Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

Instytut Informatyki

Praca dyplomowa inżynierska

na kierunku Informatyka
w specjalności Inżynieria Systemów Informatycznych

Implementacja komputerowa wybranych pojęć teorii kategorii

Sylwia Błach

Numer albumu 269268

promotor
prof. dr hab. inż. Jan Mulawka

WARSZAWA 2018

Streszczenie

Tytuł: Implementacja komputerowa wybranych pojęć teorii kategorii

Teoria kategorii stanowi niezwykle użyteczny dział matematyki, który jest wykorzystywany przez naukowców różnych dziedzin do modelowania złożonych problemów. Dzięki wysokiej abstrakcji oraz uniwersalizmowi stanowi doskonałe narzędzie do poszukiwania analogii, porównywania konceptów na pozór zupełnie do siebie niepodobnych i nieporównywalnych. To na fundamentach teorii kategorii powstały bardzo popularne obecnie języki programowania funkcyjnego. Celem tej pracy było dostarczenie biblioteki implementującej wybrane pojęcia tej dziedziny tak, aby mogła służyć badaczom do łatwego korzystania z osiągnięć teorii kategorii oraz tym samym do spopularyzowania tego działu matematyki. Ideą przyświecającą przy projektowaniu biblioteki była łatwość korzystania z niej przy jednoczesnej otwartości na własne implementacje użytkowników. Język Haskell, w którym została zapisana, szczególnie nadaje się do tego typu zastosowań.

Słowa kluczowe: *teoria kategorii, kategoria, obiekt, morfizm, Haskell.*

Abstract

Title: *Computer implementation of selected category theory concepts*

Category theory is a very useful mathematics field of study that is widely used by scientists from various backgrounds of research. Thanks to its high abstraction and universalism it is a perfect tool for seeking analogies, comparing concepts that seem to be uncomparable. Functional programming languages, commonly used nowadays, are based on category theory. The aim of this thesis is to provide a library implementing selected category theory concepts in research to serve scientists and to popularize that field of study. A program in Haskell language has been written which suits especially well for that kind of applications.

Keywords: *category theory, category, object, morphism, Haskell*

załącznik do zarządzenia nr 28/2016 r.
Rektora PW

Politechnika Warszawska

„załącznik nr 3 do zarządzenia nr 24/2016 Rektora PW

.....
miejscowość i data

.....
imię i nazwisko studenta

.....
numer albumu

.....
kierunek studiów

OŚWIADCZENIE

Świadomy/-a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie, pod opieką kierującego pracą dyplomową.

Jednocześnie oświadczam, że:

- niniejsza praca dyplomowa nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- niniejsza praca dyplomowa nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
- niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów lub tytułów zawodowych,
- wszystkie informacje umieszczone w niniejszej pracy, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami,
- znam regulacje prawne Politechniki Warszawskiej w sprawie zarządzania prawami autorskimi i prawami pokrewnymi, prawami własności przemysłowej oraz zasadami komercjalizacji.

Oświadczam, że treść pracy dyplomowej w wersji drukowanej, treść pracy dyplomowej zawartej na nośniku elektronicznym (płycie kompaktowej) oraz treść pracy dyplomowej w module APD systemu USOS są identyczne.

.....
czytelny podpis studenta"

Spis treści

1 Wstęp **3**

1.1 Wprowadzenie . . . 3

1.1.1 Teoria kategorii - źródła w filozofii . . . 3

1.1.2 Teoria kategorii i jej miejsce w matematyce . . . 4

1.2 Zakres i cel pracy . . . 5

1.3 Struktura pracy . . . 5

2 Podstawy teorii kategorii **6**

2.1 Pojęcie kategorii . . . 6

2.1.1 Relacja identycznościowa . . . 7

2.1.2 Złożenie morfizmów . . . 7

2.1.3 Łączność . . . 8

2.2 Morfizmy specjalne . . . 8

2.3 Operacje na kategoriach . . . 8

2.3.1 Kategoria odwrotna . . . 8

2.3.2 Produkt kategorii . . . 9

2.3.3 Płat kategorii . . . 10

2.4 Funktor . . . 10

2.4.1 Funktory specjalne . . . 11

2.5 Równoważność kategorii . . . 12

2.6 Teoria kategorii a teoria zbiorów . . . 12

2.6.1 Produkt w teorii kategorii . . . 13

3 Propozycja rozwiązania oparta o język Haskell **15**

3.1 Podstawowe właściwości języka Haskell . . . 15

4 Implementacja **18**

4.1 Model danych . . . 18

4.2 Opis bibliotek . . . 20

4.2.1 Moduł *Morphism* . . . 20

4.2.2 Moduł *Category* . . . 21

4.2.3 Moduł *Functor* . . . 22

4.2.4 Moduł *CategoryEquivalenceResolver* . . . 23

4.2.5 Problem porównywania morfizmów w kategorii - moduł *EquivalenceResolver* 23
4.2.6 Problem porównywania kategorii - moduł *CategoryEquivalenceResolver* 25

5 Testowanie aplikacji **28**

6 Eksperymentowanie z opracowaną aplikacją **31**
6.1 Zmniejszenie rozmiaru modelowanej struktury 31
6.2 Modelowanie sieci komputerowej za pomocą kategorii 32
6.2.1 Przykład zamodelowania sieci komputerowej za pomocą kategorii 34

7 Wnioski **37**
7.1 Zrealizowanie celu pracy 37
7.2 Propozycja dalszego rozwoju aplikacji 38

Rozdział 1

Wstęp

1.1 Wprowadzenie

1.1.1 Teoria kategorii - źródła w filozofii

Teoria kategorii jest to dział matematyki, którego początków należy szukać w filozofii. Pojęcie kategorii jest nieodłącznie związane z ontologią - działem filozofii zajmującym się badaniem bytów oraz ich cech. Filozofowie na przestrzeni wieków definiowali własne zbiory kategorii w celu sklasyfikowania wszelkich bytów, jakie mogły podlegać badaniom myślicieli.

Kategoriami zajmowali się już starożytni filozofowie. Arystoteles [2] stworzył własny zbiór kategorii, na który składały się:

- substancja – np. ten oto pies
- ilość – np. trzy
- jakość – np. dobry
- stosunek (relacja) – np. bogatszy
- miejsce – np. w domu
- czas – np. w czerwcu
- położenie – np. horyzontalne
- stan – np. ma na sobie buty
- działanie – np. jeździ
- doznawanie – np. jest mu gorąco.

Powyższe kategorie zostały podzielone na dwie grupy: pierwotne, do których należała tylko pierwsza spośród wymienionych (substancja) oraz przypadłościowe, do których zaliczały się pozostałe. Różnica pomiędzy tymi grupami polegała na tym, że byty klasyfikowane jako kategorie pierwotne mogły być podmiotami w zdaniach, zaś kategorie przypadłościowe predykatami. Arystoteles jako sposób wyróżnienia takich kategorii podał fakt, że każda z nich odpowiada na inne pytania, np. “kto?” - pies (substancja), “gdzie?” w domu (miejsce). Choć podział ten może wydawać się podyktowany lingwistyką, to w gruncie rzeczy tak nie jest. Według jego twórcy kategorie te odzwierciedlają faktycznie instniejące byty - nie pochodzą tylko

z rozróżnienia językowego. W ontologii ten podział pochodzenia pojęcia na: z konstrukcji językowych (czyli wymyślony przez człowieka) lub istniejący bez względu na sposób jego opisu, jest bardzo istotnym problemem. Dodatkowo należy dodać, że według Arystotelesa podział ten był wyczerpujący, tzn. każdy byt można było zaklasyfikować do jednej z jego kategorii.

Z kolei Immanuel Kant [2] zdefiniował własny zbiór kategorii, przy czym przyjął zupełnie inne podejście niż Arystoteles. Według Kanta niemożliwym jest jednoznaczne sklasyfikowanie wszystkich dostępnych poznaniu bytów; jego kategorie odnoszą się do rodzajów sądów, czyli sposobów, w jaki człowiek wyraża wiedzę. Klasyfikacja Kanta przedstawia się następująco:

- kategoria ilości: jedność, wielość, ogół, np. zdania typu "dla wszystkich...", "istnieje...", "niektóre..."
- kategoria jakości: realność, przeczenie, ograniczenie, np. "jest obecny...", "nie jest obecny...", "jest obecny do momentu..."
- kategoria stosunku: substancja, przyczyna, wspólnota, np. "X jest częścią Y", "X jest przyczyną Y"
- kategoria modalności: możliwość/niemożliwość, istnienie/nieistnienie, konieczność/przypadkowość, np. "niemożliwym jest...".

Natomiast współcześni filozofowie podzielają pogląd Kanta, że nie można zdefiniować kompletnego podziału wszystkich bytów na kategorie. Zamiast tego ontologia ogranicza się do podawania różnic pomiędzy pojęciami (dlaczego X można zakwalifikować jako pewną kategorię, a Y już nie).

1.1.2 Teoria kategorii i jej miejsce w matematyce

Teoria kategorii, jako dział matematyki, cechuje się formalizmem opisu swoich pojęć, w tym przede wszystkim pojęcia kategorii, w przeciwieństwie do ontologii, w której wiele filozofów definiowało na różne sposoby kategorie. Za początek teorii kategorii uważa się pracę "General theory of natural equivalences" Samuela Eilenberga i Saundersa MacLane'a. Twórcy zdefiniowali w niej najważniejsze pojęcia tej dziedziny: kategorię, funktor, transformację naturalną. Teoria kategorii szybko zyskała uznanie w świecie matematyki, ponieważ, jak się okazało, jej pojęcia są tak uniwersalne, że za ich pomocą można przedstawić wiele innych teorii matematycznych, tzn. przedefiniować pojęcia np. teorii zbiorów, logiki, topologii za pomocą pojęć teoriokategoriowych. Ten nowy rodzaj opisu zdefiniowanych już wcześniej pojęć daje bardzo ważną cechę - uniwersalizm. Dzięki temu, za pomocą teorii kategorii można badać, porównywać, szukać analogii nie tylko standardowo pomiędzy pojęciami, ale również pomiędzy całymi teoriami matematycznymi.

1.2 Zakres i cel pracy

Jak wynika z punktu 1.1.2, teoria kategorii, ze względu na swoją abstrakcyjną naturę, nie jest tematem, który często zajmuje informatyków piszących nowe oprogramowanie. Istniejące w tym zakresie rozwiązania próbują implementować jej pojęcia, jednak albo są one bardzo niekompletne (np. zdefiniowane tylko pojęcie kategorii) albo implementują tylko jakiś szczególny przypadek kategorii, np. kategoria Hask w Haskellu - typy danych i przekształcenia na nich. Istotą teorii kategorii jest zaś uniwersalizm jej pojęć. Często jest ona wykorzystywana do modelowania problemów, które to modelowanie ma dać użytkownikowi szerszy pogląd na problem - odrzucić zbędne szczegóły i skupić się na istocie. Przy tak ubogich narzędziach do tego procesu może to być dużą przeszkodą w wykorzystaniu pełnego potencjału teorii kategorii. Motywacją do zajęcia się tym zagadnieniem jest chęć zmiany tego stanu rzeczy i przez to spopularyzowanie teorii kategorii. Zakresem niniejszej pracy jest komputerowa implementacja niektórych pojeć teorii kategorii.

Celem tej pracy jest stworzenie biblioteki, która mogłaby posłużyć inżynierom, matematykom oraz innym naukowcom w wygodnym rozwiązywaniu problemów związanych z ich dziedziną zainteresowań przy użyciu teorii kategorii. Oprócz tego istotnym dążeniem jest to, aby możliwie jak najwierniej odtworzyć pojęcia teorii kategorii w języku programowania, tak aby dodatkowych, z punktu widzenia matematycznego zbędnych struktur, pól czy funkcji było jak najmniej. To zagwarantowałoby tej implementacji prostotę oraz mozliwość rozbudowywania o dalsze pojęcia teorii kategorii.

1.3 Struktura pracy

Praca składa się z sześciu rozdziałów.

W drugim rozdziale szczegółowo omówiono zagadnienia teorii kategorii - te, które zostały zaimplementowane w pracy, np. obiekt, morfizm, kategoria, morfizmy specjalne, funktor itp.

W rozdziale trzecim z kolei zaprezentowano najważniejsze właściwości języka, w jakim została napisana aplikacja składająca się na pracę - Haskella.

Rozdział czwarty stanowi opis implementacji bibliotek przygotowanych w ramach pracy.

W rodziale piątym opisano sposób, w jaki przetestowano aplikację.

W rozdziale szóstym przedstawiono przykład praktycznego wykorzystania teorii kategorii.

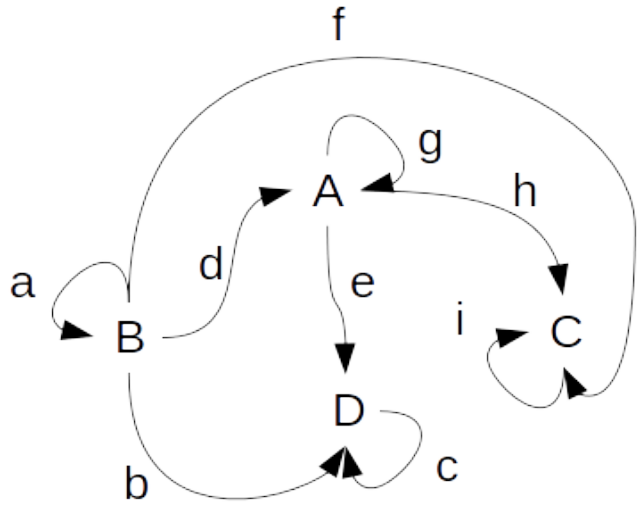
W rozdziale siódmym znalazło się podsumowanie wraz z opisem tego, jak praca mogłaby zostać rozwinięta w przyszłości.

Rozdział 2

Podstawy teorii kategorii

2.1 Pojęcie kategorii

Kategoria składa się z dwóch rodzajów pojęć: obiektów i morfizmów. Morfizmy przekształcają jeden obiekt w drugi (jest to więc relacja binarna). Struktura obiektów nie jest przedmiotem badań teorii kategorii, jedyne czym się ona zajmuje to sposób, w jaki zorganizowane są w kategorii morfizmy. W tym sensie kategorie mogą reprezentować pojęcie trudnej do zdefiniowania w innych działach matematyki struktury matematycznej. Na rysunku 3.1 przedstawiono przykład kategorii. Literami A-D oznaczono obiekty, zaś strzałki oznaczone symbolami a-i reprezentują morfizmy.



Rysunek 2.1: Przykładowa kategoria (źródło: rys. własnego autorstwa)

Obiekt, który przekształca morfizm f (z którego wychodzi strzałka) oznaczamy jako $dom(f)$ (domena), zaś obiekt, do którego domena jest przekształcana - kodomena jako $cod(f)$. Tę zależność oznacza się:

$$f : X \to Y, \quad dom(f) = X \quad \wedge \quad cod(f) = Y \tag{2.1}$$

Jak wspomiano wcześniej, na kategorię składają się obiekty i morfizmy. Każda kategoria posiada dwa zbiory: C^0 - zbiór obiektów, oraz C^1 - zbiór morfizmów. Ponadto:

$$hom(A, B) = \{f : A \to B; \quad f \in C^1; \quad A, B \in C^0\} \tag{2.2}$$

Zbiór $hom(A, B)$ nazywamy homsetem w kategorii C i jest to zbiór wszystkich morfizmów kategorii z obiektu A do B.

Aby dana struktura mogła zostać nazwana kategorią, musi spełnić określone warunki, które zostały opisane w poniższych podrozdziałach.

2.1.1 Relacja identycznościowa

Dla kategorii C prawdziwym jest warunek:

$$\forall_{A \in C^0} \quad \exists id_A \in C^1 \quad dom(id_A) = cod(id_A) = A \tag{2.3}$$

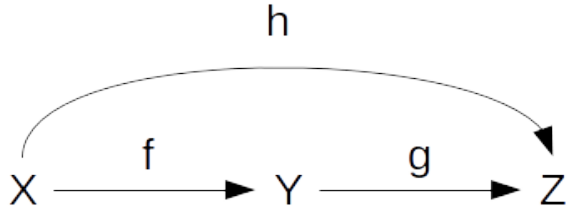
Morfizm id_A nazywany jest morfizmem identyczościowym. Przeprowadza on dowolny obiekt w niego samego. Każdy obiekt kategorii musi posiadać własny morfizm identycznościowy.

2.1.2 Złożenie morfizmów

W kategorii C zdefiniowano operację złożenia morfizmów $\circ$:

$$\forall(f, g \in C^1 \quad cod(f) = dom(g)) \quad \exists h \in C^1 \quad dom(h) = dom(f) \quad \wedge \\ cod(h) = cod(g) \quad \wedge \quad g \circ f = h \tag{2.4}$$

Każda para morfizmów, która może zostać złożona (skomponowana) (domena jednego morfizmu jest kodomeną innego), powinna mieć skojarzone ze sobą złożenie morfizmów, które również jest morfizmem o opisanych w powyższym wzorze właściwościach (rysunek 3.2)



Rysunek 2.2: Złożenie morfizmów (źródło: rys. własnego autorstwa)

Dodatkowo dla morfizmów identycznościowych spełnione są warunki:

$$\forall f \in C^1 \quad f : X \to Y \implies (f \circ id_X = f = id_Y \circ f) \tag{2.5}$$

2.1.3 Łączność

Podstawowe działanie na morfizmach - ich składanie, jest łączne.

$$\forall_{f,g,h\in C^1}(f : A \to B \quad \wedge \quad g : B \to C \quad \wedge \quad h : C \to D) \implies (h \circ g) \circ f = h \circ (g \circ f) \quad (2.6)$$

2.2 Morfizmy specjalne

Teoria kategorii definiuje następujace morfizmy specjalne:

- epimorfizm
- monomorfizm
- izomorfizm
- retrakcja
- koretrakcja

Aby morfizm $f : X \to Y$ w kategorii C można było nazwać epimorfizmem, musi spełniać warunek:

$$\forall_{g,h\in C^1}(g, h : Y \to Z \quad \wedge \quad g \circ f = h \circ f) \implies (g = h) \quad (2.7)$$

Monomorfizm $f : X \to Y$ spełnia następujący warunek:

$$\forall_{g,h\in C^1}(g, h : Z \to X \quad \wedge \quad f \circ g = f \circ h) \implies (g = h) \quad (2.8)$$

Izomorfizm $f : X \to Y$ w kategorii C musi spełniać poniższy warunek:

$$\exists_{g\in C^1} g : Y \to X \quad \wedge \quad g \circ f = id_X \quad \wedge \quad f \circ g = id_Y \quad (2.9)$$

Retrakcja $f : X \to Y$ spełnia następujący warunek:

$$\exists_{g\in C^1} g : Y \to X \quad \wedge \quad f \circ g = id_Y \quad (2.10)$$

Koretrakcja $f : X \to Y$ w kategorii C musi spełniać poniższy warunek:

$$\exists_{g\in C^1} g : Y \to X \quad \wedge \quad g \circ f = id_X \quad (2.11)$$

2.3 Operacje na kategoriach

2.3.1 Kategoria odwrotna

Z kategorii C można uzyskać kategorię do niej odwrotną, stosując operację odwracania. Zależność pomiędzy kategorią C a kategorią do niej odwrotną C^{op}:

- obiekty - obiekty w kategorii odwrotnej są takie same jak w kategorii bazowej - C^0

- morfizmy - morfizmy w kategorii odwrotnej uzyskujemy odwracając morfizmy C^1 z kategorii bazowej:

$$\forall_{f^{op} \in C^{op^1}} f^{op} : Y \to X \quad \exists_{f \in C^1} f : X \to Y \tag{2.12}$$

  gdzie C^{op^1} - zbiór morfizmów kategorii C^{op}
- złożenia - do złożeń, jako że są one szczególnymi przypadkami morfizmów, stosuje się zasadę opisaną powyżej; dzięki temu w kategorii odwrotnej zachowany jest aksjomat o złożeniach
- relacje identyczności - tak samo jak morfizmy - należy odwrócić identyczności z kategorii bazowej, przy czym, ponieważ $dom(id) = cod(id)$ dla każdego morfizmu identycznościowego id w kategorii, to w ostateczności pozostają takie same.

Kategoria odwrotna jest kategorią dualną do swojej kategorii bazowej. W teorii kategorii prawie każde pojęcie ma swój dualny odpowiednik. Taka dualna struktura często wykazuje bardzo podobne (symetryczne) właściwości do bazowej, np. dany morfizm, który jest epi w kategorii C, jest mono w kategorii C^{op}.

2.3.2 Produkt kategorii

Dla dwóch kategorii C oraz D zdefiniowano ich produkt $C \times D$ o poniższych właściwościach:

- obiekty - zbiór obiektów produktu kategorii $C \times D$

$$(C \times D)^0 = \{< A_C, A_D >, \quad A_C \in C^0 \quad \wedge \quad A_D \in D^0\} \tag{2.13}$$

- morfizmy - zbiór morfizmów w kategorii będącej produktem przedstawia się następująco:

$$(C \times D)^1 = \{< f_C, f_D >, \quad f_C \in C^1 \quad \wedge \quad f_D \in D^1\} \tag{2.14}$$

- złożenia - złożenia w kategorii produktu kategorii C i D spełniają zależność:

$$\begin{gathered}\forall(f_C, g_C, h_C \in C^1 \quad \wedge \quad f_C : X_C \to Y_C \quad \wedge \quad g_C : Y_C \to Z_C \quad \wedge \\ h_C : X_C \to Z_C \quad \wedge \quad X_C, Y_C, Z_C \in C^0 \quad \wedge \quad g_C \circ f_C = h_C) \\ \forall(f_D, g_D, h_D \in D^1 \quad \wedge \quad f_D : X_D \to Y_D \quad \wedge \quad g_D : Y_D \to Z_D \quad \wedge \\ h_D : X_D \to Z_D \quad \wedge \quad X_D, Y_D, Z_D \in D^0 \quad \wedge \quad g_D \circ f_D = h_D) \\ \exists_{<f_C,f_D>,<g_C,g_D>,<h_C,h_D> \in (C \times D)^{op^1}} < g_C, g_D > \circ < f_C, f_D > = < h_C, h_D >\end{gathered} \tag{2.15}$$

- relacje identyczności - relacje identyczności wyglądają analogicznie jak relacje złożenia - są to produkty kartezjeńskie relacji identycznościowych z kategorii C oraz D.

2.3.3 Płat kategorii

Dla kategorii C oraz obiektu $X \in C^0$ zdefiniowano operację płat kategorii X nad C o następujących właściwościach:

- obiekty - zbiór obiektów w kategorii C/X:

$$(C/X)^0 = \{f, f \in C^1 \quad \wedge \quad cod(f) = X\} \tag{2.16}$$

  jest to zbiór wszystkich morfizmów o kodomenie X w kategorii C
- morfizmy - zbiór morfizmów:

$$(C/X)^1 = \{f, f \in C^1 \quad \wedge \quad g \circ f = h; \quad g, h \in (C/X)^0\} \tag{2.17}$$

- złożenia - złożenia zdefiniowane są standardowo
- relacje identyczności - relacją identyczności dla $f \in C^1 \wedge f \in (C/X)^1$ w kategorii C/X jest relacja identyczności domeny morfizmu f z kategorii C.

2.4 Funktor

Funktor jest to mapowanie pomiędzy dwiema kategoriami C i D, oznaczane jako:

$$F : C \to D \tag{2.18}$$

Funktor mapuje obiekty i morfizmy kategorii C do obiektów i morfizmów kategorii D. Zmapowany obiekt a kategorii C oznaczamy jako $F(a)$; mapowanie morfizmu f z kategorii C wygląda analogicznie: $F(f)$. Mapowania te muszą spełniać następujące właściwości:

- każdy obiekt/morfizm z kategorii źródłowej musi mieć określony odpowiedni obiekt/morfizm z kategorii docelowej

$$\forall a \in C^0 \quad \exists b \in D^0 \quad F(a) \quad = \quad b \tag{2.19}$$

$$\forall f \in C^1 \quad \exists g \in D^1 \quad F(f) \quad = \quad g \tag{2.20}$$

- mapowania morfizmów muszą zachowywać dziedzinę i kodziedzinę, co jest rozumiane następująco:

$$\forall f \in C^1 \quad F(dom(f)) = dom(F(f)) \quad \wedge \quad F(cod(f)) = cod(F(f)) \tag{2.21}$$

- mapowanie musi zachowywać identyczności:

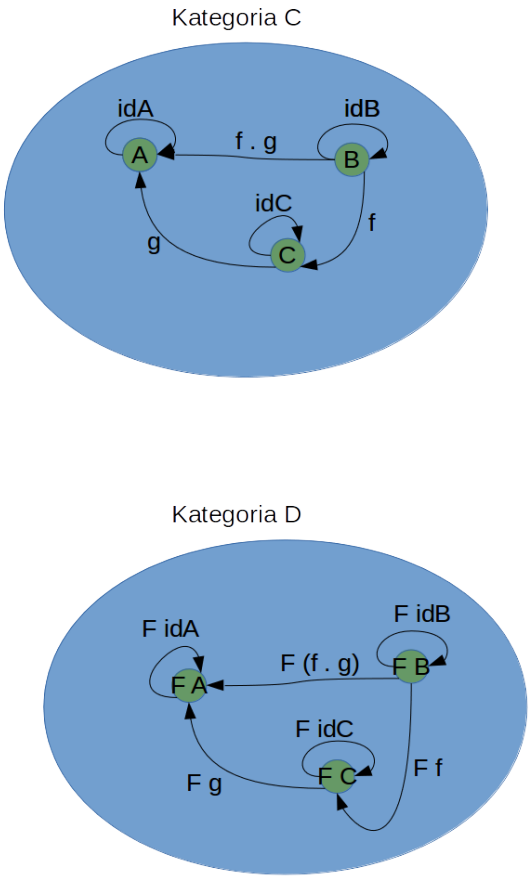
$$\forall a \in C^0 \quad F(1_a) = 1_{F(a)} \tag{2.22}$$

- mapowanie musi zachowywać złożenia:

$$\forall f, g, h \in C^1 \quad g \circ f = h \quad \Longrightarrow \quad F(g \circ f) = F(g) \circ F(f) \tag{2.23}$$

Powyższe własności funktora sprawiają, że ma on bardzo ważną właściwość, jaką jest zachowywanie struktury kategorii. Funktor może mapować wiele obiektów/morfizmów kategorii źródłowej do jednego obiektu/morfizmu kategorii docelowej; może również mapować tylko do części obiektów/morfizmów kategorii docelowej. Pewnym jest jednak, że mapowana kategoria będzie miała identyczną strukturę jak część kategorii, do której jest mapowana.

Na rysunku 2.3 przedstawiono przykładowy funktor. Operacja $g.f$ oznacza złożenie morfizmów (kolejność morfizmów odwrotna niż dla operatora $\circ$).



Rysunek 2.3: Przykładowy funktor

2.4.1 Funktory specjalne

Wyróżnia się następujące funktory specjalne:

- funktor pełny; funktor $F : C \to D$ jest pełny, gdy spełnia równanie:

$$\forall a, b \in C^0 \quad Hom_C(a, b) \to Hom_D(F(a), F(b)) - surjekcja \tag{2.24}$$

- funktor wierny; funktor $F : C \to D$ można określić mianem wiernego, gdy posiada właściwość:

$$\forall a, b \in C^0 \quad Hom_C(a, b) \to Hom_D(F(a), F(b)) - injekcja \tag{2.25}$$

- funktor gęsty; aby funktor $F : C \to D$ był gęsty, musi spełniać poniższe równanie (operator $\cong$ oznacza izomorfizm obiektów):

$$\forall a \in D^0 \quad \exists b \in C^0 \quad d \cong F(b) \tag{2.26}$$

2.5 Równoważność kategorii

Dwie kategorie C i D są równoważne, gdy spełniają warunek:

$$\exists C \to D, G : D \to C \quad F \circ G = id_D \quad \wedge \quad G \circ F = id_C \tag{2.27}$$

Warunek mówi tyle, że aby uznać dwie kategorie za równoważne, muszą istnieć dwa funktory F i G mapujące obiekty i morfizmy pomiędzy tymi kategoriami tak, że złożenie tych funktorów jest morfizmem identycznościowym odpowieniej kategorii. Złożenie funktorów jest to operacja polegająca na sekwencyjnym zmapowaniu kategorii (analogicznie jak jest rozumiana dla morfizmów). Morfizm identycznościowy kategorii pochodzi z kategorii kategorii Cat, której obiektami są kategorie, zaś morfizmami funktory. W praktyce powyższy warunek równoważności kategorii okazuje się niepraktyczny. Aby dowieść za jego pomocą równoważność kategorii C i D, należy znaleźć dwa funktory oraz porównać je z morfizmami identycznościowymi odpowiednich kategorii. Można do tego celu wykorzystać inny warunek równoważności kategorii przedstawiony poniżej:

$$\exists F : C \to D \quad F - \text{funktor gęsty, wierny i pełny} \quad \Longrightarrow \quad C \text{ równoważne } D \tag{2.28}$$

W tym wypadku należy znaleźć jeden funktor pomiędzy sprawdzanymi pod kątem równoważności kategoriami i dowieść, że spełnia odpowiednie warunki.

2.6 Teoria kategorii a teoria zbiorów

Teoria kategorii może z powodzeniem zastępować inne teorie w opisie różnych pojęć matematycznych. Jej pozornie ubogi, ale wysoce abstrakcyjny aparat pojęciowy okazał się wielką zaletą w tym zadaniu. Jednocześnie niektóre teorie matematyczne

okazują się być szczególnymi przypadkami teorii kategorii. Do tego grona można zaliczyć teorię zbiorów.

Zbiory i funkcje to szczególne przypadki kolejno obiektów oraz morfizmów. Przykład kategorii: zbiór liczb naturalnych jako obiekt; funkcja dodawania i mnożenia jako morfizmy przekształcające zbiór liczb naturalnych w niego samego, funkcja $id(n) = n$ jako morfizm identycznościowy. Morfizmy specjalne mają swoje odpowiedniki w teorii zbiorów:

- epimorfizm - surjekcja w teorii zbiorów
- monomorfizm - injekcja
- izomorfizm - bijekcja.

Różnica pomiędzy oboma działami matematyki polega na tym, że w teorii kategorii nie rozpatrujemy zawartości obiektów (czyli zbiorów w tym przypadku), podczas gdy w teorii zbiorów jest to konieczne. To ograniczenie nie sprawia jednak, że nie da się zdefiniować niektórych pojęć wywodzących się z teorii mnogości, nawet tych korzystających w swojej pierwotnej definicji z relacji zawierania. Przykład podano niżej.

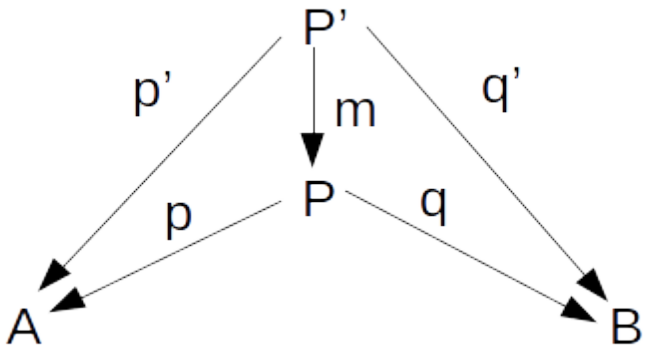
2.6.1 Produkt w teorii kategorii

Produkt kartezjański zbiorów A i B w teorii zbiorów definiowany jest następująco:

$$A \times B = \{(a, b); \quad a \in A \quad \wedge \quad b \in B\} \tag{2.29}$$

Jak wspomniano wcześniej, aby zdefiniować taki produkt, należy dostarczyć definicję relacji zawierania $\in$. Definicja takiego produktu w teorii kategorii, jako że nie rozpatrujemy "wnętrza" obiektu, którym w tym wypadku jest zbiór, nie może opierać się na tej relacji. W zamian za to wykorzystuje ona tzw. *uniwersalną konstrukcję*. Uniwersalna konstrukcja to sposób opisywania pojęć oraz dowodzenia twierdzeń wykorzystywany w teorii kategorii. Definiowanie pojęć polega na pokazaniu, jaką strukturę morfizmów musi zawierać kategoria, aby została sklasyfikowana jako dany byt matematyczny. I tak produkt kartezjański został pokazany na rysunku 2.4 (zbędne informacje, np. identyczności, pominięto).

Produktem jest tutaj obiekt P. Jest on domeną dwóch morfizmów p oraz q. Morfizmy te są interpretowane jako projekcje (rzutowania) produktu na jego składowe: A i B. W algebraicznych typach danych byłyby to funkcje zwracające kolejne składniki produktu, np. n-ty element krotki. Dodatkowo obiekt będącys produktem jest unikalny. W sytuacji, gdy w kategorii występuje wiele obiektów z projekcjami przekształcającymi w zadane obiekty, dokonuje się wyboru odpowiedniego poprzez sprawdzenie, czy nie występuje morfizm m, przekształcający jednego kandydata do miana produktu w inny. Ten spośród kandydatów, który nie jest domeną żadnego tego typu morfizmu, jest produktem. Dodatkowo muszą istnieć morfizmy z każdego



Rysunek 2.4: Definicja produktu kartezjańskiego w teorii kategorii (źródło: rys. własnego autorstwa)

kandydata do miana produktu do faktycznego produktu. Morfizm m oznacza tutaj "najlepszość" obiektu P w wyścigu do miana produktu obiektów A i B.

Poniżej przedstawiono warunki, jakie musi spełniać kategoria z produktem.

$$\exists_{f,g} dom(f) = dom(g) = P \quad \wedge \quad cod(f) = A \quad \wedge \quad cod(g) = B \tag{2.30}$$

$$\forall_{P' \neq P} \exists_{f,g,f',g'} (dom(f) = dom(g) = P \quad \wedge \quad cod(f) = A = cod(f') \quad \wedge \\ cod(g) = B = cod(g') \quad \wedge \quad dom(f') = P' = dom(g')) \implies (f' = m \circ f \wedge g' = m \circ g) \tag{2.31}$$

Porównując oba sposoby zapisu produktu, teoriokategoriowy i z teorii mnogości, rodzi się pytanie, jakie są zalety jednego i drugiego rozwiązania. To drugie jest z pewnością bardziej intuicyjne i zwięzłe. Co więc daje nam definicja produktu zapisana przy użyciu uniwersalnej konstrukcji? Przede wszystkim uniwersalizm. Taka definicja odnosi się do każdego rodzaju produktu: zarówno podanego wyżej produktu kartezjańskiego zbiorów, jak i np. produktu przestrzeni topologicznych czy grup. Mówi ona bowiem o *interfejsie* produktu - jest to "coś" z czego można uzyskać jego składowe za pomocą projekcji. Przez to, że teoria kategorii nie wnika w zawartość obiektów, nie trzeba dostarczać dodatkowych informacji odnośnie np. relacji produktu i jego składowych, tego, jak zdefiniowane są operacje na produkcie, w teorii przestrzeni topologicznych zaś tego, jak wygląda jego topologia. Teoria kategorii odrzuca te rozważania na rzecz większej abstrakcji. Dzięki temu można łatwiej zauważyć pewne prawidłowości, np. czym tak naprawdę jest produkt.

Rozdział 3

Propozycja rozwiązania oparta o język Haskell

3.1 Podstawowe właściwości języka Haskell

Haskell jest językiem programowania ogólnego przeznaczenia. Został nazwany na cześć amerykańskiego matematyka Haskella Curry'ego, zajmującego się m.in. logiką kombinatoryczną. Powstał on na Uniwersytecie w Glasgow; stąd też nazwa jego najpopularniejszego kompilatora - ghc (*"Glasgow Haskell compiler"*).

Haskell jest językiem czysto funkcyjnym. Niesie to za sobą wiele konsekwencji. Przede wszystkim podstawową jednostką w Haskellu jest wyrażenie. Wyrażenia cechują się tym, że przyjmują argumenty na wejściu, obliczają wartość oraz zwracają wynik na wyjście. Wyrażenie to nic innego jak opis zależności pomiędzy argumentami a wyjściem. Każda linia kodu napisana w Haskellu jest wyrażeniem i powinna spełniać powyższe wymogi.

Powtarzalne części kodu można grupować w funkcje. W paradygmacie funkcyjnym preferowanym typem funkcji jest funkcja całkowita, czyli zdefiniowana na całej dziedzinie. W ogólności w jezykach programowania można mieć do czynienia z funkcjami częściowymi, np. funkcja dzieląca dwie liczby jest niezdefiniowana dla dzielnika równego 0. Dodatkowo funkcje napisane w Haskellu są funkcjami w ujęciu matematycznym - wywołane dowolną ilość razy dla tych samych argumentów zawsze zwrócą ten sam wynik. Niesie to ze sobą wiele istotnych zalet - wynik dla takiej funkcji można zapamiętywać, co jest bardzo przydatne w systemach współbieżnych (brak stanowości, sekcji krytycznych) oraz ułatwione testowanie (wiadomo, jaki wynik powinna zawsze zwrócić funkcja).

Haskell stanowi implementację rachunku lambda. Jak wspomniano wcześniej, podstawowymi jednostkami są wyrażenia, które można składać oraz ewaluować. Składanie wyrażeń polega na obliczeniu wyniku pierwszego wyrażenia oraz podaniu go na wejście drugiego. Ewaluacja to zaś po prostu obliczenie wartości wyrażenia. Dzięki powyższym cechom, Haskell posiada kolejną właściwość - częściową aplikację

argumentów. Właściwość ta umożliwia aplikowanie do funkcji kolejnych argumentów w różnych miejscach programu (odłożenie w czasie wyewaluowania wartości funkcji).

Kolejną istotną cechą Haskella jest to, że funkcje są traktowane jak wyrażenia pierwszej klasy. Oznacza to, że z syntaktycznego punktu widzenia nie ma różnicy pomiędzy wyrażeniem (np. zdefiniowaniem stałej) a funkcją. Funkcje można przekazywać jako argument innych funkcji (funkcje wyższego rzędu), można je zwracać, przypisywać do stałych itp.

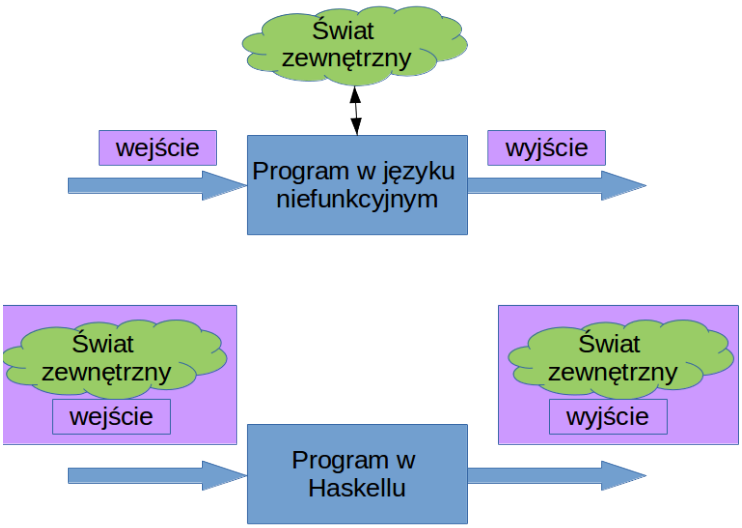
Inną ciekawą cechą Haskella jest tzw. leniwość. Polega ona na tym, że wyrażenia w kodzie Haskella są ewaluowane dopiero wtedy, gdy jest to konieczne, np. gdy trzeba wypisać obliczaną wartość na standardowe wyjście, albo porównać ją w wyrażeniu warunkowym, aby wybrać odpowiednią ścieżkę przetwarzania. Jeśli nie zachodzi taka potrzeba, wartość nie jest obliczana, a dane wyrażenie lub wyrażenia są przekazywane dalej w formie deklaratywnej. Dzięki temu można zdefiniować oraz wykonywać niektóre operacje na nieskończonych listach.

Następną, nie mniej ważną cechą Haskella, jest zwięzłość przy jednoczesnej ekspresyjności. Dzięki temu kod napisany w Haskellu w większości przypadków liczy mniej linii kodu niż napisany w innych językach oraz jest dużo czytelniejszy. Umożliwiają to takie właściwości, jak: zaawansowana funkcjonalność dopasowywania wzorca, zwięzły, przypominający zapis matematyczny sposób definiowania list (tzw. *"list comprehensions"*) oraz minimalistyczna składnia (np. przy wywołaniu funkcji nie trzeba otaczać argumentów nawiasami okrągłymi ze względu na to, że wywołanie funkcji jest powszechnym wyrażeniem w językach funkcyjnych).

Haskell posiada możliwość definiowania własnych typów, w tym typów generycznych. System typów w Haskellu jest jednak bogatszy niż w np. w Javie czy C++. Włączając odpowiednie rozszerzenie języka można bowiem osiągnąć pełną funkcjonalność Systemu F [1], np. zdefiniować funkcję generyczną, która w tym samym wywołaniu ten sam argument potraktuje jako inny typ w różnych miejscach, w których ten argument jest używany. Ponadto system typów w Haskellu zawiera zwięzłe w użyciu implementacje algebraicznych typów danych: sumy oraz produktu. Dzięki temu można w łatwy sposób tworzyć własne, rozbudowane typy w oparciu o kompozycję już istniejących.

Haskell, jako język czysto funkcyjny, w szczególny sposób traktuje operacje, w których następuje modyfikacja zewnętrznych dla programu zasobów, tzw. efekty uboczne. Efektami ubocznymi jest np. wypisanie ciągu znaków na standardowe wyjście, wysłanie żądania HTTP, modyfikacja pliku w systemie plików itp. Jak wspomniano wcześniej, w paradygmacie funkcyjnym stanowość jest niepożądana, ponieważ zaburza przejrzystość referencyjną (*"referential transparency"*). Funkcja nieposiadająca tej cechy, wywołana wiele razy dla tych samych argumentów w ogólnym przypadku zwróci inne wyniki. Efekty uboczne są zaś niczym innym jak modyfikacją globalnego stanu. Aby uniknąć tych niepożądanych cech, problem ten rozwiązano poprzez "opakowanie" operacji "nieczystych", czyli zawierających efekty uboczne w monady. Monady przechowywują informację nie tylko o wykonywanych

operacjach, ale również o stanie "świata zewnętrznego" w momencie ich wywołania (rysunek 2.1). Ponadto zdefiniowane są dla nich zasady składania monad (czyli łączenia operacji ze składanych monad) oraz sposób obsługi błędów.



Rysunek 3.1: Wywołanie operacji z efektami ubocznymi w języku niefunkcyjnym vs. wywołanie monady w Haskellu (źródło: rys. własnego autorstwa)

Jak widać, język ten ma wiele ciekawych właściwości, które mogą okazać się przydatne dla zagadnień implementacji maszynowej teorii kategorii.

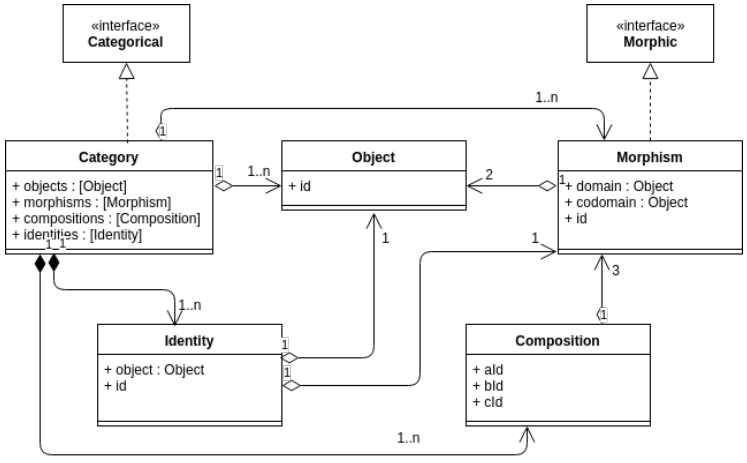
Rozdział 4

Implementacja

W ramach tej pracy inżynierskiej zostały stworzone dwie niezależne biblioteki zawierające implementację kategorii oraz wybranych operacji na nich.

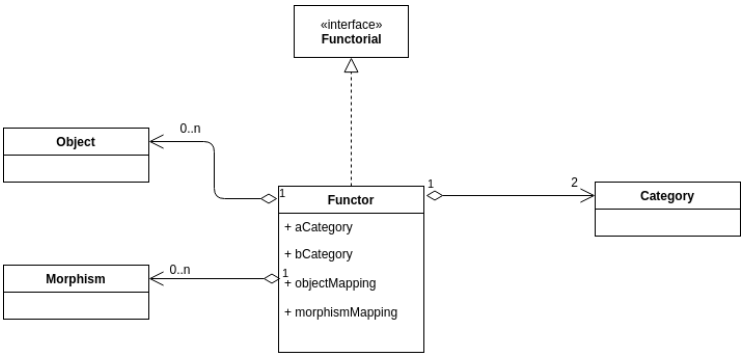
4.1 Model danych

Diagram klas biblioteki *Category* został przedstawiony na rysunku 4.1 oraz 4.2. Na rysunku 4.1 znalazły się klasy wchodzące w skład definicji kategorii, zaś na rysunku 4.2 - w skład definicji funktora.



Rysunek 4.1: Diagram klas biblioteki *Category* dla kategorii (źródło: rys. własnego autorstwa)

Model danych biblioteki *StructuralCategory* nie różni się znacząco od przedstawionego powyżej. Różnica w modelu danych obu bibliotek polega na innym sposobie identyfikowania obiektów reprezentowanych przez typ *Object*. W bibliotece *Category* każdy obiekt ma nadany identyfikator *id* o polimorficznym typie. W kategoriach matematycznych obiekty nie są reprezentowane przez żadne identyfikatory (ich zawartość nie jest w ogóle rozważana w tej dziedzinie). W przypadku języka programowania i przetwarzania komputerowego zaszła jednak potrzeba nadania obiektom tego pola - w przecienym przypadku każda instancja tego typu byłaby nieodróżnialna - klasa *Object* byłaby singletonem. Ten sam problem dotyczy morfizmów.



Rysunek 4.2: Diagram klas biblioteki *Category* dla funktora (źródło: rys. własnego autorstwa)

Teoria kategorii nie zabrania istnienia wielu morfizmów o tej samej dziedzinie i kodziedzinie (homset jest zbiorem), należało więc nadać specjalne pole dla odróżnienia ich, pomimo że w samej teroii kategorii ne ma mowy o żadnych identyfikatorach. Obiekty i morfizmy są tam odróżniane tylko i wyłącznie na podstawie struktury kategorii. Biorąc to pod uwagę, biblioteka *StructuralCategory* rozwiązuje problem identyfikatorów połowicznie. Obiekty nie posiadają tutaj własnych identyfikatorów, lecz składają się na nie dwa zbiory identyfikatorów morfizmów: pierwszy zbiór dotyczy morfizmów o kodziedzinie w danym obiekcie (morfizmy wchodzące), a drugi morfizmów o dziedzinie w rozważanym obiekcie (morfizmy wychodzące). Model danych tej biblioteki lepiej oddaje istotę kategorii, jako bytu badanego pod kątem jego struktury (układu morfizmów).

W obu bibliotekach zdefiniowano interfejsy *Categorical* i *Morphic*, w bibliotece *Category* znalazł się również interfejs *Functorial*. Interfejsy zawierają tylko te sygnatury metod, które są niezbędne do nazwania danego typu implementującego go odpowiednio kategorią, morfizmem oraz funktorem. Dzięki temu użytkownik może sobie zdefiniować wedle uznania model danych i powiązania między typami. Obowiązuje go tylko konieczność zaimplementowania funkcji o określonej sygnaturze.

Morfizm zawiera, oprócz identyfikatora, dwa obiekty typu *Object* reprezentujące kolejno dziedzinę i kodziedzinę. *Category* składa się z czterech list: pierwszej obiektów typu *Object*, drugiej *Morphism*, trzeciej *Composition*, a czwartej *Identity*. Pierwsze dwie reprezentują standardowo zbiory C^0 i C^1. Typ *Composition* zawiera informacje o złożeniach; dla obu bibliotek ma on postać krotki o długości 3, zawierającej identyfikatory obiektów typu *Morphism* biorących udział w złożeniu. *Identity* reprezentuje z kolei identyczność - jest to krotka o długości 2, zawierająca obiekt typu *Object* oraz identyfikator obiektu typu *Morphism*, czyli obiekt wraz z informacją o morfizmie będącym jego relacją identyczności. Implementacja klasy *Functorial* - *Functor*, oprócz dwóch kategorii reprezentujących mapowane kategorie, zawiera mapowania obiektów oraz morfizmów. Mapowania te mogą być wyrażone na dwa sposoby: albo jako funkcja pobierająca obiekt/morfizm i zwracająca jego mapowanie, albo jako tablica mieszająca przechowująca pary klucz - wartość, gdzie klucz -

obiekt/morfizm mapowany, wartość - obiekt/morfizm, do którego następuje mapowanie.

Dla ułatwienia w dalszej części pracy zamiast określenia obiekt typu *Object/Morphism/Category/Identity/Composition/Functor* będą używane kolejno określenia obiekt/morfizm/kategoria/identyczność/kompozycja/funktor.

Zaimplementowana biblioteka zawiera jeszcze jedno ograniczenie - można zdefiniować tylko kategorie o skończonej ilości obiektów i morfizmów. Zaś w jej matematycznym odpowiedniku istnieją kategorie:

- lokalnie małe - takie, których wszystkie homsety są zbiorami (czyli nie są zbiorami zbiorów albo nie zawierają w sobie zbioru zbiorów)
- duże - ich homsety mogą być zbiorami zbiorów.

Te kategorie w ogólności mogą mieć nieskończenie wiele obiektów i morfizmów. Opisywane odstępstwo od matematycznego modelu w implementacji wynika z oczywistego faktu o ograniczonej ilości pamięci w komputerach.

4.2 Opis bibliotek

Poniżej przedstawiono opis najważniejszych funkcji zaimplementowanych w bibliotekach *Category* i *StructuredCategory*.

Oznaczenie "*nazwa* : *Typ*" oznacza zmienną o nazwie *nazwa* i typie *Typ*. Gdy typ zmiennej jest małą literą alfabetu - zmienna o typie polimorficznym.

4.2.1 Moduł *Morphism*

Poniżej znajdują się nazwy funkcji z klasy typów *Morphic*, wraz z najważniejszymi informacjami dotyczącymi ich implementacji z modułu *Morphism*:

1. **dom**
   - argumenty: *morfizm* : *Morphism*
   - wartość zwracana: dziedzina morfizmu *morfizm*
2. **cod**
   - argumenty: *morfizm* : *Morphism*
   - wartość zwracana: kodziedzina morfizmu *morfizm*
3. **(.)** - operator złożenia
   - argumenty: *morfizmA* : *Morphism*, *morfizmB* : *Morphism*, *id* : a
   - wartość zwracana: gdy możliwe jest złożenie *morfizmA* i *morfizmB* (w tej kolejności) to zwraca to złożenie opakowane w monadę *Maybe* nadając mu identyfikator *id*; gdy niemożliwe jest złożenie tych dwóch morfizmów zwracany jest *Nothing*;
4. **eq**
   - argumenty: *morfizmA* : *Morphism*, *morfizmB* : *Morphism*
   - wartość zwracana: wartość typu *Boolean*, bedąca wynikiem porównania argumentów

– opis: funkcja porównuje morfizmy po identyfikatorach

4.2.2 Moduł *Category*

Poniżej znajdują się nazwy funkcji z klasy typów *Categorical*, wraz z najważniejszymi informacjami dotyczącymi ich implementacji z modułu *Category*:

1. **obj**
   - argumenty: *kategoria* : *Category*
   - wartość zwracana: zbiór C^0 kategorii *kategoria*
2. **morph**
   - argumenty: *kategoria* : *Category*
   - wartość zwracana: zbiór C^1 kategorii *kategoria*
3. **hom**
   - argumenty: *kategoria* : *Category*, *obiektA* : *Object*, *obiektB* : *Object*
   - wartość zwracana: $hom(obiektA, obiektB)$ w kategorii *kategoria*
4. **id**
   - argumenty: *obiekt* : *Object*, *kategoria* : *Category*
   - wartość zwracana: morfizm identycznościowy obiektu *obiekt* w kategorii *kategoria*
5. **(.)**
   - argumenty: *morfizmA* : *Morphism*, *morfizmA* : *Morphism*, *kategoria* : *Category*
   - wartość zwracana: morfizm będący złożeniem morfizmów *morfizmA* i *morfizmA* w kategorii *kategoria*, opakowany w monadę *Maybe*; gdy nie da się złożyć morfizmów wejściowych zwraca *Nothing*
6. **inEpiRelation**
   - argumenty: *sprawdzanyMorfizm* : *Morphism*, *kategoria* : *Category*, *morfizmA* : *Morphism*, *morfizmB* : *Morphism*
   - wartość zwracana: funkcja sprawdza, czy *sprawdzanyMorfizm* jest w realcji epi (czy jest kandydatem na bycie epimorfizmem) z pozostałymi morfizmami przekazywanymi do funkcji w *kategoria*, tzn. czy spełnia równanie (3.7) dla tych konkretnych morfizmów
7. **inMonoRelation**
   - argumenty: *sprawdzanyMorfizm* : *Morphism*, *kategoria* : *Category*, *morfizmA* : *Morphism*, *morfizmB* : *Morphism*
   - wartość zwracana: funkcja sprawdza, czy *sprawdzanyMorfizm* jest w realcji mono (czy jest kandydatem na bycie monomorfizmem) z pozostałymi morfizmami przekazywanymi do funkcji w *kategoria*, tzn. czy spełnia równanie (3.8) dla tych konkretnych morfizmów
8. **inRetractRelation**
   - argumenty: *sprawdzanyMorfizm* : *Morphism*, *morfizm* : *Morphism*, *kategoria* : *Category*

- wartość zwracana: funkcja sprawdza, czy *sprawdzanyMorfizm* jest w realcji retrakcji (czy jest kandydatem na bycie retrakcją) z *morfizm* w *kategoria*, tzn. czy spełnia równanie (3.9) dla tego morfizmu

9. **inCoretractRelation**
   - argumenty: *sprawdzanyMorfizm* : *Morphism*, *morfizm* : *Morphism*, *kategoria* : *Category*
   - wartość zwracana: funkcja sprawdza, czy *sprawdzanyMorfizm* jest w realcji koretrakcji (czy jest kandydatem na bycie koretrakcją) z *morfizm* w *kategoria*, tzn. czy spełnia równanie (3.10) dla tego morfizmu
10. **inIsoRelation**
    - argumenty: *sprawdzanyMorfizm* : *Morphism*, *morfizm* : *Morphism*, *kategoria* : *Category*
    - wartość zwracana: funkcja sprawdza, czy *sprawdzanyMorfizm* jest w realcji izo (czy jest kandydatem na bycie izomorfizmem) z *morfizm* w *kategoria*, tzn. czy spełnia równanie (3.11) dla tego morfizmu
11. **isEpi**
    - argumenty: *morfizm* : *Morphism*, *kategoria* : *Category*
    - wartość zwracana: funkcja sprawdza, czy *sprawdzanyMorfizm* jest epimorfizmem w *kategoria*
12. **isMono**
    - argumenty: *morfizm* : *Morphism*, *kategoria* : *Category*
    - wartość zwracana: funkcja sprawdza, czy *sprawdzanyMorfizm* jest monomorfizmem w *kategoria*
13. **isRetract**
    - argumenty: *morfizm* : *Morphism*, *kategoria* : *Category*
    - wartość zwracana: funkcja sprawdza, czy *sprawdzanyMorfizm* jest retrakcją w *kategoria*
14. **isCoretract**
    - argumenty: *morfizm* : *Morphism*, *kategoria* : *Category*
    - wartość zwracana: funkcja sprawdza, czy *sprawdzanyMorfizm* jest koretrakcją w *kategoria*
15. **isIso**
    - argumenty: *morfizm* : *Morphism*, *kategoria* : *Category*
    - wartość zwracana: funkcja sprawdza, czy *sprawdzanyMorfizm* jest izomorfizmem w *kategoria*

4.2.3 Moduł *Functor*

Poniżej znajdują się nazwy funkcji z klasy typów *Functorial*, wraz z najważniejszymi informacjami dotyczącymi ich implementacji z modułu *Functor*:

1. **mapObjects**
   - argumenty: *funktor* : *Functor*

- wartość zwracana: mapowanie obiektów dla przekazanego jako argument funktora

2. **mapMorphisms**
   - argumenty: *funktor* : *Functor*
   - wartość zwracana: mapowanie morfizmów dla przekazanego jako argument funktora
3. **isFull**
   - argumenty: *funktor* : *Functor*
   - wartość zwracana: wartość logiczna sprawdzenia, czy *funktor* jest pełny
4. **isFaithful**
   - argumenty: *funktor* : *Functor*
   - wartość zwracana: wartość logiczna sprawdzenia, czy *funktor* jest wierny
5. **isDense**
   - argumenty: *funktor* : *Functor*
   - wartość zwracana: wartość logiczna sprawdzenia, czy *funktor* jest gęsty
6. **yieldEquivalence**
   - argumenty: *funktor* : *Functor*
   - wartość zwracana: wartość logiczna sprawdzenia, czy *funktor* dowodzi równoważności kategorii, pomiędzy którymi mapuje - czy jest jednocześnie gęsty, wierny i pełny

4.2.4 Moduł *CategoryEquivalenceResolver*

Poniżej znajdują się nazwy funkcji z modułu *CategoryEquivalenceResolver*:

1. **eq**
   - argumenty: *resolver* : *EquivalenceResolver*, *kategoriaA* : *Category*, *kategoriaB* : *Category*
   - wartość zwracana: wartość logiczna sprawdzenia, czy przekazane na wejściu kategorie są równoważne
2. **eqFunctor**
   - argumenty: *resolver* : *EquivalenceResolver*, *kategoriaA* : *Category*, *kategoriaB* : *Category*
   - wartość zwracana: funktor, który dowodzi równoważności przekazanych jako argumenty kategorii, gdy te kategorie są równoważne; gdy nie są - wartość *Nothing*

4.2.5 Problem porównywania morfizmów w kategorii - moduł *EquivalenceResolver*

W teorii kategorii dowodzenie równoważności morfizmów jest nietrywialnym zadaniem (równoważność nie ma tu nic wspólnego z zastosowaną w bibiotece operacją porównania morfizmów poprzez ich identyfikatory). Teoria kategorii nie wyklucza

istnienia różnych morfizmów o tych samych dziedzinach i kodziedzinach, np. w kategorii zbiorów i funkcji jako obiekty i morfizmy istnieją morfizmy dodawania i odejmowania o dziedzinie i kodziedzinie w obiekcie zbioru liczb rzeczywistych. W teorii zbiorów porównywanie funkcji ma pewien sens, np. można je zdefiniować następująco:

$$\forall_{f,g:X\to Y} f = g \iff \forall_{x\in X} f(x) = g(x) \tag{4.1}$$

Natomiast w teorii kategorii, w związku z tym, że nie posiada ona definicji przynależności elemenu to obiektu oraz nie rozważa jakie obiekty wchodzą w jego skład, taka definicja byłaby niedopuszczalna. W związku z tym dowodzić równoważność morfizmów należy przekształcając odpowiednio aksjomaty teorii kategorii.

Do automatycznego dowodzenia równoważności morfizmów został zaimplementowany moduł *EquivalenceResolver*. Moduł ten zawiera definicję klasy typów *EquivalenceResolver* zawierającej jedną funkcję sprawdzającą równoważność dwóch morfizmów w zadanej kategorii. Obiekt ten ma możliwość wnioskowania o równoważności (lub jej braku) zadanych morfizmów na podstawie dostarczonych, zapisanych w sformalizowany sposób reguł. Każda reguła powinna mieć następującą sygnaturę:

```
equivalenceRule = Morphism -> Morphism -> Category -> Bool
```

EquivalenceResolver przetwarza kolejne reguły do momentu aż z którejś z nich nie wywinioskuje o równoważności morfizmów.

W ramach tej pracy zostały zaimplementowane cztery tego typu reguły oraz został utworzony domyślny obiekt *EquivalenceResolvera* z tymi regułami. Zaimplementowane reguły:

- reguła wykorzystująca definicję epimorfizmu
- reguła wykorzystująca definicję monomorfizmu
- reguła wykorzystująca definicję retrakcji
- reguła wykorzystująca definicję koretrakcji

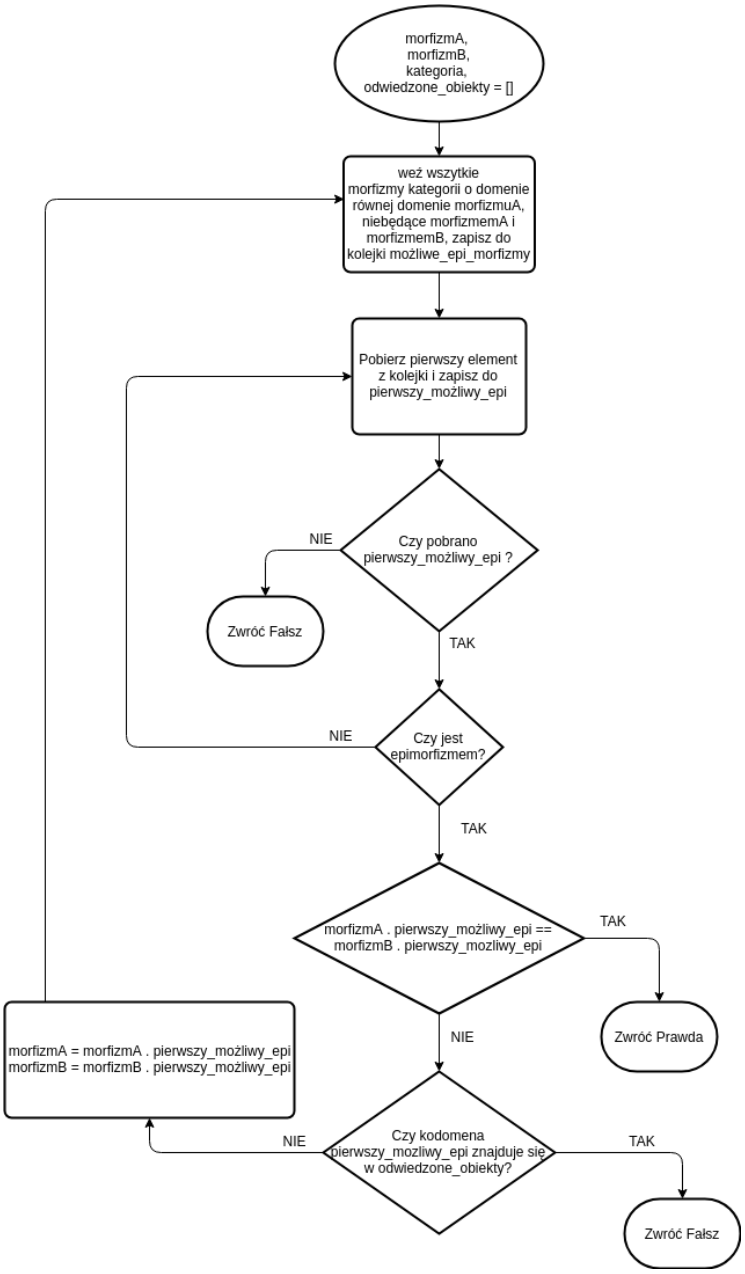
Rysunek 4.3 przestawia schemat blokowy reguły wykorzystujacej definicję epimorfizmu.

Dla reguły wykorzystującej definicję monomorfizmu, jako że jest to pojęcie dualne do epimorfizmu, schemat blokowy wygląda analogicznie; nie zostanie więc tutaj przedstawiony.

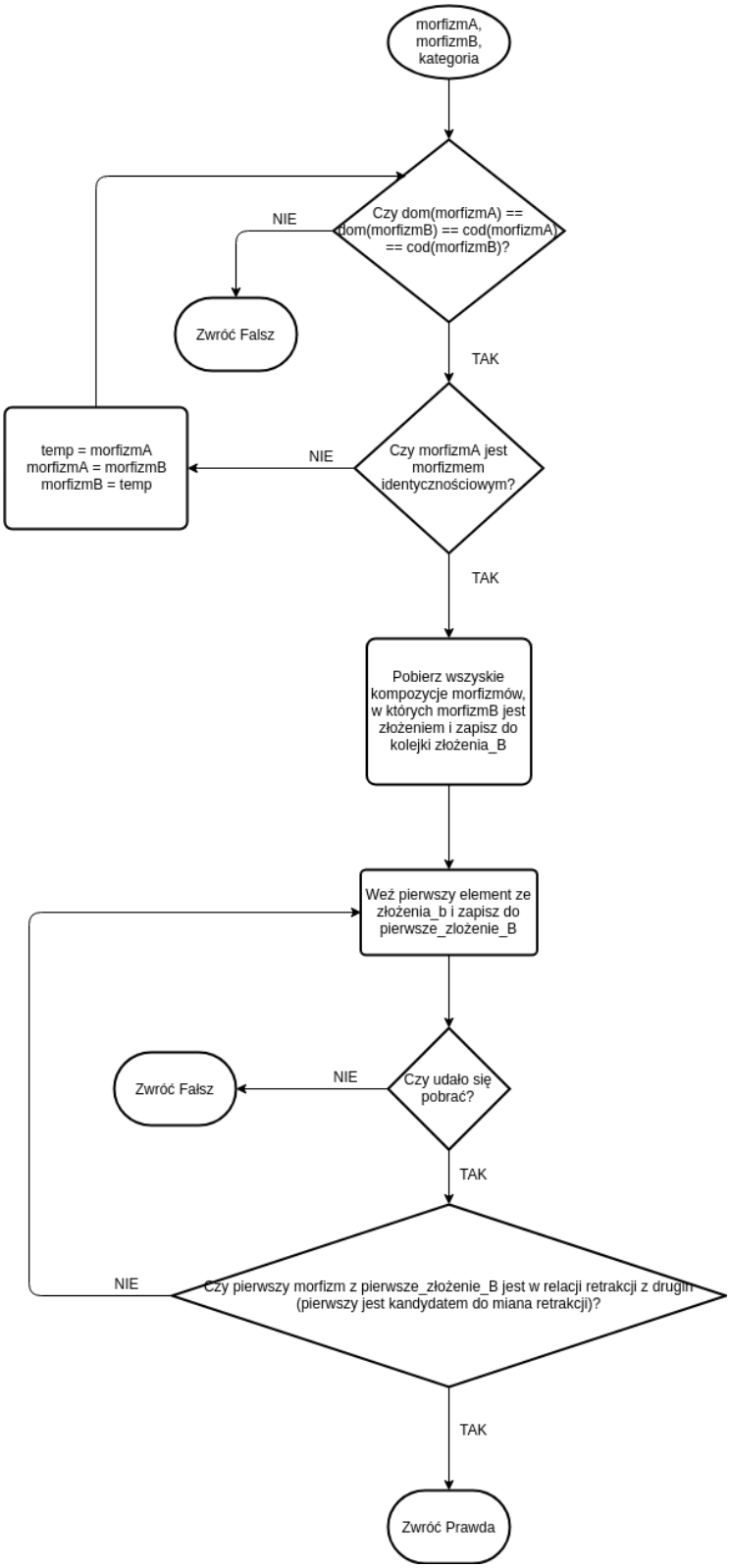
Na rysunku 4.4 umieszczono schemat blokowy reguły wykorzystującej defnicje retrakcji. Z tych samych powodów, dla których nie umieszczono schematu blokowego dla reguły bazującej na definicji monomorfizmu, pominięty zostanie też schemat reguły z definicji koretrakcji.

4.2.6 Problem porównywania kategorii - moduł *CategoryEquivalenceResolver*

W celu porównywania dwóch kategorii pod kątem równoważności zaimplementowano moduł *CategoryEquivalenceResolver*, który udostępnia operacje do tego niezbędne. Porównanie kategorii odbywa się następująco: generowane są wszystkie możliwe mapowania obiektów z zadanych kategorii; potem dla każdego mapowania obiektów generowane są możliwe mapowania morfizmów (brane pod uwagę są tylko te mapowania, które zachowują dziedzinę oraz kodziedzinę morfizmów). Następnie tworzona jest instancja funktora, która jest sprawdzana pod kątem występowania właściwości: pełności, wierności i gęstości. Jeśli któryś ze sprawdzanych funktorów posiada wszystkie te właściwości, to zwracana jest informacja o równoważności kategorii.



Rysunek 4.3: Schemat blokowy działania reguły badającej tożsamość morfizmów, wykorzystującej definicję epimorfizmu (źródło: rys. własnego autorstwa)



Rysunek 4.4: Schemat blokowy działania reguły badającej tożsamość morfizmów, wykorzystującej definicję retrakcji (źródło: rys. własnego autorstwa)

Rozdział 5

Testowanie aplikacji

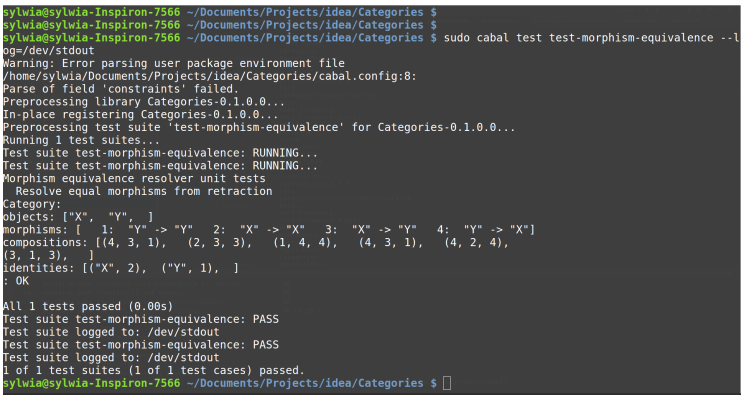
Poprawność wykonania zaimplementowanych operacji została zweryfikowana przez napisanie testów jednostkowych w bibliotece HUnit.

Przygotowano następujace zestawy testów:

1. **test-category-morphism** - zestaw testów dla modułu *Morphism* w bibliotece *Category*; znajdują się w nim m. in. następujące testy:
   - sprawdzenie, czy dwa morfizmy z identycznymi identyfikatorami są identyczne
   - sprawdzenie, czy zwrócono poprawny morfizm odwrotny
   - sprawdzenie, czy utworzono prawidłowy morfizm będący złożeniem dwóch morfizmów, gdy możliwe jest ich złożenie
   - sprawdzenie, czy zwrócono *Nothing*, gdy próba utworzenia morfizmu będącego złożeniem dwóch morfizmów dla których niemozliwe jest określenie złożenia
2. **test-category-utils** - zbiór testów dla modułu z pomocniczymi funkcjami wykorzystywanymi w bibliotece
3. **test-structural-category-morphis** - zestaw testów dla modułu *Morphism* w bibliotece *StructuralCategory*; znajdują się w nim testy takie, jak w odpowiednim module biblioteki *Category*
4. **test-category** - zestaw testów dla modułu *Category* w bibliotece *Category*; znajdują się w nim m. in. następujące testy:
   - sprawdzenie, czy utworzono prawidłową kategorię odwrotną
   - sprawdzenie, czy utworzono prawidłową kategorię produktu dwóch kategorii
   - sprawdzenie, czy utworzono prawidłową kategorię płata nad obiektem
   - sprawdzenie, czy prawidłowo rozpoznano monomorfizm/epimorfizm/retrakcję/koretrakcję
   - sprawdzenie, czy zwrócono Fałsz, gdy następuje sprawdzenie, czy morfizm jest epi wtedy, gdy nie jest

5. **test-structural-category** - zestaw testów dla modułu *Category* w bibliotece *StructuralCategory*; znajdują się w nim testy takie, jak w odpowiednim module biblioteki *Category*
6. **test-morphism-equivalence** - zestaw testów dla modułu *MorphismEquivalenceResolver* w bibliotece *Category*
7. **test-category-functor** - zestaw testów dla modułu *Functor* w bibliotece *Category*; znajdują się w nim m. in. następujące testy:
   - sprawdzenie, czy zwrócono Prawda, gdy funktor gęsty/pełny/wierny jest sprawdzany, czy jest gęsty/pełny/wierny
   - sprawdzenie, czy zwrócono Fałsz, gdy funktor, który nie jest gęsty/pełny/wierny jest sprawdzany, czy jest gęsty/pełny/wierny
   - sprawdzenie, czy Prawda, gdy funktor zachowuje strukturę mapowanej kategorii
8. **test-category-equivalence** - zestaw testów dla modułu *Functor* w bibliotece *CategoryEquivalenceResolver*; znajdują się w nim m. in. następujące testy:
   - sprawdzenie, czy zwrócono Prawda, gdy porównanie dwóch równoważnych kategorii
   - sprawdzenie, czy zwrócono Prawda, gdy porównanie dwóch nierównoważnych kategorii

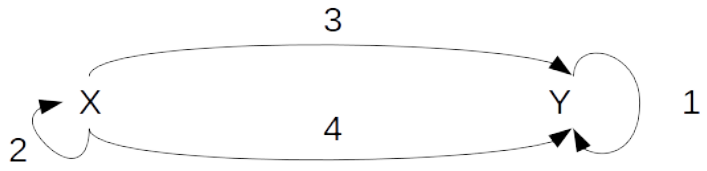
Na zrzucie ekranu 5.1 znajduje się przyklad poprawnie wykonanego testu dowodzenia równoważności morfizmów z defnicji retrakcji i koretrakcji - w testowanej kategorii znajdują się oba te rodzaje morfizmów. Na ekranie widoczny jest log, który wygenrował test - zapis testowanej kategorii. Na rysunku 5.6 znalazła się testowana kategoria. Program dowiódł równoważności morfizmów złożenia morfizmów 4 i 3 z morfizmem identycznościowym 1.



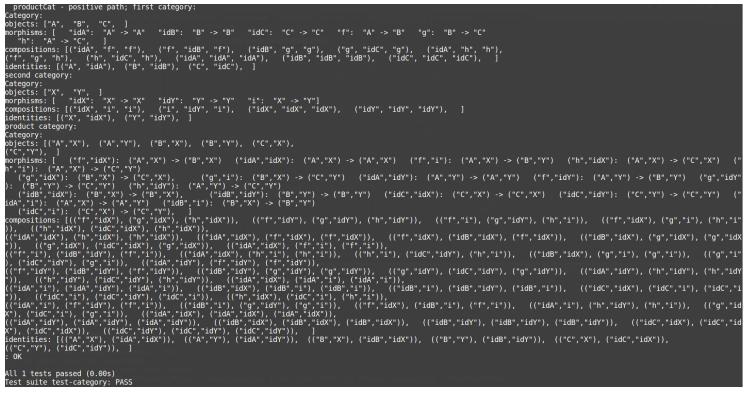
Rysunek 5.1: Poprawnie wykonany test dla modułu *EquivalenceResolver* - dowodznie równoważności z definicji retrakcji

Na zrzucie ekranu 5.2 znajduje się przyklad poprawnie wykonanego testu tworzenia kategorii produktu. Program wypisał na ekran strukturę kategorii wchodzących w skład produktu (*first, second category*) oraz ich produktu.

Przeprowadzone testy pokazują, że zaimplementowany program działa poprawnie.



Rysunek 5.2: Kategoria wykorzystana do przetestowania dowodzenia równoważności morfizmów z definicji retrakcji



Rysunek 5.3: Poprawnie wykonany test tworzenia kategorii produktu

Rozdział 6

Eksperymentowanie z opracowaną aplikacją

Przy użycu zaimplementowanej biblioteki można w łatwy sposób modelować różnego rodzaju struktury topologiczne, takie jak:

- grafy
- sieci komputerowe
- sieci przepływowe
- sieci Petriego itp.

Poniżej przedstawiono przykłady wykorzystania utworzonej biblioteki.

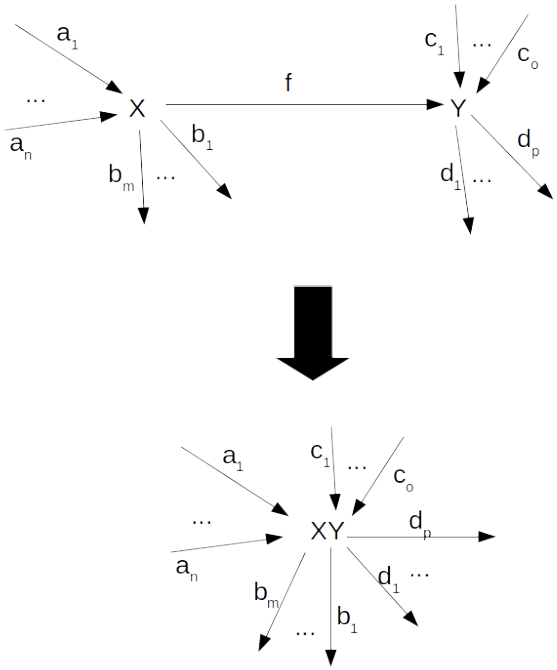
6.1 Zmniejszenie rozmiaru modelowanej struktury

Modelując niektóre struktury przy użyciu zaimplementowanej biblioteki może się okazać, że rozmiar powstałej w wyniku tego modelowania kategorii jest bardzo duży. Stan ten sprawia, że przetwarzanie takiej kategorii staje się uciążliwe - może skutkować dużym zużyciem pamięci lub długim czasem przetwarzania. W tym celu warto jest sprawdzić przed przystąpieniem do właściwego przetwarzania, czy kategoria nie zawiera izomorficznych obiektów. Obiekty izomorficzne X, Y to takie obiekty, które są w następującej relacji:

$$f : X \to Y \tag{6.1}$$

gdzie morfizm f jest izomorfizmem. Obiekty izomorficzne można zastąpić jednym obiektem (tak jak przedstawiono to na rysunku 6.1) i w ten sposób zmniejszyć rozmiar rozważanej kategorii.

Taki zabieg można zastosować przy użyciu funkcji $isIso$, udostępnionej przez bibliotekę.



Rysunek 6.1: Izomorficzne obiekty. (źródło: rys. własnego autorstwa)

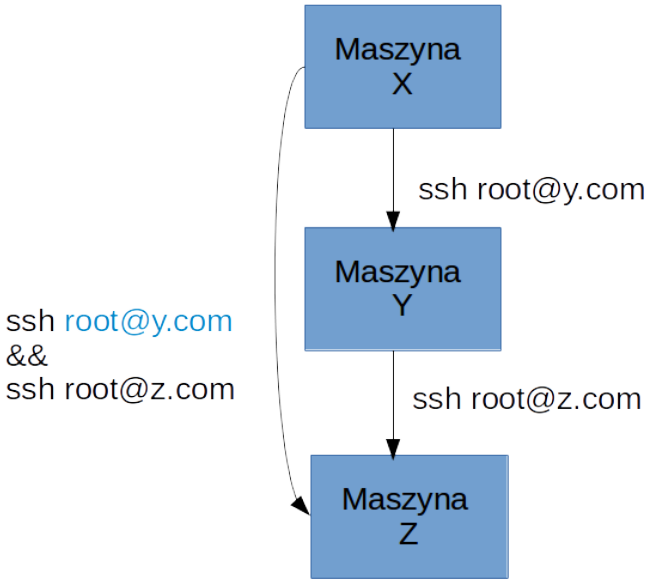
6.2 Modelowanie sieci komputerowej za pomocą kategorii

Sieci komputerowe są przykładem struktury, którą można zamodelować przy użyciu kategorii. W ten sposób można analizować ich topologię bez zagłębiania się w wewnętrzną strukturę jej węzłów (czyli maszyn podłączonych do sieci).

W przedstawionej poniżej kategorii będącej odpowiednikiem sieci komputerowej obiektami są maszyny (komputery), zaś morfizmy reprezentują usługi, za pomocą których można się łączyć z maszynami (dostęp do zasobów drugiej maszyny). Fakt istnienia morfizmu pomiędzy obiektami X i Y odpowiada możliwości połączenia się daną usługą przez maszynę odpowiadającą domenie morfizmu z maszyną odpowiadającą kodomenie morfizmu. Tak rozumiana kategoria sieci komputerowych spełnia warunki przedstawione w podrozdziałach 2.1.1-2.1.3, które każda kategoria spełniać powinna. Poniżej przedstawiono, jak są rozumiane te zagadnienia w tej kategorii i w jaki sposób ta kategoria je spełnia:‘

- relacje identyczności - każda maszyna ma dostęp do wszystkich swoich zasobów; pominięto tutaj przypadek istnienia praw dostępu; można jednak zamodelować również prawa dostępu - należy wtedy wyodrębnić osobne obiekty reprezentujące grupy użytkowników o tych samych prawach dostępu)

- złożenia - złożenie dwóch morfizmów w tym wypadku oznacza sekwencję akcji, jaką można wykonać na kolejnych maszynach, np. połączenie się protokołem SSH z maszyny X do maszyny Y, a następnie tym samym protokołem do maszyny Z może być reprezentowane jako jedna akcja (rysunek 6.2)
- łączność - w tym wypadku nie ma większego znaczenia; oznacza tyle, że nie ma znaczenia w jakiej kolejności składamy akcje (czyli połączenia z maszyną).



Rysunek 6.2: Złożenie w kategorii sieci komputerowych. (źródło: rys. własnego autorstwa)

Tak zamodelowaną sieć można sprawdzić pod kątem istnienia w jej strukturze określonych właściwości (czy sieć ma określoną topologię). Aby to zrobić należy ją porównywać za pomocą funkcji *eq* z modułu *CategoryEquivalenceResolver* z odpowiednią kategorią. Ta odpowiednia kategoria reprezentuje sprawdzaną właściwość. Jest to możliwe, ponieważ tożsamość kategorii odpowiada tożsamości sieci. Tożsamość kategorii jest rozmiana jako fakt istnienia funktora mapującego z jednej sprawdzanej kategorii do drugiej, który posiada pewne własności. Własności te w przypadku kategorii sieci komputerowej oznaczają:

- pełność funktora - jeśli w docelowej kategorii sieci istnieje dany morfizm, czyli możliwość połączenia się daną usługą z maszyny X do Y, to aby kategorie były tożsame, funktor musi uwzględnić w mapowaniu wszystkie te usługi; naturalnym jest, że nie może żadnej pominąć, gdyż wtedy pary obiektów X i jego mapowanie X' oraz Y i jego mapowanie Y' posiadałyby inne zbiory możliwych do skorzystania usług
- wierność funktora - funktor nie może zmapować wielu morfizmów (usług) z tego samego homsetu do jednego morfizmu z docelowej kategorii; w kategorii

sieci morfizmy te reprezentują bowiem zupełnie inne usługi (np. dostępu za pomocą protokołu SSH nie można utożsamiać ze zdalnym pulpitem)

- gęstość funktora - jeśli jakaś maszyna z docelowej kategorii nie brała udziału w mapowaniu (żadna maszyna ze źródłowej kategorii nie została do niej zmapowana), to musi być ona izomorficzna z inną z tej kategorii, dla której takie mapowanie nastąpiło; w przeciwnym przypadku oznaczałoby to, że docelowa kategoria ma jakieś nadmiarowe maszyny, a na tej podstawie można wnioskować o różności porównywanych kategorii sieci komputerowych.

6.2.1 Przykład zamodelowania sieci komputerowej za pomocą kategorii

W przykładzie rozpatrywana jest sieć komputerowa z czterema maszynami, które mogą oferować następujące usługi:

- dostęp za pomocą protokołu SSH - *ssh*
- możliwość wykonania polecenia *ping* - *ping*
- możliwość podłączenia się zdalnym pulpitem - *RD* (*remote desktop*).

W tablicy przedstawiono wyniki złożenia powyższych morfizmów, czyli wynik kolejnego wykonania akcji wchodzących w skład złożenia.

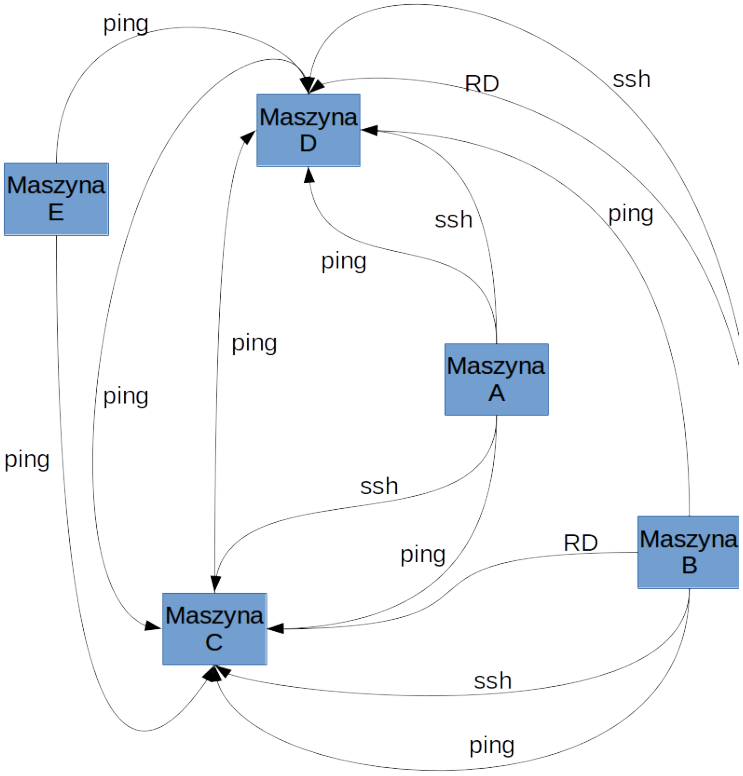
Tabela 6.1: Złożenia operacji odpowiedających morfizmom w kategorii sieci

f	g	g → f
ping	RD	zabronione
ping	ping	ping
ping	ssh	zabronione
ssh	ping	ping
ssh	RD	zabronione
ssh	ssh	ssh
RD	RD	RD
RD	ping	ping
RD	ssh	ssh

Wynik złożenia oznacza akcję wynikową wykonania kolejno obu akcji wchodzących w skład złożenia. Niektóre kombinacje akcji są zabronione, np. nie można po wysłaniu polecenia ping z maszyny X do maszyny Y zalogować się z maszyny X do maszyny Z za pomocą SSH, nawet jesli istnieje możliwość takiego logowania się pomiędzy Y i Z. To obostrzenie jest specyficzne dla modelowanej dziedziny (nie zaś do teorii kategorii w ogóle) i należy je mieć na uwadze modelując sieć. Dodatkowo założono, że złożenie dwóch operacji ping jest operacją ping.

Badana sieć została przedstawiona na rysunku 6.3:

Kategoria, z którą porównana została powyższa sieć została przedstawiona na rysunku 6.4. Kategoria ta reprezentuje strukturę gwiaździstą sieci komputerowej (architektura klient - serwer). Dla uproszczenia nie uwzględniono relacji identyczności. W kategorii tej wyróżniono obiekt reprezentujący serwer - wszystkie inne

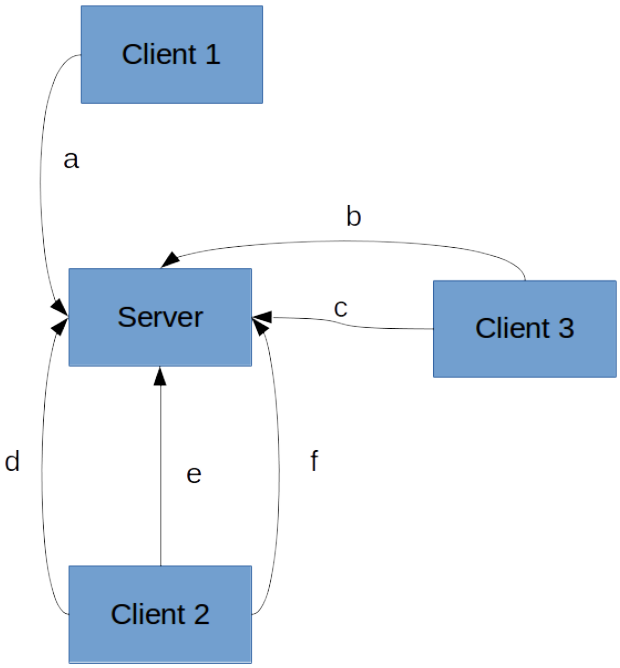


Rysunek 6.3: Badana sieć komputerowa (źródło: rys. własnego autorstwa)

obiekty muszą mieć morfizmy z kodziedziną w obiekcie serwera - komunikować się z serwerem. Klient pierwszy reprezentuje klienty, które komunikują się z serwerem tylko w jeden sposób, klient drugi - na dwa, trzeci - na trzy, jako że w rozpatrywanym przypadku mamy trzy usługi - SSH, zdalny pulpit i *ping*. Klienty nie mogą się ze sobą komunikować.

Poprzez porównanie tych dwóch kategorii można sprawdzić, że badana sieć ma taką strukturę, mimo tego, że na pierwszy rzut oka tego nie widać. Okazuje się jednak, iż faktycznie sieć ta ma taką topologię; maszyny C i D są serwerami (izomorficznymi obiektami kategorii), zaś maszyny B i A to ich klienty. Na zrzucie ekranu 6.5 przedstawiono wynik wykonania testu.

Jak widać z powyższego przykładu, utworzone oprogramowanie pozwala wnioskować o topologii sieci, które można zamodelować kategoriami. Za pomocą narzędzi teorii kategorii można odkrywać nowe informacje na temat tych struktur.



Rysunek 6.4: Kategoria reprezentująca wzorzec architektury klient - serwer (źródło: rys. własnego autorstwa)

```
sylwia@sylwia-Inspiron-7566 ~/Documents/Projects/idea/Categories $ sudo cabal test test-category-equivalence --l
og=/dev/stdout
Warning: Error parsing user package environment file
/home/sylwia/Documents/Projects/idea/Categories/cabal.config:8:
Parse of field 'constraints' failed.
Preprocessing library Categories-0.1.0.0...
In-place registering Categories-0.1.0.0...
Preprocessing test suite 'test-category-equivalence' for Categories-0.1.0.0...
Running 1 test suites...
Test suite test-category-equivalence: RUNNING...
Test suite test-category-equivalence: RUNNING...
Category equivalence resolver unit tests
  eq - positive path (network with star topology): OK (0.25s)

All 1 tests passed (0.25s)
Test suite test-category-equivalence: PASS
Test suite logged to: /dev/stdout
Test suite test-category-equivalence: PASS
Test suite logged to: /dev/stdout
1 of 1 test suites (1 of 1 test cases) passed.
sylwia@sylwia-Inspiron-7566 ~/Documents/Projects/idea/Categories $
```

Rysunek 6.5: Wynik wykonania testu (źródło: rys. własnego autorstwa)

Rozdział 7

Wnioski

Teoria kategorii, dzięki wysokiemu poziomowi abstrakcji, może zapewniać wygodny oraz uniwersalny sposób opisu na pozór różniących się obiektów, np. zbiorów danych, teorii matematycznych. Jest ona wykorzystywana w inżynierii informatycznej - głównie w języku programowania funkcyjnego Haskell, który implementuje jej pojęcia, takie jak m. in.: produkt, koprodukt, monada, funktor. Implementacja ta jednak dotyczy jednej konkretnej kategorii - *Set*, przez co może być efektywnie wykorzystywana głównie w programowaniu. W ramach tej pracy inżynierskiej stworzono biblioteki umożliwiające uniwersalne wykorzystywanie narzędzi teorii kategorii, niezależnej od samej istoty programowania (typów danych, przekształceń na typach). Biblioteka ta może być szczególnie cenna przy analizowaniu struktury topologicznej np. sieci, grafów. Dzięki zaimplementowanym operacjom porównywania kategorii można automatycznie dowodzić fakt spełniania własności topologicznych badanych struktur.

7.1 Zrealizowanie celu pracy

W ramach pracy udało się zaimplementować biblioteki z podstawowymi pojęciami teorii kategorii. Biblioteki te oferują przejrzysty interfejs programistyczny, który pozwala wygodnie rozbudowywać je o nowe moduły. Szczególnie ważną funkcjonalnością jest możliwość porównywania morfizmów, która pozwala minimalizować skalę analizowanych przy użyciu tej biblioteki problemów - gdy zostanie dowiedziona równość dwóch morfizmów, jeden z nich można wyeliminować ze struktury i zastąpić go drugim. Bardzo ważne w omawianych bibliotekach jest to, że ich model jest przystosowany do kolejnych pojęć teorii kategorii. Spośród istniejących rozwiązań, wbudowanych w język Haskell, żadna nie jest tak rozbudowana, zazwyczaj zawierają one kilka definicji pojęć. Opisywana biblioteka oprócz nich zawiera operacje na zaimplementowanych obiektach, np. operacje na kategoriach. Dodatkowo biblioteka *Category* zawiera moduł z operacjami porównywania kategorii. Aplikacja liczy 7328 linii kodu. Oprócz samej implementacji dołączono zbiór testów jednostkowych sprawdzających poprawność opearcji - zarówno pojedynczych, mniejszych funkcji,

jak i większych funkcjonalności, np. porównywania kategorii i morfizmów. Pokrycie kodu testami jest na poziomie 80%-90%. Jest to wysoki poziom pokrycia testami, który pozwala przyjąć, że implementacja jest prawidłowa.

7.2 Propozycja dalszego rozwoju aplikacji

W ramach dalszego rozwoju biblioteki można rozwinąć ją o kolejne pojęcia teorii kategorii - transformację naturalną, monoid, granicę. Inną propozycją rozwoju biblioteki jest dodanie innych i rozwinięcie istniejących sposobów reprezentacj kategorii (inne struktury danych itp.) tak, by użytkownik miał możliwość wybrania najbardziej wygodnego sposobu używania biblioteki. Aplikację można również poprawić pod kątem wydajności. W szczególności należałoby przepisać moduł z porównaniami kategorii tak, by wspierał on wielowątkowość. Z uwagi na funkcyjną naturę języka, w którym biblioteka została napisana, czyli Haskella, implementacja ta nie wiązałaby się z koniecznością synchronizowania wątków, co znacznie by ją uprościło. Zrównolegleniu mógłby ulec proces rekursywnego sprawdzania kolejnych funktorów mapujących porównywane kategorie, gdyż jest to proces niezależny od innych wywołań rekurencyjnych.

Bibliografia

[1] System F. https://crypto.stanford.edu/~blynn/lambda/systemf.html.

[2] Categories. https://plato.stanford.edu/entries/categories/.

[3] Kant's "categories". http://www.thelogician.net/LOGICAL-and-SPIRITUAL-REFLECTIONS/Kant/Kant-Categories-B5.htm.

[4] S. Eilenberg, S. McLane. *General Theory of Natural Equivalences, 1942.*

[5] Ch. Allen, J. Moronuki. *Haskell Programming from First Principles.*

[6] B. Milewski. *Category Theory for Programmers.*

[7] M. Zawadowski. *Elementy Teorii Kategorii, 2018.*

[8] S. McLane. *Categories for the Working Mathematician, 1998.*

Wykaz symboli i skrótów

1. C - symbol kategorii
2. $X \to Y$ - przekształcenie z X do Y
3. id_X - morfizm identycznościowy dla X
4. (X, Y) - produkt X i Y
5. C^op - kategoria odwrotna
6. f^op - morfizm odwrotny
7. C/X - płat kategorii C nad obiektem X
8. $C \times D$ - produkt kategorii C i D

Spis wykorzystanych rysunków

1. (2.1) Przykładowa kategoria, str. 6
2. (2.2) Złożenie morfizmów, str. 7
3. (2.3) Przykładowy funktor, str. 11
4. (2.4) Definicja produktu kartezjańskiego w teorii kategorii, str. 14
5. (3.1) Wywołanie operacji z efektami ubocznymi w języku niefunkcyjnym vs. wywołanie monady w Haskellu, str. 17
6. (4.1), Diagram klas biblioteki *Category* dla kategorii, str. 18
7. (4.2), Diagram klas biblioteki *Category* dla funktora, str. 19
8. (4.3), Schemat blokowy działania reguły badającej tożsamość morfizmów, wykorzystującej definicję epimorfizmu, str. 26
9. (4.4), Schemat blokowy działania reguły badającej tożsamość morfizmów, wykorzystującej definicję retrakcji, str. 27
10. (5.1), Poprawnie wykonany test dla modułu *EquivalenceResolver* - dowodznie równoważności z definicji retrakcji, str. 29
11. (5.2), Kategoria wykorzystana do przetestowania dowodzenia równoważności morfizmów z definicji retrakcji, str. 30
12. (5.3), Poprawnie wykonany test tworzenia kategorii produktu, str. 30
13. (6.1), Izomorficzne obiekty, str. 32
14. (6.2), Złożenie w kategorii sieci komputerowych, str. 33
15. (6.3), Badana sieć komputerowa, str. 35
16. (6.4), Kategoria reprezentująca wzorzec architektury serwer - klient, str. 36
17. (6.5), Wynik wykonania testu, str. 36

Spis tabel

1. (6.1) Złożenia operacji odpowiadających morfizmom w kategorii sieci, str. 34