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

11 sierpnia 2018

Streszczenie

. . .

. . .

Spis treści

Rozdział 1.

Wprowadzenie

Współcześnie powstaje wiele narzędzi ułatwiających i przyśpieszających pracę programistów. W szczególności istnieją zintegrowane środowiska programistyczne (IDE), które próbują dostarczyć jak najwięcej takich narzędzi w jednym spójnym środowisku. Wśród najbardziej podstawowych funkcjonalności takich środowisk można znaleźć m. in.:

- skok do definicji (stałej, funkcji lub zmiennej)
- informacja o typie zmiennej
- autouzupełnianie pełnej nazwy aktualnie pisanej stałej, zmiennej lub nazwy funkcji
- 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 nie występujące 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, trudność implementacji 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, skok do definicji). 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, które to zadanie znacznie utrudniają (np. w C, wywołanie funkcji która jest przechowana we wskaźniku), 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 typu wyrażenia jest szczególnie ciężkie z dwóch powodów:

- wiele różnych dostępnych konstrukcji języka, takich jak wielokrotne dziedziczenie, klasy singletonowe, *prepend* (przypis: nie wiem jak ten mechanizm się nazywa w nomenklaturze języków programowania) sprawia że zbudowanie hierarchii klas i poruszanie się po niej jest niełatwe
- społeczność rubiego jest przyzwyczajona do korzystania z metaprogramowania w bibliotekach i projektach, a dynamicznie generowane (w czasie uruchomienia programu) nazwy zmiennych, metod oraz klas sprawiają że efektywna statyczna analiza tych fragmentów kodu jest niemożliwa

1.1.3. Aktualny stan

Do tworzenia oprogramowania w Ruby, programiści najczęściej wykorzystują Rubymine IDE lub edytory tekstu wspierane wtyczkami, takie jak Vim, Emacs czy Atom (przypis: strzał na razie, ale nic istotnego to nie zmieni). 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 żadnego wkładu 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 jaki kod jest często produkowany (a więc – z którym programiści muszą pracować na co dzień), nawet jeżeli jest to kod niekoniecznie dobrej jakości. 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 dla ich zaimplementowania są dwa zadania:

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

Informacja o typie zmiennej

Już sama informacja o typie jest cenna dla programisty. Implementacja przedstawienia tego typu programiście wynika bezpośrednio z (2). Mówimy o języku dynamicznie typowanym, więc w dalszej części pracy zostanie zdefiniowane czym jest typ. 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 chce dowiedzieć się jak wyglądają definicje różnych bytów, takich jak zmienne lokalne, stałe czy metody w kodzie źródłowym.

Przypadek zmiennej lokalnej jest nieskomplikowany i rzadko potrzebny, ponieważ krótkie metody z reguły powodują że definicja jest widoczna na pierwszy rzut oka. Inaczej jest w przypadku stałej. Wbrew pozorom, odszukanie takiej definicji nie jest proste, ponieważ w Rubim stałe mogą być przypisywane ponownie (sic), a reguły zagnieżdżania i referencji nie są łatwe (z reguły są zrozumiałe dla osób z kilkuletnim doświadczeniem).

Jednak najtrudniejszą część, a jednocześnie najbardziej potrzebną jest skok do definicji metody. Metoda jest wywoływana na jakimś obiekcie, więc musimy znać "typ" zmiennej aby zlokalizować odpowiednią definicję.

Autouzupełnianie

Pisząc kod, programista chce uniknąć napisania błędnej nazwy metody i pomaga w tym funkcjonalność autouzupełniania. Polega ono na wyświetleniu listy metod które można wywołać na danym wyrażeniu. Podobnie jak w przypadku skoku do definicji, aby wiedzieć jakie metody zaproponować programiście, musimy znać jak najdokładniej typ wyrażenia.

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 to spora część frontendu typowego interpretera, zawierający analizę leksykalną, składniową i semantyczną. Proces ten polega na trzech etapach:

1. Sparsowaniu pliku źródłowego do AST (*abstract syntax tree*)
2. Przejście przez AST funkcją dodającą wierzchołki i krawędzie do DFG (*data flow graph*) oraz jednocześnie 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

2.3.1. Wstęp

Graf który budujemy jest skierowany i może zawierać cykle. Wierzchołki są etykietowane, tj. każdy wierzchołek ma swój "rodzaj" oraz być może dodatkowe parametry (zależnie od rodzaju). Ponadto wierzchołek może mieć przypisaną lokalizację w kodzie źródłowym. Lokalizacja składa się ze ścieżki do pliku, pozycji początkowej w kodzie źródłowym (numer linii i numer kolumny) oraz pozycji końcowej w kodzie źródłowym. Krawędzie są nieetykietowane.

2.3.2. Literały

Na początek warto popatrzeć na najprostsze AST, czyli literały. AST literału liczby całkowitej (np. `42`) zostanie przekształcony na pojedyńczy wierzchołek rodzaju `int`. Rodzaj będzie potrzebny funkcji wnioskującej typy, która na podstawie rodzaju, dodatkowych parametrów wierzchołka oraz wierzchołków z krawędzi wejściowych będzie decydowała jakiego typu jest dany wierzchołek. W tym prostym przypadku wierzchołek nie ma żadnych parametrów i nie zwracamy uwagi na krawędzie wejściowe (bo ich, z metody budowania grafu, dla wierzchołka tego rodzaju nawet nie będzie) tylko przypisujemy wszystkim wierzchołkom rodzaju `int` typ `Nominal(Integer)`.

Tak samo jest z literałami liczbami wymiernych, zmiennopozycyjnych, zespolonych oraz wartości `true`, `false` oraz `nil`.

Podobnie sytuacja ma się w przypadku literałów tekstowych (wierzchołek rodzaju `str`) oraz symboli (wierzchołek rodzaju `sym`). Drobna różnica jest taka, że

tekst może być z interpolacją (np. ‘‘foo#{somevariable}bar’’), a w takim przypadku najpierw rekurencyjnie przetwarzamy wszystkie interpolowane (?) wyrażenia, ponieważ typ wyrażeń interpolowanych też może być potrzebny programiście. Jednakże, funkcja wnioskująca przypisuje wierzchołkowi rodzaju `str` zawsze typ `Nominal(String)` niezależnie od krawędzi wejściowych. Analogicznie jest w przypadku symboli.

2.3.3. Literały tablic i słowników

Pierwszym ciekawym przypadkiem jest literał tablicy, np. `[42, ‘‘string’’]`. Dla takiego AST, najpierw budujemy wierzchołki dla wszystkich poddrzew - w tym przykładzie dostaniemy dwa wierzchołki, odpowiednio rodzaju `int` oraz `str`. Następnie dodajemy wierzchołek rodzaju `array`, który będzie wynikiem przetwarzania tego AST, oraz dodamy krawędzie z wierzchołków `int` i `str` do wierzchołka `array`.

Funkcja wnioskująca typ dla wierzchołka rodzaju `array` zawsze patrzy na typy wszystkich wierzchołków krawędzi wejściowych, tworzy z nich jeden typ Union – w naszym przykładzie `Union(Integer or String)` oraz finalnie przypisuje temu wierzchołkowi typ parametryzowany `Generic(Array)<Union(Integer or String)>`.

2.3.4. Zmienne globalne

W ogólności nie możemy wiedzieć w jakiej kolejności zostaną zmienne globalne, więc zakładamy, że w momencie referencji zmiennej globalnej jej wartość może pochodzić z dowolnego przypisania tej zmiennej w całym kodzie źródłowym. W tym celu, dla każdego identyfikatora (nie wystąpienia!) zmiennej globalnej tworzymy specjalny wierzchołek rodzaju `gvar_definition`. Za każdym razem gdy przetwarzamy przypisanie jakiejś wartości do danej zmiennej globalnej, tworzymy krawędź od tej wartości, do danego wierzchołka `gvar_definition`. Odpowiednio, przetwarzając referencję zmiennej globalnej, tworzymy krawędź od `gvar_definition` do tej wartości. W ten sposób, funkcja wnioskująca typy przekaże wszystkie możliwe przypisania danej zmiennej globalnej do wszystkich ich wystąpień w programie.

2.3.5. Klasy i moduły

Funkcja budująca graf, przyjmuje jako zależność instancję Bazy danych - na początku przetwarzania projektu pustą. W bazie przechowujemy wszystkie informacje, które pozwalają nam szybko odpowiadać na żądania, które wysyła do nas użytkownik. Dlatego też przechowujemy tam informacje np. o znalezionych definicjach stałych i metod.

Podczas przechodzenia przez AST pliku, przetwarzając wierzchołek definicji klasy lub modułu, zapisujemy tę definicję w bazie. W Ruby klasa (i moduł) są war-

tościami, więc odrębnie zapisujemy definicję klasy (i informację np. o jej rodzicu), a oddzielnie fakt, że klasa ta została przypisana do pewnej stałej. Pozwala to na wspieranie otwartych klas, oraz w przyszłości (przyp: mogę w sobie w pracy magisterskiej mówić o 'przyszlosci'?) klas/modułów anonimowych.

Funkcja budująca graf ma też kontekst, który przechowuje kilka informacji o aktualnym stanie budowania grafu. Po pierwsze, wie w jakim zagnieżdżeniu klas / modułów jesteśmy, oraz jeżeli aktualnie przetwarzamy definicję metody, to co to jest za metoda. Ponadto, mamy słownik mapujący identyfikatory obecnych zmiennych lokalnych do ich możliwych definicji (w stylu klasycznej analizy statycznej *Reaching Definitions*). Wreszcie, trzymamy też krótką informację o aktualnie zadeklarowanej widzialności definiowanych metod (publiczne, prywatne lub chronione).

2.3.6. Inne stałe

Ponieważ zdefiniowane stałe mają domyślnie globalną widzialność oraz mogą być przypisywane ponownie, ich przypisywanie nie różni się bardzo od zwykłych zmiennych globalnych. Po pierwsze, odróżnia je to, że nie są prostym identyfikatorem (jak `$foo`), ale złożonym z kilku, niekoniecznie statycznych (jeśli `x = Foo` to `x::Bar` jest poprawną referencją do `Foo::Bar`), więc wymagają specjalnych struktur danych w bazie, algorytmu rozwiązywania tych referencji i (przez to, że referencje mogą być dynamiczne) niekoniecznie są zawsze obliczalne w statycznej analizie.

Jest to też pierwsze miejsce, w którym stosujemy heurystyki bazujące na popularnych idiomach w Ruby. Przykładowo, w wyrażeniu `NetworkError = Class.new(StandardError)` najpierw tworzymy klasę anonimową dziedziczącą po klasie `StandardError`, a następnie przypisujemy ją do stałej `NetworkError`. Normalnie, nie wiedzielibyśmy o tym, że `NetworkError` jest klasą (bo nie śledzimy wartości), ale stosujemy heurystykę szukającą wyrażenia wg wzoru `X = Class.new(Y)`.

2.3.7. Metody

Podobnie jak jest to ze stałymi, podczas budowania grafu zapisujemy znalezione definicje metod. Dokładniej, zapisujemy nazwę, identyfikator definicji klasy/modułu wewnątrz której została metoda zdefiniowana, argumenty formalne, widoczność, oraz lokalizację w pliku. Zapisujemy też w bazie wierzchołki odpowiadające argumentom formalnym i wartości zwracanej przez metodę.

2.3.8. Zmienne lokalne

Tak jak zostało to wspomniane, wierzchołki o przypisaniach zmiennych lokalnych są przechowywane w kontekście w postaci podobnej do klasycznej statycznej analizy *Reaching Definitions*, więc są dość dokładne.

2.3.9. Dziedziczenie

Ruby wspiera wiele rodzajów dziedziczenia, ale poszczególne konstrukty nie są od siebie istotnie różne. Jednak ponieważ to po czym dziedziczymy, składniowo, nie musi być identyfikatorem, ale może być dowolnym wyrażeniem (więc poprawne są konstrukty typu `class Foo < $x` czy class Foo ¡ self), znowu niemożliwa jest w pełni poprawna analiza. Aktualnie ograniczamy się tylko do obsługiwania sytuacji gdy to po czym dziedziczymy jest poprawnym identyfikatorem stałej, w przeciwnym wypadku ignorujemy to wyrażenie. W przyszłości łatwo można by wprowadzić heurystykę pozwalającą np. na dziedziczenie po strukturach (`class Foo < Struct.new(:field)`), ponieważ jest kilka takich idiomatycznych przypadków w których dziedziczymy po wyrażeniach nie będących identyfikatorami.

2.3.10. Zmienne instancji

Skoro Ruby jest językiem obiektowym, nie może zabraknąć zmiennych instancji. Aktualnie traktujemy je w podobny sposób jak zmienne globalne - nie próbujemy śledzić dokładnego flow, tylko zbieramy wszystkie możliwe przypisania zmiennych instancji, wszystkie możliwe użycia, i je łączymy ze sobą przez wierzchołek pośredni `ivar_definition`. Aktualnie ignorujemy fakt, że zmienna `@x` w klasie `Foo` i `Bar` jest tą samą zmienną, jeśli `Bar` jest rodzicem `Foo`, ale łatwo rozszerzyć kod o tę funkcjonalność (przyp: może to jeszcze zrobię w sumie, bo szkoda tego tłumaczenia).

2.3.11. attr reader, private

Są w Ruby metody *klasowe* (tj. metody działające na tzw. *eigenclass* danej klasy, a nie instancji), które definiują inne metody, więc są już de facto metaprogramowaniem. W bibliotece standardowej Rubiego najpopularniejsze to `attr_reader` i `attr_writer`. `attr_reader(:foo)` definiuje metodę odczytującą zmienną instancji, o takiej samej nazwie, jest więc odpowiednikiem kodu `def foo; @foo; end`. Oczywiście, `attr_reader` przyjmuje dowolną liczbę argumentów, więc szybko można za jego pomocą zdefiniować wiele takich funkcji.

Tutaj funkcja budująca ucieka się do heurystyki wykrywającej użycie funkcji o nazwie `attr_reader` i zamiast uznawać to wywołanie za zwykłe wywołanie metody, definiuje odpowiednią metodę dla każdego argumentu. Dodatkowo, ustawia lokalizację występowania definicji tak zdefiniowanej metody na linijkę z `attr_reader`, więc programista, który użyje skoku do definicji łatwo dowie się, że metoda ta jest definiowana za pomocą `attr_reader`. Bardzo podobnie jest z metodą `attr_writer`, która definiuje przypisanie do zmiennej instancji (tzw. *setter*).

2.3.12. Widzialność metod

W większości języków obiektowych, `private`, `public` i `protected` są słowami kluczowymi (a przez to - posiadają własne AST), w Ruby jednakże są to zwykłe wywołania metod klasowych. Z tego powodu, znowu korzystamy z heurystyk. W tym wypadku jednak takie heurystyki są mniej akceptowalne, ponieważ te nazwy są mogą prawdopodobnie pojawić się w produkcyjnych programach – natknąłem się na ten problem osobiście (przyp: nie wiem czy powinienem pisać takie rzeczy, w pierwszej osobie). Jednak z braku lepszego rozwiązania, stosujemy tutaj też również tylko prostą heurystykę po nazwie metody. W tym wypadku, analizujemy argumenty jakie otrzymuje wywołanie metody `private` i wysyłamy do bazy żądanie zmiany widzialności metod o podanych jako argumenty nazwach, w definicji klasy którą aktualnie przetwarzamy (informację o aktualnie przetwarzanej definicji klasy trzymamy w kontekście).

2.3.13. Wywołania metod

Pomimo tego, że jest to prawdopodobnie najważniejszy AST, jego analiza podczas budowania grafu nie jest bardzo skomplikowana. Funkcja budująca przetwarza rekurencyjnie obiekt na którym wywoływana jest metoda, przekazywane argumenty oraz tworzy dodatkowy wierzchołek na wynik tej metody. Następnie wszystkie te wierzchołki łączy w rekord i wrzuca do kolejki wywołań. Kolejka ta będzie używana dopiero podczas typowania.

2.3.14. Wywołania super

Wywołania super dzielą się na dwa rodzaje (i są to dwa różne wierzchołki AST). Pierwszy to zwykłe wywołanie `super(args)` znane z innych języków programowania, wywołujące odpowiadającą metodę z definicji rodzica i przekazującą podane do `super` argumenty. Analiza tego wywołania jest bardzo podobna do zwykłego wywołania metody poza tym, że do rekordu wywołania dopisujemy jeszcze identyfikator aktualnie przetwarzanej metody, aby łatwo znaleźć rodzica i odpowiadającą w nim metodę. Drugi rodzaj wywołania jest bezargumentowy, tj. `super`. W takim wypadku, wszystkie argumenty przekazane do tej metody są przez interpreter podane do tego bezargumentowego wywołania `super`, tj. metoda rodzica zostaje wywołana dokładnie z takimi samymi argumentami. W takim wypadku, jedyne co robimy to zapisujemy w bazie wierzchołek odpowiadający wywołaniu bezargumentowego `super`.

2.3.15. Bloki i lambdy

Jak w większości współczesnych języków programowania, w Ruby również funkcje są *first class citizens*, a więc są wartościami, które mogą być przypisywane do zmiennych i przekazywane jako argumenty. W Ruby są obecne też bloki, które są niczym innym jak przekazywaną do wywołania metody funkcją, którą można wywoływać wewnątrz metody za pomocą specjalnej składni `yield`. Każde użycie składni `yield` wewnątrz metody wywołuje tę funkcję i przekazuje do niej argumenty (jeśli zostały jakieś przekazane do `yield`).

Analizując słowo kluczowe `yield`, tworzymy specjalny rekord składający się z przekazanych argumentów i specjalnego wierzchołka na wynik - bardzo podobnie jak w przypadku wywołania metody. Dodajemy ten rekord do specjalnej listy połączonej z definicją aktualnie przetwarzanej metody.

Analizując blok, podejmujemy działania podobne, jak przy definiowaniu zwykłej metody, tj. analizujemy argumenty, przygotowujemy wierzchołki na argumenty formalne, wierzchołek na wynik bloku i wszystko łączymy w rekord. Ruby nie posiada specjalnego słowa kluczowego na zbudowanie funkcji anonimowej, jest to po prostu blok przekazany do funkcji o nazwie `lambda`. Prosta heurystyka znowu pozwala rozróżnić przypadki, kiedy po prostu budujemy funkcję anonimową, a kiedy przekazujemy blok do wywołania metody. W tym pierwszym przypadku, zapisujemy identyfikator (automatycznie wygenerowany, na potrzeby analizy) tej funkcji w wierzchołku, a w drugim – zapisujemy ten identyfikator przy wywołaniu metody.

2.3.16. Biblioteka standardowa

2.3.17. self

2.3.18. Named arguments

2.4. Wnioskowanie typu

Kiedy graf jest gotowy, przekazujemy go do funkcji wnioskującej typy wszystkich wierzchołków. Funkcja ta jest podzielona na kilka etapów. Do kolejki z wierzchołkami są na początku dodawane wszystkie wierzchołki, a typ wszystkich jest ustawiany na `Bottom`. Następnie przetwarzamy każdy wierzchołek (nazwijmy go `X`) z kolejki. Jeżeli po inferencji typu, typ wierzchołka `X` się zmienił, patrzymy na wszystkie krawędzie wychodzące z `X` i dodajemy docelowe wierzchołki tych krawędzi do kolejki, jeśli ich tam jeszcze nie ma. Gdy przetworzymy wszystkie wierzchołki, przechodzimy do przetwarzania wywołań metod. Wywołanie metody przetwarzamy tylko, jeśli wszystkie jej argumenty oraz obiekt na którym wywoływana jest metoda, są ukonkretnione (tj. żaden z nich nie jest typu `Bottom`). Następnie, w zależności od obliczonego typu

obiektu na którym jest wywoływana metoda, oraz jej nazwy, zachodzi kilka heurystyk opisanych w następnych częściach. Schemat we wszystkich jest podobny i polega na znalezieniu w bazie definicji metody, która została wywołana, a następnie połączeniu wierzchołków argumentów wywołania (przyp: jest jakaś lepsza polska nazwa na tzw. actual arguments?) z argumentami formalnymi oraz wierzchołka wyniku metody z wierzchołkiem wyniku wywołania. Dzięki temu że wszystkie argumenty są zdefiniowane, możemy tutaj przeprowadzić kilka dodatkowych heurystyk. Są obecne specjalne mechanizmy pilnujące, że daną definicję metody przypiszemy do danego wywołania tylko raz (bo wywołania mogą być przetwarzane więcej niż raz). Następnie, jeśli w efekcie przetwarzania metod zostały dodane jakieś wierzchołki do kolejki, to wracamy do etapu przetwarzania wierzchołków. Jeżeli żadne nie zostały dodane, to przetwarzamy znowu wywołania metod, tym razem obsługując wywołanie nawet wtedy, jeśli któryś z argumentów nie jest zdefiniowany.

2.4.1. Heurystyki obsługi wywołań

Najpopularniejsza jest sytuacja, gdy wywnioskowaliśmy, że obiekt na którym jest wywoływana metoda (przyp: da się to jakoś zwięźlej?) jest jest typu `NominalType(Foo)`. Wtedy najpierw sprawdzamy, czy to wywołanie nie jest jednym z wywołań metod polimorficznych, które obsługujemy w specjalny sposób. Takich metod jest skończona liczba i są to metody z biblioteki standardowej. Jeżeli nie, to wykonywane jest żądanie do bazy o definicję metody instancji o danej nazwie, dla klasy `Foo`. Jeżeli taka metoda istniała, to następuje parę rzeczy. Najpierw, wspomniane już połączenie wierzchołków argumentów wywołania z wierzchołkami argumentów formalnych, oraz wierzchołka wyniku metody z wierzchołkiem wyniku wywołania. Następnie sprawdzamy, czy w definicji znalezionej metody są jakieś wywołania bezargumentowego `super`. Jeśli tak, to łączymy wierzchołki argumentów wywołania z wierzchołkami argumentów formalnych metody rodzica – odpowiadającą metodę rodzica znajdujemy przez odpowiednie zapytanie do bazy.

Metoda klasowa

Konstruktor

Lambda

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