Analiza statyczna języka Ruby

(Static analysis for Ruby language)

Rafał Łasocha

Praca licencjacka

Promotor: prof. Witold Charatonik

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

20 maja 2018

Streszczenie

. . .

. . .

Spis treści

Rozdział 1.

Wprowadzenie

Współcześnie powstaje wiele narzędzi ułatwiających i przyśpieszających pracę programistów. W szczególności dla wielu języków programowania istnieją zintegrowane środowiska programistyczne (IDE), które próbują dostarczyć jak najwięcej takich narzędzi w jednym spójnym środowisku. Jednymi z najbardziej podstawowych funkcjonalności takich środowisk są:

- autouzupełnianie pełnej nazwy aktualnie pisanej stałej, zmiennej lub nazwy funkcji
- skok do definicji (stałej, funkcji lub zmiennej)
- informacja o typie zmiennej
- refactoring - zmiana nazwy klasy (globalnie, w całym projekcie), wydzielenie metody, zmiennej, etc.

Niniejsza praca przedstawia jak można wykorzystać analizę statyczną plików źródłowych, aby dostarczyć powyższe funkcjonalności w języku Ruby, który z racji swojej dynamicznej natury stawia pewne problemy mało istotne w językach statycznie typowanych.

1.1. Założenia i cele

1.1.1. Utrudnienia w językach słabo typowanych

Zintegrowane środowiska nie są niczym nowym i są popularne wśród programistów od wielu lat. Jednakże, implementacja narzędzi bazujących na analizie kodu źródłowego zależy od niektórych cech języków programowania. W językach silnie typowanych jest to znacznie prostsze, ponieważ gdy wiemy już jakiego typu jest

zmienna - lista dostępnych nazw funkcji jest łatwa do obliczenia (autouzupełnianie). Podobnie, języki silnie typowane często mają *static dispatch*, więc w momencie kompilacji wiemy gdzie jest kod źródłowy, który zostanie użyty w danym wywołaniu funkcji (skok do definicji). Oczywiście, w każdym języku możemy używać konstrukcji (np. w C, wywołanie funkcji która jest przechowana we wskaźniku), które to zadanie znacznie utrudniają, ale nie używa się ich zbyt często. W językach dynamicznie typowanych (takich jak Ruby, Python), gdzie język nie dostarcza kompilatora, który dostarczałby informacji o kodzie źródłowym, informacje te musimy sobie znaleźć sami.

1.1.2. Utrudnienia w Ruby

W języku Ruby, o którym jest ta praca, statyczne wywnioskowanie definicji jest szczególnie ciężkie z dwóch powodów:

- wielu różnych dostępnych konstrukcji języka, takich jak wielokrotne dziedziczenie, klasy singletonowe, *prepend* (przypis: nie wiem jak ten mechanizm się nazywa w nomenklaturze języków programowania), metaprogramowanie, które pozwalają na tworzenie skomplikowanych obiektów i hierarchii klas
- społeczności rubiego, która w udostępnianych bibliotekach oraz projektach chętnie korzysta z tych skomplikowanych konstrukcji oraz metaprogramowania. Z jednej strony, duża ekspresja języka pozwala na szybkie wytwarzanie oprogramowania, z drugiej strony, analiza statyczna kodu, a nawet analiza kodu przez programistę jest trudniejsza (problemy podobne, z którymi zmagał się Smalltalk)

1.1.3. Aktualny stan

Do tworzenia oprogramowania w Ruby, programiści najczęściej wykorzystują Rubymine, Vim, Emacs, Atom (przypis: strzał na razie, ale nic istotnego to nie zmieni). Rubymine jest jedynym IDE w tym rankingu, pozostałe narzędzia to tylko edytory tekstu, wspierane wtyczkami. Rubymine dostarcza funkcjonalności wspomnianych we wstępie, jednak jest to narzędzie płatne i zamknięte. Edytory tekstu same w sobie nie posiadają tych funkcjonalności, ale istnieją wtyczki, które w jakiejś części te funkcjonalności dostarczają. Jak to bywa z analizą statyczną, efektywność tych narzędzi to pewne spektrum, jedne narzędzia dostarczają lepsze autouzupełnianie, a inne gorsze, nie jest to binarne i jakość tych funkcjonalności ma duże znaczenie dla programisty. Najpopularniejsze wtyczki to:

- CTags - zbiera symbole z całego projektu i ich lokalizacje, nie radzi sobie w żaden sposób z metaprogramowaniem i nie potrafi dostarczyć tych funkcjonalności zależnie od kontekstu (np. skok do definicji metody zależy od miejsca, w

którym ją się wywołuje)

- Robe - dostarcza informacje zależne od kontekstu, ale wykorzystuje bardzo proste heurystyki do zawężenia wyników
- Solargraph - dostarcza podobne funkcjonalności do tych wymienionych we wstępie (oprócz *refactoringu*), stosuje do tego analizę statyczną oraz wykorzystuje inne źródła (takie jak dokumentacja YARD)
- RGL - odmiennie od pozostałych narzędzi, dostarcza system typów i inferencję typów do ruby. Wymaga jednak adnotacji klas i metod, tj. inferencja typów zachodzi tylko w "otypowanych" przez programistę metodach.

1.1.4. Założenia i cele

Celem pracy jest napisanie wtyczki działającej w zwykłych edytorach tekstu, która będzie oferować narzędzia wspierające pracę programisty. Nie powinna wymagać od programisty zaawansowanej konfiguracji oraz żadnej pracy ze strony programisty (w szczególności pisania adnotacji typów). Powinna dostarczyć jak najbardziej trafne informacje, głównie wykorzystując analizę statyczną, ale również inne źródła, jeśli jest to możliwe. Informacje dostarczane przez wtyczkę niekoniecznie muszą być bezpieczne (w rozumieniu bezpiecznej analizy statycznej), bo jest to tylko wsparcie programisty, pozwalające sprawniej mu się poruszać po kodzie i go pisać. Powinna brać pod uwagę praktyczne problemy wynikające nie tylko z technicznych zawiłości języka, ale również praktycznych. W szczególności, powinna jak najlepiej sobie radzić z metaprogramowaniem, które jest używane przez społeczność. Ponadto, powinna brać pod uwagę że kod źródłowy Ruby jest napisany w C (a więc, biblioteki standardowej nie możemy "przeanalizować") oraz w miarę możliwości fakt, że istnieją biblioteki korzystające z FFI (*Foreign Function Interface*), a więc ich kod źródłowy jest napisany w innym języku (z reguły C).

Rozdział 2.

Rozwiązanie

2.1. Architektura

Architektura wtyczki jest dwuczęściowa:

1. Protokół komunikacji z edytorem tekstu
2. Serwer, który dostarcza funkcjonalności, niezależny od edytora

2.1.1. Protokół komunikacji z edytorem tekstu

We wtyczce wykorzystywany jest LSP (*Language Server Protocol*). Jest to protokół standaryzujący komunikację pomiędzy edytorami tekstu, a serwerami dostarczającymi informacji o projekcie. Dzięki temu, chcąc napisać serwer, który potrafi komunikować się z n edytorami tekstu, wystarczy zaimplementować w tym serwerze protokół LSP, który wspiera edytor tekstu (również często przez oddzielną wtyczkę). Ta część jest czysto inżynieryjna, więc w dalszej części pracy skupię się tylko na algorytmach wykorzystywanych w serwerze.

2.2. Proces pozyskiwania informacji (?)

2.2.1. Funkcjonalności

Przeanalizujmy jeszcze raz funkcjonalności, które chcielibyśmy uwzględnić w naszym serwerze. Pokażemy, że kluczowe jest:

1. Zebranie informacji o hierarchii klas, modułów oraz występujących w nich metod
2. Wywnioskowanie “typu” (klasy, czasami coś więcej) zmiennych

Informacja o typie zmiennej

Informacja o typie sama przez się (przyp. kolokwializm?) jest informacją cenną dla programisty. Implementacja wynika bezpośrednio z (2). Mówimy o języku dynamicznie typowanym, więc w dalszej części pracy zostanie zdefiniowane czym jest typ, jednak ponieważ ruby jest językiem w pełni obiektowym i nie ma w nim typów prymitywnych, intuicyjnie typ możemy (z reguły, nie zawsze) powiązać z klasą obiektu, który jest przypisany do zmiennej.

Skok do definicji

Programista często dowiedzieć się jak wygląda definicja różnych bytów (?) w kodzie źródłowym:

1. definicja zmiennej lokalnej - nieskomplikowany przypadek i rzadko potrzebny, ponieważ krótkie metody z reguły powodują że definicja jest widoczna na pierwszy rzut oka
2. definicja stałej - wbrew pozorom, nie jest to proste, ponieważ w rubim stałe mogą być przypisywane ponownie (sic) oraz reguły zagnieżdżania i referencji nie są łatwe (z reguły zrozumiałe przez osoby z kilkuletnim doświadczeniem)
3. definicja metody - najtrudniejsza część, a jednocześnie najbardziej potrzebna. Będąc w kodzie źródłowym w miejscu wywołania metody, chcielibyśmy skoczyć do jej definicji. Metoda jest wywoływana na jakimś obiekcie, więc musimy znać "typ" zmiennej aby zlokalizować odpowiednią definicję.

Autouzupełnianie

Podobnie jak wyżej, musimy znać typ zmiennej aby wiedzieć jakie metody powinniśmy zasugerować programiście.

Refactoring

Aby móc przeprowadzać w miarę bezpieczne automatyczne zmiany kodu źródłowego, takie jak:

- Zmiana nazwy klasy (globalnie, w całym projekcie)
- Wydzielenie metody/zmiennej
- *Inline* (?) metody/zmiennej

również potrzebujemy informacji o typie zmiennej (przyp. rozwinąć dlaczego).

2.2.2. Indeksowanie

Uruchomienie serwera rozpoczyna przeindeksowanie wszystkich plików. Indeksowanie polega na:

1. Sparsowaniu pliku źródłowego do AST (*abstract syntax tree*)

2. Przejście przez AST funkcją dodającą wierzchołki i krawędzie do DFG (*data flow graph*) oraz zbierającą informacje o definicjach klas, modułów i metod. Wierzchołki są często przypisane do danego węzła w AST, a więc również ścieżki i pozycji w pliku.

3. Uruchomienie funkcji wnioskującej typ wszystkich wierzchołków na podstawie informacji w grafie i hierarchii klas.

2.3. Budowanie grafu

(na razie mniej więcej wymienione części o których trzeba powiedzieć)

2.3.1. Wstęp

2.3.2. Parę prostych przypadków, zmienne lokalne i literały

2.3.3. Parę pozornie łatwych, a podłych przypadków

2.3.4. Klasy, moduły i inne stałe

2.3.5. Metody

2.3.6. Dziedziczenie

2.3.7. attr reader, private

2.3.8. Wywołania metod

2.3.9. Wywołania super

2.3.10. Klasy singletonowe

2.3.11. Include/extend/prepare

2.3.12. Zmienne globalne instancji, klasowe, instancji klas

2.3.13. Bloki / lambdy

2.3.14. Struktury

2.3.15. Biblioteka standardowa

2.4. Wnioskowanie typu

W sporej części będą punkty jak wyżej, dodatkowo:

2.4.1. Typy

2.4.2. Specjalny typ - Array/Tuple

2.4.3. Specjalny typ - Hash

2.4.4. Wywołania popularnych funkcji polimorficznych, Array.map

2.4.5. Szczególne funkcje w Object

2.5. Future work

2.5.1. More metaprogramming intelligence

2.5.2. Bundler, gems and libraries

2.5.3. YARD

2.5.4. Rust implementation

2.5.5. Caching

2.5.6. Tracking effects, ex. exceptions

2.5.7. Tracepoint