Y790 - Controversy in Programming Languages to a Layperson

Ambrose Bonnaire-Sergeant (0003410123)

February 9, 2016

1 Micro vs. Macro Gradual Typing

Consider the following function **add** that adds two numbers, written with the Clojure programming language.

```
(defn add [x y]
  (+ x y))
```

We use our new definition to add 40 and 2, returning 42.

```
(add 40 2)
;=> 42
```

Adding **40** to **nil** does not make sense, since **nil** is not a number. But Clojure runs the program anyway, resulting in a cryptic error message.

```
(add 40 nil)
; NullPointerException
```

Using a different approach, we can improve this error message. A type system is a preliminary integrity check before running a program, precisely detecting errors like adding to **nil**. Without a type system, error messages are poor. Languages without type systems are known as "untyped". Clojure is "untyped" and thus lacks a type system.

In contrast, here is a similar interaction in a "typed" language which uses type system, called Typed Clojure.

```
(ann add-typed [Number Number -> Number])
(defn add-typed [x y]
  (+ x y))

(add-typed 40 nil)
; Type Error:
:   Given nil when expecting a Number in the
;   second argument of ‘add-typed‘.
```

There are two differences from Clojure. The first line we provide a type annotation on **add-typed** saying it can only be a function that takes two **Number**s and returns a **Number**. More importantly, unlike the Clojure example, the error is clear and precise.

As demonstrated, type systems give improved error messages. Errors are often more descriptive. In the example, the type system expects a **Number**, but instead finds a value of type **nil**. Furthermore, errors are also more precise. In our example, the type system reports the second argument of **add-typed** as the cause of the problem, instead of reporting a symptom like a **NullPointerException** in untyped languages.

Preserving these improved error messages of typed code is then a crucial part of gradual typing, an approach to integrating typed and untyped programs. The central idea of gradual typing is safe interoperation. What is "safe" interoperation? Intuitively, typed languages rely on stronger invariants via the type system; for example Typed Clojure relies on the fact that the type system checks

```
(add-typed 40 nil)
```

to guarantee a better error message. However, running this code in Clojure (an untyped language) produces the same **NullPointerException** as if the definition of **add-typed** was written in an untyped language, that is, without a type system. But the type system verified the definition of **add-typed**! The key to safe interoperation is preventing this situation: safe interoperation protects the invariants of typed code from untyped code.

The "language boundary" extends this notion of safe interoperation. The languages boundary mediates interactions between typed and untyped programs. It enforces invariants as values flow out of one language into another. Usually it wraps values in immediate or delayed runtime checks, also called "contracts". In order to facilitate safe interoperation, the mediator chooses appropriate contracts based on the invariants of the typed language. Examples of contracts range from immediate, like a number check if a value of type **Number** is expected, to delayed, like a function contract that checks each input and corresponding output conforms to particular types (it is impossible to check function immediately without

the code that generates it, which is compiled away).