POLITECHNIKA POZNAŃSKA

Iwo Błądek

Automatyczna synteza programów w konwencjonalnych językach programowania

Praca magisterska

Promotor: dr hab. inż. Krzysztof Krawiec, prof. nadzw.

Poznań, 2015

Spis treści

Rozdział 1

Wstęp

Zdolność do formułowania wysokopoziomowych algorytmów postępowania w celu rozwiązywania problemów jest jedną z charakterystycznych cech człowieka. Jednym z głównych celów sztucznej inteligencji jest przeniesienie tej zdolności na maszyny, kierunek badań znany pod nazwą *silnej sztucznej inteligencji*. Pomimo wielu wysiłków i postępów na tym polu, wciąż nie udało się stworzyć ogólnej sztucznej inteligencji dorównującej człowiekowi. Udało się za to, z dużym powodzeniem, zastosować metody sztucznej inteligencji i uczenia maszynowego do zawężonych klas problemów.

Algorytmy postępowania, przypuszczalnie nawet te realizowane przez tak złożone układy obliczeniowe jak mózg ludzki, można uogólnić jako programy. Automatyczna synteza programów, której poświęcona jest ta praca, jest gałęzią szeroko pojmowanej sztucznej inteligencji, która zajmuje się badaniem możliwości tworzenia i uzupełniania programów metodami algorytmicznymi. Można ją więc traktować, pomimo fundamentalnych różnic w sposobie realizacji wyznaczonego celu, jako w pewnym sensie analogiczną do ogólnej sztucznej inteligencji.

Systemy automatycznej syntezy programów mają wiele potencjalnych praktycznych zastosowań, spośród których najistotniejsze są: umożliwienie programowania zwykłym (tj. niezaawansowanym, bez wykształcenia informatycznego) użytkownikom, wspomaganie programisty w pracy i wykorzystanie do rozwiązywania problemów badawczych. Wciąż jednak zintegrowane środowiska tworzenia aplikacji, *IDE* (ang. *Integrated Development Environment*), nie umożliwiają programistom korzystania z tych możliwości. Tworzone póki co prototypowe, badawcze systemy syntezy operują w większości na sztucznie stworzonych językach, takich jak np. Push. Systemy syntezy operujące na powszechnie stosowanych przez programistów językach, które nazywane będą w tej pracy *konwencjonalnymi językami programowania*, mają wiele ograniczeń lub potrzebują dużo czasu na uzyskanie satysfakcjonujących rozwiązań. Wiele ośrodków na świecie pracuje obecnie nad tą technologią i jest duża szansa, że problemy te zostaną rozwiązane w przyszłości.

1.1 Cel i zakres pracy

Ogólnym celem pracy było zbadanie obecnych możliwości automatycznej syntezy programów w konwencjonalnych językach programowania i zaprojektowanie własnego systemu tego typu. Z racji wielości różnych podejść do syntezy programów i możliwych form

informacji określającej cel syntezy, w pracy tej ograniczono się do syntezy programów przy użyciu programowania genetycznego na podstawie specyfikacji w postaci par wejście-wyjście. W szczególności oznacza to, że programy syntetyzowane są od podstaw, czyli bez wstępnego rozwiązania dostarczonego przez użytkownika.

Cele szczegółowe, które podjęte zostały w realizacji tej pracy, to:

- zapoznanie się z dotychczasowymi pracami dotyczącymi syntezy programów za pomocą programowania genetycznego w konwencjonalnych językach programowania,
- implementacja prototypowego systemu syntezy zdolnego do generowania programów w języku programowania Scala [25],
- zbadanie skuteczności tego systemu na zbiorze wybranych problemów testowych.

1.2 Struktura pracy

Struktura pracy jest następująca. W rozdziale 2 omówione zostaną podstawowe pojęcia związane z syntezą programów i programowaniem genetycznym. Rozdział 3 zawiera opis aktualnie istniejących rozwiązań pozwalających na syntezę programów w konwencjonalnych językach programowania. W rozdziale 4 omówione są mechanizmy refleksji w języku Scala. W rozdziale 5 opisane są ogólne założenia autorskiego systemu syntezy. Najważniejsze zagadnienia związane z jego implementacją zostały z kolei opisane w rozdziale 6. W rozdziale 7 przedstawiona jest skuteczność zaimplementowanego systemu na wybranym zbiorze problemów. Rozdział 8 stanowi podsumowanie pracy.

1.3 Podziękowania

Praca zrealizowana w ramach projektu 2014/15/B/ST6/05205 „Skalowalne metaheurystyki dla automatycznej syntezy programów” przyznanego przez Narodowego Centrum Nauki.

Rozdział 2

Podstawy teoretyczne

2.1 Automatyczna synteza programów

Program można ogólnie zdefiniować jako zapis pewnej procedury algorytmicznej, który może zostać wykonany na odpowiednio zaprojektowanej maszynie. Celem wykonania programu najczęściej jest przeprowadzenie szeregu operacji na danych wejściowych w celu uzyskania wyniku. Można znaleźć wiele analogii pomiędzy pojęciem programu a pojęciem funkcji w matematyce, takich jak chociażby deterministyczne (w większości praktycznych przypadków) mapowanie wejścia na wyjście. Program jest jednak pojęciem bardziej ogólnym, gdyż pozwala między innymi na efekty uboczne, czyli zmiany stanu środowiska wykonania inne, niż zwracany wynik.

Zadaniem automatycznej syntezy programów jest wygenerowanie programu w określonym z góry *języku programowania* realizującego zadaną *specyfikację*. Specyfikacja to dowolna informacja, która zwiększa naszą wiedzę o zadaniu realizowanym przez program. Informacja ta jest, w całości bądź częściowo, wykorzystywana do kierowania procesu poszukiwania programu i sprawdzania poprawności już wygenerowanych programów. Specyfikacja może przyjmować różne formy, z których najczęściej wykorzystywane to: pary wejście-wyjście, związki logiczne między wejściem a wyjściem, demonstracja wykonywania kroków, opis funkcjonalności w języku naturalnym.

Proces syntezy można traktować jako przeszukiwanie przestrzeni wszystkich możliwych programów w celu znalezienia programu poprawnego, to znaczy takiego, który jest całkowicie zgodny ze specyfikacją. W niektórych zastosowaniach, np. przy problemach regresji symbolicznej, osłabia się to założenie i dopuszcza się programy, które zwracają wyniki zbliżone do optymalnych. Pojedynczą specyfikację może teoretycznie realizować wiele programów, tak więc jej odpowiednie sformułowanie w celu wyeliminowania niejednoznaczności jest bardzo istotne.

Programy, poza kryterium poprawności, można oceniać również pod względem szeregu innych kryteriów np. długości kodu źródłowego lub czasu wykonania. W ogólności problem syntezy jest problemem wielokryterialnym. W tej pracy rozważane będzie jednak tylko jedno kryterium, którym jest zgodność wygenerowanych programów ze specyfikacją.

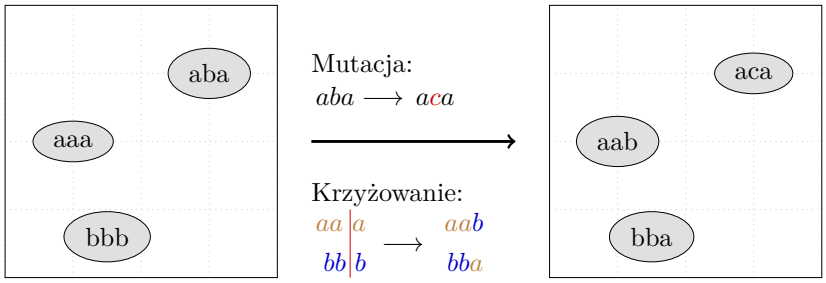
2.1.1 Zastosowania

Automatyczna synteza programów ma wiele potencjalnych zastosowań. Najważniejsze z nich zostały wymienione i pokrótce opisane w poniższych punktach.

- **Wspomaganie programisty w pracy.** Pomoc ta może odbywać się na wiele sposobów m.in. poprzez generowanie fragmentów kodu na podstawie testów oraz automatyczne wyszukiwanie i poprawianie błędów [35]. Bardzo ciekawa koncepcja, *programowanie przez szkicowanie*, przedstawiona została w pracach [30, 31]. Polega ono na tym, że programista pisząc program może w nim zamiast części kodu umieszczać specjalne znaczniki wskazujące na luki w implementacji. Tak napisany program można traktować jako szkic całości (i stąd nazwa), który zostanie wypełniony w następnym kroku przez syntezator przy uwzględnieniu dodatkowej informacji mówiącej o tym, co dane fragmenty mają realizować.
- **Umożliwienie użytkownikom bez specjalistycznej wiedzy automatyzację wykonywanych czynności.** Użytkownicy komputerów często stoją przed koniecznością wielokrotnego powtarzania takich samych bądź bardzo podobnych czynności. Takie czynności relatywnie łatwo można by zaprogramować, jednak obecnie większość użytkowników komputerów nie posiada do tego odpowiednich umiejętności. Rozwiązaniem problemu może być system automatycznej syntezy, któremu użytkownik zada albo bardzo prostą specyfikację (np. w języku naturalnym) albo zaprezentuje kolejne kroki do wykonania. Wdrożony do użytku program FlashFill [13], wchodzący w skład pakietu MS Office 2013, pozwala użytkownikom zamiast pisania formuły przekształcającej dane podać wyniki dla kilku przykładów, które są następnie wykorzystywane do indukcji formuły transformującej pozostałe dane.
- **Pomoc przy rozwiązywaniu problemów badawczych.** Istnieją problemy, które z racji swej złożoności sprawiają problem ludzkim ekspertom albo wymagają od nich dużo czasu. Synteza programów może dla niektórych z tych problemów zostać użyta do znalezienia wstępnych, albo nawet ostatecznych, rozwiązań. Z metod syntezy zdecydowanie najbardziej znane na tym polu jest programowanie genetyczne, w którym odniesiono sukcesy między innymi w wytwarzaniu programów dla komputerów kwantowych [23] albo układów elektronicznych [20]. Słynny w tym kontekście jest przykład anteny o wyjątkowo dobrych właściwościach wyewoluowanej przy pomocy algorytmu genetycznego dla mikrosatelity Space Technology 5 zbudowanej przez NASA [21].

2.2 Algorytmy ewolucyjne

Algorytmy ewolucyjne (ang. *Evolutionary algorithms, EA*) to metaheurystyka inspirowana ewolucją biologiczną, służąca do rozwiązywania problemów optymalizacyjnych. Według jej założeń, przez czas działania algorytmu utrzymywana jest populacja osobników będących rozwiązaniami problemu. Każdy osobnik podlega ocenie zgodności z celem optymalizacji i ma przypisywany odpowiedni *fitness*, który jest miarą jakości osobników i pozwala je między sobą porównywać. W każdej iteracji na podstawie starej populacji



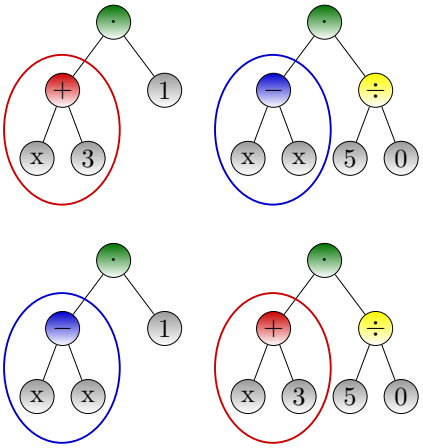
Rysunek 2.1: Ilustracja działania operatorów mutacji i krzyżowania wykorzystywanych w standardowym algorytmie ewolucyjnym. Po lewej stronie znajduje się stara populacja, a po prawej nowa, wypełniona zmodyfikowanymi rozwiązaniami ze starej.

tworzona jest nowa. Osobniki do nowej populacji wybierane są zgodnie z pewną regułą selekcji np. regułą turniejową, w której losowane ze zwracaniem jest k osobników i osobnik z najwyższym fitnessem dodawany jest do nowej populacji po wykonaniu na nim pewnego, losowo wybranego, operatora ewolucji. Celem stosowania tych operatorów jest realizacja przeszukiwania przestrzeni wszystkich rozwiązań.

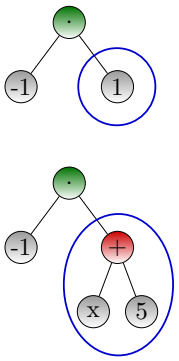
W standardowym algorytmie ewolucyjnym definiuje się dwa takie operatory: mutację, polegającą na losowej zamianie symbolu w reprezentacji rozwiązania, i krzyżowanie, polegające na wymianie fragmentów genotypu między dwoma wyselekcjonowanymi rozwiązaniami. Na Rys. 2.1 znajduje się ilustracja działania omówionych powyżej operatorów.

2.3 Programowanie genetyczne

Programowanie genetyczne (ang. *Genetic Programming, GP*), które spopularyzował John Koza poczynając od pracy [17], jest zasadniczo szczególnym przypadkiem algorytmu ewolucyjnego, w którym ewolucji podlegają programy. Ewolucja programów pozwala ewoluować metody rozwiązywania całych klas problemów, stąd też istnieje duże zainteresowanie tą dziedziną wiedzy. Ogólny przegląd wyników uzyskanych w dziedzinie GP można znaleźć w [27].



Rysunek 2.2: Działanie operatora krzyżowania w standardowym GP.



Rysunek 2.3: Działanie operatora mutacji w standardowym GP.

W standardowym algorytmie programowania genetycznego programy reprezentowane są jako drzewa. Stałe lub zmienne, nazywane *terminalami*, są w tym drzewie liśćmi, podczas gdy funkcje przyjmujące argumenty, nazywane *nieterminalami*, są węzłami wewnętrznymi drzewa.

Drzewo programu może zostać „wykonane" dla pewnych danych wejściowych. Ewaluacja drzewa postępuje od liści do korzenia (ang. *bottom-up*) – by obliczyć wartość zwracaną przez dany węzeł nieterminalny, musimy najpierw obliczyć wartości jego argumentów (poddrzew). Warto w tym miejscu wspomnieć, że w programowaniu genetycznym reprezentacja programów nie jest odgórnie narzucona – poza reprezentacją drzewiastą istnieją również inne reprezentacje, z których najbardziej znane to liniowa i grafowa.

2.3.1 Operatory przeszukiwania

Programowanie genetyczne opiera się na tych samych zasadach, co algorytmy ewolucyjne. Dlatego więc również tutaj określone są operatory przeszukiwania przestrzeni rozwiązań: krzyżowanie i mutacja. Są to, w swoim „standardowym" ujęciu i różnych wariantach, zdecydowanie najczęściej stosowane operatory i wykorzystywane są też w zaimplementowanym systemie syntezy. Przegląd operatorów przeszukiwania stosowanych w GP można znaleźć w [27].

Działanie standardowego krzyżowania i mutacji w GP zostanie przedstawione na przykładzie reprezentacji drzewiastej. Krzyżowanie jest dla takiego przypadku operacją zdefiniowaną na dwóch osobnikach i polega na wymianie między nimi losowych poddrzew. Proces ten przedstawiony jest wizualnie na Rys. 2.2. Mutacja z kolei polega na losowym wybraniu jednego węzła w drzewie i zastąpieniu go, wraz z jego potomkami, losowo wygenerowanym poddrzewem, co pokazane jest na Rys. 2.3. Wybór operatora do zastosowania na aktualnie wybranych osobnikach jest losowy i zależny od parametrów ustalonych przez użytkownika podczas inicjalizacji algorytmu.

2.3.2 Inicjalizacja populacji

W GP znane są pewne strategie tworzenia populacji początkowej, które zyskały popularność w środowisku badaczy zajmujących się tą tematyką. Najbardziej znane spośród nich zostały opisane przez Johna Kozę [18], i są to:

full – węzły wybierane są losowo, do momentu osiągnięcia limitu głębokości, ze zbioru nieterminali, a następnie wyłącznie ze zbioru terminali.

grow – węzły wybierane są losowo, do momentu osiągnięcia limitu głębokości, zarówno ze zbioru nieterminali jak i terminali, a następnie wyłącznie ze zbioru terminali.

ramped half-and-half – połowa populacji generowana jest za pomocą *full*, a połowa za pomocą *grow*, dla różnych limitów głębokości (dlatego *ramped*).

Drzewa wygenerowane metodą *full* mają stosunkowo mało zróżnicowany kształt (jak nazwa wskazuje, są pełne) i wszystkie liście są na dokładnie tej samej głębokości. Metoda *grow* generuje o wiele bardziej różnorodne drzewa, jednak ich kształty są mocno zależne od liczności zbiorów terminali i nieterminali – jeżeli zbiór terminali jest o wiele większy od

zbioru nieterminali, to drzewa będą miały tendencję do niskiej wysokości. *Ramped half-and-half* jest kompromisem pomiędzy tymi dwoma metodami i pozwala osiągnąć najbardziej zdywersyfikowane populacje [18].

2.3.3 Selekcja osobników

Istnieje wiele metod wyboru osobników do reprodukcji w GP. Wybór ten ma duży wpływ na działanie algorytmu. Przykładowe metody selekcji to:

Proporcjonalna – każdemu osobnikowi przydzielone jest prawdopodobieństwo wylosowania takie jak jego fitness podzielony przez sumaryczny fitness wszystkich osobników z populacji. Następnie dokonywane jest losowanie osobnika zgodnie z powstałym rozkładem.

Turniejowa – za każdym razem, kiedy ma zostać wybrany jakiś osobnik, z populacji losowane jest k osobników, gdzie k jest rozmiarem turnieju, i spośród nich wybierany do reprodukcji jest ten z najwyższą wartością fitnessu.

Lexicase – w miarę młoda metoda selekcji opisana w pracach [16, 32]. Podczas wyboru każdego nowego osobnika tworzona jest losowa permutacja P zbioru testów oraz zbiór dopuszczalnych osobników D zawierający początkowo całą populację. Testy brane są pod uwagę zgodnie z kolejnością określoną przez P. Dla aktualnie rozpatrywanego testu z P w zbiorze D pozostawiane są te osobniki z tego zbioru, które osiągają na tym teście najlepszy wynik, po czym rozpatrywany jest następny test. Jeżeli w którymś momencie w zbiorze D pozostanie jeden osobnik, to jest on wybierany. Jeżeli po rozpatrzeniu wszystkich testów zbiór D zawiera więcej niż jeden element, to wybierany jest losowy osobnik z tego zbioru.

Zaletą tej metody jest to, że wspiera rozwiązania, które dobrze rozwiązują konkretne testy, dzięki czemu nie są one tak zdominowane przez rozwiązania „kompromisowe". Ma to szczególne znaczenie przy problemach, w których interesujące są tylko rozwiązania w pełni poprawne.

2.3.4 Strongly Typed GP

W standardowym GP wszystkie zmienne, stałe, wartości zwracane przez funkcje oraz przyjmowane przez nie argumenty muszą być tego samego typu. Oznacza to, że każda funkcja musi być zdefiniowana dla dowolnych argumentów możliwych w dziedzinie problemu i musi zwrócić wynik również należący do tej dziedziny. Własność ta w algebrze nazywana jest *zamknięciem* ze względu na daną operację. W praktycznych zastosowaniach związane są z tym co najmniej dwa problemy.

Po pierwsze, kiedy ewoluowany program musi operować na fundamentalnie różnych obiektach, takich jak np. liczby lub ciągi znaków, wszystkie funkcje muszą mieć zdefiniowane wartości dla wszystkich możliwych kombinacji argumentów. Często można tego problemu uniknąć definiując funkcje w taki sposób, by zwracały określoną stałą wartość w przypadku niepożądanego rodzaju obiektu lub by osobnik zawierający niedozwoloną kombinację argumentów otrzymał minimalny fitness.

Nie rozwiązuje to jednak drugiego problemu, którym jest zwiększenie przestrzeni przeszukiwania o rozwiązania wykorzystujące tego typu, niepożądane według naszej eksperckiej wiedzy, kombinacje argumentów. Uwzględnianie informacji o różnej naturze przetwarzanych danych jest istotnym elementem wiedzy dziedzinowej o wykonywanych przez programy operacjach i pozwala znacznie zmniejszyć przestrzeń przeszukiwania, poprawiając tym samym w zdecydowanej większości przypadków osiągi algorytmu przeszukiwania.

W pracy [24] zaproponowany został po raz pierwszy wariant GP uwzględniający informację o typach węzłów. Wariant ten nazwany został jako *Strongly Typed GP* (*STGP*), co po polsku można by przetłumaczyć jako silnie typowane GP. Silne typowanie jest podobne koncepcyjnie do tego, które występuje w większości języków programowania.

W silnie typowanym GP dla każdego elementu (terminala, nieterminala) jego zwracany typ musi być określony przed rozpoczęciem algorytmu. Dla nieterminali dodatkowo określone muszą być typy przyjmowanych argumentów. Operatory ewolucji i inicjalizacja populacji zmodyfikowane są w taki sposób, by każda instrukcja będąca argumentem danego nieterminala zwracała typ odpowiadający typowi tego argumentu.

Rozdział 3

Przegląd powiązanych prac

3.1 Wprowadzenie

Celem tego rozdziału jest opisanie dotychczasowych zastosowań programowania genetycznego do automatycznej syntezy programów. Ograniczymy się przy tym do przykładów dotyczących albo wprost, albo pośrednio ewolucji programów w konwencjonalnych językach programowania, które są głównym punktem zainteresowania tej pracy. W sekcji 3.2 opisana zostanie pokrewna do dziedziny automatycznej syntezy dziedzina automatycznego ulepszania programów. Następnie omówiona zostanie synteza programów w paradygmatach programowania funkcyjnego (sekcja 3.3) i obiektowego (sekcja 3.4). Ponadto w sekcji 3.5 omówione zostaną prace dotyczące automatycznej syntezy programów w języku Scala.

3.2 Automatyczne ulepszanie programów

3.2.1 Wprowadzenie

Automatyczne ulepszanie programów jest dziedziną mocno powiązaną z syntezą programów. Podstawowa różnica polega na tym, że w zadaniu syntezy programy generowane są od podstaw, podczas gdy w zadaniu ulepszania użytkownik podaje pewien program wyjściowy. Autorski system syntezy zaproponowany w ramach pracy, a powstały we współpracy z Mateuszem Poszwą, może służyć do obu zadań. Niniejsza praca omawia wykorzystanie go do zadania syntezy. Omówienie zastosowania systemu do ulepszania programów opisał Mateusz Poszwa w swojej pracy magisterskiej [28]. Jakkolwiek ulepszanie programów jest raczej luźno powiązane tematycznie z niniejszą pracą, zainteresowany czytelnik znajdzie w dalszej części tej sekcji odniesienia do kilku wyników z tej dziedziny.

3.2.2 Omówienie wybranych prac

Ulepszanie może dotyczyć teoretycznie dowolnego kryterium. W przypadku praktycznych zastosowań jako pierwsze przychodzą na myśl kryteria poprawności i efektywności. Przykład systemu wyszukującego błędy i proponującego sposób ich poprawy w kodzie napisanym w języku C opisany jest w [35]. Algorytm rejestrował ścieżki w programie, przez które przechodziły wykonania niespełniające testów, i na drodze ewolucji próbował

je zmodyfikować w taki sposób, by testy zostały spełnione. Niedawno opublikowana praca [29] eksploruje z kolei możliwość wykorzystania istniejących repozytoriów do wyszukiwania fragmentów kodu, które mogą się okazać przydatne do poprawienia aktualnie analizowanego programu. W kontekście poprawy efektywności Langdon i inni osiągnęli przy pomocy GP ponad 35% poprawę jeżeli chodzi o szybkość działania programu realizującego obliczenia na kartach graficznych [19].

3.3 Synteza programów funkcyjnych

3.3.1 Wprowadzenie

Programowanie funkcyjne jest paradygmatem programowania, w którym podstawową jednostką programu jest funkcja. Podstawy teoretyczne języków funkcyjnych oparte są na *rachunku lambda*. Główne cechy programowania funkcyjnego to:

- swobodne używanie funkcji jako argumentów innych funkcji, czyli traktowanie ich tak jak każdej innej wartości w programie (ang. *functions as first class citizens*),
- możliwość definiowania anonimowych funkcji,
- unikanie, albo w przypadku czystych języków funkcyjnych brak operacji zmieniających stan programu,
- powszechne stosowanie niezmiennych, niemodyfikowalnych (ang. *immutable*) danych. „Modyfikacja" odbywa się poprzez utworzenie nowego obiektu.

Obliczenia w tych językach traktowane są jako wykonanie funkcji w rozumieniu matematycznym, czyli mapowania pewnego wejścia na wyjście, w związku z czym w językach tym unika się (albo nawet nie można w ogóle stosować) operacji zmieniających stan. W czystym programowaniu funkcyjnym nie występują żadne efekty uboczne obliczeń, w szczególności nie istnieje pojęcie zmiennej. Przykładowym językiem tego typu jest Haskell. Bardziej popularne są jednak języki, które pozwalają stosować efekty uboczne. Do tej rodziny zaliczają się: Erlang, Common Lisp, Scala, Scheme, Standard ML. Elementy typowe dla programowania funkcyjnego, takie jak na przykład anonimowe funkcje, można znaleźć również w niektórych językach obiektowych, takich jak Java lub C#.

3.3.2 Omówienie wybranych prac

Konwencjonalne języki programowania zgodne z paradygmatem programowania funkcyjnego stanowią prawdopodobnie najłatwiejszy cel syntezy spośród wszystkich konwencjonalnych języków. Wynika to z naturalnej zgodności tych języków z reprezentacją drzewiastą, która może zostać właściwie bez zmian wykorzystana do syntezy. W językach funkcyjnych unika się, lub wręcz nie można stosować efektów ubocznych. Przy takich założeniach funkcja zachowuje się zawsze tak samo dla tych samych wejść i cały program łatwo można przedstawić jako drzewo wywołań funkcji dla pewnych argumentów. W takim wypadku standardowe operatory przeszukiwania: krzyżowanie (zamiana poddrzew) i mutacja (wygenerowanie na nowo pewnego poddrzewa), mogą zasadniczo pozostać bez zmian. Wykorzystanie programowania genetycznego do ewolucji programów funkcyjnych

badane było w wielu pracach. Wiele z nich dotyczy jednak języków stworzonych dla celów badawczych.

Forrest Briggs i Melissa O'Neill zaproponowali wykorzystanie rachunku kombinatorów jako reprezentacji dla programów funkcyjnych [9]. Rachunek kombinatorów jest równoważny rachunkowi lambda, na którym oparte są języki funkcyjne. Opiera się on na prostych transformacjach funkcji i stałych. Działanie kombinatorów K i S przedstawione jest poniżej, przy czym α i β są dowolnymi wyrażeniami rachunku kombinatorów i mogą zawierać w sobie dalsze użycia kombinatorów K i S, funkcje lub stałe.

$$\begin{aligned} ((K\ \alpha)\ \beta) &\rightarrow \alpha \\ (((S\ \alpha)\ \beta)\ \gamma) &\rightarrow ((\alpha\ \gamma)\ (\beta\ \gamma)) \end{aligned}$$

Można wykazać, że te dwie proste reguły transformacji wystarczają, by rachunek kombinatorów był Turing-kompletny, to znaczy możliwe jest wyrażenie za jego pomocą dowolnych obliczeń, które da się zrealizować na maszynie Turinga. Jak pokazali autorzy cytowanej pracy, rachunek kombinatorów pozwala wprowadzać do programów zmienne lokalne nie wymuszając zmiany operatorów ewolucji. Takie zmienne zazwyczaj wymagają kłopotliwej, zależnej od użytej reprezentacji, obsługi, gdyż poprzez krzyżowanie można łatwo doprowadzić do sytuacji, w której powstały program jest niepoprawny.

Tina Yu i Chris Clack zaproponowali polimorficzny system typów dla GP o nazwie *PolyGP* [36], oparty na typowanym rachunku lambda. Zaproponowany przez nich system rozszerza silnie typowane GP o możliwość syntetyzowania funkcji generycznych. Tabela typów wykorzystywana w oryginalnym ujęciu STGP do sprawdzania, czy dana operacja może zostać zastosowana, została tutaj zastąpiona algorytmem unifikacji, który sprawdza, czy typy w danym elemencie mogą zostać dopasowane w taki sposób, by dany element zwracał typ zgodny z wymaganym. Podobny algorytm unifikacji został wykorzystany w autorskim systemie syntezy (rozdział 5). Ciekawą cechą *PolyGP* jest również możliwość generowania anonimowych funkcji, nazywanych w literaturze dotyczącej programowania funkcyjnego *abstrakcjami lambda* (ang. *lambda abstractions*), i wykorzystywanie ich jako terminali typu funkcyjnego.

3.4 Synteza programów obiektowych

3.4.1 Wprowadzenie

Programowanie obiektowe jest obecnie najpowszechniej stosowanym paradygmatem programowania. Wynika to prawdopodobnie z tego, że pozwala ono w wygodny i intuicyjny sposób modelować zależności między danymi. W programowaniu obiektowym podstawową jednostką programu jest *obiekt*. Każdy obiekt reprezentuje pewien zbiór powiązanych ze sobą danych i udostępnia zestaw operacji (nazywanych *metodami*), możliwych do wykonania na tych danych. Klasa jest uogólnieniem całych grup obiektów i określa ich zachowanie oraz składowe elementy stanu. Przykładowo, jako klasę można by traktować ogólne pojęcie 'Samochód', a jako obiekty konkretne samochody (instancje klasy 'Samochód'). W obiektowych językach programowania często wprowadza się relację *dziedziczenia* między klasami. W naszym przykładzie, klasa 'Samochód ciężarowy', która trzyma dodatkową

informację o przewożonych towarach, mogłaby dziedziczyć po klasie 'Samochód', dzięki czemu obiekty klasy 'Samochód ciężarowy' będą miały dostęp do wszystkich informacji i metod zdefiniowanych w klasie 'Samochód'. Mechanizm *polimorfizmu* pozwala z kolei traktować klasy dziedziczące tak samo jak ich „rodzica" (nazywanego *nadklasą*; analogicznie „dziecko" to *podklasa*). Wraz z możliwością *przeciążania* metod, czyli zdefiniowaniu na nowo danej metody nadklasy w podklasie, programista zyskuje potężne narzędzie abstrakcji, które pozwala mu traktować wszystkie samochody tak samo, a rzeczywiste zachowania będą właściwe dla typu samochodu.

3.4.2 Omówienie wybranych prac

Wykorzystanie GP do generowania programów obiektowych określane jest w literaturze jako *OOGP* (ang. *Object Oriented GP*). Zaprojektowanie hierarchii klas i zdefiniowanie ich metod wymaga bardzo dużej wiedzy eksperckiej i zrozumienia problemu. W złożonych klasach tworzonych przez programistów metody często odwołują się do innych metod z tej samej klasy, co czyni implementację bardziej modularną i łatwiejszą do modyfikacji. Jednak zapewnienie tego typu modularności w automatycznie generowanych programach jest trudne. Wszystko wskazuje na to, że automatyczna synteza przy użyciu GP programów z samodzielnie definiowanymi klasami wymagać będzie jeszcze dużo prac w tej dziedzinie [5]. Aktualnie prace nad OOGP skupione są przede wszystkim na generowaniu implementacji metod w zadanych z góry architekturach.

Pionierem zastosowania GP do generowania programów obiektowych prawdopodobnie jest Wilker Shane Bruce [10], który analizował efektywność generowania implementacji metod dla klas reprezentujących proste struktury danych, takie jak stos czy kolejka. Metody operowały na współdzielonej indeksowanej pamięci. W pracy zostało między innymi porównane równoczesne syntetyzowanie implementacji wszystkich metod z syntetyzowaniem ich indywidualnie. Ten drugi wariant, co nie jest wielki zaskoczeniem, okazał się kilka rzędów wielkości bardziej wydajny. Podobnie STGP okazało się dawać lepsze rezultaty niż brak uwzględnienia typów w ewolucji. Programy reprezentowane były jako drzewa złożone ze zbioru predefiniowanych przez autora instrukcji i nie dotyczyły żadnego istniejącego języka programowania.

Pierwszą pracą pokazującą wykorzystanie OOGP w konwencjonalnym języku programowania jest [5]. Do uzyskania informacji o dostępnych metodach i zmiennych została wykorzystana refleksja. Programy generowane były w Javie. Ciało wybranej metody reprezentowane było jako sekwencja wywołań pozostałych metod klasy na dostępnych obiektach (potencjalnie zmieniając ich stan), przy czym zwracany był tylko wynik ostatniego elementu sekwencji. Sama praca miała na celu pokazanie, że taka ewolucja jest możliwa, i omawiała wykorzystanie systemu na bardzo prostym przypadku.

Następną istotną pracą mieszczącą się w nurcie generowania implementacji metod jest [22], w której Simon Lucas demonstruje swój system zdolny do ewolucji implementacji metod w Javie, pozostawiając w zakresie przyszłych prac ewolucję interfejsów i klas. Implementacja metody reprezentowana jest w jego systemie jako sekwencja *instrukcji*. Każda instrukcja składa się z odniesienia do metody, odniesienia do obiektu (które jest ignorowane, jeżeli wybrana metoda jest statyczna) i tablicy obiektów, w której zawarte są argumenty dla metody. Użytkownik przed rozpoczęciem syntezy specyfikuje dwa globalne

zbiory obiektów. Jeden z nich zawiera obiekty, na rzecz których wywoływane mogą być metody, a drugi zawiera obiekty możliwe do użycia jako argumenty metod. Poprzez użycie refleksji, system jest w stanie dla każdego obiektu określić zbiór dopuszczalnych metod, czyli takich, dla których można dopasować argumenty z ogólnego zbioru obiektów-argumentów. W systemie do ewolucji wykorzystana została tylko mutacja, która z jednakowym prawdopodobieństwem albo dodaje (losowo wygenerowaną), albo usuwa instrukcję z sekwencji. W artykule działanie systemu zademonstrowane zostało na przykładzie ewolucji programów tworzących obrazy złożone z podstawowych figur geometrycznych. Programy te korzystają z obiektów i funkcji zdefiniowanych w już istniejących bibliotekach. Praca ta, podobnie jak poprzednio omówiona, miała na celu przede wszystkim skierowanie uwagi na możliwość wykorzystania GP do ewolucji programów w językach zorientowanych obiektowo.

Simon Lucas, we współpracy z Alexandros Agapitos, kontynuował swoje prace w zakresie OOGP i z powodzeniem wykorzystał swój system do ewolucji funkcji rekurencyjnych w Javie [7] i efektywnych (o złożoności $O(n \log n)$) rekurencyjnych algorytmów sortowania [6].

3.5 Synteza programów w języku Scala

Scala [25] jest relatywnie młodym językiem programowania kompilowanym do bajtkodu, a przez to wykonywalnym na maszynie wirtualnej Javy. Zachowana jest więc w przypadku Scali istotna zaleta kompatybilności pomiędzy różnymi systemami operacyjnymi. Sam język pozwala na programowanie zarówno w paradygmacie obiektowym jak i funkcyjnym, dając programiście do dyspozycji wiele składniowych konstrukcji niedostępnych w Javie.

Nie było dotychczas wielu prac dotyczących ewolucji programów w języku programowania Scala. W tej sekcji przedstawione zostaną dwa systemy, które potrafią generować programy w Scali. Oba wykorzystują programowanie genetyczne, jednak przetwarzają programy na zupełnie różnych poziomach – jeden z nich reprezentuje je jako AST, a drugi jako bajtkod. Zaprojektowany w ramach tej pracy system realizuje GP za pomocą jeszcze innej reprezentacji, mianowicie zbioru prekompilowanych instrukcji (funkcji).

3.5.1 Ewolucja AST – Gen-O-Fix

Gen-O-Fix [34], zaprojektowany przez J. Swana, M. Epitropakisa i J. Woodwarda, jest systemem automatycznego ulepszania programów napisanych w Scali. Zadanie ulepszania programów różni się od zadania syntezy tym, że elementem wejścia systemu jest dodatkowo wstępna wersja programu, która jest następnie optymalizowana ze względu na pewien zbiór kryteriów. Omówimy jednak ten system bardziej szczegółowo z tej racji, że dotyczy Scali.

Jak już zostało wspomniane, Gen-O-Fix wykorzystuje do działania programowanie genetyczne. Programy reprezentowane są wewnętrznie jako drzewa AST. Zbiór możliwych do wykorzystania terminali i nieterminali określany jest na podstawie początkowego (dostarczonego przez użytkownika) programu, ale może zostać wzbogacony o dowolne instrukcje na podobnej zasadzie, jak w standardowym GP definiowany jest zbiór funkcji. Nowe drzewo AST w celu oceny musi zostać, z dużym narzutem czasowym, skompilowane, czego nie da się w ogólności uniknąć w kontekście optymalizacji wymagań niefunkcjonalnych (takich jak

np. rzeczywisty czas działania). W Gen-O-Fix drzewa AST modyfikowane są przez operacje krzyżowania i mutacji działające w oparciu o reguły transformacji, które gwarantowały tworzenie poprawnych drzew. Do implementacji reguł transformacji wykorzystywane jest dopasowywanie wzorca (ang. *pattern matching*), realizowane w Scali za pomocą instrukcji `match`.

Gen-O-Fix realizuje koncepcję *wbudowanego dynamicznego ulepszania* (ang. *EDI, Embedded Dynamic Improvement*) [11]. Polega ona na ciągłym ulepszaniu w tle wskazanych przez twórcę oprogramowania fragmentów systemu informatycznego. Ulepszane mogą być takie właściwości jak na przykład zużycie energii, zajmowana pamięć czy efektywność. Poprawność zmian zapewniona jest przez zestaw testów. Zaletą tego podejścia jest automatyczne reagowanie na zmiany w środowisku, w którym wykonywany jest program (np. dostępne zasoby obliczeniowe).

3.5.2 Ewolucja bajtkodu – FINCH

FINCH [26], zaprojektowany przez M. Orlova i M. Sippera, reprezentuje programy w postaci *bajtkodu*. Każdy program w Javie lub Scali kompilowany jest do uniwersalnego ze względu na system operacyjny bajtkodu. Podczas wykonywania fragmentu programu maszyna wirtualna Javy podejmuje decyzję, czy fragment ten zostanie zinterpretowany, czy też skompilowany do kodu maszynowego danej platformy – kompilacja *JIT* (ang. *Just-In-Time compilation*). Twórcy FINCH'a zrealizowali swój system z myślą o Javie, jednak, jak sami zaznaczają w artykule, może być on równie dobrze wykorzystany do ewolucji programów w Scali. Rozróżnienie to ma sens tylko wtedy, kiedy chcemy uzyskać kod źródłowy wygenerowanego programu. W systemie ma to miejsce przez *dekompilację* bajtkodu, czyli próbę odtworzenia kodu źródłowego w Javie, który po kompilacji dałby właśnie taki bajtkod. Wykorzystanie systemu do uzyskania kodu źródłowego Scali wymagałoby więc teoretycznie wyłącznie odpowiedniego dekompilatora[1].

Jak twierdzą autorzy, ewolucja bajtkodu pozwala łatwo ewoluować niczym nieograniczone programy, czyli programy nie zawężone wyłącznie do pewnego podzbioru wszystkich możliwych programów w danym języku. Jak zauważają w artykule, kod źródłowy jest przeznaczony do czytania i przetwarzania przez ludzi, i jako taki nie stanowi dobrej reprezentacji dla ewolucji. Trzeba tu zaznaczyć, że autorzy rozpatrują wyłącznie wariant ewolucji w pełni poprawnych programów, twierdząc, że w przeciwnym wypadku populacji zabraknie dywersyfikacji, gdyż niekompilujący się kod będzie miał minimalną wartość fitnessu i będzie równocześnie bardzo prawdopodobnym rezultatem krzyżowania lub mutacji. Również gramatyka języka, zdaniem autorów, nie dostarcza istotnych informacji dotyczących semantyki i słabo się nadaje do ewolucji programów niezawężonych do podzbioru języka.

FINCH zasadniczo powstał z myślą o ulepszaniu programów, jednak w praktyce łatwo go również wykorzystać do syntezy. Użytkownik podaje systemowi na wejście początkowy program, którego kopie tworzą całą początkową populację. FINCH wykorzystuje do przeszukiwania wyłącznie krzyżowanie, które zostało zaprojektowane tak, by tworzone programy zawsze były poprawne. W programie początkowym powinny więc być zawarte wszystkie instrukcje bajtkodu, które mają brać udział w procesie poszukiwania wzorcowego

[1] Nie można jednak wykluczyć, że pewne kombinacje bajtkodu, mimo że dekompilowalne do Javy, nie będą mogły zostać zdekompilowane do Scali. Autorzy systemu nie przeprowadzili tego typu testów.

rozwiązania. Początkowy program może być właściwie dowolny (musi się jednak kompilować), co, kosztem czasu działania algorytmu, pozwala łatwo zastosować go również do zadań syntezy.

Wyniki uzyskane przez FINCH'a wydają się być bardzo dobre. Potrafił on z praktycznie 100%-ową skutecznością rozwiązywać problemy regresji symbolicznej dla wielomianów do 9-tego stopnia, sumowania liczb z tablicy i poprawy programu realizującego strategię przeszukiwania stanów prostej gry. Pełne omówienie wyników można znaleźć w [26].

Rozdział 4

Mechanizmy refleksji w języku Scala

4.1 Wprowadzenie

Refleksja (ang. *reflection*) w programowaniu jest zdolnością do przetwarzania podczas wykonywania programu informacji dotyczących jego struktury. Na przykład w językach obiektowych umożliwia ona wypisanie podczas działania programu nazw wszystkich metod danej klasy lub tworzenie obiektów klas nieznanych podczas kompilacji. Mechanizmy refleksji leżą u podstaw zaprojektowanego systemu syntezy programów opisanego w sekcji 5, tak więc konieczne jest ich wcześniejsze omówienie.

Mechanizmy refleksji można podzielić na dwie kategorie: czasu kompilacji i czasu wykonania. Pierwsze z nich wykorzystywane są podczas kompilacji programu i służą przede wszystkim do generowania kodu. Drugie mają miejsce podczas działania programu i to na ich omówieniu, z racji wykorzystania w zaprojektowanym systemie syntezy, skupimy się w dalszej części tej sekcji.

Sposoby użycia i możliwości refleksji są bardzo mocno zależne od konkretnego języka programowania, tak więc zawęzimy się w naszych rozważaniach do refleksji w Scali [2, 12]. Refleksję czasu wykonania można w Scali podzielić, w zależności od sposobu użycia, na statyczną albo dynamiczną. Do metod refleksji dynamicznej można też zaklasyfikować przetwarzanie *drzew AST*, któremu poświęcona została osobna podsekcja.

4.2 Refleksja statyczna

Refleksja statyczna polega na przetwarzaniu informacji dotyczących elementów znanych w momencie kompilacji programu. Na Listingu 4.1 pokazane jest, jak poprzez mechanizm refleksji można utworzyć obiekt klasy znanej podczas kompilacji. Zauważyć można pojawiające się w nazwach klas i zmiennych pojęcie *lustra*[1] (tłumaczenie z ang. *mirror*). Lustra w Scali to klasy, które służą do wykonywania operacji metajęzykowych na pewnych, zależnych od rodzaju lustra, elementach struktury programu (takich jak np. metody albo klasy).

W linii nr 9 na Listingu 4.1 tworzone jest lustro, które pozwala uzyskać dostęp do wszystkich klas programu załadowanych przez maszynę wirtualną Javy. Z tego lustra mo-

[1] Alternatywnym tłumaczeniem mogłoby być *odbicie*, jednak słowo „lustro" zdaniem autora pracy niesie ze sobą znaczenie urządzenia, które „wytwarza" odbicia, co lepiej pasuje do natury tego mechanizmu.

Listing 4.1: Wykorzystanie refleksji statycznej do utworzeniu obiektu klasy.

```
1  class Person(name:String, age:Int) {
2    private def getName():String = name
3    def getAge():Int = age
4    override def toString():String = name + ", age " + age
5  }
6
7  def main(args: Array[String]) {
8    // Utworzenie lustra dla wszystkich wczytanych klas.
9    val mirror:Mirror = runtimeMirror(getClass.getClassLoader)
10
11   // Utworzenie lustra dla klasy Person.
12   val classSymbol:ClassSymbol = typeOf[Person].typeSymbol.asClass
13   val classMirror:ClassMirror = mirror.reflectClass(classSymbol)
14
15   // Utworzenie lustra dla konstruktora klasy Person.
16   val constrSymbol:MethodSymbol = typeOf[Person].decl(termNames.CONSTRUCTOR).asMethod
17   val constructorMirror:MethodMirror = classMirror.reflectConstructor(constrSymbol)
18
19   // Wywołanie konstruktora i uzyskanie obiektu klasy Person.
20   val person:Any = constructorMirror.apply("Iwo", 24)
21   println("Person is: " + person)
22   /*
23   * Person is: Iwo, age 24
24   */
25 }
```

żemy uzyskać lustro dedykowane konkretnej klasie (*ClassMirror*; linia 13). Wykorzystujemy do tego celu *symbol* danej klasy, który przechowuje informacje powiązane z daną „nazwą" rozpoznawaną w języku. Każdy element rozpoznawalny w programie posiada swój własny symbol. W następnym kroku, w linii 17, tworzone jest lustro dla konkretnej metody (klasa *MethodMirror*), w tym wypadku konstruktora. Lustra metod mają specjalną właściwość, mianowicie można wykorzystać metodę *apply* w celu wykonania tej metody. Jak widać na listingu, wywołanie konstruktora z odpowiednimi argumentami zwraca nam obiekt klasy *Person* (traktowany jako *Any*).

Na Listingu 4.2 znajduje się kod wypisujący nazwy wszystkich zadeklarowanych metod, również tych prywatnych, w klasie *Person*. Kod ten wykorzystuje zmienne z poprzedniego przykładu. Posiadając symbol metody prywatnej i lustro klasy, można bardzo łatwo tę metodę wywołać, co byłoby niemożliwe bez użycia refleksji. Symbol pozwala uzyskać wszystkie istotne informacje związane z danym elementem. W szczególności symbole odpowiadające metodom pozwalają uzyskać informacje o ewentualnych parametrach typu, przyjmowanych argumentach i typie zwracanej wartości.

Mechanizm luster w refleksji opiera się na enkapsulacji, wyraźnemu oddzieleniu od pozostałych elementów języka i możliwie dokładnemu odzwierciedleniu struktury języka [8]. Przykładem implementacji refleksji nie bazującej na lustrach jest refleksja w Javie (Java Core Reflection), w której informacje o klasie można uzyskać z każdego obiektu poprzez użycie metody *getClass* – narusza to założenie o rozdzielności zwykłych elementów języka

Listing 4.2: Wykorzystanie refleksji do wypisania zadeklarowanych metod klasy.

```
val declaredInPerson = classSymbol.info.decls
declaredInPerson.foreach { x:Symbol =>
  if (x.isMethod)
    println(x)
}
/*
* constructor Person
* method getName
* method getAge
* method toString
*/
```

od operacji metajęzykowych.

4.3 Refleksja dynamiczna

Zdarzają się sytuacje, w których pojawia się potrzeba utworzenia podczas działania programu instancji klasy, która nie jest znana w momencie kompilacji. Podobna sytuacja ma na przykład miejsce w zaprojektowanym systemie syntezy, opisanym w rozdziale 5, w którym na wejście podawany jest obiekt (w rozumieniu Scali) użytkownika zawierający definicje funkcji do użycia w syntezie. Obiekt ten musi zostać skompilowany i przetworzony w celu wyodrębnienia tych funkcji. Nie możemy w takiej sytuacji zastosować refleksji statycznej, gdyż w momencie kompilacji stworzony przez użytkownika obiekt zawierający funkcje nie jest jeszcze znany. Do tego rodzaju zastosowań wykorzystuje się *refleksję dynamiczną*.

Refleksja dynamiczna omówiona zostanie na przykładzie tworzenia instancji klasy, której kod jest podawany na wejście przez użytkownika. Przede wszystkim kod ten musi zostać skompilowany. Kompilacja nie wchodzi bezpośrednio w skład mechanizmów refleksji, więc jej pis zostanie tutaj pominięty. Opis procesu kompilacji w systemie Gcore znajduje się w sekcji 6.3.

Załóżymy, że klasa została już, podczas działania programu, skompilowana i posiadamy *classloader* (klasa odpowiedzialna w Javie i Scali za wczytywanie klas potrzebnych podczas działania aplikacji), mający dostęp do bajtkodu skompilowanej klasy. Listing 4.3 zawiera kod źródłowy funkcji, która może dynamicznie utworzyć instancję tej klasy. Różnica w stosunku do refleksji statycznej polega w zasadzie wyłącznie na sposobie uzyskania lustra klasy. W tym wypadku musimy wykorzystać jej nazwę, zamiast wprost odwołać się do jej typu. Nazwę z kodu źródłowego można wyodrębnić przez wykorzystanie parsowania drzewa AST. Warto odnotować, że instancja dynamicznie wczytanej klasy będzie traktowana w aplikacji jako *Any*. By móc wykonać metodę na rzecz tej instancji, konieczne będzie utworzenie i zachowanie dla niej lustra (odbywa się to tak samo, jak w przypadku refleksji statycznej).

Listing 4.3: Kod źródłowy funkcji, która dynamicznie tworzy instancję teoretycznie dowolnej niezagnieżdżonej klasy, nawet nieznanej podczas kompilacji. Na wejście podawane są: classloader potrafiący wczytać bajtkod klasy, nazwa klasy oraz argumenty potrzebne do konstruktora.

```
1  def createInstanceOfClass(classLoader:ScalaClassLoader,
2                            className:String,
3                            args:List[Any]):Any = {
4    // Tworzenie głównego lustra na podstawie classloader'a.
5    val mirror = scala.reflect.runtime.universe.runtimeMirror(classLoader)
6
7    // Tworzenie lustra dla klasy.
8    val classSymbol = mirror.staticClass(className)
9    val classMirror = mirror.reflectClass(classSymbol)
10
11   // Utworzenie lustra dla konstruktora klasy UserClass.
12   val constrSymbol:MethodSymbol = classSymbol.info.decl(termNames.CONSTRUCTOR).asMethod
13   val constructorMirror:MethodMirror = classMirror.reflectConstructor(constrSymbol)
14
15   // Utworzenie obiektu klasy.
16   constructorMirror(args:_*)
17 }
```

4.4 Drzewa AST

Drzewo AST (ang. *Abstract Syntax Tree*) wyrażenia zapisanego w pewnym języku programowania jest wysokopoziomowym opisem w postaci drzewa językowej struktury tego wyrażenia. Mówiąc inaczej, drzewo AST abstrahuje od szczegółów składniowych języka i reprezentuje program na poziomie elementów wynikających z gramatyki języka. Na Listingu 4.4 przedstawione jest drzewo AST prostego wyrażenia w Scali. Jak można zauważyć, węzłami drzewa są instancje różnych klas reprezentujących elementy języka. Przykładowo *Block* reprezentuje blok kodu, a *Apply* wywołanie funkcji. Na poziomie AST nie występują jeszcze konkretne obiekty – odpowiadają im póki co same nazwy, na listingu są to np. *"x"* i *"$plus"*. Zamiana nazw na odpowiadające im obiekty i sprawdzenie zgodności typów odbywa się dopiero na etapie kompilacji.

Drzewa AST pozwalają w wygodny sposób uzyskiwać informacje dotyczące kodu źródłowego, takie jak np. nazwę zdefiniowanego obiektu. W Scali wykorzystać można w tym celu *traverser*, który przechodzi przez wszystkie węzły drzewa, i, przy wykorzystaniu mechanizmu dopasowywania (instrukcja *match*), może wykonywać operacje zależne od rozpoznawanych typów węzłów. Manualne parsowanie kodu źródłowego wymaga uwzględnienia wszystkich możliwych konstrukcji języka, co może łatwo prowadzić do błędów. W bibliotece refleksji Scali znajdują się również funkcje, które pozwalają zamienić drzewo AST z powrotem na kod źródłowy, co pozwala stosować ten mechanizm również do modyfikacji kodu źródłowego.

Drzewo AST może zostać również skompilowane i wykonane. Na Listingu 4.5 przedstawiony jest proces uzyskania drzewa AST z kodu źródłowego reprezentowanego jako tekst, a następnie skompilowania i wykonania tego drzewa. Do operacji na drzewach AST

wykorzystywana jest instancja klasy *ToolBox*. W linii 12 tworzona jest ona z lustra dla classloader'a aplikacji. Metoda *parse* (linia 15) dokonuje zamiany kodu źródłowego na postać drzewa AST (klasa *Tree*). Drzewo jest kompilowane i wykonywane przy użyciu metody *eval* (linia 18). *Eval* zwraca wartość wynikową kompilowanego programu, tak więc w celu uzyskania rezultatu działania funkcji *fun* musieliśmy w linii 8 dodać po średniku jej wywołanie dla argumentów x i y.

Listing 4.4: Drzewo AST dla wyrażenia {x+y; z=5} w Scali.

```
1  Block(
2      List(
3          Apply(
4              Select(
5                  Ident(TermName("x")),
6                  TermName("$plus")
7              ),
8              List(Ident(TermName("y")))
9          )
10     ),
11     Assign(
12         Ident(TermName("z")),
13         Literal(Constant(5))
14     )
15 )
```

Listing 4.5: Kod źródłowy realizujący zamianę kodu źródłowego Scali na drzewo AST, a następnie jego kompilację i wykonanie.

```
1  import scala.reflect.runtime._
2  import scala.reflect.runtime.universe._
3  import scala.tools.reflect.ToolBox
4
5  def main(args: Array[String]) {
6    def invokeFunction(x:Int, y:Int):Int = {
7      // Kod do wykonania.
8      val code:String = s"def fun(x:Int, y:Int):Int = x+y; fun($x, $y)"
9
10     // Tworzenie toolBoxa, służącego do wykonywania podstawowych operacji na AST.
11     val mirror = universe.runtimeMirror(getClass.getClassLoader)
12     val toolBox = mirror.mkToolBox()
13
14     // Utworzenie drzewa AST.
15     val tree:Tree = toolBox.parse(code)
16
17     // Skompilowanie i wykonanie drzewa AST.
18     val result:Any = toolBox.eval(tree)
19     result.asInstanceOf[Int]
20   }
21
22   println("Result: " + invokeFunction(9, 3))
```

```
23   /*
24   * Result: 12
25   */
26 }
```

Rozdział 5

System syntezy

5.1 Wprowadzenie

W ramach pracy zaprojektowany i zaimplementowany został system syntezy o nazwie GCORE, umożliwiający generowanie programów w języku Scala [25]. Napisany został on również w języku Scala.

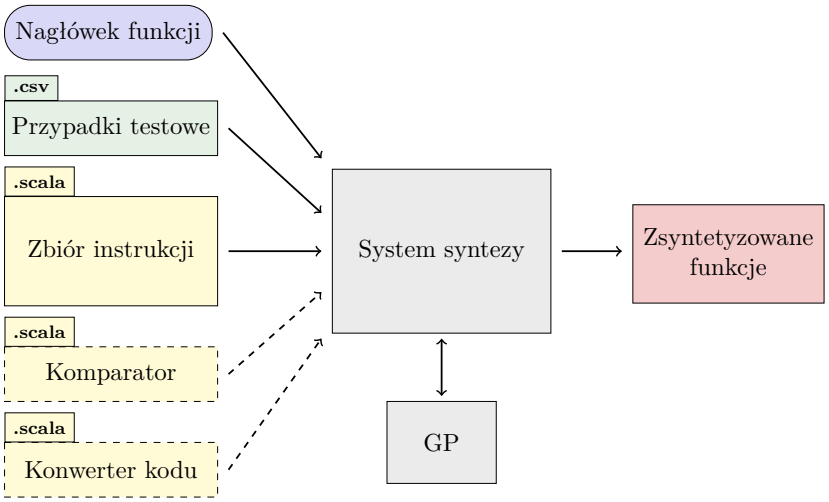
5.2 Ogólne założenia systemu

Podczas projektowaniu systemu przyjęto następujące ogólne założenia:

- Algorytmem syntezy będzie programowanie genetyczne.
- Cel zadania syntezy określony będzie poprzez zbiór przypadków testowych $T = \{(x, y)\}$, gdzie x jest wejściem dla programu a y oczekiwaną wartością, którą powinien zwrócić ten program dla tego wejścia.
- Użytkownik podaje na wejście systemu dwa zbiory funkcji (instrukcji): przyjmujących argumenty F i bezargumentowych (generatory stałych) G.
- Funkcje z obu zbiorów podawane są na wejście systemu w formie kodu źródłowego.
- Syntetyzowane programy będą się składać z funkcji ze zbioru F i wartości zwracanych przez funkcje ze zbioru G.
- Funkcje ze zbiorów F i G będą traktowane przez system jako czarne skrzynki.
- Zadaniem systemu jest zsyntetyzowanie takiej funkcji f będącej złożeniem funkcji z F i wartości uzyskanych z G, że $\forall_{(x,y)\in T}\ f(x) = y$.
- System syntezy ma być uniwersalny, to znaczy zmiana zadania nie powinna pociągać za sobą konieczności rekompilacji całego systemu.

5.3 Schemat działania systemu

Na Rys. 5.1 przedstawiony jest ogólny schemat działania systemu. Po lewej stronie rysunku znajdują się, specyficzne dla danego problemu, informacje podawane przez użytkownika na wejście systemu. Element po prawej stronie odpowiada wynikowi zwracanemu



Rysunek 5.1: Ilustracja przedstawia wysokopoziomowy schemat działania zaprojektowanego systemu syntezy. Elementy po lewej stronie rysunku są wejściem dla systemu syntezy i tworzone są przez użytkownika w zależności od problemu, który ma zostać rozwiązany. Element po prawej stronie reprezentuje ciało funkcji zwracane przez system. Na rysunku nie zostały uwzględnione parametry linii poleceń.

przez system. Jak można zauważyć, użytkownik musi podać na wejście następujące elementy:

- nagłówek funkcji,
- przypadki testowe,
- zbiór instrukcji,
- komparator (opcjonalnie),
- konwerter kodu (opcjonalnie).

W sekcji 5.5 zostaną omówione elementy podawane na wejście systemu, w sekcji 5.6 proces syntezy, a w sekcji 5.7 zwracane przez system wyniki.

5.4 Reprezentacja programów

5.4.1 Wprowadzenie

W zaprojektowanym systemie do syntetyzowania programów wykorzystywane jest programowanie genetyczne. Z wyborem tym związana jest kwestia ustalenia sposobu reprezentacji rozwiązań, w tym wypadku programów w Scali.

Scala jest językiem kompilowanym, tak więc jeżeli syntetyzowane programy byłyby przetwarzane w formie kodu źródłowego lub *drzewa AST*, to istniałaby konieczność ich kompilacji w celu oceny ich zachowania. Przeprowadzone eksperymenty pokazały, że czas kompilacji nawet krótkiego programu jest stosunkowo długi. Przykładowo, średnie czasy wywołania funkcji dodającej dwie liczby całkowite są następujące: 33.59 ms przy kompilowaniu kodu źródłowego w postaci tekstu i 63.7 ms przy kompilowaniu drzewa AST. Widać wyraźnie, co zresztą nie jest zaskakujące, że kompilacja generuje duży narzut cza-

sowy. Zastanawiający jest fakt, że kompilowanie drzewa AST trwa prawie dwukrotnie dłużej. Autorowi nie udało się jednak znaleźć informacji, z czego to wynika.

Algorytmy ewolucyjne opierają się na przeszukiwaniu, potencjalnie bardzo dużej, przestrzeni rozwiązań i efektywność oceny osobników jest, przy założeniu pewnego limitu czasowego, istotnym czynnikiem wpływającym na jakość ostatecznego rozwiązania. Z drugiej strony można zadać sobie pytanie, czy w ogóle jest możliwe uniknięcie kompilacji w tym kontekście – istota stosowania operatorów przeszukiwania (m.in. krzyżowania i mutacji) polega właśnie na eksploracji przestrzeni możliwych rozwiązań, co w tym wypadku oznacza generowanie dużej liczby nowych programów.

5.4.2 Opis koncepcji

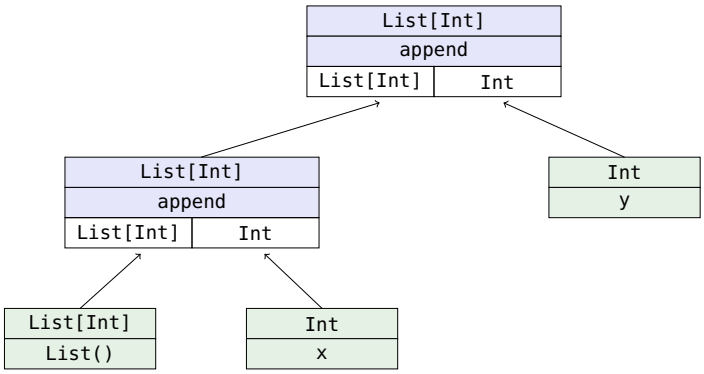
Okazuje się, że przy pewnych dodatkowych założeniach i wykorzystaniu odpowiednich mechanizmów refleksji, można uniknąć czasochłonnej kompilacji. Podstawy refleksji w Scali zostały wprowadzone w rozdziale 4. Jak mogliśmy zauważyć, możliwe jest dynamiczne utworzenie obiektu danej klasy i wykonywanie jego metod poprzez utworzenie dedykowanego lustra dla każdej z nich. Jeżeli założymy, że osobniki w populacji rozwiązań będą składane wyłącznie z takich wcześniej skompilowanych funkcji, to znika potrzeba kompilacji osobników po każdej modyfikacji.

Wykorzystanie luster przy tym samym zadaniu, które wykonywały wspomniane wcześniej metody kompilacji, pozwoliło osiągnąć średni czas wykonania funkcji wymienionej w podsekcji 5.4.1 wynoszący 0.03 ms, czyli ponad 1000-krotnie krótszy niż przy kompilacji kodu źródłowego. W metodzie wykorzystującej lustra proces kompilacji ma miejsce tylko raz podczas inicjalizacji systemu, a nie przy każdej ewaluacji programu. Z tych też względów zdecydowano się ewoluować w systemie jedynie programy składane z uprzednio zdefiniowanych funkcji.

W zaprojektowanym systemie funkcje definiowane są przez użytkownika, tworząc formalizm, który można by nazwać językiem dziedzinowym (ang. *DSL, Domain Specific Language*), to znaczy językiem stworzonym z myślą o rozwiązywaniu problemów w pewnym specyficznym zastosowaniu. Z perspektywy systemu syntezy operacje wykonywane wewnątrz funkcji nie mają znaczenia,

Takie podejście ma i zalety i wady. Wykonywanie programów przebiega, dzięki wykorzystaniu luster, znacznie szybciej i operujemy na wysokopoziomowych konstrukcjach językowych, co może przyspieszyć syntezę. Odbywa się to jednak kosztem ogólności generowanych programów, które będą składane wyłącznie z tych zdefiniowanych przez użytkownika instrukcji. Co więcej, ostateczne rozwiązanie będzie korzystało ze zdefiniowanych funkcji, tak więc nie będzie w pełni autonomicznym programem Scali.

Z racji przyjętej koncepcji wykorzystania zbioru wcześniej skompilowanych funkcji, najbardziej naturalną reprezentacją programów jest reprezentacja drzewiasta. W podsekcji 5.4.3 zostanie ona przedstawiona na przykładzie. Od strony technicznej reprezentacja omawiana jest w sekcji 6.5.



Rysunek 5.2: Przykładowe drzewo programu zbudowane przy wykorzystaniu nieterminali z Tabeli 5.1 oraz terminali: zmiennych x i y, pustej listy, stałych liczb całkowitych.

5.4.3 Przykład drzewa programu

Załóżmy, że użytkownik dostarczył systemowi instrukcje (funkcje) podane w Tabeli 5.1. Sposób, w jaki użytkownik może definiować instrukcje, zostanie omówiony w następnej sekcji – póki co założymy, że zostały one już zarejestrowane w systemie, podobnie jak odpowiednie generatory stałych. Niech celem użytkownika będzie wygenerowania ciała funkcji, która przyjmując dwie liczby całkowite (w Scali takim liczbom odpowiada typ Int) jako argumenty, oznaczone odpowiednio jako x i y, utworzy listę je zawierającą (typ List[Int]).

Rysunek 5.2 przedstawia przykładowe drzewo programu rozwiązującego powyżej przedstawiony problem korzystając z funkcji wymienionych w tabeli. Każdej instrukcji odpowiada osobny blok. Pierwszy wiersz każdego bloku zawsze zawiera typ wartości zwracanej przez ten blok. Drugi wiersz zawiera w przypadku nieterminali nazwę funkcji (w tym wypadku *append*), a w przypadku terminali albo nazwę zmiennej (x, y) albo konkretną wartość (na rysunku przykładem takiej wartości jest pusta lista, oznaczona jako *List()*). Blok reprezentujący nieterminale posiada też trzeci rząd, którego kolumny mówią o liczbie i typach przyjmowanych argumentów. W poprawnie zbudowanym drzewie programu do każdej kolumny w trzecim wierszu bloku reprezentującego nieterminal przypisany jest inny blok, przy czym typ zwracany przez ten blok musi być zgodny z typem argumentu. W systemie uwzględniony został polimorfizm ze względu na typ, tak więc do argumentu o typie listy mógłby zostać przypisany dowolny blok zwracający typ dziedziczący po ty-

Tabela 5.1: Przykładowy zestaw instrukcji podanych przez użytkownika w celu ewolucji programów operujących na listach (typ List) zawierających liczby całkowite (typ Int).

Nazwa	Typy argumentów	Typ wyniku	Opis wyniku
`size`	List[Int]	Int	Liczba elementów na liście.
`head`	List[Int]	Int	Pierwszy element listy.
`tail`	List[Int]	List[Int]	Lista zawierająca wszystkie elementy poza pierwszym.
`append`	List[Int], Int	List[Int]	Lista powstała przez dodanie podanej jako argument liczby na koniec podanej jako argument listy.

pie List. Użytkownik, o czym będzie wspomniane dalej, nie jest ograniczony wyłącznie do standardowych typów Scali i może definiować własne. System typów zrealizowany w GCORE jest oparty na silnie typowanym GP, opisanym w sekcji 2.3.4.

Żeby skompilowana funkcja powiązana z instrukcją *append* mogła zostać wykonana i zwrócić wynik, instrukcja musi dostać na wejście wszystkie wymagane argumenty. Oznacza to, że instrukcje będące *dziećmi* danej instrukcji X, przez co mamy na myśli to, że stanowią argumenty dla X, muszą zostać wykonane najpierw, co pociąga za sobą z kolei konieczność ewaluacji ich argumentów. Proces ten można porównać do rekurencji, która schodzi coraz niżej w drzewie programu aż nie napotka instrukcji reprezentującej terminal, który może zwrócić wartość bez żadnych argumentów.

5.5 Dane wejściowe systemu

5.5.1 Nagłówek funkcji

Nagłówek funkcji dostarcza informację o tym, jakie argumenty ma przyjmować i jaki typ ma zwracać funkcja, którą użytkownik chce zsyntetyzować. Musi być on podany na wejście jako poprawne wyrażenie Scali. Nagłówek może być podany zarówno jako parametr z linii poleceń, jak i zawarty w pliku z instrukcjami. W tym drugim przypadku, w celu odróżnienia go od zwykłych instrukcji, ciałem syntetyzowanej funkcji musi być wyrażenie Scali *???*, które odpowiada rzuceniu wyjątku o braku implementacji.

Przykładowy nagłówek, podawany jako parametr, może wyglądać tak jak na Listingu 5.1. Przykład nagłówka zawartego w pliku źródłowym z instrukcjami można znaleźć na Listingu 5.2 w linii nr 3.

Listing 5.1: Przykładowa postać nagłówka funkcji do syntezy.

```
1 def max(list:List[Int]):Int = ???
```

Wszystkie argumenty syntetyzowanej funkcji dodawane są do zbioru terminali jako zmienne o takim typie, w jakim zostały zdefiniowane. Ich wartość podczas danego uruchomienia programu jest określana na podstawie wartości odczytanych z aktualnie rozpatrywanego przypadku testowego.

5.5.2 Zbiór instrukcji

Użytkownik definiuje funkcje do wykorzystania w syntetyzowanym programie tworząc plik źródłowy Scali, zawierający pojedynczy obiekt w rozumieniu Scali (słowo kluczowe *object*), który jest odpowiednikiem instancji klasy realizującej wzorzec projektowy singleton w Javie. W obiekcie tym użytkownik implementuje wszystkie funkcje, które będą użyte do tworzenia programów, korzystając przy tym z wbudowanych typów Scali bądź typów, które sam w tym obiekcie zdefiniuje. Przykładowy kod źródłowy przedstawiony jest na Listingu 5.2.

Poprzez wykorzystanie refleksji odczytane i zarejestrowane jako nieterminale zostaną wszystkie publiczne metody przyjmujące co najmniej jeden argument. Dla przedstawio-

Listing 5.2: Przykładowy kod źródłowy będący specyfikacją zbioru instrukcji.

```
import scala.util.Random
object Obj {
  def toBeSynthesized(list:List[Int]):List[Int] = ???

  class UserClass(val x:Int, val y:Int) {
    def sumModulo(mod:Int):Int = (x+y) % mod
  }
  private var intsCreated = 0;

  // Definicje nieterminali
  def size[T](list:List[T]):Int = list.size
  def head[T](list:List[T]):T = list.head
  def tail[T](list:List[T]):List[T] = list.tail
  def append[T](list:List[T], x:T):List[T] = list :+ x

  // Definicje generatorów stałych
  private val rng = new Random()
  def randomBoolean(): Boolean = rng.nextBoolean()
  def randomInt(): Int = { intsCreated+=1; rng.nextInt(8) }
  def emptyList[T]():List[T] = List[T]()
}
```

nego listingu zarejestrowane zostaną: *size*, *head*, *tail*, *append*. Brane są pod uwagę tylko funkcje bezpośrednio zawarte w obiekcie, dlatego też funkcja *sumModulo* w wewnętrznej klasie *UserClass* nie została uwzględniona. Po utworzeniu dla każdej z tych funkcji lustra i opakowaniu go w specjalnej klasie, funkcje te dodawane są do zbioru nieterminali. Jak można zauważyć na listingu, funkcje mogą być generyczne. Sposób obsługi funkcji generycznych opisany jest w sekcji dotyczącej implementacji.

Funkcje nie przyjmujące żadnych argumentów, czyli o *arności* 0, traktowane są jako *generatory stałych* o takim typie, jaki zwraca dana funkcja. Jeżeli w systemie zaistnieje potrzeba wygenerowania terminala określonego typu, co ma miejsce podczas tworzenia populacji początkowej i stosowania operatorów przeszukiwania, to wybierany jest losowy generator spośród tych, które zwracają ten typ, i wartość zwrócona przez ten generator dołączana jest do drzewa jako terminal. Dla przedstawionego listingu zarejestrowane zostaną następujące generatory stałych: *randomBoolean*, *randomInt*, *emptyList*. Jak widać na przykładzie *emptyList*, generator może być funkcją generyczną. Zostanie on jednak wzięty pod uwagę tylko wtedy, kiedy będzie możliwe podstawienie pod jego *argumenty typu* (oznaczane zwyczajowo dużymi literami; na listingu przez T) takich konkretnych typów, że powstanie ostatecznie typ zgodny z potrzebnym typem. Jako że funkcja będąca generatorem stałych jest wykonywana za każdym razem, kiedy potrzebny jest nowa stała, można w ten sposób zaimplementować własny generator liczb pseudolosowych o dowolnym rozkładzie.

Wszystkie metody w zbiorze instrukcji operują w kontekście obiektu, w którym są zdefiniowane. Oznacza to, że obiekt może posiadać stan i być modyfikowany w trakcie działania przez wykonywane funkcje. Na listingu w zmiennej *intsCreated* przechowywana

jest liczba wyprodukowanych stałych typu *Int*.

W pliku z instrukcjami można również definiować nowe klasy i importować pakiety. Jak zostało pokazane w podsekcji dotyczącej przypadków testowych, klasy te mogą być wykorzystywane również w pozostałych elementach, które są wczytywane przez system.

5.5.3 Przypadki testowe

Określenie oczekiwanego zachowania syntetyzowanej funkcji (czyli jej *semantyki*) odbywa się poprzez podanie listy par wejście-wyjście, nazywanych *przypadkami testowymi* (lub *testami*). Każdy przypadek testowy jest przyporządkowaniem dla pewnych argumentów syntetyzowanej funkcji wartości, którą powinna dla nich zwrócić ta funkcja. Każda wartość jest wyrażeniem Scali i musi móc zostać skompilowana. GCORE wczytuje przypadki testowe z pliku w formacie CSV, w którym każda linia odpowiada jednemu przypadkowi, a kolejne wartości opisujące przypadek oddzielone są średnikiem. Przykładowe pliki CSV z testami przedstawione są na Listingach 5.3 i 5.4. Ostatnią wartością w wierszu zawsze jest oczekiwany wynik przypadku testowego. Pozostałe wartości to argumenty funkcji podawane w takiej samej kolejności, w jakiej zostały zdefiniowane w nagłówku.

Kompilacja przypadków testowych, po ich transformacji wewnątrz systemu do listy, wykonywana jest po skompilowaniu obiektu z instrukcjami. W obiekcie tym użytkownik może zdefiniować własne klasy, a następnie wykorzystać je przy specyfikacji wykonywanego zadania. Listing 5.4 pokazuje wykorzystanie obiektów własnych klas przy definiowaniu przypadków testowych. W analogiczny sposób można z nich korzystać w nagłówku funkcji. Dzięki umożliwieniu wykorzystania własnych klas do ewolucji programów, użytkownik ma pełną kontrolę nad tym, pod jaką postacią dane będą przetwarzane przez syntetyzowaną funkcję.

Listing 5.3: Przykładowa zawartość pliku CSV z przypadkami testowymi dla problemu znalezienia na liście liczb parzystych.

```
1 List(1,1+1,3,4); List(2,4)
2 List(1,3,5);  List()
3 List(12); List(12)
```

Listing 5.4: Przykładowa zawartość pliku CSV z przypadkami testowymi, pokazująca sposób wykorzystania klasy zdefiniowanej przez użytkownika. Aby ewolucja mogła się powieść, syntetyzowana funkcja musi przyjmować obiekty tej klasy jako argumenty.

```
1 new Obj.UserClass(2,4); new Obj.UserClass(4,3); 13
2 new Obj.UserClass(0,1); new Obj.UserClass(1,0); 2
3 new Obj.UserClass(1,2); new Obj.UserClass(3,4); 10
```

5.5.4 Komparator

Komparator jest opcjonalnym elementem, który użytkownik może dostarczyć na wejście jako plik źródłowy Scali zawierający obiekt o nazwie *Comparator* posiadający metodę

Listing 5.5: Kod źródłowy domyślnego komparatora.

```
object Comparator {
  def compare(output: Any, expected: Any): Double = {
    if (output == expected) 0
    else 1
  }
}
```

compare. Celem użycia komparatora jest, dla każdego przypadku testowego, ilościowe określenie na ile wynik zwrócony przez dany program z populacji odpowiada wynikowi oczekiwanemu. Liczby zwracane przez komparator wykorzystywane są do obliczania fitnessu ewaluowanych programów.

Kod domyślnego komparatora pokazany jest na Listingu 5.5 i zwraca: 1, jeżeli wynik zwrócony przez program jest równy wartości oczekiwanej testu, i 0 w przeciwnym wypadku. Jak można zauważyć, domyślny komparator oparty jest na wykorzystaniu metody == klasy *Any*, z której w Scali dziedziczą wszystkie inne klasy. W ogólności argumenty komparatora mogą być dowolnego typu, jednak jeżeli nie będzie zgodności pomiędzy ich typami, a typem zwracanym przez syntetyzowaną funkcję, to wystąpi błąd.

Możliwość zdefiniowania własnego komparatora może się okazać szczególnie przydatna w przypadku takich problemów, dla których, przy danym wejściu, program może zwrócić wiele wartości traktowanych jako poprawne. Innymi słowy, pożądane wyjście jest wówczas określone bardziej przez *ograniczenie* niż poprzez podanie jednej poprawnej wartości. Przykładem takiego problemu może być na przykład zadanie wygenerowania programu, który zamienia miejscami dwa dowolne elementy listy.

Komparator może też zostać wykorzystany jako *wyrocznia*, która realizuje proces obliczeniowy skutkujący uzyskaniem oceny poprawności wyniku. W przypadku wykorzystania komparatora jako wyroczni należy wprowadzić „sztuczne” wartości (np. *null*) w ostatniej kolumnie pliku z przypadkami testowymi (oczekiwany wynik) i podać ścieżkę do stworzonego komparatora jako parametr systemu. W kodzie komparatora realizującego wyrocznię zmienna *expected* może być ignorowana.

5.6 Proces syntezy

Proces syntezy rozpoczynany jest od inicjalizacji populacji początkowej, w trakcie której generowane są drzewa programów zgodne z reprezentacją przedstawioną w sekcji 5.4. Po inicjalizacji populacji przeprowadzana jest jej ewolucja. Szczegóły stosowanych w Gcore operatorów mutacji i krzyżowania opisane są w sekcji 6.6. System syntezy jest tak skonstruowany, że łatwo byłoby zastąpić programowanie genetyczne dowolną inną metaheurystyką.

W systemie syntezy ocena osobnika (fitness) jest równa sumie liczby testów, których ten osobnik nie spełnia. Miara ta jest więc minimalizowana, czyli rozwiązania optymalne mają fitness równy 0. Proces przeszukiwania trwa tak długo, aż nie zostanie przekroczona maksymalna liczba pokoleń lub nie zostanie znalezione rozwiązanie optymalne.

Listing 5.6: Kod źródłowy domyślnego konwertera.

```
object CodePrinter {
  def print(obj: Any):String = {
    obj match {
      case a : String => '"' + a + '"'
      case _ => obj.toString()
    }
  }
}
```

5.7 Wynik syntezy

Jak zostało wspomniane wcześniej, wynikiem syntezy jest ciało funkcji, które odpowiada wygenerowanemu programowi radzącemu sobie najlepiej na przypadkach testowych. Jednak programy wewnątrz systemu reprezentowane są nie jako kod źródłowy, a jako drzewa (zob. Rys. 5.2). Oznacza to, że program uznany przez system za najlepszy musi zostać przetransformowany do postaci kodu źródłowego zanim zostanie zaprezentowany użytkownikowi. Niektóre elementy składowe programów stosunkowo łatwo podlegają takiej transformacji, inne z kolei wymagają dodatkowej informacji ze strony użytkownika.

Transformacja nieterminali jest prosta. Każdy z nich reprezentuje pewną funkcję zarejestrowaną w systemie syntezy i zgodnie z założeniami funkcje te uznajemy za część ostatecznego rozwiązania. Ma to duży sens, ponieważ zazwyczaj programy tworzone przez programistów wykorzystują funkcje zdefiniowane w projekcie lub w różnych bibliotekach dostępnych dla danego języka. Nazwy zarejestrowanych funkcji są znane, tak więc każdy węzeł nieterminalny można zastąpić odpowiednią nazwą funkcji i w nawiasach wypisać wartości jej argumentów, tworząc tym samym poprawne wywołanie funkcji w Scali.

Transformacja zmiennych przebiega podobnie jak nieterminali, gdyż posiadają swoją nazwę, która wystarcza do ich poprawnego użycia w wynikowym programie.

Z transformacją stałych związane są jednak pewne problemy, wynikające z ich reprezentacji wewnątrz systemu w postaci wartości (konkretnego obiektu). Nie posiadamy informacji o kodzie źródłowym, który mógłby doprowadzić do powstania tego obiektu. Problem ten został rozwiązany w sposób wymagający od użytkownika podania dodatkowych informacji w postaci *konwertera kodu*.

Zadaniem konwertera jest rozpoznanie typu obiektu i zwrócenie fragmentu kodu źródłowego, który po wykonaniu pozwoliłby uzyskać dokładnie taki obiekt. Domyślny konwerter zawiera prostą regułę transformacji dla stałych klasy *String* i wywołanie metody *toString* dla stałych innych typów. Przedstawiony jest on na Listingu 5.6. Użytkownik może podać ścieżkę do własnego konwertera jako parametr systemu. Obiekt konwertera musi się nazywać *CodePrinter* i zawierać metodę *print* o sygnaturze takiej jak w przykładzie. Obiekt ten zostanie skompilowany w tym samym kontekście co wcześniej wspomniane obiekty, dzięki czemu możliwe są odwołania do klas zdefiniowanych w tamtych obiektach.

Dla wielu podstawowych typów Scali, między innymi typów liczbowych, wynik operacji *toString* jest poprawnym zapisem „tworzenia" tych obiektów. Pozwala to stosować

obj.toString() w przypadku niedopasowania *obj* do żadnej klasy.

Rozdział 6

Implementacja systemu syntezy

6.1 Wprowadzanie

W tym rozdziale omówione zostaną najważniejsze kwestie związane z implementacją systemu GCORE. Kod źródłowy projektu znajduje się na dołączonej do niniejszej pracy płycie DVD.

6.2 Narzędzia

GCORE zaimplementowany został w Scali 2.11.6 w środowisku Eclipse [1]. Wykorzystane zostały wstępne wersje bibliotek do obliczeń ewolucyjnych opracowane przez promotora tej pracy – SCEVO [3] i SCAPS [4]. Dalsza część sekcji zawiera krótki opis tych bibliotek.

6.2.1 ScEvo

SCEVO (*Scala framework for Evolutionary computation*) jest biblioteką Scali rozwijaną przez Krzysztofa Krawca i służącą do przeprowadzania obliczeń ewolucyjnych. Jej najważniejsze cechy, podane za stroną internetową, to:

- małe zużycie zasobów,
- brak zależności w stosunku do zewnętrznych bibliotek,
- udostępnienie dwóch interfejsów: mixinowego i funkcyjnego.

Biblioteka jest jeszcze wciąż rozwijana i na stan obecny nie jest dostępna publicznie. Poniżej przedstawiony jest opis ogólnej filozofii biblioteki wraz z podaniem klas wykorzystywanych w interfejsie funkcyjnym.

W SCEVO główną klasą służącą do przeprowadzania obliczeń jest *Algorithm*. Najbardziej interesującą z punktu widzenia tej pracy jest jej podklasa *IterativeAlgorithm*, która reprezentuje algorytm zmieniający stan populacji (*PopState*) w dyskretnych krokach. Populacja może zawierać albo same rozwiązania (*Solution*), albo krotki zawierające rozwiązanie i jego ewaluację (*Evaluation*). Jedynym warunkiem, który musi spełniać ewaluacja, jest tworzenie częściowego porządku na zbiorze rozwiązań.

IterativeAlgorithm potrzebuje do działania opisu pojedynczej iteracji w postaci funkcji, która zmienia jeden stan populacji z ocenionymi rozwiązaniami w drugi, oraz warunku zatrzymania obliczeń (np. wykonanie pewnej liczby iteracji lub znalezienie rozwiązania optymalnego). Kolejne kroki funkcji iteracji można składać za pomocą wbudowanej funkcji Scali *andThen*, która wyjście jednej funkcji przekazuje na wejście drugiej. Dzięki takiemu rozwiązaniu można modułowo konstruować ciąg operacji składających się na iterację algorytmu.

W ScEvo wbudowanych jest wiele klas, które realizują podstawowe operacje dla algorytmów ewolucyjnych, takie jak np. selekcję osobników do reprodukcji. Przykład użycia ScEvo do rozwiązania prostego problemu optymalizacji można znaleźć na stronie internetowej projektu [3].

6.2.2 ScaPS

ScaPS (*Scala framework for Program Synthesis*) jest biblioteką Scali rozwijaną przez Krzysztofa Krawca i zawierającą zbiór narzędzi do automatycznej syntezy programów przy wykorzystaniu programowania genetycznego. ScaPS oferuje [4]:

- gotową implementację GP dla reprezentacji drzewiastej,
- zestaw ponad dwudziestu benchmarków z dziedzin logiki, algebry i regresji symbolicznej,
- możliwość definiowania własnych zestawów instrukcji,
- różne tryby oceny osobników, w tym oceny wielokryterialne.

ScaPS, podobnie jak ScEvo, nie został w pełni ukończony i również nie jest jeszcze dostępny publicznie.

ScaPS realizuje ewolucję wykorzystując do tego bibliotekę ScEvo. Ewolucji podlegają w nim programy (klasa *Program*) złożone z instrukcji (*Instruction*). Programy są tutaj odpowiednikiem rozwiązania ze ScEvo. Instrukcje mają znaczenie w kontekście domeny (*ComputingDomain*), która zawiera metodę *semantics*. Metoda ta dla danego wejścia określonego przez aktualnie rozważany przypadek testowy zwraca funkcję, która pozwala wykonywać dowolne programy dopuszczalne w danej domenie.

Zadaniem klas dziedziczących po *SearchDriver* jest ocena programu, która zazwyczaj ma miejsce na podstawie zbioru testów. Oceny programów pozwalają określić kierunek przeszukiwania i niekoniecznie muszą dotyczyć wyłącznie poprawności całego programu.

W momencie rozpoczynania implementacji Gcore w ScaPS'ie nie było zaimplementowanej obsługi STGP. Dodanie takiej obsługi stało się więc jednym z zadań do wykonania w ramach pracy. Wymagało to stworzenia wyspecjalizowanych dla tego przypadku wersji praktycznie wszystkich najważniejszych komponentów.

6.3 Kompilacja obiektów

Użytkownik dostarcza na wejście Gcore zestaw funkcji w postaci pliku źródłowego zawierającego obiekt Scali. Obiekt ten musi zostać w systemie skompilowany, gdyż koncepcja Gcore zakłada wykorzystanie prekompilowanych funkcji w celu uniknięcia narzutu

czasowego związanego z każdorazowym kompilowaniem rozwiązań w celu ich oceny. Klasą zawierającą wszystkie operacje związane z procesem kompilacji jest *GcoreSourceCompiler*.

Do przeprowadzenia kompilacji wykorzystana została klasa *scala.tools.nsc.Global.Run*. Jej metoda, *compileSources*, przyjmuje listę z zawartością plików źródłowych i dokonuje ich kompilacji do bajtkodu. Wynikowe pliki klas zapisywane są w wirtualnym folderze (klasa *VirtualDirectory*). Dzięki temu pliki klas składowane są, w sposób przezroczysty dla klas odpowiedzialnych za kompilację, w pamięci operacyjnej. Opcje kompilacji, czyli między innymi wskazanie docelowej lokalizacji plików klas, ustawiane są podczas tworzenia instancji klasy *Global*, z której następnie można utworzyć instancję klasy *Run*.

W systemie wykorzystywany jest również classloader (klasa *AbstractFileClassLoader*), który ma dostęp do wspomnianego wyżej wirtualnego folderu. Jak zostało pokazane w sekcji 4.3, classloader i nazwa obiektu, uzyskana z oryginalnego kodu źródłowego przez parsowanie drzewa AST, wystarczają, by móc przetwarzać klasy i obiekty znane temu classloaderowi korzystając z mechanizmów refleksji oferowanych przez Scalę.

6.4 System typów

Zadaniem systemu typów w GP jest ograniczenie przestrzeni przeszukiwania przez wykorzystanie wiedzy dotyczącej rodzaju przetwarzanych obiektów. Jeżeli język, w którym syntetyzowane są programy, wykorzystuje typowanie, to konieczne jest również odwzorowanie tego typowania w systemie syntezy w celu zapewnienia syntaktycznej poprawności wyniku.

6.4.1 Opis

System typów w GCORE jest zmodyfikowaną wersją STGP dopuszczającą polimorfizm ze względu na relację dziedziczenia między klasami. Ma on następujące cechy:

1. każda wartość zwracana lub przyjmowana przez instrukcje ma przypisany typ powiązany z klasą Scali, do której należy,
2. dopuszczone są typy generyczne, jednak przy dodawaniu instrukcji do drzewa programu zostaną ukonkretnione do najogólniejszego możliwego w danym kontekście typu,
3. obiekt typu T może zostać podstawiony za każdy argument wymagający albo T, albo nadtypu T (relacje dziedziczenia są takie same jak w Scali),
4. jeżeli instrukcja przyjmuje jako argument funkcję o arności 0, to można za ten argument podstawić dowolną instrukcję o takim samym zwracanym typie jak zwracany typ funkcji.

Punkt 4. umotywowany był dopuszczeniem leniwej ewaluacji argumentów (np. w instrukcji warunkowej *if*), która konieczna była do zrealizowania rekurencji.

6.4.2 Algorytm unifikacji typów

Instrukcje generyczne muszą zostać ukonkretnione przed ich wstawieniem do drzewa programu. Za sprawdzenie, czy daną generyczną instrukcję I zwracającą typ t_I można ukonkretnić w taki sposób, by zwracała typ t_{arg}, odpowiada *algorytm unifikacji*. Algorytm ten jest dość skomplikowany i musi uwzględniać wiele technicznych niuansów. Jego zarys wygląda tak:

1. Sprawdzane jest, czy typ t_I zwracany przez instrukcję ma taki sam *konstruktor typu* (np. konstruktorem typu *Map[String,List[T]]* jest *Map*) jak typ docelowy t_{arg}. Typy proste (np. Int i String) nie mają konstruktorów i nie przejdą tego warunku.
2. Jeżeli konstruktory typów są takie same, to wykonywany jest rekurencyjnie punkt 1. dla wszystkich argumentów typu t_I i t_{arg} (dla przykładu z mapą argumenty typu to *String* i *List[T]*). Jeżeli dla wszystkich argumentów typu istnieje poprawne podstawienie to są ze sobą zgodne i zwracana jest lista podstawień.
3. Jeżeli konstruktory typów się różnią to sprawdzane jest, czy t_I jest typem generycznym (takie typy oznaczane są zazwyczaj literą T). Jeżeli t_I jest typem generycznym, to zwracane jest podstawienie t_I na t_{arg}. Jeżeli t_I nie jest typem generycznym, to sprawdzane są relacje dziedziczenia między typami konkretnymi. Jeżeli t_I dziedziczy po t_{arg}, to typy te są zgodne.

Podczas działania algorytmu unifikacji z wszystkich poziomów rekurencji zbierane są podstawienia typów generycznych na konkretne. W przypadku duplikatów na liście podstawień sprawdzane jest, czy zachodzi pomiędzy nimi relacja dziedziczenia. Jeżeli tak, to typ generyczny podstawiony zostanie na typ niżej położony w hierarchii i ukonkretnienie się powiodło. Jeżeli nie, to zachodzi sprzeczność i ukonkretnienie się nie powiodło. Jeżeli jakiś typ generyczny w metodzie nie ma swojego podstawienia (dzieje się tak np. dla typu T w metodzie *size[T](l:List[T]):Int*, bo porównane zostanie *Int* z t_{arg}) to podstawiany jest jako najogólniejszy typ, czyli *Any*.

6.5 Wewnętrzna reprezentacja programów

W tej sekcji omówiona zostanie implementacja reprezentacji programów w Gcore. Ogólny opis przedstawiony już został w sekcji 5.4, jednak z punktu widzenia tej sekcji dotyczył on wyłącznie instrukcji używanych w drzewie programu (podsekcja 6.5.2). Poza nimi w systemie rozróżnione są jeszcze *szablony instrukcji*.

6.5.1 Reprezentacja szablonu instrukcji

W STGP możliwe jest wykorzystywanie instrukcji generycznych, które nie mogą zostać użyte bezpośrednio w drzewie programu – jeżeli można odpowiednio ukonkretnić ich typ, to za każdym razem przed wstawieniem do drzewa wyprodukowany musi zostać wariant instrukcji dla tego konkretnego typu. Również losowe generatory stałych nie mogą zostać bezpośrednio umieszczone w drzewie, tylko wygenerowana przez nie wartość.

Elementy te jednak wciąż wchodzą w skład specyfikacji zadania i są rozpatrywane podczas losowania instrukcji do użycia w danym kontekście. Uogólnieniem w Gcore dla

elementów specyfikacji zbioru instrukcji jest klasa ***InstrTypedOp***. Elementy te będą dalej nazywane *szablonami instrukcji*. Podklasą *InstrTypedOp* dla elementów generowanych dla konkretnego typu (takich jak np. wspomniane funkcje generyczne) jest *GeneratorTypedOp*.

6.5.2 Reprezentacja instrukcji

Gdy w ScaPS'ie zachodzi konieczność utworzenia instrukcji zwracającej pewien typ, wewnątrz *domeny* (klasa *ScalaDomain*) sprawdzane jest, które szablony instrukcji mogą zostać użyte. Spośród nich wybierany jest losowo jeden, i jest on następnie transformowany do postaci, którą można umieścić w drzewie programu. Postaci tej odpowiada w Gcore klasa ***TypedOp***. Czynności wykonywane podczas transformacji różnych rodzajów szablonów instrukcji przedstawione są w Tabeli 6.1.

Na Listingu 6.1 przedstawione są w pseudokodzie najważniejsze pola obu wymienionych klas. Można zauważyć między nimi wiele podobieństw.

Listing 6.1: Skład klas *InstrTypedOp* i *TypedOp* przedstawiony w pseudokodzie.

```
1 InstrTypedOp(RET_TYPE, OP_DESC, ARG_TYPES)
2 TypedOp(PARENT_TYPE, RET_TYPE, OP, ARG_TYPES, ARGS)
```

Poniżej przedstawione jest znaczenie poszczególnych elementów:

- **RET_TYPE:** Typ zwracany przez daną instrukcję. W przypadku *TypedOp'a* zawsze jest to typ konkretny. *InstrTypedOp* może określać zwracany typ jako generyczny.
- **PARENT_TYPE:** Typ argumentu, za który podstawiony został w drzewie dany *TypedOp*. Przydatny podczas wykonywania modyfikacji na elementach drzewa programu. W przypadku korzenia jest to typ wartości zwracanej przez syntetyzowaną funkcję.
- **OP_DESC:** Cechy przechowywanej operacji, takie jak na przykład to, czy jest ona terminalem.
- **OP:** Obiekt wykonujący działania danej instrukcji i zwracający wartość w deklarowanym typie.
- **ARG_TYPES:** Typy argumentów, które przyjmuje dana instrukcja.
- **ARGS:** Argumenty w formie innych *TypedOp'ów* potrzebne do wykonania danej

Tabela 6.1: Czynności wykonywane podczas transformacji różnych rodzajów szablonów instrukcji (*InstrTypedOp*) do konkretnych instrukcji możliwych do użycia w drzewie programu (*TypedOp*).

Rodzaj szablonu instrukcji	Zamiana do konkretnej instrukcji
zmienna	trywialna transformacja
generator stałych konkretnych	wygenerowanie wartości
generator stałych generycznych	wygenerowanie wartości i ukonkretnienie typu
funkcja konkretna	trywialna transformacja
funkcja generyczna	ukonkretnienie typów

instrukcji. Dostarczane (przez utworzenie nowego obiektu) podczas konstruowania drzewa programu. Instrukcja zwracana bezpośrednio z domeny nie ma ich dołączonych.

6.5.3 Reprezentacja programu

Program powstaje poprzez utworzenie *TypedOp'a* zawierającego jako argumenty inne *TypedOp'y*. Przykład programu zbudowanego w ten sposób przedstawiony jest na Listingu 6.2, na którym *add* zawiera jako argumenty *TypedOp'y* dla stałych 1 i 5. Można zauważyć, że typ „rodzica" terminali *c1* i *c5* jest taki sam, jak typy argumentów przyjmowanych przez nieterminal *add*. W Scali *Int* dziedziczy po *Long*, tak więc przedstawiona sytuacja jest poprawna.

Listing 6.2: Wewnętrzna reprezentacja drzewa programu realizującego operację $1 + 5$. Dla poprawy czytelności zastosowano drobne uproszczenia.

```
1 val c1  = TypedOp(LONG, INT, 1, List())
2 val c5  = TypedOp(LONG, INT, 5, List())
3 val add = TypedOp(STRING, STRING, (($1+$2).toString), List(LONG,LONG), c1, c5)
```

6.6 Ewolucja

Ewolucja w GCORE działa w oparciu o operatory przeszukiwania podobne do tych wykorzystywanych w STGP [24] i zmodyfikowane w taki sposób, by uwzględnić polimorfizm ze względu na relację dziedziczenia.

Inicjalizacja populacji dokonywana jest przez algorytm *ramped half-and-half* z ustalonym poprzez parametr użytkownika pojedynczym limitem głębokości drzewa (w oryginalnej wersji jest to zakres limitów głębokości).

W GCORE możliwe do wyboru są dwie metody selekcji: turniejowa i lexicase. Ich działanie zostało opisane w sekcji 2.3.3.

W tej sekcji przez *węzeł* będziemy rozumieć pojedynczą instrukcję (*TypedOp*) w drzewie. Każdy węzeł ma przypisane jako argumenty inne węzły, tak więc konstytuuje też w pewnym sensie całą gałąź drzewa. Strategia wyboru losowego węzła w drzewie polega na wylosowaniu z rozkładem równomiernym głębokości, która dla danego węzła zdefiniowana jest jako odległość w liczbie ścieżek od korzenia. Głębokość d losowana jest ze zbioru $(0, \ldots, d_{max})$, gdzie d_{max} to głębokość najbardziej odległego od korzenia liścia. Następnie spośród wszystkich elementów znajdujących się na głębokości d wybierany jest, zgodnie z rozkładem równomiernym, jeden.

6.6.1 Krzyżowanie

Krzyżowanie w GCORE przebiega następująco:

1. Z populacji wybierane są, zgodnie z aktualnie stosowaną metodą selekcji, dwa programy X i Y.

2. W pierwszym z wybranych programów (X) wybierany jest, zgodnie z ustaloną strategią wyboru, węzeł x.
3. Z drugiego programu (Y) wyodrębniany jest zbiór S tych węzłów, które pod względem typu mogą być podstawione w miejsce x (czyli sprawdzana jest zgodność z typem argumentu, za który x jest podczepione), i jednocześnie za które podstawione może zostać x (analogiczne sprawdzenie). Ze zbioru S wybierany jest losowo węzeł y. Jeżeli zbiór był pusty, to oba osobniki niezmienione wracają do populacji.
4. Jeżeli x i y zostały wybrane, to następuje zamiana odpowiadających im poddrzew pomiędzy programami X i Y, w wyniku czego powstają programy X' i Y'. Przed zamianą poddrzew następuje wymiana typu rodzica między x i y .

6.6.2 Mutacja poddrzewa

Mutacja poddrzewa w GCORE przebiega następująco:

1. Z populacji wybierany jest jeden program X.
2. Z X wybierany jest jeden węzeł x.
3. Zgodnie z metodą *grow* generowane jest nowe poddrzewo dla typu rodzica x.
4. W miejsce poddrzewa odpowiadającego węzłowi x podstawiane jest nowo-wygenerowane poddrzewo, w wyniku czego powstaje nowy program X'. Jako typ rodzica ustawiany jest w korzeniu nowego poddrzewa typ rodzica x.

6.6.3 Mutacja jednopunktowa

W GCORE zdefiniowana została również operacja mutacji jednopunktowej, która zmienia w drzewie wyłącznie jedną instrukcję. Mutacja jednopunktowa przebiega podobnie jak mutacja poddrzewa, tylko sprawdzany jest dodatkowy warunek czy zgadzają się arności i typy argumentów pomiędzy wybranym do zamiany węzłem a instrukcjami, które potencjalnie mogą go zastąpić. Z tych dopuszczalnych instrukcji wybierana jest następnie losowo jedna.

6.7 Wykonywanie programów

W GCORE każdy nieterminal poza znacznikiem rekurencji zawiera lustro pozwalające wykonać odpowiednią metodę z obiektu podanego przez użytkownika na wejście. Jedyne co jest konieczne do wykonania danej metody to dostarczenie argumentów.

Wykonywanie programu rozpoczyna się od korzenia i jest procedurą rekurencyjną. By funkcja w korzeniu mogła zostać wykonana, muszą najpierw zostać obliczone wartości jej argumentów. Tak więc dla każdego argumentu wołana jest procedura wykonania programu tak długo, aż argument nie będzie liściem, który zawsze jest w stanie zwrócić swoją wartość.

Instrukcja rekurencji traktowana jest w systemie jako normalny nieterminal przyjmujący argumenty o takich samych typach i liczbie jak syntetyzowana funkcja. Główna różnica polega jednak na tym, że rekurencja jest wykonywana w zupełnie inny sposób

niż pozostałem nieterminale. Podczas wykonywania programu sprawdzane jest, czy aktualnie wykonywana instrukcja to znacznik rekurencji. Jeżeli tak, to obliczane są wartości jej argumentów a następnie wywoływana jest główna metoda wykonująca programy dla korzenia ze zmienionymi wartościami zmiennych.

Rozdział 7

Eksperymenty obliczeniowe

7.1 Wprowadzanie

W ramach pracy przeprowadzone zostały eksperymenty obliczeniowe mające na celu przetestowanie skuteczności GCORE i sprawdzenie, czy jest w stanie realizować syntezę programów w języku programowania Scala.

7.2 Zbiór benchmarków

W Tabeli 7.1 przedstawiona jest ogólna charakterystyka wszystkich testowych problemów. Kolejne podsekcje zawierają ich krótkie omówienie.

7.2.1 Multipleksery

Multiplekser w elektronice jest układem posiadającym 2^k wejść danych, k wejść adresowych i 1 wyjście. Wejścia adresowe wskazują razem na jedno wejście danych, które ma zostać przekazane na wyjście.

Synteza programu imitującego zachowanie multipleksera jest powszechnie wykorzystywanym benchmarkiem w GP. Benchmark ten zdefiniował John Koza [18]. Najczęściej stosowane są warianty:

- *mux3* – 2 wejścia danych i 1 wejście adresowe,
- *mux6* – 4 wejścia danych i 2 wejścia adresowe,
- *mux11* – 8 wejść danych i 3 wejścia adresowe.

Tabela 7.1: Ogólna charakterystyka testowanych problemów.

	mux3	mux6	a1	a2	fact	parser1	parser2	parser3
liczba argumentów	3	6	1	1	1	1	1	1
liczba instrukcji	4	4	1	1	11	18	18	18
liczba gen. stałych	1	1	0	0	2	6	6	6
liczba testów	8	64	27	27	11	8	5	6

Do testów systemu wykorzystane zostały tylko *mux3* i *mux6*. Dla obecnej wersji systemu pełen zestaw testów dla *mux11* przekroczył maksymalny dopuszczalny rozmiar kodu źródłowego klasy w Javie.

Z punktu widzenia syntezy wygenerowana ma zostać funkcja logiczna o sygnaturze $f(d_{2^k-1}, \ldots, d_0, a_{k-1}, \ldots, a_0)$, gdzie d_i to i-te wejście danych, a a_i to i-te wejście adresowe. Wszystkie zmienne są typu logicznego (*Boolean* w Scali). Do syntezy zostały wykorzystane następujące instrukcje: operacje logiczne ($\wedge, \vee, \neg$) oraz instrukcja warunkowa *if-else*. Stałe losowane są ze zbioru wartości logicznych $\{0, 1\}$. Zbiór testów dla problemów multiplekserów składa się ze wszystkich możliwych kombinacji wejść.

7.2.2 Algebry

Wykorzystane w eksperymentach benchmarki dotyczące algebr pochodzą z publikacji [33]. Zadaniem jest znalezienie funkcji $t(x, y, z)$, gdzie x, y i z są elementami danej algebry, która określana jest w literaturze anglojęzycznej jako *discriminator term* (autorowi polska nazwa nie jest znana). Funkcja ta opisana jest równaniem:

$$t(x, y, z) = \begin{cases} x & \text{jeżeli } x \neq y \\ z & \text{jeżeli } x = y \end{cases}$$

Ciało funkcji t złożone jest wyłącznie z jednego operatora $*$ zdefiniowanego w danej algebrze. Tabela 7.2 przedstawia działanie tego operatora odpowiednio w algebrach A_1 i A_2. Operator $*$ jest jedyną wykorzystywaną w tym problemie instrukcją. Stałe nie są wykorzystywane. Zbiór testów składa się ze wszystkich możliwych 27 kombinacji wejść dla funkcji t.

7.2.3 Silnia

Zadanie obliczenia silnii z podanej liczby jest problemem regresji symbolicznej. Funkcja silnii opisana jest wzorem $f(n) = \prod_{k=1}^{n} k$, dla $n \in \mathbb{N}$, przy czym przyjmuje się, że $0! = 1$. Powyższą definicję można zapisać rekurencyjnie jako:

$$f(n) = \begin{cases} 1 & \text{jeżeli } n = 0 \\ n \cdot f(n-1) & \text{jeżeli } n > 0 \end{cases}$$

W GCORE nie może zostać w praktyce wygenerowany program rozwiązujący ten problem iteracyjnie, co wynika z ograniczenia na czysto funkcyjną naturę generowanych programów i brak efektów ubocznych. Możliwe jest jednak rozwiązanie rekurencyjne.

Tabela 7.2: Tabele przedstawiają działanie operatora $*$ w algebrach A_1 i A_2.

A_1 $*$	0	1	2
0	2	1	2
1	1	0	0
2	0	0	1

A_2 $*$	0	1	2
0	2	0	2
1	1	0	2
2	1	2	1

Dla tego problemu zbiór instrukcji złożony jest z: standardowych operacji arytmetycznych $(+, -, \cdot, \div)$, operacji logicznych$(\wedge, \vee, \neg)$, operacji porównywania ze sobą liczb $(>, <, =)$, instrukcji warunkowej *if-else* oraz syntetyzowanej funkcji silnii $f(n)$ (rekurencja). Stałe losowane są ze zbioru liczb całkowitych z przedziału $[0, 10]$ oraz ze zbioru wartości logicznych $\{0, 1\}$. Testy obejmują n z zakresu $[0, 10]$.

7.2.4 Przetwarzanie tekstu

Programista dość często staje przed koniecznością napisania programu do uzyskania pewnych informacji zawartych w tekście lub transformacji jednego tekstu w inny. Zadaniami tego typu są na przykład przetwarzanie argumentów linii poleceń albo wczytywanie danych z pliku tekstowego.

Parsowanie tekstu nie było jeszcze rozpatrywane jako osobna rodzina benchmarków dla GP, chociaż w pracy [14] jako „tradycyjny" benchmark dla GP zaproponowana została właśnie synteza programu realizującego główną funkcjonalność Unix-owego programu *word count* (komenda *wc*), który zlicza liczbę wierszy, słów i liter w pliku. Z kolei w pracy [15] jako zbiór benchmarków proponowany jest zbiór ćwiczeniowych zdań programistycznych, wśród których kilka dotyczy przetwarzania tekstów. Świadczy to o rosnącym zainteresowaniu tego rodzaju problemami.

Na potrzeby eksperymentów zostały wymyślone własne problemy przetwarzania tekstów, i są to:

- *parser1*: zamiana ciągu znaków zawierającego liczby całkowite oddzielone spacjami na listę liczb całkowitych.
- *parser2*: na podstawie tekstu ma zostać utworzona lista zawierająca pierwsze słowa z każdego wiersza. Kolejność słów ma być taka sama jak odpowiadających im wierszy.
- *parser3*: przetransformowanie zawartości pliku właściwości (ang. *properties*) Javy do listy zawierającej wszystkie klucze i wartości (przechowywane jako tekst) w takiej samej kolejności, w jakiej wystąpiły w pliku.

Zbiór instrukcji jest wspólny dla wszystkich problemów parsowania tekstu i jest większy niż dla pozostałych problemów. Zawiera on:

- operacje na ciągach znaków (*split*, *substr*, *splitByWhitespaces*, *splitByLines*, *addStrings*, *toInt*, *size*),
- operacje na listach (*size*, *head*, *tail*, *append*, *get*, *isEmpty*, *map*),
- operacje na wartościach logicznych $(\wedge, \vee, \neg)$
- instrukcję warunkową *if-else*.

Dostępne w tych problemach stałe to: liczby całkowite z przedziału $[0, 7]$, wartości logiczne $\{0, 1\}$, pusta lista, losowe 1-literowe i 2-literowe ciągi drukowalnych znaków oraz stałe funkcyjne utworzone dla każdej instrukcji. W przypadku parserów testy są arbitralnie wybranymi zadaniami o różnym stopniu trudności.

Tabela 7.3: Parametry eksperymentu dla każdego problemu i metody selekcji. Wyjątki wymienione są w sekcji 7.3.

nazwa parametru	wartość
liczba uruchomień	100
maksymalna liczba pokoleń	200
rozmiar populacji	500
prawdopodobieństwo mutacji	0.5
prawdopodobieństwo mutacji jednopunktowej	0.1
prawdopodobieństwo krzyżowania	0.4
prawdopodobieństwo wyboru stałej zamiast zmiennej	0.1
limit głębokości drzew podczas inicjalizacji	3

7.3 Parametry

Tabela 7.3 zawiera wartości najważniejszych parametrów systemu wspólne dla wszystkich problemów. Jedynymi wyjątkami są problemy przetwarzania tekstów, w których zdecydowano się zwiększyć prawdopodobieństwo wyboru stałej do 0.25, oraz rekurencja, dla której wykonano 50 uruchomień.

7.4 Omówienie wyników

Tabela 7.4 przedstawia wyniki uzyskane przez system dla poszczególnych metod selekcji. Liczba sukcesów to liczba uruchomień algorytmu, w których udało się wygenerować optymalne rozwiązanie. Niezależnie od tego, czy ostateczne rozwiązanie było optymalne czy nie, dokonywana była próba przetransformowania go do postaci kodu źródłowego. Procent udanych kompilacji mówi o tym, ile transformacji najlepszego osiągniętego rozwiązania zakończyło się powodzeniem i wynikowy kod się kompilował.

Można dokonać kilku ciekawych obserwacji. Przede wszystkim selekcja lexicase w problemie algebr osiągała prawie 100% skuteczność, podczas gdy selekcja turniejowa na tym

Tabela 7.4: Wyniki uzyskane przez system dla poszczególnych problemów i metod selekcji.

	Procent sukcesów			Procent udanych kompilacji		
	tour4	tour7	lexicase	tour4	tour7	lexicase
mux3	100	100	100	100	100	100
mux6	100	100	100	100	100	100
a1	0	0	94	100	100	100
a2	0	0	98	100	100	100
fact	0	0	0	50	62.5	78.8
parser1	93	87	92	96	93	93
parser2	2	1	2	93	94	86
parser3	0	0	0	95	94	97

Tabela 7.5: Średnie czasy trwania obliczeń dla poszczególnych problemów i metod selekcji.

	tour4	tour7	lexicase
mux3	16	14	19
mux6	726	479	230
a1	48	41	1117
a2	46	43	958
fact	8715	3063	6744
parser1	2788	3716	2809
parser2	17420	17982	13562
parser3	16475	15412	13524

samym problemie nie zdołała rozwiązać żadnego przykładu. Na pozostałych problemach metody selekcji wypadły porównywalnie. System syntezy nie zdołał sobie w żadnej konfiguracji poradzić z problemem silnii. Do uzyskania poprawnego rozwiązania konieczne było użycie rekurencji, a rekurencja wymaga warunku, który pozwali ją zakończyć na którymś poziomie głębokości. Wyewoluowanie takiego przypadku jest jednak trudne. Gdyby syntezator uczyłby się przez wzmocnienie to najprawdopodobniej nauczyłby się, że użycie rekurencji gwarantuje przekroczenie limitu czasowego i niską ocenę.

Rozwiązania zazwyczaj bez większych problemów udawało się przetransformować do postaci kodu źródłowego, lecz jak można zauważyć, nie zawsze. Nieudane kompilacje wynikają przede wszystkim z możliwości przezroczystego wykorzystania funkcji bezargumentowej do leniwej ewaluacji podgałęzi, co ma miejsce na przykład w przypadku instrukcji warunkowej. Zmienna typu T może zostać podstawiona pod typ *Function1[T]* i konwersja na funkcję jest dokonywana podczas przekazywania argumentów. W module transformacji zabrakło obsługi wszystkich możliwych przypadków, jakie mogą w takim wypadku powstać.

W Tabeli 7.5 zamieszczone są czasy trwania obliczeń dla poszczególnych wariantów eksperymentu. Czas zużywany przez system generalnie był dość długi, szczególnie w przypadku rekurencji. Na podstawie danych w tabeli mogłoby się wydawać, że czasy rekurencji nie są takie złe w porównaniu do pozostałych, jednak średnie czasy działania dotyczą tych uruchomień, które w ogóle się zakończyły. W przypadku silnii około połowy uruchomień przekroczyło limit jednej doby i zostało zakończonych.

W Gcore wynik każdego zapytania o instrukcje pasujące do danego typu jest zapamiętywany, co pozwala bardzo szybko uzyskać wynik przy następnym zapytaniu o ten sam typ. Czas obliczeń być może zostałby dodatkowo przyspieszony, gdyby zapamiętywany był podczas ewaluacji wynik zwrócony przez dane poddrzewo, gdyż główny narzut czasowy stanowi właśnie ocena rozwiązań. Odbyłoby się to dużym kosztem pamięci, ale nawet przechowywanie informacji tylko o najczęściej wykorzystywanych poddrzewach mogłoby dać znaczne przyspieszenie.

7.5 Przykładowe rozwiązania

Przykładowe znalezione rozwiązanie dla problemu *mux6* przedstawione jest na poniższym listingu. Można zauważyć, że w rozwiązaniu występują nadmiarowe elementy, takie jak np. *and(a0, a0)*. W Gcore nie zostało zaimplementowane upraszczanie wygenerowanych programów, choć niewątpliwie dla skutecznego realizowania zadania wspomagania rozwoju oprogramowania byłoby to konieczne.

```
def mux6(x3:Boolean, x2:Boolean, x1:Boolean, x0:Boolean,
         a1:Boolean, a0:Boolean): Boolean = {
  if_else(
    if_else(
      or(a1, a0),
      and(a0, a0),
      not(a0)
    ),
    if_else(
      if_else(false, x0, a1),
      x3,
      if_else(a0, x1, x0)
    ),
    and(or(a0, x2), or(or(x1, a0), a1))
  )
}
```

Systemowi udało się również znaleźć krótki, bo zawierający tylko 28 użyć operatora $*$, *discriminator term* dla algebry A_1. Rozwiązanie tego samego problemu przedstawione w oryginalnej pracy zawierało 39 użyć operatora [33]. Zsyntetyzowana funkcja przedstawiona jest w kontekście obiektu z instrukcjami na poniższym listingu. Cały kod pochodzi wprost z wyjścia systemu. Można zauważyć, że w wielu miejscach pojawiły się kropki i średniki – jest to efekt translacji drzewa AST obiektu do postaci kodu źródłowego.

```
object Obj {
  private val algebra = List(2, 1, 2, 1, 0, 0, 0, 0, 1);
  def *(x: Int, y: Int): Int = {
    require(x.<(3));
    require(y.<(3));
    algebra((3).*(x).+(y))
  };
  def disc(x: Int, y: Int, z: Int): Int = *(*(*(*(*(x, *(*(*(y, y), x),
   *(x, *(*(z, *(y, z)), x)))), x), *(*(*(y, y), x), *(x, y))), x),
   *(*(*(*(x, *(*(y, *(x, x)), x)), y), x), *(z, y))), *(*(*(y, x), y), y))
}
```

Przykładowe znalezione optymalne rozwiązanie dla problemu *parser2* przedstawione jest na poniższym listingu wraz z odpowiednikiem w Scali, który został utworzony na podstawie implementacji zdefiniowanych dla tego problemu instrukcji.

```
// Wynik zwrócony z systemu (DSL)
def parse(text:String):List[String] =
  list_map(list_map(splitByLines(text), splitByWhitespaces), list_head)

// Odpowiednik w Scali utworzony na podstawie kodu źródłowego instrukcji
def parse(text:String):List[String] = {
  val lines:List[String] = text.split("\n").toList
  val linesWords:List[List[String]] = lines.map(x => x.split("\\s+").toList)
  linesWords.map(x => x.head)
}
```

7.6 Podsumowanie wyników

Wyniki osiągane przez system były generalnie takie, jakich się spodziewano. Jest to zasadniczo wstępna wersja systemu, w której nacisk położony został przede wszystkim na rozwiązanie kwestii technicznych związanych z syntezą w języku Scala, a nie efektywności przeszukiwania. Wykorzystanie standardowego GP przy takim rozmiarze populacji i liczbie pokoleń nie mogło gwarantować sukcesu w bardziej złożonych problemach.

Zaskakujące były jednak tak dobre wyniki osiągnięte dla selekcji lexicase. Wykonane eksperymenty wydają się potwierdzać, że ten rodzaj selekcji pozwala osiągać lepsze rezultaty niż selekcja turniejowa dla problemów, w których istotna jest pełna poprawność rozwiązań (przechodzenie wszystkich testów).

Rozdział 8

Podsumowanie

8.1 Zakres pracy

Przegląd literatury wykazał, że istnieje bardzo duże zainteresowanie tematyką automatycznej syntezy programów. Jednak mimo tak dużego zainteresowania, systemy tego typu nie stały się składnikiem każdego zintegrowanego środowiska rozwoju oprogramowania. Główne przyczyny tego stanu rzeczy to:

- bardzo duża przestrzeń możliwych rozwiązań,
- problem ze skalowalnością istniejących podejść do syntezy.

Jednym z głównych celów pracy było zbadanie, na ile możliwa jest synteza programów w statycznie kompilowanym konwencjonalnym języku programowania przy użyciu programowania genetycznego. Jako język do badania wybrana została Scala. Rozpoznana została trudność związana z długim czasem kompilacji rozwiązań. W związku z tym opracowana została koncepcja pozwalająca uniknąć konieczności ciągłej kompilacji, jednak kosztem ogólności generowanych programów, które zawężone zostały do wywołań stałego zbioru funkcji.

Przeprowadzone eksperymenty obliczeniowe pokazały, że system w obecnej postaci może skutecznie syntetyzować poprawne programy w Scali realizujące nieskomplikowane zadania. Pod względem technicznym system może zostać bez większych problemów wdrożony do praktycznego użytku, na przykład w postaci wtyczki do Eclipse'a.

Podsumowując, w ramach pracy:

- dokonano przeglądu literatury dotyczącej syntezy programów przy użyciu programowania genetycznego,
- opracowana została koncepcja syntezy programu w kompilowanym języku programowania za pomocą zbioru prekompilowanych instrukcji (funkcji),
- zrealizowany został system GCORE implementujący tę koncepcję,
- przetestowano działanie GCORE na zbiorze problemów testowych.

8.2 Dalsze kierunki badań

Zaimplementowany w ramach pracy system wykorzystuje do syntezy programów prosty wariant programowania genetycznego. Dalszym kierunkiem rozwoju mogłoby być poprawienie skuteczności przeszukiwania przestrzeni rozwiązań. Można by w tym celu spróbować wykorzystać dodatkowe informacje zbierane podczas ewaluacji fragmentów rozwiązań – młody kierunek badań określany jako „semantyczne GP" (ang. *Semantic GP*). Rozważyć można również wykorzystanie innych metod syntezy lub dołączenie do GP różnych ich elementów.

Założona na etapie projektowania koncepcja zbioru prekompilowanych funkcji mocno ogranicza podlegający ewolucji zakres języka do funkcji bezpośrednio zdefiniowanych przez użytkownika, a w ogólności do programów wyłącznie w postaci czysto funkcyjnej. Odczuwalność pierwszego z tych problemów można stosunkowo łatwo zniwelować poprzez automatyczne pobieranie funkcji z wybranych zewnętrznych pakietów lub klas. Istotne jakościowe rozszerzenie zakresu generowanych programów można uzyskać na przykład poprzez umożliwienie tworzenia w generowanych programach tymczasowych zmiennych lub wywoływania metod na konkretnych obiektach w duchu zorientowanego obiektowo GP. Co więcej wszystko wskazuje na to, że takie rozszerzenie zakresu nie pociągałoby za sobą konieczności kompilacji każdego wytworzonego na drodze ewolucji rozwiązania.

Nagłówek podawany przez użytkownika musi w obecnej wersji systemu spełniać ograniczenie polegające na tym, że wszystkie zawarte w nim typy muszą być konkretne. Innymi słowy, nie można syntetyzować funkcji generycznej. Wynika to z trudności koncepcyjnych i implementacyjnych związanych z takim przypadkiem. Jego uwzględnienie uczyniłoby system bardziej uniwersalnym.

Cel zadania w GCORE określany jest za pomocą zbioru przypadków testowych. Możliwe są jednak inne sposoby specyfikacji zadania. Jednym z bardziej interesujących rodzajów specyfikacji są *kontrakty*, czyli wyrażenia w pewnym języku formalnym opisujące zależność wyjścia od wejścia. Być może wykorzystanie tego typu specyfikacji zapewniłoby wygodniejszą interakcję programisty z syntezatorem w ramach zintegrowanego środowiska rozwoju oprogramowania.

Dodatek A

Zawartość płyty DVD

Płyta DVD dołączona do pracy zawiera:

- elektroniczną wersję pracy,
- udokumentowany kod źródłowy GCORE wraz ze wszystkimi zależnościami w postaci projektu Scali,
- wyniki eksperymentów obliczeniowych.

Bibliografia

[1] Eclipse. https://eclipse.org/.

[2] Scala documentation: Reflection overview. http://docs.scala-lang.org/overviews/reflection/overview.html.

[3] Scala framework for evolutionary computation. http://www.cs.put.poznan.pl/kkrawiec/wiki/?n=Site.Scevo.

[4] Scaps. http://www.cs.put.poznan.pl/kkrawiec/wiki/?n=Site.Scaps.

[5] Russell J. Abbott. Object-oriented genetic programming, an initial implementation. In *Proceedings of the Sixth International Conference on Computational Intelligence and Natural Computing*, Embassy Suites Hotel and Conference Center, Cary, North Carolina USA, September 26-30 2003.

[6] Alexandros Agapitos and Simon M. Lucas. Evolving efficient recursive sorting algorithms. In Gary G. Yen, Lipo Wang, Piero Bonissone, and Simon M. Lucas, editors, *Proceedings of the 2006 IEEE Congress on Evolutionary Computation*, pages 9227–9234, Vancouver, 6-21 July 2006. IEEE Press.

[7] Alexandros Agapitos and Simon M. Lucas. Learning recursive functions with object oriented genetic programming. In Pierre Collet, Marco Tomassini, Marc Ebner, Steven Gustafson, and Anikó Ekárt, editors, *Proceedings of the 9th European Conference on Genetic Programming*, volume 3905 of *Lecture Notes in Computer Science*, pages 166–177, Budapest, Hungary, 10 - 12 April 2006. Springer.

[8] Gilad Bracha and David Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 331–344, New York, NY, USA, 2004. ACM.

[9] Forrest Briggs and Melissa O'Neill. Functional genetic programming with combinators. In The Long Pham, Hai Khoi Le, and Xuan Hoai Nguyen, editors, *Proceedings of the Third Asian-Pacific workshop on Genetic Programming*, pages 110–127, Military Technical Academy, Hanoi, VietNam, 2006.

[10] Wilker Shane Bruce. Automatic generation of object-oriented programs using genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 267–272, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[11] Nathan Burles, Jerry Swan, Edward Bowles, Alexander E.I. Brownlee, Zoltan A. Kocsis, and Nadarajen Veerapen. Embedded dynamic improvement. In *Proceedings of the Companion Publication of the 2015 on Genetic and Evolutionary Computation Conference*, GECCO Companion '15, pages 831–832, New York, NY, USA, 2015. ACM.

[12] Yohann Coppel. Reflecting Scala. Semester project report, Laboratory for Programming Methods. Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland, January 2008.

[13] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, August 2012.

[14] Thomas Helmuth and Lee Spector. Word count as a traditional programming benchmark problem for genetic programming. In Dirk V. Arnold, editor, *GECCO*, pages 919–926. ACM, 2014.

[15] Thomas Helmuth and Lee Spector. General program synthesis benchmark suite. In Juan Luis Jiménez Laredo, Sara Silva, and Anna Isabel Esparcia-Alcázar, editors, *GECCO*, pages 1039–1046. ACM, 2015.

[16] Thomas Helmuth, Lee Spector, and James Matheson. Solving uncompromising problems with lexicase selection. *IEEE Transactions on Evolutionary Computation*. Accepted for future publication.

[17] J. R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In *Proc. of the Eleventh Int. Joint Conf. on Artificial Intelligence*, pages 768–774, 1989.

[18] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[19] William B. Langdon, Marc Modat, Justyna Petke, and Mark Harman. Improving 3D medical image registration CUDA software with genetic programming. In Dirk V. Arnold, editor, *GECCO*, pages 951–958. ACM, 2014.

[20] J. D. Lohn and S. P. Colombano. A circuit representation technique for automated design. *IEEE Transactions on Evolutionary Computation*, 3(3):205–219, 1999.

[21] Jason Lohn, Gregory Hornby, and Derek Linden. An evolved antenna for deployment on nasa's space technology 5 mission. In Una-May O'Reilly, Tina Yu, Rick L. Riolo, and Bill Worzel, editors, *Genetic Programming Theory and Practice II*, chapter 18, pages 301–315. Springer, Ann Arbor, 13-15 May 2004.

[22] Simon Lucas. Exploiting reflection in object oriented genetic programming. In Maarten Keijzer, Una-May O'Reilly, Simon M. Lucas, Ernesto Costa, and Terence Soule, editors, *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings*, volume 3003 of *LNCS*, pages 369–378, Coimbra, Portugal, 5-7 April 2004. Springer-Verlag.

[23] Paul Massey, John A. Clark, and Susan A. Stepney. Human-competitive evolution of quantum computing artefacts by genetic programming. *Evol. Comput.*, 14(1):21–40, March 2006.

[24] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[25] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[26] Michael Orlov and Moshe Sipper. Flight of the FINCH through the java wilderness. *IEEE Trans. Evolutionary Computation*, 15(2):166–182, 2011.

[27] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A Field Guide to Genetic Programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008.

[28] Mateusz Poszwa. Automatic improvement of programs in conventional programming languages. Master's thesis, Politechnika Poznańska, 2015. (w przygotowaniu).

[29] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 43–54, New York, NY, USA, 2015. ACM.

[30] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. Programming by sketching for bit-streaming programs. *SIGPLAN Not.*, 40(6):281–294, June 2005.

[31] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *SIGOPS Oper. Syst. Rev.*, 40(5):404–415, October 2006.

[32] Lee Spector. Assessment of problem modality by differential performance of lexicase selection in genetic programming: a preliminary report. In Kent McClymont and Ed Keedwell, editors, *1st workshop on Understanding Problems (GECCO-UP)*, pages 401–408, Philadelphia, Pennsylvania, USA, 7-11 July 2012. ACM.

[33] Lee Spector, David M. Clark, Ian Lindsay, Bradford Barr, and Jon Klein. Genetic programming for finite algebras. In Maarten Keijzer, Giuliano Antoniol, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Nikolaus Hansen, John H. Holmes, Gregory S. Hornby, Daniel Howard, James Kennedy, Sanjeev Kumar, Fernando G. Lobo, Julian Francis Miller, Jason Moore, Frank Neumann, Martin Pelikan, Jordan

Pollack, Kumara Sastry, Kenneth Stanley, Adrian Stoica, El-Ghazali Talbi, and Ingo Wegener, editors, *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1291–1298, Atlanta, GA, USA, 12-16 July 2008. ACM.

[34] Jerry Swan, Michael G. Epitropakis, and John R. Woodward. Gen-o-fix: An embeddable framework for dynamic adaptive genetic improvement programming. Technical Report CSM-195, Computing Science and Mathematics, University of Stirling, Stirling FK9 4LA, Scotland, January 2014.

[35] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.

[36] Tina Yu and Chris Clack. Polygp: A polymorphic genetic programming system in haskell. In *Proc. of the 3rd Annual Conf. Genetic Programming*, pages 416–421. Morgan Kaufmann, 1998.