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. 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 the benefits from the two, which in return gives birth to many possible solutions. One particular example among them is 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 $\tau_0, \tau_1, \ldots$ 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 $\tau_0 \rightarrow \tau_1$. We use $o : \tau$ to mean that a term o is assigned type τ. For example $f : (\mathbf{num}, \mathbf{num}) \rightarrow \mathbf{num}$ is a function that takes a tuple of two numbers as input and returns a number.

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 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 keep type information available at runtime.

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 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 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.

1.2.2.1 Dynamic Type and Type Consistency

Gradual type system introduces a special type on top of a static type system to capture the idea that type information might only be partially known ahead of execution. The special type is usually called a “dynamic type”, we introduce a special type **dyn** for it.

To deal with dynamic types, type consistency is introduced 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; $\mathbf{num} \rightarrow \mathbf{dyn}$ is consistent with both $\mathbf{dyn} \rightarrow \mathbf{num}$ and $\mathbf{dyn} \rightarrow \mathbf{dyn}$, because all these types are functions that accepts one value and returns another and **num** and **dyn** are consistent with each other. However $\mathbf{dyn} \rightarrow \mathbf{dyn}$ is never consistent with **bool**, because the former is a function type while the latter a boolean value, whose type structures do not match.

In order to perform static typechecking, it is required that all terms in the program to must have types ahead of execution. In a gradual type system, this is no longer required: for a specific term, if programmers choose not to give a type annotation, it will be assigned type **dyn**. This still allows static typechecking to detect errors because for types that contains **dyn**, it is not totally ignored, but checked up to the point where the structure of type is known.

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 still 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.

1.2.2.2 Cast Insertion and Runtime Typechecking

As we can see gradual type system attempts to contain both statically and dynamically typed programs. However, because of presence of dynamic types, the type system cannot be sound by only have static typechecking. Imagine function $f : \mathbf{num} \rightarrow \mathbf{dyn}$ and variable $v : \mathbf{dyn}$, while the function application fv can pass static typechecking, the runtime value of t might turn out to be a string and now the application is ill-typed.

The problem lies in the fact that some terms might be assigned with types that contains **dyn**, its runtime counterpart are precise, which is not checked during static typechecking. To solve this problem, casts are inserted at appropriate locations, which then allows type errors to be detected at runtime. Suppose we have a cast function $c : \mathbf{dyn} \to \mathbf{num}$, function application fv then becomes $f(cv)$, which still typechecks, but whenever we need to pass a runtime value to f, its runtime is always checked by c to decide whether to pass the value to f or raise a type error.

1.2.2.3 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.

1.2.2.4 Gradual Guarantee

Regarding the term gradual typing itself, Siek, Vitousek, Cimini and Boyland's work gives a detailed explanation of the original intention. Several works we are going to discuss in the following few chapters respects such idea:

- Gradual typing includes both fully static and fully dynamic
- Gradual typing provides sound interoperability
- Gradual typing enables gradual evolution

1.2.3 Alternatives

There are other type systems that bring benefits of the two together. Cartwright and Fagan's introduced soft typing, which is a type system that programers do not write type annotations but use type inference to assign appropriate types to terms, and runtime checks are inserted as needed. The drawback of this design is that programmers do not have control over types. Quasi-static typing is another attempt, with the idea of dynamic type in use, it chooses to use a subtyping relation and allows both up-casts and down-casts. A second pass of plausibility checking detects incompatible casts and signals the program in question as ill-typed. However its typechecking algorithm did not receive a correct proof, and it does not statically catch all type errors (TODO: ref) in the process of attempting the proof, Siek and Taha finds a better solution, which results in gradual typing that we have seen today.

1.2.4 Discussion

Among various research works that combines benefit of static and dynamic type checking gradual typing remains one of the most popular approach. In the sprite of gradual guarantee, 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 chapter 2, we will focus on benefits of making these extensions. Chapter 3 discusses and categorizes common and language-specific challenges researchers have to face and solutions to them. Some of these challenges, however, still remains as open questions, which will be covered in chapter 4.

Chapter 2: Benefits of Gradual Typing

Gradual typing brings existing languages not just the good parts of both static and dynamic typing. The synergy also creates surprising new benefits to languages users.

In this chapter, we will explore some literatures and see in detail about these benefits that gradual typing have brought in.

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 $f : Point \Rightarrow any$. 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**.

Reticulated derives class type from this definition:

```
Class(1DPoint){
  move: Function(Named(self: 1DPoint, x: Int), 1DPoint)
}
```

And object types are similar:

```
Object(1DPoint){
  move: Function(Named(x: Int), 1DPoint)
}
```

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.

2.3.3 Dynamic Semantics

Reticulated Python is not only a practical gradual type extension, but serves as an experiment of exploring different dynamic semantics

2.3.4 Other Features

There are other features of Reticulate. In particular, since it is not always possible to know what module would be loaded at runtime, static typechecking sometimes occurs right after module is loaded.

Despite Python syntax supports annotation at function definition, it does not provide a way of annotating local variables with tyoe. Instead, Reticulated takes a step further and perform dataflow-based type inference.

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.

2.4.1 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.

2.4.2 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.

2.4.3 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`.

2.4.4 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`.

2.4.5 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 $C^\sharp$ 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.

2.5.1 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.

2.5.2 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.

Chapter 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.

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. On the other hand, 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 accomplishes the changes and helps development in these languages.

This chapter will explore various issues related to making gradual typing extensions, discuss possible solutions and evaluate choices researchers have to made about them.

3.1 Extensions to Type Systems

Gradual type system supports both fully static and fully dynamic typing, which requires support for traditional types including basic type, function type and types for structural data and a notion of dynamic type to allow presence of partial type information in the system. While type soundness is established by a combined 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.

This section will separate these extensions into two categories: **Essential extensions** are required 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.

3.1.1 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 one, which have quite a different story about how the extension is implemented.

3.1.1.1 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.

3.1.1.2 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)

3.1.2 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.

3.1.2.1 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.

3.1.2.2 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 [2], 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.

3.1.2.3 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.

3.1.2.4 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 such 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 [1].

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.

3.3.1 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.

3.3.2 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 have all methods that A requires, it is not explicitly extending A to form a proper relation of inheritance.

Structural type is a 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.

Among languages we studied, Typed Scheme extends its structure system and uses nominal types (but thanks to the expressiveness of refinement type, it is still possible to test an object structurally); Reticulated Python chooses the structural approach - despite class inheritance being allowed, duck typing is generally considered idiomatic approach in Python. On the other hand, both Safe TypeScript and Gradualtalk introduces a hybrid approach to deal with the coexistence: the former treats class types nominally, but allows viewing it structurally while Gradualtalk allows a unified syntax to allow an object to be typed nominally and structurally at the same time.

3.3.3 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 [3], 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 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.

3.3.3.1 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 value might fail because from outside it is a `Proxy` rather than an instance of the relevant class.

3.3.3.2 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`.

3.3.3.3 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.

3.3.3.4 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.

3.3.4 Differential Subtyping

Safe TypeScript features a calculus that decreases RTTI overtime - reducing runtime checks when checks could have been performed once for all.

3.3.5 Self Types

The use of self types is seen in Gradualtalk to allow type information to be preserved in chained invocations.

3.3.6 Recursive Data Types

3.4 Blame Tracking

3.5 Eval function

Languages of highly dynamic nature often come with a function `eval`, which gets a string or abstract syntax tree as input, interprets it as if it is one part of the program and return the result of execution. Such functions cannot be precisely typed and several languages have to treat `eval` function specially for this reason.

3.6 Other Performance Concerns

3.6.1 Type Erasure

3.6.2 Large Arrays

For programs that need runtime typechecks on large arrays, the overhead could be wasteful but significant.

3.7 Language-specific Challenges

3.7.1 TypeScript

3.7.2 Typed Scheme and Macros

3.7.3 Python

3.7.4 Gradualtalk and Live system

3.7.5 $C^{\sharp}$ 4.0

Chapter 4: Related Work

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

Chapter 5: Conclusion

Bibliography

[1] Lintaro Ina and Atsushi Igarashi. Gradual typing for generics. In *ACM SIGPLAN Notices*, volume 46, pages 609–624. ACM, 2011.

[2] Jeremy Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming*, pages 2–27. Springer, 2007.

[3] Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.