A Practical Optional Type System for Clojure

Ambrose Bonnaire-Sergeant <abonnairesergeant@gmail.com> (20350292)
Supervised by Rowan Davies <rowan@csse.uwa.edu.au>

The University of Western Australia

Background

Static and Dynamic Types

Traditionally there has been a clear distinction between programming languages that perform type checking at compile-time, and those that do not. A recent trend is to aim to combine the advantages of both kinds of languages by adding optional static type systems to languages without static type checking.

Clojure

Clojure is a recent and increasingly popular dynamically typed programming language. Designed to be pragmatic, it is a dialect of Lisp supporting functional programming and immutability by default, while being hosted on, and providing direct interoperability with, popular platforms.

Contributions

We take the lessons learnt from several projects to design *Typed Clojure*, an optional type system for Clojure running on the Java Virtual Machine. The main contributions are:

- We develop a prototype type checker based on Typed Racket [1], which is able to type check many Clojure idioms
- We describe a novel use of *occurrence typing* for eliminating erroneous access to Java's *null* pointer
- We show how a combination of occurrence typing and intersection types allows accurate typing checking of the most common usages of Clojure's *sequence abstraction*
- We capture the Clojure idiom of using maps with known keyword fields by using *heterogeneous map types*

Treatment of *null*

Clojure represents Java's *null* as the value `nil`, which has the singleton type `nil` in Typed Clojure. This means, unlike Java's static type system, *null* and reference types are separated as static types. The programmer can then annotate Java methods and fields, specifying where *null* is allowed.

Listing 1: Annotating Java methods that never return *null*

```
(non-nil-return java.lang.Object/getClass :all)
(non-nil-return java.io.File/getName :all)
```

Sequence Abstraction

Typed Clojure types can express the most common usages of Clojure's sequence abstraction, which is a set of functions providing a common interface to collections. These functions include:

- `seq`, which takes a collection (or `nil`) and returns
  - a true value if the argument is non-empty
  - a false value if the argument is empty (or `nil`)
- `first`, which takes a collection (or `nil`) and returns
  - the first item of the argument if the argument is non-empty
  - `nil` if the argument is empty (or `nil`)

By encoding these invariants in the respective types, we can type check common combinations of these functions.

Listing 2: Assuming `coll` is a collection of numbers currently in local scope, we can infer that this expression always returns a number

```
(if (seq coll)
  (first coll)
  0)
```

Heterogeneous Maps

Heterogeneous map types that record the *presence* of keyword keys are sufficient to type check the most common usages of the Clojure idiom of using plain maps with known keyword keys.

Map Literals `{:a 1 :b 2}` creates a map value with two entries, and is of the heterogeneous map type `'{:a Number :b Number}`

Associating keyword keys `(assoc {:a 1 :b 2} :c 3)` extends the first argument with a new entry with key `:c`, and is of type `'{:a Number :b Number :c Number}`

Dissociating keyword keys `(dissoc {:a 1 :b 2} :a)` dissociates from the first argument the entry with key `:a`, and is of type `'{:b Number}`

Keyword lookup `(get {:a 1 :b 2} :a)` returns the value corresponding to the key `:a` in the first argument, and is of type `Number`

Source

Typed Clojure is available at:
`https://github.com/frenchy64/typed-clojure`

Experiments

Monads

Almost all of a Clojure library for monadic programming (*algo.monads*) was successfully ported to Typed Clojure, including most monad, monad function, and monad transformer definitions. This experiment warranted an extension for type functions (functions at the type level) with variance.

Java Interoperability

A function relying heavily on Java interoperability was ported successfully. This function chained several Java calls together, and Typed Clojure was able to express and check invariants related to occurrences of *null* and primitive arrays.

Future Work

There is significant future work.

- Implement a blame calculus to improve error messages when using untyped code
- Support type checking multimethod definitions
- Support Clojure records, which are datatypes with known keyword keys
- Port Typed Clojure to other Clojure implementations

Conclusion

This work has demonstrated that it is both practical and useful to design and implement an optional type system for Clojure that preserves current Clojure style.

References

[1] Tobin-Hochstadt, S. Typed Scheme: From Scripts to Programs PhD Dissertation, Northeastern University, 2010

Bachelor of Computer Science Honours (Semester 1, 2012)