Zastosowanie analizy statycznej języka Ruby w edytorach tekstu

(Applications of Ruby language static analysis in text editors)

Rafał Łasocha

Praca magisterska

Promotor: prof. Witold Charatonik

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

22 listopada 2018

Streszczenie

Współcześni programiści mogą wybierać spośród wielu różnych edytorów tekstu do edycji kodu źródłowego. Jedną z rzeczy ułatwiających ich pracę jest wsparcie edytora dla danego języka programowania pozwalające na szybkie poruszanie się po definicjach lub podpowiedzi podczas edycji kodu. Praca ma na celu opisanie wtyczki, która dzięki analizie statycznej kodu źródłowego dostarcza te funkcjonalności programistom języka Ruby. Z powodu dynamicznej natury tego języka, analiza statyczna wymaga wielu heurystyk i równowagi pomiędzy bezpieczeństwem analizy a użytecznością dla programistów.

Currently programmers have ability to choose from variety of text editors to edit source code. Editor support for programming languages is one of the features allowing programmers to accurate jumping to definitions or providing him with suggestions when editing source code. The purpose of this master's thesis is to describe the plug-in delivering these functionalities to Ruby programmers thanks to static code analysis. Due to Ruby's dynamic nature, static analysis require multiple heuristics and balancing between analysis' safety and usefulness for programmers.

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

Niniejsza praca przedstawia jak można wykorzystać analizę statyczną plików źródłowych, aby dostarczyć powyższe funkcjonalności w języku Ruby [5], 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, są dużo popularniejsze dla języków silnie typowanych, ponieważ w takich językach implementacja narzędzi bazujących na analizie kodu źródłowego jest znacznie prostsza. Głównym powodem jest to, że jeżeli już wiadomo jakiego typu jest dane wyrażenie, lista dostępnych metod, które możemy na nim wywołać jest łatwa do obliczenia (autouzupełnianie). Ponadto, 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 czy klasy singletonowe sprawiają, że zbudowanie hierarchii klas i poruszanie się po niej jest niełatwe
- społeczność języka Ruby 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 [9] lub edytory tekstu wspierane wtyczkami, takie jak Vim [11], Emacs [4] czy Atom [1]. 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 ([2]) – 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 ([8]) – dostarcza informacje zależne od kontekstu, ale wykorzystuje bardzo proste heurystyki do zawężenia wyników
- Solargraph ([10]) – dostarcza podobne funkcjonalności do tych wymienionych we wstępie, stosuje do tego analizę statyczną oraz wykorzystuje inne źródła

(takie jak dokumentacja YARD [12]), ale analiza statyczna nie jest wykorzystywana do wnioskowania typów argumentów formalnych na podstawie znalezionych wywołań danej metody

- RGL ([7]) – 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*, [6]). 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 Ruby 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 zawsze wywoływana na jakimś wyrażeniu, więc musimy znać "typ" obiektu do którego ewaluuje się dane wyrażenie aby zlokalizować odpowiednią definicję.

Autouzupełnianie

Pisząc kod, programista chce uniknąć napisania błędnej nazwy metody. Pomaga w tym funkcjonalność autouzupełniania, które polega 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, na którym próbujemy użyć funkcjonalności autouzupełniania.

2.2.2. Indeksowanie

Uruchomienie serwera rozpoczyna przeindeksowanie wszystkich plików. Indeksowanie to spora część frontendu typowego interpretera, zawierającego analizę leksykalną, składniową i semantyczną. Proces ten składa się z trzech etapów:

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

2. Przejściu 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. Idea algorytmu budującego graf przepływu dla danych bazującego na AST została zaczerpnięta z pracy DeFouw, Grove i Chambers [3], która miała na celu optymalizację kompilatorów dla obiektowych języków programowania.

3. Uruchomieniu funkcji wnioskującej typ wszystkich wierzchołków z DFG na podstawie informacji w grafie i hierarchii klas.

Rozdział 3.

Budowanie grafu przepływu danych

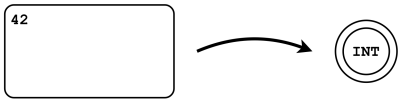
3.1. Wstęp

Graf DFG budujemy po sparsowaniu pliku zródłowego do AST i będziemy go używać do ustalenia typów wszystkich wyrażeń. Podczas budowania grafu jednocześnie zbieramy informacje o definicjach klas, modułów i metod. Graf ten 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.

3.2. Literały

Na początek warto popatrzeć na najprostsze przypadki drzew AST, czyli literały. AST literału liczby całkowitej (np. `42`) zostanie przekształcony na pojedynczy wierzchołek rodzaju `int` (Rys. **??**). 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 liczb wymiernych, zmiennopozycyjnych, zespolonych oraz literałami `true`, `false` oraz `nil`.

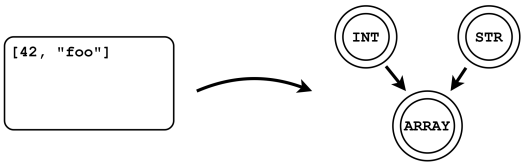


Rysunek 3.1: Budowanie grafu dla literału liczby całkowitej
Po lewej kod źródłowy, po prawej graf zbudowany z tego kodu źródłowego.

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.

3.3. Literały tablic

Pierwszym ciekawym przypadkiem jest literał tablicy, np. `[42, ‘‘string’’]` (Rys. **??**). 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`.



Rysunek 3.2: Budowanie grafu dla literału tablicy

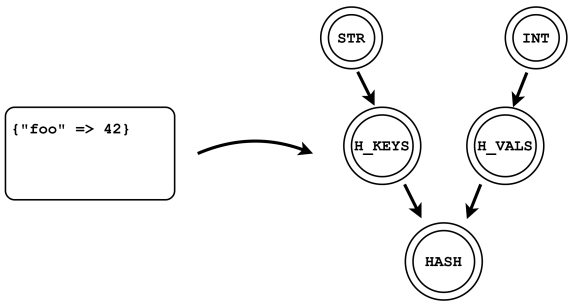
Funkcja wnioskująca typ dla wierzchołka rodzaju `array` zawsze patrzy na typy wszystkich wierzchołków krawędzi wejściowych, tworzy z nich jeden typ Union – w naszym przykładzie `Union(Integer or String)` oraz finalnie przypisuje temu wierzchołkowi typ parametryzowany `Generic(Array)<Union(Integer or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

3.4. Słowniki

W słownikach zarówno klucze i wartości mogą być różnego typu. Literał budowania słownika może również w sobie zawierać klucze (i oczywiście wartości), które nie są literałami. Przetwarzając literał słownika, budujemy trzy wierzchołki: `hash`,

`hash_keys` i `hash_values` (Rys. **??**). Następnie przetwarzamy wszystkie klucze, oraz dodajemy dla każdego klucza krawędź pomiędzy wierzchołkiem wyrażenia klucza, a wierzchołkiem `hash_keys`. Podobnie przetwarzamy wszystkie wartości i dodajemy krawędź pomiędzy wierzchołkami wartości, a wierzchołkiem `hash_values`. Na końcu dodajemy dwie krawędzie: pomiędzy `hash_keys` oraz `hash` i `hash_values` oraz `hash`.

Funkcja wnioskująca typ dla wierzchołka rodzajów `hash_keys` oraz `hash_values` działa grupująco – tj. zbiera wszystkie typy z wierzchołków z krawędzi wejściowych i łączy je w jeden typ `Union(...)`. Funkcja typująca wierzchołek `hash` patrzy na rodzaje wierzchołków i buduje finalny typ `GenericType(Hash)<typ otrzymany z hash_keys, typ otrzymany z hash_values>`.



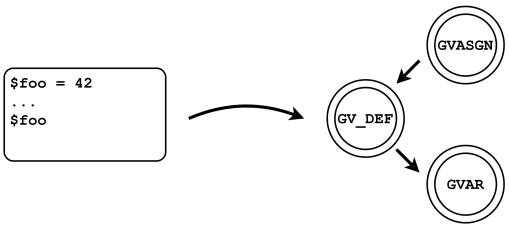
Rysunek 3.3: Budowanie grafu dla literału słownika

3.5. Zmienne globalne

W ogólności nie możemy wiedzieć w jakiej kolejności zostaną przypisane 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` (`gv_def` na rysunku **??**). 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.

3.6. 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 wysyłane do nas przez



Rysunek 3.4: Budowanie grafu dla zmiennych globalnych

użytkownika. Dlatego też przechowujemy tam informacje np. o znalezionych podczas budowania grafu definicjach stałych i metod. Gdy podczas przechodzenia przez drzewo AST przetwarzamy wierzchołek definicji klasy lub modułu, zapisujemy tę definicję w bazie. W Ruby klasa (i moduł) są wartościami, 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 klas/modułów anonimowych.

Funkcja budująca graf jest rekurencyjna i przekazuje sobie dwa argumenty: podgraf AST do przetworzenia, oraz kontekst. Kontekst przechowuje kilka informacji o aktualnym stanie budowania grafu. Po pierwsze, wie jakie jest aktualne zagnieżdżenie klas / modułów, oraz jeżeli aktualnie przetwarzamy definicję metody, to co to jest za metoda. W kontekście staramy się również w miarę możliwości śledzić czym jest aktualnie `self`. Ponadto mamy słownik mapujący identyfikatory obecnych zmiennych lokalnych do ich możliwych definicji (w stylu klasycznej analizy statycznej *Reaching Definitions*). Wreszcie, trzymamy w nim też krótką informację o aktualnie zadeklarowanej widzialności definiowanych metod (publiczne, prywatne lub chronione).

3.7. Inne stałe

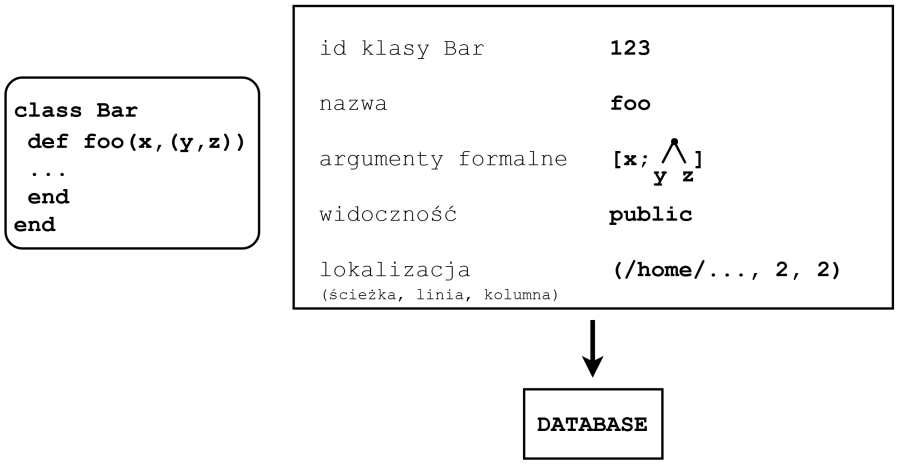
Ponieważ zdefiniowane stałe mają domyślnie globalną widzialność oraz mogą być przypisywane ponownie, ich obsługa nie różni się bardzo od zwykłych zmiennych globalnych. Po pierwsze, odróżnia je to, że nie są prostym identyfikatorem (jak np. `$foo`). Stałe mogą być złożone z kilku, niekoniecznie statycznych identyfikatorów (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. Jako przykład weźmy następujący idiom tworzący w zwięzły sposób nową klasę. Wyrażenie `NetworkError = Class.new(StandardError)` najpierw tworzymy klasę anonimową dziedziczącą po klasie `StandardError`, a na-

stępnie 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)`. Jeżeli heurystyka znalazła takie wyrażenie, to postępujemy z nim podobnie jak z definicją klasy. W przeciwnym przypadku, jak z definicją zwykłej stałej.

3.8. Metody

Podobnie jak jest to ze stałymi, podczas budowania grafu zapisujemy w Bazie znalezione definicje metod. Dokładniej, zapisujemy nazwę, identyfikator definicji klasy/modułu wewnątrz której została metoda zdefiniowana, drzewo argumentów formalnych (w Ruby argumenty formalne przez dekonstruktory mają postać drzewa), widoczność, oraz lokalizację w pliku. Zapisujemy też w bazie wierzchołki odpowiadające argumentom formalnym i wartości zwracanej przez metodę. Następnie ustawiamy tę metodę jako aktualnie przetwarzaną w kontekście i budujemy graf dla ciała tej metody.



Rysunek 3.5: Zapisywanie informacji o metodzie do bazy danych

3.9. 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. Podczas przechodzenia przez AST, w kontekście przekazujemy słownik mapujący identyfikatory zmiennych lokalnych w aktualnym zasięgu leksykalnym (ang. *lexical scope*) na możliwe przypisania danej zmiennej lokalnej, a dokładniej – wierzchołki tych przypisań.

3.10. 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.

3.11. Zmienne instancji

Skoro Ruby jest językiem obiektowym, nie może zabraknąć zmiennych instancji. Traktujemy je w podobny sposób jak zmienne globalne – nie próbujemy śledzić dokładnego przepływu, 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.

3.12. Eigenclass

Ruby, będąc językiem w pełni obiektowym, nie ma mechanizmu metod statycznych znanego np. z C++, ale ma podobnie działające tzw. metody klasowe. Klasy są obiektem, a jeżeli coś jest obiektem (instancją) to musi mieć również swoją klasę. Naturalnym wydawałoby się, że klasa jest instancją klasy `Class`, ale w Ruby mamy dodatkowy element – każda definicja klasy tworzy dla tej klasy również `eigenclass`. Właściwym łańcuchem dziedziczenia jest więc `Foo < eigenclass Foo < Class`.

Otwierając definicję klasy (ale gdy ciągle nie jesteśmy wewnątrz definicji metody), `self` jest właśnie instancją tej *eigenclass*. W definicji klasy możemy wtedy korzystać ze składni `def self.foo` aby zdefiniować metodę na *eigenclass* lub `class << self` aby otworzyć definicję *eigenclass*. W ten sposób zdefiniowane metody zachowują się jak metody statyczne, tj. można je wywołać przez składnię `Klasa.foo(...)`.

Składnie te są reprezentowane przez konkretne wierzchołki, ale w miejscu `self` może być dowolne wyrażenie – możemy więc napisać np. `def Foo.bar` i zdefiniować metodę na *eigenclass* klasy `Foo` będąc wewnątrz kompletnie innej metody. Jednakże

prawie zawsze jest to `self`, więc obsługujemy tylko tę składnię. Obsługa definicji metody wewnątrz *eigenclass* (`def self.foo; ...; end`) lub definicji *eigenclass* (`class << self; ...; end`) jest bardzo podobna do obsługi zwykłej definicji metody lub klasy, z tą różnicą, że zamiast zdefiniować metodę na klasie, najpierw pobieramy z bazy *eigenclass*, a dopiero potem dodajemy definicję. Jeśli *eigenclass* nie ma, to dodajemy pustą definicję klasy jako tę właśnie *eigenclass*. Następnie postępujemy tak jak ze zwykłą metodą, czyli budujemy graf dla ciała tej metody. Różnice w budowaniu metod definiowanych na *eigenclass*, a zwykłymi metodami są przedstawione na Alg. **??**.

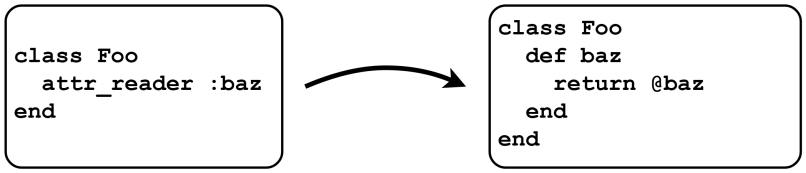
Algorytm 1: Fragmenty funkcji budującej obsługującej definiowanie metody instancji i metody klasowej

```
1  function buildGraph (ast, context):
2  ...
3  if ast.type = 'def' then                          // ast dla ‘def foo ...‘
4  |   ...
5  |   database.addMethod(..., context.currentlyAnalyzedKlassId);
6  |   ...
7  end
8  if ast.type = 'defs' then                      // ast dla ‘def self.foo ...‘
9  |   ...
10 |   eigenclass = database.getEigenclassOf(context.currentlyAnalyzedKlassId);
11 |   database.addMethod(..., eigenclass.id);
12 |   ...
13 end
```

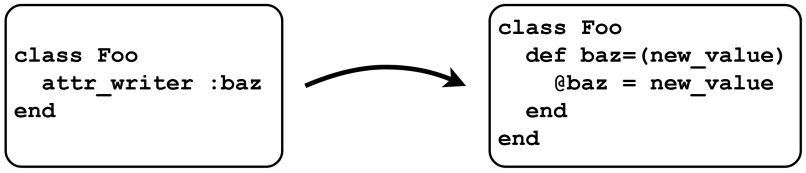
3.13. attr_reader, attr_writer

W Ruby wbudowane są metody *klasowe*, 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. 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 (Rys. **??**). 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*, Rys. **??**).



Rysunek 3.6: Transformacja wywołania `attr_reader`



Rysunek 3.7: Transformacja wywołania `attr_writer`

3.14. 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 mogą prawdopodobnie pojawić się w produkcyjnych programach – autor tej pracy natknął się na taki przypadek osobiście. Jednak z braku lepszego rozwiązania, stosujemy tutaj 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).

3.15. Wywołania metod

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

Algorytm 2: Fragment funkcji budującej obsługującej wywołanie funkcji

```
1  function buildGraph (ast, context):
2  ...
3  if ast.type = 'send' then
4      ...
5      if ast.messageName = 'private' and ast.sendObj = null then
6          for methodName ∈ getMethodNamesToChangeFromAst(ast) do
7              database.changeVisibility(context.currentlyAnalyzedKlassId,
                methodName, 'private');
8          end
9          return;
10     end
11     ...
12 end
```

3.16. 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`.

3.17. 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 użyć `yield` 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.

3.18. Biblioteka standardowa

Ruby jest językiem interpretowanym, więc definicje podstawowych klas i metod są bezpośrednio w interpreterze. Interpreter jest napisany w C, więc nie mamy dostępu do implementacji tych definicji w Ruby które moglibyśmy przeanalizować. Aby nasza analiza była praktyczna, definicje biblioteki standardowej powinny być w pewien sposób obsłużone.

Metody w bibliotece standardowej możemy podzielić na dwie grupy. W większości metod, jedyne co nas interesuje to zwracany typ – wyobraźmy sobie metody typu `File.read(filepath)`. Wystarczające jest jeżeli o tej metodzie zapiszemy, że zawsze zwraca typ `NominalType(String)`. W taki sposób możemy sobie poradzić z większością metod.

Cięższym przypadkiem są polimorficzne metody przetwarzania danych, takie jak `map` lub `filter` w klasie `Array`. Tablica jest typem parametryzowanym (np. `Generic(Array)<Integer>`), więc nie ma tutaj stałego typu który moglibyśmy uniwersalnie zwracać. Drugą opcją byłoby dodanie odpowiednich wierzchołków i krawędzi odpowiadających temu, jak dana metoda działa, i traktowanie jej dokładnie tak samo jak zwykłej metody podczas typowania (jedyną różnicą byłoby więc to, że graf dla definicji tej metody byłby napisany przez autora analizy, a nie automatycznie na podstawie AST). Jednak podczas typowania łączymy każde użycie metody z tą samą definicją formalną. Oznaczałoby to, że jeżeli w dwóch różnych miejscach kodu wywołujemy metodę `map`, raz na obiekcie typu `Generic(Array)<Integer>`, a raz na `Generic(Array)<String>`, to wywnioskowanym zwracanym typem dla obu tych wywołań byłby `Generic(Array)<Integer or String>`. A ponieważ w praktyce wywołujemy `map` na listach wielu różnych obiektów, taki zwracany typ byłby bardzo długi i bezużyteczny.

Trzecią opcją byłoby zapisanie przez autora analizy tylko wzoru jak wygląda graf dla tej definicji metody, ale tworzenie świeżych wierzchołków i krawędzi tego

grafu w każdym miejscu, w którym ta metoda jest wywoływana. Jest to rozwiązanie przyjęte w tej pracy. Oczywistą wadą jest większy rozmiar grafu, ale rozwiązanie to sprawdza się w praktycznych zastosowaniach.

3.19. self

`self` jest konstrukcją języków zorientowanych obiektowo, która sprawia problemy przy analizie statycznej. Jest tak ponieważ `self` jest bardzo często używany (w szczególności właściwie wszystkie wywołania metod prywatnych to wywołania na wyrażeniu `self`) oraz jednocześnie jego dokładny typ jest trudny do określenia. Nasza analiza korzysta z bardzo uproszczonego założenia, że `self` jest zawsze typem odpowiadającym definicji klasy, w której jest użyte. W przyszłości jednak można by było spróbować z założeniem, że `self` jest typu odpowiadającego danej definicji klasy lub jednej z jej podklas. Z jednej strony, takie rozwiązanie byłoby bardziej bezpieczną analizą. Z drugiej strony, w każdym projekcie są klasy podstawowe, po których dziedziczy wiele innych definicji klas. W takich przypadkach bezpieczniejsze podejście mogłoby się okazać niepraktyczne, ponieważ `self` byłby typowany jako unia np. 100 różnych klas.

3.20. Wyjątki

Obsługa wyjątków nie jest skomplikowana i główną trudnością jest odpowiednie obsłużenie zasięgu leksykalnego dla zmiennych lokalnych. Jedyną rzeczą wartą wspomnienia jest fakt, że `raise SomeError` jest pierwszym przykładem wyrażenia, któremu celowo przypisujemy podczas typowania typ `Bottom` (tj. brak typu), ponieważ nie zwraca to wyrażenie żadnej wartości.

3.21. Pętle

Pętle i słowa kluczowe z nimi związane (`break`, `next` itp.) również nie są zbyt interesujące. Po pierwsze dlatego, że z reguły wyrażenia związane z pętlami zwracają `nil` lub nic (tj. są typu `Bottom`), więc jedyne na co trzeba uważać, to znowu poprawne obsłużenie zasięgu leksykalnego zmiennych lokalnych. Po drugie dlatego, że pętle w Ruby są używane bardzo rzadko i iteracje w idiomatycznym Ruby wyraża się za pomocą funkcyjnych `map`, `filter`, `each` i innych.

3.22. Return

Ruby posiada znane z innych języków programowania słowo kluczowe `return`, kończące wykonywanie funkcji i zwrócenie wartości jako wynik funkcji. Aby to wspierać, pobieramy z kontekstu wierzchołki aktualnie przetwarzanej metody, a wśród nich wierzchołek rodzaju `call_result`. Jedyne co musimy zrobić to dodać krawędź pomiędzy wierzchołkiem wyrażenia przekazanego w `return`, a wierzchołkiem `call_result` pobranym wcześniej z kontekstu.

Rozdział 4.

Wnioskowanie typu

4.1. Typy

Mamy następujące typy:

- `Bottom` – typ oznaczający brak typu, używany kiedy typ wierzchołka jest jeszcze nie znany lub typu nie posiada (np. słowa kluczowe `raise` lub `break`)
- `NominalType(Foo)` – podstawowy typ, oznacza że wyrażenie jest instancją klasy Foo.
- `ClassType(Foo)` – typ oznaczający że wyrażenie jest klasą `Foo`. Innymi słowy, jest instancją klasy `Class` lub instancją *eigenclass* klasy `Foo`, ale tak dokładna informacja jest oczywiście przydatniejsza niż samo `NominalType(Class)`, dlatego mamy oddzielny typ na ten specjalny przypadek.
- `GenericType(Foo)<typ1, typ2, ...>` – typ oznaczający jeden z typów generycznych. Opisane w pracy są dwa: `Array` posiadający jeden parametr, oraz `Hash` posiadający dwa parametry (typ kluczy i typ wartości). Parametry są typami. Można sobie łatwo wyobrazić rozszerzenie na inne klasy z biblioteki standardowej, takie jak `Set` lub `Range`.
- `UnionType(typ1, typ2, ...)` – typ oznaczającą dowolny z wymienionych typów. Większość rodzajów wierzchołków w funkcji typującej jedyne co robi to bierze typy z wszystkich wierzchołków wejściowych i tworzy z nich jeden `UnionType`.
- `MainType` – typ szczególny, mało istotny w typowaniu. Definicje metod, które są zdefiniowane nie w definicji klasy, są zdefiniowane na specjalnym obiekcie `main`.

4.2. Funkcja wnioskująca typy

Kiedy graf jest gotowy, przekazujemy go do funkcji wnioskującej typy wszystkich wierzchołków (Alg. **??** i **??**). 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 wyrażenia na którym jest wywoływana metoda, oraz jej nazwy, wywołujemy kilka heurystyk opisanych w następnych częściach.

Algorytm 3: Pseudokod funkcji typującej

```
1  function typeNodes (nodesWorklist, messageSendsWorklist):
2  while !nodesWorklist.empty? do
3      while !nodesWorklist.empty? do
4          while !nodesWorklist.empty? do
5              processNode(nodesWorklist.pop())
6          end
7          for messageSend ∈ messageSendWorklist do
8              if satisfiedMessageSend?(messageSend) then
9                  processMessageSend(messageSend)
10             end
11         end
12     end
13     for messageSend ∈ messageSendWorklist do
14         processMessageSend(messageSend)
15     end
16 end
```

Algorytm 4: Pseudokod funkcji typującej pojedynczy wierzchołek

```
1  function processNode (node):
2  currentType = types[node]
3  types[node] = computeType(node, graph.incomingVertices(node))
4  if currentType ≠ types[node] then
5      for targetNode ∈ graph.adjacentVertices(node) do
6          nodesWorklist.push(targetNode)
7      end
8  end
```

Schemat we wszystkich jest podobny i polega na znalezieniu w bazie definicji metody, która została wywołana, a następnie połączeniu wierzchołków argumentów

wywołania z argumentami formalnymi oraz wierzchołka wyniku metody z wierzchołkiem wyniku wywołania. Dzięki temu że wszystkie argumenty są zdefiniowane, możemy tutaj przeprowadzić kilka dodatkowych heurystyk. Używamy specjalnych mechanizmów pilnujących, 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.

4.3. Heurystyki obsługi wywołań

Na początku sprawdzamy czy wywołanie nie pasuje do jednej z heurystyk opisanych poniżej (Alg. **??**). W przeciwnym przypadku zakładamy że to jest zwykłe wywołanie, które omówimy szczegółowo w następnej sekcji.

Algorytm 5: Pseudokod funkcji przetwarzającej wywołanie metody

```
1  function processMessageSend (messageSend):
2  for possibleType ∈ types[messageSend.sendObj].possibleTypes do
3      if messageAlreadyHandledByType?(messageSend, possibleType) then
4          next
5      end
6      messageName = messageSend.name;
7      if possibleType.is?(ClassType) and messageName = 'new' then
8          handleConstructorSend(possibleType.name, messageSend);
9      else if possibleType.is?(ProcType) and messageName = 'call' then
10         handleProcCall(possibleType, messageSend);
11     else if possibleType.is?(ClassType) then
12         handleClassSend(possibleType.name, messageSend);
13     else
14         handleInstanceSend(possibleType.name, messageSend);
15     end
16     markMessageSendAsHandled(messageSend, possibleType)
17 end
```

Po pierwsze, wywołanie może wyglądać jak wywołanie konstruktora, tj. jest wywoływana metoda `new` na typie reprezentującym klasę (Alg. **??**). Ten przypadek jest prawie taki sam jak obsłużenie zwykłego wywołania metody z tą różnicą, że zamiast zwrócić wynik konstruktora, dodajemy do grafu specjalny wierzchołek rodzaju `constructor` i podajemy mu nazwę inicjalizowanej klasy jako parametr (Alg. **??**). Dodajemy potem krawędź między tym wierzchołkiem, a wierzchołkiem wyniku metody. Obsługa tego specjalnego wierzchołka to pobranie parametru i zwrócenie go jako `GenericType` (jeśli nazwa klasy jest klasą generyczną `Array` lub `Hash`) lub

`NominalType` (Alg. **??**). Dzięki temu, zwracamy (jako wynik wywołania konstruktora) typ instancji zamiast typu klasowego na którym konstruktor był wywołany, tj. jeżeli konstruktor był wywołany na wierzchołku typu `ClassType(Foo)` to zwracamy typ `NominalType(Foo)`.

Algorytm 6: Pseudokod funkcji obsługującej wywołanie konstruktora

```
1 function handleConstructorSend (className, messageSend):
2 constructorMethod = database.findInstanceMethodFromClassName(className,
    'initialize');
3 if constructorMethod then
4 |   connectConstructorToCallResult(className, messageSend.resultNode);
5 else
6 |   handleRegularMessageSend(constructorMethod, messageSend);
7 |   connectConstructorToCallResult(className, messageSend.resultNode);
8 end
```

Algorytm 7: Pseudokod funkcji dołączającej specjalny wierzchołek konstruktora

```
1 function connectConstructorToCallResult (className, resultNode):
2 constructorNode = Node('constructor', name: className);
3 graph.addEdge(constructorNode, resultNode);
4 nodesWorklist.push(constructorNode);
```

Algorytm 8: Fragment funkcji typującej obsługującej konstruktor

```
1 function computeType (node, adjacentNodes):
2 ...
3 if node.kind = 'constructor' then
4 |   if node.params.name = 'Array' then return GenericType('Array');
5 |   else if node.params.name = 'Hash' then return GenericType('Hash');
6 |   else return NominalType(node.params.name);
7 end
8 ...
```

Drugim przypadkiem jest kiedy obiekt na którym wywołujemy metodę jest typu `ProcType`, a nazwa metody to `call`. Oznacza to wywołanie funkcji anonimowej. W `ProcType` trzymamy identyfikator funkcji anonimowej, wystarczy go więc tylko pobrać i wyciągnąć z bazy definicję funkcji (Alg. **??**). Reszta procedury jest taka jak w przypadku zwykłego wywołania metody.

Trzeci przypadek to metoda klasowa (Alg. **??**). Uznajemy wywołanie za wywołanie metody klasowej, jeśli obiekt na którym wywołujemy metodę jest pewnego typu klasowego, tj. `ClassType`. W tym przypadku jedyną różnicą od zwykłego wywołania jest zapytanie do bazy danych, które szuka metod zdefiniowanych na eigenclass, a nie w definicji klasy.

Algorytm 9: Pseudokod funkcji obsługującej wywołanie funkcji anonimowej

```
1 function handleProcCall (lambdaType, messageSend):
2 foundLambda = database.getLambda(lambdaType.lambdaId);
3 connectActualArgsToFormalArgs(messageSend.args, foundLambda.args);
4 graph.addEdge(foundLambda.resultNode, messageSend.resultNode);
5 nodesWorklist.push(messageSend.resultNode);
```

Algorytm 10: Pseudokod funkcji obsługującej wywołanie metody klasowej

```
1 function handleClassSend (className, messageSend):
2 foundMethod = database.findClassMethodFromClassName(className,
    messageSend.name);
3 if foundMethod then
4 |   handleRegularMessageSend(foundMethod, messageSend);
5 |   connectMethodResultToResultNode(foundMethod.nodes,
  |     messageSend.resultNode);
6 end
```

Tak jak zostało wspomniane wyżej, jeżeli żaden z tych przypadków nie zachodzi, uznajemy wywołanie za klasyczne wywołanie metody na instancji pewnego obiektu (Alg. **??**). Szukamy w takim wypadku metody o odpowiadającej do wywołania nazwie, która została zdefiniowana w klasie obiektu na którym metoda jest wywoływana.

Algorytm 11: Pseudokod funkcji obsługującej wywołanie metody instancji

```
1 function handleInstanceSend (className, messageSend):
2 foundMethod = database.findInstanceMethodFromClassName(className,
    messageSend.name);
3 if foundMethod then
4 |   handleRegularMessageSend(foundMethod, messageSend);
5 |   connectMethodResultToResultNode(foundMethod.nodes,
  |     messageSend.resultNode);
6 end
```

4.4. Obsługa wywołania

Po powyższych heurystykach możemy założyć, że mamy już znalezioną definicję metody, do której chcemy podłączyć wywołanie. Definicja, niezależnie od tego czy była to klasyczna definicja metody czy definicja funkcji anonimowej, ma obowiązkowo kilka właściwości: drzewo argumentów formalnych, wierzchołek wyniku

definicji, listę użyć bezargumentowego super, listę użyć słowa kluczowego `yield` (tj. wywołań bloków) oraz być może wierzchołek wywoływacza (wierzchołek wywoływacza jest specjalnym mechanizmem, opisanym dokładniej w sekcji poświęconej wywołaniom polimorficznym).

Najpierw następuje dodanie krawędzi między wierzchołkiem obiektu, na którym jest wywoływana metoda, oraz specjalnego wierzchołka rodzaju `caller` dla tej definicji metody, o ile on istnieje.

Algorytm 12: Pseudokod funkcji obsługującej łączenie wierzchołków metody z wierzchołkami wywołania

```
1  function handleRegularMessageSend (foundMethod, messageSend):
2  nodes = foundMethod.nodes;
3  if nodes.caller then
4  |  graph.addEdge(messageSend.sendObj, nodes.caller);
5  |  nodesWorklist.push(nodes.caller);
6  end
7  connectActualArgsToFormalArgs(messageSend.args, foundMethod.args);
8  for zsuperCall ∈ nodes.zsupers do
9  |  superMethod = database.findSuperMethod(foundMethod.id);
10 |  if not superMethod then next;
11 |  connectActualArgsToFormalArgs(messageSend.args, superMethod.args);
12 |  if zsuperCall.block then
13 |  |  connectYieldsToBlock(superMethod.nodes.yields, zsuperCall.block);
14 |  end
15 |  connectYieldsToBlock(superMethod.nodes.yields, messageSend.block);
16 |  connectMethodResultToResultNode(superMethod, zsuperCall.resultNode);
17 end
18 connectYieldsToBlock(foundMethod.nodes.yields, messageSend.block);
```

Najważniejsza część to wspomniane już połączenie wierzchołków argumentów wywołania z wierzchołkami argumentów formalnych, oraz wierzchołka wyniku metody z wierzchołkiem wyniku wywołania. Są dwa powody, które utrudniają zrobienie poprawnego dopasowania argumentów wywołania i argumentów formalnych.

Pierwszym jest tzw. *splat argument*. Zarówno w argumentach wywołania (Rys. **??**) jak i argumentach formalnych (Rys. **??**), możemy użyć składni *splat*, która w przypadku argumentów wywołania spłaszcza przekazaną listę jako listę argumentów, a w przypadku argumentów formalnych buduje tablicę z przekazanych argumentów. Jest to mechanizm znany z wielu języków pozwalający na definiowanie i wywoływanie funkcji o zmiennej liczbie argumentów.

Drugi to *named arguments* czyli argumenty, które w wywołaniu muszą być przekazane razem z nazwą argumentu formalnego, ale dzięki temu mogą być przekazane w dowolnej kolejności. Niestety, jest to tylko pewna sztuczka głównie po stronie parsera. Mechanizm *named arguments* w Ruby polega na tym, że jeżeli ostatni przekazywany argument jest słownikiem, to jest traktowany jako słownik zawierający

WYWOŁANIE	DEFINICJA
`obj.foo(*some_list)` `obj.foo(y, *some_list)`	`def foo(a,b=5)`

Rysunek 4.1: Trudność w dopasowaniu argumentów gdy splat argument użyty w argumentach wywołania. `some_list` jest listą argumentów o nieznanej długości.

WYWOŁANIE	DEFINICJA
`obj.foo(x, y)`	`def foo(*args)`

Rysunek 4.2: Trudność w dopasowaniu argumentów gdy splat argument użyty w argumentach formalnych. `some_list` jest listą argumentów o dowolnej długości.

przekazywane *named arguments*. Z tego powodu, przekazywane argumenty wcale nie muszą być przekazywane explicit, ale mogą być przekazane np. przez zwykłą zmienną, która jest słownikiem (Rys. **??**). Dlatego też aby poprawnie dopasować przekazane *named arguments* do tych w definicji wywołania, musielibyśmy śledzić dokładne mapowanie typów w każdym słowniku.

WYWOŁANIE	DEFINICJA
`obj.foo(**some_hash)`	`def foo(x:,y:)`

Rysunek 4.3: Trudność w dopasowaniu argumentów gdy *named arguments* pobrane z nieznanego słownika.

Trzecim etapem jest połączenie przekazanych argumentów do metody z klasy nadrzędnej oraz wyników tej metody z wszystkimi wywołaniami bezargumentowego `super`.

Kolejnym krokiem jest obsługa bloku (Alg. **??**). Blok mamy przypisany do wywołania albo w postaci wierzchołka z identyfikatorem funkcji anonimowej (tj. przekazany blok był znany już przy budowaniu grafu), albo wierzchołka grafu, który został otypowany na `ProcType` i zawiera w sobie identyfikator funkcji anonimowej. Następuje więc przejście po wszystkich użyciach słowa kluczowego `yield` w tej metodzie i połączeniu przekazanych do `yield` argumentów z argumentami formalnymi znalezionej funkcji anonimowej oraz wierzchołka wyniku funkcji anonimowej z wierzchołkiem wyniky `yield`. Jest to procedura identyczna do opisanego wyżej łączenia argumentów wywołania z argumentami formalnymi.

Na końcu, następuje połączenie wierzchołka wyniku metody oraz wierzchołka

Algorytm 13: Pseudokod obsługi bloku podczas wywołania

```
1  function connectYieldsToBlock (yieldNodes, blockNode):
2  for yieldNode ∈ yieldNodes do
3  |  lambdaIds = lambdaIdsOfBlock(blockNode);
4  |  for lambdaId ∈ lambdaIdsOfBlock(blockNode) do
5  |  |  blockLambda = database.getLambda(lambdaId);
6  |  |  connectActualArgsToFormalArgs(yieldNode.args, blockLambda.args);
7  |  |  graph.addEdge(blockLambda.resultNode, yieldNode.resultNode);
8  |  |  nodesWorklist.push(yieldNode.resultNode);
9  |  end
10 end
```

wyniku wywołania.

4.5. Wywołania funkcji polimorficznych

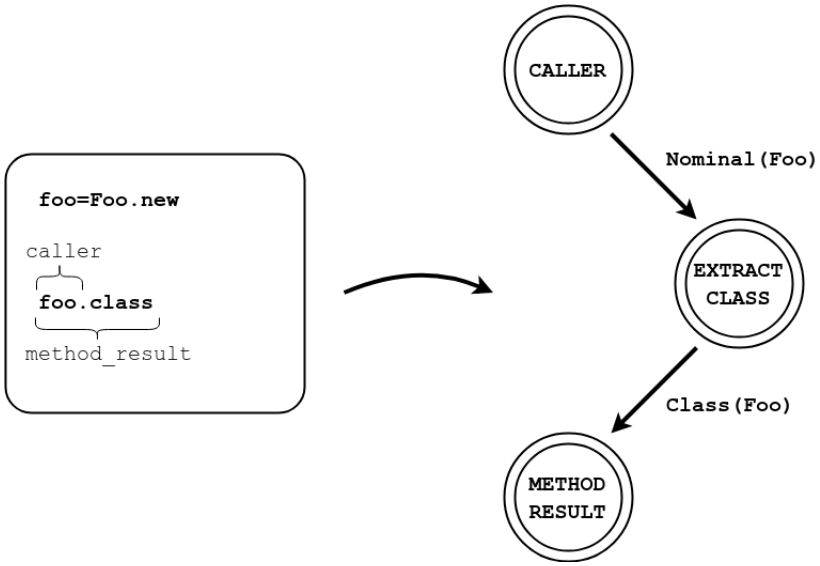
Jak już zostało wspomniane, definicja metody w Bazie ma w sobie kilka wierzchołków, które są istotne z punktu widzenia analizy, takie jak wierzchołek wyniku metody, wierzchołki argumentów i inne związane z blokami i wywołaniami bezargumentowego `super`. Te wierzchołki mogą zostać przechowywane na dwa sposoby. Pierwszy z nich to po prostu dokładnie te wierzchołki, które są obecne w grafie. Drugi sposób to przechowywanie szkicu podgrafu (tj. wierzchołki i krawędzie, które nie należą do grafu – są tylko wzorem), z którego przy okazji przetwarzania każdego wywołania są tworzone świeże wierzchołki.

Świeże wierzchołki dla każdego wywołania oznaczają też, że możemy skorzystać z typu wyrażenia na którym metoda została wywołana. Dlatego też, jeżeli obsługujemy wywołanie metody której definicja korzysta z podgrafu, łączymy wierzchołek wyrażenia na którym została wywołana metoda z wierzchołkiem rodzaju `caller`. Dzięki temu możemy bez problemu zbudować podgrafy działające poprawnie dla takich metod polimorficznych jak `map` z `Array` lub po prostu funkcje, które zwracają obiekt na którym zostały wywołane (nie było to możliwe bez podgrafu), takie jak np. `freeze`.

4.6. Inne wierzchołki specjalne

Podejście z podgrafami pozwala też na obsługę niektórych metod w specjalny sposób, bez dużej ingerencji w funkcję typującą. Przykładem takiej metody jest metoda `class`, którą możemy wywołać na dowolnym obiekcie, a która zwraca klasę tego obiektu. Dzięki podgrafowi, definicja takiej metody to tylko dwie krawędzie, widoczne na rysunku **??**. Specjalny wierzchołek `extract_class` na podstawie typu wierzchołka z krawędzi wejściowej zwraca odpowiedni typ, który zwróciłaby metoda

`class`. Na przykład, jeżeli typ wejściowy to `NominalType(Foo)`, to typem wyjściowym będzie `ClassType(Foo)`.



Rysunek 4.4: Wykorzystanie mechanizmu subgrafów na przykładzie metody `class`.

4.7. Pozostałe wierzchołki

Pozostałe wierzchołki są obsługiwane w bardzo prosty sposób. Część z nich zwraca ustalony typ, np. typ `NominalType('String')` dla wierzchołka rodzaju `str`. Inne robią prostą operację na jednym z wejściowych typów, jak opakowanie wejściowych typów w typ generyczny tablicy. Pozostałe, po prostu biorą wszystkie wejściowe wierzchołki i tworzą zbiorczy typ `UnionType` z typów wierzchołków wejściowych.

Rozdział 5.

Podsumowanie

Głównym celem niniejszej pracy było podjęcie próby zaimplementowania wtyczki wspomagającej pracę programisty Ruby. Opracowany projekt spełnia ten cel przez dostarczenie funkcjonalności skoku do definicji i autouzupełniania, a autor tej pracy korzysta z tego projektu w codziennej pracy.

Pomimo dynamicznej natury języka Ruby udało się stworzyć narzędzie, które z zadowalającym efektem potrafi wywnioskować przybliżony typ wyrażeń, korzystając z grafu przepływu danych i heurystyk. Posiadanie tej informacji pozwoliłoby również zaimplementować szereg innych dodatkowych funkcjonalności obecnych w komercyjnych rozwiązaniach.

Największą wadą obecnego projektu jest konieczność uruchomienia od nowa funkcji typującej w przypadku zmiany kodu źródłowego. Ponieważ w praktyce programista spędza znacznie więcej czasu czytając kod, niż go pisząc, nie jest to wada czyniąca wtyczkę bezwartościową, ale utrudnia ona implementowanie niektórych funkcjonalności. Jednocześnie stworzenie takiego inkrementalnego algorytmu typującego byłoby zdecydowanie najciekawszym kierunkiem rozwoju tego projektu.

Drugą wadą projektu jest efektywność – cały projekt, jako projekt badawczy, został napisany w Ruby, czyli języku o wysokiej ekspresji, ale jednak języku interpretowanym i ze słabym wsparciem wielowątkowości. Oznaczało to, że przy bardzo dużych projektach indeksowanie potrafiło zająć kilkanaście minut. Projekt jest głównie ograniczony przez CPU, więc przepisanie go do dowolnego języka kompilowanego powinno dać znaczny skok wydajności.

Przez to, że projekt opiera się na heurystykach, jest wiele możliwości jego rozwoju, które pozwalałyby jeszcze dokładniej określać typ wyrażeń. Ponadto, sam system typów mógłby zostać rozbudowany, na przykład wprowadzając specjalne typy listy i krotek, zależnie od rodzaju inicjalizacji tablicy, podobnie jak to robi RDL [7]. Jeszcze innym, bardziej inżynieryjnym kierunkiem byłaby integracja z niektórymi dotychczasowymi narzędziami popularnymi w społeczności języka Ruby, takimi jak system dokumentacji YARD [12]. Integracja z systemami dokumentacji (które

często posiadają jakiś język opisu typów) pozwoliłoby również na szczątkową implementację silnego typowania, informującą programistę kiedy wywołanie funkcji jest używane z argumentami niepoprawnego typu.

Appendices

Dodatek A

Instalacja i uruchomienie

A1. Instalacja języka Ruby

Aby zainstalować język Ruby na systemie Linux najlepiej skorzystać z narzędzia `rbenv` wraz z dodatkiem `ruby-build`. `rbenv` służy do zarządzania zainstalowanymi interpreterami języka Ruby, a `ruby-build` do łatwego skompilowania dowolnej wersji interpretera ze źródeł.

Aby zainstalować `rbenv` można skorzystać z instalatora automatycznego dostępnego pod adresem: `https://github.com/rbenv/rbenv-installer#rbenv-installer`, dodatek `ruby-build` zostanie zainstalowany automatycznie.

Drugą opcją jest instalacja ręczna, dostępna pod adresem `https://github.com/rbenv/rbenv#basic-github-checkout`.

Rekomendowaną wersją ruby do instalacji jest `2.5.1`, jednak projekt powinien działać dobrze na każdej wersji od 2.3 wzwyż:

```
$ rbenv install 2.5.1
$ rbenv global 2.5.1
```

Komenda `rbenv global` służy do ustawienia domyślnej wersji interpretera.

A2. Pobranie projektu

Aby pobrać projekt wystarczy sklonować jego repozytorium GIT:

```
$ git clone https://github.com/swistak35/orbacle.git
```

A3. Instalacja projektu – wersja automatyczna

W internetowej bazie bibliotek języka Ruby jest dostępna jedna z wersji projektu. Można ją zainstalować korzystając z następującego polecenia:

```
$ gem install orbacle --version 0.2.0
$ rbenv rehash
$ orbaclerun --help
...
```

A4. Instalacja projektu – wersja z repozytorium

Najpierw należy pobrać projekt z repozytorium, tak jak zostało to opisane w sekcji **??**.

Następnie należy zainstalować bibliotekę `bundler` do zarządzania zależnościami w projektach:

```
$ gem install bundler --version 1.16.4
...
$ rbenv rehash
$ bundler --version
Bundler version 1.16.4
```

Następnie należy wejść do katalogu projektu. W projekcie jest dostępne polecenie `make full`, które instaluje zależności, buduje paczkę i instaluje ją w systemie.

```
$ cd orbacle
$ make full
...
$ rbenv rehash
$ orbaclerun --help
...
```

A5. Konfiguracja w VIM

Jednym z edytorów w którym można użyć tego projektu jest `vim`. Dostępna jest gotowa konfiguracja której można użyć. Jest istotne, aby pobrać ją do katalogu `/.vim-orbacle`.

```
$ git clone https://github.com/swistak35/msc-demo.git ~/.vim-orbacle
```

Następnie można uruchomić edytor VIM z konfiguracją z tego repozytorium. Projekt jest napisany w Ruby, więc może być przykładowym projektem na którym można przetestować działanie wtyczki.

```
$ cd sciezka_do_projektu
$ vim -u ~/.vim-orbacle/vimrc
```

Dostępne są następujące skróty klawiszowe:

- `J` – skok do definicji (stałej, funkcji lub zmiennej)
- `T` – informacja o typie zmiennej
- `Ctrl+X Ctrl+O` – autouzupełnianie pełnej nazwy aktualnie pisanej nazwy funkcji

Bibliografia

[1] *Atom*. URL: https://atom.io (term. wiz. 01. 10. 2018).

[2] *CTags project*. URL: http://ctags.sourceforge.net (term. wiz. 01. 10. 2018).

[3] Greg DeFouw, David Grove i Craig Chambers. *Fast Interprocedural Class Analysis*. 1997.

[4] *GNU/Emacs*. URL: https://www.gnu.org/software/emacs/ (term. wiz. 01. 10. 2018).

[5] *Język programowania Ruby*. URL: https://www.ruby-lang.org/pl/ (term. wiz. 01. 10. 2018).

[6] *Language Server Protocol*. URL: https://microsoft.github.io/language-server-protocol/ (term. wiz. 01. 10. 2018).

[7] *RDL – Types, type checking, and contracts for Ruby*. URL: https://github.com/plum-umd/rdl (term. wiz. 01. 10. 2018).

[8] *Robe*. URL: https://github.com/dgutov/robe (term. wiz. 01. 10. 2018).

[9] *RubyMine: Ruby on Rails IDE by JetBrains*. URL: https://www.jetbrains.com/ruby/ (term. wiz. 01. 10. 2018).

[10] *Solargraph*. URL: https://github.com/castwide/solargraph (term. wiz. 01. 10. 2018).

[11] *Vim – the ubiquitous text editor*. URL: https://www.vim.org (term. wiz. 01. 10. 2018).

[12] *Yay! A Ruby Documentation Tool*. URL: https://yardoc.org (term. wiz. 01. 10. 2018).