MP19 @ II UWr 2 kwietnia 2019 r.

Lista zagadnień nr 6

Przed zajęciami

Zakończyliśmy pierwszą, wstępną część wykładu. Należy zatem przeczytać ze zrozumieniem resztę **Rozdziału 2** podręcznika, potrafić **definiować** skomplikowane **drzewiaste struktury danych**, formułować związane z ich definicjami **zasady indukcji**, i **dowodzić własności programów** korzystających z takich struktur. Należy również rozumieć pojęcie **nieużytków** i ich wpływ na działanie i złożoność programu, a także zaznajomić się z pojęciem reprezentacji wielorakich i programowania z ich użyciem.

Uwaga: Ze względu na późny termin ogłoszenia listy zagadnień, kartkówka w tym tygodniu nie odbędzie się.

Na zajęciach

Ćwiczenie 1.

Zapisz poniższe wyrażenia tak, aby cytowanie pojawiało się wyłącznie przed identyfikatorami (tj., nie cytuj nic poza symbolami):

```
'(+ 4 3 (* 5 8))

'(+ 4 3 '(* 5 8))

`(+ 4 3 (5 . 8))

`(+ 4 3 ,(* 5 8))
```

Ćwiczenie 2.

Zaimplementuj procedurę `paths`, które zwróci listę ścieżek w drzewie binarnym z wykładu. Ścieżka niech będzie listą etykiet w kolejności od korzenia do liścia. Niech liść kończący ścieżkę też występuje w odpowiedzi i oznaczony będzie symbolem `'*`. Na przykład zapytanie

```
(paths (node 3
   (node 2
      (leaf)
      (node 6
         (node 7 (leaf) (leaf))
         (leaf)))
   (node 7 (leaf) (leaf))))
```

powinno dać odpowiedź

```
'((3 2 *) (3 2 6 7 *) (3 2 6 7 *) (3 2 6 *) (3 7 *) (3 7 *))
```

Ćwiczenie 3.

Zaimplementuj procedurę, która zdradza, czy w danym wyrażeniu arytmetycznym jest więcej mnożeń czy dodawań.

Ćwiczenie 4.

Rozważmy prostą składnię opisu dokumentów tekstowych: tekst będzie składał się z tytułu, autora i ciągu rozdziałów; każdy rozdział zaś składa się z tytułu i ciągu elementów z których każdy może być akapitem tekstu lub podrozdziałem. Podrozdziały mają taką samą strukturę jak rozdziały, zaś akapit zawiera (niepusty) ciąg napisów.

Użyj *struktur* wprowadzonych na wykładzie i rekurencyjnych predykatów żeby zdefiniować typ danych reprezentujących teksty, przyjmując że tytuł i autor dokumentu, a także tytuły rozdziałów są napisami (rozpoznawanymi predykatem `string?`).

Ćwiczenie 5.

Na podstawie zdefiniowanego powyżej typu dokumentów możemy łatwo wygenerować stronę w HTMLu.[1] Zdefiniuj procedurę `text->html` zwracającą dla zadanego tekstu stronę HTML (jako napis). Będziesz potrzebować następujących tagów: `html`, `body`, `h1`–`h6`, `p` i być może `div`. Wygeneruj prostą stronę internetową i wyświetl ją w przeglądarce. Uwaga: zadanie nie ma *jednego*, najbardziej poprawnego rozwiązania — trzeba podjąć pewne decyzje projektowe.

[1] A przynajmniej prostą, brzydką aproksymację strony internetowej à la wczesne lata 90.

Ćwiczenie 6.

Oczywiście, statyczne strony internetowe są raczej nudne — szczęśliwie mamy pod ręką język programowania który pozwala na tworzenie procedur i generowanie danych dynamicznie. Używając form `quasiquote` i `unquote` zdefiniuj procedurę która uruchomiona z parametrem `n` generuje dokument opisujący rozkład liczby `n` na czynniki pierwsze.[2]

Ćwiczenie 7.

Do naszej reprezentacji dokumentów dodajemy listy które mogą pojawiać się w rozdziałach/podrozdziałach, i które zawierają pewną liczbę elementów z których każdy zawiera listę akapitów tekstu i/lub podlist (o analogicznych wymogach strukturalnych). Rozszerz reprezentację tekstu o listy i procedurę `text->html` (będziesz potrzebować tagów `ul` i `li`).

Zadanie domowe (na pracownię)

W tym tygodniu rozszerzymy rozwiązanie zeszłotygodniowego zadania rezolucji o dwie istotne z praktycznego punktu widzenia optymalizacje: subsumpcję i specjalne traktowanie klauzul składających się z jednego literału.

W tym celu zauważmy że niektóre formuły możemy uznać za „trudniejsze" od innych: będziemy mówić że φ_1 jest trudniejsza niż φ_2 (co zapiszemy jako $\varphi_1 \sqsubseteq \varphi_2$) jeśli każde wartościowanie spełniające φ_1 spełnia również φ_2. Oczywiście w tym sensie formuły sprzeczne (w tym fałsz, $\bot$) są najtrudniejsze ze wszystkich, a tautologie — najłatwiejsze. Zauważmy teraz, że jeśli mówimy o klauzulach — czyli dysjunkcjach skończonej liczby literałów, łatwo możemy stwierdzić czy jedna z nich jest trudniejsza od drugiej: mamy $c_1 \sqsubseteq c_2 \iff c_1 \subseteq c_2$, gdzie zawieranie interpretujemy zgodnie z intuicją: mamy $c_1 \subseteq c_2$ jeśli c_2 zawiera wszystkie literały z których składa się c_1 (i być może inne).

Ostatnią obserwacją której potrzebujemy jest to, że możemy użyć naszego pojęcia trudniejszych klauzul, żeby wyrzucić ze zbioru budowanego w procesie poszukiwania dowodu rezolucyjnego dowodu sprzeczności wszystkie klauzule „łatwe", tj. takie od których są w zbiorze klauzule trudniejsze (zwróćcie uwagę że nasza implementacja już wykonuje część tej optymalizacji nie dodając do zbioru klauzul trywialnych). Formalnie, możemy zmodyfikować nasz operator

[2]Nie potrzeba dużo więcej żeby użyć Racketa do generowania prawdziwych stron internetowych.

F tak aby zachowywał w generowanym zbiorze tylko te klauzule, które nie są łatwiejsze od innych klauzul ze zbioru i znaleźć punkt stały tego operatora.

Drugą obserwacją którą możemy wykonać jest to, że klauzule składające się z jednego literału są specjalne: jeśli mamy zbiór $X \cup \{l\}$, gdzie l jest literałem, który jest zamknięty na trudność (czyli taki że dla każdego $c \in X$ mamy $l \not\sqsubseteq c$), to bardzo łatwo możemy wykonać rezolucję względem klauzuli l. Po pierwsze, zmienna z l nie może pojawiać się w X z tym samym znakiem co w l (taka klauzula byłaby łatwiejsza od l). Po drugie, jeśli mamy w X klauzulę w której pojawia się z odwrotną polarnością (zapiszmy taką klauzulę jako $c \vee \bar{l}$), to rezolwentą tej klauzuli względem l będzie c — klauzula z *usuniętym* wystąpieniem zmiennej która występuje w l z przeciwnym niż w l znakiem. Co więcej, otrzymana klauzula c jest trudniejsza niż $c \vee \bar{l}$, czyli nie musimy zapamiętywać starej klauzuli! To znaczy, że możemy wykonać rezolucję bardzo „tanio”: otrzymany zbiór będzie ściśle mniejszy niż ten od którego zaczynaliśmy, a mimo to nie stracimy poprawności.

Ćwiczenie 8.

Dwie powyższe optymalizacje potrafią poważnie przyspieszyć działanie naszego algorytmu. Zaimplementuj je! W tym celu będziesz musiał(a):

- Zdefiniować dwuargumentową procedurę `subsumes?` sprawdzającą czy pierwsza z klauzul będących argumentami jest trudniejsza niż druga;
- Przeczytać *ze zrozumieniem* implementację zbiorów klauzul do przetworzenia, w szczególności procedury `clause-set-add`
- Zmodyfikować powyższą procedurę tak aby nigdy nie dodawać klauzuli która jest łatwiejsza niż klauzula już zawarta w zbiorze i jednocześnie by wyrzucać ze zbioru wszystkie klauzule łatwiejsze od właśnie dodawanej (to kończy pierwszą z optymalizacji)
- Dodać specjalne traktowanie klauzul jednoelementowych: to najłatwiej zrobić dodając do interfejsu procedurę `clause-set-map`, analogiczną do procedury `map` dla list, która pozwoli usunąć niechciany literał ze wszystkich klauzul w zbiorze.