Types for Dynamic Languages

CS 801 PhD Seminar Report

Meetesh Kalpesh Mehta
23D0361

14th May, 2024

Department of Computer Science and Engineering
Indian Institute of Technology Bombay

Contents

Abstract . . . 1

1 Introduction **1**
1.1 Related Works . . . 4
1.2 Type Declarations From Existing Types . . . 5
1.3 Data Declarations For Completely New Types . . . 7
1.4 Challenges in Describing Types for Dynamic Languages . . . 7
1.5 Summary . . . 8

2 Type Inference and Their Limitations **9**
2.1 Milner Type Polymorphism . . . 9
2.1.1 Modeling Types as a System of Linear Equations . . . 9
2.1.2 Limitations of Milner's Type Inference System . . . 11
2.2 Summary . . . 13

3 Strength of Type Systems **14**
3.1 Describing Types in TypeScript . . . 14
3.2 Pattern Matching . . . 15
3.2.1 Example 1 . . . 15
3.2.2 Example 2 . . . 16
3.3 Pattern Matching With Recursion . . . 17
3.4 Conditional Typing . . . 18
3.5 Summary . . . 19

4 Type Systems for Program Optimization **20**
4.1 Typed Intermediate Language for ML . . . 20
4.1.1 Intensional Polymorphism . . . 21
4.1.2 Nearly Tag-Free Garbage Collection . . . 21
4.2 Confinement Types . . . 22
4.2.1 Type Rules . . . 22
4.2.2 Expression Typing Rules . . . 24
4.3 Summary . . . 25

5 Types In Dynamic Languages **26**
5.1 A Case for Typed R . . . 26
5.2 A Case for Typing Dynamic Objects in Python . . . 27
5.2.1 Class-Based Types . . . 27
5.2.2 Object-Based Types . . . 28
5.2.3 Results . . . 28
5.3 Summary . . . 28

6 Conclusion and Future Work **30**

Abstract

Dynamic languages like Javascript and R are highly expressive and often allow programmers to write code without worrying about describing various technical details. With features like dynamic typing, the developer can focus on *what values do* and not think about *how they work*. Traditionally, type systems were used to catch logical inconsistencies (usually using inference rules) during compilation; however in dynamically typed languages this is not possible. Over time, different kinds of type systems have been used to perform powerful optimizations and even aid the garbage collector.

Designing a type system for dynamically typed languages like R and Javascript is hard, not only because of their untyped nature but also because of loose(er) definitions of well-known concepts like classes. An object may gain or lose fields during execution; does the object change its type during execution or should there be a different way to model such types. Efforts on statically inferring types in presence of such language features has proven futile. However, if we forget the limitations that such dynamism causes and focus more on how it helps, we can look at these languages differently. A React app is usually trivially understood by simply reading it top to bottom; that is, there is an inherent structure to the code; React programmers understand this structure of writing code while the compiler does not. A programmer reasons about the program regarding states and higher-level abstractions, while a compiler breaks this coherence when generating lower-level code for performing analysis.

The study in this report is mainly practical; it focuses on the previous works that have used "richer" type systems to capture various properties of programs and compares them to interesting modern examples. We talk about what "types" really mean to programmers of different languages and then look at how one might classify them. We then look at classical works in this area, how type inference works, and notable optimizations performed using them. Finally, we study works that use types to model interesting problems and present our observations on using type systems in modern dynamic languages.

Chapter 1

Introduction

Types were introduced to programming languages to catch common programming errors. In the early days of programming, it was (and still is) highly desirable to be able to catch programming errors as early as possible. Manipulation of memory directly, though granting more freedom, turned out to be very error-prone. The introduction of type systems made significant strides, providing users with a robust and reliable feedback loop where most common programmer errors could be caught during compile time, Milner [12] proved that a correctly type-checked program could not go wrong at runtime. In a more traditional static type system like C and C++, a majority of type errors can be caught at compile time. In such languages, however, errors that slip through the type checker may manifest as unexpected segmentation faults. On the other side of the spectrum, languages like Java use a managed runtime to defer some of these checks to the runtime. Here, invalid memory manipulation is not just hard but impossible by design. Static type checking is very desirable for high performance, as statically ensuring a type allows the compiler to get rid of it completely. In dynamic languages where certain type guarantees cannot be ensured at static time, approaches that employ dynamic checks are more popular.

The notion of types, in the most general sense, is the restriction on the domain of values that a specific variable may hold. Algorithmically, it is an invariant on the variable that holds at every reachable state of the program.

Types in statically typed languages In statically typed languages like C and C++, all the variables that hold values in the run-time are checked and verified statically at compilation time. This allows the compiler to point out obvious errors that the programmer might have made, such as storing a 64-bit value into a 32-bit variable (such errors are common in these languages). Consider the following code snippet:

```
1 int a = 123456 + "Hello" + &main; // Typecheck ERROR: binop...
```

Here, `a` is computed by adding an integer, string, and a pointer. The typechecker determines that adding an integer to a pointer or a string is semantically invalid and throws a type error.

The typechecking system of C/C++ is not foolproof though; typecasts can easily bypass such checks:

```
1 int a = 123456; // Some malicious virtual address
2 void (fun_ptr)(int) = (void ()(int))a;
3 fun_ptr(1);
```

Here, we cast the integer `123456` into a function pointer. This can allow the runtime to jump to a random virtual addresses (most likely to end up in a segmentation fault).

In context of a statically typechecked object-oriented language like C++, variables of a certain class may end up pointing to objects of a completely different class at run-time. If such violations are not caught statically, the program can end up performing dangerous changes to the memory. Consider the following code snippet:

```
1 class A {public: int a = 10; int b = 20;};
2 class B {public: uint64_t a = 0;};
```

```
3 int main() {
4   A *a = new A();
5   B b = new B();
6   std::cout << a->a << a->b << std::endl;
7   std::cout << b->a << std::endl;
8   a = ((A) b);
9   a->b = 20; a->a = 10; // Typecheck OK, Dynamic OK, UNSAFE
10  std::cout << b->a << std::endl;
11  return 0;
12 }
```

At Line 8 we cast an object of type `B` to an object of type `A` and make the variable `a` point to it. Code in Line 9-11 works on the assumption that `a` is of type `A` while it is actually of type `B`. Such code is prone to segmentation faults and random crashes.

Even though we know the types of the variables at compile type, finding out if all operations on these variables are valid or not cannot always be determined. In languages like C and C++, the compiler assumes that the program is correct, and checking arbitrary type casts is left completely to the programmer. Such code only ends up crashing when some illegal memory is accessed. This leads to a situation where seasoned programmers can add relevant runtime checks to ensure such properties, producing very fast and efficient code, while inexperienced programmers end up with hard-to-debug code.

Types in statically typed languages with managed runtimes In contrast to C/C++, statically typed languages like Java, which are designed to be run in managed runtimes[1], provide a guarantee that a variable cannot point to an object of an unrelated type. That is, a variable of type τ is allowed to point to any object of type τ' if and only if $\tau' \sqsubseteq \tau$. This is done with a combination of static and dynamic checks. Let us consider the following code snippet:

```
1 class A {}
2 class B extends A {}
3 class C extends B {}
4
5 public class Main {
6   public static void main(String[] args) {
7     A obj1 = new A(); // Typecheck OK, dynamic OK
8     A obj2 = new B(); // Typecheck OK, dynamic OK
9     A obj3 = new C(); // Typecheck OK, dynamic OK
10    Random rand = new Random();
11    int rand_int = rand.nextInt(10);
12    Object obj4 = new Object();
13    if (rand_int > 5) {
14      obj4 = new Object();
15    } else {
16      obj4 = new A();
17    }
18    A obj = (A)obj4; // Typecheck OK, dynamic?
19    System.out.println("obj is " + obj);
20  }
21 }
```

The above Java program compiles successfully. The three variables `obj1`, `obj2`, and `obj3` (declared at Lines 7 to 9) point to objects of type `A`, `B`, and `C` respectively. Here, the subtype condition $\tau' \sqsubseteq \tau$ is satisfied and typechecks successfully. On Line 18, we see that `obj` conditionally points to an object of type `Object` or `A` (declared at Line 14 and Line 16, respectively), which means that the cast at Line 18 may conditionally be invalid. Languages like C++ let such errors slip by, while the Java Virtual Machine ensures type safety here by inserting a runtime guard right before `A obj = (A)obj4` that prevents the program from continuing execution if the typecast is semantically invalid. Such guards are known to slow down the run-time, but a slowdown is acceptable if it provides such an important guarantee at run-time.

[1] Java Virtual Machine, the managed runtime in this case, interprets Java bytecode

MODULE *Diehard*

$$
\begin{aligned}
&\textsc{extends}\ Integers\\
&\textsc{variables}\ small,\ big\\
&Init \triangleq \land (small = 0)\\
&\qquad\quad \land (big = 0)
\end{aligned}
$$

A type check ensures the following predicates hold true got all program states

$$
\begin{aligned}
TypeOK \triangleq\ & \land (small \in 0\,..\,3)\\
& \land (big \in 0\,..\,5)
\end{aligned}
$$

$$
\begin{aligned}
FillSmall &\triangleq (small' = 3) \land (big' = big)\\
FillBig &\triangleq (small' = small) \land (big' = 5)\\
EmptySmall &\triangleq (small' = 0) \land (big' = big)\\
EmptyBig &\triangleq (small' = small) \land (big' = 0)
\end{aligned}
$$

$$
\begin{aligned}
PourSmallToBig \triangleq\ & \textsc{if}\ (big + small \leq 5)\\
& \quad \textsc{then}\ \land big' = big + small\\
& \qquad\qquad \land small' = 0\\
& \quad \textsc{else}\ \ \land big' = 5\\
& \qquad\qquad \land small' = small - (5 - big)
\end{aligned}
$$

$$
\begin{aligned}
PourBigToSmall \triangleq\ & \textsc{if}\ (big + small \leq 3)\\
& \quad \textsc{then}\ \land small' = big + small\\
& \qquad\qquad \land big' = 0\\
& \quad \textsc{else}\ \ \land small' = 3\\
& \qquad\qquad \land big' = big - (3 - small)
\end{aligned}
$$

$$
\begin{aligned}
Next \triangleq\ & \lor EmptySmall \lor EmptyBig \lor FillSmall\\
& \lor FillBig \lor PourSmallToBig \lor PourBigToSmall
\end{aligned}
$$

Figure 1.1: TLA+ specification for solving the “die-hard” problem.

Typechecks in model checking Figure 1.1 showcases a solution for the “die-hard” problem using TLA+ [10]. This problem involves a 3-gallon jug, a 5-gallon jug, and a water faucet, with the goal of obtaining exactly 4 gallons of water. The solution is structured into three steps: defining the initial state of the system (*Init*), specifying possible actions (*Next*), and adding invariants for type safety and termination conditions.

The *Init* step initializes variables, setting both the small and big jugs to be initially empty. The *Next* step defines possible actions, such as emptying or filling the jugs, and transferring water between them. Additionally, the invariant *TypeOK* ensures type safety, preventing scenarios like the 5-gallon jug holding 6 gallons of water. The invariant $big \neq 4$ acts as the termination condition. The *TypeOK* type check is particularly notable because it does not specify when it should be performed; rather, it must hold true throughout any proposed solution. In languages with strict typing, static analysis tools can enforce these invariants offline, while dynamically typed languages may rely on runtime checks.

Types in dynamically typed languages In dynamically typed languages like JavaScript, R, and Python, variables are not bound to hold values of a specific type throughout the program's execution. This flexibility allows variables to hold values of different types during run-time. For instance, consider the following JavaScript code snippet:

```
let a = 10;              // 'a' is an integer
a = (a, b) => a < b;     // 'a' is a function
a = [1, 3, 2].sort(a); // 'a' is a list of integers
```

In this example, the type of variable `a` is first an integer (at Line 1), then it becomes a function of two arguments (at Line 2) and finally holds an array of integers (at Line 3). Apart from being one of these possible types at runtime, in languages like R, even the complete set of possible types cannot be determined statically; this is because the environment is reified at runtime, meaning we can never guarantee the reaching definitions for any variables statically. Any static analysis must consider all the possible paths a program might take, which in dynamic languages, can lead to imprecise analysis results, leaving little room for optimizations. In some cases types are generalized implicitly without warning, which can lead to situations like this:

```
1 let a = (a, b) => a < b;
2 let b = (a, b) => a < b;
3 let c = a + b; // What does it even mean to add two functions?
```

Here, variables `a` and `b` are functions, and the type of `c` is unclear. One might expect an error at runtime when trying to add two functions together, but this code runs successfully. The resulting type of `c` is a string representing the concatenation of the two function bodies as follows:

```
1 (a, b) => a < b(a, b) => a < b
```

Here a function was silently coerced into a string, such implicit casts can become sources of errors, both to the programmers and people implementing the language. When calling `c` as a function, it results in the following error:

```
1 let d = c(1, 2);
2 // Uncaught TypeError: c is not a function
```

The dynamic nature of typing in languages like JavaScript underscores the importance of understanding type conversions correctly during development. However, despite the challenges that dynamically typed languages introduce, they contribute to code that is concise and easy to read.

Statically compiling code for such languages does not typically yield significant performance improvements over plain interpretation. This is because common optimizations cannot be applied in the presence of such dynamism. As a result, the burden of compiling fast code shifts to the runtime environment. These languages often utilize Virtual Machines (VMs) along with Just-In-Time (JIT) compilation techniques. These techniques specialize hot parts of the code based on profiling and speculation, optimizing performance dynamically during runtime execution.

1.1 Related Works

Types have been around for a long time; typed lambda calculus was first popularized by [12]. This work also introduced the notion of type correctness, which states that "a well-typed program cannot go wrong". Over the years, types have also been used for optimization. One of the first works that discusses the possibilities of optimizing languages supporting polymorphism is [16]. Most of the optimizations discussed in this work, in some form, appear as a part of the modern Java compiler. It was understood long ago that statically inferring precise types is complicated for dynamic languages. SELF95 [7] uses runtime profiling and performs speculative optimizations to speed up the program. Techniques for dynamic deoptimization [8] solved the problem of recovering from failed speculations by using frame states. In recent times, many works on The R programming language made use of these ideas and introduced novel optimizations [6], [5], [4].

In the last 25 years, object-oriented programming has emerged as one of the most powerful abstractions; this was made possible partly due to the powerful dynamic optimizations provided by the JVM, which can profile the runtime and optimize code on the fly. Expressing objects as types proved to be very natural, Most real-world tasks can be expressed in terms of objects that interact with each other in well-defined ways. A recent work pointed out that even though Java classes are popular, they do not capture a fundamental property of what they define as the "state"; they named this concept "typestate" [2] of an object. For instance, a file may be in two states, either open or closed. When a file is closed, the only action/method associated with it is opening the file; any other action is semantically invalid. Modeling the state of a file as being open or closed using types allows the type system to detect semantic bugs in the program where a particular piece of code is trying to write to a closed file.

Describing types for dynamic languages has been studied empirically in recent times. As an instance, one recent work [17] on the R programming language collects function signatures by running and profiling a large corpus of library code, which is then verified by a proposed type system for R. On the other hand, recent work on Python [15] discusses how types of objects should be described. They look at objects through the lens of dynamism, namely constructor-based dynamism and object evolution. A popular type system developed for JavaScript is TypeScript [18]; it uses a compiler to typecheck the programs statically and then can be transpiled to JavaScript.

1.2 Type Declarations From Existing Types

In Haskell, types that derive from existing types are declared using the `type` keyword. For instance, in Haskell, there is really no such type as a String; it is simply a synonym to a list of characters defined as follows:

```
1 // Type String in Haskell
2 type String = [Char]
3
4 // One might view it as a typedef in C.
5 typedef char* string;
```

We are also allowed to declare types in terms of other declared types. For instance the following type declaration of `Distance` is defined in terms of `Point`.

```
1 // Type Distance described in terms of Type Point in Haskell
2 type Point = (Int,Int)
3 type Distance = Point -> Point -> Int
4
5 // One might view it as the following in C (this is not entire
    true as we will see later)
6 typedef struct Point {
7   int a, b;
8 } Point;
9 typedef int(*Distance)(Point, Point);
```

Even though the type mechanisms of Haskell and C/C++ seem to be very similar in nature, there are a few stark differences. The first limitation of the `type` keyword is the inability to declare recursive types. Though it is popularly believed that C and C++ do not support recursive types, this is not entirely true. Over the years, C++ has gained support for *Containers of Incomplete Types*. This allows C++ to declare recursive data structures of the following form:

```
1 // Invalid in Haskell
2 type Tree = (Int, [Tree])
3
4 // Valid in C++
5 struct Tree {
6   int data;
7   std::vector<Tree> next;
8 };
9
10 // "Not" a recursive type in C, but has similar functionality
11 struct Tree {
12   int data;
13   Tree * next;
14 };
```

We see that the Haskell type system imposes some strict limitations on the types declared using `type`. This is by design as this allows for a natural way to describe structural equivalence in types. Let us consider the following code snippet:

```
1 // Type A and Type B in Haskell are synonyms
2 type A = (Int,Int)
3 type B = (Int,Int)
```

```
4
5 f :: A -> B
6 f (a,b) = (b,a)
7
8 g :: A -> Int
9 g (a,b) = a + b
10
11 // Executing function f and g
12 g (f (1,2))
```

In the above example, the function `f` takes a parameter of type `A` and returns a value of type `B`. In Haskell, `A` and `B` are exactly the same thing; they can be used interchangably. This functionality can be achieved in C using `typedefs`, while structures of the same form are not naturally the same. For example, the following structs cannot be used interchangably:

```
1 typedef struct A {
2   int a, b;
3 } A;
4
5 typedef struct B {
6   int a, b;
7 } B;
8
9 B f(A a) { B res; res.a = a.b; res.b = a.a; return res; }
10 int g(A a) { return a.a + a.b; }
11
12 int main() {
13   A a;
14   a.a = 1;
15   a.b = 2;
16   g(f(a)); // There is an error here, invalid argument type
17   return 0;
18 }
```

Instead of extracting out the types and declaring typedefs, a C programmer might proceed to cast struct of type `A` into type `B` using expressions like `g((A)f(a))`, which is bad practice. This shows a philosophical difference of how types are supposed to used in these two languages; Both are capable of representing the same ideas but both promote different kind of reasoning about the programs.

Finally, similar to generics in C++, we can pass parameters to types in Haskell.

```
1 // Parameterized type in Haskell
2 type Together a b = (a, b)
3
4 // C++ Equivalent using generics, not exactly semantically
    equivalent!
5 template <class T, class U>
6 class Together {
7   T a;
8   U b;
9 };
```

The reason for why Haskell types declared using `type` are treated as synonyms while types declared using `struct` in C/C++ is the concept of "constructor". In case of structs and classes, the concept of constructor is implicit, whereas in case of Haskell it is explicit. This means that when we want to associate values with a specific unique tag (i.e. a classname, structure name, etc) we use the "data" mechanism in Haskell. We see this mechanism in the following section.

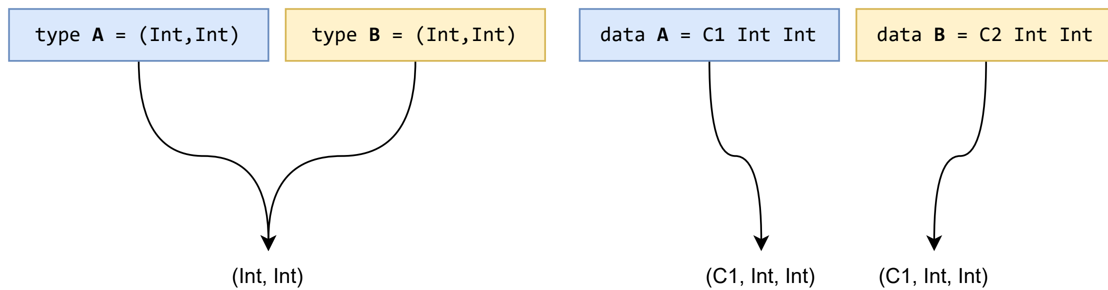


Figure 1.2: Constructors in Haskell.

1.3 Data Declarations For Completely New Types

Sometimes it may be desirable to introduce completely new types to the language altogether; in Haskell this is done using the *data* mechanism. Figure 1.2 shows how `data` maps values differently than `type`. We can see how the concept of constructor was implicit in case of structures in C and C++ [2]; in Haskell this property is made explicit.

Constructors also allow Haskell programs to support intensional polymorphism (we want to invoke different functionality depending on the type of the argument passed to a function) whenever desired; this is popularly implemented using pattern matching. Let us consider the following code snippet:

```
1 data Shape = Circle Float | Cylinder Float Float
2
3 area :: Shape -> Float
4 area (Circle r) = pi * r * r
5 area (Cylinder r h) = pi * r * r * h
```

Here the type `Shape` can be made using two kinds of "constructors", namely `Circle` and `Cylinder`. Compared to normal functions, the constructor functions simply exist to contain pieces of data. The constructor can then be used for pattern matching (as seen in the area function). The area function, takes an object of type `Shape` as input and returns a `Float`, here the constructor is used for pattern matching. The following code snippet shows how this can be viewed in the sense of imperative languages like C/C++/Java.

```
1 area :: Shape → Float
2
3 area □ = if □ instanceof Circle
4             then return area' ((Circle)□)
5           elif □ instanceof Cylinder
6             then return area'' ((Cylinder)□)
7           else Throw Error
8
9 area' (Circle □) = pi * □ * □
10
11 area'' (Cylinder □ □') = pi * □ * □ * □'
```

The `data` mechanism also supports recursive types to be defined; consider the following code snippet.

```
1   data Tree = Node Int [Tree]
```

1.4 Challenges in Describing Types for Dynamic Languages

Most of the concepts discussed in the previous two sections are very intuitive. The underlying assumption with types is that shapes are mostly fixed and are not subject to change during run-time. In case of languages like R, Javascript and Python this may not necessarily be true.

[2] C++ uses name equivalence for structures.

Types in dynamically typed languages tend to be even more nuanced, for instance consider the following code example in Python:

```
class A:
  def __init__(self,title,width):
    self.title = title
    self.width = width
    if self.width is not None:
      self.height = self.width

  def setheight(self,height):
    self.height = height

panel1 = Panel(Text(),42)            # Object with three fields
panel2 = Panel("Example Table",None) # Object with two fields
panel2.width = 5      # modification
panel2.setHeight(42) # extension
```

Traditionally, there is an assumption that objects of the same class will have the same shape, this notion does not hold true in dynamic languages such as Python and JavaScript. At Line 11 and Line 12, two distinct objects are instantiated. The first object contains three fields, namely `title`, `width`, and `height`, whereas the second object possesses only two fields (`height` is missing). Line 13 modifies the `width` field of the second object, converting its type from `NoneType` to `Int`. Line 14 then introduces a new field to the second object. As the program progresses we see that not only the types of the fields may change but also its shape. Figure 1.3 demonstrates the progression of objects referenced by `panel1` and `panel2`. Efforts on modeling type systems where such dynamism is prevalent is an active area of research.

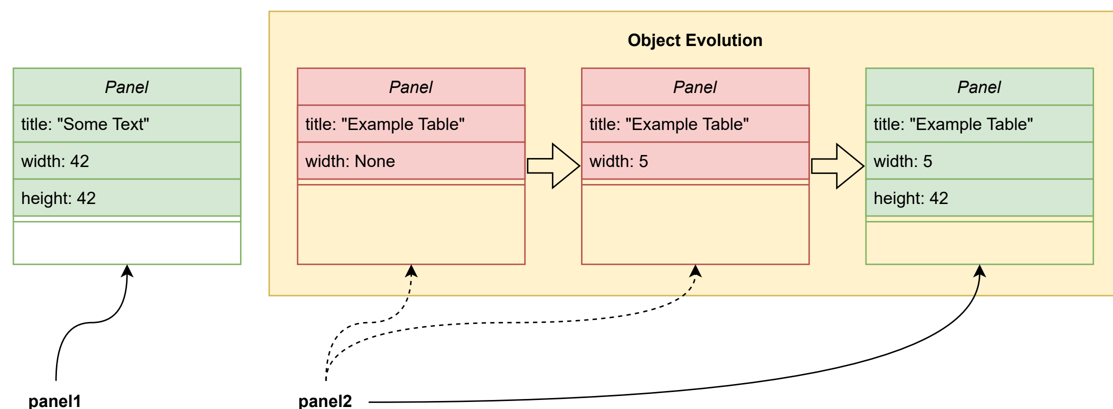


Figure 1.3: Object Evolution in python.

1.5 Summary

This chapter discussed the concept of types in various different contexts. We learnt that types in languages are a compile time property in languages like C and C++, whereas their meaning is mixed in managed languages like Java. In model checkers they are used as invariants on the program variables ensuring that they do not contain invalid values at any reachable program state. Section 1.1 discussed the related works when it comes to modeling, describing and using type systems. Section 1.2 discussed the way Haskell allows creation of types and compares them to languages like C and C++. Section 1.3 discussed how new primitives are added to Haskell and the concept of constructors. Section 1.4 discussed the challenges in modeling types in dynamically typed languages where types of objects can change and even evolve over time.

Chapter 2

Type Inference and Their Limitations

Building an intuition for types is easy; in a lot of ways, they are analogous to sets in math. Some specialized programming languages like Agda [1] provide sets as fundamental quantities to reason with, but reasoning about computing systems with sets alone is not always convenient. So, programming languages allow users to define restrictions on arbitrary sets, known as "types". Approaches like model checking, such as TLA+ [10], can generally enforce invariants of the form "this never contains", "this is exactly", and "one of the following" on the variables at each step. The ability to add these invariants is invaluable to find errors in the proposed logic.

This chapter answers the following questions: *How is type inference done? What are its limitations? Are some type systems stronger than others?*

2.1 Milner Type Polymorphism

Polymorphism allows users to define functions that work with various types, similar to many primitive operations in math. For instance, addition in math polymorphically changes its meaning when we work with integers vs. when we work with matrices. Such polymorphism is not new in programming languages; however, a formal type discipline was lacking. Milner's work [12] explores the fundamental concepts and practical implications of type polymorphism within the context of programming languages.

2.1.1 Modeling Types as a System of Linear Equations

Let us consider the following code example that defines the `map` function.

```
1 letrec map f m = if (null (m)) then nil
2                  else cons (f (car(m))) (map f cdr(m))
```

The function `map` takes a function `f` and a list `m` and applies the function `f` to each element of the list `m` to obtain a new list. The first argument is a function that takes elements of type α and produces a result of type β, and the second argument is a list of type α. The result of applying the function `map` on these two arguments produces a list of type β. Generally, this is represented as follows:

```
1 typeof(map) = (α → β) → α list → β list
```

The identification of such types for functions can be done by modeling them as linear equations. Let us first consider the free identifiers[1] in the function body of `map`. We will consider the following generic types for these free identifiers:

```
1 null : α list → bool
2 nil : α list
```

[1] These are identifiers in the function body that are not bound by any argument of the current or parent function

```
3 car : α list → α
4 cdr : α list → α list
5 cons : α → α list → α list
```

For identifiers that occur once we denote the types for each identifier `id` using σ_{id} along with type variables $\tau_1, ...\tau_5$ (note that within a type instantiation we use the same type variable for the different instantiations).

$$
\begin{array}{ll}
1 & \sigma_{null} = \tau_1 \ \ list \rightarrow bool \\
2 & \sigma_{nil} = \tau_2 \ \ list \\
3 & \sigma_{car} = \tau_3 \ \ list \rightarrow \tau_3 \\
4 & \sigma_{cdr} = \tau_4 \ \ list \rightarrow \tau_4 \ \ list \\
5 & \sigma_{cons} = \tau_5 \rightarrow \tau_5 \ \ list \rightarrow \tau_5 \ \ list
\end{array}
$$

For remaining identifiers, we ensure the same type for each identifier is used; this lets us tie these type equations together. Also, the equations must be satisfied for some arbitrary types $\rho_1, \rho_2 ...$

$$
\begin{array}{ll}
1 & \sigma_{map} = \sigma_f \rightarrow \sigma_m \rightarrow \rho_1 \\
2 & \sigma_{null} = \sigma_m \rightarrow bool \\
3 & \sigma_{car} = \sigma_m \rightarrow \rho_2 \\
4 & \sigma_{cdr} = \sigma_m \rightarrow \rho_3 \\
5 & \sigma_f = \rho_2 \rightarrow \rho_4 \\
6 & \sigma_{map} = \sigma_f \rightarrow \rho_3 \rightarrow \rho_5 \\
7 & \sigma_{cons} = \rho_4 \rightarrow \rho_5 \rightarrow \rho_6 \\
8 & \rho_1 = \sigma_{nil} = \rho_6
\end{array}
$$

We now solve this set of linear equations using Robinsons Unification Algorithm [14]. Let $\sigma_t = (\alpha \rightarrow \beta)$ and $\sigma_m = \alpha \ list$. Each step of unification is depicted in a different color to drive the intuitive understanding of how the algorithm proceeds.

$$
\begin{array}{ll}
1 & \sigma_{map} = (\alpha \rightarrow \beta) \rightarrow \alpha \ \ list \rightarrow \rho_1 \\
2 & \sigma_{null} = \alpha \ \ list \rightarrow bool \\
3 & \sigma_{car} = \alpha \ \ list \rightarrow \alpha \\
4 & \sigma_{cdr} = \alpha \ \ list \rightarrow \rho_3 \\
5 & \sigma_f = (\alpha \rightarrow \beta) \\
6 & \sigma_{map} = (\alpha \rightarrow \beta) \rightarrow \rho_3 \rightarrow \rho_5 \\
7 & \sigma_{cons} = \beta \rightarrow \rho_5 \rightarrow \rho_6 \\
8 & \rho_1 = \sigma_{nil} = \rho_6
\end{array}
$$

$$
\begin{array}{ll}
1 & \sigma_{map} = (\alpha \rightarrow \beta) \rightarrow \alpha \ \ list \rightarrow \rho_1 \\
2 & \sigma_{null} = \alpha \ \ list \rightarrow bool \\
3 & \sigma_{car} = \alpha \ \ list \rightarrow \alpha \\
4 & \sigma_{cdr} = \alpha \ \ list \rightarrow \alpha \ \ list \\
5 & \sigma_f = (\alpha \rightarrow \beta) \\
6 & \sigma_{map} = (\alpha \rightarrow \beta) \rightarrow \alpha \ \ list \rightarrow \rho_5 \\
7 & \sigma_{cons} = \beta \rightarrow \rho_5 \rightarrow \rho_6 \\
8 & \rho_1 = \sigma_{nil} = \rho_6
\end{array}
$$

$$
\begin{array}{ll}
1 & \sigma_{map} = (\alpha \rightarrow \beta) \rightarrow \alpha \ \ list \rightarrow \rho_1 \\
2 & \sigma_{null} = \alpha \ \ list \rightarrow bool \\
3 & \sigma_{car} = \alpha \ \ list \rightarrow \alpha \\
4 & \sigma_{cdr} = \alpha \ \ list \rightarrow \alpha \ \ list \\
5 & \sigma_f = (\alpha \rightarrow \beta) \\
6 & \sigma_{map} = (\alpha \rightarrow \beta) \rightarrow \alpha \ \ list \rightarrow \beta \ \ list \\
7 & \sigma_{cons} = \beta \rightarrow \beta \ \ list \rightarrow \beta \ \ list \\
8 & \rho_1 = \sigma_{nil} = \beta \ \ list
\end{array}
$$

$$
\begin{array}{ll}
1 & \sigma_{map} = (\alpha \rightarrow \beta) \rightarrow \alpha \ \ list \rightarrow \beta \ \ list
\end{array}
$$

Such unification-based type-checking is very popular in modern programming languages, wherein most compilers implement variations of this algorithm.

Example 1: Inference in C++

C++ is a well-known statically typed language supporting advanced features such as generics and templates. Some time ago, the `auto` keyword was introduced to the language based on popular demand. This feature allows programmers to skip providing type declarations for the variables, and the compiler automatically determines the type during compilation. Let us consider the following C++ code snippet.

```
1 std::vector<int> foo(int a) {
2   auto vec;
3   return vec;
4 }
```

If we used the same tools of modeling we studied earlier, we could model the problem of inference of the type of `vec` as follows.

1 $\sigma_{foo} = \tau_1 \rightarrow \tau_2$
2 $\sigma_{vec} = \tau_3$
3 $\sigma_{ret} = \sigma_{foo} = \sigma_{vec}$

Let us solve the equations for τ_3 by substuting the known types of `foo`.

1 $\sigma_{foo} = \sigma_a \rightarrow \sigma_{ret}$
2 $\sigma_{vec} = \tau_3$
3 $\sigma_{ret} = \sigma_{foo} = \sigma_{vec}$

1 $\sigma_{foo} = int \rightarrow int\ \ vector$
2 $\sigma_{vec} = int\ \ vector$
3 $\sigma_{ret} = \sigma_{foo} = \sigma_{vec}$

Though this looks straightforward, the C++ compiler fails such inference. The specific rule in the C++ specification that disallows such inference is:

...the type of the variable that is being declared will be automatically deduced from its initializer...

Here, we see that the variable `vec` has no initializer; under such scenarios, the C++ compiler fails to make type inference, even though it may seem like an arbitrary limitation in this case.

Example 2: Inference in TypeScript

If we rewrite the previous example in TypeScript we find that this works without any problems.

```
1 function foo(a: number): number[] {
2   let vec;
3   return vec;
4 }
5 let v: number[] = foo();
```

2.1.2 Limitations of Milner's Type Inference System

At first look, type inference based on unification is a natural way to infer types of variables and perform type checking. However, there are some fundamental limitations to approaching such inference. The primary consideration is how to model generic types; in the previous example, it was natural to assume that two instances of the same generic type will behave the same, but this is not always true. Consider the following example of applying the previously defined function `map`(see Section 2.1.1) being applied twice, here `tokl` is a list of tokens.

```
1 map(sqroot, map(length, tokl))
2
```

3 $\sigma_{map} = (tok \rightarrow int) \rightarrow tok\ \ list \rightarrow int\ \ list$
4 $\sigma_{map} = (int \rightarrow real) \rightarrow int\ \ list \rightarrow real\ \ list$

In this example we see that there are two different ways to infer the type of the function `map`. Handling this situation is difficult, and exposes some limitations of the previously presented approach.

To see what options we have when we encounter such limitations let us consider another example, the function `reversepair` reverses the elements of the tuple and returns the new tuple.

```
1 let reversepair (x, y) = (reverse(x),reverse(y))
```

Modeling the same using unification gives us the following; here the function `reverse` has the signature $\alpha\ list \rightarrow \alpha\ list$.

$$
\begin{aligned}
&1\quad \sigma_{reversepair} = (\tau_1, \tau_2) \rightarrow \tau_3\\
&2\quad \sigma_{reverse} = \tau_3 \rightarrow \tau_4\\
&3\quad \sigma_{reverse} = \tau_5 \rightarrow \tau_6\\
&4\quad \sigma_{reversepair} = (\tau_3, \tau_5)
\end{aligned}
$$

$$
\begin{aligned}
&1\quad \sigma_{reversepair} = (\sigma_x, \sigma_y) \rightarrow \tau_3\\
&2\quad \sigma_{reverse} = \sigma_x \rightarrow \tau_4\\
&3\quad \sigma_{reverse} = \sigma_y \rightarrow \tau_6\\
&4\quad \sigma_{reversepair} = (\sigma_x, \sigma_y)
\end{aligned}
$$

Unifying with argument supplied as $(\alpha\ list, \beta\ list)$:

$$
\begin{aligned}
&1\quad \sigma_{reversepair} = (\alpha\ list, \beta\ list) \rightarrow \tau_3\\
&2\quad \sigma_{reverse} = \alpha\ list \rightarrow \tau_4\\
&3\quad \sigma_{reverse} = \beta\ list \rightarrow \tau_6\\
&4\quad \sigma_{reversepair} = (\alpha\ list, \beta\ list)
\end{aligned}
$$

If we look at the result of unification, we run into the following problem. Here we are forced to unify the two different definitions for $\sigma_{reverse}$.

$$
\begin{aligned}
&1\quad \sigma_{reversepair} = (\alpha\ list, \beta\ list) \rightarrow \tau_3\\
&2\quad \sigma_{reverse} = \alpha\ list \rightarrow \alpha\ list\\
&3\quad \sigma_{reverse} = \beta\ list \rightarrow \beta\ list\\
&4\quad \sigma_{reversepair} = (\alpha\ list, \beta\ list)
\end{aligned}
$$

As a result of the previous unification we infer $\alpha == \beta$.

$$
\begin{aligned}
&1\quad \sigma_{reversepair} = (\alpha\ list, \alpha\ list) \rightarrow (\alpha\ list, \alpha\ list)\\
&2\quad \sigma_{reverse} = \alpha\ list \rightarrow \alpha\ list\\
&3\quad \sigma_{reversepair} = (\alpha\ list, \alpha\ list)
\end{aligned}
$$

What does this mean? Firstly it means that for `reversepair` function to be typechecked correctly, it must only accept tuples where the type of both of its elements is same. Some approaches to handle such situations are rather straighforward. We could create a separate copy of the method `reverse` on each instance as follows.

$$
\begin{aligned}
&1\quad \sigma_{reversepair} = (\alpha\ list, \beta\ list) \rightarrow (\alpha\ list, \beta\ list)\\
&2\quad \sigma_{reverse_a} = \alpha\ list \rightarrow \alpha\ list\\
&3\quad \sigma_{reverse_b} = \beta\ list \rightarrow \beta\ list\\
&4\quad \sigma_{reversepair} = (\alpha\ list, \alpha\ list)
\end{aligned}
$$

Another idea would be to instantiate type variables; Milner's work terms them as generic. However, not all generic types can instantiated, *only the types that do not appear as part of any enclosing formal parameters are allowed to be instantiated.* In practice, this is equivalent to the previously described solution. However, the updated definition of generics is also not complete and runs into complications. For example the following function that does the same thing as the previous example cannot be well typed using this system:

```
1 let Func f (a,b) = (f(a), f(b))
```

Here we are not allowed to instantiate the type variables for `f`, which means we end up restricting the domain of `f` in `func`.

An assumption about conditionals It is worth noting that all the previous examples presented naturally assumed that both the branches of a conditional return values of the same type. This is by design for many languages, but not always true. Some languages like Javascript allow different types to be returned from different branches. The modeling of such languages requires the concept of *conditional/dependent types* (discussed in the upcoming chapters) which allows such behaviour to be typechecked.

2.2 Summary

This chapter presented the concepts of type inference presented by Milner. Section 2.1.1 presented the way programs can be modeled as linear equations and solved using Robinson's Unification Algorithm. Section 2.1.1 and Section 2.1.1 gave some examples on how inference may work differently in different modern languages. Section 2.1.2 discussed the limitations of Milner's system and suggested how they can be overcome.

Chapter 3

Strength of Type Systems

Type checking involves modeling the program using type variables and then performing unification. If the unification is successful, we say that the type checker was successful; otherwise, we throw a type error. Naturally, the question emerges: are all types of systems of all languages the same? In the last chapter, we encountered a seemingly trivial type inference limitation in C++ (see Section 2.1.1) and found that it was not a limitation in case of TypeScript [18]. This first part of this chapter shows how types are described in TypeScript and the second part describes how complex properties can be type checked.

3.1 Describing Types in TypeScript

Similar to different ways types are described in Haskell (see Section 1.2 and Section 1.3), we will quickly look at how types get described in TypeScript. Let us consider the following TypeScript code snippet.

```
1  // 1. Using values in types
2  type Music = "Eminem" | "LinkinPark";
3  // Domain of Music = { "Eminem", "LinkinPark" }
4
5  // 2. Using values as types
6  type Mee = { name: "Meetesh", address: "IITB" };
7  type Yoo = [ 1, 2, 3 ];
8  // type Mee = {
9  //     name: "Meetesh";
10 //     address: "IITB";
11 // }
12 //
13 // type Yoo = [1, 2, 3]
14
15 // 3. Infer types from values
16 const foo = { bar: "baz", bam : { a: 12, b: {}} };
17 type Foo = typeof foo;
18 // type Foo = {
19 //   bar: string;
20 //   bam: {
21 //       a: number;
22 //       b: {};
23 //   };
24 // }
```

The only notable difference is the lack of keyword `data`; instead, `type` keyword suffices when declaring completely new types. In the first case, `Music` is a type with the domain $\{“Eminem”, “LinkinPark”\}$. The domain of the second case is the entire object (TypeScript allows complex values to be types); when we make use of types `Mee` and `Yoo`, it will unify every single value. The third case is similar to the second one, but the `typeof` operator allows us to

infer types in a more general sense. To get a clearer sense of what these complex types look like let us consider more examples:

```
// Example 1
type Mee = { name: "Meetesh", address: "IITB" };
let a : Mee = { name: "Meetesh", address: "IITM" }; // TypeCheck
    ERROR

// Example 2
let a : Mee = { } as Mee // TypeCheck OK, bypass the TypeChecker

// Example 3
const foo = {
                bar: {
                  a: 12,
                  b: { name: "Meetesh", address: "IITB" }
                }
              };
type b = typeof foo
// type b = {
//     bar: string;
//     bam: {
//         a: number;
//         b: {
//             name: string;
//             address: string;
//         };
//     };
// }

// Example 4
const baz = { bar : { a: 12, b: a} };
type c = typeof baz

// type c = {
//     bar: {
//         a: number;
//         b: Mee; // Notice the difference here
//     };
// }
```

The first example shows how values get matched when complex types are checked in TypeScript; here, `"IITM"` cannot be matched to `"IITB"`. However, in the second example we see that we can bypass the typecheck by using `as`. In the third example, we see a complex type being inferred from a value using `typeof` keyword. In the fourth example, we see that the resultant type `c` finds infers the value of `bar.b` as `Mee`. This is because we explicitly made use of an object of type `Mee` i.e. `a`.

3.2 Pattern Matching

Pattern matching in principle is similar to the one presented in Chapter 1.3. It allows us to compare a value against a specific structure and reason about them. An interesting feature of TypeScript is the support for pattern matching and recursion when describing types. Let us consider the following examples.

3.2.1 Example 1

Let us consider the following TypeScript example that makes use of pattern matching to infer the type `Foo1`.

```
type ExtractFoo<T> =
```

```
2   T extends { foo: infer U }
3   ? U
4   : T extends { bar: { foo: infer U } }
5     ? U
6     : never;
7
8 const withFoo = { foo: new Date() };
9 type Foo1 = ExtractFoo<typeof withFoo>;
```

First, the object `withFoo` declared at Line 8 gets evaluated:

```
1 const withFoo = { foo: new Date() };
2 // withFoo = {
3 //   foo: [Date Object]
4 // }
```

Second, the `typeof` operator is being used to infer the type of `withFoo`, which gives us the following:

```
1 // const withFoo: {
2 //   foo: Date;
3 // }
```

Third, the inferred type is given to the `ExtractFoo` function. The following snippets show the evaluation process, step-by-step.

```
1 ExtractFoo { foo: Date } =
2     if { foo: Date } extends { foo: □ }
3       return □
4     if { foo: Date } extends { bar: { foo: □' } }
5       return □'
6     else
7       unreachable
```

```
1 ExtractFoo { foo: Date } =
2     if { foo: Date } extends { foo: Date }
3       return Date
4     if { foo: Date } extends { bar: { foo: □' } }
5       return □'
6     else
7       unreachable
```

```
1 ExtractFoo { foo: Date } = Date
```

The `infer` keyword is used to match arbitrary patterns; they can be used to perform selections such as first element of a list, the remaining elements of a list, etc.

3.2.2 Example 2

```
1 type InferT<T> =
2   T extends readOnly [...infer U]
3   ? U
4   : T;
```

Here `InferT` gets rid of the readOnly property of a type and returns the listof types `U`.

```
1 type InferT (readonly [int, string, float]) =
2   (readonly [int, string, float]) extends readOnly [□]
3   ? □
4   : (readonly [int, string, float]);
```

```
type InferT (readonly [int, string, float]) =
  (readonly [int, string, float]) extends readOnly [
   listof[int, string, float]]
  ? listof[int, string, float]
  : (readonly [int, string, float]);
```

```
type InferT (readonly [int, string, float]) = listof[int, string, float]
```

3.3 Pattern Matching With Recursion

A type system's strength is indirectly tested by how many valid behaviours it can typecheck successfully. Just being able to typecheck is not sufficient though; we want to be as precise as possible. Precise types are instrumental to further optimizations down the line. For instance, consider a varidaic [1] `rev` function in javascript with the following signature:

```
rev :: argList(α) → [α]

let a = [1,"Hello", 1.1] // What is the typeof a here?
let b = rev(a).first() // we want to infer b as 'float'
```

The `rev` function takes a variable number of arguments as input and returns a list. We pass the list `a` to this function. Here the type of `a` is `[(Int | String | Float)]`, meaning each element of the array is one of these three types. If we use a simplistic type definition for `rev` :: $argList(\alpha) \to \alpha$ (which says that it is a function that can take a list of arguments of type α and return a list of type α) the type of `b` will be inferred as `(Int | String | Float)` . In essence we want to describe what the `rev` function does to the type system more precisely. We do this in two steps, first we describe how inference of type of `a` can be improved, next we look at how we can model the type of `rev` such that the type inferred for `b` is precise.

For `a`, we can preserve the order of these types by using `const` as follows:

```
let a = [1,"Hello", 1.1] as const
```

Here the types for `a` are correctly distinguished, `[Int, String, Float]`. Now, we want to model a more precise type for `rev`. Let's say somewhere in our language's standard library we have a `rev` function that is implemented as follows:

```
// Code somewhere
function rev(...args) {
  return args.reverse();
}
```

This function takes a list of arguments and reverses the arguments. This is basically a reverse function, on the arguments.

Now our goal is to typecheck our program which makes use of this function, we would like to able to describe to our type checker the following predicate.

Any types provided to the function `rev` *in order* $\tau_1, \tau_2, \tau_3, .., \tau_n$ *will give back* $\tau_n, \tau_{n-1}, .., \tau_1$.

More formally, we want to type inference of `rev` to be the following:

```
rev :: (α1 × α2 × α3...αn) → [αn, αn−1, αn−1...α1]
```

We never come across the need for this kind of a typecheck in languages like Java and C++, because they simply disallow non-homogenous types of lists. That does not mean we cannot store such objects; it simply means that the type system will generalize all objects being stored into the most generic supported type. In case of Java this would be the `Object` class.

```
type Reverse<T extends any[]> =
  T extends [infer T1, ...infer Ts]
  ? [ ...Reverse<Ts>, T1 ]
  : T;
```

[1] Ability to take variable number of arguments.

```
5
6 declare function rev<T extends any[]>(...ts: T): Reverse<T>;
7 const isReverse = rev(1, true, 'hero');
8 % [string, boolean, number]
9
10 const isReverse1 = rev(...[1, true, 'hero'] as const);
11 % ['hero', true, 1]
```

In the above code snippet we see the parameterized type `Reverse`, which takes as input one parameter of type array and pattern matches its body. We see that the logic for pattern matching `[infer T1, ...infer Ts]` is similar to the pattern matching found in Haskell where we pattern match against `(x:xs)`. We can recursively describe the definition of `Reverse` in terms of itself; the implementation is very similar to the reverse function that can be written in Haskell as follows.

```
1 reverse :: [a] -> [a]
2 reverse [] = []
3 reverse (x:xs) = reverse xs ++ [x]
```

It is very interesting to see that a program written in Haskell to reverse a list, with minor modifications can be used for static typechecking of the same logic for a Javascript program. *One is a strongly typed languge with room for very little dynamism and the other is a dynamically typed language with no support for static type checking.*

3.4 Conditional Typing

When there is a possibility of narrowing the type of a variable, we would like to keep the most precise type around. Let us consider the following example in TypeScript. Say we have an API call, that based on some condition returns one of two results.

```
1 type Basic = {
2   user: string;
3   url: string;
4 }
5
6 type Extra = {
7   user: string;
8   url: string;
9   extra: {
10     last_online: string;
11     hours_online: number;
12   };
13 }
14
15 function apiCall(extra: boolean): Basic | Extra {
16   if (extra) {
17     return new Extra()
18   } else {
19     return new Basic()
20   }
21 }
22
23 const val1 = apiCall(true); // We want a precise type here!
24 console.log("Online for : " + val1.extra.hours_online);
```

In the given code snippet, the current inference of `val1` (at Line 23) gives us the result `(Basic | Extra)`. In the following print statement (at Line 24) we are dereferening `val1` using the fields `val1.extra.hours_online`. Due to the imprecise type inferred for `val1` we cannot be sure if this dereference will always succeed, it may fail at runtime. Due to this imprecision we end up adding runtime checks that ensure the fields actually exist during the execution.

Conditional types can allow the typesystem to narrow the types of values based on conditionals. The resultant types are allowed to be derive from existing types as-well-as values (sometimes termed as being "dependent" or "dependently typed"). Consider the following updated code snippet making use of conditional types in TypeScript:

```
1  type CondT<T extends boolean> =
2      T extends true ?
3        Extra
4      : Basic;
5
6  function apiCall<E extends boolean>(extra: E): CondT<E>;
7
8  function apiCall(extra: boolean): CondT<typeof extra> {
9    if (extra) {
10     return new Extra() : CondT<typeof extra>
11   } else {
12     return new Basic() : CondT~<typeof extra>
13   }
14 }
15
16 const val1 = apiCall(true); // type Extra inferred
17 console.log("Online for : "+val1.extra.hours_online); // no checks
```

In this example at Line 16, the type inferred type for `val1` now is `Extra`. This is because the call to `apiCall` describes its return type as being dependent on the argument `extra` passed to it. This is represented as `CondT<typeof extra>` at Line 8. When an argument is passed to `CondT`, its type is resolved by using pattern matching (see Line 2). Also notice the how type inference will unify Lines 8,10 and 12, ensuring the returned types are consistent.

The ability for types to depend on other values is also sometimes referred to as dependent types, however, TypeScript does not support it fully yet. However, features such as pattern matching and conditional types allows the type system to perform a large number of checks that a dependently typed system would be able to do.

3.5 Summary

This chapter focused mainly on the powerful abstractions provided by type systems of languages like TypeScript, which can allow us to typecheck complex program behaviours that many popular languages do no currently support. Section 3.1 provided a primer on how types are described in TypeScript. Section 3.2 discussed the concept of pattern matching in context of type systems. Section 3.3 discussed the concept of pattern matching along with recursion in context of type systems. Section 3.3 also discussed the problem of typechecking reversal of arguments in a varidaic function. Finally, Section 3.4 presented the concept of conditional typing along with its ability to be used for narrowing.

Chapter 4

Type Systems for Program Optimization

Types in intermediate languages have been a popular means of performing compiler optimization. Typed intermediate languages allow the compiler to disambiguate polymorphism, determine reaching stores, and, generally, reason about dynamic language features. One of the earliest works that proposed using a typed intermediate language to perform optimization was by [16]. The optimizations proposed by this work have now become standard in languages like Java. The general rationale behind this work was to alleviate the overhead due to polymorphism, both in terms of polymorphic calls and performing garbage collection. Later, languages like MLton [20] extended these ideas to eliminate all polymorphism by transforming the whole program. Apart from optimization, they can also guarantee specific safety properties of a language. [21] proposes a non-intrusive extension to Java that guarantees that specific classes do not escape the scope of its packages. Popular languages, like Rust, take this a step further and provide memory safety using ownership types.

Traditional optimizations, based on types, are only sometimes possible for dynamically typed languages. Type inference is hard (in languages like Python and Javascript) and nearly impossible (in languages like R), coupled with the fact that the traditional notion of types is not even true because of how objects exist in these runtimes. For instance, optimizing programs written in the R Programming Language is particularly hard [4]; it implements lazy evaluation of arguments, supports late binding, and allows users to modify the runtime stack on the fly. To enable meaningful optimizations based on types for dynamic languages, managed runtimes like V8 [19] for Javascript, JVM [9] for Java, and Rsh [22] for R. These runtimes use speculation (mainly on types) to allow optimizations. However, it is not always clear what "types" an intermediate language should focus on in such languages.

4.1 Typed Intermediate Language for ML

TIL [16] introduced a typed intermediate language for optimizing ML code. The programs produced by TIL were three times faster than conventional ML programs while using one-fifth of the memory. Keeping types around as code while transforming it from one intermediate language to another is quite helpful; for instance, knowing the types of variables allows the compiler to fix variable sizes and even load them onto the stack instead of having all allocations on the heap. Over the years, this separation of stack variables and heap objects has become a standard technique when modeling memory abstractions (points-to analysis performed on Java programs).

This work was one of the first to discuss the usefulness of statically knowing types in the optimization context where objects and polymorphism exist. This work discusses two main optimizations: (i) intensional polymorphism and (ii) nearly tag-free garbage collection. The first optimizations allows polymorphic calls to treat types as values inside functions for dynamic dispatch at run-time while also optimizing call sites where precise types can be determined statically. The second optimization removes tags for most stack variables. In strongly typed languages, types for all variables are known statically (if not precise types, at least whether they are pointers or not is known); this allows the compiler to associate tag information with functions.

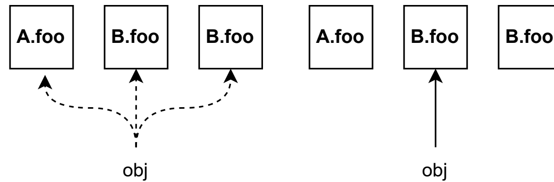


Figure 4.1: Using type information to disambiguate call site in Java.

Further, stack frames can also use liveness analysis results to optimize garbage collection.

4.1.1 Intensional Polymorphism

Intensional polymorphism allows types to be constructed and passed as values at runtime. A good reason for supporting runtime polymorphism is that it allows more expressive programming. An object can be used to pass and receive multiple arguments from a function, concepts of inheritance can allow heterogeneous lists to be created and operated on at runtime, and programming patterns like visitor patterns can be used to write highly extensible code. Let us consider the following function `sub`:

```
fun sub[α] (x:α array, i: int) =
  typecase α of
    int => intsub(x, i)
  | float => floatsub(x, i)
  | ptr(τ) => ptrsub(x, i)
```

Here, the function dynamically dispatches based on the run-time type of the argument `x`. In languages like Java and C++, we implement this logic using dispatch tables; the core concept, however, remains the same. If α can be determined statically, then the optimizer can eliminate the typecase directly to obtain the following function:

```
fun sub[float] (x, 10) = floatsub(x, 10)
```

Introducing types to the run-time, though providing the benefits of dynamism and expressiveness, comes with the following costs. Firstly, maintaining the representation of types at run-time becomes a storage overhead. In cases where we cannot infer precise types statically, an appropriate dispatch mechanism needs to be implemented, leading to more extensive function code. Second, the type information must be propagated through each compilation stage, requiring all passes to be type-preserving. The following Java code represents how this optimization is implemented in modern Java programs (Figure 4.1 shows this process diagrammatically).

```
class A { void foo(); }
class B extends A { void foo(); }
class C extends B { void foo(); }
...
A obj = new B();
obj.foo(); // The call site here can be disambiguated to B.foo
```

4.1.2 Nearly Tag-Free Garbage Collection

Suppose a garbage collector only has partial information about the location of pointers. In that case, it uses conservative techniques like conservative-pointer finding[3] that may end up treating arbitrary bit patterns as pointer addresses. Such approximate approaches often have arbitrary limitations on data representation due to garbage collection. TIL records enough information about types at compilation time so that at any point in the runtime where garbage collection can occur, we have the information about all the pointers. Compared to conservative techniques, this allows the garbage collector to collect all unreachable objects at runtime.

In this approach, the types of stack variables are used to determine pointer variables statically. The garbage collection is said to be "nearly" tag-free because tags now only end up in heap-allocated objects. The layout of each stack frame is modified so that the locations of all pointers can checked by looking at the stack frame. The precision of garbage collection is

further improved by additionally recording information about pointers no longer in us (liveness information).

4.2 Confinement Types

The idea of confinement [21] is simple: we have a Java program, and we want to ensure that an object of type `confined` cannot escape the scope of its package (nor should it be possible to construct an object of this type outside the package). The idea is to declare classes as `confined` and use the type system to enforce these rules. If the program type checks successfully, we can be sure that an object cannot escape the scope of its package. Let us consider the following Java code snippet.

```java
package p;

public class Table {
  private Bucket[] buckets;
  public Object[] get(Object key) { return buckets; }
}

class Bucket {
  Bucket next;
  Object key, val;
}
```

In the above example, the field `buckets` escapes through `Table.get`, even though it is declared `private`. The example shows how an object reference can leak out of a package, and now users can access an object meant to be confined to this package. The `private` keyword in Java cannot be used to enforce the property of confinement statically. Confinement is forced using two sets of constraints.

- *Confinement rules*: applies to the classes defined in the same package as the confined class. These rules ensure that confined types are not exposed to the public **nor widened to nonconfined types**.
  - $C1$: A confined type must not appear in the type of a `public` (or `protected`) field or the return type of a `public` (or `protected`) method.
  - $C2$: A confined type must not be `public`.
  - $C3$: Method invoked on an expression of confined type must either be defined in a confined class or be anonymous.
  - $C4$: Subtypes of a confined type must be confined.
  - $C5$: Confined types can be widened only to other confined types.
  - $C6$: Overriding must preserve anonymity of the methods.
- *Anonymity rules*: applies to methods inherited by the confined classes, potentially library code, and ensures that these methods do not leak a reference to the distinguished variable `this`, which may refer to an object of confined type.
  - $A1$: The `this` reference is used only to select fields and as the receiver in the invocation of other anonymous methods.

This work proposes ConfinedFJ, which extends Featherweight Java. Featherweight Java limits its calculus to five basic operations (object construction, method invocation, field access, casts, and local unstable access). In ConfinedFJ, classes can be declared public or confined, and methods can optionally be declared anonymous. Also, class names are pairs of identifiers bundling a package and class names, just as in Java.

4.2.1 Type Rules

This section describes the type rules to support confinement and anonymity as discussed in the previous section.

Rule 1 : Class C is confined if its definition in the class table starts with the $conf$ prefix.

$$\frac{CT(C) = conf\ class\ C\ \lhd D\{...\}}{conf(C)} \tag{4.1}$$

Rule 2 : Class C is visible to any arbitrary class D if class C is public.

$$\frac{public(C)}{visible(C, D)} \tag{4.2}$$

Rule 3 : Class C is visible to class D if they belong to the same package.

$$\frac{packof(C) = packof(D)}{visible(C, D)} \tag{4.3}$$

Rule 4 (safe-subtyping) : Safe sub-typing is allowed if both classes belong to confined types.

$$\frac{C \prec D\ \ conf(C) \Rightarrow conf(D)}{C \preceq D} \tag{4.4}$$

Rule 5 : The type of a method m defined in class C is given by $mtype$.

$$\frac{mdef(m, C) = D\quad [anon]\ B\ m(\overline{Bx})\ \{return\ e;\}) \in methods(D)}{mtype(m, C) = \overline{B} \to B} \tag{4.5}$$

Rule 6 : A method can be overridden only if it was not previously defined (in a callable scope) or if the new method which is being declared is anonymous then the method it overrides must also be anonymous.

$$\frac{either\ m\ is\ not\ defined\ in\ D\ or\ any\ of\ its\ parents}{override(m, C, D)} \tag{4.6}$$

OR

$$\frac{mtype(M, C) = \overline{C} \to C_0\quad mtype(m, D) = \overline{C} \to C_0(anon(m, D) \Rightarrow anon(m, C))}{override(m, C, D)} \tag{4.7}$$

Rule 7 : A method m defined in class C_0' is anonymous in class C_0 if it is declared using the $anon$ keyword.

$$\frac{mdef(m, C_0) = C_0'\quad anon\ C\ m(\overline{Cx})\{...\} \in methods(C_0')}{anon(m, C_0)} \tag{4.8}$$

Rule 8 : Anonymity constraints enforce the following constraints on the syntax.

- If an expression e is anonymous in Class C then the typecast of e into C' is also anonymous.
- Anonymous expressions $\overline{e}$ in class C when passed to an object constructor of type C' must also return an expression anonymous in class C.
- The assignment of `this` to a variable is disallowed.
- If e is anonymous in class C then $e.f$ is also anonymous in class C.
- If e is anonymous in class C and $\overline{e}$ is anonymous in class C then method call $e.m(\overline{e})$ is also anonymous in class C.
- Field selection using ‘this‘ pointer is always anonymous.

- If a method m is anonymous in class C and expression $\overline{e}$ is anonymous in class C then method call $this.m(\overline{e})$ is anonymous in class C.

4.2.2 Expression Typing Rules

Rule T-VAR : Type of a well-typed variable x is found in Γ.

$$\Gamma \vdash x : \Gamma(x) \tag{4.9}$$

Rule T-FIELD : The fields of a well-typed expression e with fields $\overline{f}$ are also well-typed.

$$\frac{\Gamma \vdash e : C \quad fields(C) = (\overline{C\ f})}{\Gamma \vdash e.f_1 : C_i} \tag{4.10}$$

Rule T-INVK : The resultant type of calling a method (derived from a well-typed expression e of type C_0) m (with arguments $\overline{e}$ of type $\overline{C}$) of type $\overline{D} \to \overline{C}$ (where $\overline{C}$ is a safe-subtype of $\overline{D}$) defined in D_0 such that either C_0 subtypes D_0 or the method is anaonymous in D_0 is C.

$$\frac{\Gamma \vdash e : C_0 \quad \Gamma \vdash \overline{e} : \overline{C} \quad mtype(m, C_0) = \overline{D} \to C \quad \overline{C} \preceq \overline{D} \qquad mdef(m, C_0) = D_0(C_0 \preceq D_0 \vee anon(m, D_0))}{\Gamma \vdash e.m(\overline{e}) : C} \tag{4.11}$$

Rule T-NEW : Arguments $\overline{e}$ of type $\overline{C}$ passed to a *new* constructor of class C which has fields $\overline{f}$ of type $\overline{D}$ (where arguments passed are safe-subtypes of fields $\overline{C} \preceq \overline{D}$) is well-typed in Γ with type C.

$$\frac{fields(C) = (\overline{D\ f}) \quad \Gamma \vdash \overline{e} : \overline{C} \quad \overline{C} \preceq \overline{D}}{\Gamma \vdash new\ C(\overline{e}) : C} \tag{4.12}$$

Rule T-UCAST : If an expression e of type D is being cast to type C then either D is not confined or C is confined too.

$$\frac{\Gamma \vdash e : D \quad conf(D) \Rightarrow conf(C)}{\Gamma \vdash (C)e : C} \tag{4.13}$$

Rule T-METHOD : For a method m (defined in C_0) with body e the following constraints must hold. (i) If m is anonymous then e must also be anonymous (ii) Method body e must be visible to the defining class C_0 (iii) Method body e (with type D) must be a subtype of the method return type C. (iv) (see Rule 6) Method definition is anonymity preserving.

$$\frac{\overline{x} : \overline{C}, this : C_0 \vdash e : D \quad D \preceq C \quad override(m, C_0, D_0) \qquad \overline{x} : \overline{C}, this : C_0 \vdash visible(e, C_0) \quad (anon(m, C_0) \Rightarrow anon(e, C_0))}{[anon]\ C\ m(\overline{C\ m})\ \{\ return\ e;\ \}\ OK\ IN\ C_0 \triangleleft D_0} \tag{4.14}$$

Rule T-CLASS : For a confined class C which extends D the following must hold. (i) If D is confined, then so is C. (ii) If D is visible C. (iii) The rules of T-METHOD are recursively applied and must hold.

Notice that there are no rules restricting types $\overline{C}$ for fields $\overline{f}$. This is because confinement is checked when methods are checked (recursively with their bodies).

$$\frac{fields(D) = (\overline{D\ g}) \quad K = C(\overline{D\ g}, \overline{C\ f})\ \{\ super(\overline{g});\ this.\overline{f} = \overline{f};\ \} \qquad visible(D, C) \quad (conf(D) \Rightarrow conf(C)) \quad \overline{M}\ OK\ IN\ C \triangleleft D}{[conf]\ class\ C \triangleleft D\ \{\ \overline{C\ f};\ K\overline{M}\ \}\ \ OK} \tag{4.15}$$

Rules for static expressions are left out as they are easy to derive in the same way and available in the paper. In essence, it ensures operations such as creation of new objects, method calls, etc recursively (all arguments as well) follow the rules of visibility.

Example: The following program no longer typechecks in the updated typing system, the widening of type `Bucket[]` to type `Object` violates T-METHOD where the body e is not a safe-subtype of method's declared return type $Object[]$.

```
1  package p;
2
3  public class Table {
4    Bucket[] buckets;
5    public Object[] get(Object key) { return buckets; }
6  }
7
8  conf class Bucket {
9    Bucket next;
10   Object key, val;
11 }
```

4.3 Summary

Computing types statically is not only beneficial to catching programming errors but can also help in optimizing dynamic features provided by some programming languages. The first Section 4.1 of this chapter discussed how using typed intermediate languages can help in optimizing programming languages. On the one hand, types were added as a runtime quantity to support the expressive nature of intensional polymorphism 4.1.1 while types were removed from stack variables 4.1.2 to make garbage collection more precise and efficient. The second Section 4.2 showed how types can used to guarantee some safety-related properties of a program. The formal definitions and rules of inference were given for a subset of Java called ConfinedFJ.

Chapter 5

Types In Dynamic Languages

Performing simple static optimizations is futile in dynamically typed languages like R because of prevalent features like lazy evaluation, late binding, and object evolution. We studied two works that look at ways of introducing types to such languages. One common thing across both works is that they rely on runtime instrumentation to gather facts about types.

5.1 A Case for Typed R

A recent work that proposes a type system for The R programming Language is [17]. They use *TypeTracer*, which extracts function signatures by instrumenting calls to functions. Once sufficient information is gathered by executing a large number of functions, the executions are then verified against the observed signature. This is done by a package called `ContractR`, which decorates function bodies with type assertions.

In the proposed type system for R, the authors mainly make use of primitive types, approximating higher-order functions as $any \rightarrow any$ (*any* is the most generic type which can be anything) and adding simple subtyping rules (in R, all the primitive types have a unique representation for invalid values called "NA"; subtyping rules were added to allow subtyping of the following form: "int" subtypes "int + NA" represented as $int[\ ] <: {}^\wedge int[\ ]$ where ${}^\wedge int[\ ]$ is subtype with support for NA values). Figure 5.1 shows supported types in the R Type Language.

The workflow of the approach is intuitive and stems from the learning that optimizing functions across function boundaries is the key to optimizing R. For instance, much previous work shows that the eager calling convention can significantly improve optimizations that the JIT compiler can perform. Improving information maintained at function boundaries is likely to improve performance in such a system. This work found that under this type of system, 80% of the functions are monomorphic or have only one polymorphic argument.

T	::=	**any**	*top type*
	|	**null**	*null type*
	|	**env**	*environment type*
	|	S	*scalar type*
	|	V	*vector type*
	|	$T \mid T$	*union type*
	|	$?\,T$	*nullable type*
	|	$\langle A_1, \ldots A_n \rangle \rightarrow T$	*function type*
	|	**list**$\langle T \rangle$	*list type*
	|	**class**$\langle ID_1, \ldots ID_n \rangle$	*class type*

A	::=	T	*arguments*
	|	...	*dots*
V	::=	S[]	*vector types*
	|	^S[]	*na vector types*
S	::=	**int**	*integer*
	|	**chr**	*character*
	|	**dbl**	*double*
	|	**lgl**	*logical*
	|	**clx**	*complex*
	|	**raw**	*raw*

Figure 5.1: R Type Language

5.2 A Case for Typing Dynamic Objects in Python

In great contrast to the previous work, this [15] work takes a different look at objects. In the case of R, objects are often very loosely defined (meaning they do not necessarily have any well-defined structure). As a consequence, any primitive with a field called `class` is treated as an object of that class. This is why focusing on such aspects of R makes more intuitive sense. This is, however, different in Python; Python code is usually not in such disarray. A lot of primitive and object-oriented code is written in Python. Well-defined server frameworks that handle real-world clients are also written in Python. This means that objects of different shapes and sizes will be encountered when optimizing Python. The work takes a different approach by studying in detail the different types and shapes of objects, how they mutate, and what benefit concepts of flow sensitivity provide. Going back to the example from Chapter 1, consider this slightly modified code.

```
1  class Panel:
2    def __init__(self,p1,p2):
3      self.title = title
4      self.width = width
5      if self.width is not None:
6        self.height = self.width
7
8    def _title(self):
9      if isinstance(self.title, str):
10       return Text.from_markup(self.title)
11     else
12       return self.title.copy()
13
14   def setheight(self,height):
15     self.height = height
16
17   def measure(self):
18     return self.width * self.height
19
20 panel1 = Panel(Text(),42)
21 panel2 = Panel("Example Table",None)
22 panel2.width = 5 # modification
23 panel2.setHeight(42) # extension
```

Constructor Polymorphism : Objects `panel1` and `panel2` are constructed at Line 20 and Line 21 respectively. In both cases, the constructor method is called, and it returns the following types. $typeof(panel1) = \{title : Text,\ width : int,\ height : int\}$ and $typeof(panel2) = \{title : Str,\ width : NoneType\}$.

Object Evolution : Behaviour of `panel1` and `panel2` changes over time. One of three things is possible: (i) extension (refers to the addition of new attributes, see Line 23), (ii) modification (refers to modification of types of existing attributes, see Line 22), and (iii) deletion (refers to deletion of attributes).

5.2.1 Class-Based Types

Class-based types are modeled similarly to traditional class-based systems, where a class contains attributes. If an instance of a class does not have an attribute, it is simply said that the attribute is NULL or undefined; allowing equal treatment of all objects of a class. This, however, leads to some imprecisions in the case of Python. In the previous code snippet, let us consider the function `measure`. We cannot statically determine if the function always succeeds (without throwing an exception) or not. One approach to alleviate the imprecision caused by unionized types is to use local type refinements; see function `title` for an example. The refinement, `isinstance`,*str*, ensures that in the true branch, the type of `self.title` is a `str`.

5.2.2 Object-Based Types

This approach uses object evolution to model their types. The following two abstractions can be used to adjust the precision of such type assignments.

Store Abstraction All possible types that can be observed are unionized to obtain the following relation.

```
1 Γ = {panel1 : Panel@20, panel2 : Panel@21}
2 CT = {
3   Panel@20 : {title : Text, width : int, height : int},
4   Panel@21 : {title : str, width : int | NoneType, height : int |
      abs}
5   }
```

Flow Sensitivity The classtable can be modified as the program flows.

```
1 Γ = {panel1 : Panel@20, panel2 : Panel@21}
2 CT = {..., Panel@21 : {title : str, width : NoneType, height : abs
    }}
3 CT = {.., Panel@21 : {title : str, width : int | NoneType, height
    : abs}}
4 CT = {.., Panel@21 : {title : str, width : int | NoneType, height
    : int | abs}}
```

Observe that in the previous example, we only performed weak updates. This is due to the possibility of an alias existing during runtime; this creates the situation where we cannot again type check `measure` to ensure that it always runs successfully.

5.2.3 Results

This study used the top 50 popular Python projects on Github[1](whose testing framework was *pytest*). The main aim was to study the prevalence of dynamic behaviors in these programs. All events were recorded in a trace file, and results were classified on the basis of constructor polymorphism and object evolution. The following notable results were found.

Constructor Polymorphism. Around 20% of constructors were polymorphic. Most polymorphic constructors have a low degree (less than five), indicating that only a few object types are made. 87% are polymorphic on attribute types, 6% differ on the attributes that exist inside them, and 7% exhibit both of these behaviors. A study of separability based on 0-call site, 2-call site, 4-call site, and argument-type context showed that the resultant type produced by a constructor was heavily tied to input arguments more than 80% of the times this was the case.

Object Evolution. It was found that out of all the runtime objects, around 27% objects and 33% classes expose object evolution. Compared to the deletion of attributes, extension and modification are prevalent. Out of all the objects that evolve, evolution is monotonic in nature (here, we say evolution is monotonic only if the addition of new attributes takes place and types only change to their subtypes).

Effectiveness of Class-Based Types vs Object-Based Types. Though object-based types were found to be much more precise, their implementation was much more complex when compared to class-based types. It is worth noting that by using local refinements, the precision of class-based types was significantly improved without adding a large amount of complexity.

5.3 Summary

This chapter studied the work done in recent years on designing designing impactful type systems for dynamic languages. In it two such studies, one for the R programming language and another for python. The first study on The R Programming Language (see Section 5.1) showed how a

[1]libraries that have special hardware requirements or have special hardware requirements were excluded

simple and effective type system could be implemented to type-check R. This type of system provides contracts between function calls and their expected types; this has the potential to improve the correctness of R programs while improving the precision of JIT feedback. The second study (see Section 5.2) presents a comparative study of how dynamic objects are created and used in the wild. This work presented a comparative study on classifying object dynamism based on constructor based polymorphism and object evolution. They classify the benefits each approach provides while commenting on possible future directions.

Chapter 6

Conclusion and Future Work

Type systems have become invaluable tools for reasoning about programs. Different languages implement different type systems that provide different kinds of guarantees; some languages like C and C++ enforce these rules only statically, while languages like Java ensure type properties at runtime. In dynamically typed languages, determining types of variables at compile time is difficult. Even though languages like TypeScript aim to bring a strong type discipline (with support for features like pattern matching, recursion, and conditional typing) to Javascript, the code still relies on transpilation to plain Javascript (which means statically computed guarantees may not reach the execution environment). Often, in dynamically typed languages, complex structural types themselves hold looser meanings, making static analysis without speculation fruitless.

One interesting idea to reason about objects is based on the concept of "states"[2]. For instance, consider a class that implements the idea of a File. A File object naturally tends to be either open or closed, and depending on some "state" property, certain actions may be valid or invalid. Consider the following Java code snippet:

```
class File {
  int myFD = -1;
  public int  read ()  { ... }
  public void close()  { ... }
  public void open ()  { ... }
}
```

If we reason about the state of the objects; we can determine that the use of `read`/`close` and `open` methods is mutually exclusive. Methods `read` or `close` can only be invoked on an already open file, while method `open` is only valid on a closed file. Thus, state can be bifurcated as follows:

```
state File {
  int myFD = -1;
}
state OpenFile extends File {
  public int read() { ... }
  public void close() [OpenFile>>ClosedFile] { ... }
}
state ClosedFile extends File {
  public void open() [ClosedFile>>OpenFile] { ... }
}
```

Though intuitive, this concept was unpopular among developers and showed little interest in becoming a standard Java language feature. This was because performing such an analysis using typestates requires precise alias information, which is a complex problem in the presence of multiple threads. It may be argued that "typestate" is a natural property of many programs, and developers can, with some effort, add annotations to describe these. However, such a "feature" sometimes becomes an extra effort, which developers actively avoid using. Even the addition of an optional typesystem, like TypeScript, is a challenging task. Many developers believe that

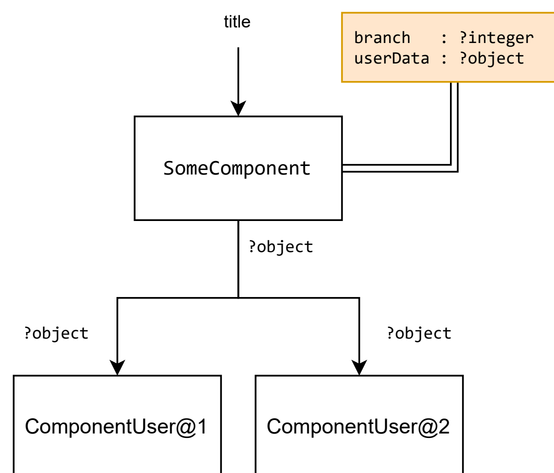
migrating large codebases from clean, untyped Javascript code to verbose TypeScript code simply makes it harder to maintain.

Reasoning about the program in more abstract terms can sometimes be useful, for instance, consider "purity" of a function. If we can write a program analysis that can find all the pure functions in a language like C or Java, we can generate code that can be memoized, computed parallelly, and much more. This is what we call a "desirable property", but sadly, in reality, we hardly come across such cases. On the other hand, if we consider specialized frameworks/libraries of these languages, such as React [13], we find a lot of pure functions. This is because the structure of the React code enables local reasoning, where developers think in terms of components, and the composition of these components makes up a user interface, making writing impure code unnatural.

One novel way to look at creating type systems for dynamic languages like Javascript and R could be to ignore all the lower-level details and rely entirely on the abstraction provided by high-level libraries and frameworks. Let us consider the following React code snippet:

```
1  function SomeComponent(title) {
2    let [branch, setBranch] = useState(1);
3    let [userData, setUserData] = useState(undefined);
4
5    const handleBranchChange = (value) => {
6      if (value == 2) {
7        setBranch(2);
8        setUserData({
9          name: "Meetesh",
10         email: "meeteshmehta@cse.iitb.ac.in"
11       });
12     }
13   }
14
15   return (
16     <div>
17       { branch === 1 && <ComponentUser data={userData} /> }
18       { branch === 2 && <ComponentUser data={userData} /> }
19       <button onClick={handleBranchChange}>change branch to 2</
     button>
20     </div>
21   )
22 }
```

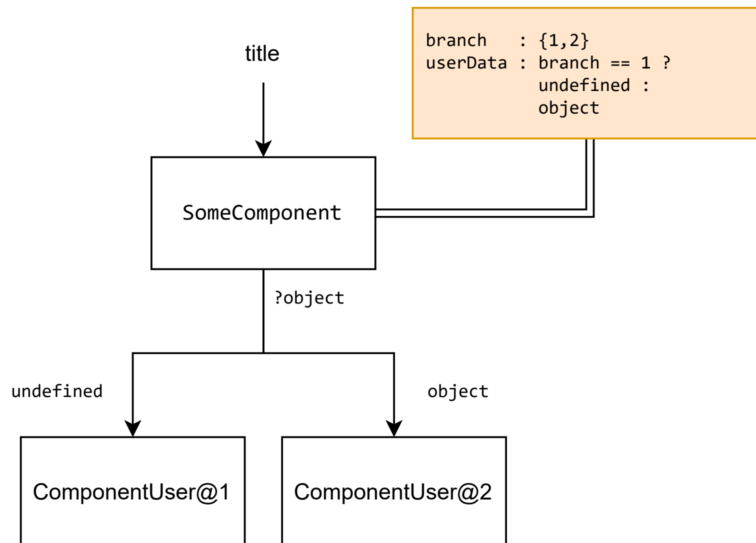
The code defines a component called `SomeComponent` which takes one argument as input. The code can be visualized as follows:



A React component is read as follows:

- The component takes one argument `title`; such values are called props or properties and signify that these are static values. The programmer is not supposed to write/modify these values inside the component. One might imagine it to be **final** fields in the local context.
- `branch` and `userData` are state variables. If any state variable changes, the entire component and its affected children are re-rendered.
  - *Finding redundant re-renderings could be done using a compiler-based static analysis. Recent (unreleased) work by Meta [11] (creators and maintainers of React) shows their work on a compiler that uses static analysis to memoize values of components.*
- On the first render of component `SomeComponent` initializes the state variables (`branch` and `userData`) with initial values (initial value for `branch` is `1` and `undefined` for `userdate`). This leads to the first `ComponentUser` being rendered (both `ComponentUser` are mutually exclusive and dependent on the value of `branch`).
  - *Notice how react also makes it easy to figure when and how these state variables might change, i.e., the method* `setBranch` *and setUserData can be used to trivially determine what triggers a state update.*
  - *This obviously makes a case for aliases to exist and leave function boundaries, but we can argue that does not happen often (as this way of programming makes it more natural to reason locally).*
- When the button *change branch to 2* is pressed, it calls the method `handleBranchChange`.
  - Inside the `handleBranchChange` method, the state variable for `branch` is updated to 2, and at the same time `userData` is updated to a new value.
  - *What do we mean by "at the same time"? This needs to be modeled precisely during analysis, but for the sake of simplicity, we just assume state updates happen in a sequence and are always ready before their next use.*
  - The component is re-rendered when `branch` value is re-rendered.
  - *Notice that only values have changed here; updating the virtual DOM is expensive. Can we do static analysis, bypass this re-rendering step, and update the DOM directly? It is a higher-order version of constant propagation.*

Often, not all children's components are rendered in React programs; there is usually a dependency on some variable (it might be a property or a state variable or any value derived using these). This is a significant part of how rendering logic is represented in React; we must narrow our focus on them. Ideally, we want to determine their universe or part of the universe that can allow us to disambiguate mutually exclusive conditions. Let us consider the following figure:



Here, we see that the state variable `branch` can be either `1` or `2` and nothing else. We use this information to conditionally type all variables as being dependent on the type of `branch`. In

the above figure, we find that we can narrow down the types of values passed to `SomeComponent` based on the type of `branch`. If the value of `branch` is `1`, `undefined` is passed to `ComponentUser`. When the value of `branch` is `2`, `object` of the form { `name: String, email: String` } is passed to `ComponentUser`.

In specialized programming languages, domain-specific knowledge can guide high-level compiler optimizations. For instance, in a programming language framework like React, programs are often pure and structured as trees. However, optimizing React poses challenges due to the lack of a suitable intermediate language that could facilitate these optimizations effectively. This intermediate language would need to incorporate concepts such as state variables, batched state updates, asynchronous callbacks, and component tree updates. Our future work will focus on exploring the design of such a language, understanding its primitives, and extending these concepts to other similar systems.

References

[1] Agda. Agda, 2007. Retrieved May 8, 2024.

[2] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, page 1015–1022, New York, NY, USA, 2009. Association for Computing Machinery.

[3] Hans-Juergen Boehm. Space efficient conservative garbage collection. *SIGPLAN Not.*, 28(6):197–206, jun 1993.

[4] Olivier Flückiger, Guido Chari, Jan Ječmen, Ming-Ho Yee, Jakob Hain, and Jan Vitek. R melts brains: An ir for first-class environments and lazy effectful arguments. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2019, page 55–66, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Olivier Flückiger, Guido Chari, Ming-Ho Yee, Jan Ječmen, Jakob Hain, and Jan Vitek. Contextual dispatch for function specialization. *Proc. ACM Program. Lang.*, 4(OOPSLA), 2020.

[6] Olivier Flückiger, Jan Ječmen, Sebastián Krynski, and Jan Vitek. Deoptless: Speculation with dispatched on-stack replacement and specialized continuations. In *Conference on Programming Language Design and Implementation (PLDI)*, 2022.

[7] Urs Holzle. *Adaptive optimization for self: Reconciling high performance with exploratory programming*. PhD thesis, 1994. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2023-02-23.

[8] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. volume 27, pages 32–43, 07 1992.

[9] JVM. Jvm, 2007. Retrieved May 8, 2024.

[10] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., USA, 2002.

[11] meta. meta, 2004. Retrieved May 8, 2024.

[12] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[13] React. React, 2013. Retrieved May 8, 2024.

[14] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, jan 1965.

[15] Ke Sun, Sheng Chen, Meng Wang, and Dan Hao. What types are needed for typing dynamic objects? a python-based empirical study. In Chung-Kil Hur, editor, *Programming Languages and Systems*, pages 24–45, Singapore, 2023. Springer Nature Singapore.

[16] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. Til: a type-directed optimizing compiler for ml. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, page 181–192, New York, NY, USA, 1996. Association for Computing Machinery.

[17] Alexi Turcotte, Aviral Goel, Filip Křikava, and Jan Vitek. Designing types for r, empirically. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[18] TypeScript. Typescript, 2012. Retrieved May 8, 2024.

[19] V8. V8, 2008. Retrieved May 8, 2024.

[20] Stephen Weeks. Whole-program compilation in mlton. In *Proceedings of the 2006 Workshop on ML*, ML '06, page 1, New York, NY, USA, 2006. Association for Computing Machinery.

[21] TIAN ZHAO, JENS PALSBERG, and JAN VITEK. Type-based confinement. *Journal of Functional Programming*, 16(1):83–128, 2006.

[22] Ř. Ř, 2016. Retrieved May 8, 2024.