# Funkcyjny język programowania dla zbiorów z atomami

*(autoreferat rozprawy doktorskiej)*

**Michał Szynwelski**

*Wydział Matematyki, Informatyki i Mechaniki, Uniwersytet Warszawski*

Marzec 2022

## Motywacja

Koncepcja nieskończoności to ważny temat w informatyce. Na pierwszy rzut oka mogłoby się wydawać, że programy komputerowe, dysponując tylko skończoną pamięcią oraz w zamierzeniu wykonujące obliczenia w skończonym czasie, mogą operować tylko na skończonych danych. Okazuje się jednak, że wiele algorytmów i struktur danych daje się rozszerzyć do świata obiektów nieskończonych. Na przykład algorytm dodawania elementu na początek listy jednokierunkowej wygląda tak samo niezależnie od tego, czy dana lista jest skończona czy nieskończona.

Jednak nie wszystkie operacje mają taką własność. Procedury wyszukiwania elementu na liście nie można łatwo rozszerzyć na listy nieskończone. Podobnie nie każda reprezentacja struktury danych może być wykorzystana dla struktur nieskończonych. Na przykład reprezentacja przechowująca wszystkie elementy listy nie może być użyta dla list nieskończonych.

Dlatego w przypadku struktur nieskończonych kluczowe jest, aby reprezentować je w sposób skończony oraz aby efektywnie wykonywać na nich operacje. Najlepiej, by taka reprezentacja dodatkowo była na tyle abstrakcyjna, żeby jej użytkownik wykonywał na niej operacje w ten sam sposób, niezależnie od tego, czy reprezentujemy skończoną czy nieskończoną strukturę.

W świecie języków programowania skończona reprezentacja struktur nieskończonych często realizowana jest za pomocą leniwego podejścia. Zamiast przechowywać wszystkie elementy danej struktury, przechowywany jest algorytm wyliczania poszczególnych wartości. Wyliczenie to jest uruchamiane dopiero kiedy dana wartość faktycznie jest potrzebna. Za pomocą takiego podejścia można w efektywny sposób wykonywać operacje filtrowania lub mapowania elementów listy. Jednakże takie rozwiązanie ma swoje ograniczenia. Na przykład nie można w skończonym czasie sprawdzić, czy lista zawiera dany element albo nie można takiej listy wyświetlić na ekranie.

W matematyce bogatym źródłem nieskończonych obiektów, które można w sposób skończony reprezentować, są relacyjne struktury definiowalne w logice pierwszego

rzędu. Na przykład zbiór par liczb naturalnych, które nie są sobie równe, może zostać przedstawiony następująco:

$$\{(a, b) \mid a, b \in \mathbb{N}, a \neq b\}$$

Jak widać powyższa notacja jest czytelna i zrozumiała, a sprawdzenie, czy dany element należy do powyższego zbioru, sprowadza się do zweryfikowania, czy jest to para różnych liczb naturalnych. I w tym aspekcie powyższa reprezentacja ma przewagę nad podejściem leniwym.

W podobny sposób można reprezentować bardziej złożone obiekty. Przykładowo nieskończony graf pełny, w którym zbiór wierzchołków to zbiór liczb naturalnych, a krawędziami są pary nieuporządkowe różnych liczb naturalnych, wygląda następująco:

$$(\mathbb{N},\ \{\{a, b\} \mid a, b \in \mathbb{N}, a \neq b\})$$

Obie powyższe struktury są definiowalne w logice pierwszego rzędu nad strukturą liczb naturalnych $\mathbb{N}$ z relacją równości. Gdybyśmy rozważali liczby wymierne, to używając tej samej notacji można reprezentować na przykład zbiór domkniętych przedziałów następująco:

$$\{\{c \mid c \in \mathbb{Q}, a \leq c \leq b\} \mid a, b \in \mathbb{Q}\} \tag{1}$$

Powyższy zbiór jest definiowalny w logice pierwszego rzędu nad zbiorem $\mathbb{Q}$ z relacją porządku $\leq$.

Elementy struktur relacyjnych, takie jak liczby naturalne lub wymierne w powyższych przykładach, nazywamy *atomami*. Zbiór relacji pochodzących z tych struktur w sposób naturalny ogranicza obliczenia, które będą na nich wykonywane. Oznacza to, że w przypadku liczb naturalnych nie mamy ambicji, żeby sprawdzać, czy dany zbiór zawiera tylko liczby parzyste. Takie sprawdzenie nie jest możliwe za pomocą dostępnej relacji równości. Za to możliwe jest sprawdzenie na przykład, czy zbiór jest pusty lub czy zawiera się w innym zbiorze.

Poza reprezentowaniem potencjalnie nieskończonych skruktur danych, chcielibyśmy także w sposób efektywny wykonywać na nich operacje. Wspomniana efektywność jest mocno oparta na własnościach użytych struktur relacyjnych w logice pierwszego rzędu. Dlatego struktura atomów powinna posiadać rozstrzygalną teorię pierwszego rzędu. Struktury, które spełniają tę własność, to wspomniane już:

- $(N, =)$ – liczby naturalne z relacją równości,
- $(Q, \leq)$ – liczby wymierne z relacją porządku.

Celem rozprawy jest stworzenie funkcyjnego języka programowania, który operuje na powyższych strukturach relacyjnych. Dodatkowo chcielibyśmy, aby z języka korzystało się tak samo, niezależnie czy obliczenia są wykonywane na obiektach skończonych, czy nieskończonych.

Rozważmy jako przykład program napisany w języku Haskell, który wylicza domknięcie przechodnie relacji binarnej. Relacja jest reprezentowana przez zbiór par. W celu wyliczenia domknięcia zdefiniujmy najpierw funkcję, która wyliczy złożenie relacji:

```
compose :: (Ord a, Ord b, Ord c) => Set(a,b) -> Set(b,c) -> Set(a,c)
compose r s = sum (map (\(a,b) -> map (\(_,c) -> (a,c))
                                      (filter ((==b) . fst) s))
                       r)
```

Wykorzystując tę funkcję, możemy wyliczyć przechodnie domknięcie następująco:

```
transitiveClosure :: Ord a => Set(a,a) -> Set(a,a)
transitiveClosure r = let r' = union r (compose r r)
                      in if r==r' then r else (transitiveClosure r')
```

Powyższy przykład wykorzystuje funkcje pomocnicze z modułu `Data.Set`, który służy do obsługi zbiorów skończonych.

```
sum = unions . elems :: Set (Set a) -> Set a
map :: Ord b => (a -> b) -> Set a -> Set b
filter :: (a -> Bool) -> Set a -> Set a
union :: Ord a => Set a -> Set a -> Set a
```

Jednym z celów rozprawy jest stworzenie języka programowania zawierającego odpowiednik konstruktora typu `Set` dla zbiorów nieskończonych. Wówczas powyższy kod będzie mógł zostać wykorzystany zarówno dla zbiorów skończonych, jak i dla nieskończonych definiowalnych w logice pierwszego rzędu.

Aby osiągnąć ten cel, stworzony został język programowania N$\lambda$ [Klin and Szynwelski, 2016] jako rozszerzenie rachunku lambda z typami prostymi. Rozprawa zawiera dokładny opis składni i semantyki tego języka oraz pokazuje jego podstawowe własności. Wraz z teoretycznym opisem języka N$\lambda$ została dostarczona jego implementacja w języku Haskell. Rozprawa objaśnia szczegóły implementacji oraz przedstawia przykładowe przypadki użycia.

W ramach niniejszego referatu przeanalizujemy podstawowe pojęcia związane ze zbiorami z atomami, czyli strukturami, na których oparty jest model obliczeniowy języka N$\lambda$. Następnie scharakteryzujemy podstawową wersję tego języka oraz krótko opiszemy jego implementację.

## Zbiory z atomami

Obliczenia na zbiorach z atomami odbywają się zawsze w kontekście wybranej przeliczalnej nieskończonej struktury relacyjnej nad skończoną sygnaturą. Taką strukturę oznaczamy $\mathcal{A}$, a jej elementy nazywamy *atomami*. Dla uproszczenia notacji zbiór atomów także będziemy oznaczać przez $\mathcal{A}$. Sygnatura zawiera symbole relacji interpretowanych na zbiorze atomów. Aby struktura mogła być użyta w naszym modelu obliczeniowym, musi spełniać następujące warunki:

- posiadać rozstrzygalną teorię pierwszego rzędu,
- być oligomorficzna,
- być homogeniczna.

Dwa ostatnie warunki to podstawowe własności z Teorii Modeli (szczegóły można znaleźć w [Hodges, 1993]).

Powyższe trzy warunki spełniają między innymi atomy równościowe $(\mathbb{N}, =)$ oraz atomy uporządkowane $(\mathbb{Q}, \leq)$.

Przez *zbiór z atomami* rozumiemy w pewnym uproszczeniu zbiór złożony z atomów lub innych prostszych zbiorów z atomami. Dodatkowo taki zbiór musi być zbudowany za pomocą formuł pierwszego rzędu, które wykorzystują tylko relacje pochodzące z ustalonej struktury relacyjnej atomów oraz skończonej liczby stałych odnoszących się do konkretnych atomów.

Przykładem zbioru z atomami dla struktury $(\mathbb{N}, =)$ jest zbiór par liczb naturalnych[1], gdzie pierwsza liczba jest różna od 7, a druga różna od 13:

$$\{(a, b) \mid a, b \in \mathbb{N}, a \neq 7 \wedge b \neq 13\} \tag{2}$$

Dla struktury $(\mathbb{Q}, \leq)$ jako przykład może służyć zbiór przedziałów otwartych zawierających liczbę $\frac{3}{4}$:

$$\{\{c \mid c \in \mathbb{N},\ a < c < b\} \mid a, b \in \mathbb{N},\ a < \frac{3}{4} < b\} \tag{3}$$

Jak widać na powyższych przykładach, zbiory te są skonstruowane za pomocą relacji ze struktury (nierówność i relacja mniejszości może zostać uzyskana za pomocą formuł pierwszego rzędu odpowiednio z relacji $=$ i $\leq$). W omawianych przykładach mamy też odwołanie do konkretnych wartości atomów (odpowiednio zbiory $\{7, 13\}$ i $\{\frac{3}{4}\}$). Skończony zbiór tych konkretnych wartości nazywamy *wsparciem* zbioru z atomami (ang. *support*).

Dla zbiorów z atomami możemy zdefiniować pojęcie *orbity*. Dwa elementy zbioru z atomami znajdują się w tej samej orbicie, jeżeli istnieje *automorfizm*, który jest identycznością na wsparciu zbioru z atomami, mapujący pierwszy element na drugi. Idea skończonej reprezentacji nieskończonych zbiorów z atomami opiera się na dodatkowym warunku, który taki zbiór musi spełniać. Mówimy, że zbiór jest *skończenie orbitowy*, jeżeli można go podzielić na skończenie wiele orbit. Innymi słowy, taki zbiór jest skończony z dokładnością do automorfizmów atomów. Zatem można powiedzieć, że orbitowa skończoność jest uogólnieniem skończoności. W szczególności zbiory skończone również są skończenie orbitowe.

Przykład (2) jest zbiorem o czterech orbitach w odniesieniu do wsparcia $\{7, 13\}$: podzbiór par równych liczb, podzbiór par z pierwszą liczbą równą 13, podzbiór par z drugą liczbą równą 7 oraz podzbiór pozostałych elementów zbioru. Oczywiście wszystkie podzbiory muszą spełniać warunek z definicji zbioru. Z kolei zbiór (3) jest jednoorbitowy.

Zbiory z atomami tworzą uniwersum teorii zbiorów, podobne do uniwersum von Neumanna dla klasycznych zbiorów, o wielu przydatnych własnościach. Na przykład zbiory te są zamknięte na sumy skończone, produkty, przecięcia, różnice, rzutowania i wiele innych operacji. Dzięki tym własnościom możemy dla zbiorów z atomami rozpatrywać standardowe pojęcia matematyczne jak funkcje czy relacje równoważności. Co więcej możemy też rozważać struktury danych, które oparte są na zbiorach z atomami, zamiast tylko na zbiorach skończonych. Na przykład grafy, dla których zbiory wierzchołków i krawędzi są zbiorami z atomami. Podobnie można zdefiniować automaty, maszyny Turinga, gramatyki bezkontekstowe i wiele innych (więcej przykładów można znaleźć w [Bojańczyk et al., 2014] lub w [Bojańczyk, 2019]).

Jak widać na przykładach zbiorów z atomami, mogą one zostać zapisane za pomocą standardowej notacji matematycznej (zwanej po angielsku *set-builder notation*). Dla skończenie orbitowych zbiorów z atomami notację tę możemy zdefiniować następująco:

[1] para $(x, y)$ może być reprezentowane za pomocą kodowania Kuratowskiego $\{\{x\}, \{x, y\}\}$

**Definicja 1.** *Dla ustalonej struktury relacyjnej $\mathcal{A}$ oraz przeliczalnego zbioru zmiennych atomowych* ***wyrażeniem zbioru*** *nazywamy skończony zbiór $\{\xi_1, ..., \xi_n\}$ wyrażeń postaci:*

$$\xi \quad = \quad e \ : \ \phi \ \ for \ \ a_1, \ldots, a_k \in \mathcal{A},$$

*gdzie*

- *$e$ jest wyrażeniem zbioru lub zmienną atomową,*
- *$\phi$ jest formułą pierwszego rzędu nad sygnaturą atomów oraz zmiennymi atomowymi,*
- *$a_1, \ldots, a_k$ są zmiennymi atomowymi.*

Zbiory, które mogą być reprezentowane w ten sposób nazywamy zbiorami *definiowalnymi*. Jeżeli wyrażenie zbioru posiada zmienne wolne, to żeby wyznaczyć zbiór definiowalny, trzeba dostarczyć wartościowanie tych zmiennych.

Okazuje się, że zbiór jest definiowalny wtedy i tylko wtedy, gdy jest skończenie orbitowym zbiorem z atomami ([Bojańczyk, 2019, Theorem 4.10]). Dlatego wyrażenie zbioru jest odpowiednią skończoną reprezentacją dla skończenie orbitowych zbiorów z atomami. Dokładnie ta reprezentacja jest użyta w modelu obliczeniowym języka programowania N$\lambda$.

## Język programowania

Język N$\lambda$ jest silnie typowanym językiem funkcyjnym, który umożliwia obliczenia na skończenie orbitowych zbiorach z atomami. Zbiory te są wewnętrznie reprezentowane przez wyrażenia zbioru. Nie oznacza to jednak, że użytkownik języka definiuje zbiory poprzez te wyrażenia. Programy w N$\lambda$ składają się z funkcji do tworzenia i modyfikowania zbiorów jak w językach programowania opartych na zbiorach skończonych. Zatem to nie użytkownik konstruuje wyrażenia zbioru, ale są one tworzone w trakcie wywoływania funkcji tworzących zbiór, a następnie przez kolejne funkcje wyrażenia te są zmieniane. Wyrażenia służą także do wyświetlania zbioru jako wynik wykonania programu.

Podstawowa wersja języka jest rozszerzeniem rachunku lambda z typami prostymi [Church, 1940, Barendregt, 1993]. Typy języka są podzielone na dwa rodzaje: typy *elementów* i typy *funkcyjne*. Zbiory mogą zawierać tylko obiekty posiadające pierwszy rodzaj typów. Wszystkie typy w podstawowej wersji N$\lambda$ wyglądają następująco:

$$\tau ::= \mathbb{A} \mid \mathbb{B} \mid \mathbb{S}\tau$$
$$\alpha, \beta ::= \tau \mid \alpha \to \beta$$

Typy elementów są oznaczone za pomocą $\tau$ i mogą przyjmować następującą postać:

- $\mathbb{A}$ jest typem atomów ze struktury relacyjnej $\mathcal{A}$,
- $\mathbb{B}$ jest typem wartości logicznych (zakres typu zostanie następnie rozszerzony o formuły pierwszego rzędu),
- $\mathbb{S}$ jest jednoargumentowym konstruktorem typu, który może być zastosowany tylko do typu elementów. Obiekty, które mają typ $\mathbb{S}\tau$ to zbiory definiowalne, które zawierają elementy typu $\tau$.

Poza powyższymi typami, mamy też standardowy konstruktor typu $\rightarrow$, który tworzy typ funkcyjny.

Aby dostarczyć funkcyjny język programowania, który pozwala wykonywać obliczenia na zbiorach definiowalnych, rozszerzamy standardowy zbiór termów rachunku lambda z typami prostymi: zmienne, abstrakcje i aplikacje o zestaw stałych. Zatem składnia języka wygląda następująco:

$$M ::= x \mid \lambda x.M \mid MM \mid C$$

$C$ obejmuje następujący zbiór stałych:

| | | |
|---|---|---|
| `empty` | $: \mathbb{S}\tau$ | (zbiór pusty) |
| `atoms` | $: \mathbb{S}\mathbb{A}$ | (zbiór wszystkich atomów) |
| `insert` | $: \tau \rightarrow \mathbb{S}\tau \rightarrow \mathbb{S}\tau$ | (dodaje element do zbioru) |
| `map` | $: (\tau_1 \rightarrow \tau_2) \rightarrow \mathbb{S}\tau_1 \rightarrow \mathbb{S}\tau_2$ | (aplikuje funkcję do każdego elementu zbioru) |
| `sum` | $: \mathbb{S}\mathbb{S}\tau \rightarrow \mathbb{S}\tau$ | (suma rodziny zbiorów) |
| `true`, `false` | $: \mathbb{B}$ | (wartości logiczne) |
| `not` | $: \mathbb{B} \rightarrow \mathbb{B}$ | (negacja) |
| `and`, `or` | $: \mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B}$ | (koniunkcja i alternatywa) |
| `isEmpty` | $: \mathbb{S}\tau \rightarrow \mathbb{B}$ | (sprawdzenie pustości zbioru) |
| `if` | $: \mathbb{B} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$ | (wyrażenie warunkowe) |

Ponadto dodajemy stałe, które wprost wynikają z wybranej struktury relacyjnej $\mathcal{A}$ atomów. Dla atomów równościowych będzie to:

$$\mathtt{eq}_{\mathbb{A}} : \mathbb{A} \rightarrow \mathbb{A} \rightarrow \mathbb{B} \qquad \text{(relacja równości na atomach)}$$

Ponadto dla atomów uporządkowanych mamy:

$$\mathtt{leq}_{\mathbb{A}} : \mathbb{A} \rightarrow \mathbb{A} \rightarrow \mathbb{B} \qquad \text{(relacja porządku na atomach).}$$

W przypadku wybrania innej struktury relacyjnej, ta część języka może się zmienić.

Korzystając z powyższych podstawowych funkcji, można zdefiniować wiele dodatkowych funkcji, które pozwolą w wygodny sposób wykonywać oczekiwane operacje na zbiorach definiowalnych. Na przykład można zdefiniować poniższe funkcje:

| | | |
|---|---|---|
| `singleton` | $: \tau \rightarrow \mathbb{S}\tau$ | (tworzy zbiór jednoelementowy) |
| `filter` | $: (\tau \rightarrow \mathbb{B}) \rightarrow \mathbb{S}\tau \rightarrow \mathbb{S}\tau$ | (zbiór elementów spełniających podany warunek) |
| `delete` | $: \tau \rightarrow \mathbb{S}\tau \rightarrow \mathbb{S}\tau$ | (usuwa dany element ze zbioru) |
| `exists` | $: (\tau \rightarrow \mathbb{B}) \rightarrow \mathbb{S}\tau \rightarrow \mathbb{B}$ | (czy istnieje element spełniający dany warunek) |
| `forall` | $: (\tau \rightarrow \mathbb{B}) \rightarrow \mathbb{S}\tau \rightarrow \mathbb{B}$ | (czy wszystkie elementy spełniają dany warunek) |
| `contains` | $: \mathbb{S}\tau \rightarrow \tau \rightarrow \mathbb{B}$ | (czy zbiór posiada dany element) |
| `union` | $: \mathbb{S}\tau \rightarrow \mathbb{S}\tau \rightarrow \mathbb{S}\tau$ | (suma dwóch zbiorów) |
| `intersection` | $: \mathbb{S}\tau \rightarrow \mathbb{S}\tau \rightarrow \mathbb{S}\tau$ | (przecięcie dwóch zbiorów) |
| `isSubsetOf` | $: \mathbb{S}\tau \rightarrow \mathbb{S}\tau \rightarrow \mathbb{B}$ | (relacja podzbioru) |
| $\mathtt{eq}_{\mathbb{S}\tau}$ | $: \mathbb{S}\tau \rightarrow \mathbb{S}\tau \rightarrow \mathbb{B}$ | (równość dwóch zbiorów) |

$$\texttt{implies} : \mathbb{B} \to \mathbb{B} \to \mathbb{B} \qquad \text{(implikacja)}$$
$$\texttt{eq}_{\mathbb{B}} : \mathbb{B} \to \mathbb{B} \to \mathbb{B} \qquad \text{(równoważność wartości logicznych)}$$

z następującą implementacją:

$$\begin{aligned}
\texttt{singleton x} &= \texttt{insert x empty}\\
\texttt{filter f s} &= \texttt{sum (map } (\lambda\texttt{x.if (f x) (singleton x) empty)}\texttt{ s)}\\
\texttt{delete x s} &= \texttt{filter } (\lambda\texttt{y.not (eq}_{\tau}\texttt{ x y))}\texttt{ s}\\
\texttt{exists f s} &= \texttt{not (isEmpty (filter f s))}\\
\texttt{forall f s} &= \texttt{isEmpty (filter } (\lambda\texttt{x.not (f x))}\texttt{ s)}\\
\texttt{contains s x} &= \texttt{exists } (\texttt{eq}_{\tau}\texttt{ x)}\texttt{ s}\\
\texttt{union s t} &= \texttt{sum (insert s (singleton t))}\\
\texttt{intersection s t} &= \texttt{filter (contains t) s}\\
\texttt{isSubsetOf s t} &= \texttt{forall (contains t) s}\\
\texttt{eq}_{\mathbb{S}\tau}\texttt{ s t} &= \texttt{and (isSubsetOf s t) (isSubsetOf t s)}
\end{aligned}$$

$$\begin{aligned}
\texttt{implies p q} &= \texttt{or (not p) q}\\
\texttt{eq}_{\mathbb{B}}\texttt{ p q} &= \texttt{and (implies p q) (implies q p)}
\end{aligned}$$

Przytoczony zestaw funkcji pozwala na implementację wielu programów korzystających ze zbiorów definiowalnych. Na przykład zbiór (1) może zostać wyliczony za pomocą programu:

```
sum
  (map
    (λx.map
         (λy.filter
              (λz.(and (leq_𝔸 x z) (leq_𝔸 z y))
              atoms)
         atoms)
    atoms)
```

Z powyższego opisu języka oraz podanych przykładów nie wynika jednak, w jaki sposób konstruować zbiory nieskończone i wykonywać na nich obliczenia. Dokładny model obliczeniowy języka przedstawiono w rozprawie w postaci semantyki operacyjnej. Tutaj ograniczymy się do podstawowych założeń.

- Zbiory nie są reprezentowane poprzez zapamiętanie ich elementów, ale za pomocą wyrażeń zbiorów (Definicja 1).
- Zmienne w języku są podzielone na dwie grupy. Standardowe zmienne termowe $x$ służą do konstruowania abstrakcji oraz mogą przyjmować dowolny typ. Dodatkowo mamy do dyspozycji zmienne atomowe $a$, które mogą mieć tylko typ $\mathbb{A}$, ale w przeciwieństwie do zmiennych termowych, mogą pojawiać się w formułach pierwszego rzędu i w kontekstach.

- Obiekty typu $\mathbb{B}$ to nie tylko wartości logiczne, ale formuły pierwszego rzędu nad zmiennymi atomowymi.
- Redukcja termów w semantyce operacyjnej odbywa się w ustalonym kontekście, który precyzuje relacje jakie zachodzą między zmiennymi atomowymi.
- Czasami warunek $\phi$ w wyrażeniu warunkowym (`if` $\phi\, a\, b$) nie jest jednoznacznie prawdziwy lub fałszywy. W takim przypadkach nie jest jasne, czy wyrażenie warunkowe powinno przyjąć wartość $a$ lub $b$. Dlatego wprowadza się nową konstrukcję zwaną *wariantami*, która przyjmuje wartość $a$ lub $b$ w zależności od wartości formuły $\phi$ w danym kontekście.

Formalnie gramatyka języka jest rozszerzona do następującej postaci:

$$M ::= x \mid \lambda x.M \mid MM \mid C \mid a \mid \phi \mid \{M\colon \phi \text{ for } \sigma, \ldots, M\colon \phi \text{ for } \sigma\} \mid a\colon \phi | \cdots | a\colon \phi$$

gdzie

- $C$ oznacza ten sam zbiór stałych co wcześniej,
- $a$ to przeliczalny zbiór zmiennych atomowych (rozłączny od zmiennych termowych oznaczonych przez $x$),
- $\phi$ obejmuje formuły pierwszego rzędu nad sygnaturą struktury relacyjnej $\mathcal{A}$ oraz zmiennymi atomowymi,
- $\{M\colon \phi \text{ for } \sigma, \ldots, M\colon \phi \text{ for } \sigma\}$ to wyrażenie zbioru, w którym $\rho$ jest skończoną listą zmiennych atomowych,
- $a\colon \phi | \cdots | a\colon \phi$ oznacza term w postaci wariantów.

Model obliczeniowy języka został przedstawiony w rozprawie poprzez semantykę operacyjną zawierającą 25 reguł redukcyjnych dla wyrażeń składających się z powyższych termów. Każda reguła redukcyjna ma zastosowanie w określonym kontekście, będącym formułą pierwszego rzędu. Program N$\lambda$ jest redukowany za pomocą tych reguł, aż do osiągnięcia postaci normalnej, która jest wynikiem działania programu.

Poza semantyką operacyjną praca prezentuje również reguły typowania, za pomocą których każdemu prawidłowemu wyrażeniu języka można przypisać jeden ze wspomnianych typów.

W rozprawie opisana jest także semantyka denotacyjna, która przypisuje termom ich znaczenie w postaci obiektów matematycznych takich jak zbiory, funkcje, atomy lub formuły.

Tak zdefiniowany język ma kilka oczekiwanych własności:

- Reguły redukcyjne semantyki operacyjnej nie zmieniają typu redukowanego wyrażenia (własność ta nazywana jest *subject reduction* [Curry and Feys, 1958]).
- Semantyka operacyjna i denotacyjna języka ściśle sobie odpowiadają. Oznacza to, że reguły redukcyjne semantyki operacyjnej nie zmieniają znaczenia (denotacji) programów języka.

- Semantyka operacyjna spełnia twierdzenie Churcha-Rossera [Church and Rosser, 1936]. Wynika z tego, że kolejność stosowania reguł redukcyjnych dla wybranego programu N$\lambda$ nie wpływa na końcowy wynik tego programu.
- Semantyka operacyjna ma własność silnej normalizacji [Hankin, 1994], czyli proces redukcji zawsze kończy się termem w postaci normalnej. Innymi słowy, dla żadnego programu nie istnieje nieskończony ciąg redukcji.

Poza opisem podstawowej wersji języka rozprawa prezentuje także możliwe rozszerzenia tej wersji o konstrukcje występujące we wszystkich znaczących pełnoprawnych funkcyjnych językach programowania. Elementy, o które język może zostać rozszerzony, to między innymi rekurencja, polimorfizm czy typy algebraiczne.

Język programowania N$\lambda$ to jedno z podejść do obliczeń na zbiorach definiowalnych. Rozprawa zawiera opisy innych koncepcji realizacji tego zagadnienia. Chodzi o język LOIS, który umożliwia takie obliczenia w paradygmacie języka imperatywnego [Bojańczyk and Toruńczyk, 2012], [Kopczyński and Toruńczyk, 2016], [Kopczyński and Toruńczyk, 2017]. Inne opisane podejście to model obliczeń, w którym zbiory skończenie orbitowe są reprezentowane poprzez wybrane elementy poszczególnych orbit. Taka metoda była użyta u bezpośredniego poprzednika języka N$\lambda$ [Bojańczyk et al., 2012].

## Implementacja

Poza opisem teoretycznym języka N$\lambda$ rozprawa dostarcza implementację w języku Haskell [Szynwelski, 2022c, Szynwelski, 2022b].

Haskell jest nowoczesnym językiem funkcyjnym ogólnego przeznaczenia. Charakteryzuje się rozbudowanym poliformicznym silnym systemem typów. Jego dużą zaletą jest leniwe podejście i czystość funkcji (brak skutków ubocznych). Wyróżnia się syntaktycznym wsparciem monad oraz zwięzłością i czytelnością kodu. Dlatego implementacja języka N$\lambda$ to połączenie wszystkich zalet Haskella i modelu obliczeń na zbiorach nieskończonych.

W związku z tym, że składnia Haskella jest inspirowana rachunkiem lambda, również składnia implementacji języka N$\lambda$ jest zbliżona do termów języka opisanych w części teoretycznej. W miejsce typów

$$\mathbb{A},\ \mathbb{B},\ \mathbb{S}\tau,\ \alpha \to \beta$$

mamy

$$\texttt{Atom},\ \texttt{Formula},\ \texttt{Set a},\ \texttt{a} \to \texttt{b}.$$

Natomiast stałe języka w implementacji wyglądają następująco:

```
empty :: Set a
atoms :: Set Atom
insert :: Nominal a => a -> Set a -> Set a
map :: (Nominal a, Nominal b) => (a -> b) -> Set a -> Set b
sum :: Nominal a => Set (Set a) -> Set a
true :: Formula
false :: Formula
not :: Formula -> Formula
```

```
(/\) :: Formula -> Formula -> Formula
(\/) :: Formula -> Formula -> Formula
isEmpty :: Set a -> Formula
ite :: Conditional a => Formula -> a -> a -> a
```

Dodatkowo język zawiera stałe wynikające ze struktury relacyjnej:

```
eq :: Nominal a => a -> a -> Formula
le :: Atom -> Atom -> Formula
```

Do realizacji zagadnień związanych z kontekstami, wyrażeniami warunkowymi lub typami elementów zostały użyte Haskellowe klasy typów. W implementacji języka mają one nazwy `Contextual`, `Conditional`, `Nominal`.

Model obliczeniowy języka N$\lambda$ w dużym stopniu oparty jest na rozwiązywaniu formuł pierwszego rzędu, toteż implementację zintegrowano z zewnętrznym narzędziem typu *SMT Solver* [Barrett et al., 2009]. W tym wypadku zostało wybrane rozwiązanie o nazwie Z3 firmy Microsoft [De Moura and Bjørner, 2008, Bjørner et al., 2021].

Aby zwiększyć efektywność rozwiązywania formuł, została zaimplementowana eliminacja kwantyfikatorów we wspieranych strukturach relacyjnych oparta na metodzie zaproponowanej przez [Loos and Weispfenning, 1993] oraz [Nipkow, 2008].

Poza realizacją samego języka w repozytorium kodu znajdują się także przykładowe przypadki użycia języka. Zaimplementowane zostały między innymi algorytmy na nieskończonych grafach i automatach. Zagadnienia, które zostały rozwiązane w N$\lambda$ to między innymi domknięcie przechodnie grafu, sprawdzenie jego spójności, wyszukiwanie cyklów, badanie dwudzielności czy problemu kolorowania.

Dla algorytmów operujących na nieskończonych automatach dostępne są funkcje, badające pustość automatu, jego sumę lub przecięcie, sprawdzenie, czy jest to automat deterministyczny, czy w końcu algorytm minimalizacji automatu deterministycznego. Poza tym język N$\lambda$ został wykorzystany do implementacji algorytmu uczenia automatów [Moerman et al., 2016].

## Publikacje

- Wstępne wyniki, które stały się podstawą doktoratu:

  [Klin and Szynwelski, 2016] Klin, B. and Szynwelski, M. (2016). SMT Solving for Functional Programming over Infinite Structures. *Electronic Proceedings in Theoretical Computer Science*, 207:57–75

- Pierwsze poważniejsze użycie N$\lambda$ do implementacji algorytmu uczenia automatów:

  [Moerman et al., 2016] Moerman, J., Sammartino, M., Silva, A., Klin, B., and Szynwelski, M. (2016). Learning nominal automata. *ACM SIGPLAN Notices*, 52

- Strona internetowa z opisem języka i pełną dokumentacją oraz repozytorium kodu, gdzie użytkownicy mogą też zgłaszać problemy lub zapotrzebowanie na fukcjonalność:

  [Szynwelski, 2022c, Szynwelski, 2022a, Szynwelski, 2022b]

  - N$\lambda$ website: https://www.mimuw.edu.pl/~szynwelski/nlambda/
  - N$\lambda$ documentation: https://www.mimuw.edu.pl/~szynwelski/nlambda/doc/
  - N$\lambda$ repository on GitHub: https://github.com/szynwelski/nlambda

# Literatura

[Barendregt, 1993] Barendregt, H. P. (1993). *Lambda Calculi with Types*, page 117–309. Oxford University Press, Inc., USA.

[Barrett et al., 2009] Barrett, C., Sebastiani, R., Seshia, S., and Tinelli, C. (2009). Satisfiability modulo theories. In Biere, A., Heule, M., Maaren, H., and Walsh, T., editors, *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*, chapter 12, page 825–885. IOS Press.

[Bjørner et al., 2021] Bjørner, N., De Moura, L., et al. (2021). Z3. https://github.com/Z3Prover/z3. [Online; dostęp: marzec 2022].

[Bojańczyk, 2019] Bojańczyk, M. (2019). *Slightly Infinite Sets*. University of Warsaw. https://www.mimuw.edu.pl/~bojan/paper/atom-book [Online; dostęp: marzec 2022].

[Bojańczyk et al., 2012] Bojańczyk, M., Braud, L., Klin, B., and Lasota, S. (2012). Towards nominal computation. *SIGPLAN Not.*, 47(1):401–412.

[Bojańczyk et al., 2014] Bojańczyk, M., Klin, B., and Lasota, S. (2014). Automata theory in nominal sets. *Logical Methods in Computer Science*, 10.

[Bojańczyk and Toruńczyk, 2012] Bojańczyk, M. and Toruńczyk, S. (2012). Imperative programming in sets with atoms. 18:4–15.

[Church, 1940] Church, A. (1940). A formulation of the simple theory of types. *J. Symb. Log.*, 5:56–68.

[Church and Rosser, 1936] Church, A. and Rosser, J. B. (1936). Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472 – 482.

[Curry and Feys, 1958] Curry, H. B. and Feys, R. (1958). *Combinatory Logic Vol. 1*. North-Holland Publishing Company.

[De Moura and Bjørner, 2008] De Moura, L. and Bjørner, N. (2008). Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg. Springer-Verlag.

[Hankin, 1994] Hankin, C. (1994). *Lambda Calculi: A Guide for Computer Scientists*. Graduate texts in computer science. Clarendon Press.

[Hodges, 1993] Hodges, W. (1993). *Model Theory*. Encyclopedia of Mathematics and its Applications. Cambridge University Press.

[Klin and Szynwelski, 2016] Klin, B. and Szynwelski, M. (2016). SMT Solving for Functional Programming over Infinite Structures. *Electronic Proceedings in Theoretical Computer Science*, 207:57–75.

[Kopczyński and Toruńczyk, 2016] Kopczyński, E. and Toruńczyk, S. (2016). LOIS: an Application of SMT Solvers.

[Kopczyński and Toruńczyk, 2017] Kopczyński, E. and Toruńczyk, S. (2017). LOIS: syntax and semantics. *ACM SIGPLAN Notices*, 52:586–598.

[Loos and Weispfenning, 1993] Loos, R. and Weispfenning, V. (1993). Applying linear quantifier elimination. *Comput. J.*, 36:450–462.

[Moerman et al., 2016] Moerman, J., Sammartino, M., Silva, A., Klin, B., and Szynwelski, M. (2016). Learning nominal automata. *ACM SIGPLAN Notices*, 52.

[Nipkow, 2008] Nipkow, T. (2008). Linear quantifier elimination. *Journal of Automated Reasoning*, 45:189–212.

[Szynwelski, 2022a] Szynwelski, M. (2022a). Nλ documentation. https://www.mimuw.edu.pl/~szynwelski/nlambda/doc/. [Online; dostęp: marzec 2022].

[Szynwelski, 2022b] Szynwelski, M. (2022b). Nλ repository on GitHub. https://github.com/szynwelski/nlambda. [Online; dostęp: marzec 2022].

[Szynwelski, 2022c] Szynwelski, M. (2022c). Nλ website. https://www.mimuw.edu.pl/~szynwelski/nlambda/. [Online; dostęp: marzec 2022].