Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Michał Borkowski
Nr albumu: 370727

Jakub Bujak
Nr albumu: 370737

Marian Dziubiak
Nr albumu: 370784

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 contructs and features

Spis treści

Wprowadzenie

NianioLang jest językiem programowania ogólnego przeznaczenia, którego twórcą jest założyciel firmy Atinea – Andrzej Gąsienica-Samek. Poniższa praca realizowana jest na zlecenie firmy Atinea, która wykorzystuje NianioLang. 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, w przypadku kompilacji do C, zaimplementowano system zarządzania obiektami w pamięci przez zliczanie referencji, aby pozbyć sie 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[2] 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 wbudowancyh 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 dostępami 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 była 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. Dzięki temu zmniejszony został czas wykonania aplikacji, pozwalając kompilatorowi GCC na zastosowanie dodatkowych optymalizacji. W tym celu wprowadzone zostały nowe typy z przestrzeni nazw `own`, które będą bardziej niskopoziomowymi odpowiednikami typów z przestrzeni nazw `ptd` (rekordy, tablice i warianty). Dzięki nałożonym na nie ograniczeniom możliwe jest znaczne zwiększenie wydajności programów pisanych w NianioLangu przy zachowaniu niezmienionej semantyki języka. Jednocześnie zmienione zostały pewne podstawowe typy, mianowicie `ptd::sim` został rozbity na `ptd::int` oraz `ptd::string`, oraz wprowadzony został typ `ptd::bool`. Liczby całkowite i wartości boole'owskie mają w języku C natywną reprezentację i używając ich bezpośrednio uzyskujemy prostszy kod wynikowy w C, który jest łatwiejszy do optymalizacji przez kompilator GCC.

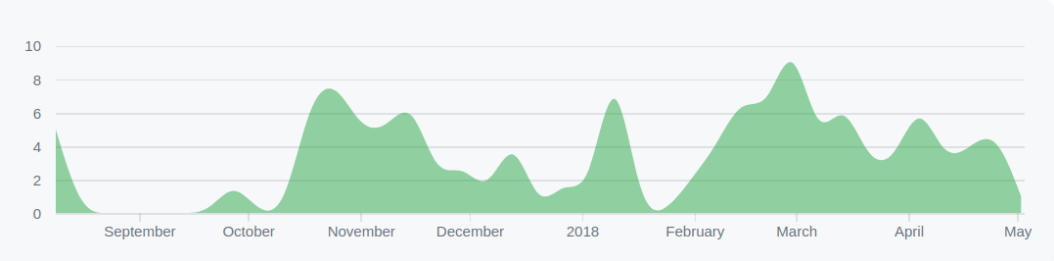
Skutkiem zmian jakie musiały zostać wprowadzone by osiągnąć cel projektu było utracenie możliwości kompilacji NianioLanga do języków innych niż C do czasu implementacji nowych typów w pozostałych językach. To zadanie pozostaje jednak poza zakresem niniejszej pracy.

1.4. Podstawowe pojęcia

Rozdział 2

Metodyka pracy

Prace nad kodem prowadzone były regularnie od połowy października do maja, z wyłączeniem sesji na przełomie lutego i stycznia. Na rysunku 2.1 przedstawiony został wykres zmian, które wpływały na główną gałąź repozytorium kodu. Co tydzień zespół prezentował swoje poczynania opiekunowi, co było stałym motywatorem aby coś robić.



Rysunek 2.1: Liczba commitów na głównej gałęzi repozytorium w czasie trwania projektu.

2.1. Korzystanie z systemu kontroli wersji

Podczas pracy zespołowej niezmiernie istotna jest wymiana i zarządzanie kawałkami kodu, jakie piszą poszczególni członkowie zespołu. Ponieważ każdy w zespole pracował już wcześniej z systemem kontroli wersji Git, to został on użyty w tym projekcie. Dodatkowym atutem tego systemu kontroli wersji jest szeroka dostępność darmowych publicznych repozytoriów kodu online. Kod projektu umieszczony został na portalu GitHub, który jest największym serwisem pozwalającym na hostowanie kodu online.

Początkowo każdy w zespole miał swoją gałąź w repozytorium, oznaczoną jego imieniem, na której wprowadzał swoje zmiany. Jednak w trakcie pracy wyszło, że praktyczniejszym podejściem jest tworzenie osobnych gałęzi kodu dla każdego zadania. GitHub oprócz repozytorium kodu, udostępnia również narzędzia do zarządzania zadaniami w projekcie, co zostało wykorzystane.

2.2. Zgłaszanie zmian i code review

Po napisaniu kodu rozwiązującego dane zadanie, zgłaszany był tzw. Pull Request, czyli żądanie zatwierdzenia zmian. Aby zmiany zostały zatwierdzone i umieszczone w głównej gałęzi projektu, musiały być zatwierdzone przez conajmniej jednego członka zespołu, innego niż ten

który zgłosił PR. GitHub udostępnia widok naniesionych zmian wraz z możliwością dodawania komentarzy do konkretnych fragmentów kodu. Możliwy więc był dialog między recenzentem kodu, a jego twórcą, tak aby ostatecznie zatwierdzone zmiany spełniały oczekiwania obu stron.

2.3. Techniki komunikacji w zespole

Większość komunikacji w zespole odbywała się online przez portal społecznościowy Facebook. Na konwersacji grupowej na tym portalu zespół dogadywał większość detali implementacyjnych przed rozpoczęciem ich kodowania, a w początkowej fazie projektu, głównym tematem była specyfika języka NianioLang, z którym większość członków zespołu miała po raz pierwszy styczność.

Drugim kanałem komunikacji były zadania i PRy na GitHubie. Zadania w prosty sposób opisywały co i kto ma zrobić, a opisy PRów określały co zostało przez daną osobę zrobione. Tej komunikacji w projekcie było najmniej.

Ostatnim sposobem komunikacji były cotygodniowe spotkania podczas zajęć z Zespołowego Projektu Programistycznego, po których zespół spędzał 10-20 minut na dogadywaniu się, kto wybiera, które zadanie, a w przypadku pytań lub problemów, na ich rozwiązaniu.

Rozdział 3

Kompilator NianioLanga

3.1. Budowanie drzewa AST

Pierwszym etapem kompilacji jest parsowanie programu wejściowego do drzewa AST. Takie drzewo jest strukturą przechowującą informacje o składniowej roli wyrażeń, które przetwarzać można na dalszych etapach kompilacji. Celem jest przetworzenie tekstu programu do postaci przypominającej derywację gramatyki języka. Chociaż książkowa definicja drzewa AST mówi wyłącznie o składni, kompilator NianioLanga już na tym etapie dokonuje pewnej podstawowej analizy semantycznej, gdyż wydzielenie tych czynności spowodowałoby dużą duplikację kodu związanego z czynnością obchodzenia drzewa.

Najpierw parser próbuje przeczytać listę importowanych modułów, gdyż zgodnie ze składnią języka musi ona zostać umieszczona na początku programu. Po tym etapie następuje parsowanie listy funkcji. Każdy etap parsowania odbywa się poprzez analizę możliwie najdłuższej ilości tekstu, dopisanie błędów, ustawienie wskaźnika następnej pozycji oraz innych pomocniczych wartości, które są przechowywane w zmiennej reprezentującej stan parsera, a na końcu zwrócenie sparsowanej wartości. Nie ma znaczenia, co zwróci dana funkcja w razie błędu, gdyż wtedy kompilacja zostanie zatrzymana, a użytkownik poinformowany o błędzie. W przypadku, gdy nie jest jasne, co należy sparsować (na przykład nie wiadomo, czy oczekujemy kolejnej deklaracji importu modułu, czy definicji funkcji), próbuje się przeczytać różne wartości, aż któraś zostanie poprawnie sparsowana. Ważne jest, by różne możliwości przetwarzać od największej do najmniejszej, jeżeli reprezentacja jedna może być początkiem drugiej, np. najpierw parsować należy operatory, a dopiero potem wyrażenia, chyba że kolejność nie ma znaczenia. Przykładowo w wyrażeniu *1 + 2 - 3* nie ma znaczenia, czy spróbujemy sparsować najpierw operator dodawania, czy odejmowania, ale operator musi być sparsowany zanim sparsowana zostanie liczba. Innym przykładem jest `if`, po którym może nastąpić `else` i najpierw należy spróbować sparsować blok `else` zanim przejdzie się do parsowania następnych komend.

Parsowanie funkcji odbywa się poprzez sparsowanie nagłówka, a następnie sparsowanie jej wnętrza jako jednej komendy. Przyjęte jest, że komenda może być także listą komend, co parsuje się w sposób rekurencyjny. Ułatwia to parsowanie takich komend jak `for` czy `if`. Podejście to umożliwia także tworzenie rekurencyjnych funkcji, z których każda odpowiedzialna jest za jeden element gramatyki języka. Funkcje te sposób działania opierają na opisanym w poprzednim paragrafie schemacie.

Projekt nie wymagał znaczących zmian na etapie parsowania, dlatego nie zostaną one tutaj omówione. Jedynym wyjątkiem jest dodanie sprawdzania po sparsowaniu funkcji, czy nie definiuje ona typu, a następnie wpisania tego typu do drzewa. Zostało to jednak wykonane

przy użyciu dostępnych już funkcji. Jest to wykonywane, by na dalszych etapach można było zapytać o dowolny definiowany typ, co ułatwia proces kompilacji. Sytuacja błędna nie jest jednak obsługiwana, stanie się to dopiero podczas sprawdzania poprawności typów.

3.2. Analiza semantyczna

Drugim etapem kompilacji jest sprawdzenie poprawności typów. Wbrew nazwie etap ten nie ma na celu wyłącznie sprawdzenia, czy program jest poprawny typowo – do drzewa programu wpisywane są informacje o uzyskanych typach. Jest to konieczne, by móc dopasować typy nianiolanga do typów języka wynikowego. Ponadto zachodzi tutaj analiza opisana w rozdziale Konwersja own → im.

Na tym etapie sprawdza się, czy zaimportowane moduły faktycznie istnieją, czy definicje funkcji nie dublują się, oraz w końcu czy używane typy zgodne są z oczekiwanymi. Trzeci element rozumiany jest dwojako. Po pierwsze, jeżeli dana zmienna ma zadeklarowany przez użytkownika typ, sprawdza się, czy używana jest ona wyłącznie przez funkcje, które takiego typu oczekują. Ze względu na nietrywialny system typów wydzielone są specjalne funkcje sprawdzające czy i w jakim zakresie dwa typy są zgodne. Przykładowo jeżeli `im` używany jest w kontekście `hash`, to zgodnym typem jest `hash`. Jeżeli zaś funkcja oczekuje `int` a podawany jest `array`, to zgłaszany jest błąd. W tym kontekście o typach myśleć można jako o zbiorach możliwych wartości, na których wykonywane są operacje, przede wszystkim przecięcia.

Drugie rozumienie, które stosuje type checker, jest podobne do pierwszego, lecz w kontekście wnioskowania typów na podstawie wykonywanych na nich operacji. Jeżeli dana zmienna inicjalizowana jest jako `int`, a następnie używana jako `hash`, to zgłaszany jest błąd. Warto zaznaczyć, że operacje na typach wywnioskowanych są subtelniejsze, gdyż wiele typów jest konwertowalnych na `im` oraz w drugą stronę, przez co nie jest możliwa całkowita, ścisła kontrola typów, ta ma miejsce dopiero w runtime. Ponownie, wydzielone są funkcje operujące na typach w analogii do zbiorów.

Samą budowę type checker ma podobną w pewnym sensie do parsera. W rekurencyjny sposób sprawdzamy, jakie typy mają poszczególne elementy drzewa oraz czy są one poprawne. Tak jak w parserze zapisane jest, że operator `+` zawiera słowo kluczowe `+`, tak odpowiednia funkcja w type checkerze wie, że argumenty tego operatora muszą mieć typ zwracany `int`. Ponownie pozwala to na podział funkcji w taki sposób, by każda dotyczyła jednego, jasno wyszczególnionego elementu.

3.3. Architektura nlasma

nlasm to język stosowany w kompilatorze jako pomost pomiędzy NianioLangiem a językiem wyjściowym, gdyż bezpośrednia kompilacja NianioLanga do C, lub też innego języka, może okazać się trudna, ponieważ używane struktury językowe znacząco od siebie odbiegają. Umożliwia to w łatwy sposób tworzenie modułów kompilujących NianioLanga do różnych języków. Taka technika jest szeroko stosowana w świecie kompilatorów, jednym z najszerzej znanych przykładów jest LLVM. Obecność autorskiego nlasma podyktowana jest przede wszystkim potrzebą języka pośredniego, który z jednej strony będzie możliwie najprostszy, z drugiej w pewnym sensie zachowa strukturę programu wyjściowego.

Funkcje w nlasmie nie operują na zmiennych, lecz na rejestrach. Każdy rejestr ma swój unikalny w obrębie funkcji numer oraz typ. Także argumenty funkcji są rejestrami. Wprowadzenie typów do nlasma było jednym z elementów projektu, gdyż o ile poprzednio wszystkie zmienne konwertowane były do typu `im`, o tyle teraz przechować należy strukturę typów `own`

czy `int`. Typy w nlasmie mają charakter wyłącznie informacyjny, to znaczy nie mają żadnego znaczenia semantycznego.

Najważniejszą cechą nlasma jest nie drzewiasta, lecz liniowa struktura programu. Oznacza to, że argumentami wywołań mogą być wyłącznie rejestry, a nie zmienne czy inne wywołania. W znacznym stopniu ułatwia to dalszą pracę kompilatora, przede wszystkim z powodu mniejszej liczby niuansów dotyczących na przykład wartości zwracanych, przez co kod generatora języka docelowego może być znacznie prostszy. Ponadto generowanie kodu może odbywać się w sposób liniowy, to znaczy każda komenda nlasma może mieć jednoznaczne, proste odzwierciedlenie w języku docelowym. Dzięki temu generator może dokonywać analizy wyłącznie w zakresie charakterystycznym dla danego języka, na przykład do generatora C zostało wprowadzone sortowanie topologiczne deklaracji typów.

Inną ważną cechą nlasma, wynikającą częściowo z poprzedniej, jest brak pętli. Translator NianioLanga do nlasm musi rozbić każdą pętlę na etykietę oraz komendę `goto`. Takie podejście ułatwia tłumaczenie kodu na język docelowy, gdyż w różnych językach pętle działają w odrobinę inny sposób, zaś `goto` tworzy nieskomplikowany, wspólny interfejs.

Ponadto nlasm zawiera polecenia służące do zarządzania pamięcią. Zakłada się, że język docelowy będzie korzystał ze zliczania referencji w celu zwalniania pamięci. Aby nie pisać tej samej funkcjonalności oddzielnie dla wielu generatorów, nlasm zawiera polecenia mówiące o tym, że licznik referencji danego rejestru (o ile jest on wskaźnikiem) należy zmniejszyć lub zwiększyć, co można w prosty sposób przełożyć na wywołania innych języków. Oczywiście, jeżeli generator wybierze inną metodę zarządzania pamięcią, na przykład generując kod Javy, może te polecenia po prostu zignorować. W przypadku generowania języka z manualnym zarządzaniem pamięcią, na przykład C, polecenia te są bardzo pomocne.

Przykładowe komendy nlasma:

- if_goto – sprawdza wartość w rejestrze. Jeżeli prawda, wykonuje skok pod etykietę
- clear – sygnalizuje, że wartość licznika referencji powinna spaść
- call – wywołuje funkcję o danej nazwie z danymi rejestrami jako argumentami, zapisuje wynik w danym rejestrze
- move – kopiuje wartość między dwoma rejestrami

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 → im

Konwersja typów `own` na typ `ptd::im`, który jest uniwersalną reprezentacją typów z przestrzeni `ptd`, jest naturalną potrzebą języka. Obiekty o typie z przestrzeni nazw `own` są ograniczone w zasięgu, więc chcąc zwrócić wartość takiego obiektu lub przekazać do funkcji, która przyjmuje argumenty z przestrzeni nazw `ptd`, musimy go przekonwertować.

Analiza przypadków użycia konwersji pokazała, że konwersja z typów `ptd` na typy `own` nie ma szczególnego zastosowania i nie musi być przez język obsługiwana, przede wszystkim dlatego, że tworzenie typu statycznego z typu dynamicznego w pewnym sensie mija się z celem korzystania z typów statycznych w ogóle, gdyż narzut wydajnościowy związany z tworzeniem obiektów typu `ptd::im`, a potem konstruowanie z nich obiektów typu `own` byłby zbyt duży. 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.

Istnieje wiele sposobów na implementację konwersji. Pierwszy z nich polega na zapisaniu w bibliotece C języka ogólnej funkcji konwertujących obiekty `own` na obiekty `ptd::im`. Podejście to ma jednak wiele wad, 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 sposób polega na stworzeniu funkcji-wydmuszki przyjmującej w argumencie obiekt `own`, która w trakcie kompilacji podmieniana jest na właściwy kod dokonujący konstrukcji obiektu `ptd::im`. Duże problemy stwarza jednak implementacja dla typów rekurencyjnych

(warianty) oraz puchnięcie kodu wynikowego, zawierającego bardzo podobne, potencjalnie bardzo długie sekcje.

Trzeci sposób, który został zrealizowany, jest niejako połączeniem dwóch poprzednich – dla każdego typu nazwanego oraz dla każdego typu, który korzysta z funkcji-wydmuszki, generowana jest funkcja, podpinana w miejsce wydmuszki. Można to w pewnym sensie porównać do generowania szablonów w C++, aczkolwiek ten system jest bardziej wyspecjalizowany.

Pierwszym elementem rozwiązania jest zebranie informacji o tym, jakie funkcje należy wygenerować, co przeprowadzane jest w dwóch miejscach. Ze względu na możliwość używania typów z innych modułów, niż w danym momencie wykonywany, każdy moduł udostępniać musi komplet funkcji do rzutowania typów, które definiuje. Najpierw, po załadowaniu drzewa AST modułu do type checkera, dokonywany jest obchód po funkcjach definiujących typy sprawdzający, czy nie definiują one typów `own`. Miejsce to zostało wybrane, gdyż jest to pierwsze miejsce, gdzie pojawia się konkretna wiedza o typach zmiennych. Następnie, podczas sprawdzania typów funkcji, za każdym razem, gdy napotykana jest funkcja-wydmuszka, zapamiętywany jest typ argumentu, z jakim została ona wywołana. Typy te są również rekurencyjnie sprawdzane, czy nie zawierają innych typów `own`, do których także należy stworzyć stosowne funkcje.

Drugim elementem rozwiązania jest generowanie kodu. Tworzona jest zmienna tekstowa, która następnie wypełniana jest instrukcjami tworzącymi obiekt `ptd::im`. Ponieważ typy mogą być zagnieżdżone, w razie potrzeb wywołują one inne funkcje konwertujące wewnętrzne obiekty `own` na `im`. Warto podkreślić, że w przypadku typów `own::rec` oraz `own::var` funkcje rzutujące nawet podobne typy są istotnie różne. Rzutując rekordy należy explicite przypisać wszystkie pola, nie jest możliwe zrobienie tego w ogólny sposób pętlą, zaś rzutując warianty wykonać należy pełną instrukcję `match`.

Trzecim elementem jest wstrzyknięcie kodu. Sprawdzanie typów rozdzielone jest od wpisywania ich do drzewa AST i w trakcie tego wpisywania podmieniane są wywołania funkcji-wydmuszki na wywołania odpowiedniej funkcji o automatycznie generowanej nazwie. Generowana nazwa jest tworzona na postawie nazwy dla typów nazwanych, oraz na podstawie zawartości dla typów anonimowych – uzyskujemy w ten sposób deduplikację kodu, generując w kodzie wynikowym C tylko jedną funkcję dla danego typu.

Pozostaje jeszcze problem dodania definicji nowych funkcji do drzewa AST. Wygenerowany kod jest parsowany do samodzielnego modułu, który przechodzi przez type checker w sposób podobny do normalnego kodu. Po przejściu przez type checker moduły są łączone w jeden. Gdy cały program przejdzie przez type checker dalsza kompilacja przebiega tak, jak gdyby użytkownik sam utworzył sztucznie wygenerowane funkcje.

Generując funkcje, a z nich drzewo AST, można mieć wątpliwości, czy nie lepiej byłoby od razu tworzyć gotowe do podczepienia poddrzewa, czy nawet kod wynikowy w C, zamiast marnować czas procesora na przejście przez parser i type checker. Zautomatyzowane tworzenie drzewa AST daje gwarancję, że w przypadku dalszej ewolucji kompilatora nie trzeba będzie pamiętać o tym, aby każda zmiana miała odzwierciedlenie w module generowania kodu. Ze względu na dużą dynamikę rozwoju języka jest to cecha pożądana. Ponadto zmniejszanie czasu kompilacji programów nie było głównym celem projektu, a analiza przypadków użycia wykazała, że narzut czasowy jest w pełni akceptowalny.

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.

[2] NianioLang Compiler - GitHub