Adding Gradual Types to Existing Languages

Javran Cheng

University of Maryland

Abstract. 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

1 Introduction

Type system defines sets of rules about programs, which can be checked by a machine in a systematic way. 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 allow auxiliary features like 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 section, we start off by discussing static and dynamic type systems, which have inspired many lines of research that combine both within one system. One instance among them is gradual typing, the main focus of this survey.

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.

Notation In the following sections we use $\tau_0, \tau_1, \ldots$ for type variables. **bool**, **num** and **str** are types of boolean values, numbers and strings respectively. Tuple types are notated as $(\tau_0, \tau_1, \ldots)$, and the type notation for functions that takes as input a value of type τ_0 and returns a value of τ_1 is $\tau_0 \to \tau_1$. We use $o : \tau$ to mean that a term o is assigned type τ. For example $f : (\mathbf{num}, \mathbf{num}) \to \mathbf{num}$ is a function that takes a tuple of two numbers as input and returns a number.

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 of typechecking is paid 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, extra maintenance on type annotations is required just to allow program execution. This might be undesirable in situations where a trial-and-error style of programming is preferable.

Having static known type information also helps in terms of performance. For example, In machine instructions, primitive arithmetic operations would only deal with operands of compatible types: addition can only add together two integers or two floating numbers. However, in many programming languages, addition is polymorphic and when it comes to the case of adding an integer and an floating number, the integer is implicitly converted into a floating number before performing the actual addition. For a statically typed system, type information can be used ahead of execution to figure out exactly whether it is necessary to insert conversions and where they are required. But a dynamic typed language might struggle because type information is only available at runtime. As a consequence, similar decisions about conversions have to be made at runtime, imposing performance penalty.

Besides extra effort of maintaining type annotations, the disadvantage of a statically type language often lies in the lack of runtime flexibility. If we want to write a program that deals with data whose structure is unknown at compile time, runtime inspection must be possible. While typical dynamically typed languages allow inspect of types or object properties, some extra work are required for a statically typed language to maintain and check runtime type information.

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.

Dynamic Type Systems Dynamic type systems do 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 improbable. And when it comes to prototyping, it is usually more important to allow a trial-and-error type of style of programming over concerns about robustness and correctness. In such a scenario, it is more convenient if the type system does not prevent an incomplete or ill-typed program from execution.

However, it is also easy to spot weaknesses of dynamic type systems. Despite that dynamically typed languages do not often provide a way of writing

type annotations, programmers usually have facts about behaviors of programs. These facts are often described in comments or through the naming of variables and functions without a systematic way of verifying them. As program develops, it is easy to make changes but leaving these descriptions untouched. This troubles future maintainers to locate the problem and rediscover new 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 interpret its value as a boolean at runtime.

Opportunity of optimization could also be obscured for a dynamically type language. Array cloning, as an example, can be implemented efficiently as memory copy instructions knowing the array in question contains only primitive values but no reference or pointers. But without prior knowledge like this, every element of the array has to be traversed and even recursively cloned, which is the usual case for dynamically typed languages without sophisticated mechanism of optimization.

1.2 Combining the benefits of the two

One might have noticed that static type systems and dynamic type systems are not incompatible in a fundamental level: the former does type checking ahead of execution while the latter maintain and inspect type information at runtime.

Indeed, researchers have been looking at different angles of the possibility of integrating these two type disciplines within one. Several lines of research are motivated, and in this section, we will discuss about these research works.

Design Goals Summarizing pros and cons of static and dynamic typed languages, we now have a picture of a resulting type system integrating the two, it should be able to:

- **Detect and rule out type 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, leaving no need of type-checking during execution.
- **Delay typechecking until it is required at runtime.** A dynamic type system is known for its permissiveness on what programs it can accept: ill-typed or partially written programs are still allowed to execute all the way to its termination or until reaching their incomplete or ill-typed parts. This 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, a statically typed language receives benefits from utilizing type information for other purposes like optimization or providing hints to development tools. For the resulting system, we should be able to have these benefits as well.

A Introduction to Gradual Typing Gradual typing [27] is one possible approach that meets our design goals. 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 fact, there are more instances of dynamically typed languages making extensions to support gradual typing than that of statically typed ones.

Dynamic Type and Type Consistency In order to capture the idea that type information might only be partially known ahead of execution, gradual type system introduces a special type, which is called "dynamic type" by convention, on top of a static type system and extends type judgment to allow a more permissive form of type checking in the presence of dynamic types. For the rest of this survey, we will use type **dyn** to indicate dynamic types when no language-specific notations are available.

To demonstrate necessary changes to type judgments, we use statically typed lambda calculus as a starting point.

All type rules are preserved except for function applications, which are replaced by two rules in gradually typed lambda calculus:

$$\text{(GAPP1)}\ \frac{\Gamma \vdash_G e_1 : \mathbf{dyn} \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash_G e_1 e_2 : \mathbf{dyn}}$$

$$\text{(GAPP2)}\ \frac{\Gamma \vdash_G e_1 : \tau \to \tau' \qquad \Gamma \vdash_G e_2 : \tau_2 \qquad \tau_2 \sim \tau}{\Gamma \vdash_G e_1 e_2 : \tau'}$$

While GAPP1 allows a value of dynamic type to be typed as if it is a function, GAPP2 is more interesting: it resembles function application of a static type system, but instead of demanding $\tau_2 = \tau$, it uses a new relation $\sim$, which is called type consistency relation. To deal with dynamic types, type consistency is introduced to accommodate type judgments for gradual typing. The following shows rules for type consistency relation.

$$\text{(CREFL)}\ \frac{}{\tau \sim \tau} \qquad \text{(CFUNC)}\ \frac{\sigma_1 \sim \tau_1 \qquad \sigma_2 \sim \tau_2}{\sigma_1 \to \sigma_2 \sim \tau_1 \to \tau_2} \qquad \text{(CUNR)}\ \frac{}{\tau \sim \mathbf{dyn}}$$

$$\text{(CUNL)}\ \frac{}{\mathbf{dyn} \sim \tau}$$

As we can see, if dynamic types are not present at all, type consistency is the same as type equality. Therefore we still can typecheck all static programs like

static type systems do. But we now can talk about types with partial information. Roughly speaking, two types are consistent when there are sensible values that belong to both of them. We demonstrate this by giving few examples:

- **bool** is consistent with type **dyn**. Despite that the latter is less precise, all boolean values can also be of type **dyn**.
- **num** $\to$ **dyn** is consistent with both **dyn** $\to$ **num** and **dyn** $\to$ **dyn**, because a function of type **num** $\to$ **num** can belong to all these types at the same time.
- **dyn** $\to$ **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.

Now that we have dynamic types, it is no longer a requirement to require every term in a program to have a type ahead of execution. For any term missing type information, we can simply assigned it with type **dyn**, and type rules will be permissive to accept it as long as type consistency is met.

On the other hand, a dynamically type program can now be statically typechecked by considering it to be a program without any type annotation. This is still beneficial, as some literal values and primitives will have their built-in types, an obvious misuse like taking the square root of a string value can now be detected ahead of execution instead of raising runtime exceptions.

As a side note, it is intended that type consistency is not a transitive relation: given that $\tau_0 \sim$ **dyn** and **dyn** $\sim \tau_1$, transitivity will imply $\tau_0 \sim \tau_1$, which would allow any pair of types to match and therefore render static typechecking useless.

Cast Insertion and Runtime Typechecking As we can see gradual type system attempts to contain both statically and dynamically typed programs. However, because of the presence of dynamic types, the type system cannot be sound by only have static typechecking. Imagine function f : **num** $\to$ **dyn** and variable v : **dyn**, while function application $f(v)$ can pass static typechecking, the runtime value of t might turn out to be a string and this violates f's input type **num**.

We can see what have gone wrong: while any number is also of type **dyn**, not all values of **dyn** type can be numbers. In general, when we are using a value of τ_0 as if it is a value of τ_1 and τ_0 is less precise than τ_1 (i.e. when **downcasting** a value), we need confirmation that this conversion from τ_0 to τ_1 is safe. This is exactly the idea behind **cast insertion** for gradual types.

Casts are functions that when given a value and a type, check whether the value agrees with the type, then either return the value if it is the case, or report an error or abort the program. Before program execution, casts are inserted to guard locations where downcasting happens. Cast insertion together with static type checking grant soundness for gradual type systems: a program either terminates successfully, raises type errors ahead of execution or during execution, or results in error not captured by the type system.

For the previous example, suppose we now have a cast c : **dyn** $\to$ **num**. By inserting the cast, function application $f(v)$ becomes $f(c(v))$. This still typechecks statically, but whenever we need to pass a runtime value to f, the value

is always checked by c to decide whether to pass the value to f or raise a type error. Now that in the body of f, the type system guarantees that its argument is a number.

Optional Type Annotations and "pay-as-you-go" It is important to realize that gradual type systems are not just about accepting both fully typed programs and those without any type annotation - it accepts programs that are partially typed. This gives programmers more control over what they need from typechecking (**"pay-as-you-go"**). As an example, one might wish to implement a function f that expects a string value as input. There are multiple choices:

- We can give f a type annotation: $f : \mathbf{str} \to \mathbf{dyn}$. By doing so, type system takes the responsibility of making sure that the input type is indeed a string. This is the only available choice in statically typed languages.
- We can do it with our program: right after entering the body of f, we write code to inspect type of the argument and react accordingly. A JavaScript program would do exactly this by branching on the result of evaluating `x && typeof x === "string"` (suppose `x` is the argument). This is often seen in a dynamically typed language, as there could be no other means of verification. But if the type system of a gradually typed one does not have sufficient type expressiveness to describe desired invariant, this still remains an option.
- For time or performance concerns or when programmer expects a frequent change of types during prototyping, we might wish to skip all static or runtime checks for a portion of a program, which can be achieved by leaving no type annotation. So programmers get just what they want from a type system. [1]

By making typechecking a choice, we have opened up some flexibilities: 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 types strengthens it with type error detection ahead of execution, machine-proved correctness, more opportunities of optimization and all other benefits that a statically typed language can enjoy.

Criteria for Gradual Typing There are certainly many possible ways of integrating static and dynamic typing and gradual typing just happens to be one of them. Siek, Vitousek, Cimini and Boyland's work [28] further introduces **gradual guarantee** property to formalize the idea behind gradual typing. Our intention is not about giving proofs about whether a specific type system is qualified as gradual typing, but to highlight the goals shared among this particular family of type systems.

In the original work, the following aspects are demonstrated:

[1] A gradual type system might still maintain runtime type information, which could impose some overhead. See "type erasure" for detail and some viable solutions.

- **Gradual typing includes both fully static and fully dynamic** A gradually typed language is equivalent to a superset of both a statically typed language and a dynamically typed one: When a program is fully type-annotated with no presence of dynamic types, it should behave the same as its statically typed counterpart; on the other hand, a program without any type annotation should behave the same as its dynamically typed counterpart. Executing such a program results in either a trapped error [7] or successful termination.
- **Gradual typing provides sound interoperability** As for partially typed programs, a gradual type system guarantees that, if the program typechecks statically, runtime type errors only raises from portions with dynamic types.
- **Gradual typing enables gradual evolution** Adding type annotations moves a program in the direction of static typing whereas removing type annotations moves in the direction of dynamic typing. Gradual evolution is the idea that programmers can freely add or remove type annotations without worrying about changing program behavior, which allows programs to evolve overtime: new features can be prototyped with no type annotation and later type can be gradually added to the corresponding portion. And old type annotations can be removed, giving more room for refactoring or experimenting new ideas without upsetting type system.

Discussion Among various research works that combines benefit of static and dynamic type checking, gradual typing extensions aim at making all existing programs written in the original one still valid with little difference in semantics. In addition, unlike other approaches, programmers have better control over should and should not be typechecked.

Despite that gradual type systems are well-studied research topics, extending existing languages to support them is far from trivial task. To name few differences, while theoretical type systems assume primitive types as simple as just numbers and booleans, a practical language have various kinds of them including integers, floating numbers, strings and arrays, which have not been addressed much. In addition, language-specific features, idioms and common practice needs to be carefully studied to allow existing code and language users to transit smoothly into the extended language. In this survey we will explore literatures that extends existing languages to support gradual typing. In section 2, we will focus on benefits brought by these extensions. Section 3 discusses and categorizes common and language-specific challenges researchers have to face and solutions to them. Related work will be covered in section 4, and we draw our conclusion in section 5.

2 Benefits of Gradual Typing

Gradual typing not just allows language users to write partially typed programs, it also make it possible to bring in features meant for static or dynamic typing. In fact, these extras are no less important than gradual typing itself, as it helps make the language more convenient to use and therefore motivates programmers to use gradual typing to their advantages.

In this section, we will explore some literatures from aspects of language users and see in detail about these benefits gradual typing have granted us.

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 even most experienced programmers.

Among tools and extensions to JavaScript that attempt to ease this pain, TypeScript is one closely related to gradual typing. Its syntax is a superset of JavaScript that supports optional type annotations [3]. A compiler typechecks TypeScript source code and compile it into plain JavaScript, making it ready to be executed out of box for JavaScript interpreters.

However, TypeScript is intentionally unsound: it is designed to typecheck programs only statically and removes all traces of type when emitting code in JavaScript. Therefore invariants about types cannot be enforced at runtime and benefits from the type system is limited.

Nonetheless, TypeScript still serves as a reasonable starting point for Safe TypeScript by Rastogi, Swamy, Fournet, Bierman & Vekris [22]. Safe TypeScript shares the same syntax with TypeScript, but features a sound gradual type system and efficient implementation[2].

In this part we will visit language features of Safe TypeScript and see how it improves JavaScript.

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 some lines of code are placed right after entering the body of a function to check its argument types, report errors if any before proceeding with actual implementation. This can be seen in the following code snippet:

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

[2] It is suggested that Safe TypeScript only cover a large fragment of JavaScript, but we assume that this coverage is sufficient to support all essential features of JavaScript.

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

As a toy example, the function accepts a string value, then returns another with exclamation sign concatenated to it. This does not look complicated, but imagine in real projects, there would be multiple arguments to a function and some of them have to go through this process of checking. Without machine-checkable and consistent way of enforcing these invariants, it is easy to make mistakes while maintaining code written in such a style.

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. Safe TypeScript then 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 unnecessary and therefore can be safely removed:

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

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

```
f('one'); // typechecks
f(1); // does not typecheck
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 entering the body of `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.

This benefit of detecting type errors earlier is not a specific one for Safe TypeScript. In general, adding a phase of static typechecking to a dynamically typed language might render some runtime checks no longer necessary. Once these checks are removed, programs might end up running more efficiently without compromising correctness.

Object-Oriented Programming in Safe TypeScript Although JavaScript does not support classes[3], its prototype-based model allows essential features of object-oriented programming to be simulated in JavaScript. Designed to be a superset of JavaScript, Safe TypeScript must accommodate such a programming style as well.

TypeScript introduces the concept of interface to allow describing the structure of runtime objects and Safe TypeScript follows this concept as well. The following example defines interface `Point`, which is a valid type for objects that has two fields `x` and `y` whose types are all `numbers`:

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

function f(o: Point) {
   // omitted
}
f({x: 1, y: 0}); // typechecks
f({x: 1, y: "a"}); // does not typecheck
f({x: 1, y: 0, z: 2}); // typechecks
f(1); // does not typecheck
```

As demonstrated by the example above, interface can only typecheck with objects that has all expected fields of expected types.

Classes are given special treatment[4] to cooperate better with model used by JavaScript. To be more precise, it treats class types nominally but allows it to be viewed structurally.

In addition to the `Point` interface defined above, suppose we have the following definition:

```
class CPoint implements Point {
  constructor(public x: number, public y: number) {
    this.x = x; this.y = y;
  }
```

[3] Classes were not standard features of JavaScript as of publication date of Safe TypeScript.

[4] While TypeScript treats classes structurally, this does not match with JavaScript semantics. See "Nominal Type, Structural Type and Subtyping" of section 3.3 for detail.

```
}

function f1(v : {x: number, y: number}) { return true; }
// f2 is equivalent to f1
function f2(v : Point) { return true; }
function f3(v : CPoint) { return true; }

var obj1 = {x: 1, y: 2}; var obj2 = new CPoint(1,2);
f1(obj1); // typechecks
f1(obj2); // typechecks
f3(obj1); // does not typecheck
f3(obj2); // typechecks
```

This example demonstrates few features of interest:

- **Structural view of classes** Since interfaces are structural, `f1` and `f2` are equivalent. In addition, objects created from object literals and instance created from classes can both typecheck against it. In the example above, despite that `obj1` and `obj2` are created in different ways, they both have expected fields of expected types and therefore function calls `f1(obj1)` and `f1(obj2)` do not raise any type error.
- **Interface declaration** In fact, `implements Point` is optional since interfaces are structural. It is nonetheless a good practice to write it out. Because this not only serves as a simple documentation, but it also allows Safe TypeScript to understand programmer's intention so type system can warn user when a declared interface is not actually satisfied.
- **Nominal view of classes** When using class names as types, Safe TypeScript treats them nominally, this is consistent with JavaScript's object model and `instanceof` operator, as the latter does not allow a structural match. Therefore while `f3(obj2)` typechecks, `f3(obj1)` does not.

2.2 From Scheme to Typed Scheme

Scheme is another dynamically typed language used for casual scripting as well as industrial applications. Instead of relying on any specific type discipline, programmers reason about their code in informal ways, which makes the language highly flexible. But for the same reason, it becomes a real challenge to design gradual type systems that can work well with scheme programs.

Tobin-Hochstadt and Felleisen's work on Typed Scheme [31] provides a novel and practical solution. In the same paper, the notion of occurrence typing is introduced, which cooperates with Scheme well to form the building block of a gradual type system for Scheme.

Support Informal Reasoning The following code[5] shows a function definition as typical style of programming in this language:

[5] Most of the examples in this section are adapted from [31].

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

It defines a function `creal`, which takes as argument a complex number. Rather than relying on type systems, the contract is informally written in the comment, which indicates that input value is supposed to be either a number or a pair of numbers representing real and imaginary part respectively for a complex number. Predicate[6] `number?` distinguishes number representation from the other one. So in first clause of `cond`, we know for sure that `x` is a number, whereas the second clause deals with remaining case, so `x` must be a pair with proper use of `car` to extract its real part. There are few key observations from this example:

- Difference occurrences of the same variable might have different set of possible values.
- This style of informal reasoning relies on result of applying predicates, and a predicate distinguishes one particular set of values from others. For example, `number?` can distinguish numbers from other values.
- By reasoning about the execution path taken to reach certain point in the program, predicates about a variable can be accumulated to further constraint the set of possible values.

These observations suggest that we could use predicates as some sort of types when extending the language with gradual typing: say all values that make predicate `number?` return true are `Numbers`, and all values that make predicate `cons?` return true are pairs, we should be able to encode this complex number representation using types. In fact, Typed Scheme version of it begins with such a type definition:

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

Note that `U` constructs a union type, it combines multiple types together so a value satisfying any of these types will also satisfy the resulting union type. As a separate note,`Cplx` is just an alias for the union type, it nonetheless improves readability and allows other parts of the program to refer to it by name.

The body of the function in Typed Scheme looks like:

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

[6] By Scheme convention, a predicate is a procedure that always return a boolean value.

Note that despite `x` is of type `Cplx` as it enters the function body, it is actually a number in first clause of `cond` and a pair of numbers in the second clause. This phenomenon is captured by the idea of **occurrence typing**: one variable can have different types for its different occurrences, and the type depends on execution path it takes to reach that location. The use of predicate `number?` allows type system to know that `x` in the first clause must be a number. By eliminating the number type from `Cplx`, the type system can further conclude that in the second clause `x` must be a pair of numbers.

With the help of local type inference [21], explicitly giving the return type `Number` is sufficient to let the type system know that every clause of `cond` returns a number so this part of the code can typecheck successfully.

Here is another example that shows a different kind of reasoning that Scheme programmers would rely on: suppose we have stored a list of values of various types 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, the invariant of `filter` guarantees that `(filter number? xs)` will always return a list of numbers so it will work properly to produce the desired result.

Typed Scheme is capable of encoding invariants about `filter` with its type system, and this expression requires no modification at all to typecheck in Typed Scheme. Having integrated the code into Typed Scheme, it can now enjoy some benefits meant for static typing: suppose `map` is mistakenly used instead of `filter` or `number?` is replaced by other predicates insufficient to conclude whether a value in question is indeed a number, the type system is able to recognize such errors and warnings will be given ahead of execution.

Refinement Type We have been using `number?` to distinguish numbers from other type of values. In general, given a predicate `f`, we can define a type whose values are the set of values that `f` hold true. This idea is captured in Typed Scheme with refinement types [14]. The function below shows a valid use of refinement types:

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

In this example, type `(Refinement even?)` is defined by predicate `even?`, which returns true if and only if input value is an even number. so we can expected this function to either a value satisfying `even?`, or throw an error.

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 implements predicate `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 so that when a programmer mistakenly uses string that contains raw input rather than verified strings, predicate `sql-safe?` will raise a type error instead of allowing program to proceed with potentially malicious raw input.

There are many other features introduced by Typed Scheme, but we can clearly see that extending Scheme with gradual typing brings benefits of static typing into the language.

2.3 From Python 3 to Reticulated Python

Python 3 (Python for short) is another dynamic typed language. M. Vitousek, M. Kent and Baker's work on Reticulated Python [32] (Reticulated for short) explored extending Python with gradual typing.

Reticulated makes use of Python's annotation syntax [33] to allow type annotation on function definitions and their arguments. 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 syntactically Python annotation allows arbitrary expressions, Reticulated Python only use few base types (`Int`, `String` and `Float`, to name a few) and a limited set of constructors (for example, `List`, `Dict`, `Object` and `Class` whose meanings are clear from their names), which nonetheless gives us sufficient expressiveness.

Function Types Besides conventional function invocation, Python supports named arguments, which allows a function to be invoked with explicit argument names. The example below shows 3 valid and calls to function `distance` and all these calls are equivalent with `x=10` and `y=20` upon entering function body.

```
distance(10,20)
distance(x=10, y=20)
distance(y=20, x=10)
```

Reticulated supports named arguments as well as conventional function invocations through 3 different ways of constructing function parameter types. For example, function `distance`, can be given type `Function(P, Int)` with 3 possible parameter types `P`:

- `P = Named(x: Int, y: Int))` This type indicates that the function has two arguments `x` and `y` in that order, which can either be called with or without use of named arguments. In other words, `Named` can be used in any places where `Pos` is expected.

- `P = Pos(Int, Int)` This type contains less information than `Named`, so it has limited support for calls with named arguments. Nonetheless this is a traditional function types used in many other languages.
- `P = Arb` This type allows functions of arbitrary parameters. This allows typing functions that are otherwise hard to type. But since it contains no useful information about a function, type system enforces no invariant on it.

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[7]:

```
@fields({'x': Int})
class Point1D:
  def __init__(self: Point1D):
    self.x = 0
  def move(self: Point1D, x: Int)->1DPoint:
    self.x += x
    return self

p = Point1D()
```

It defines a class `Point1D` with one field `x` of `Int` type. And an object is created from it through the constructor and is bound to variable `p`.

Reticulated derives class type from this definition:

```
Class(Point1D){
  move: Function(Named(self: Point1D, x: Int), Point1D)
}
```

And object types are similar:

```
Object(Point1D){
  move: Function(Named(x: Int), Point1D)
}
```

While object can access methods of its class, it is also possible to assign methods and fields to objects to change its behavior. Therefore Reticulated makes object type open with respect to consistency.

Other Features There are other features of Reticulate. In particular, it is not only a practical gradual type extension, but serves as an experiment of exploring different dynamic semantics which will have a large impact on the performance of

[7] This example is adapted from [32]

a gradually typed language. Please refer to **Objects and Dynamic Semantics** of section 3.3 for a detailed discussion.

In addition, despite Python syntax supports annotation at function definition, it does not provide a way of annotating local variables with types. Instead, Reticulated perform dataflow-based type inference on code blocks, which allows it to infer variable types for programmers.

2.4 From Smalltalk to Gradualtalk

Smalltalk is a language of highly dynamic nature: it supports live programming, which means programs with incomplete methods should be accepted, and programmers in Smalltalk relies on idioms that are tricky to type properly. Allende, Callau, Fabry, Tanter and Denker's work on Gradualtalk shows us a well combination of various features in order to extend Smalltalk into a gradual typed language.

Annotating Programs with Types In Smalltalk there are objects that receives and processes messages. The following example computes euclidean distance for points:

```
Point >> distanceTo: p
| dx dy |
dx := self x - p x.
dy := self y - p y.
^ (dx squared + dy squared) sqrt
```

Gradualtalk extends this syntax that allows type annotation on parameter, return value and local variables.

```
Point >> (Number) distanceTo: (Point) p
| dx dy |
dx := self x - p x
dy := self y - p y
^ (dx squared + dy squared) sqrt
```

Since local variable **dx** and **dy** does not explicit type-annotated, they receive **Dyn** as default type.

Blocks are basic features in Smalltalk, whose corresponding types are available in Gradualtalk in the form of normal function types.

```
Polygon >> (Number) perimeter: (Point Point -> Number) metricBlock
...
```

This definition expects **metricBlock** to be block that takes as argument two points, and returns a number.

Self Types One practice in Smalltalk is to return object itself for setters to allow chained calls:

```
Point >> (Point) y: (Number) aNumber
y := aNumber.
```

Notice that if we want to extend this class in future, its setter with be tagged with type **Point** and some type information will be lost in the way. To deal with this problem, Gradualtalk introduces self type:

```
Point >> (Self) y: (Number) aNumber
y := aNumber.
```

If any other class inherits from it, the self type will make sure to point to that class instead of sticking with **Point**, this allows type information to be preserved.

Union Types Gradualtalk supports union types, which is useful when source code contains different branches and each of them might return different types.

```
Boolean >> (a | b)
  ifTrue: (-> a) trueBlock
  ifFalse: (-> b) falseBlock
```

Note that **trueBlock** and **falseBlock** might return different types, allowing this method to have `a` or `b` will not be satisfactory, and simply using `Dyn` type returns in a lose of information. Gradualtalk solves this problem by introducing union types: `a | b` is a type by itself, which is compatible with both `a` and `b`.

Structural and Nominal Types A structural type describes the shape of an object.

```
RBParser >> bracketsOfNode: ({left (-> Integer) . right (->Integer)}) node
```

The method above defines the argument type of `node` to have 2 methods: `left` and `right` and both of them will return an integer when invoked.

A nominal type, on the other hand, is induced by classes (for example an instance of `String` will have `String` type). In addition, subtyping relations of them are formed from existing inheritance relation.

Gradual type managed to unify them in an interesting way.

```
RBParser >> bracketsOfNode: (RBNode {
left (-> Integer) .
right (->Integer)
}) node
```

Note that in addition to the structural type we have seen, `RBNode` is a nominal type, the type of `node` above limits the value to be not just an instance that understands certain methods, but also requires it to be an instance of `RBNode`.

Another interesting example is the combination of structural type and nominal type `Dyn`:

```
Canvas >> (Self) drawPoint: (Dyn {x (-> Integer) . y (->Integer)}) point
... point x. "safe call"
... point y. "safe call"
... point z. "not an error, considering point to be Dyn"
```

Besides calling `x` and `y`, a call to `z` does not raise an error ahead of execution, because `Dyn` can be casted to allow use of `z`.

Type Aliases Gradualtalk supports type aliases. This does not expand expressiveness of type itself, but it make reusing existing types convenient.

With type aliases comes the concepts of protocols, which are just type aliases for structural and nominal types.

2.5 From *C♯*3.0 to *C♯* 4.0

While we have seen many examples that a dynamically typed language gets gradual type extensions, Bierman, Meijer and Torgersen's work on C♯ 4.0 shows us a practical example of statically typed language extends towards gradual typing. This extension makes interoperation between statically and dynamically typed code more convenient and brings in features meant for dynamically typed languages.

Interaction with Dynamic Objects The following code snippet shows how C♯ 3.0 interact with JavaScript:

```
...
ScriptObject map = win.CreateInstance("VEMap", "myMap");
map.Invoke("LoadMap");
...
string latitude, longtitude;
...
var x = win.CreateInstance("VELatLong", latitude, longitude)
var pin = map.Invoke("AddPushpin", x);
pin.Invoke("SetTitle", name);
...
```

In order to interact with JavaScript, method calls are made using string-based interface, which is verbose, fragile to maintain.

With C♯ 4.0, the source code can be simplified to:

```
...
dynamic map = win.CreateInstance("VEMap", "myMap");
map.LoadMap();
...
string latitude, longtitude;
...
var x = win.CreateInstance("VELatLong", latitude, longitude)
var pin = map.AddPushpin(x);
pin.SetTitle(name);
...
```

Notice how types are changed to **dynamic** and methods are called as if there are regular objects. These improvements are more than just syntactic: gradual typing is implemented so that typechecking can be involved and this part of code can benefit from it.

Using Features Intended For Dynamic Languages The extension also makes available some features meant for dynamic languages directly in C♯, one example is the use of **ExpandoObject**

```
...
dynamic contact = new ExpandoObject();
contact.Name = "Erik"
contact.Address = new ExpandoObject();
contact.Address.State = "WA";
...
```

For a statically typed language, to maintain structured object, one either needs to declare a record type with matching structure, or use a dictionary in which one key and one value type should be declared. However, with the introduction of **ExpandoObject**, by simply assigning values to properties of it, they are brought into existence without boilerplate about type casts. This helps in terms of fast prototyping, in which programmers cares more about having a working implementation than maintain type precision.

3 The Way Towards Gradual Typing

In previous chapter we have explored the benefits gradual typing can bring in for existing languages. Now we takes a step further to discuss the way towards making these extensions.

Despite that we have gradual guarantee [28] and other theoretical works to serve as guidelines for gradual typing extensions, there is no canonical way of supporting gradual typing and design choices have to be made depending on various facts like original type discipline, common practice and performance.

Additionally, adding gradual types to an existing language is not just an effort of putting theories of gradual typing into practical use, but also introductions of new language features that take the advantage of gradual typing and help development or improve performance in these languages.

In this chapter, we will explore various issues regarding type system extensions, design choices, implementation, and other technical details. This allows us to obtain more understanding and insight about the way towards gradual typing for existing languages.

3.1 Extensions to Type Systems

For statically typed languages, gradual type system introduces dynamic types, which then allows presence of partial type information in types. For dynamically typed languages, gradual type extension enables type annotations or enriches its type expressiveness to allow programmers to enforce invariants in a more uniform and machine-checkable way. While type soundness is established by a cooperative work of static and dynamic typechecking, gradual typing itself opens up possibility of some other interesting extensions that makes the extended language more convenient to use and more attractive to programmers.

These extensions can be split into two categories: **Essential extensions** are those necessary for a complete gradual typing support; and **complementary extensions** are those that improves type expressiveness but a type system without them does not compromise completeness.

Essential Extensions We are looking at research works that falls into type disciplines of two different nature: JavaScript, Scheme, Python 3 and Smalltalk are all dynamically typed languages, from which much similarity shows up; $C\sharp$ 3.0, on the other hand, is statically typed, which have quite a different story about how the extension is implemented.

Extension from Dynamically Typed Languages The approach used by Safe TypeScript and Reticulated Python is to first perform static typechecking in the presence of dynamic types. If it is successful, programs written in extended languages are then translated into original languages. The translated program could contain runtime checks with some values instrumented, these checks and instruments are either implemented by inserting expressions or statements in place or

making calls to external libraries (which is called runtime in short) shipped with the extended languages.

This ensures a high compatibility with original languages: source code written in extended language can be compiled and then used as if it is written in the original language, and thanks to the fact that all existing dynamically typed languages we are discussing have decent supports for modern package managers, including runtime support is as simple as making a dependency declaration to package metadata. Besides, by compiling to the original languages, we can to some extent be confident that the language implementations are independent of specific implementations of the original languages, albeit some language extensions do make assumption about implementations of their original languages.

Gradualtalk is an extension to Pharo Smalltalk. It consists of 3 parts: the core allows representing types in Smalltalk, the typechecker is responsible of performing static typechecks and finally a type dictionary for storing type information.

Typed Scheme takes a similar approach: the implementation is a macro (a source-to-source transformer) that does typechecking and expands source code. This phase either results in static type error being reported, or source code translated to PLT scheme with some runtime casts inserted in the form of contracts.

Extension from A Static Type System The work on $C^\sharp$ 4.0 shows us how statically typed language can be extended to support gradual typing. Similar to Siek's original approach, the compiler assign types to terms using built-in knowledge, explicit type annotations and type inference, then programs are translated into another core language that consists of less constructions and dynamic types are either turned into casts or removed in this process to produce final executables. But unlike original work on functional languages, $C^\sharp$ 4.0 achieves transitivity on subtyping: the problem occurs when dynamic types are involved, allowing subtyping relation to be established by using the dynamic type as middle ground. This problem is solved by not allowing dynamic types to be converted to any other types.

(TODO: achieve transitivity)

Complementary Extensions While essential extensions establish the completeness of gradual type systems, complementary extensions are not exactly requirements. After all, programmers can always choose not to use explicitly typed terms and leave types of them inferred or defaulted to dynamic types. This is however unsatisfactory, as using dynamic types often result in a lost of type information in the process. Therefore extra features of type systems are usually introduced to improve type expressiveness and in general make language more convenient to use.

Union Type Practical languages utilize control flows to be able to determine the sequence of operations to perform base on runtime values. This allows same block of code to return values of different types and simply assigned returned value to a most general type results in information loss.

Imagine the following code in Typed Scheme:

```
(lambda (x : number)
  (if (> x 0) x #f))
```

Depending on the test `(> x 0)`, we either get a number or the boolean value `#t` as result, which can be expressed through union type `(∪ Number Boolean)`.

Union type is a simple extension employed by Typed Scheme and Gradualtalk to solve this problem. By notation, an union type $U = \tau_0 \cup \tau_1 \cup \ldots$ is a superset of all τ_xs involved. For any value v, $v : U$ typechecks as long as $v : \tau_x$ does. By performing flow analysis on a block of code, we can find a list of types of return values to then construct a union type to allow preserving more type information.

Type for Objects Object allows related values and codes to be organized together and is an important concept in various languages. Many language rely heavily on object-oriented programming, which demands proper type system support for them. Siek and Taha pioneered gradual typing for objects [26], which treats objects structurally as a pack of methods with labels. However, when applying to existing languages, this approach results in various issues that goes beyond type system. We will have a separated section regarding these issues.

Occurrence Typing Supporting gradual typing requires having matching types for values and functions in the languages. Most extensions we have discussed have concepts of classes and objects, they are typed either structurally or nominally. However, instead of using classes or objects like other languages do, typical Scheme programs prefers using composed data structures as simple as just pairs, vectors and they are distinguished through testing the shape of the value or symbol comparison. To solve this potential mismatch between values and types, occurrence typing is introduced. The idea is that the difference occurrences of the same variable would have different types depending on the context where it occurs. Types are defined by specifying some boolean-returning unary functions as type discriminators. When such a function `p?` in question returns non-falsy value for a value `v`, the type system will consider `v` to have value `p`. This way, a `if`-expression that depends of value of `(p? v)`, will have its two branches with different types of `v`, one with type `p` and another with non-`p` type.

Despite this extension is shown only in Typed Scheme, we believe it can be applied to other languages as well: the latest version of TypeScript has implemented **type guard**, which uses this exact technique to improves its type system.

Parametric Polymorphism The idea behind parametric polymorphism (a.k.a generic programming in some languages) is that some functions and structures can work uniformly regardless of the value it is operating on. Besides its presence in $C^\sharp$ 3.0, Safe TypeScript, Typed Scheme, Reticulated Python and Gradualtalk all include this feature to further extend their type expressiveness.

The following illustrates this feature in Safe TypeScript:

```
interface Pair<A,B> { fst: A; snd: B }
function pair<A,B>(a: A, b: B}: Pair<A,B> {
  return {fst: a, snd: b};
}
```

In this example, `Pair` is a structure that contains two pieces of data and `pair` a function as its constructor. The two pieces of data can be retrieved by accessing `fst` and `snd` attributes without specific knowledge of actual data types. Type variables are `A` and `B`, which allows getting data types back once `Pair<A,B>` is known.

This feature is made possible in Safe TypeScript by extending typing context with type variables and allowing type abstraction at appropriate places (e.g. in interfaces or function declarations as shown above). In Gradualtalk, same feature is implemented based on Ina and Igarashi's work [17].

3.2 Cast Insertion

Cast insertion provides the mechanism for runtime typechecking. Casts are functions that takes a single value, checks whether the value is of expected type then either throws type error or proceed the program with the value just checked. Static typechecking for a gradual type system has an extra purpose of inserting casts: when dynamic types are encountered during static typechecking, the type information at hand is insufficient to guarantee type safety. Therefore casts are inserted in appropriate places so they can be checked at runtime.

Taking Safe TypeScript as an example:

```
function toOrigin(q: {x: number, y: number}) {q.x = 0; q.y = 0;}
function toOrigin3d(p: {x: number, y: number, z: number}){
  toOrigin(p); p.z = 0;
}
toOrigin3d({x: 17, y: 0, z: 42})
```

The code above is translated into JavaScript:

```
function toOrigin(q) { q.x = 0 ; q.y = 0; }
function toOrigin3d(p) {
  toOrigin(RT.shallowTag(p, {"z": RT.num})); p.z = 0;
}
toOrigin3d({x: 17, y: 0, z: 42})
```

in which `RT` is Safe TypeScript runtime. Notice how argument of `toOrigin` requires an object that contains both `x` and `y` attributes while in the body of `toOrigin3d`, it only tags object `p`, which is known to have `x`, `y` and `z`, with extra attribute `z`. Thanks to static typechecking, we have reduce the amount of runtime checks by skipping checking types already known statically.

While static typechecking is performed during compilation, dynamic typechecking is embedded in code and therefore impose a runtime overhead over

original programs. To make the performance of partially typed programs close with its dynamically typed counterparts, It is essential to reduce the amount of dynamic checks. For value of primitive types, we can do no better than simple cast insertions, but there are still space when it comes to structured data and functions, researchers have come up with different strategies to balance between type safety, efficiency or design simplicity. We will explore thees strategies in next section.

3.3 Object-Oriented Programming

Object-oriented programming is a popular style of programming. All languages we have discussed so far supports it to some extent. Some have developed intensive support for it, among which some even have the language itself designed to embrace its philosophy. To have a sound and efficient support for it becomes center of topic when it comes to runtime performance of supporting gradual typing.

Unified Concepts Despite that each language have a slightly different model of objects, they do have concepts in common. In this section, we give definition to some important concepts in object-oriented programmming to allow discussing them in an unified way.

An **object** o is a single value that contains **fields** (somethings also called **properties** or **attributes**) and **methods**. Unique labels ls are assigned to all fields and methods. By using notation $o.l$, we can **access** (i.e. **read** from or **write** to) fields and methods of object o. Fields are values, and methods are functions within whose body gain access to a special variable (called `this` or `self` by convention) which can be used to refer to the object itself.

A special kind of singleton objects are called **classes**. For a class c, we can create (a.k.a. **instantiate**) objects by calling constructor. The created object is considered an **instance** of class c.

Classes can form a partial order of subclass relations: if a class c_0 **extends** from another class c_1, we say c_0 is a **subclass** of c_1 and c_1 a **superclass** of c_0. Methods can be shared through the mechanism of class-instance and inheritance relation: when we try to access $o.l$, l is looked up in the order of instance itself (some languages disallow objects to have their own methods therefore skipping this step), its class, and superclass of its class, all the way through this chain of inheritance until hitting the top object. The lookup resolves to the first successful one, but if no method matching l is found, program throws an error to inform about an unknown method being invoked.

Sometimes the concept of **interfaces** (or **protocols**) is used. It is a set of field or method labels with associating any values or functions to them. Any object that can resolve all labels list by an interface is considered to **implement** that interface.

Nominal Type, Structural Type and Subtyping There are two different approaches of typing objects: **nominal types** are derived from class definitions, these types usually have a matching name with its corresponding class. Nominal types follow the traditional sense of inheritance: one must explicitly extend from another class to make it a subclass of the other (with a commonly assumed exception that topmost class is always a superclass of other classes). **Structural types**, on the other hand, views an object structurally. Inheritance, or rather subtyping, in this case is closer to an interface: if one object can resolve all labels of another object, it is considered a subtype of it. This approach is sometimes known as duck typing.

Due to the highly dynamic nature of some languages, these two views can coexist and impose differences and difficulties and researchers need clean way to work around this issue.

To illustrate the problem, suppose we have the following definitions in Python (assume that all methods will return values of the same type):

```
class A:
  def foo(self):
    ...

class B(A):
  def bar(self):

class C:
  def foo(self):
    ...
  def bar(self):
    ...
```

While a nominal type allows a variable $v : A$ to be assigned an instance of B, assigning an instance of C to v is forbidden by the type system: despite having all methods that A requires, it is not explicitly extending A to form a proper relation of inheritance.

Structural type is more of a relaxed relation in this case: A is simply $\{foo : \tau\}$ (for whatever appropriate type τ), B and C are both equivalent to $\{foo : \tau, bar : \tau\}$. Therefore structurally, B and C are both subtypes of A despite no being explicitly declared.

Another interesting example comes from TypeScript. Suppose we have the following definition[8]:

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

class MovablePoint implements Point {
  constructor(public x: number, public y: number) {
    this.x = x; this.y = y;
```

[8] This example is taken and adapted from [22].

```
  }
  public move(dx: number, dy: number} {
    this.x += dx; this.y += dy;
  }
}

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

// typechecks, but returns false
mustBeTrue({
  x: 1, y: 2,
  move: function(dx,dy) {
    ...
  }
});
```

One might expect calls to function `mustBeTrue` to always return `true`, but in TypeScript this is not the case. As TypeScript treats classes structurally, any object that has field `x`, `y` and `move` of matching types can typecheck with `MovablePoint` without being an actual instance of it. So `instanceof` operator, which only respects the prototype chain, is inconsistent with TypeScript's type system.

To solve this problem, Safe TypeScript treats class types nominally but allows it to be viewed structurally. Taking the same example but assume that we are using Safe TypeScript, instances of `MovablePoint` can still be used in places where type {`x: number, y: number, move(dx: number, dy:number): any`} or {`x: number, y: number`} is expected. But in order to typecheck a value against `MovablePoint`, the value now has to be an instance of `MovablePoint`.

Among other languages we studied, Typed Scheme extends its structure system and uses nominal types [9]; Reticulated Python chooses the structural approach - despite class inheritance being allowed, duck typing is generally considered idiomatic approach in Python. On the other hand, Gradualtalk takes an approach similar to that of Safe TypeScript's: it implements a unified syntax to allow an object to be typed nominally and structurally at the same time.

Objects and Dynamic Semantics While most of extensions we have studied so far agrees on how casts are inserted around primitive values as originally suggested in [27], objects and dynamic semantics are always center of discussion among languages that relies heavily on it.

For this part, our discussion follows the work on Reticulated Python, as Reticulated is not just an implementation of gradually typed Python, but also a

[9] Thanks to the expressiveness of refinement type, it is still possible to test an object structurally.

testbed of 3 different dynamic semantics, which covers most of the cases. Along the way, approaches from other literature of similar nature will be discuss and added as well.

There are 3 different approaches implemented by Reticulated, known as **guarded**, **transient** and **monotonic**.

For starter, an example in Reticulated is given:

```
class Foo:
  bar = 42
def g(x):
  x.bar = 'hello'
def f(x:Object({bar: Int})->Int:
  g(x)
  return x.bar
f(Foo())
```

Note that function `g` mutates the type of `x.bar` therefore its callee `f` no longer have a `x` of proper type to return `Int`. When different dynamic semantics are applied, the result varies as well and in following parts we will visit these approaches in order.

The Guarded Dynamic Semantics This approach wraps actual objects in a proxy which builds itself up using sequences of casts. This keeps proxies to always be one step alway from the actual object rather than building a chain of proxies that compromises efficiency over time. Method invocations and field accesses are relayed to the actual object, field writes and return values of method invocation are checked at the boundaries of statically and dynamically typed code.

After translation, the relevant functions have casts inserted as following:

```
def g(x):
  cast(x, Dyn, Object({bar: Dyn})).bar = 'hello'
def f(x:Object({bar: Int}))->Int:
  g(cast(x, Object({bar: Int}), Dyn))
  return x.bar
f(cast(Foo(), Object({bar:Dyn}), Object({bar:Int})))
```

When the control reach body of function `g`, `g` will have the most precise type `Object(bar:Int)` inferred from the sequence of operations. And the write to field `bar` will try casting a string into `Int`, which results in a failure.

Note that this approach have the problem that object identity is not preserved:

```
x is cast(x, Dyn, Object({bar:Int}))
```

As `cast` function wrap `x` in a proxy, it is no longer considered the same object as `x` therefore returns `False`. However instance tests will still work, as proxies are considered a subclass. And for a similar reason, type test on such a value might fail because from outside it is a `Proxy` rather than an instance of the relevant class.

The Transient Dynamic Semantics Instead of using proxies, the transient approach inserts casts at use sites or at sites where the value becomes relevant. Under this approach, the same Reticulated function is translated as following:

```
def g(x):
  cast(x, Dyn, Object({bar: Dyn})).bar = 'hello'
def f(x:Object({bar: Int}))->Int:
  check(x, Object({bar: Int}))
  g(cast(x, Object({bar: Int}), Dyn))
  return check(x.bar, Int)
f(cast(Foo(), Object({bar:Dyn}), Object({bar:Int})))
```

There are two differences: first, despite having the same name, `cast` is a different function that just checks whether cast on object is allowed and then returns it instead of wrapping it in a proxy; second, calls to `check` are inserted around the body of `f`. In this dynamic semantics, it is the final call to `check` inside the body of `f` that detects type error and throws, as `x.bar` is mutated and its type no longer matches the return type of `f`.

The Monotonic Dynamic Semantics The monotonic approach relies on a slightly strong assumption than the other two approaches: if any update happens to object fields, it will not change its type. Instead of using a proxy, the object keep type information of its own. As casts are execute on objects overtime, its type information gets locked down to always become more precise.

The translation is the same as that of guarded dynamic semantics, but with `cast` being a different function that applies monotonic approach.

```
def g(x):
  cast(x, Dyn, Object({bar: Dyn})).bar = 'hello'
def f(x:Object({bar: Int}))->Int:
  g(cast(x, Object({bar: Int}), Dyn))
  return x.bar
f(cast(Foo(), Object({bar:Dyn}), Object({bar:Int})))
```

Right before call to `f`, object newly created from **Foo()** is locked down to have a type at least as precise as `Object(bar:Int)`, this locks down type of `x.bar` therefore raises an error when the body of `g` tries to write a mismatching type to it.

A similar approach is used by Safe TypeScript, in which runtime type information (RTTI) is maintained on objects themselves. And as program proceeds, RTTI decreases with respect to the subtyping relation, which avoids some unnecessary runtime checks to be performed over and over again. Additionally, another improvement done by Safe TypeScript is the strategy of shallowly tagging objects in RTTI, this avoids tagging objects that are never touched at the cost of having to propagate RTTI when needed, in practice researcher finds it to be a good tradeoff.

Type Dictionary This approach is used by Gradualtalk, but can also be adapted to other languages of similar nature: in Smalltalk, certain classes are not allowed to be modified because the virtual machine relies on them to implement fundamental behaviors. Therefore language implementor maintains a type dictionary that keeps type information for objects instead of tagging directly on object themselves. This can be considered to be strategy for prototyping gradual typing extension for a language when efficiency is not a urgent concern as work on Gradualtalk is not (yet) focusing on performance either, and the tagging strategy is not being stated in detail to give more insight.

RTTI and Differential Subtyping Runtime Type Information (RTTI) is one part of the mechanism for Safe TypeScript to implement its dynamic typechecking. Much like the monotonic approach found in Reticulated Python, with casts inserted in appropriate places and objects instrumented, runtime casts can be performed.

One key contribution of Safe TypeScript is its use of differential subtyping: it is an extension that not only tells subtyping relations, but also allows splitting a structural type into slices of interest. The observation is that, during static typechecking, when subtyping relation is used, there is a potential loss in precision. We can extend subtyping relation to compute the exact set of type information, and encode in the form of RTTI instead of keeping types of all fields around therefore improve performance.

Type Inference Programs with type inference and those with gradual typing have some similarity in the sense that both allow programmers to write less type annotation themselves. Many languages we have mentioned so far supports type inference to some extent. TypeScript implements local type inference within method bodies, as Safe TypeScript is built on top of it, it inherits this features as well. Reticulated Python implements local type inference by using dataflow analysis to compute possible types for each variables and then join these types to determine what type should be assigned to each variables. A local type inference is also employed in Typed Scheme.

Type Erasure and Optional Typing Sometimes it is helpful to ensure that no runtime overhead is imposed on some values: for a tag-based implementation of gradual typing, an large array of objects will suffer from unnecessary taggings. In such cases, some languages provide a way to ensure that no tag is attached to these objects. Note that this is different from using dynamic types: when using dynamic types, as we have seen in dynamic semantics, some implementation of gradual typing allows structure of types to become more precise during execution, an erased type ensures that no runtime tagging will happen therefore typing is avoided. Of course, without tagging it is impossible to enforce some invariants at runtime, which remains as a choice that programmers can do.

3.4 Challenges

This section shows challenges and open questions.

Common Challenges

Synchronization One problem of making extensions to an existing language lie in the fact that the extended one needs extra maintenance efforts to keep being a superset of the original language. If the language being extended is actively changing or extending its syntax, our gradual typed version will have trouble keeping it up or even run into the problem of conflicting syntax. This is particular true in case of JavaScript: nowadays language proposals are still being reviewed and accepted, which requires Safe TypeScript to bring in these new features as well.

Libraries Maintenance Despite that gradual typed systems claim to provide freedom of making choices between statically or dynamically typed. For this feature to be useful at all, type signatures must be provided and kept up-to-date, which might require maintenance efforts.

Eval function Languages of highly dynamic nature would provide a `eval` function, which when given a string or a abstract syntax tree, evaluates it as if it is one part of the program and returns the result. These functions are difficult to give a type more precise than that of a function. However, the use of `eval` function is generally considered bad practices and might impose threats to security because this might allow arbitrary code to be executed.

JavaScript Originally designed for casual scripting and extended over a long period of time, JavaScript admittedly contains many design flaws that programmers must be aware of and work around them. Any language extension claimed to fully support JavaScript inherits these problems as well. Some goes to the path of writing another language that compiles to JavaScript, therefore completely avoiding dealing with JavaScript directly for programmers. Other like Safe TypeScript, only support a reasonable subset of JavaScript - some use of language features are considered deprecated or bad practice, so not fully supporting all programs that JavaScript accepts does not raise much concern in practice.

Scheme Typical Scheme practice involve heavy use of macros. Despite Typed Scheme is able to handle most of the macros by expanding them before typechecking, some macro systems of sufficient complication require some understanding of their own invariants, which remains future work.

Another challenge that scheme faces is that the type expressiveness of Typed Scheme not sufficient for some highly flexible functions: some functions can be of indefinite arity, for example `(map f xs)`, `(map f xs ys)` and `(map f xs ys`

`zs)` can all be valid given `f` of expected arity; a similar situation is with `apply` function, which accepts a function and a list of arguments. Typed Scheme has special cases to deal with these cases, but it is more desirable if these types can be expressed using the type language.

Python 3.0 One problem with Python is that Python does its own runtime checks which cannot be prevented even if it can be proved in Reticulated that some particular checks are unnecessary.

Another potential problem with Python is that the annotation syntax is allowed in the original language albeit not being used. Therefore it is possible that some other library might want to use annotation syntax for other purposes, which is unlikely to be compatible with Reticulated.

Smalltalk Smalltalk features a live system: developers might modify code, run it and debug it using the same execution environment, this raises several challenges in the development of Gradualtalk:

- **Granularity of the compiling process** Smalltalk's compile unit is smaller than that of many other languages. Instead of doing per-class compilation, Smalltalk compiles in the unit of method to allow a live system. This causes troubles when defining methods with circular dependencies. Gradualtalk solves this problem by decoupling typechecking and compiling process, so a type error does not prevent compilation of a incomplete program.
- **Work done by the typechecker can be obsolete** When new methods are added or type signature for an existing type changes, typecheckers can produce obsolete results. This is solved by using a dependency tracking mechanism and with the use of ghost entities.
- **Type errors in critical code** Sometimes type errors can be critical: for example, if default error handling function itself contains some kind of error, this would result in an infinite loop. This is solved by allowing cast insertion to be disabled.

4 Related Work

4.1 Gradual Typing

Siek and Taha's work on Gradually Typed Lambda Calculus [27] establishes the theoretical foundation for gradual typing. It is suggested in the same paper that Cecil [9], Boo [11], extensions to Visual Basic.NET and $C^\sharp$ proposed by Meijer and Drayton [18], extensions to Java proposed by Gray, Findler and Flatt [15] and the Bigloo [6] [24] dialect of Scheme [1] could be formalized using this foundation. By the same authors, the idea is pushed further for object-based languages [26]. As the concept of gradual typing gains its popularity and being used vaguely to refer to any work regarding integrating of static and dynamic type system, Siek, Vitousek, Cimini and Boyland present gradual guarantee [28] to reiterate the intention behind gradual typing and serves a more clear guideline for languages claimed to be gradually typed.

4.2 Soft Typing

Cartwright and Fagan's Soft Typing [8], is a type system intended to free programers from writing type annotations. Type inference is used instead to assign appropriate types to terms. While this design improves performance of dynamically typed languages as the inferred information is used to removing some runtime dispatching, it does not give programmers control over types.

4.3 Partial Type Systems

The idea of allowing partial types in a type system is not new. Anderson and Drossopoulou formalized BabyJ [2], which has nominal type system that allows permissive types. This concept is similar to dynamic types in a gradual type system. But unlike gradual typing, the type system is nominal and does not takes into account function types. Both Thatte's Quasi-Static Typing [30] relies on subtyping, which creates a fundamental problem that prevents type system from catching all type errors even when programs are fully type annotated. Another type system that relies on subtyping is developed by Riely and Hennessy [23] on a partial type system for Dπ, type **Ibad** is used for untyped parts. However, by treating **Ibad** as the bottom type, it could allow **Ibad** to be implicitly coercible to arbitrary types.

4.4 Type Annotation for Performance

Some languages includes optional type annotations, but the annotation is used for improving performance rather than typechecking. Common LISP [29] and Dylan [12] [25] fall into this category.

4.5 Type Inference

Type inference [10] [16] [19] removes some burden of writing type annotations from programmers by using available type information to reconstruct types for terms. It is syntactically similar to gradual typing as both approaches allow programmer to omit type annotations. While type inference simply rejects a program when it does not have sufficient information to fill out all missing types, a gradual type system allows partial type to present in a program and insert casts to enforce invariants at runtime instead. Local type inference [21] sometimes are used in a gradual type system to either free programmers from having to write out all type annotations or to accommodate the fact that some languages do not have a way of attaching variables to an explicit type.

4.6 Optional Type System

In an optional type system [4], type annotations are optional and have no impact on the semantics of the language. Example includes Strongtalk [5]. While static typechecking is still present, no runtime checks are inserted so the language does not receive full benefits of a type system. Nonetheless, sometimes it might be desirable to ensure that certain part of the program does not have runtime checks inserted by the type system. Some gradual type systems like Safe TypeScript [22] does allow writing types in a specific way to effectively switch some part of the program to use an optional type system.

4.7 Type System Integrations

There are other examples of combining different type systems and taking advantage of both. Ou, Tan, Mandelbaum and Walker [20] define a language that combines standard static typing with dependent typing. Flanagan's Hybrid Type Checking [13] combines standard static typing with refinement types. Automated theorem proving technique is employed to try to satisfy predicates and runtime checks are inserted when theorem prover yields no conclusive results.

5 Conclusion

Gradual typing is an approach that brings benefits of static and dynamic type system together in a sound and efficient manner. From aspect of programmers, gradual type extensions makes it possible for dynamically typed languages to express program invariants in a unified and machine-checkable way, and for statically typed languages, programs are allowed to be partially typed to obtain some more flexibility for situations like refactoring or fast-prototyping. Additionally, the ability of evolving the program towards either fully static or fully dynamic programs is also open and intended to be smooth.

In this survey, we motivated gradual typing, visited designs and implementations of Safe TypeScript, Typed Scheme, Reticulated Python, Gradualtalk and $C^\sharp$. The way towards gradual typing for an existing language is not unique, because there are different language goals, programming idiom and established user community to be taken into account. However, to create a convincing gradual type extension, the way often involves not just making type system extensions and choosing dynamic semantics wisely, but also complementary features that enriches type expressiveness or reveals more opportunities for optimization or development tool supports.

There are still open questions present in making and maintaining these language extensions and the majority is about performance and keeping up with original languages. But this is understandable as many gradual type extensions we have discussed so far serve as proof of concept or prioritize correctness over performance. There are considerable amount of room for improvement and we remain confident that gradual typing extensions are practical and their benefit are significant for existing languages in long run.

Acknowledgment

References

1. Harold Abelson, R. Kent Dybvig, Christopher T. Haynes, Guillermo Juan Rozas, NI Adams, Daniel P. Friedman, E Kohlbecker, GL Steele, David H Bartley, Robert Halstead, et al. Revised 5 report on the algorithmic language scheme. *Higher-order and symbolic computation*, 11(1):7–105, 1998.
2. Christopher Anderson and Sophia Drossopoulou. Babyj: From object based to class based programming via types. *Electronic Notes in Theoretical Computer Science*, 82(8):53–81, 2003.
3. Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.
4. Gilad Bracha. Pluggable type systems. In *OOPSLA workshop on revival of dynamic languages*, volume 4, 2004.
5. Gilad Bracha and David Griswold. Strongtalk: Typechecking smalltalk in a production environment. In *ACM SIGPLAN Notices*, volume 28, pages 215–230. ACM, 1993.
6. Yannis Bres, Bernard Paul Serpette, and Manuel Serrano. Compiling scheme programs to .net common intermediate language. *.NET Technologies 2004*, page 25, 2004.
7. Luca Cardelli. In *The Computer Science and Engineering Handbook*, chapter Type Systems. CRC Press, 1997.
8. Robert Cartwright and Mike Fagan. Soft typing. In *ACM SIGPLAN Notices*, volume 26, pages 278–292. ACM, 1991.
9. Craig Chambers. the cecil group. the cecil language: specification and rationale, version 3.2. *Te. report, Department of Computer Science and Engineering, University of Washington, Box*, 352350:98195–2350, 2004.
10. Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.
11. Rodrigo Barreto de Oliveira. The boo programming language, 2005.
12. Neal Feinberg, Sonya E Keene, Robert O Mathews, and P Tucker Withington. *Dylan programming: an object-oriented and dynamic language*. Addison Wesley Longman Publishing Co., Inc., 1996.
13. Cormac Flanagan. Hybrid type checking. In *ACM Sigplan Notices*, volume 41, pages 245–256. ACM, 2006.
14. Tim Freeman and Frank Pfenning. *Refinement types for ML*, volume 26. ACM, 1991.
15. Kathryn E Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained interoperability through mirrors and contracts. In *ACM SIGPLAN Notices*, volume 40, pages 231–245. ACM, 2005.
16. Roger Hindley. The principle type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.
17. Lintaro Ina and Atsushi Igarashi. Gradual typing for generics. In *ACM SIGPLAN Notices*, volume 46, pages 609–624. ACM, 2011.
18. Erik Meijer and Peter Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. OOPSLA, 2004.
19. Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

20. Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In *Exploring new frontiers of theoretical informatics*, pages 437–450. Springer, 2004.
21. Benjamin C Pierce and David N Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000.
22. Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for typescript. *ACM SIGPLAN Notices*, 50(1):167–180, 2015.
23. James Riely and Matthew Hennessy. Trust and partial typing in open systems of mobile agents. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–104. ACM, 1999.
24. Manuel Serrano. Bigloo: a practical scheme compiler. *Inria-Rocquencourt, April*, 2002.
25. Andrew Shalit. *The Dylan reference manual: the definitive guide to the new object-oriented dynamic language*. Addison Wesley Longman Publishing Co., Inc., 1996.
26. Jeremy Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming*, pages 2–27. Springer, 2007.
27. Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.
28. Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.
29. Guy L Steele Jr. An overview of common lisp. In *Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 98–107. ACM, 1982.
30. Satish Thatte. Quasi-static typing. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 367–381. ACM, 1989.
31. Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. *ACM SIGPLAN Notices*, 43(1):395–406, 2008.
32. Michael M Vitousek, Andrew M Kent, Jeremy G Siek, and Jim Baker. Design and evaluation of gradual typing for python. In *ACM SIGPLAN Notices*, volume 50, pages 45–56. ACM, 2014.
33. Collin Winter. Function annotations. https://www.python.org/dev/peps/pep-3107/, 2006.