Gdzie w programach sens, gdzie logika
Zadania kwalifikacyjne

Michał Horodecki

WWW22, 2026

1 Wstęp

1. Rozwiązania proszę wysyłać przez stronę warsztatów
2. Nie trzeba robić wszystkich zadań – zrobienie około połowy jest jak najbardziej okej, jak również nadesłanie częściowych rozwiązań a nawet zalążków i pomysłów.
3. Zadania staram się oceniać na bieżąco i jak najbardziej można dosyłać poprawki po ocenie aby poprawić wynik (i się czegoś nauczyć!).
4. Forma rozwiązań dowolna, byle była czytelna. Osobiście polecam narzędzia takie jak Latex i Typst.
5. Odradzam używanie AI do rozwiązywania zadań bo mija się to z celem warsztatów. Natomiast jeśli jakieś polecenie lub definicja nie jest do końca jasna to AI może być dobrym punktem wyjścia w celu jej zrozumienia (choć zachęcam do kontaktu ze mną jeśli coś jest niejasne, ja też popełniam błędy).
6. W razie jakichkolwiek pytań i wątpliwości śmiało piszcie na `michalhorodecki2002+www@gmail.com`.

2 Napisz coś o sobie [5p]

W paru zdaniach powiedz co Cię interesuje, co zaciekawiło Cię w tych warsztatach, i co chciałxbyś na nich zobaczyć.

3 Programowanie

3.1 Po co nam typy [3p]

W kilku zdaniach opisz:
- po co Twoim zdaniem są typy danych w programowaniu
- jakie znasz typy i mechanizmy/podejścia które na nich operują
- przykład programu który nie działa(łby) poprawnie bez typów ale błąd jest wykryty po ich dodaniu

3.2 Ograniczenia typów [2p]

Czasem typy ograniczają nas trochę za bardzo.
Podaj przykład prostego programu (funkcji) i jego dwóch implementacji (być może w dwóch różnych językach), takich że:
- Obie wersje semantycznie robią to samo
- Jedna wersja uruchamia się, działa, i zwraca oczekiwany wynik
- Druga wersja nie kompiluje się na skutek niekompatybilnych typów (ale bez typów program by działał poprawnie)

Hint: Można jedną implementację napisać w języku takim jak Python albo JavaScript, a drugą w języku takim jak Java albo C++.

3.3 Czego wymagamy od systemu typów? [4p]

Idealny system typów powinien spełniać dwie dość naturalne własności:
- Nie akceptuje żadnego niepoprawnego programu
- Akceptuje każdy poprawny program

Odpowiedz na dwa pytania:
1. Od czego zależy czy da się zrobić idealny system typów? Intuicyjnie kiedy jest łatwiej taki zrobić?
2. Jeśli nie da się zrobić idealnie, to w którą stronę jest prościej (a wręcz zawsze się da, a nawet trzeba)?

3.4 Logika intuicjonistyczna na typach [6p]

Typ funkcji jednoargumentowej zawsze jest w postaci `A -> B`, gdzie `A` jest typem argumentu a `B` typem wyniku. Istnienie funkcji `f` takiego typu można traktować jako implikację $A \Rightarrow B$ – jeśli `A` jest prawdziwe (mamy wartość `x` typu `A`) to `B` jest prawdziwe (mamy wartość `f(x)` typu `B`).

Przykładowo, funkcja `f(x) = x` jest dowodem na to, że dla **dowolnego** `A` jest spełniona trywialna implikacja $A \Rightarrow A$.
Warto jeszcze zaznaczyć tę dowolność po `A` – funkcje `f(x: int) = x + 1` albo `f(x: str) = x.reverse()` również są typu `A -> A` ale dla konkretnych `A` będących `int` lub `str`.

3.4.1 [2p]

Zapisz, w dowolnym języku funkcje, które dla dowolnych `A, B, C` koncepcyjnie mają typy:
1. `A -> B -> A`
2. `(A -> B -> C) -> (A -> B) -> A -> C`

Najłatwiej może być w czymś typu Python lub JS, gdzie nie ma statycznych typów.

3.4.2 [2p]

Mając dane dwie funkcje, dla dowolnych `A, B, C`:
- `f: A -> B -> A`
- `g: (A -> B -> C) -> (A -> B) -> A -> C`

napisz funkcję `h: A -> A`
(Hint: trzeba skorzystać z dowolności `A,B,C` i użyć `f` dwa razy z różnymi `A, B`)

3.4.3 [2p]

Czy Twoim zdaniem da się zrobić funkcję `(A -> B) -> A` (uwaga! to co innego niż `A -> B -> A`) – uzasadnij tok rozumowania.

4 Rachunek lambda

4.1 Wstęp

Ponieważ języków programowania jest dużo, mają skomplikowaną składnię, i robią też dużo technicznych rzeczy, to w literaturze modeluje je się czymś prostym – najczęściej jakimś wariantem rachunku lambda.

Na nasze potrzeby o rachunku lambda będziemy myśleć po prostu jako o bardzo ograniczonym języku programowania.
W podstawowej wersji mamy trzy rodzaje wyrażeń:

- zmienne – $x, y, z, ...$
- definicje anonimowych funkcji – $\lambda x.E$ – Przy czym:
  - x jest zmienną, E jest dowolnym wyrażeniem
  - Interpretujemy to jako funkcję $x \Rightarrow E$, która bierze x a zwraca E (w którym być może x występuje, ale nie musi)
  - Każda funkcja przyjmuje dokładnie jeden argument
- wywołanie funkcji – $F(E)$ – na przykład $(\lambda x.x)(y)$
  - F, E mogą być dowolnymi wyrażeniami, niekoniecznie funkcjami – w szczególności $x(y)$ też jest ok.

W razie czego odsyłam do Wikipedii: https://pl.wikipedia.org/wiki/Rachunek_lambda lub do mnie osobiście.

Sam koncept nie jest też bardzo oderwany od programowania – można się z nim spotkać w językach takich jak:

- JavaScript – `(x => x + 1)(123)`
- Python – `(lambda x: x + 1)(123)`
- C++ – `[](auto x){return x + 1;}(123)`

Gdyby w rachunku lambda było dodawanie (a będziemy takie rozważać) to zapisalibyśmy wszystkie trzy jako $(\lambda x.x + 1)(123)$

4.2 Funkcje wieloargumentowe (kuryfikacja) [4p]

Powiedzieliśmy że każda funkcja ma być jednoargumentowa – uzasadnij że to nie zmienia siły wyrazu rachunku lambda. Innymi słowy, pokaż jak przetłumaczyć wyrażenia postaci `(x, y, z) => ...` oraz `((x, y) => ...)(123, 456)` w taki sposób aby wynik nadal był taki sam.

4.3 Czego brakuje? [4p]

Wymień dwie konstrukcje (inne niż operatory takie jak dodawanie/odejmowanie) których brakuje w podstawowej wersji rachunku lambda, a które są przydatne w programowaniu.
Zaproponuj jak wyglądałyby w składni (może być na przykładach).

4.4 Wykonanie [4p]

Intuicja za wykonaniem jest prosta – ilekroć widzimy gdzieś wywołanie funkcji to możemy je zastąpić przez jej wynik, czyli podstawienie wartości za wszystkie wystąpienia danego argumentu.
Przykładowo krok obliczenia $(\lambda x.\lambda y.x + y + x)(123)$ prowadzi nas do $\lambda y.123 + y + 123$.
Oczywiście w dużym wyrażeniu takich miejsc może być wiele i wtedy trzeba jakoś wybrać gdzie wykonujemy krok.

4.4.1 Nieskończone wykonanie [2p]

Napisz wyrażenie w podstawowej wersji rachunku lambda, które można wykonywać w nieskończoność.

4.4.2 Skończone i nieskończone wykonanie [2p]

Napisz wyrażenie w podstawowej wersji rachunku lambda, które można wykonać na co najmniej dwa sposoby:

- Jeden sposób pozwala na nieskończone wykonanie
- Drugi sposób nie pozwala na nieskończone wykonanie, tj. po skończonej liczbie kroków nie ma już co liczyć

5 Bonus: Kmina systemu typów [10p]

Zastanów się jak można podejść do zrobienia systemu, czyli albo algorytmu, albo deklaratywnego opisu „jeśli umielibyśmy tu wymyślić taki typ, to tu umielibyśmy wymyślić taki“ – dokładna treść znajduje się poniżej.

Zadanie jest otwarte – nie oczekuję konkretnej poprawnej odpowiedzi, chodzi głównie o próbę zrozumienia tematu, w szczególności liczą się:
- jakiekolwiek pomysły, nawet jeśli nie do końca działają
- rozwiązania z dodatkowymi założeniami które mogą upraszczać lub trochę zmieniać problem
- przykłady w stylu „to wygląda jakby mogło mieć taki typ, a to wygląda jakby mogło nie mieć typu“
- wszelkie konstruktywne pytania

5.1 Treść zadania

5.1.1 Język

Jako język weźmy nasz rachunek lambda w podstawowej wersji – mamy tylko zmienne, funkcje jednoargumentowe, wywołania funkcji i nic więcej.

Najsensowniejszą definicją poprawności jaką możemy przyjąć jest taka, że jak mamy wyrażenie $f(x)$ to f ma być funkcją, w przeciwnym razie takie wyrażenie jest niepoprawne.

5.1.2 Typy

Typy będziemy chcieli mieć takie:
- 0 jest typem podstawowym, możemy go interpretować trochę jak `any`, `object`, albo `void*` – pod spodem jest jakaś wartość ale nie mamy pojęcia jaka
- jeśli σ jest typem i τ jest typem to $\sigma \to \tau$ jest typem funkcji z σ w τ (czyli można robić dowolne funkcje na tym co już mamy) – interpretacja jest taka, że pod spodem jest funkcja którą można wywołać.

Innymi słowy typami są 0, $0 \to 0$, $0 \to (0 \to 0)$, $(0 \to 0) \to 0$, i tak dalej…

5.1.3 Cel

Dostajemy jakieś wyrażenie e i chcemy sprawdzić czy można mu przypisać jakiś typ τ w powyższej postaci – taki że jeśli przypisaliśmy mu typ to e jest poprawne – czyli gdybyśmy próbowali je policzyć to się nie wywali (czyli zawsze wywołujemy coś co jest funkcją).

5.1.4 Przykłady

- $\lambda x.x$ może dostać typ $0 \to 0$
- $\lambda x.\lambda y.x$ może dostać typ $0 \to (0 \to 0)$
- $\lambda x.\lambda y.x(y)$ może dostać typ $(0 \to 0) \to (0 \to 0)$
- $(\lambda x.x)(\lambda x.x)$ może dostać typ $0 \to 0$
- $\lambda x.x(x)$ może nie dostać typu, bo żaden skończony typ nie pozwala na wywołanie samego siebie w ten sposób

Ale tak naprawdę finalna definicja co kiedy może dostać typ należy do Was (starając się jednak dać typ tym rzeczom co wyglądają poprawnie i nie dać tym co wyglądają niepoprawnie).