynamically typed programs during static typechecking. But what is more important is that it creates a middle ground in which programs can be partially annotated with types: in a gradual type system, terms are either explicitly given a type, or, when type annotation is missing, given **dyn** type. This renders all type annotations to be optional and allows programmers to have more control over what they need from typechecking: for an originally statically typed program, we can now write terms without giving a type, this allows easy prototyping; for an originally dynamically typed program, some parts of it might already been stable and robust enough, annotating these parts with type allows some effort of ensuring correctness being shifted to typechecking.

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: $f : \mathbf{str} \rightarrow \mathbf{dyn}$. 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 finds the balance between two, it allows us to give input value a type without worrying about giving a return type if not needed, therefore 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 $x : \mathbf{dyn}$ to a function $f : number \rightarrow \mathbf{dyn}$ 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

Gradual type system features optional type annotations with soundness, which is compatible with both statically and dynamically typed programs, type safety are preserved and semantics are kept consistent. This makes it an ideal type system that existing languages can be extended to support.

In this chapter, we will explore some research works that extends existing languages to support gradual typing. As we will see, this is not a trivial task: since it requires the extended language to still support existing programs written in the original language with little to none modification, language features, idiom and practice commonly used among its community needs to be carefully understood to allow a smooth and non-intrusive experience.

2.1 From JavaScript to Safe TypeScript

Starting as a scripting language, JavaScript has grown into a language that powers many large web applications. Unfortunately, as a language that exists over a long period of time, the language evolves but many of its flaws are still left as it is for compatibility reasons, hindering productivity of many programmers.

Among tools and extensions to JavaScript that attempt to ease this pain, TypeScript is one closely related to gradual typing: it is a superset of JavaScript that supports optional type annotations, a compiler typechecks TypeScript code ahead of execution and compile code into plain JavaScript source code, making it ready to be executed out of box for JavaScript interpreters. However, TypeScript is intentionally unsound: only static typechecking is performed, which does not prevent runtime type errors. One noticeable improvement to TypeScript 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.

In this section we will visit some language features of Safe TypeScript and see how it improves JavaScript. Note that TypeScript uses **any** for dynamic type.

2.1.1 Type Checking for Free

Despite JavaScript is capable of examining type of values at runtime, it does not have support for type annotations. This leads to a common practice in which function bodies will have code just for checking its argument types and throw errors before proceeding with actual implementation:

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

```
}
```

As a toy example, the function accepts a string value, then returns another with exclamation sign concatenated to it. But imagine in real projects there will be multiple arguments to a function and some of them have to go through this process of checking, it will soon become less maintainable.

By extending the language with type annotations, we can do something better in Safe TypeScript:

```
function f(x : string) : string {
    // now this check becomes unnecessary
    if (typeof x !== 'string') {
        // throw error
    }
    return x + '!';
}
```

In the code above, we just declared a function **f** that accepts one variable x of **string** type, and returns a value of **string** type. The code for runtime checking is removed, instead, Safe TypeScript performs typechecking and insert casts when needed to ensure that, when **f** is called, its argument is indeed a **string**. This renders the **typeof** check at the beginning useless: when we enter the function body **x** is guaranteed to be a **string**, therefore the body of the **if** statement cannot possibly be entered. This allows us to get rid of the check totally:

```
function f(x : string) : string {
```

```
    return x + '!';
}
```

The use site of **f** might look like the following:

```
f('one'); // good
f(1); // bad
function g(x) {
    return f(x); // might be unsafe
}
```

Here Safe TypeScript is able to tell that: calling **f** with a string literal is safe, and no extra cast is needed; the second call is immediately rejected because number literal is clearly not a **string**; and for the third case where **x** is not explicitly given a type, Safe TypeScript compiler will insert runtime type cast and either allow it to proceed to the body **f** when **x** is indeed a string, or throw an error before even calling **f**.

Notice that by making use of type annotations, Safe TypeScript not only allows clearer code, but also able to spot some type errors ahead of execution: imagine the function application **f(1)**, when its written in plain JavaScript, we have to wait for the check in function body to throw an error, while annotating **x** with a type allows Safe TypeScript to spot the error even before execution.

2.1.2 Object-Oriented Programming in Safe TypeScript

JavaScript is a prototype-based language. This means runtime objects form a chain in which one object can be a prototype of other objects and to perform method invocation, method names are searched first in object itself and then along the chain, and the first one with matching method name is used for the invocation. Using well-known techniques allows JavaScript to simulate objected-oriented programming. Designed to be a superset of JavaScript, Safe TypeScript must accommodate such programming style as well.

TypeScript introduces the concept of interface to allow representing the structure of runtime objects.

```
interface Point { x: number; y: number }
```

This defines **Point** type to be an object that has two fields **x** and **y** whose types are all **number**s. For example, the invocation with object literal $f($**{x: 1, y: 0}**$)$ typechecks with $f : \mathit{Point} \Rightarrow \mathit{any}$. An interface can be implemented by classes:

```
class MovablePoint implements Point {
  constructor(public x: number, public y: number) {this.x = x; this.y = y;}
  public move(dx: number, dy: number} { this.x += dx; this.y += dy; }
}
```

While in TypeScript, all classes are viewed structurally, this does not match with JavaScript semantics. Consider the following function:

```
function mustBeTrue(x : MovablePoint) {
  return !x || x instanceof MovablePoint;
}
```

One might expect this function to always return **true**. However, a structural view will consider object literal $v =$ **{x: 0, y: 0, move(dx: number, dy:number) }** to match with **MovablePoint**, but **mustBeTrue(v)** will return **false** because JavaScript will expect v to have **MovablePoint** somewhere along its prototype chain.

To solve this problem, Safe TypeScript takes a different approach: it treats class types nominally but can be viewed structurally. Namely, **MovablePoint** is still a subtype of $t_m =$ **{x: number, y: number, move(dx: number, dy:number): any}** and **{x: number, y: number}**. But t_m is no longer a subtype of **MovablePoint**.

2.2 From Scheme to Typed Scheme

Scheme is a dynamically typed language. It is used for casual scripting as well as industrial applications. Like other dynamic typed languages, programmers are not attached to using any type discipline but rely on various kind of reasoning on their needs. This makes the language flexible but challenging to assign proper and precise types to terms.

Tobin-Hochstadt and Felleisen's work on Typed Scheme introduces the notion of occurrence typing, which extends Scheme with gradual typing.

2.2.1 Support Informal Reasoning

The following code shows a function definition as typical style of programming in this language:

```
;; a Complex is either
;; - a number
;; - (cons number number)
(define (creal x)
  (cond [(number? x) x]
        [else (car x)]))
```

The code above defines a function **creal**, which takes as argument a complex number: a real number is simply a value satisfying **number?**, while an imaginary number is a pair of **number**. Function **number?** distinguishes two different representation of numbers: in the the first branch of **cond**, **x** is treated like a number while in the second branch **x** is a pair and **car** is used to extract its real part.

One might have noticed that while input value **x** could be either a number or a pair of numbers, predicate **number?** distinguishes between them and in difference branches of the **cond** expression, **x** gets different types.

It is important that type system should be able to assign same variable with different types depending on the context where variable occurs, this is exactly the notion of occurrence typing.

The Typed Scheme version begins with definition of complex number:

```
;; a Complex is either
;; - a number
;; - (cons number number)
(define-type-alias Cplx (Union Number (cons Number Number)))
```

While **Cplx** is just an alias for the union type, it improves readability and allows other parts of the program to refer to it by name. The body of the function looks like:

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

Note that we are explicitly giving the return type **Number**, and the type system is capable of inferencing that both branch of **cond** expression returns a number so it will still typecheck.

Another different kind of reasoning is also being used among Scheme programmers: Suppose we have stored a list of various types of values in **xs** and the following code will compute the sum of all numbers from **xs**:

```
(foldl + 0 (filter number? xs))
```

Note that despite **xs** is a list that contains not just numbers, it is guaranteed that **(filter number? xs)** will return a list of numbers so **foldl** will work properly to produce the desired result.

The expression requires no modification at all to typecheck in Typed Scheme and it receives type system benefits: suppose **map** is mistakenly used instead of **filter** or **number?** is replaced by other predicates insufficient to test whether a value in question is indeed a value, the type system is able to recognize such errors and give warnings.

2.2.2 Refinement Type

Note that in Scheme, we use **number?** to distinguish numbers from other type of values. In general, given a boolean-valued unary function we can define the set of values that produces **true** when fed to this function. This allows Typed Scheme to support refinement type.

```
(: just-even (Number -> (Refinement even?)))
(define (just-even n)
  (if (even? n) n (error 'not-even)))
```

The code above uses boolean-valued function **even?** and its corresponding refinement type **Refinement even?**, so we can expect a value of such type to be not just numbers, but also only even ones.

More practical examples include use refinement types to distinguish raw input from validated ones:

```
(: sql-safe? (String -> Boolean))
(define (sql-safe? s) ...)
(declare-refinement sql-safe?)
```

In this example, user defines function **sql-safe?** to verify that a raw string contains no SQL injection or other contents of malicious attempt. Then a refinement type is declared using this very function. This makes available type **(Refinement sql-safe?)** to rest part of the program to avoid mistakenly using raw input rather than verified safe ones.

As we can see occurrence typing not just enables gradual typing, but also allows refinement type, which provides us a powerful tool to use user-defined functions to define custom types of better precision.

2.3 From Python 3 to Reticulated Python

Python 3 (Python for short) is another popular dynamic typed language. M. Vitousek, M. Kent and Baker's work on Reticulated Python (Reticulated for short) explored extending Python with gradual typing.

Python comes with annotation syntax. This syntax allows expressions to be optionally attached to function definitions and their arguments, which makes it suitable for type annotation.

Given proper definition of **Int**, we can write annotated function **distance** like below:

```
def distance(x: Int, y: Int)-> Int:
  return abs(x - y)
```

While annotation allows arbitrary expressions, Reticulated Python only uses types built from several type constructors.

2.3.1 Function Types

Python 3 has different ways of calling the same function. Suppose we want to use **distance** defined above, we can use positional arguments like **distance(3,4)**, or use keywords like **distance(y=4, x=3)** and the result should be the same.

To support keyword calls, **Named** constructor is used to give **distance** type **Function(Named(x: Int, y: Int), Int)**. To support traditional positional arguments, **Pos** is used instead. The type system also allows **Named** to be used as if it is constructed by **Pos** when their length and element types correspond. So functions can be called in both ways without problem. Additionally **Arb** is another constructor that allows functions of arbitrary parameters.

2.3.2 Class and Object Types

Python is an object-oriented programming language. Programmers define classes and create objects from them. Both classes and objects are runtime values in Python and Reticulated provides corresponding types for both.

Consider the following example:

```
@fields({'x': Int})
class 1DPoint:
  def __init__(self: 1DPoint):
    self.x = 0
  def move(self: 1DPoint, x: Int)->1DPoint:
    self.x += x
```

```
    return self

p = 1DPoint()
```

It defines a class **1DPoint** with one field **x** of **Int** type. And an object is created from it through the constructor and is bound to variable **p**.

structural typing to match for Python's duck typing

2.3.3 Dynamic Semantics

2.3.4 Other Features

2.4 C$^\sharp$ 4.0

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

syntactic improvement dynamic Object ("Expando Object")

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