

**Uniwersytet Warszawski**  
Wydział Matematyki, Informatyki i Mechaniki

**Michał Borkowski**

Nr albumu: 370727  
**Marian Dziubiak**

Nr albumu: 370784

**Jakub Bujak**

Nr albumu: 370737  
**Marek Puzyna**

Nr albumu: 371359

# Kompilacja NianioLanga do efektywnych struktur języka C

Praca licencjacka  
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem  
**mgr. Radosława Bartosiaka**  
Instytut Informatyki

Maj 2018

## **Oświadczenie kierującego pracą**

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data

Podpis kierującego pracą

## **Oświadczenie autora (autorów) pracy**

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data

Podpisy autorów pracy

## **Streszczenie**

W pracy opisujemy...

## **Słowa kluczowe**

kompilacja, języki programowania, analiza semantyczna, NianioLang, system typów

## **Dziedzina pracy (kody wg programu Socrates-Erasmus)**

11.3 Informatyka

## **Klasyfikacja tematyczna**

D. Software

D.3. Programming languages

D.3.3. Language constructs and features



# Spis treści

<b>Wprowadzenie</b> . . . . .	5
<b>1. Wstęp</b> . . . . .	7
1.1. Wprowadzenie do NianioLanga . . . . .	7
1.2. Typy w NianioLangu . . . . .	7
1.3. Cele projektu . . . . .	8
1.4. Podstawowe pojęcia . . . . .	8
<b>2. Metodyka pracy</b> . . . . .	9
2.1. Korzystanie z systemu kontroli wersji . . . . .	9
2.2. Zgłaszanie zmian i code review . . . . .	9
2.3. Techniki komunikacji w zespole . . . . .	9
<b>3. Kompilator NianioLanga</b> . . . . .	11
3.1. Budowanie drzewa AST . . . . .	11
3.2. Analiza semantyczna . . . . .	11
3.3. Architektura nlasma . . . . .	11
3.4. Translacja drzewa AST do nlasma . . . . .	11
3.5. Generowanie kodu C na podstawie nlasma . . . . .	11
3.6. Implementacja typów NianioLanga w C . . . . .	11
<b>4. Zmiana systemu typów</b> . . . . .	13
4.1. Rozdzielenie typu <code>ptd::sim</code> . . . . .	13
4.2. Typy <code>own</code> . . . . .	13
<b>5. Rozszerzenie nlasma</b> . . . . .	15
5.1. Przekazywanie informacji o typach z drzewa AST . . . . .	15
5.2. Statyczne sprawdzanie poprawności . . . . .	15
<b>6. Nowe implementacje typów</b> . . . . .	17
6.1. Typy proste . . . . .	17
6.2. Tablice . . . . .	17
6.3. Rekordy . . . . .	17
6.4. Typy wariantowe . . . . .	17
6.5. Konwersja <code>own</code> $\rightarrow$ <code>im</code> . . . . .	17
<b>7. Efekty optymalizacji i wnioski</b> . . . . .	19
7.1. Porównanie czasu wykonania programów . . . . .	19

8. Wkład poszczególnych członków zespołu . . . . .	21
Bibliografia . . . . .	23

# Wprowadzenie

NianioLang jest językiem programowania ogólnego przeznaczenia, którego twórcą jest założyciel firmy Atinea – Andrzej Gąsienica-Samek. Istniejący kompilator umożliwia translację NianioLanga do kilku języków, między innymi do Javy, JavaScriptu i C. Ze względu na chęć uproszczenia kompilacji NianioLanga utworzono środowisko uruchomieniowe dostarczające odpowiednie abstrakcje i pozwalające na korzystanie w C z dynamicznych struktur odpowiadających typom, jakie są dostępne do użycia w NianioLangu. Takie rozwiązanie nie jest niestety optymalne, szczególnie w przypadku niskopoziomowego języka jakim jest C. W tej pracy opisujemy wprowadzenie nowych typów danych i ich wsparcia w kompilatorze, co umożliwi generowanie natywnego kodu w C i znacznie zwiększy wydajność kompilowanych aplikacji przy zachowaniu podstawowych założeń języka.





# Rozdział 1

## Wstęp

### 1.1. Wprowadzenie do NianioLanga

NianioLang jest proceduralnym, imperatywnym językiem, którego celem jest uproszczenie pisania rozproszonych aplikacji stosując wzorzec projektowy Nianio[1]. Celem twórców języka jest dostarczenie narzędzia umożliwiającego operowanie na niemutowalnych strukturach o semantyce zbliżonej do języków funkcyjnych, ale prostszego w użyciu. Konstrukcje takie jak wskaźniki zostały usunięte, a w ich miejsce zaimplementowano system zarządzania obiektami w pamięci przez zliczanie referencji, aby pozbyć się jednego z najczęstszych źródeł błędów, występujących przy programowaniu niskopoziomowym. Takie zabiegi zmniejszają nieco wydajność języka, ale jego twórcy doszli do wniosku, że zysk z wysokopoziomowego podejścia do pisania aplikacji jest wystarczająco duży, aby tworzenie aplikacji w NianioLangu było opłacalne.

Kompilator NianioLanga jest rozwijany przez firmę Atinea, która używa NianioLanga w swoich projektach (m.in. InstaDB.com). Kod źródłowy kompilatora jest dostępny na platformie GitHub na licencji MIT.

### 1.2. Typy w NianioLangu

W NianioLangu mamy doczynienia z kilkoma wbudowanymi typami. Wszelkie typy tworzone przez użytkownika, to właściwie aliasy na typy wbudowane, co pomaga w zarządzaniu abstrakcją w programie. Dostępnych jest pięć wbudowanych typów:

- `ptd::sim` – liczby całkowite, zmiennoprzecinkowe oraz ciągi znaków
- `ptd::rec` – rekordy, czyli odpowiedniki struktur znanych np. z C
- `ptd::hash` – słowniki o kluczach będących ciągami znaków i wartościach danego typu
- `ptd::var` – typ wariantowy, który reprezentuje obiekty, mogące być w dokładnie jednym z określonego zbioru stanów i dodatkowo zawierać dane o typie właściwym dla danego stanu
- `ptd::arr` – tablice danych tego samego typu

Typy te mogą być ze sobą łączone w bardziej skompilowane konstrukcje, na przykład `ptd::arr(ptd::sim())` definiuje typ tablicy wartości prostych, a `ptd::rec({a => ptd::sim(), b => ptd::sim()})` definiuje typ rekordu o dwóch polach będących wartościami prostymi.

Podstawowymi założeniami systemu typów w NianioLangu są: niemutowalność struktur i opcjonalne typowanie.

- Niemutowalność struktur w NianioLangu jest pojęciem słabszym, niż w językach funkcyjnych. Oznacza ona, że język daje gwarancję, iż między dwoma kolejnymi dostęпами do zmiennej jej wartość nie ulegnie zmianie. W klasycznych językach imperatywnych, posiadających wskaźniki lub referencje nie jest to prawdą – jeśli istnieją dwa wskaźniki do jednej zmiennej, wartość odczytana z jednego z nich może się zmieniać nawet bez jego jawnej modyfikacji.
- Opcjonalne typowanie oznacza gwarancję, że dodanie lub usunięcie typów z programu nie zmieni jego semantyki. W skrajnym przypadku można usunąć z programu całą informację o typach i będzie on działał bez zmian (oczywiście może to być ze szkodą dla łatwości utrzymania lub wydajności). Jest to podejście przeciwne do stosowanego w takich językach jak C++, czy Haskell, których złożone systemy typów mają istotny wpływ na semantykę programu.

W wielu językach istnieje znacznie więcej wbudowanych typów, jednak powyższe są wystarczające, aby zbudować skomplikowane aplikacje, a jednocześnie dość proste, żeby rozpoczęcie programowania w NianioLangu nie było dla programisty wyzwaniem. W rozdziale *Kompilator NianioLanga* opisana została implementacja powyższych typów w języku C.

### 1.3. Cele projektu

Celem projektu jest modyfikacja kompilatora NianioLanga, w taki sposób, aby kod wynikowy w C zawierał mniejszą liczbę wywołań funkcji oraz skomplikowanych struktur dla prostych typów istniejących już w języku C. W tym celu wprowadzone zostają nowe typy z przestrzeni nazw `own`, które będą bardziej niskopoziomowymi odpowiednikami typów z przestrzeni nazw `ptd` (mianowicie rekordy, tablice i warianty). Dzięki nałożonym na nie ograniczeniom możliwe jest znaczne zwiększenie wydajności programów pisanych w NianioLangu przy zachowaniu niezmienionej semantyki języka.

### 1.4. Podstawowe pojęcia

## Rozdział 2

# Metodyka pracy

2.1. Korzystanie z systemu kontroli wersji

2.2. Zgłaszanie zmian i code review

2.3. Techniki komunikacji w zespole



## Rozdział 3

# Kompilator NianioLanga

### 3.1. Budowanie drzewa AST

### 3.2. Analiza semantyczna

### 3.3. Architektura nlasma

*Rejestry, wywołania funkcji, deklaracje typów, itp.*

### 3.4. Translacja drzewa AST do nlasma

### 3.5. Generowanie kodu C na podstawie nlasma

### 3.6. Implementacja typów NianioLanga w C



## Rozdział 4

# Zmiana systemu typów

### 4.1. Rozdzielenie typu `ptd::sim`

### 4.2. Typy `own`

*Jaki jest cel tych typów, ich semantyka, jakie ograniczenia na ich użycie nakładamy (w stosunku do typów `ptd`).*





## Rozdział 5

# Rozszerzenie nlasma

5.1. Przekazywanie informacji o typach z drzewa AST

5.2. Statyczne sprawdzanie poprawności



## Rozdział 6

# Nowe implementacje typów

*W tej sekcji w każdym podrozdziale będziemy opisywać dlaczego dotychczasowe rozwiązanie było nieefektywne, jak można je było poprawić, które rozwiązanie wybraliśmy, dlaczego.*

### 6.1. Typy proste

### 6.2. Tablice

### 6.3. Rekordy

### 6.4. Typy wariantowe

### 6.5. Konwersja `own` $\rightarrow$ `im`

Konwersja `own`ów na `im`y jest naturalną potrzebą języka - pomimo że typy tak naprawdę reprezentują inne podejście do przechowywania danych, nierzadko pojawia się konieczność konwertowania jednych na drugie. Podczas prac doszliśmy do wniosku, że przypadek konwersji `im`ów na `own`y nie musi być przez język obsługiwany, przede wszystkim dlatego, że tworzenie typu statycznego z typu dynamicznego w pewnym sensie mija się z celem korzystania z typów statycznych w ogóle, gdyż nie zabezpiecza to przed błędem w chwili konwersji. W drugą stronę zaś konwersja pozwala na łączenie nowego kodu ze starym oraz odejście od statycznego typowania w chwili, gdy chcemy korzystać z funkcji obsługujących wiele typów, na przykład implementując kontenery.

Początkowo mieliśmy kilka pomysłów dotyczących sposobu implementacji. Pierwszy pomysł polegał na zapisaniu w bibliotece C języka ogólnej funkcji konwertujących `own`y na `im`y. Szybko dostrzegliśmy wady tego podejścia, przede wszystkim trudność implementacji, dużą podatność na błędy oraz wątpliwą przenośność rozwiązania. Szczególnie trzeci argument jest ważny, gdyż dalsze kierunki rozwoju języka prawdopodobnie przewidują przywrócenie do kompilatora możliwości kompilowania na inne języki niż C.

Drugi pomysł polegał na tym, by stworzyć funkcję-wydmuszkę, w miejsce której kompilator `inline`-owałby właściwą definicję `ima` skonwertowanego z `own`a. Tutaj z kolei pojawił się problem typów rekurencyjnych (warianty) oraz puchnięcia kodu wynikowego, zawierającego bardzo podobne, potencjalnie bardzo długie sekcje.

Trzeci pomysł, który został faktycznie zrealizowany, jest pomysłem pośrednim - dla każdego typu nazwanego oraz dla każdego typu, który korzysta z funkcji-wydmuszki generować funkcję, której wywołanie byłoby następnie podpinane. Można to w pewnym sensie porównać

do generowania szablonów w C++, chociaż nasz system jest oczywiście bardziej wyspecjalizowany.

Pierwszym elementem rozwiązania jest zebranie informacji o tym, jakie funkcje należy wygenerować. Postanowiliśmy robić to w dwóch miejscach. Po pierwsze w momencie załadowania drzewa AST modułu do type checkera przechodzimy po funkcjach definiujących typy, czy przypadkiem nie korzystają one z ownów. Miejsce to zostało wybrane, gdyż jest to pierwsze miejsce, gdzie pojawia się konkretna już wiedza o typach zmiennych. Po drugie podczas sprawdzania typowania funkcji za każdym razem gdy natrafimy na naszą funkcję-wydmuszkę zapamiętujemy typ argumentu, z jakim została wywołana. Oczywiście pamiętać należy o przeskanowaniu, czy dane typy nie zawierają zagnieżdżonych ownów, do których także należy stworzyć stosowne funkcje.

Drugim elementem rozwiązania jest generowanie kodu. Jest to dosyć trywialne zagadnienie, gdyż na podstawie drzewa reprezentującego typ po prostu wypisujemy do stringa kolejne instrukcje przypisujące. Warto zwrócić jedynie uwagę na fakt, że typy mogą być zagnieżdżone (więc czasem trzeba wołać inną funkcję konwertującego wewnętrznego owna na ima), a dla rekordów trzeba wypisać przypisanie wszystkich wartości, kod wynikowy nie może robić tego pętlą.

Trzecim elementem jest wstrzyknięcie kodu. Skoro za miejsce generowania przyjęliśmy type checker, to nie możemy zrobić tego wcześniej. Z drugiej strony nie widzieliśmy potrzeby robienia tego później. Chcieliśmy, by kompilator mógł jak najszybciej wrócić do normalnego trybu pracy. Wobec tego korzystając z faktu, że sprawdzanie typów rozdzielone jest od wpisywania ich do drzewa AST, w trakcie wpisywania podmieniamy wywołania do funkcji-wydmuszki na wywołania odpowiedniej funkcji o automatycznie generowanej nazwie. Nazwa generowana jest na podstawie nazwy dla typów nazwanych oraz na podstawie zawartości dla anonimowych - rozwiązanie to ma taką zaletę, że redukujemy liczbę identycznych funkcji w kodzie wynikowym.

W tym momencie pozostaje jedynie problem dodania definicji nowych funkcji do drzewa AST. Wygenerowany kod jest parsowany do samodzielnego modułu, następnie przechodzi przez type checkera w sposób podobny do normalnego kodu - jest jedynie parę drobnych różnic technicznych polegających na rozwiązaniu problemu, że taki moduł odwołuje się do funkcji, których nie ma, gdyż są w naszym oryginalnym module. Są to jednak szczegóły implementacyjne, które można zobaczyć w kodzie kompilatora. Po przejściu przez type checker moduły są łączone w jeden. Gdy cały program przejdzie przez type checker dalsza kompilacja przebiega tak, jakby użytkownik sam wpisał sztucznie wygenerowane funkcje.

Skoro sami generujemy funkcje i wiemy, jak wygląda drzewo AST czy nie lepiej byłoby od razu tworzyć gotowe do podłączenia moduły, zamiast marnować czas procesora na przejście przez parser i type checker? Doszliśmy do wniosku, że czas kompilacji ma nieduże znaczenie w trakcie pisania programów. Automatyczne tworzenie drzewa AST daje zaś gwarancję, że w przypadku dalszej ewolucji kompilatora nie trzeba będzie pamiętać o tym, aby każda zmiana miała także i odzwierciedlenie w module generowania kodu. Ma to duże znaczenie przede wszystkim ze względu na fakt, że podczas kompilacji zakładamy, że generowane przez nas same struktury są poprawne, a tylko program użytkownika może zawierać błędy.

## Rozdział 7

# Efekty optymalizacji i wnioski

### 7.1. Porównanie czasu wykonania programów



## Rozdział 8

# Wkład poszczególnych członków zespołu

*Co zrobiliśmy w rozbiciu na osoby*





# Bibliografia

- [1] LK, *Wzorzec „Nianio” przykład* - 2006.