A Practical Optional Type System for Clojure

Ambrose Bonnaire-Sergeant

Statically typed vs. Dynamically typed

- Traditional distinction
- Dynamically typed
  - eg. Javascript, Ruby, Python
  - No type checking at compile time
- Statically typed
  - eg. Java, C, Haskell
  - Performs type checking at compile time

Statically typed

- +ve
  - Helps avoid common errors
- -ve
  - Type checking is mandatory

Dynamically typed

- +ve
  - More flexible idioms are possible
- -ve
  - Fewer errors caught at compile time

Clojure

- Released October 2007 by Hickey
- Dynamically typed
- Emphasis on immutability, functional programming
- Implementations for
  - Java Virtual Machine (Clojure)
  - Common Language Runtime (ClojureCLR)
  - Javascript virtual machines (ClojureScript)

Motivation

- Higher-order programming styles
  - Static type systems can help verify this code as correct
- No similar tools in Clojure ecosystem

Typed Racket

- Typed sister language of Racket
- Understands many Racket idioms
- Emphasis on module-by-module porting of untyped code
- Requires fundamental differences to standard static type systems (like Java's, C's) to support existing idioms
  - Needs to express more precise invariants
  - Union types, intersection types, *occurrence typing*

Classifying optional type systems

- Different programmers have different concepts of *types* and *type systems*
- Reynolds (2002) and Pfenning (2008) distinguish between *intrinsic* and *extrinsic* type systems

Intrinsic type systems

- Traditional type systems
- Types determined at *compile time* define a runtime semantics
  - ie. programs must pass the type checker to be meaningful
- eg. Java, C, Haskell, ML

Extrinsic type systems

- Runtime semantics do not depend on static type checker
- Can be considered an “extra” layer of checking
- eg. SML CIDRE, by Davies (2005)
  - Adds “refinement-types” to Standard ML
- Optional type systems are *extrinsic* type systems

Typed Clojure

- An optional type system for Clojure
  - Write Clojure code as normal
  - Add types when helpful, while preserving style
- Largely based on lessons learnt from Typed Racket
  - With important additions supporting Clojure idioms
- Intended for everyday use by Clojure programmers

Example - Maybe Monad

```
(defmonad maybe-m
  [m-zero   nil
   m-result (fn m-result-maybe [v] v)
   m-bind   (fn m-bind-maybe [mv f]
              (if (nil? mv)
                nil
                (f mv)))
   ...
   ])
```

Example - Maybe Monad

```
(defmonad maybe-m
  [m-zero   nil
   m-result (fn m-result-maybe [v] v)
   m-bind   (fn m-bind-maybe [mv f]
              (if (nil? mv)
                nil
                (f mv)))
   ...
   ])
```

Example - Maybe Monad

```
(defmonad maybe-m
  [m-zero   nil
   m-result (fn m-result-maybe [v] v)
   m-bind   (fn m-bind-maybe [mv f]
              (if (nil? mv)
                nil
                (f mv)))
   ...
   ])
```

Occurrence typing

```
(ann clojure.core/nil? [Any -> boolean
                        :filters {:then (is nil 0)
                                  :else (! nil 0)}])
```

Monads

```
(ann maybe-m (MonadPlusZero
               (TFn [[x :variance :covariant]]
                 (U nil x))))
(defmonad maybe-m
  [m-zero   nil
   m-result (ann-form
              (fn m-result-maybe [v] v)
              (All [x]
                [x -> (U nil x)]))
   m-bind   (ann-form
              (fn m-bind-maybe [mv f]
                (if (nil? mv) nil (f mv)))
              (All [x y]
                [(U nil x) [x -> (U nil y)] -> (U nil y)]))
   ...
   ])
```

Contributions

- Prototype type checker for Clojure based on Typed Racket
- Novel use of occurrence typing for Java interoperability
    - *null* is directly expressible as a static type, made possible by occurrence typing
- Show how a combination of intersection types and occurrence typing can type check common usages of Clojure's *sequence abstraction*
- Accommodate Clojure's idiomatic usage of hash-maps with *heterogeneous map types*
- Identify the main future issues to typing Clojure code

Implementation

- Majority of the effort was spent programming the type checker
- Approx. 9,300 lines implementation
  - Bidirectional checking ~2000 lines
  - Occurrence typing ~2000 lines
    - Typed Racket port
  - Variable-arity polymorphism ~1000 lines
    - Most complicated part (mostly involved porting from Typed Racket)
  - Annotates some core Clojure libraries

Experiments

- Monad library
  - Almost all monad, monad transformer, monad function definitions ported
  - Motivated an extension with type functions (functions at the type level)
- Java Interoperability
  - Ported a function that uses reflection
  - Complicated invariants with respect to *null*

Future work

- Blame calculus
  - Improves errors when interfacing with untyped code
- Multimethods
- Prove soundness of the type system
  - eg. types are preserved during evaluation
  - Designed to be sound, not formally proven yet
  - Likely using standard techniques from programming language research

Conclusion

- Interest exists
  - Talk will be given at Clojure Conj 2012
  - Google Summer of Code 2012 project for Clojure
- Appears to be both practical and useful

Demo