# Statyczna analiza do zwijania funkcji wieloargumentowych

(Static analysis for folding multi-argument functions)

Szymon Jędras

Praca inżynierska

**Promotor:** Piotr Polesiuk

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

28 sierpnia 2024

**Streszczenie**

Currying jest nierozłączną częścią każdego współczesnego języka funkcyjnego. Opiera się na systemie typów, w którym funkcje są reprezentowane przez strzałki jednoargumentowe. Takie uproszczenie odbiera możliwość optymalizowania kodu używając technik opartych na wieloargumentowości funkcji, pomimo że większość pisanych funkcji jest wieloargumentowych. W tej pracy rozważymy system typów i efektów, który utrzymuje informacje o sposobach aplikowania funkcji. Wykorzystamy te informacje do wykonania statycznej analizy kodu. Następnie używając uzyskanych informacji przeprowadzimy optymalizację, pilnując, żeby nie zmienić obserwowalnego zachowania programu. Opowiemy o algorytmie pozwalającym na efektywne przeprowadzenie analizy i następującą transformację kodu.

---

Currying is an inseparable part of every modern functional language. It is based on a type system in which functions are represented by single-argument arrows. This simplification removes the possibility of optimizing code using techniques based on multi-argument functions, even though the majority of them are written that way. In this thesis, we consider a type-and-effect system that retains information about how functions are applied. We will use it to perform a static code analysis. Then, using the obtained insight, we will perform an optimization while ensuring the observable behavior of the program remains unchanged. We will discuss an algorithm that allows for efficient analysis and subsequent code transformation.

# Spis treści

**1. Wstęp** **7**

1.1. Problem . . . . . 7

1.2. Promyk nadziei . . . . . 8

**2. Analiza** **9**

2.1. Składnia języków . . . . . 9

2.2. Rachunek początkowy . . . . . 10

2.3. Język wieloargumentowy . . . . . 12

2.4. Rozwiązywanie podtypowania . . . . . 15

**3. Algorytm** **19**

3.1. Idea algorytmu . . . . . 19

3.2. Wymagania . . . . . 20

3.3. Struktura danych algorytmu . . . . . 20

3.4. Analiza efektów . . . . . 21

3.5. Analiza składania . . . . . 22

**4. Implementacja** **25**

4.1. Funkcjonalność . . . . . 25

4.2. Pipeline . . . . . 26

4.3. Wyniki optymalizacji . . . . . 27

**5. Dalsze prace** **29**

5.1. Dokładność pomiarów . . . . . 29

5.2. Dokładniejszy typechecking . . . . . 29

5.3. Polimorfizm efektowy . . . . . 29

**Bibliografia** **31**

# Rozdział 1.

# Wstęp

System typów ze strzałkami (ang. arrow types) stanowi fundament wielu języków programowania, opartych na rachunku lambda. Jego prostota jest jednym z kluczowych powodów popularności tego modelu wśród twórców języków funkcyjnych. Typy te są intuicyjne i łatwe do zrozumienia, co ułatwia analizę typów i implementację narzędzi wspomagających kompilację. W modelu tym każda funkcja jest traktowana jako wartość, która może być przekazywana jako argument do innej funkcji lub zwracana jako wynik. Jednym z najważniejszych aspektów tego podejścia jest możliwość częściowej aplikacji funkcji, znanej również jako currying[2]. Currying polega na tym, że funkcja wieloargumentowa jest przekształcana w serię funkcji jednoargumentowych. Takie podejście przynosi wiele korzyści, zwłaszcza w kontekście składania funkcji, ponieważ pozwala na bardziej elastyczne budowanie złożonych operacji z prostych elementów. Dzięki curryingowi programiści mogą tworzyć bardziej modularne i komponenty wielokrotnego użytku, co sprzyja pisaniu czytelnego i łatwego w utrzymania kodu.

## 1.1. Problem

Pomimo wielu zalet, currying niesie jednak za sobą również pewne wyzwania, zwłaszcza wydajności. Jednym z nich jest konieczność częstego alokowania pamięci. Każdorazowe wywołanie funkcji jednoargumentowej w ramach curryingu prowadzi do tworzenia nowego domknięcia (ang. closure), które w przypadku funkcji wieloargumentowych zostaje od razu zaaplikowane. To z kolei wiąże się z dodatkowymi kosztami alokacji pamięci. W skrajnych przypadkach, może to prowadzić do znacznego obciążenia systemu zarządzania pamięcią, co w językach z automatycznym zarządzaniem pamięcią, powoduje częstsze uruchamianie garbage collectora, więc duży nakład czasowy.

Sytuacja ta staje się szczególnie problematyczna w systemach, w których dynamiczne alokowanie pamięci i uruchamianie garbage collectora jest niedopuszczalne,

na przykład w oprogramowaniu czasu rzeczywistego. W takich przypadkach, częste tworzenie domknięć może prowadzić do nieakceptowalnych opóźnień.

Obecnie istnieją rozwiązania wspomnianych problemów. Implementacja środowiska wykonującego programy w OCamlu, nazywająca się ZINC [5], wydaje się mieć ciastko i je jeść jednocześnie dopuszcza currying, ale też unika alokacji przy każdej częściowej aplikacji. Robi to jednak kosztami w trakcie wykonywania, ponieważ każda aplikacja musi liczyć czy przekazuje wystarczająco argumentów do wykonania kodu, czy jest zmuszona do alokacji domknięcia.

## 1.2. Promyk nadziei

Jednak nie cała nadzieja stracona. O ile currying jest potężnym narzędziem, nie zawsze jest w pełni wykorzystywany. Częściowa aplikacja pozwala na wiele, ale często aplikujemy funkcję w pełni. Dobrze to zjawisko pokazują funkcje działające na strukturach danych.

```
1 let data = DataReader.read_from_file path
2
3 let average = (List.fold_left (+.) 0.0 data) / (List.length data)
4
5 let variance_squared = List.map (fun x -> (x -. average) ** 2.0) data
6   |> List.fold_left (+.) 0.0
7 let variance = sqrt variance_squared
```

Powyższy kod liczy średnią oraz wariancję pewnego zestawu danych, wczytanego przez moduł `DataReader`. Do zrobienia tego wykorzystuje funkcje z biblioteki standardowej do manipulowania listami. Co więcej, znaczna część z tych metod dostaje wszystkie oczekiwane argumenty. I nie trudno się dziwić. W tym wypadku ma żadnej korzyści z częściowego aplikowania takich funkcji.

Wykorzystamy ten nawyk programistów. Przedstawimy statyczną analizę opartą na systemie typów i efektów, podobną do kilku opisanych już w literaturze [8, 6, 7]. Przedstawimy również optymalizację, które pozwolą ograniczyć liczbę tworzonych domknięć. Co więcej, w przeciwieństwie do maszyny ZINC analiza będzie w pełni statyczna, co pozwoli na odciążenie runtime'u.

# Rozdział 2.

# Analiza

Zaczniemy od wspomnianej analizy. W tym rozdziale skupimy się na teoretycznej części problemu. Zaczniemy od języka dopuszczającego currying i skończymy na języku z funkcjami wieloargumentowymi. Opiszemy interesujące nas języki, zasady redukcji tworzące semantykę tych języków i systemy typów. Robiąc to postawimy sobie za cel zachowanie semantyki tych programów. Zatem przedstawimy też sposób tłumaczenia pomiędzy tymi językami.

## 2.1. Składnia języków

Zaczynając od definiowania języków. Wszystkie rozpatrywane języki będą współdzielić składnię. Zaczniemy więc od zdefiniowania jej.

$$
\begin{array}{rl}
\text{Zmienne} & x, y, z, \ldots \\
\text{Wyrażenia} & e ::= v \mid e\ \vec{e} \\
\text{Wartości} & v ::= x \mid \texttt{fun}\ \vec{x}.e \mid \texttt{fix}\ f\ \vec{x}.e \mid \texttt{write}_{\#n}
\end{array}
$$

Rachunek będzie się składał z kilku klasycznych części składniowych, czyli zmiennych, abstrakcji i aplikacji. Co więcej, w celu bliższego przyjrzenia się szczegółom analizy, rozszerzymy język o rekurencyjną abstrakcję oraz wartość $\texttt{write}_{\#n}$. Służy ona do wprowadzenia efektów ubocznych do języka. Będziemy ją etykietować indeksem $\#n$ (z unikatowym $n$ przy każdym wystąpieniu). Możemy zatem o niej myśleć jak o wbudowanej funkcji przyjmującej wartość `unit`, zwracającej wartość `unit` i powodującej powstanie unikatowego efektu ubocznego za każdym wywołaniem.

Dodatkowo, o ile konstrukcja `let` nie jest częścią żadnego z poniższych rachunków, będzie ona używana w przykładach kodu dla czytelności. W kontekście semantyki

i typowania, można o niej myśleć jak o połączeniu abstrakcji i aplikacji.

$$\texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \cong\ (\texttt{fun}\ x.e_2)\ e_1$$

## 2.2. Rachunek początkowy

Mając już składnię języka, zaczniemy od opisania języka początkowego, poprzez podanie opisującego go systemu typów oraz semantyki operacyjnej.

$$\begin{aligned} \text{Czystość} \quad & \varepsilon ::= i \mid p \\ \text{Typy} \quad & \tau ::= \texttt{unit} \mid \tau \to_\varepsilon \tau \end{aligned}$$

Wprowadzimy prosty system typów składający się z typu `unit` oraz strzałki. Jest to klasyczny sposób typowania funkcji w językach, w których dopuszczamy użycie curringu. Strzałkę będziemy parametryzować jej czystością. Przyjmiemy konwencję w której $p$ oznacza, że strzałka jest czysta, podczas gdy $i$ oznacza nie czystą strzałkę. O ile w wyjściowym języku nie jest wymagana taka parametryzacja, warto zacząć tutaj, ponieważ wcześniejsza transformacja nie jest tak ciekawa. Nie mniej, w późniejszym rozdziale pokażemy algorytm do efektywne znalezienie czystości strzałek.

$$\frac{}{\Gamma \vdash () : \texttt{unit}/p} \qquad \frac{}{\Gamma \vdash \texttt{write}_{\#n} : \texttt{unit} \to_i \texttt{unit}/p} \qquad \frac{(x:\tau) \in \Gamma}{\Gamma \vdash x : \tau/p}$$

$$\frac{\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau'/\varepsilon}{\Gamma \vdash \texttt{fun}\ x_1, \ldots, x_n.\, e : \tau_1 \to_p \cdots \to_p \tau_n \to_\varepsilon \tau'/p}$$

$$\frac{\Gamma, f : \tau_1 \to_p \cdots \to_p \tau_n \to_i \tau', x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau'/\varepsilon}{\Gamma \vdash \texttt{fix}\ x_1, \ldots, x_n . e : \tau_1 \to_p \cdots \to_p \tau_n \to_i \tau'/p}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau_1 \to_{\varepsilon_1} \cdots \to_{\varepsilon_{n-1}} \tau_n \to_{\varepsilon_n} \tau'/\varepsilon \\ \Gamma \vdash e_j : \tau_j/\varepsilon'_j \quad \text{dla } j = 1..n \\ \varepsilon' = \varepsilon \sqcup \bigsqcup_{0<j\le n} (\varepsilon'_j \sqcup \varepsilon_j)\end{array}}{\Gamma \vdash e\ e_1 \ldots e_n : \tau'/\varepsilon'} \qquad \frac{\Gamma \vdash e : \tau'/\varepsilon' \qquad \tau' <: \tau \qquad \varepsilon' <: \varepsilon}{\Gamma \vdash e : \tau/\varepsilon}$$

Rysunek 2.1: Zasady typowania

Zasady typowania (Rysunek 2.1) być może nie są klasyczne, nie mniej takie jak się można spodziewać spodziewać od rachunku lambda. Tak więc abstrakcje

zwykłe i rekurencyjne przyjmują `n` argumentów, wtedy ich typ to szereg co najmniej `n` strzałek. Aplikacja wymaga od typu aplikowanego wyrażenia bycia szeregiem strzałek o odpowiedniej długości. Jedna rzecz, o której wypada wspomnieć, abstrakcja stworzona konstrukcją `fix` będzie zawsze nieczysta. Jest to konieczne, ponieważ rekurencja w naszym języku wprowadza możliwość powstania zapętlenia programu, co z kolei może spowodować, że niektóre efekty pojawią się wielokrotnie, a inne wcale. Wprowadzimy też relację podtypowania i podefektowania, która tradycyjnie będzie zwrotna i przechodnia, a jej jedyną nietrywialną zasadą podtypowania będzie brudzenie strzałki.

$$\frac{}{\tau \ <: \ \tau} \qquad \frac{\tau_1 \ <: \ \tau_2 \qquad \tau_2 \ <: \ \tau_3}{\tau_1 \ <: \ \tau_3} \qquad \frac{\tau_1' \ <: \ \tau_1 \qquad \tau_2 \ <: \ \tau_2' \qquad \varepsilon \ <: \ \varepsilon'}{\tau_1 \to_{\varepsilon} \tau_2 \ <: \ \tau_1' \to_{\varepsilon'} \tau_2'}$$

$$\frac{}{\varepsilon \ <: \ \varepsilon} \qquad \frac{}{p \ <: \ i}$$

Wspominaliśmy już kilka razy o niezmienianiu semantyki programu, warto więc ją zdefiniować. Rozpatrując język z efektami ubocznymi, będzie nas interesować ich kolejność. Niech $\sigma$ będzie metazmienną reprezentującą szereg obserwowalnych efektów. Kolejne efekty będą generowane przez wywołania wartość $\texttt{write}_{\#n}$.

Definiując semantykę przeskoczymy nad wieloma pojęciami opisywanymi już w literaturze[1], na przykład przesłanianiem zmiennych i rozwiążemy ten problem założeniem, że każda zmienna ma unikatową nazwę.

$$\frac{\langle e_1, \sigma\rangle \to \langle e_1', \sigma'\rangle}{\langle e_1 \ e_2 \dots e_n, \sigma\rangle \to \langle e_1' \ e_2 \dots e_n, \sigma'\rangle} \qquad \frac{\langle e_2, \sigma\rangle \to \langle e_2', \sigma'\rangle}{\langle v_1 \ e_2 \dots e_n, \sigma\rangle \to \langle v_1 \ e_2' \dots e_n, \sigma'\rangle}$$

$$\frac{}{\langle \texttt{write}_{\#n} \ (), \sigma\rangle \rightharpoonup \langle (), \{\#n\} \cdot \sigma\rangle}$$

$$\frac{}{\langle (\texttt{fun} \ x_1 \ x_2 \dots x_n.e)v \ e \dots, \sigma\rangle \rightharpoonup \langle (\texttt{fun} \ x_2 \dots x_n.e\{v/x_1\}) \ e \dots, \sigma\rangle}$$

$$\frac{v' = \texttt{fix} \ g \ \vec{y}.\, e\{g/f, \overrightarrow{y/x}\} \qquad \vec{y}, g \text{ - fresh}}{\langle (\texttt{fix} \ f \ x_1 \ x_2 \dots x_n.e)v \ e \dots, \sigma\rangle \rightharpoonup \langle (\texttt{fun} \ x_2 \dots x_n.e\{v'/f, v/x\}) \ e \dots, \sigma\rangle}$$

Rysunek 2.2: Semantyka pierwotnego języka

Redukując ten język skupimy się na zasadach redukcji zorientowanych wokół aplikacji (Rysunek 2.2). Ewaluacja aplikacji będzie się odbywała od lewej do prawej i aplikować będziemy po jednym argumencie na raz. Warto również wspomnieć, że aplikowanie wartości $\texttt{write}_{\#n}$ jest jedynym sposobem wpłynięcia na ciąg efektów ubocznych.

## 2.3. Język wieloargumentowy

Teraz wprowadzimy język wieloargumentowy. Będziemy mieli kilka wymagań. Pierwsze, żeby wszystkie funkcje wieloargumentowe otrzymywały od razu tyle argumentów, ile oczekują. W ten sposób będziemy symulować ograniczenie alokowania domknięć. Drugim będzie zachowanie kolejności wykonywania efektów ubocznych. Te oczekiwania można opisać definiując semantykę redukcyjną (Rysunek 2.3).

$$\frac{\langle e, \sigma\rangle \to \langle e', \sigma'\rangle}{\langle e\ \vec{e}, \sigma\rangle \to \langle e'\ \vec{e}, \sigma'\rangle} \qquad \frac{|\vec{x}| = |\vec{v}|}{\langle (\texttt{fun}\ \vec{x}.e)\vec{v}, \sigma\rangle \rightharpoonup \langle e\{\overrightarrow{v/x}\}, \sigma\rangle}$$

$$\frac{\langle e_j, \sigma\rangle \to \langle e'_j, \sigma'\rangle}{\langle v\ \ v_1 \dots v_{j-1} e_j e_{j+1} \dots e_n, \sigma\rangle \to \langle v\ \ v_1 \dots v_{j-1} e'_j e_{j+1} \dots e_n, \sigma'\rangle}$$

$$\frac{|\vec{x}| = |\vec{v}| \qquad v' = \texttt{fix}\ g\ \vec{y}.\, e\{g/f, \overrightarrow{y/x}\} \qquad \vec{y}, g \text{ - fresh}}{\langle (\texttt{fix}\ f\ \vec{x}.e)\vec{v}, \sigma\rangle \rightharpoonup \langle e\{v'/f, \overrightarrow{v/x}\}, \sigma\rangle}$$

Rysunek 2.3: Oczekiwana semantyka języka wieloargumentowego

Od teraz do przeprowadzenia redukcji abstrakcji, będziemy wymagać podania dokładnej liczby argumentów. Podobnie z abstrakcją rekurencyjną. Zachowujemy również kolejność ewaluacji od lewej do prawej w wypadku aplikacji, jedynie rozszerzamy tę zasadę by obsłużyć wieloargumentowość.

Żeby zapewnić spełnienie wspomnianych wymagań, wprowadzimy drobne zmiany do systemu typów (Rysunek 2.4). Strzałki będą od teraz parametryzowane ich efektem oraz zwinięciem, w przeciwieństwie do tylko efektu. Jest to jeden ze sposobów na reprezentację faktu, że funkcja przyjmuje wiele arguemntów. Na pierwszy rzut oka może się wydawać łatwiejsze, żeby przekazywać funkcji listę argumentów. I końcowo tak będziemy myśleć o zwiniętych strzałkach, jednak przy dokonywaniu analizy myślenie o zwiniętych strzałkach, będzie bardziej pomocne. Pozwoli to nam na dokładniejsze porównywanie typów oraz da nam możliwość rozróżnienia zwiniętych strzałek czystych od nie czystych. Wybór reprezentacji używającej przecinków nie jawnie pozwoli na napisanie typu strzałki, w którym każda strzałka jest zwinięta, na przykład $\texttt{unit} \to_{\langle ,,p\rangle} \texttt{unit} \to_{\langle ,,p\rangle} \texttt{unit}$. W dosyć oczywisty sposób taki typ nie ma sensu, więc nałożymy wymaganie na zwinięte strzałki. Tak więc w typie $\tau_1 \to_{\langle f,\varepsilon\rangle} \tau_2$, $f$ może być zwinięte, jedynie gdy $\tau_2$ jest strzałka. Wprowadzimy też lukier syntaktyczny. Gdziekolwiek napiszemy $\tau_1 \to_\varepsilon \tau_2$, będziemy mieli na myśli rozwiniętą strzałkę, to znaczy $\tau_1 \to_{\langle \varepsilon,\to\rangle} \tau_2$, z kolei $\tau_1,^\varepsilon \tau_2$, będzie oznaczało zwiniętą strzałkę, czyli $\tau_1 \to_{\langle \varepsilon,,\rangle} \tau_2$.

Zmienią się również zasady podtypowania. Zamiast tylko efektem, w nowym języku strzałki są również parametryzowane ich stanem zwinięcia. Zamiast rozpisy-

$$
\begin{array}{lll}
\text{Czystość} & \varepsilon ::= & i \mid p \\
\text{Zwijanie} & f ::= & \rightarrow \mid , \\
 & \alpha := & \langle \varepsilon, f \rangle \\
 & & \\
\text{Typy} & \tau ::= & \texttt{unit} \mid \tau \rightarrow_{\alpha} \tau
\end{array}
$$

Rysunek 2.4: Gramatyka typów dla wieloargumentowego rachunku lambda

wania wszystkich zasad relacji, skupimy się nad zależnościami między parametrami strzałek (Rysunek 2.5).

$$
\begin{array}{ccc}
\langle \rightarrow, p \rangle & \begin{array}{c} <: \\ :> \end{array} & \langle , , p \rangle \\
\wedge\!\!\vdots & & \wedge\!\!\vdots \\
\langle \rightarrow, i \rangle & <: & \langle , , i \rangle
\end{array}
$$

Rysunek 2.5: Podtypowanie parametrów strzałek

Intuicja za tymi zasadami jest następująca. Czyste funkcje można zwijać i rozwijać, jako że nie wydzielają efektów. Na poziomie semantyki wartości typu strzałki zwiniętej i rozwiniętej są równoważne. Czyste funkcje można zastosować w każdym miejscu, gdzie spodziewamy się nieczystej.

Na koniec rozmawiając o strzałkach nie czystych, podtypowanie zachodzi tylko w jedną stronę. Rozwiniętą nieczystą strzałkę można zastosować w miejscu, gdzie oczekujemy zwiniętej. Dodanie zasady w drugą stronę miałoby bardzo duże i niebezpieczne konsekwencje na semantyce programów. Ze względu na to jak będziemy rozwiązywać podtypowanie, które opiszemy w następnej sekcji, dodanie zasady w drugą stronę dopuściło by możliwość oddalenia przez analizę efektu, czego staramy się uniknąć.

Na przykład poniższy kod, byłby narażony na taki problem.

```
1  let g = fun _.
2    let _ = write () in
3    fun _. ()
4  in
5  let f1 = fun g1.
6    g1 ()
7  in
8  let f2 = fun g2.
9    let _ = f1 g2 in
10   g2 () ()
11 in
12 f2 g
```

W powyższym fragmencie kodu, `f2` ma typ $(\tau_1, \tau_2 \to \tau_3) \to \tau_3$, podczas gdy `g` ma typ $\tau_1 \to \tau_2 \to \tau_3$. Bez tłumaczenia do języka wieloargumentowego, skoro $g$ powoduje efekt, po otrzymaniu jednego argumentu, spodziewamy się zobaczyć ten efekt dwa razy, jednak dopuszczając podtypowanie w obie strony, zobaczymy ten efekt tylko raz. Dzieje się tak ponieważ przekazując `g` do `f2` wstawiana jest koercja, która musi być rozwiązana (powód i sposób jak to się dzieje zostanie opisany w następnej sekcji).

Co ciekawe problem pojawia się jedynie, kiedy obecne są zasady w obie strony. Podtypowanie z zasadą w jedną stronę nie będzie miało negatywnych konsekwencji, niezależnie w którą stronę będzie skierowane. Wybieramy skierowanie w tę stronę, ponieważ ma pozytywny wpływ na wyniki optymalizacji.

$$\frac{\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau'/\varepsilon}{\Gamma \vdash \texttt{fun}\ x_1, \ldots, x_n.e : \tau_1, {}^p \ldots, {}^p \tau_n \to_\varepsilon \tau'/p}$$

$$\frac{\Gamma, f : \tau_1^p, \ldots, {}^p \tau_n \to_i \tau', x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau'/\varepsilon}{\Gamma \vdash \texttt{fix}\ x_1, \ldots, x_n.e : \tau_1, {}^p \ldots, {}^p \tau_n \to_i \tau'/p}$$

$$\frac{\Gamma \vdash e : \tau_1, {}^p \ldots, {}^p \tau_n \to_{\varepsilon'} \tau'/\varepsilon \qquad \Gamma \vdash e_j : \tau_j/\varepsilon_j \quad \text{dla } j = 1..n \qquad \varepsilon'' = \varepsilon \sqcup \varepsilon' \sqcup \bigsqcup_{0<j\leq n} \varepsilon_j}{\Gamma \vdash e\ e_1 \ldots e_n : \tau'/\varepsilon''}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau_1, {}^p \ldots, {}^p \tau_{k-1}, {}^i \tau_k, {}^{\varepsilon_{k+1}}, \ldots, {}^{\varepsilon_{n-1}} \tau_n \to_{\varepsilon_n} \tau'/\varepsilon \\ \Gamma \vdash e_j : \tau_j/\varepsilon'_j \quad \text{dla } j = 1..k \\ \Gamma \vdash e_l : \tau_l/p \quad \text{dla } l = k+1..n \\ \varepsilon' = \varepsilon \sqcup \bigsqcup_{0<j\leq k} \varepsilon'_j \sqcup \bigsqcup_{k<j\leq n} \varepsilon_j\end{array}}{\Gamma \vdash e\ e_1 \ldots e_n : \tau'/\varepsilon'}$$

Rysunek 2.6: Wybrane zasady typowania w języku wieloargumentowym

W końcu możemy przejść do zasad typowania, pokazane na rysunku 2.6. To one będą dyktowały ograniczenia w tłumaczeniu i zapewnią utrzymanie semantyki programów. Relacja typowania dla tego języka zmienia reguły dla konstrukcji abstrakcji oraz aplikacji. W wypadku abstrakcji liczba argumentów będzie dyktowana późniejszymi użyciami tej wartości. Dużo ciekawiej jednak dzieje się przy aplikacji. W wypadku tej konstrukcji mamy do czynienia z dwiema zasadami. Pierwsza jest stosowana, kiedy mamy do czynienia z czystą funkcją. Aplikowanie argumentów nie będzie miało efektów ubocznych, więc możemy bezpiecznie przekazać ich, ile tylko się da.

Druga zasada jest potrzebna, kiedy przy aplikowaniu funkcji nieczystych. Listę operandów takiej aplikacji dzielimy na trzy części. Pierwsza część to argumenty, których przekazanie nie spowoduje efektów. Tak jak w poprzedniej zasadzie, nie interesuje nas czy redukcja tych wyrażeń do wartości spowoduje jakieś efekty. Druga część składa się z jednego argumentu. Jest to argument, po przekazaniu którego przekraczamy nieczystą część strzałki. Tutaj efekty wynikające z redukcji do wartości również nas nie interesują. Ostatnią częścią jest lista argumentów, których redukcja nie może produkować efektów. Jeśli pomyślimy o tym w kontekście języka ze strzałkami jednoargumentowymi, to jest moment, w którym efekt pod zwinięta strzałką z drugiej części powinien się wydarzyć, ale jeśli obliczenie wszystkich następnych operandów nie wywołuje efektu, możemy je wyliczyć za darmo, przekazać w tej aplikacji i w ten sposób zaoszczędzić na tworzeniu domknięcia. W tej części nie interesuje nas efekt aplikowania tych argumentów, ponieważ i tak wydarzą się w odpowiedniej kolejności. Warto wspomnieć, zarówno pierwsza część może być pusta (aplikacja pierwszego argumentu powoduje przekroczenie nieczystego przecinku), jak i trzecia (może nie być więcej argumentów po nie czystym przecinku, albo wyliczenie następnego operandu wywołuje efekt). Zauważmy również, że w wypadku braku drugiej i trzeciej części, otrzymujemy poprzednią zasadę.

## 2.4. Rozwiązywanie podtypowania

Jeden z celów, który ustanowiliśmy jest zapewnienie, że aplikacja każdej abstrakcji otrzymuje dokładnie tylu argumentów, ile oczekuje. Jednak przez podtypowanie, dopuściliśmy, żeby funkcje wyższego rzędu otrzymywały argumenty będące strzałkami o nie koniecznie tych samych punktach złożenia, jakich oczekuje wspomniana funkcja wyższego rzędu.

Żeby uniknąć tego problemu zdefiniujmy tłumaczenie na język, który zapożycza składnię z poprzednich, semantykę z drugiego języka, ale w systemie typów podtypowanie znowu służy tylko do podefektowania (pozbywamy się zasad podtypowania z rysunku 2.5). Z tego powodu tłumaczenie ograniczy się do rozwiązywania koercji podtypowania. Dokładne opisanie matematyczne tego tłumaczenia w sposób czytelny jest trudne, dlatego tutaj ograniczymy się do przejścia przez kilka przykładów tłumacząc kluczowe fragmenty. Dokładniejszy opis można znaleźć w implementacji[1] w postaci funkcji `add_coercion`.

[1] Znajduje się ona na publicznym repozytorium pod adresem `https://github.com/Foxinio/type-inference`.

Rozważmy następujący kod:

```
let f = fun g.
  let g1 = g () () in
  g1 ()
in
let g = fun x1.
  let _ = () in
  fun x2 x3. ()
in
f g
```

Funkcja $f$ będzie miała typ $(\texttt{unit},^p \texttt{unit} \to_p \texttt{unit} \to_p \tau) \to_p \tau$, Podczas, gdy $g$ będzie miała typ $\texttt{unit} \to_p \texttt{unit},^p \texttt{unit} \to_p \tau'$. Będziemy zatem musieli zastosować koercję, i w tym wypadku otrzymamy następujący kod.

```
...
f (fun x1 x2. fun x3. (g x1) x2 x3)
```

Ogólne zasady budowania koercji są trudne do opisania matematycznie w czytelny sposób, ale dosyć intuicyjne. Zaczniemy od podzielenia ciągu strzałek tak, żeby wszystkie typy na negatywnych pozycjach znalazły się na liście list typów. Na przykład typ $\tau_1, \tau_2, \tau_3 \to \tau_4, \tau_5 \to \tau_6 \to \tau$, przy założeniu że $\tau$ nie jest strzałką, podzielimy na $[[\tau_1, \tau_2, \tau_3], [\tau_4, \tau_5], [\tau_6]]$, więc wewnętrzne listy będą reprezentować sekcje złożenia strzałek. Zrobimy tak zarówno dla typu oczekiwanego jak i typu podanego. Następnie wygenerujemy listy świeżych zmiennych, żeby kształtem były identyczne do sekcji argumentów typu oczekiwanego i zbudujemy serię abstrakcji `fun` tak, że kolejne sekcje będą będą wiązały wspomniane argumenty. Tak więc z powyższego typu otrzymamy $[[x_1, x_2, x_3], [x_4, x_5][x_6]]$, i ułożymy z nich serię abstrakcji $\texttt{fun}\ x_1\ x_2\ x_3.\texttt{fun}\ x_4\ x_5.\texttt{fun}\ x_6.e'$. Teraz pozostaje zaaplikowanie związanych zmiennych. Zrobimy to budując $e'$ z przykładu w poprzednim zdaniu. Tym razem będzie nas interesować lista sekcji powstałą z typu podanego. Dla przykładu przyjmijmy że ten typ to $\tau_1, \tau_2 \to \tau_3 \to \tau_4, \tau_5, \tau_6 \to \tau$, więc otrzymamy listę $[[\tau_1, \tau_2], [\tau_3], [\tau_4, \tau_5, \tau_6]]$, i zmienne ułożymy w następujący sposób $[[x_1, x_2], [x_3], [x_4, x_5, x_6]]$. Tym razem będziemy je składać za pomocą aplikacji, tak więc zakładając że oryginalnym wyrażenie, dla którego budujemy koercje, jest $e''$, otrzymamy $e' \cong (((e''\ x_1)\ x_2\ x_3)x_4\ x_5\ x_6)$. Podsumowując dostaniemy zatem:

$$e'' \rightsquigarrow \texttt{fun}\ x_1\ x_2\ x_3.\texttt{fun}\ x_4\ x_5.\texttt{fun}\ x_6.\ (((e''\ x_1)\ x_2\ x_3)x_4\ x_5\ x_6).$$

Są dwa szczególne przypadki, na które warto zwrócić uwagę. W powyższym przykładzie dobraliśmy typ tak, żeby typy zgadzały się na argumentach wyrażenia podlegającego koercji, ale równie dobrze mogliśmy mieć typ oczekiwany $\tau_1, \tau_2, \tau_3 \to \tau_4, \tau_5 \to \tau_6 \to \tau$ i przekazany $\tau_1, \tau_2, \tau_3' \to \tau_4, \tau_5 \to \tau_6 \to \tau$, gdzie $\tau_3' \neq \tau_3$. Z tego powodu jakąkolwiek funkcję będziemy mieli do budowania koercji, będziemy ją wywoływać także w głąb na negatywnych pozycjach typów.

Drugim problemem, o którym należy wspomnieć, jest nieczystość strzałek. Na przykład strzałka po $\tau_1$ albo $\tau_2$ w $\tau_1, \tau_2 \to_i \tau_3 \to_i \tau_4, \tau_5, \tau_6 \to \tau$ będzie nieczysta. Żeby nie zmienić kolejności efektów ubocznych po przejściu przez nieczystą strzałkę, gdy natkniemy się na rozwiniętą strzałkę, to znaczy, gdy skończy się sekcja argumentów typu podanego, będziemy musieli wywołać wartość, związać ją w konstrukcji `let` i dalej używać wartości pod zmienną w związaną w tym wyrażeniu `let`. Przypominając więc koercję $e''$, jeśli to wyrażenie miało typ $\tau_1, \tau_2 \to_i \tau_3 \to_i \tau_4, \tau_5, \tau_6 \to \tau$, zbudowana koercja miała by następującą formę.

$$\begin{aligned}&\texttt{fun}\ x_1\ x_2\ x_3.\\ &\quad \texttt{let}\ x' = ((e''\ x_1)\ x_2\ x_3)\ in\\ &\quad \texttt{fun}\ x_4\ x_5\ x_6.(x'\ x_4\ x_5\ x_6)\end{aligned}$$

W ten sposób każdy efekt wydarzy się dokładnie wtedy, kiedy się tego spodziewamy.

# Rozdział 3.

# Algorytm

Implementując wspomnianą analizę przyda się mieć algorytm pozwalający na efektywne rozwiązanie więzów podtypowania pomiędzy parametrami strzałki. W tym rozdziale przedstawimy taki algorytm pozwalający na określenie każdego parametru po jednym przejściu przez abstrakcyjne drzewo składniowe (ang. AST).

## 3.1. Idea algorytmu

Jesteśmy postawieni przed problemem znalezienia najlepszego ustawienia parametrów strzałek. Skoro interesuje nas minimalizacja liczby stworzonych domknięć w trakcie wykonywania, poprzez najlepsze ustawienie parametrów będziemy rozumieć takie, w którym jak najwięcej strzałek jest zwiniętych. Liczba miejsc, gdzie można pozostawić zwinięte strzałki, będzie ograniczona przez budowę drzewa AST, oraz przez więzy podtypowania wynikające z wykorzystania funkcji. Orientacyjnie algorytm będzie chodził po drzewie AST zbierając więzy podtypowania i rozwiązując je, gdy tylko będzie miał taką możliwość, ustalając wartości zmiennych unifikacyjnych.

Na takie działanie algorytmu pozwala specyfika problemu. Na początek krata, po której się poruszamy składa się tylko z dwóch elementów. Co więcej, nie mamy żadnych konstrukcji syntaktycznych ani zasad typowania, które narzucają jednoznacznie wymóg czystości strzałki (taką może się wydawać reguła typowania aplikacji, jednak każdą aplikację jesteśmy w stanie podzielić na serię kilku, tak żeby nie trzeba było zmieniać wartości parametrów strzałki). Jedyne wymagania jakie są nałożone, to aby strzałka na końcu `fix` oraz `write` były brudne. Pozwala nam to na bardzo leniwe ustawianie strzałek na brudne. Podobnie jest w kwestii zwijania. Tym razem mamy miejsca gdzie wymagamy by strzałka była rozwinięta, ale nie narzucamy nigdzie zwinięcia (znowu odpowiednia transformacja programu pozwoli spełnić zasady typowania).

Cała analiza będzie się składać z dwóch pomniejszych analiz/. Pierwsze ustali czystość strzałek. Drugie przejście ustali zwinięcie strzałek, korzystając z informacji

o ich czystości. Ze względu na niesymetryczny kształt diagramu podtypowania 2.5, zrobienie tego w ten sposób znacznie uprości problem, ponieważ ustalając zwinięcie strzałek pomoże wiedza czy znajdujemy się na dole czy na górze diagramu.

## 3.2. Wymagania

Żeby algorytm mógł działać, będziemy wymagać kilku rzeczy od reprezentacji języka w pamięci. Przede wszystkim, skoro algorytm analizuje i optymalizuje typy, będziemy wymagać, żeby te typy były w jakiś sposób przetrzymywane na drzewie AST. Jako że analiza dotyczy konkretnie typów strzałek, w dołączonej implementacji trzymamy typy całej abstrakcji w strukturze reprezentującej abstrakcję w pamięci.

Drugim wymaganiem, czego można się spodziewać po pierwszym, jest znajomość kształtów typów wyrażeń, to znaczy że albo musimy wymagać anotacji typów argumentów, albo przed opisanym algorytmem, wykonać inferencję typów (na przykład przy pomocy algorytmu $\mathscr{F}$ [4], jak to jest zrobione implementacji).

## 3.3. Struktura danych algorytmu

Algorytm będzie wykorzystywał bardzo specyficzny typ danych pełniący rolę zmiennej unifikacyjnej reprezentującej wartość parametru strzałki. Te zmienne będą utrzymywały wygenerowane więzy i w niejawny oraz leniwy sposób same siebie nawzajem ustalały.

```
1 type uvar = uvar_value ref
2 and uvar_value =
3   | Impure
4   | Unknown of uvar list
```

Jego znaczenie jest proste. Jeśli po wykonaniu analizy `!uv` ma wartość `Impure`, wiemy, że reprezentuje on nieczysty efekt. A w przeciwnym razie uznamy go za czysty. Możemy tak zrobić, ponieważ w naszym języku nigdzie nie nakładamy więzu, który każe strzałce być czystą. W algorytmie albo ustalimy zmienną unifikacyjną na nieczystą, albo że pomiędzy dwoma zmiennymi zachodzi podtypowanie `uv1` <: `uv2`. Do obsłużenia tych sytuacji posłużą nam funkcje `set_impure` oraz `unify_uvar`, których kod znajduje się na rysunku 3.1. W tych dwóch funkcjach mieści się większość złożoności całego algorytmu. Jeśli zmienna `x` ma postać `Unknown lst`, `lst` przechowuje wszystkie zmienne, dla których `x` jest dolną granicą. Wtedy, jeśli `x` okaże się nieczysta, wszystkie zmienne na liście automatycznie również muszą być być nieczyste. Ta struktura będzie przetrzymywana przy strzałkach, tak jak w poprzednim rozdziale trzymaliśmy ich parametry.

```
1  let rec set_impure (x : uvar) =
2    match !x with
3    | Unknown lst ->
4      x := Impure;
5      List.iter set_impure lst
6    | Impure -> ()
7
8  let unify_uvar (x : uvar) y =
9    match !x with
10   | Unknown lst ->
11     x := Unknown (y :: lst)
12   | Impure ->
13     set_impure y
```

Rysunek 3.1: Funkcje manipulujące zmiennymi unifikacyjnymi

## 3.4. Analiza efektów

Jedyne co pozostaje to funkcja chodząca po drzewie AST — `fill_effects`. Będzie przyjmować 3 argumenty środowisko inferencji, analizowane wrażenie, oraz zmienną zmienną unifikacyjną reprezentującą spodziewany efekt analizowanego wyrażenia. Funkcja ta ma za zadanie wygenerować więzy podtypowania. Będzie to robić na dwa sposoby. Pierwszym z nich jest inferowanie typu i porównywanie go (używając funkcji `unify_subtype`) z typami trzymanymi na drzewie AST. Drugi sposób na jaki będą generowane to używając otrzymywaną w argumencie zmienną unifikacyjną. Kiedykolwiek natkniemy się na aplikację, w której przekazujemy strzałce argument uwalniając jakikolwiek efekt ona kryje, powiemy że spodziewany efekt analizowanego wyrażenia (zmienna `uv`) musi być większa od tej uwalnianej.

Na przykład w 7. wierszu zamiast otrzymanej w argumencie `uv`, do analizowania ciała funkcji przekazujemy wyciągniętą z typu abstrakcji $\texttt{uv}'$. Ta zmienna jest wykorzystywana przede wszystkim w funkcji `fill_effects_in_app`, dokładniej w 37. wierszu, gdzie aplikując wartość do strzałki uwalniamy jakikolwiek efekt się pod nią kryje, zatem tam wykonamy `unify_uvar`. Drugim szczegółem jest wykorzystanie funkcji `unify_subtype`. Jej implementacja jest głównie strukturalna, tak więc została pominięta, warto jedynie wiedzieć, że porównuje dwa typy i natrafiając na strzałki wykonuje `unify_uvar` na zmiennych unifikacyjnych do nich dołączonych. Ostatnim szczegółem jest wspomniane traktowanie `EFix` automatycznie jako nieczystego, zatem w 15. wierszu zmienna unifikacyjna przechowująca efekt tej konstrukcji jest brudzona wywołaniem `set_impure`. Podobnie wywołujemy tę funkcję natrafiając na nieczystą funkcję wbudowaną, na przykład opisaną w poprzednim rozdziale `write`.

```
let rec fill_effects env e uv =
  match e with

  ...

  | EFn (args, body, tp) ->
    let tp_expected, env, uv' = Env.extend_var env args tp in
    let tp_actual = fill_effects env body uv' in
    unify_subtype tp_actual tp_expected;
    tp

  | EFix (f, args, body, tp) ->
    let env = Env.add_var env f tp in
    let tres, env, uv' = Env.extend_var env args tp in
    set_impure uv';
    let tp_actual = fill_effects env body uv' in
    unify_subtype tp_actual tp_expected;
    tp

  | EApp (e1, es) ->
    let tp1 = fill_effects env e1 uv in
    fill_effects_in_app env es tp1 uv

  | EExtern(name, tp, eff) ->
    if eff = Impure then set_impure uv;
    tp

  ...

and fill_effects_in_app env args tp1 uv =
  let rec inner args tp1 =
    match args, tp1 with
    | [], tres -> tres
    | e :: args, TArrow(uv', targ, tres) ->
      let targ' = fill_effects env e uv in
      unify_subtypes targ' targ;
      unify_uvar uv' uv;
      inner args tres uv
  in inner args tp1
```

Rysunek 3.2: Funkcja chodząca po AST

## 3.5. Analiza składania

Analiza składania będzie się odbywać podobnie do analizy efektów, a drobne różnice opiszemy poniżej.

Przede wszystkim w liście `lst`, gdy `!uv` ma wartość `Unknown lst`, będziemy przechowywać zmienne mniejsze w sensie więzów. Zatem w tym wypadku funkcja `unify_uvar` będzie wyglądała następująco.

```
1 let unify_uvar x y =
2   match !y with
3   | Unknown lst ->
4     y := Unknown (x :: lst)
5   | Unfolded ->
6     set_unfolded x
```

Druga różnica znajduje się w odpowiedniku funkcji `set_impure` - `set_unfolded`. Ze względu na równoważność (pod względem podtypowania) funkcji zwiniętych i rozwiniętych, kiedy są czyste, będziemy iterować się po liście jedynie, kiedy będziemy mieli pewność, że strzałka nie jest czysta.

```
1 let rec set_unfolded (x : uvar) =
2   match !x with
3   | Unknown lst when is_impure x ->
4     x := Unfolded;
5     List.iter set_unfolded lst
6   | Unknown _ ->
7     x := Unfolded
8   | Unfolded -> ()
```

Kolejna różnica znajduje się w funkcji chodzącej po AST. Nie będziemy już przekazywać zmiennych unifikacyjnych w głąb drzewa, bo nie śledzimy już uwalnianych efektów. Jedyne więzy będziemy zapisywać, gdy natkniemy się na strzałki unifikując typy funkcją `unify_subtype`. Również funkcję `set_unfolded` będziemy wykonywać zarówno w przypadku konstrukcji `EFix`, jak i `EFn` (na 3.2 była by wykonana na zmiennych unifikacyjnych zwracanych w 7. i 14. wierszu). Ostatnia uwaga jest taka, że kiedykolwiek natkniemy się na typ zapisany w drzewie AST, wykonamy na nim funkcję `unfold_ends`. Funkcja ta zapewni, że gdziekolwiek mamy do czynienia z typem $\tau \to_\alpha \tau'$, jeśli $\tau'$ nie jest strzałką, $\alpha$ zostanie ustawiona jako `Unfolded`.

# Rozdział 4.

# Implementacja

Do pracy dołączona jest implementacja[1] eksperymentalnego języka, przeprowadzająca opisaną analizę. W tym rozdziale przyjrzymy się dostarczonej funkcjonalności, oraz przejdziemy przez kolejne etapy interpretera.

## 4.1. Funkcjonalność

**Inferencja typów** Język, pomimo bycia silnie typowanym, pozwala na zupełne ominięcie pisania adnotacji typowych. Jest to zasługa zastosowania algorytmu do odzyskiwania typów.

**Polimorfizm** Język jest wyposażony tak zwany let-polimorfizm. Umożliwia on pisanie generycznego kodu przy bardzo małym nakładzie pracy od implementatora języka. Generalizowanie zmiennych typowych jest ograniczone do value-restriction [3].

**Algebraiczne typy danych** Język jest wyposażony w algebraiczne typy danych, które jednocześnie pozwalają na definiowanie rekurencyjnych typów danych i parametryzowanie ich. Robi to konstrukcją `type` wyliczając konstruktory oraz ich typy i argumenty. Każdy konstruktor musi zamykać tylko jeden typ, co nie jest istotnym ograniczeniem, ponieważ język zawiera też pary i wartość unit. Naturalnie dostępna jest konstrukcja `match`.

**Typy i funkcje wbudowane** Język zawiera kilka typów wbudowanych, jak `int` czy `bool` oraz funkcje do manipulacji nimi. Są też dostępne funkcje wejścia/wyjścia, takie jak wypisywanie znaków (otrzymuje wartość typu `int` i wypisuje jej kod ASCII), liczb, a także ich wczytywanie. Dostarczona jest także funkcja `printType` pozwalająca na wypisanie typu danego wyrażenia po wykonaniu analizy.

**Rekurencja** Język pozwala na prostą formę rekurencji za pomogą operatora `fix` jak i lukru `let rec`.

[1] Znajduje się ona na publicznym repozytorium pod adresem `https://github.com/Foxinio/type-inference`.

## 4.2. Pipeline

Interpretacja języka składa się z kilku etapów.

**Leksowanie i Parsowanie** Jak w każdym interpreterze zaczyna się od tych dwóch kroków, ale nie dzieje się tam nic nadzwyczajnego.

**Dołączenie funkcji wbudowanych** Na tym poziomie dodane jest preludium wartości wbudowanych. Wszystkie operacje artmetyczne i logiczne. Operatory takie jak `or` lub `and` są inlinowane, żeby móc wpłynąć na leniwą ewaluację.

**Tłumaczenie do pośredniego AST** Na tym etapie identyfikatory wprowadzone przez użytkownika zamieniane są na unikatowe identyfikatory generowane przez interpreter. Pozwala to ominąć późniejsze problemy z przesłanianiem. W języku znajdują się dwie przestrzenie nazw: wartości i typów.

**Inferencja** Na tym etapie następuje uzupełnianie typów oraz generalizacja schematów do ML polimorfizmu. Warty wspomnienia jest algorytm generalizacji. Odbywa się ona w bardzo efektywny sposób, przez użycie poziomów unifikacji. Po krótce działają one tak, że tworząc zmienną unifikacyjną przypisujemy jej obecny poziom unifikacji i przypisując zmiennej unifikacyjnej `x` jakiś typ, propagujemy jej poziom do zmiennych unifikacyjnych obecnych w tym typie, ale tylko obniżając poziom. Teraźniejszy poziom zwiększa się przy wejściu w wyrażenie po lewej stronie `let`, gdy to wyrażenie jest wartością. W ten sposób mamy gwarancję, że przy generalizacji natrafiając na zmienną unifikacyjną o poziomie generalizowanego `let`a mamy pewność, że została ona stworzona na tym poziomie i nie pochodzi ze środowiska, co pozwala ominąc kosztownego przeglądania środowiska. Na tym poziomie rozwiązywane są też tak zwane aliasy typu.

**Tłumaczenie do Systemu F** Następnym językiem pośrednim jest System F. Na tym języku dokonywana jest analiza i optymalizacja. Został wybrany, ponieważ na tym etapie znamy już kształt typów.

**Analizy i transformacje** Teraz dzieją się opisane w poprzednich rozdziałach analizy. Ze względu że analiza składania wymaga pełnej informacji o efektach, następuje jako druga. Po analizie efektów, następuje pierwsza transformacja kodu łącząca aplikacje tak gęsto, żeby spełniona była relacja typowania opisana w drugim rozdziale. Następnie po analizie składania rozwiązywane są koercje, więc eliminowane jest podtypowanie.

**Well typed check** Następnie zachodzi sprawdzenie typowania. Służy ono do umocnienia pewności w poprawność implementacji i pozwolił na odkrycie wielu błędów w trakcie implementowania. W jego trakcie odświeżane są zmienne typowe, sprawdzane równości typów i relacji typowania z rozdziału 2.

**Wymazywanie typów** Na tym etapie upraszczamy język eliminując konstrukcje zbędne w trakcie ewaluacji. Rozwiązywane są funkcje wbudowane, dostając

ciało do wykonania. Tłumaczone są też konstrukcje `match`, by zapobiec problemowi niepełnego dopasowania wzorca.

**Ewaluacja** Jest to ostatni etap i jego specjalną rolą jest liczenie liczby stworzonych domknięć w trakcie wykonywania. Wyniki są na koniec wypisywane na standardowe wyjście razem z wynikiem obliczenia programu.

## 4.3. Wyniki optymalizacji

W celu oceny powodzenia naszej optymalizacji będziemy mierzyć ilość stworzonych domknięć. Użyjemy tej liczby jako przybliżona miara wydajności programu, ponieważ różnice w czasach wykonywania programów z optymalizacją i bez nie są wystarczające, by umożliwić wyciągnięcie wniosków.

Po uruchomieniu programu na kilku testach, wyniki są różne, ale zawsze pozytywne. Na programach niekorzystających z wielu funkcji różnice były znikome, ale gdy tylko pojawiły się jakieś funkcje wieloargumentowe różnica jest znaczna.

```
1  type list 'a =
2    | Nil : unit
3    | Cons : 'a * list 'a
4
5  let fold =
6      fix fold f acc lst =>
7      match lst with
8      | Nil _ => acc
9      | Cons lst =>
10       let hd = fst lst
11       let tl = snd lst
12       let acc = f acc hd
13       in
14       fold f acc tl
15     end
16 let lst = (Cons (1, Cons (2, Cons (3, Cons (4, Cons (5, Nil ()))))))
17 in
18 fold add 0 lst
```

Rysunek 4.1: przykładowy program testujący funkcję `fold` na listach

```
1 $ ./a.out test/tests/013_folding_list.lm
2 15
3 Closure created 21 times
4 $ ./a.out --crude-analysis test/tests/013_folding_list.lm
5 15
6 Closure created 43 times
```

Rysunek 4.2: wyniki uruchomienia 4.1

Oto wyniki z kilku innych testowych programów. Wszystkie testy można znaleźć w repozytorium implementacji (nie wspomniane testy, sprawdzały inną funkcjonalność i nie wykazały różnic w wynikach).

| Nazwa programu | Bez optymalizacji | Z optymalizacją |
|---|---|---|
| test/tests/009_map_and_fold_list.lm | 90 | 47 |
| test/tests/010_externs.lm | 16 | 10 |
| test/tests/012_fun_composition.lm | 19 | 13 |
| test/tests/013_folding_list.lm | 43 | 21 |
| test/tests/016_fun_pairs_as_arguments.lm | 55 | 15 |
| test/tests/017_inefficiencies.lm | 30 | 29 |
| test/tests/018_inefficiencies2.lm | 123 | 28 |
| test/tests/019_inefficiencies3.lm | 123 | 29 |

Tabela 4.1: Porównanie liczby domknięć z oraz bez optymalizacji dla różnych programów

W trakcie testowania implementacji, udało się natknąć na programy, które po optymalizacji powodują stworzenie większej liczby domknięć niż przed. Są to jednak programy, które trzeba napisać z myślą o przeszkodzeniu analizie i nie natknie się na nie przypadkiem, z kolei straty na wydajności są znikome w porównaniu do zysków opisanych powyżej.

# Rozdział 5.

# Dalsze prace

## 5.1. Dokładność pomiarów

Do oceny jakości optymalizacji zdecydowaliśmy się na liczenie domknięć. Jest to jakieś sposób przybliżenia oceny wydajmości programów, jednak nie jest idealny. Nie każde alokacje są tak groźne jak inne. Być może za zyskami w zmniejszonej liczby alokacji, idą ukrytym kosztem mającym znacznie większy wpływ na wydajność programów. Dokładniejsze pomiary wymagają napisania kompilatora. Pozwoliłoby to na mierzenie czasu wykonywania kodu, nie tylko przybliżania wydajności programu przez liczenie tworzonych domknięć.

## 5.2. Dokładniejszy typechecking

Obecna implementacja algorytmu analizy działa jak inferencja typów i robi to bottom-up. Konsekwencją tego jest być może nie do końca efektywne ustawienie zasad podtypowania. Mogłby w tym pomóc dwukierunkowy algorytm.

## 5.3. Polimorfizm efektowy

Analiza znajdując efekt danej strzałki, przypina jej go na stałe. Jest to bardzo zgrubna aproksymacja. Funkcje wyższego rzędu będą oznaczone jako nieczyste, jeśli tylko chociaż raz zostały wywołane z nieczystym argumentem. Nie jest to krytyczny problem. W końcu analiza działa satysfakcjonująco nawet w obecnej formie. Może jednak isnieć inny sposób, wykorzystujący polimorfizm efektowy, pozwalający na lepszą optymalizację.

# Bibliografia


[1] Alonzo Church. The Calculi of Lambda-Conversion, volume 6 of Annals of Mathematics Studies. Princeton University Press, 1941.

[2] Haskell B. Curry. Grundlagen der kombinatorischen logik. American Journal of Mathematics, 52(3):509–536, 1930.

[3] Jacques Garrigue. Relaxing the value restriction. In Functional and Logic Programming. FLOPS 2004. Lecture Notes in Computer Science, volume 2998, pages 196–213. Springer, Berlin, Heidelberg, 2004.

[4] Robin Milner. A theory of type polymorphism in programming. Journal of computer and system sciences, 17(3):348–375, 1978.

[5] E Recherche, Et Automatique, Domaine Voluceau, Calcul Programmation, and Xavier Leroy. The zinc experiment: An economical implementation of the ml language. 07 1992.

[6] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. Journal of Functional Programming, 2(3):245–271, 1992.

[7] J.P. Talpin and P. Jouvelot. The type and effect discipline. Information and Computation, 111(2):245–296, 1994.

[8] Yan Mei Tang and Pierre Jouvelot. Separate abstract interpretation for control-flow analysis. In Masami Hagiya and John C. Mitchell, editors, Theoretical Aspects of Computer Software, pages 224–243, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.