tools.analyzer

Nicola Mometto (Bronsa)

brobronsa@gmail.com
@Bronsa_
https://github.com/Bronsa

The Clojure Compilation Pipeline

CinC

tools.reader

- 3 modules:
  - clojure reader
  - edn reader
  - text readers (**string-reader**, **indexing-reader**, **source-logging-reader** ..)
- Takes code as string, returns data
- Passes code as data to the analyzer

tools.analyzer(.jvm)

- Takes clojure form from the reader
- Macroexpands the form
- Parses special forms and primitives, transforms into AST
- Does additional analysis over the AST
- Passes the AST to the emitter

tools.emitter(.jvm)

- Takes AST from the analyzer
- (Optionally) Transforms it into an internal representation
- Outputs code
- (Optionally) Loads the code in memory

tools.analyzer Goals

- Dialect and platform agnostic (common AST format)
- Data oriented
- Multi pass

Dialect/Platform Agnostic

- Split libraries: tools.analyzer, tools.analyzer.[*]
- Generic nodes (non specialised interop node types)
- Pluggable entry points:
  - **macroexpand-1**
  - **parse**
  - **create-var**, **var?**
- Namespace system abstracted away (clojure.tools.analyzer.env)
- Open dispatch via multimethods

Data Oriented

- AST as maps (à la clojurescript)
  - Node type under **:op**
  - Lexical environment threaded through analysis, stored under **:env**
  - Original form under **:form**
- Easily transformable
- Generic traversal (**:children** keys vector)

Generic Traversal (naïve approach)

```
{:op :foo
 :child1 {:op :bar ..}
 :child2 [{:op :bar ..} ..]
 :some-info {..}
 ..}
```

Generic Traversal (naïve approach)

```
{:op :foo
 :child1 {:op :bar ..}
 :child2 [{:op :bar ..} ..]
 :some-info {..}
 ..}
```

- Easy update
- Easy lookup
- Complex traversal (no explicit childrens)
- No nodes traversal ordering

Generic Traversal (nested :children map)

```
{:op :foo
 :some-info {..}
 :children {:child1 {:op :bar ..} ..}
 ..}
```

Generic Traversal (nested :children map)

```
{:op :foo
 :some-info {..}
 :children {:child1 {:op :bar ..} ..}
 ..}
```

- Easy update
- Easy lookup
- Easy traversal
- No nodes traversal ordering

Generic Traversal (nested :children vector)

```
{:op :foo
 :some-info {..}
 :children [{:op :bar ..} ..]
 ..}
```

Generic Traversal (nested :children vector)

```
{:op :foo
 :some-info {..}
 :children [{:op :bar ..} ..]
 ..}
```

- Complex lookup
- Complex update
- Easy traversal
- Explicit nodes traversal ordering

Generic Traversal (:children nodes vector)

```
{:op :foo
 :child1 {:op :bar ..}
 :some-info {..}
 :children [{:op :bar ..} ..]
 ..}
```

Generic Traversal (:children nodes vector)

```
{:op :foo
 :child1 {:op :bar ..}
 :some-info {..}
 :children [{:op :bar ..} ..]
 ..}
```

- Easy lookup
- Complex update
- Easy traversal
- Implicit nodes traversal ordering

Generic Traversal (:children keys vector)

```
{:op :foo
 :child1 {:op :bar ..}
 :some-info {..}
 :children [:child1 ..]
 ..}
```

Generic Traversal (:children keys vector)

```
{:op :foo
 :child1 {:op :bar ..}
 :some-info {..}
 :children [:child1 ..]
 ..}
```

- Easy lookup
- Easy update
- Easy traversal (indirect)
- Explicit nodes traversal ordering

Generic traversal API

clojure.tools.analyzer.ast

```
=>(def AST
    {:op :foo
     :child1 {:op :op1}
     :child2 [{:op :op2} {:op :op3}]
     :children [:child1 :child2]})

=>(ast/children AST)
;; [{:op :op1}{:op :op2} {:op :op3}]

=>(ast/prewalk AST
    (fn [node](println (:op node)) node))
;; :foo, :op1, :op2, :op3

=>(ast/postwalk AST
    (fn [node](println (:op node)) node))
;; :op1, :op2, :op3, :foo
```

Generic traversal API

clojure.tools.analyzer.ast

```
=>(def AST
    {:op :foo
     :child1 {:op :op1}
     :child2 [{:op :op2} {:op :op3}]
     :children [:child1 :child2]}

=>(ast/postwalk AST
    (fn [node](println (:op node)) node)
    :reversed)
;; :op3, :op2, :op1, :foo
```

Generic traversal API

clojure.tools.analyzer.ast

- See also:
  - **ast**/**walk**
  - **ast**/**update-children**
  - **ast**/**nodes**
- Early traversal termination via **reduced**

Multi Pass

- Analyzer does the bare minimum, just parsing code into AST
- All the real analysis is done on the AST via small specialised phases
- Phases are configurable and optional
- Phases are combined into passes and run over the AST

Phases

- Just multimethods dispatching on **:op**!
- Take an AST node and return annotated AST node
- Might be combined with other phases or require a full pass over the AST
- >30 phases included

Combining/Scheduling Phases

clojure.tools.analyzer.passes/schedule

- Phases declare required AST traversal order
- Phases declare dependencies/affected phases
- Scheduler resolves dependencies, combines phases with congruent traversal order into passes
- Scheduler optimizes pass ordering and combines them into a single function

tools.analyzer.jvm

- API:
  - **jvm/analyze** (avoid)
  - **jvm/analyze+eval** (prefer)
  - **jvm/analyze-ns**
  - **jvm/macroexpand-all**
- Config:
  - **jvm/default-passes**
  - **jvm/run-passes**
  - **jvm/default-passes-opts**
  - **jvm/empty-env**

https://clojure.github.io/tools.analyzer.jvm/spec/quickref.html

Real World Usage

- core.async
- core.typed
- Eastwood
- clj-refactor

AST querying with Datalog (Datomic/Datascript)

```
=>(require
    '[clojure.tools.analyzer.ast.query :refer [q])

=>(q '[:find ?docstring
       :where
       [?def :op :def]
       [?def :init ?m]
       [?method :body ?body]
       [?body :statements ?statement]
       [?statement :val ?docstring]
       [?statement :type :string]
       [?statement :idx 0]]
     [(ast (defn x [] "misplaced docstring" 1))])
;; #{["misplaced docstring"]}
```

Future Work (maybe)

- Rewrite tools.analyzer.jvm method matcher
- Bring tools.analyzer.js up to date with clojurescript
- Faster tools.emitter.jvm
- AOT support in tools.emitter.jvm
- More accessible documentation around available phases
- Optional type-hint enforcement
- Contributors?

Special Thanks

- GSoC
- Ambrose Bonnaire-Sergeant
- Cognitect (Timothy Baldridge, Alex Miller)
- Andy Fingerhut

Questions?