Implementacja rachunku sekwentów logiki INT z metodami kontroli pętli dla asystenta dowodzenia *Larch*

Jakub Dakowski

7 lutego 2021

1 Wstęp

Pamiętam do dziś moment, gdy na spacerze po zajęciach z przedmiotu „Logika 2" zorientowałem się, że większość pracy wkładanej w naukę metod dowodzenia poświęcana jest na żmudne przepisywanie formuł. Jednocześnie najczęstsze problemy występujące u moich znajomych zdawały się wynikać właśnie z tego – znaczna część osób nie radziła sobie ze skupieniem na strategii dowodzenia, bądź zwyczajnie źle przepisywała poprzednią formułę.

Powstała wtedy w mojej głowie idea logicznego kalkulatora prostego, który wykonywałby za użytkownika najprostsze elementy procesu dowodzenia, umożliwiając mu skupienie się na formie dowodu. Z czasem cały koncept zaczął ewoluować. Kuszące wydawało się bowiem stworzenie narzędzia uniwersalnego – takiego, w którym od skutecznego dowodzenia każdego dzieli tylko napisanie jednego skryptu.

Niedługo później zacząłem projektowanie, implementacje oraz konsultacje ze znajomymi projektantami. Wraz z rozpoczęciem prac nad esejem na przedmiot „Logiki nieklasyczne" zdecydowałem, za namową znajomych, podjąć się próby zaimplementowania logiki, która sprawiła mi najwięcej kłopotów podczas tego kursu.

W ramach tego eseju opisany zostanie proces zaprogramowania dwóch rachunków sekwentów dla logiki intuicjonistycznej wraz z metodami zapobiegania pętlom. Oprócz tego opisana zostanie próba utworzenia solvera wraz z powodami porażki oraz metodami na jego „powstanie z popiołu".

Program, łącznie z przedstawianymi tutaj implementacjami, dostępny jest w repozytorium `https://github.com/PogromcaPapai/Larch/tree/sequent`.

2 Oprogramowanie *Larch* dotychczas

Larch jest rozwijanym przeze mnie w celach dydaktycznych asystentem dowodzenia. Napisany został od podstaw[1] w Pythonie 3. Działa on w ramach paradygmatu *Plugin Oriented Programming* (*POP*) umożliwiając tym samym tworzenie wtyczek zmieniających poszczególne funkcjonalności programu. Dzięki temu, mimo bycia utworzonym jako narzędzie do implementacji tablic analitycznych, możliwe jest operowanie w nim na dowolnym drzewiastym rachunku.

Dotychczas udostępniał on możliwość dowodzenia w tabelach analitycznych KRZ bez sygnowania formuł[2]. Na przełomie listopada i grudnia 2020 rozpoczęte zostały też prace nad interfejsem graficznym stanowiącym przyjaźniejszą alternatywę dla tekstowego odpowiednika (oba przedstawione na zdjęciu 1). Wraz z tym pojawić się miała możliwość graficznego renderowania dowodów.

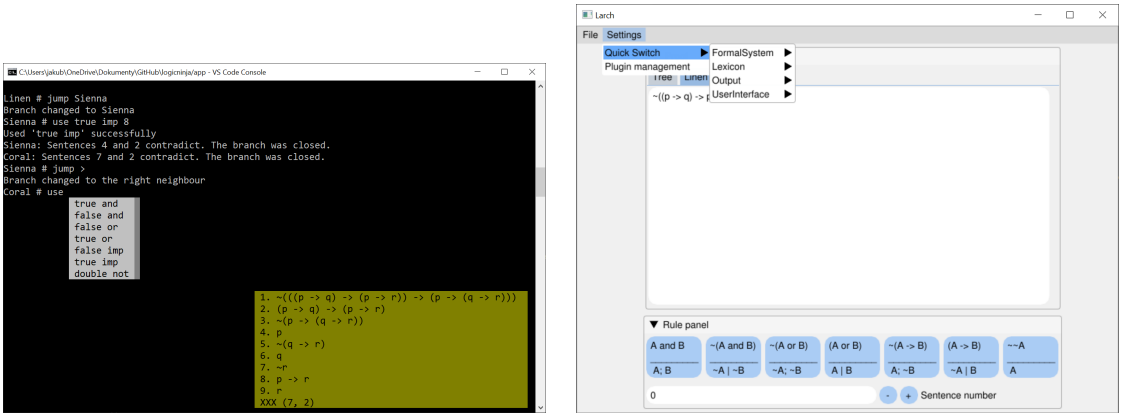
2.1 *Plugin Oriented Programming*

Głównym założeniem stojącym u jego podłoża jest założenie silnej modularności zaczerpnięte ze wspomnainego wcześniej paradygmatu *POP* (Macey, Hatch, 2019). Znaczy to tyle, że większość elementów istotnych do działania programu zaimplementowana jest w formie całkowicie odseparowanych od siebie wtyczek łączonych jednym silnikiem, który operuje na strukturze danych, konwertuje je i zarządza pluginami.

Aby zrealizować to założenie opracowałem klasę `Socket`. Jej obiekty zdolne są do wczytywania i wykonywania kodu w ramach swojej odseparowanej przestrzeni nazw. Po załadowaniu pliku sprawdzane

[1] Poza interfejsem, który utworzyłem z pomocą odpowiednich paczek, wszystko został napisane przeze mnie oraz używa tylko i wyłącznie wbudowanych modułów.

[2] Autor jest ogromnym przeciwnikiem sygnowania prawdziwością formuł w KRZ.



Rysunek 1: Interfejs linii komend oraz aktualny stan tworzonego interfejsu graficznego.

jest, czy występują w nim funkcje wzorcowe. Jest to funkcja o wystandaryzowanym interfejsie - przyjmuje określone argumenty w podanych typach i zwraca wartość opisanego typu. Opisuje się je, wraz z ich interfejsami, w pliku `__template__.py` znajdującym się w specyficznym folderze danego gniazda (w tym folderze znajdują się także dostępne wtyczki). Plik ten przyrównać można do klas abstrakcyjnych występujących w programowaniu obiektowym. Nie może on bowiem być podłączony w miejsce pluginu, ale określa formę innych pluginów. Dzięki temu program zapewnia, że dane podawane i zwracane procedurom zawsze będą takie same. Dla przykładu podać można funkcję `get_type` dostępną w *Lexicon*. Jej zadaniem jest ekstrakcja z tokenu informacji na temat jego typu (koniunkcja, zmienna, alternatywa itp.), a interfejs przedstawiony we wzorze to:

```
def get_type(token: str) -> str:
    pass
```

Informacja taka mówi programowi, że w każdej potencjalnej wtyczce znaleźć się musi funkcja `get_type` za argumenty biorąca ciąg znaków i zwracająca także ciąg znaków. Wtyczka `basic` implementuje ją w poniższy sposób.

```
def get_type(token: str) -> str:
    if token in utils.NON_CONVERTIBLE:
        return token
    else:
        return token.split('_')[0]
```

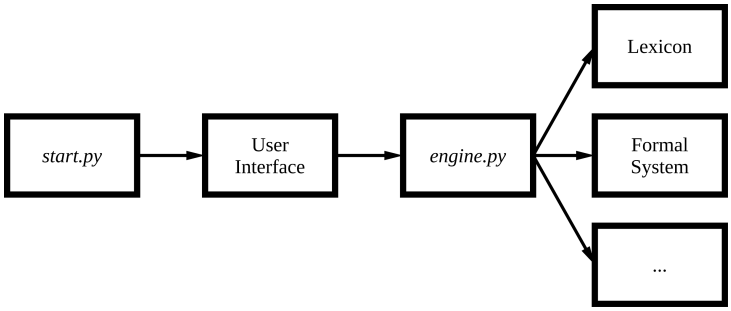
Dla domyślnego formatu tokenu w tym programie (ciąg znaków w postaci `Typ_Znak`) funkcja będzie zwracać sam typ, także będący ciągiem znaków.

Odwołania mają tutaj strukturę gwiazdy, gdzie jeden centralny skrypt uruchamia jasno określone funkcje innych modułów[3]. Architektura ta ma jedną wadę – znaczna część rozwiązań dotyczących interfejsu użytkownika przewiduje operowanie w drugim kierunku – to one obejmują rolę rdzenia wywołującego inne segmenty. W związku z tym wykorzystana została alternatywna struktura zaprezentowana na rysunku 2. Użytkownik rozpoczyna pracę skryptem *start.py*, który posiada jeden socket *UserInterface*. W jego interfejsie przewidziana jest funkcja `run` uruchamiająca interfejs graficzny oraz tworząca i wykorzystująca sesję silnika, który operuje na pozostałych wtyczkach. Możliwość operowania gniazdem *UserInterface* unikając odwołań cyklicznych zapewniona jest specjalną klasą `DummySocket`, która pozwala na wykonywanie tych samych operacji bez ładowania samego pluginu. Kosztem poniesionym przez zastosowanie takiego rozwiązania jest potrzeba zrestartowania programu, co jednak zdaje się uczciwą ceną.

2.2 Gniazda w programie

Jak już zostało powiedziane, w *Larchu* występuje gniazdo odpowiadające za interfejs użytkownika. Wymagana jest od niego tylko jedna funkcja, która nie przyjmuje argumentów, a zwraca kod zamknięcia – `run`. Dodatkowo na początku występowały 3 sockety:

[3]Konwencja nazewnicza Pythona każe nazywać biblioteki modułami.



Rysunek 2: Schemat wywołań w programie *Larch*

Lexicon. Stanowi on implementację słownika oraz parsera. Składają się na niego funkcje:

- `tokenize` przyjmująca zdanie do transkrypcji, listę używanych w danym rachunku typów i listę dodatkowo zdefiniowanych zmiennych (to jednak nigdy nie zostało zaimplementowane w silniku). Funkcja zwraca gotowe zdanie w formie listy tekstów,
- `get_lexem` oraz `get_type` zwracają odpowiednio typ danego tokenu, bądź lexem użyty przez użytkownika,
- `join_to_string` jest funkcją konwertującą zdanie do formy tekstowej, powinna być używana tylko podczas debugowania.

FormalSystem. Jest to zdecydowanie najbardziej rozbudowane gniazdo. Obsługuje ono całość semantyki rachunku – wykrywanie sprzeczności, powtórzeń i wykonywanie operacji na zdaniach. Pojawiają się w nim:

- `prepare_for_proving` dostosowująca dowodzone twierdzenie do wymagań rachunku. W rachunku sekwentów dodawana jest tutaj strzałka sekwentowa i usuwane są zbędne nawiasy,
- `check_contradict` przyjmująca dwa zdania i zwracająca wartość boolowską mówiącą o występowaniu sprzeczności między zdaniami. Jej forma znacznie zmieniła się w trakcie prac nad tym esejem. Jest to powiązane z nowym systemem kodów zamknięć opisanym głębiej w sekcji 4.1,
- `check_rule_reuse` zwracała informacje na temat możliwości ponownego rozkładania zdań. Komponent z nią powiązany okazał się zbyt mało elastyczny, w związku z czym została usunięta, a cała mechanika wcielona w funkcję `use_rule`,
- `use_rule` początkowo na podstawie nazwy reguły i podanego zdania generowała krotki zagnieżdżone zawierające zdania. Aktualnie wymaga ona o wiele większego kontekstu (przyjmuje całą gałąź, zbiór historyczny oraz dodatkowe informacje specyficzne dla reguł) i zwraca dodatkowo „komunikat" na temat modyfikacji zbioru historycznego,
- Dodatkowo występują dwie funkcje zwracające informacje dla użytkownika oraz niezaimplementowane sprawdzanie poprawności zapisu.

Output. Socket ten zajmuje się wyświetlaniem wyniku działania programu w formie wygodnej dla użytkownika. Jak na razie opracowany jest tylko jeden, `text`, wyświetlający informacje w formie prostego tekstu. W planach jest jednak utworzenie specjalnych pluginów przepisujących dowody do formatów TEX-owych (jeden z nich został zrealizowany w ramach prac nad esejem). Składają się na ten plugin dwie funkcje:

- `get_readable` jako argumenty przyjmuje formułę oraz funkcję `get_lexem` z socketu *Lexicon*. Zwraca natomiast zdanie w formie przystępnej dla użytkownika (definicję przystępności dla użytkownika zostawiam autorowi pluginu, przykładowo czysty tekst, lub kod TEX),
- `write_tree` pobiera dowód w specjalnym formacie `PrintedTree`[4], a także, ponownie, funkcję `get_lexem`. Zwraca natomiast listę ciągów znaków stanowiącą gotowe drzewo.

[4]Program nigdy nie dostarcza wtyczkom pełnej struktury danych, gdyż utrudniłoby to weryfikację poprawności działania *Socketów*.

W trakcie prac nad narzędziem dodany został także socket *Auto*, który opisany zostanie w późniejszej części pracy.

2.3 Struktura danych

Całość dowodu przechowywana jest w strukturze drzewa. W ramach oszczędności pamięci każdy węzeł, zamiast przechowywać pojedynczą formułę, stanowi reprezentację ciągu wszystkich zdań do kolejnego rozgałęzienia. Kolejne elementy dodaje się do drzewa z pomocą krotek zagnieżdżonych (takie właśnie generuje funkcja `use_rule`). Najprościej omówić je na przykładzie:

```
(
        # Gałąź 1.:
        (
                ["q"],
                ["p"]
        ),
        # Gałąź 2.:
        (
                ["not", "q"],
                ["not", "p"]
        )
)
```

Dodanie do dowodu takiej struktury spowoduje rozszerzenie jej o dwie gałęzie, z czego pierwsza zawierać będzie p oraz q, a druga $\neg p$ oraz $\neg q$. Struktura danych nie musi być drzewem binarnym, w związku z czym możliwe jest również umieszczenie trzech lub więcej krotek. Ciągły rozwój narzędzia poskutkował zablokowaniem tej funkcjonalności. Każdy węzeł poza listą zdań posiada:

- **Nazwę gałęzi** – podstawowym elementem nawigacji są gałęzie, w związku z czym istotna jest możliwość sprawnego skakania po nich. Za poradą znajomych projektantów UX są to nazwy kolorów, które są też używane jako tło w interfejsie,
- **Zbiór historii** – w ten sposób zapobiega się wielokrotnemu używaniu reguł rozgałęziających, co zaburzyłoby interpretację,
- **Listę liści** – wspólna dla całego drzewa, koresponduje z listą gałęzi pozwalając na poruszanie się po nich,
- **Informację o zamknięciu** danej gałęzi.

3 Podstawy teoretyczne

W ramach tej pracy zaimplementowany zostanie rachunek sekwentów dla logiki intuicjonistycznej wraz z solverem przeprowadzającym znaczną część dowodów. Aby jednak o tym rozważać trzeba zastanowić się, czym właściwie jest rachunek sekwentów oraz logika intuicjonistyczna.

3.1 Rachunek sekwentów

Pierwszą pracą Gentzena jest, opublikowana w 1932 roku, technika eliminacji reguły cięcia z dowodów pisanych w tym rachunku (Rathjen, Sieg, 2020). Moim zdaniem był to jeden z najważniejszych ruchów w ramach automatyzacji dowodzenia – została bowiem w pełni zlikwidowana prawdopodobnie najtrudniejsza do algorytmizacji reguła. W ten sposób Gentzen otrzymuje system dowodowy o strukturze drzewa, który posiada dwie reguły strukturalne i po dwie dla każdego spójnika. Warto jednak zaznaczyć, że oryginalnie reguły dla alternatyw i koniunkcji występują w formie dziedziczącej następnik i poprzednik. Reguły te zaprezentowano na rysunku 3.

Współcześnie spotkać można się jeszcze z tymi regułami chociażby w pracy Howe (1997), ale korzysta się też z reguły opartej na interpretacji sekwentów w postaci „Z koniunkcji poprzedników wynika alternatywa następników". Wtedy reguły mają formę przedstawioną na rysunku 4

Taki format redukuje utratę informacji między krokami, co znacznie ułatwia tworzenie niskopoziomowych solverów. Nie trzeba bowiem implementować zasad wybierających bardziej wartościowe informacje.

Lewa koniunkcja dla A	Lewa koniunkcja dla B	Prawa alternatywa dla A	Prawa alternatywa dla B
$\dfrac{A, \gamma \Rightarrow \lambda}{A \wedge B, \gamma \Rightarrow \lambda}$	$\dfrac{B, \gamma \Rightarrow \lambda}{A \wedge B, \gamma \Rightarrow \lambda}$	$\dfrac{\gamma \Rightarrow \lambda, A}{\gamma \Rightarrow \lambda, A \vee B}$	$\dfrac{\gamma \Rightarrow \lambda, B}{\gamma \Rightarrow \lambda, A \vee B}$

Rysunek 3: Formy reguł koniunkcji i alternatywy w oryginalnym rachunku

$$\frac{A, B, \gamma \Rightarrow \lambda}{A \wedge B, \gamma \Rightarrow \lambda} \quad \frac{\gamma \Rightarrow \lambda, A, B}{\gamma \Rightarrow \lambda, A \vee B}$$

Rysunek 4: Nowszy format reguł dla alternatywy i koniunkcji

Rathjen, Sieg (2020) opisują rachunek z regułami dla negacji. Warto jednak dodać, że można spotkać się z formami ją omijającą. W nich przepisuje się każdą formułę $\neg A$ do $A \to \bot$.

Dowód we wszystkich rachunkach sekwentów kończy się wraz z zamknięciem wszystkich gałęzi z pomocą jednej z dwóch reguł – występowanie tej samej formuły w poprzedniku i następniku oraz stałej falsum w poprzedniku.

3.2 Logika intuicjonistyczna

Jej początki sięgają wczesnych lat XX wieku, kiedy to Luitzen Egbertus Jan Brouwer opracowywał matematykę intuicjonistyczną. Z czasem został z niej wyekstrahowany aspekt logiczny, co poskutkowało powstaniem rachunku IRZ oraz IRP (Moschovakis, 2018). Opiera się ona na intuicjonizmie – poglądzie filozoficznym, według którego można mówić tylko o istnieniu bytów konstruowalnych. Głównym wynikiem takich założeń jest usunięcie prawa podwójnej negacji (pozostaje jednak prawo odwrotne) oraz prawa wyłączonego środka. Zachowywane jest jednak prawo Dunsa Szkota, dzięki czemu zachowana jest eksplozja prawdziwości przy sprzeczności systemu.

Interpretacja spójników w tym rachunku jest istotnie odmienna od klasycznej – nie są one bowiem rozumiane jako pewne funkcje na zbiór $\{0, 1\}$, ale jako określenia bezpośrednich związków między zdaniami. Stąd też implikacja rozumiana jest jako „istnieje transformacja A do B". Negacja w tym rachunku *de facto* nie istnieje, jej sens przenoszony jest jednak przez $A \to \bot$, co rozumiemy jako „istnieje transformacja A do fałszu".

3.3 Rachunek sekwentów dla logiki intuicjonistycznej

W związku z tymi własnościami należy przeprowadzić odpowiednią modyfikację rachunku sekwentów. Nieco ironicznie jest ona znacznie prostsza od modyfikacji w systemie aksjomatycznym. Wystarczy bowiem ograniczyć zbiór następników do bycia singletonem i odpowiednio zmodyfikować pod to reguły. Propozycję dostarcza Herbelin (1995, za: Howe (1997)). Jest nią system *MJ*. Różni się on przede wszystkim pod względem reguły dla lewej implikacji. Ze względu na potrzebę istnienia singletonu w następniku niemożliwe jest też użycie „nowoczesnej" reguły dla prawej alternatywy. Oprócz tych zmian przedstawiony jest też koncept *stoup*. Jest to etykieta nadawana sekwentom (po rozłożeniu całości prawej strony) informująca, która formuła jest aktualnie rozkładana. Warto jednak zauważyć, że ma to przede wszystkim wpływ przy automatyzacji i możliwe jest przedstawienie analogicznego rachunku bez tego elementu. Ma to za zadanie zredukowanie problemu wyboru do minimum. Nadal jednak występują obszary, w których jest on potrzebny i jedna z prezentowanych tutaj implementacji tego zjawiska nie wykorzystuje

Niestety klasyczny rachunek sekwentów dla logiki INT ma wiele wad. Jak donosi Howe (1997) szczególnie problematyczny jest brak terminacji dowodów. Wynika to z możliwości zapętlania się dowodu przez wielokrotne wykorzystanie prawej implikacji w lewej gałęzi, jak zostało to zaprezentowane tutaj:

$$\dfrac{\dfrac{\dfrac{\vdots}{p \to \bot \Rightarrow p} \quad \dfrac{}{\bot \Rightarrow p}}{p \to \bot \Rightarrow p} \quad \dfrac{}{\bot \Rightarrow p}}{p \to \bot \Rightarrow p} \quad \dfrac{}{\bot \Rightarrow p}}{p \to \bot \Rightarrow p}$$

3.4 Zapobieganie pętli

Występuje wiele metod zapobiegania pętlom w rachunkach sekwentów. Pojawia się chociażby koncepcja sygnowania formuł proponowana dla logik multimodalnych w pracy Birštunas (2007). Oprócz tego często wyróżnia się też koncepcję historii (Howe, 1997) – jest to zbiór przechowujący informacje na temat już rozłożonych zdań, który wykorzystywany jest w późniejszym etapie do wykrywania ewentualnych ponownych rozłożeń.

Warto zauważyć, że *Larch* posiadał już pewien mechanizm przypominający historię – jest nim wspomniany w sekcji 2.3 zbiór historii. Dlatego właśnie zdecydowałem skupić się na propozycjach z pracy Howego. Proponuje on w swojej pracy dwa rozwiązania – rachunek szwajcarski oraz szkocki. Reguły obu rachunków mają swoje źródło w propozycji Herbelina (1995, za: Howe (1997)), do każdej dodano jednak dodatkowe informacje o obsługiwaniu negacji (są one wtórne, wyprowadzone z odpowiednich reguł dla implikacji) oraz zachowania historii.

W obu systemach dowodzenie rozpoczyna się od użycia wszystkich reguł działających na następniku, aby następnie wybrać priorytetyzowaną formułę w następniku i skupić się na jej rozkładaniu. W razie trafienia w ślepy zaułek komputer cofa się do ostatniego wyboru i podejmuje jedną z alternatyw (fakt ten będzie potrzebny, aby potem odkryć powód mojej porażki w implementacji solvera). W związku z tym oba rachunki wykorzystują oryginalną notację reguł dla koniunkcji w poprzedniku. Utrata informacji odbywa się bowiem na drodze wyboru, dla którego alternatywa zostanie później rozpatrzona. Moim zdaniem jest to ruch zbędny i równie skutecznie zachowywałby się solver nie redukujący informacji jednocześnie zmniejszając liczbę wyborów, które muszą być podjęte.

Ze względu na rozmiar tego eseju zdecydowałem się nie umieszczać kolejnych stron przedstawiających reguły rachunków. Są one dostępne w pracy Howe (1997).

3.4.1 Rachunek szwajcarski

Jest to prostszy z proponowanych rachunków. Opiera się na stosunkowo prostym systemie przechowywania następników sekwentu przy regule lewej implikacji i resetowania pamięci, gdy reguła prawej implikacji dodaje nową formułę do następnika, bądź jeśli następuje rozgałęzienie w wyniku zastosowania reguły dla alternatywy w poprzedniku.

Według Howe (1997) rachunek ten cechuje się dość powolnym wychwytywaniem pętli – często, aby program ją dostrzegł minąć musi znaczna liczba iteracji (według ich badania czasem nawet 500 razy więcej czasu).

3.4.2 Rachunek szkocki

Problem ten rozwiązuje rachunek szkocki, który wykorzystuje odmienną metodę analizy historii. W poprzednim rachunku, przed zastosowaniem reguły dla lewej implikacji, analizowana była formuła z następnika. Natomiast tutaj badana jest formuła z poprzednika rozkładanej implikacji, a dodatkowo sprawdzany jest następnik implikacji w powtórnie stosowanej regule dla prawej implikacji. Jak piszą sami autorzy – w rachunku szwajcarskim formuła testowana jest, gdy ma opuścić następnik, a w szkockim, gdy się nim staje.

Oprócz tego oba systemy różnią się też pod względem dodawania formuł i resetowania historii. W rachunku szkockim każdy reset zbiór automatycznie wypełniany jest jakąś formułą. Zmiany w historii odbywają się znacznie częściej – w poprzednim rachunku historię rozszerzano dla jednej reguły, a resetowano dla dwóch. Tutaj historia resetowana jest w czterech przypadkach (w tym w rozszerzeniu reguły dla prawej implikacji), natomiast czyszczona jest, jak poprzednio, w dwóch przypadkach.

Co ciekawe, zdaje się, że ten system, w przeciwieństwie do wcześniej przedstawionego, do blokowania pętli używa także priorytetów. Autor wnioskuje o tym po badaniu struktury próby dowodu

$$\neg\neg(p \to q) \to (\neg q \to \neg p)$$

W rachunku szwajcarskim, mimo usunięcia z niego priorytetów, pętla była skutecznie wychwytywana, podczas gdy rachunek szkocki, opierając się na szybszej metodzie detekcji, tego narzędzia dowodowego wymagał.

Jak więc widać, wcześniejsze wykrywanie wiąże się z pewnymi kosztami. Kolejnym z nich jest pamięć. Rachunek ten wymaga znacznie większej pamięci, co może być problematyczne w niektórych sytuacjach.

4 Implementacja

W ramach pracy przygotowane zostały pluginy wprowadzające obie wspomniane rachunki. Przygotowując je potrzebne było zrealizowanie pewnych zmian zarówno w strukturze programu, jak i w proponowanych formach dowodzenia.

Podjęta została także próba zaimplementowania solvera i o ile działa on na prostych przykładach, tak w bardziej rozbudowanych przypadkach sobie nie radzi, o czym więcej napisane będzie w rozdziale 5. Zostaną jednak także opisane modyfikacje wprowadzone w celach skutecznego ujęcia tego obszaru.

4.1 Zmiany w strukturze programu

Larch oryginalnie był zaprojektowany jako narzędzie do tworzenia metod dowodzenia podobnych strukturą do tabel analitycznych, z tego wynika grupa ograniczeń, które przeszkadzały we wprowadzeniu rozpatrywanych systemów:

1. Sztywne informacje o kontekście dla reguł,
2. Zamykanie gałęzi tylko z powodu sprzeczności,
3. Prosta obsługa historii.

Wszystkie z nich zostały rozpatrzone i usunięte w taki sposób, aby zapewniać jak największą wolność dla programistów. Niestety wraz z tym potrzebne było wprowadzenie nowych konceptów, co odbyło się kosztem dostępności kodu.

4.1.1 Zarządzanie kontekstem reguł

Jak już zostało wspomniane w sekcji 2.2, do funkcji `use_rule` wprowadzony został dodatkowy parametr w postaci słownika. Z jego pomocą programowi mogą być przekazywane dodatkowe informacje na temat wykorzystywanych reguł.

Sam proces obsługi kontekstu w regule jest prosty, bardziej rozbudowana jest struktura otaczająca go. Została wprowadzona specjalna funkcja wzorcowa `get_needed_context` nie przyjmująca argumentów, a zwracająca potrzebne argumenty w formie listy nazwanych krotek `ContextDef` (prosty obiekt przypominający słownik, ale posiadający tylko określone klucze). W niej dostarczane są programowi:

- **Klucz**, pod jakim powinna być dostępna zmienna w kontekście,
- **Nazwa** wyświetlana użytkownikowi,
- **Podpowiedź** dostępna dla użytkownika,
- **Typ** zmiennej w natywnej formie Pythona, bądź jednego z określonych ciągów znaków (stanowi to dodatkową informację na temat sposobu podawania zmiennej w interfejsie).

Na podstawie tej definicji silnik przekazuje odpowiednie informacje interfejsowi oraz weryfikuje typ zmiennych zapewniając prawidłowość danych podanych do zastosowania reguły.

4.1.2 Zamykanie gałęzi

W problemie tym wyróżnić można dwa aspekty: sztywnie określony powód do zamknięcia gałęzi oraz niska informatywność dla użytkownika.

Funkcja `check_contradict` we wcześniejszych wersjach programu była przywołana jako swego rodzaju test sprzeczności między zdaniami – przyjmowała za argumenty dwa zdania, a zwracała informacje o ich sprzeczności. Samym uruchamianiem zajmował się natomiast silnik programu. Podjęta została decyzja o przeniesieniu większości ciężaru na ową funkcję, co wiązało się ze zmianą jej formy – aktualnie jej argumentami jest badana gałąź oraz jej historia (dane te nie są jak na razie wykorzystywane, ale uznałem, że bardziej wartościowe jest ich udostępnienie). Jej funkcjonalność została znacznie poszerzona – wcześniej działała jako funkcja zwracająca wartość relacji sprzeczności, a aktualnie decyduje o zamknięciu gałęzi.

W przypadku braku powodu funkcja zwraca stałą `None`. Gdy jednak zostanie wykryty powód do zamknięcia zwraca ona krotkę trzech wartości:

1. **Kod zamknięcia** informujący program o zamknięciu gałęzi,
2. **Skrót tekstowy** wyświetlany w podglądzie gałęzi i dostarczany obiektami `PrintedTree` do wypisania w dowodzie,
3. **Komunikat** informujący użytkownika dlaczego gałąź została zamknięta; W przeciwieństwie do pozostałych, nie jest zapisywany.

Możliwe, że w przyszłych działaniach wartościowe byłoby zostawienie samego kodu zamknięcia i silne związanie go z komunikatami. Warto dodać też, że nie stanowi to całości mechanizmu blokowania operacji. Funkcja `use_rule` ma zdolność wywoływania specjalnych, wychwytywanych na wyższym poziomie, wyjątków. W ten sposób informować może o specyficznych problemach z użyciem reguł. Dostępny jest też prosty mechanizm zwrócenia wartości `None` w przypadkach niespecyficznych. Podczas działania solvera brak wykonalnych operacji na gałęzi także spowoduje jej zamknięcie.

4.1.3 Nowy zbiór historii

Wcześniejsza implementacja historii miała za zadanie zapobiegać ponownemu rozgałęzianiu dowodu ze względu na tą samą formułę. Dlatego w pamięci przechowywana była tylko lista identyfikatorów zdań, a jedyną możliwością jej resetu (uznawaną za poprawną) było usunięcie całego dowodu.

Oba rachunki wymagają w swoich regułach czyszczenia zbioru. Rachunek szkocki dodatkowo wymaga możliwości dodania później nowego zdania. Było to impulsem, który doprowadził do aktualnego systemu zarządzania historią.

Aktualnie, oprócz informacji o rozszerzaniu dowodu, procedura `use_rule` zwraca także formę analogiczną do tej przedstawionej w 2.3. Jest to krotka zawierająca listy operacji, które mają zostać wykonane na każdej z powstałych gałęzi. Możliwe jest rozszerzenie zbioru poprzez dostarczenie zdania, bądź skorzystanie z jednej z predefiniowanych akcji z pomocą określonego kodu (aktualnie istnieje tylko możliwość zresetowania tablicy oraz dokonania niczego[5]). Są one wykonywane w podanej kolejności, w związku z czym ciąg $\langle -1, A\rangle$ spowoduje najpierw usunięcie historii, a potem dodanie do niej formuły A.

Na koniec warto też dodać, że przedstawiane metody dowodzenia wymagają przechowywania pojedynczych formuł z sekwentu, w związku z czym zawartość historii stanowią całe zdania. Jest to pole do przyszłej optymalizacji – możliwe byłoby bowiem kodowanie zdań w mniejszej objętości, chociażby za pomocą doskonałej funkcji haszującej.

4.2 Zmiany w rachunku

4.2.1 Priorytety

Główną zmianą wprowadzoną w rachunkach była próba ujęcia obu z nich w formie unikającej priorytetyzacji. Na bazie dotychczasowych doświadczeń wydaje się, że w rachunku szwajcarskim można mówić o sukcesie tej próby – program bowiem skutecznie wyłapuje powstające pętle. Wierzę potencjalnemu użytkownikowi i nie chciałbym nadmiernie „prowadzić go za rękę" dosłownie podając kolejne formuły do rozłożenia (szczególnie, że czasem może go to zaprowadzić w złym kierunku). Solver został więc zbudowany tak, aby w tym rachunku radzić sobie bez priorytetów. Jak jednak zostanie jeszcze wspomniane, trudno mówić o jego sukcesie.

To samo udało się dokonać dla rachunku szkockiego i ręczne dowodzenie wciąż nie korzysta z priorytetyzacji, choć da się to zmienić przez kilka linii kodu. Niestety jednak rachunek ten nie jest całkowicie odporny na pętle. Jak zostało wspomniane w sekcji 3.4.2, wydaje się, że rachunek ten korzysta częściowo z etykiet sekwentów do działania. Dlatego zaimplementowany został specjalny znacznik mówiący solverowi, że dane zdanie powinno być rozłożone jako pierwsze. Element ten zaimplementowany jest z pomocą dwóch dekoratorów, gdzie jeden używany jest przez reguły wymagające priorytetu, a drugi przez reguły bez takowego. Znacznik jest każdorazowo usuwany i dodawany na nowo, aby zapobiec jego podwojeniu.

4.2.2 Reguły

Główną wprowadzoną różnicą jest powrót do bardziej współczesnej formy reguły dla koniunkcji w poprzedniku. Zostało to dokonane, aby ograniczyć liczbę podejmowanych przez komputer decyzji.

[5] Operacja pusta, nie zmienia stanu historii

Podobnie wyglądała procedura modyfikacji reguły dla alternatywy w następniku – ponieważ jej dwie formy są zaimplementowane jako jedna reguła, dodana została także specjalna trzecia opcja pozwalająca komputerowi na wybranie bardziej obiecującej formuły.

Usunięta została reguła opisywana jako $(C)^\star$. Pełniła ona funkcję nadawania priorytetu. W rachunku szwajcarskim priorytety nie występują, natomiast w rachunku szkockim zmodyfikowano inne reguły tak, aby mogły „ustalać" priorytet. Usunięto także regułę dla negacji, gdyż jest ona regułą wtórną.

Zamknięcie gałęzi odbywa się także zgodnie z zasadami odmiennymi od oryginalnych – możliwe jest występowanie innych formuł w sekwencie, dzięki czemu wyeliminowano masowe usuwanie formuł z pomocą reguły osłabiania (przykładowo $p, q \Rightarrow p$ także zostanie zaakceptowane jako formuła zamykająca mimo niepełnej zgodności z $p \Rightarrow p$). Znacznie upraszcza to korzystanie z narzędzia.

4.2.3 Solver

Główną zmianą wprowadzoną, początkowo nieświadomie, do struktury solvera było usunięcie z niego *backtrackingu* (wycofywania kroków podjętych w dowodzie, aby podjąć inną decyzję). Jest to prawdopodobnie głównym powodem występowania problemów z jego działaniem. Nie rozpatrywano implementacji powtórnych wykonań reguł jako potencjalnych rozwiązań występujących problemów, gdyż architektura *Larcha* nie wspiera „cofania się" w dowodzie.

Algorytm solvera także jest odmienny od prezentowanego przez autorów. Program kompiluje na podstawie występujących w sekwencie zdań listę wszystkich możliwych do podjęcia akcji, które następnie są sortowane pod względem formy reguły. Kolejność powstająca w wyniku sortowania naśladuje tę proponowaną w pracy (Howe, 1997). Ze względu na nową formę nadaną regule lewej implikacji, została ona przeniesiona na początek kolejki. W ramach interpretacji sekwentów opartej na koniunkcjach i alternatywach operacja taka jest dozwolona i uzasadniona.

Hierarchia operacji przedstawiona przez Howego jest bardzo luźna. Tutaj zdecydowałem skorzystać z okazji i wprowadzić kilka heurystyk, na podstawie których reguły porządkowano. Przykładowo reguła prawej alternatywy postawiona jest na samym końcu, gdyż wiąże się z podjęciem decyzji o utracie pewnych informacji. Reguły rozgałęziające były też zawsze umieszczane za regułami nierozgałęziającymi zapobiegając szybkiemu rozrostowi wszerz.

Podjęta została także próba doboru reguł uwzględniając długość formuł sekwentu. Podejście to może okazać się wartościowe jako heurystyka. Klucz będzie musiał być jednak bardziej zaawansowany, niż $-|A|$, gdzie A to pewien ciąg znaków.

O ile sama implementacja nie okazała się udana, tak wzbogacenie potencjalnych przyszłych rozwiązań o przedstawione optymalizacje mogłoby skrócić czas dowodzenia.

W przypadku rachunku szwajcarskiego podjąłem się implementacji solvera nie wykorzystującego priorytetów. Przy *backtrackingu* zdaje się to możliwe, ale jednocześnie nieoptymalne. Jedynym zastosowaniem takich solverów mogłoby być wykorzystanie w pewnej formie podpowiedzi.

4.3 Uwagi o implementacji

Większość istotnych aspektów poruszona została w poprzednich sekcjach, w związku z czym ten podrozdział jest wyłącznie polem na istotne, ale niepasujące w inne miejsca komentarze.

- Aby nie zaburzać opartej na syntaktyce architektury asystenta zdecydowałem się ująć sekwenty w formie zwykłych formuł. W związku z tym do leksykonu dodany został token `sep` dla przecinków oraz `turnstile` oznaczający znak sekwentu.,
- Jestem dumny z formatu funkcji wykonujących reguły. Przed przekazaniem im formuły jest ona rozbijana na stronę lewą i prawą, wewnątrz funkcji są one równolegle przetwarzane, aby potem powstały dwie krotki zagnieżdżone – jedna dla prawej, a druga dla lewej strony. Owe następnie zostają połączone i zwracane są jako sekwent lub rozgałęzione sekwenty,
- W celu zaimplementowania solvera program został rozwinięty o dodatkowe gniazdo *Auto*. Wymaga ono funkcji `solve` przyjmującej jako argumenty gałąź oraz funkcji `use_rule`[6], a zwraca to samo, co wymagana w argumentach funkcja. Oprócz tego jest też funkcja bez argumentów zwracająca krotkę nazw kompatybilnych pluginów `FormalSystem`,

[6] Użyty zostałby tu delegat - pole przechowujące dane na temat metody innego obiektu wykorzystywane w razie potrzeby wywołania go. W języku Python nie istnieje jednak taka konstrukcja. Dlatego niemożliwe jest obsłużenie sytuacji w całkowitej zgodności z paradygmatem programowania obiektowego.

- Aby umożliwić wyraźne odczytywanie powstałych dowodów opracowałem plugin `TeX_infer` generujący kod możliwy do wyrenderowania w dokumentach TEX-owych przy użyciu pakietu *proof*. Opiera się on na prostej rekurencji wypisującej zawartość obiektów `PrintedTree`. Przykład zastosowania paczki, wraz z kodem źródłowym, przedstawiony jest na rysunku 5.

5 Napotkane problemy i potencjalne rozwiązania

W całym eseju pojawiały się uwagi na temat porażki, którą poniosłem w implementacji solvera. Wynikała ona przede wszystkim z braku możliwości zaimplementowania *backtrackingu*. Podczas dalszego rozwoju przeszkodę zapewne stanowić będzie „pojemność" programu. W trakcie rozwoju nie sprawdzano zachowania programu na bardzo długich dowodach, co spowodowało pojawienie się pewnych problemów w trakcie testów solvera.

5.1 *Backtracking*

Algorytm opisany przez (Howe, 1997) opiera się na podejmowaniu decyzji i wycofywaniu się z nich w razie braku zamknięcia gałęzi. Dopiero, gdy wszystkie możliwości zostaną wyczerpane, program decyduje o braku istnienia dowodu.

Niestety *Larch* nie był tworzony z myślą o zostaniu *frameworkiem* do automatyzacji dowodzenia, w związku z czym nigdy nie uwzględniono ani cofania się w dowodzie, ani przechowywania metainformacji na temat dowodu. Muszę przyznać, że sam aktualnie ubolewam nad podjętymi decyzjami – przycisk „cofnij" wydaje się aktualnie bardzo dobrym pomysłem. Gdy opracowywałem architekturę tego narzędzia, moja percepcja dowodzenia obracała się wokół logik klasycznych – tam trudno znaleźć się w sytuacji, gdzie trzeba „wymazywać" jakąś część dowodu, aby mógł on się powieść. Istnieje jednak kilka rozwiązań tego problemu.

5.1.1 „Ja tu tylko sprzątam".

W trakcie prac nad tym projektem niejednokrotnie pojawiała się u mnie myśl, aby przemianować cel *Larcha* z oprogramowania dydaktycznego na silnik operacji logicznych. W ten sposób zanegować można istnienie problemu uznając, że zadaniem tego programu nie jest decydowanie o operacjach, a jedynie obsługiwanie ich. Wiązałoby się to z decyzjami dotyczącymi struktury projektu:

1. Domyślnie działający interfejs użytkownika najlepiej rozwijałby się jako odrębna aplikacja,
2. Socket *Auto* stałby się redundantny,
3. Wartościowa byłaby refaktoryzacja kodu tak, aby możliwe było wygodne korzystanie z niego jako biblioteki,
4. Ograniczyć należałoby nadmiarowe narzędzia.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p \to \bot, ((p \to \bot) \to \bot) \to \bot, p \Rightarrow p} \quad \dfrac{}{\bot, ((p \to \bot) \to \bot) \to \bot, p \Rightarrow \bot}}{p \to \bot, ((p \to \bot) \to \bot) \to \bot, p \Rightarrow \bot}}{((p \to \bot) \to \bot) \to \bot, p \Rightarrow (p \to \bot) \to \bot} \qquad \dfrac{}{\bot, p \Rightarrow \bot}}{\dfrac{\dfrac{p, ((p \to \bot) \to \bot) \to \bot \Rightarrow \bot}{((p \to \bot) \to \bot) \to \bot \Rightarrow p \to \bot}}{\Rightarrow (((p \to \bot) \to \bot) \to \bot) \to (p \to \bot)}}$$

```
\infer{\Rightarrow (((p \rightarrow \bot) \rightarrow \bot) \rightarrow \bot) \rightarrow (p \rightarrow \bot)}
{\infer{((p \rightarrow \bot) \rightarrow \bot) \rightarrow \bot \Rightarrow p \rightarrow \bot}
{\infer{p , ((p \rightarrow \bot) \rightarrow \bot) \rightarrow \bot \Rightarrow \bot}
{\infer{((p \rightarrow \bot) \rightarrow \bot) \rightarrow \bot , p \Rightarrow (p \rightarrow \bot) \rightarrow \bot}
{\infer{p \rightarrow \bot , ((p \rightarrow \bot) \rightarrow \bot) \rightarrow \bot , p \Rightarrow \bot}
{\infer{p \rightarrow \bot , ((p \rightarrow \bot) \rightarrow \bot) \rightarrow \bot , p \Rightarrow p}{}
& \infer{\bot , ((p \rightarrow \bot) \rightarrow \bot) \rightarrow \bot , p \Rightarrow \bot}{}}}
& \infer{\bot , p \Rightarrow \bot}{}}}}
```

Rysunek 5: Wyrenderowany dowód wraz z jego kodem wygenerowanym z pomocą pluginu `TeX_infer`

W tej sytuacji całość solvera byłaby zaimplementowana na poziomie aktualnego interfejsu użytkownika. Można też stąd wnioskować o istnieniu pewnej słabej formy tego rozwiązania – socket *Auto* jako funkcjonalność interfejsu. Jak widać na rysunku 1 interfejs użytkownika jest *de facto* wyżej w hierarchii od silnika, w związku z czym możliwe jest gromadzenie tam pewnych metainformacji z prymitywną formą backtrackingu w postaci resetowania dowodu i wykonywania wszystkich operacji ponownie. Jest to rozwiązanie najtańsze, ale jednocześnie nieoptymalne i naruszające regułę separacji logiki od prezentacji[7].

5.1.2 Dodanie operacji cofania

O wiele bardziej kosztownym rozwiązaniem jest modyfikacja architektury programu w taki sposób, aby wspierał on wycofywanie akcji. Implementację tego należałoby rozpocząć od modyfikacji struktury danych tak, aby każdy węzeł odpowiadał wynikowi operacji na danej gałęzi. Dotychczas za niegroźne dla programu uważano istnienie w węzłach niebędących liściami błędnych informacji (dopóki nie dotyczyły samego zapisu formuły). *Backtracking* wymusiłby jednak analizę kodu w kierunku takich błędów. Należałoby też rozważyć stworzenie dodatkowej struktury danych przechowującej informacje o dokonanych i jeszcze dostępnych próbach. Możliwe jest, że rozwiązanie tego problemu już zostało przeze mnie utworzone i wystarczy teraz je zaadaptować do nowych warunków – w poprzednie wakacje udało mi się ukończyć program *Teresa*. Jest to solver Sudoku opierający się na tworzeniu kolejnych założeń na temat planszy, sprawdzaniu ich konsekwencji oraz wycofywaniu, jeśli dojdzie do sprzeczności (pole skazane na bycie pustym). Powinien więc istnieć tam gotowy sposób *backtrackingu*, który po przepisaniu mógłby stanowić rdzeń nowego solvera.

Prawdopodobnie wiązałaby się z tym znaczna rozbudowa gniazda *Auto*. Jego aktualna forma jest dość prymitywnym podejściem, w związku z czym socket nie radziłby sobie efektywnie w bardziej rozbudowanych przypadkach. Pojawił się swego czasu pomysł wprowadzenia pamięci poszczególnych socketów, której czyszczenie byłoby dokonywane z pomocą wspólnej dla wszystkich funkcji. Rozwiązanie to byłoby bardzo ryzykowne i wymagałoby udzielenia kredytu zaufania ewentualnym autorom w kwestii czyszczenia pamięci w ich modułach.

5.2 Ograniczenia objętościowe

Struktura *Larcha* tworzona była z myślą o przeprowadzaniu niewielkich dowodów. O ile dążono do optymalizacji większości aspektów, tak niektóre elementy nie radzą sobie z dłuższymi dowodami.

Niestety w module *prompt-toolkit* obsługującym interfejs tekstowy istnieje ograniczenie co do objętości wypisywanego tekstu, w związku z czym zdarzały się sytuacje, gdzie program przestawał informować o dalszych krokach, bądź nie wyświetlał całości wygenerowanego dowodu.

Podobny problem stanowią nazwy gałęzi. Początkowo zaplanowane było, aby maksymalną ich liczbą było około 40. Decyzja została podjęta na podstawie długości listy kolorów wykorzystywanej jako źródło nazw. W większych dowodach jest to zbyt mało, dlatego poszerzono program o możliwość nadawania nazw złożonych z losowych liczb i poszerzono maksymalną liczbę gałęzi do 1000 (zabezpieczenie przed ewentualnym wpadnięciem w pętle).

6 Podsumowanie

Esej ten można przyrównać do tragedii greckiej – wyruszyłem w podróż z nadzieją na zaimplementowanie solvera, ale w ostatnim akcie okazało się, że sam siebie skazał na niemożność ukończenia tego zadania. Nie można jednak traktować tego jako porażki. Oryginalny temat został zrealizowany, aktualnie w ramach oprogramowania *Larch* dostępne są dwa rachunki sekwentów dla logiki intuicjonistycznej z mechanizmem zapobiegania pętli. Może potrzebują one pewnego uporządkowania, ale jest to zadanie trywialne.

Zostały także przedstawione możliwości rozwiązania problemu, który powstał w związku z solverem. Zmuszają one mnie, jako autora oprogramowania, do rozważenia przyszłości projektu *Larch*. Doprowadzony został on bowiem na rozdroże, gdzie podjęta musi być decyzja, czy bardziej wartościowy jest silnik, w którym można zaimplementować aplikację, czy aplikacja. Wszystko to zgrało się z ogromnym spadkiem zainteresowania moimi poczynaniami w tym obszarze. Jak na razie zdecydowałem się zawiesić prace nad całością, aby nie tylko odpocząć od mozolnego dopisywania kodu do ponad 7500 już istniejących linii, ale też zdecydować, jaka będzie jego przyszłość.

[7] Wyraźnie oddzielne elementy powinny być odpowiedzialne za operowanie logiką wewnątrz programu i przedstawianie interfejsu.

Literatura

Birštunas, A. (2007), 'Efficient loop-check for multimodal kd45n logic', *Lithuanian Mathematical Journal* **46**(1), 55–66.

Howe, J. M. (1997), Two loop detection mechanisms: a comparison, *w:* 'International Conference on Automated Reasoning with Analytic Tableaux and Related Methods', Springer, s. 188–200.

Macey, T., Hatch, T. (2019), 'Making complex software fun and flexible with plugin oriented programming', Podcast.__init__.
URL: *https://www.pythonpodcast.com/plugin-oriented-programming-episode-240/*

Moschovakis, J. (2018), Intuitionistic Logic, *w:* E. N. Zalta, (red.), 'The Stanford Encyclopedia of Philosophy', winter 2018 edn, Metaphysics Research Lab, Stanford University.

Rathjen, M., Sieg, W. (2020), Proof Theory, *w:* E. N. Zalta, (red.), 'The Stanford Encyclopedia of Philosophy', fall 2020 edn, Metaphysics Research Lab, Stanford University.