Y790 - Controversy in Programming Languages to a Layperson

Ambrose Bonnaire-Sergeant (0003410123)

February 3, 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 can 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
```

Clojure is "untyped" and thus lacks a type system. A type system functions as a preliminary integrity check before running a program, precisely detecting errors like adding to **nil**. Instead, untyped programs are only checked as they are being run, by which time much context of the original program has been lost. This is why untyped languages often report the symptom of an error, not its cause.

In contrast, observe a comparable 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 provides a type annotation on **add-typed** saying it can only be a function that takes two **Number**s and returns a **Number**. Second, the error generated is much improved — as well as being more descriptive and precise, it occurs *before* the erroneous expression is run.

Improved error messages are a fundamental advantage of using a type system. Errors are often more descriptive. In the example, the type system says it expected a **Number**, but actually found a value of type **nil**. Errors are also more precise. The example demonstrates this by reporting the second argument of **add-typed** as the cause of the problem, instead of reporting a symptom like a **NullPointerException** in untyped languages.

Gradual typing is 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

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

is checked by the type system 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 definition of **add-typed** was verified by a type system! Preventing this situation is the key to safe interoperation: safe interoperation protects the invariants of typed code from untyped code.

Extending this notion of interoperation is the "language boundary". This concept refers to the mediator between the typed and untyped portions of our programs. The language boundary is entrusted with enforcing invariants as values flowing 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, contracts are chosen to reflect the expectations 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).

"From scripts to programs", a slogan for gradual typing, highlights another key motivation: the translation of untyped "scripts" into typed "programs". Programs are often first written in untyped languages, where they are easy to prototype and flexible to change. These characteristics become less important further in the development cycle, replaced by correctness and documentation, innate features of typed programs. The desire to rewrite the untyped program in a typed language is counterbalanced by the risk of introducing bugs in the translation process. Gradually typed languages are often based on other untyped languages, such that the translation from untyped to typed require only minimal changes, mitigating the risk of new bugs—notice the minimal difference between **add** and **add-typed**.

Even within the world of gradual typing there are two fundamental approaches. "Micro" gradual typing operates on the level of functions. It supports translation at the scale of individual functions to be typed, or even sub-parts of functions. Alternatively, "macro" gradual typing operates on the level of modules, which roughly often correspond to source files. It supports translation only at the scale of entire modules; the role of the language boundary is then to strictly control module imports and exports to maintain safe interoperation.

In practice, these two approaches share similar performance drawbacks. Crossing the language boundary is expensive as it incurs a runtime check. If the program contains a tight loop between two mutually referential functions which live on different sides of the language boundary, every iteration must pay this cost. In this respect, performance problems are likely to be more severe in micro gradual typing, since the language boundaries are finer grained and thus more numerous. That said, macro gradual typing is not immune to this issue, and pays a cost when values imported from other modules must be checked repeatedly in the body of a loop.

The micro and macro gradual typing approaches are quite similar conceptually, and not particularly adversarial. It is straightforward to view macro gradual typing as a specific instance of micro gradual typing. Researchers of either style often collaborate in research papers, and some even study both. Academic ancestry often plays a role in choosing the appropriate style of gradual typing. Reticulated Python is a gradual typing system for Python using micro gradual typing—the advisor on that project invented micro gradual typing. Typed Clojure is a gradual typing system for Clojure using macro gradual typing—it is based on the macro gradual typing approach invented for Typed Racket, the dissertation subject of Typed Clojure's advisor. Both systems are developed at Indiana University, within the same gradual typing group.

What is the future of gradual typing? Whatever it holds, researchers on both sides of the micro and macro approaches will probably continue to collaborate to solve problems in common. The performance

of gradual typing systems in practice is the next big obstacle stopping gradual typing from reaching the mainstream, so it seems a natural progression for the upcoming research in gradual typing. It is likely both approaches will be continue to be used for new gradually typed languages as long researchers in gradual typing break new ground in their respective areas.