Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI, INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ

KATEDRA INFORMATYKI STOSOWANEJ

Praca dyplomowa inżynierska

Implementacja maszyny wirtualnej dla funkcyjnych języków programowania wspierających przetwarzanie współbieżne.

Implementation of a virtual machine for functional programming languages with support for concurrent computing.

Autor: *Kajetan Rzepecki*
Kierunek studiów: *Informatyka*
Opiekun pracy: *dr inż. Piotr Matyasik*

Kraków, 2014

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i nie korzystałem ze źródeł innych niż wymienione w pracy.

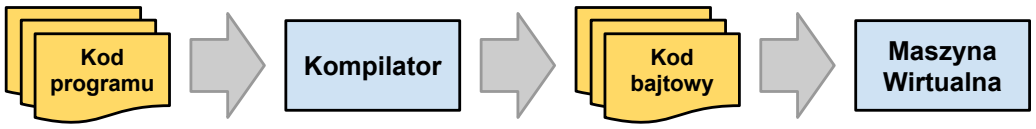
Serdecznie dziękuję Lucynie oraz siostrze Alicji za cierpliwość i wsparcie podczas tworzenia pracy dyplomowej.

Spis treści

1. Wstęp

Tematem pracy jest implementacja *maszyny wirtualnej* dla funkcyjnych języków programowania wspierających *przetwarzanie współbieżne*.

Maszyna wirtualna jest warstwą abstrakcji leżącą pomiędzy programem a rzeczywistym sprzętem, która pozwala uniezależnić ów program od rozbieżności w działaniu różnych architektur komputerów. Wystarczy zaimplementować maszynę wirtualną dla danej architektury rzeczywistego sprzętu by umożliwić uruchamianie na niej wszystkich kompatybilnych z programów. Rysunek 1.1 prezentuje uproszczony schemat takiego rozwiązania - programy docelowe zostają skompilowane do *kodu bajtowego* akceptowanego przez maszynę wirtualną a dopiero ów bajtkod jest przez nią uruchamiany.



Rysunek 1.1: Schemat interakcji z Maszyną Wirtualną.

Przetwarzanie współbieżne opiera się o współistnienie wielu procesów, które konkurują o dostęp do współdzielonych zasobów. W kontekście pracy, przetwarzanie współbieżne jest rozumiane jako asynchroniczne przekazywanie wiadomości pomiędzy działającymi, autonomicznymi procesami, czyli jako Model Aktorowy [1, 2].

Celem pracy jest stworzenie interpretera kodu bajtowego zdolnego uruchamiać kod skompilowanych programów, kolektora obiektów nieosiągalnych umożliwiającego automatyczne zarządzanie pamięcią oraz architektury symetrycznego multiprocesora (SMP) zapewniającego rzeczywistą współbieżność uruchamianych programów w oparciu o Model Aktorowy. Językiem implementacji projektu jest język `D` (w wersji 2.0 opisanej w [3]), stosunkowo nowoczesny, kompilowany do kodu maszynowego następca języka `C++`.

1.1. Motywacja pracy

Motywacją powstania pracy są problemy napotkane podczas użytkowania języka `Erlang` [4], dotyczące wydajności przesyłania wiadomości średniego rozmiaru w obecnej, standardowej jego implementacji. Problemy owe zilustrowano na listingu 1.

Zaprezentowany fragment kodu odczytuje plik w formacie JSON, który następnie jest dekodowany do wewnętrznej reprezentacji posiadającej skomplikowaną strukturę, by ostatecznie został on wysłany do dużej liczby współbieżnie działających procesów celem dalszego przetwarzania (linia 8). Rozwiązanie takie powoduje znaczący spadek wydajności.

```
1  start() ->
2      Data = file:read("file.json"),     %% <<"Dane ...">>
3      transmogrify(Data).
4
5  transmogrify(Data) ->
6      Pids = framework:spawn_bajilion_procs(fun do_stuff/1),
7      JSON = json:decode(Data),          %% {[Dane ...]}
8      framework:map_reduce(Pids, JSON). %% !#&%^@
9
10 do_stuff(JSON) ->
11     %% Operacje na danych.
12     result.
```

Listing 1: Fragment kodu prezentujący problem występujący w języku `Erlang`.

Język `Erlang` wykorzystuje skomplikowaną architekturę pamięci, która w różny sposób traktuje obiekty różnego typu. Większość obiektów, w szczególności skomplikowana strukturalnie reprezentacja danych w formacie JSON, przechowywana jest w prywatnych stertach każdego procesu i musi być kopiowana podczas przesyłania jej w wiadomościach pomiędzy nimi. Reguła ta nie dotyczy danych binarnych, w szczególności danych odczytanych z pliku, ponieważ te korzystają z innych algorytmów nie wymagających kopiowania kosztem większego zużycia pamięci.

W związku z tym, aby zaradzić problemowi opisanemu powyżej, wystarczy przenieść operację dekodowania danych odczytanych z pliku bezpośrednio do procesów na nich operujących (listing 2). W nowej wersji procesy przesyłają jedynie dane binarne, które nie wymagają kopiowania pamięci, a narzut wydajności spowodowany wielokrotnym ich dekodowaniem jest niższy niż ten spowodowany nadmiernym kopiowianiem. W efekcie, kod działa wydajniej, kosztem logiki przepływu danych i organizacji modułów.

Celem pracy jest uniknięcie problemu nadmiernego kopiowania pamięci przez wybranie odpowiedniego modelu pamięci i implementację algorytmów kolekcji obiektow nieosiągal-

```
1 transmogrify(Data) ->
2     Pids = framework:spawn_bajilion_procs(fun do_stuff/1),
3     framework:map_reduce(Pids, Data).
4
5 do_stuff(Data) ->              %% <<"Dane ...">>
6     JSON = json:decode(Data), %% {[Dane ...]} * bazylion
7     %% Operacje na danych.
8     result.
```

Listing 2: Suboptymalne rozwiązanie problemu w języku `Erlang`.

nych, które umożliwiają przesyłanie wiadomości pomiędzy procesami bez konieczności kopiowania ich zawartości.

1.2. Zawartość pracy

W skład pracy wchodzi implementacja interpretera kodu bajtowego, kolektora obiektów nieosiągalnych oraz symetrycznego multiprocesora (SMP).

Sekcja 1 opisuje cele, motywację, zakres oraz zawartość pracy.

Sekcja 2 przybliża architekturę maszyny wirtualnej ThesisVM zaimplementowanej w ramach pracy, zaczynając od reprezentacji pośredniej programów (TVMIR) i jej kompilacji do kodu bajtowego, przez interpretację kodu bajtowego i zarządzanie pamięcią do projektu przetwarzania współbieżnego.

Sekcja 3 szczegółowo opisuje implementację interpretera kodu bajtowego maszyny wirtualnej ThesisVM. Zaprezentowane zostają reprezentacje różnych obiektów, na których operuje maszyna, implementacja wpudowanych operatorów i funkcji prymitywnych oraz reprezentacja i generowanie kodu bajtowego akceptowanego przez interpreter.

Sekcja 4 szczegółowo prezentuje implementację wybranego modelu pamięci, alokatora nowych obiektów oraz kolektora obiektów nieosiągalnych.

Sekcja 5 szczegółowo opisuje implementację asynchronicznego przekazywania wiadomości i symetrycznego multiprocesora w maszynie ThesisVM. Zaprezentowana zostaje implementacja Modelu Aktorowego i harmonogramowania procesów.

Sekcja 6 zawiera podsumowanie pracy oraz zarys możliwych kierunków dalszego rozwoju projektu.

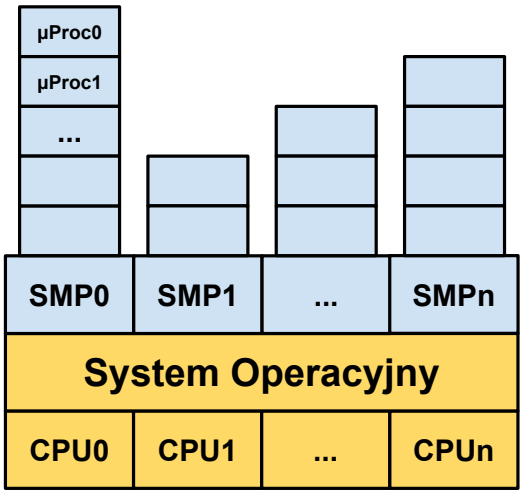
Dodatki A, B i C zawierają odpowiednio wskazówki użytkowania ThesisVM i przykładowe programy gotowe do uruchomienia na maszynie wirtualnej, spis wbudowanych operatorów i funkcji prymitywnych oraz spisy rysunków, tablic i fragmentów kodu znajdujących się w tekście pracy.

2. Architektura ThesisVM

Ninejsza sekcja opisuje architekturę maszyny wirtualnej ThesisVM powstałej na potrzeby pracy oraz języka przez nią akceptowanego.

Rysunek 2.1 zawiera schematyczną reprezentację maszyny wirtualnej ThesisVM uwzględniającą architekturę procesora sprzętu, na którym działa system operacyjny oraz sama maszyna wirtualna. Na schemacie widać poszczególne podsystemy ThesisVM, takie jak autonomiczne procesy (zwane dalej *mikroprocesami*, `µProcN`), czy symetryczne multiprocesory (zwane dalej `SMPn`).

Mikroprocesy są przypisane do symetrycznych multiprocesorów w stosunku wiele-do-jednego, to znaczy każdy mikroproces jest przypisany do dokładnie jednego symetrycznego multiprocesora, który natomiast może zarządzać zbiorem wielu mikroprocesów.



Rysunek 2.1: Architektura maszyny wirtualnej ThesisVM.

Każdy symetryczny multiprocesor działa w osobnym wątku procesora sprzętowego, zapewniając rzeczywistą współbieżność. Wszystkie `SMPn` są takie same i wykonują takie same zadania, czyli harmonogramowanie i wywłaszczanie mikroprocesów, a różni je

jedynie stan, w którym się znajdują oraz zbiór procesów, którymi zarządzają. Na schemacie widnieje mapowanie jeden-do-jednego pomiędzy rdzeniami procesora (`CPUn`) a poszczególnymi `SMPn`, nie jest to jednak wymóg konieczny i zależy od konfiguracji maszyny wirtualnej. Konfigurowalna ilość równocześnie działających SMP pomaga osiągnąć lepszą skalowalność maszyny wirtualnej i może być zmieniana dynamicznie wedle potrzeb.

Pozostając w zgodzie ze schematem przedstawionym na rysunku 1.1, interakcja z maszyną ThesisVM przebiega w analogiczny sposób. Kod programów w reprezentacji pośredniej (TVMIR) jest kompilowany do kodu bajtowego akceptowanego przez maszynę wirtualną, która następnie go ładuje i wykonuje umożliwiając zrównoleglenie obliczeń poprzez tworzenie nowych procesów i przesyłanie pomiędzy nimi wiadomości.

2.1. Reprezentacja pośrednia programów

ThesisVM wykorzystuje prostą reprezentację pośrednią programów w postaci TVMIR - języka lisp'owego z rodziny `Scheme` [5], który jest dostatecznie ekspresywny, by można w nim było zapisać nietrywialne algorytmy, a jednocześnie na tyle prosty, by ułatwić jego późniejszą kompilację do kodu bajtowego akceptowanego przez maszynę wirtualną.

Języki pośrednie reprezentacji programów są często stosowane w implementacjach wielu maszyn wirtualnych, takich jak ParrotVM, czy CoreVM [6], a także w implementacjach kompilatorów kodu maszynowego wielu języków programowania (na przykład GCC, LLVM). Reprezentacje pośrednie mają wiele zalet, począwszy od ułatwienia wsparcia dla szerszej gamy języków wysokiego poziomu, na możliwości tworzenia wygodnych założeń dodatkowych kończąc.

Na listingu 3 spisana w formacie BNF została gramatyka języka reprezentacji pośredniej wykorzystanego w maszynie wirtualnej ThesisVM. Gramatyka ta jest nieskomplikowana i w dużej mierze przypomina gramatiki różnych dialektów języka `Lisp`.

Języki z rodziny `Lisp` są bardzo wygodnym medium dla pośredniej reprezentacji programów ponieważ przedstawiają one drzewo syntaktyczne analizowanego kodu programu i nie wymagają skomplikowanego algorytmu parsowania. Dodatkowo, homoikoniczność tych języków może pomóc w tworzeniu narzędzi służących do przetwarzania kodu rozpatrywanego języka (w szczególności kompilatorów) bezpośrednio w rozpatrywanym języku. Temat ten został dogłębnie zbadany w [5]. Dodatek A zawiera przykłady kodu w języku pośredniej reprezentacji programów TVMIR.

Język reprezentacji pośredniej przedstawiony w pracy wymaga stworzenia kilku założeń dodatkowych dotyczących transformacji kodu. Najważniejszym z nich jest konieczność przeprowadzenia operacji lambda-unoszenia (ang. *lambda lifting*), opisanej bardzo dokładnie w [6], której efekt zaprezentowano na listingu 4.

```
<program>         ::= <definitions>
<definitions>     ::= <definition> <definitions> | ''
<definition>      ::= '(' 'define' '(' <symbol> <arguments> ')'
                                   <expression> ')'
<arguments>       ::= <symbol> <arguments> | ''
<expression>      ::= <value> | <application> | <primop>
                    | <conditional> | <quote> | <spawn>
<value>           ::= <list> | <symbol> | <number>
<application>     ::= '(' <expression> <expressions> ')'
<expressions>     ::= <exrpession> <exrpessions> | ''
<conditional>     ::= '(' 'if' <expression>
                               <expression>
                               <expression> ')'
<quote>           ::= ''' <expression> | '(' 'quote' <epression> ')'
<spawn>           ::= '(' 'spawn' <symbol> <expression> ')'
<primop>          ::= '(' 'primop' <symbol> <expressions> ')'
<list>            ::= '(' <expressions> ')'
<symbol>          ::= <literal-string> | <atom>
<literal-string>  ::= '"' "Dowolny literał znakowy." '"'
<atom>            ::= "Dowolny literał znakowy bez znaków białych."
<number>          ::= "Dowolny literał liczbowy."
```

Listing 3: Gramatyka języka TVMIR.

Lambda-unoszenie polega na transformacji ciał funkcji w taki sposób, by tworzone w nich funkcje anonimowe zostały przeniesione na poziom główny zasięgu nazw (ang. *top-level scope*) dzięki czemu do ich implementacji wystarczy jedynie częściowa aplikacja funkcji. Na drugiej części listingu 4 funkcja `make-adder` zwracająca anonimową funkcję została transformowana na dwie funkcje, z których `make-adder` pozostaje funkcją unarną, która korzysta z częściowej aplikacji funkcji binarnej `__make-adder_lambda0` wykonującej operację dodawania.

Pełna i poprawna implementacja operacji lambda-unoszenia jest skomplikowana, toteż nie została zawarta w dołączonym do projektu kompilatorze kodu bajtowego i musi zostać wykonana ręcznie.

Język pośredniej reprezentacji programów zastosowany w maszynie wirtualnej ThesisVM jest bardzo podobny do języka `Core Lang` wykorzystywanego w [6], jednak nie wspiera on niektórych jego konstrukcji, takich jak `let(rec)`, czy definicje dowolnych obiektów złożonych. Z drugiej strony wspiera on konstrukcje związane z Modelem Akto-

```
;; Przed lambda-unoszeniem:
(define (make-adder n)
  (lambda (x)
    (+ x n)))
```

```
;; Po lambda-unoszeniu:
(define (__make-adder_lambda0 n x)
  (+ x n))

(define (make-adder n)
  (__make-adder_lambda n))
```

Listing 4: Fragmenty kodu prezentujące operację lambda-unoszenia.

rowym (`receive`, `send` oraz `spawn`) oraz jest w stanie emulować brakujące konstrukcje odpowiednio przez wykorzystanie transformacji kodu połączonej z lambda-unoszeniem (listing 5) oraz "tagowania" list (przechowywania informacji o typie obiektu w pierwszym elemencie listy enkodującej ten obiekt).

```
;; Przed transformacją:
(define (function x)
  (let ((value (* 2 x)))
    (* value value)))

;; Po transformacji:
(define (function x)
  ((lambda (value)
     (* value value))
   (* 2 x)))
```

```
;; Po lambda-unoszeniu:
(define (__function_lambda0 value)
  (* value value))

(define (function x)
  (__function_lambda0 (* 2 x)))
```

Listing 5: Ograniczona implementacja konstrukcji `let`.

Kolejnym podobnym językiem reprezentacji pośredniej jest `Core Erlang` [7] wykorzystywany w standardowej implementacji języka `Erlang`. TVMIR jest bardzo okrojoną wersją języka `Core Erlang`, pozbawioną elementów dopasowywania wzorców, która jednak wspiera pozostałe ważne jego elementy, takie jak konstrukcje odpowiedzialne za tworzenie procesów oraz przesyłanie i odbieranie wiadomości. Istnieje możliwość rozszerzenia funkcjonalności TVMIR celem wsparcia pełnej specyfikacji `Core Erlang` [8], jednak jest to poza zakresem pracy. Więcej informacji o przyszłych kierunkach rozwoju projektu zostało zawarte w sekcji 6.4.

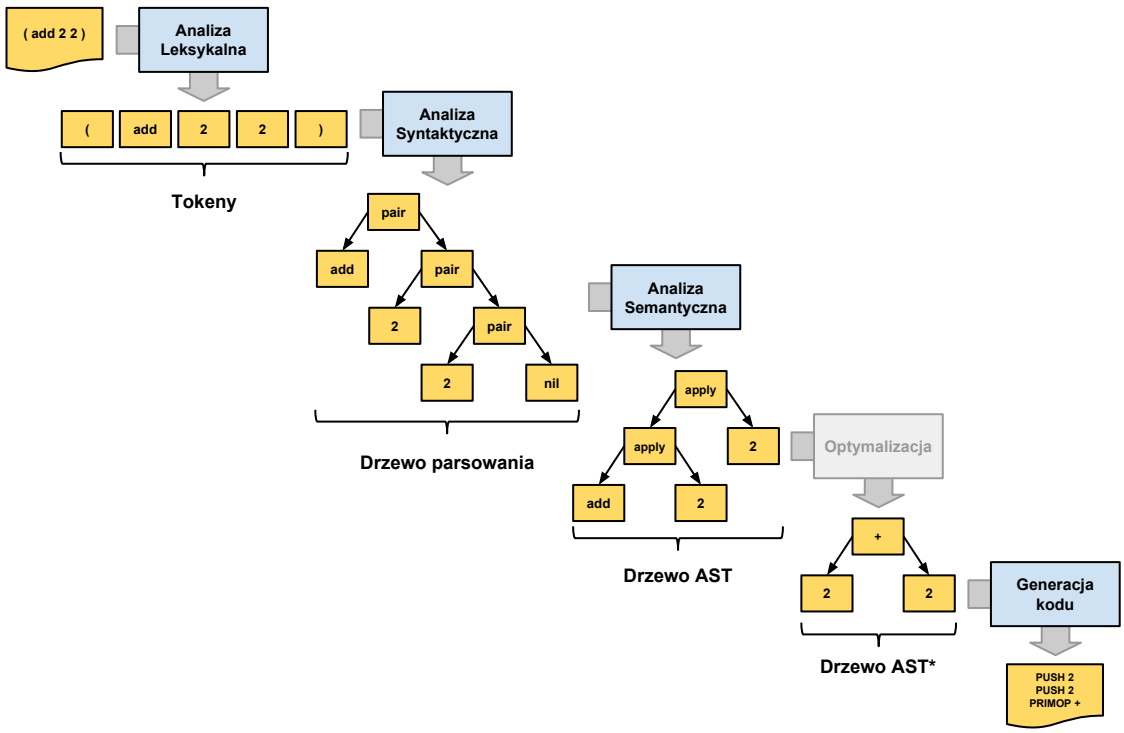
2.2. Kompilacja kodu bajtowego

Język pośredniej reprezentacji programów jest wygodnym medium do zapisu algorytmów, jednak wymaga on uprzedniego skompilowania do kodu bajtowego, który jest

akceptowany przez maszynę wirtualną ThesisVM.

Ponieważ kompilacja kodu nie jest *stricte* tematem pracy, mniej ważne szczegóły implementacji zostały pominięte, a niniejsza sekcja zarysowuje poszczególne fazy kompilacji kodu bajtowego ThesisVM.

Rysunek 2.2 zawiera schemat działania kompilatora kodu bajtowego ThesisVM wraz z przykładami pośrednich reprezentacji kompilowanego kodu w poszczególnych fazach kompilacji.



Rysunek 2.2: Schemat potokowego działania kompilatora kodu bajtowego ThesisVM wraz z przykładami reprezentacji danych poszczególnych faz kompilacji.

Kompilator został zaimplementowany w sposób *potokowy*, to znaczy poszczególne fazy są logicznie odseparowane od siebie i wykonywane jedna po drugiej. Dzięki zastosowaniu leniwych konstrukcji języka `D` [3] wszystkie te fazy odbywają się *jednocześnie* i *na rządanie* a w przypadku wykrycia błędu w danej fazie poprzednie fazy natychmiastowo się kończą, bez konieczności przetworzenia całego zestawu danych, które otrzymały na wejściu.

Pierwszą fazą jest faza analizy leksykalnej, której zadaniem jest przetworzenie *strumienia znaków* kodu źródłowego programu w pośredniej reprezentacji TVMIR do *strumienia tokenów*, czyli elementarnych ciągów znaków będących leksemami języka. Faza ta przeprowadza także walidację składni na poziomie tokenów oraz filtrację niepotrzebnych tokenów (takich jak znaki białe, które nie mają znaczenia w TVMIR).

Drugą fazą jest faza analizy syntaktycznej, której zadaniem jest przetworzenie powstającego leniwie *strumienia tokenów* na *wstępne drzewo parsowania* składające się z prymitywnych konstrukcji języka TVMIR, takich jak listy, symbole i liczby. Faza ta waliduje składnię na poziomie zaawansowanych konstrukcji języka, które dzięki jego homoikoniczności zbudowane są z prymitywniejszych jego konstrukcji.

Trzecią fazą jest faza analizy semantycznej, której zadaniem jest przetworzenie *wstępnego drzewa parsowania* na bardziej abstrakcyjne *drzewo składniowe* (ang. *Abstract Syntax Tree*, *AST*) składające się semantycznie znaczących węzłów, takich jak aplikacja funkcji, wywołania operatorów wbudowanych, czy odwołania do zmiennych. Faza ta waliduje kod na poziomie semantycznym, sprawdzając poprawność wykorzystania różnych konstrukcji języka TVMIR.

Czwartą fazą jest faza optymalizacji, której zadaniem jest transformacja *drzewa składniowego* powstałego w poprzedniej fazie do jego ekwiwalentu działającego szybciej po skompilowaniu. Faza ta obecnie nie wykonuje żadnych interesujących transformacji, jednak istnieje możliwość rozszerzenia jej funkcjonalności w przyszłości (opisane krótko w sekcji 6.4).

Ostatnią, piątą fazą kompilacji jest faza generacji kodu bajtowego akceptowanego przez ThesisVM. Zadaniem tej fazy jest przetworzenie *drzewa składniowego* do *strumienia kodu bajtowego* za pomocą reguł kompilacji zgodnych z wybranym modelem maszyny wirtualnej.

2.3. Interpretacja kodu bajtowego

Istnieje wiele różnych modeli maszyn wirtualnych cechujących się różnymi architekturami interpreterów kodu bajtowego, czy nawet stopniem abstrakcyjności (tak zwane maszyny abstrakcyjne).

Pod względem architektury interpretera kodu bajtowego można wyróżnić trzy główne architektury maszyn wirtualnych:

- architekturę **stosową**, korzystającą ekskluzywnie z jednego lub wielu stosów podczas przetwarzania danych, która charakteryzuje się krótkimi, pod względem zajmowanej pamięci, instrukcjami;
- architekturę **rejestrową**, korzystającą ekskluzywnie z wielu rejestrów podczas przetwarzania danych, która charakteryzuje się instrukcjami przyjmującymi wiele argumentów określających adresy rejestrów maszyny;
- architektury **hybrydowe**, łączące dwa powyższe rozwiązania w różnym stopniu.

Pod względem abstrakcyjności maszyny wirtualne można podzielić na dwie główne grupy:

– **niskopoziomowe**, do których należą maszyny implementujące wyżej wymienione architektury; główną cechą maszyn niskopoziomowych jest obecność stosunkowo nieskomplikowanego kodu bajtowego, który jest przez maszynę interpretowany podczas jej działania;

– **wysokopoziomowe**, które wymagają niestandardowego traktowania kodu programów; na przykład maszyna redukcji grafowych G-machine wykorzystująca grafową naturę kodu języków funkcyjnych do zrównoleglenia jego ewaluacji, opisana szczegółowo w [6].

Od wyboru architektury interpretera kodu bajtowego bardzo często zależą dostępne funkcjonalności docelowego języka programowania. W celu wybrania odpowiedniej architektury należy przeprowadzić szczegółową analizę porządanych funkcjonalności implementowanego języka i możliwości ich zrealizowania w poszczególnych modelach maszyny wirtualnej. Szczegółowa analiza wpływu języka na możliwość jego zaimplementowania w danej architekturze została zawarta w [9] wraz z praktycznymi wskazówkami dotyczącymi implementacji maszyn wirtualnych, co okazało się niezastąpionym źródłem wiedzy pomocnym przy implementacji ThesisVM.

Interpreter kodu bajtowego zaimplementowany w ramach pracy wykorzystuje niskopoziomową architekturę stosową wykorzystującą wiele stosów oraz niewielki zbiór rejestrów i jest zmodyfikowaną wersją interpretera opisanego w [6, rozdział 4]. Szczegółowy opis implementacji został zawarty w dedykowanej mu sekcji 3 pracy.

2.4. Zarządzanie pamięcią

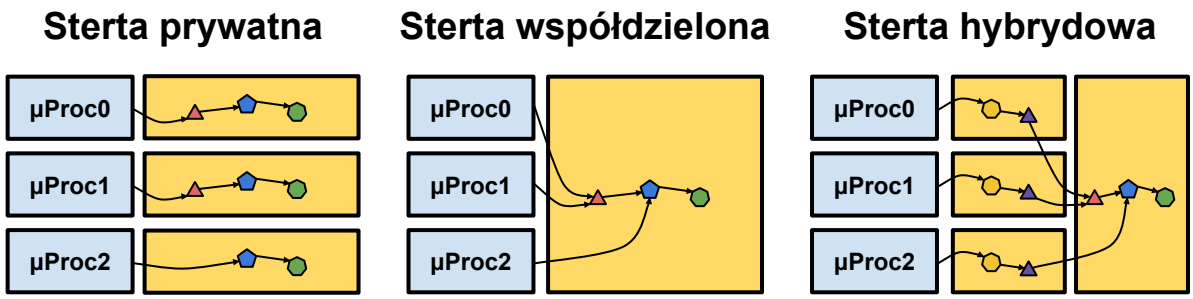
Ważnym aspektem architektury maszyny wirtualnej jest sposób w jaki wykorzystuje ona pamięć operacyjną i rozdziela ją pomiędzy procesy w niej działające, czyli architektura wykorzystania sterty (ang. *heap architecture*).

Rysunek 2.3 przedstawia trzy główne architektury wykorzystania sterty w środowisku wielo-procesowym, gdzie wiele autonomicznych procesów konkuruje o zasób jakim jest pamięć:

– architektura **sterty prywatnej**, charakteryzująca się zupełną separacją pamięci poszczególnych procesów, co prowadzi do konieczności kopiowania obiektów składających się na wiadomości przesyłane pomiędzy nimi;

– architektura **sterty współdzielonej**, charakteryzująca się współdzieleniem jednego obszaru pamięci pomiędzy wszystkie procesy, dzięki czemu wiadomości (a także ich części) mogą być współdzielone przez procesy bez konieczności ich kopiowania;

– architektura **hybrydowa**, mająca za zadanie połączenie zalet obu powyższych rozwiązań przez separację danych lokalnych procesów i współdzielenie danych wiadomości przesyłanych pomiędzy procesami; rozwiązanie to wymaga skomplikowanej, statycznej analizy kodu programów, która nie zawsze może być przeprowadzona.



Rysunek 2.3: Porównanie modeli wykorzystania pamięci maszyn wirtualnych.

Szczegółowa analiza wydajności architektur przedstawionych na rysunku 2.3 w kontekście języka `Erlang`, do semantyki którego ThesisVM jest bardzo zbliżona, została zawarta w [10]. Na podstawie tej analizy zdecydowano się zastosować architekturę sterty współdzielonej, która minimalizuje problem kopiowania pamięci (*ergo*, spełnia nieformalny cel pracy sformułowany w sekcji 1.1) oraz nie wymaga skomplikowanej statycznej analizy kodu programów. Implementacja pozostawia jednak możliwość późniejszej modyfikacji architektury wykorzystania sterty.

Z problemem architektury wykorzystania sterty ściśle związany jest problem wyboru algorytmu alokacji pamięci. W [11] zawarto obszerne zestawienie algorytmów alokacji pamięci, na podstawie, którego zdecydowano się wykorzystać alokatory kaskadowe, *cache*'ujące pamięć zwolnionych obiektów w celu optymalizacji alokacji. Implementacja zastosowanego alokatora została zawarta w sekcji 4.

Ostatnim aspektem zarządzania pamięci maszyny wirtualnej jest kolekcja pamięci obiektow nieosiągalnych. Kolektory obiektów nieosiągalnych można podzielić na dwa typy, ze względu na dane, które analizują:

– kolektory **śledzące** (ang. *tracing-GC*), które okresowo trawersują zbiór obiektów bazowych (ang. *root-set*) celem oznaczenia wszystkich obiektów *osiągalnych* w danej chwili w systemie;

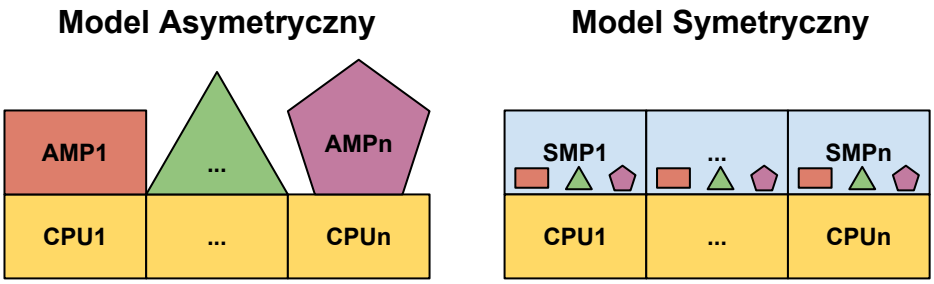
– kolektory **zliczające** (ang. *reference-counting-GC*), które na bieżąco zliczają ilość aktywnych referencji do każdego obiektu i natychmiastowo usuwają obiekty, których licznik referencji osiąga zero, co oznacza, że dany obiekt jest *nieosiągalny*.

Kolektory różnych typów mają bardzo różne charakterystyki wydajnościowe w zależności od architektury wykorzystania sterty zastosowanej w maszynie wirtualnej. Kolektory śledzące przeważnie generują długie pauzy w architekturach współdzielonych, natomiast kolektory zliczające prezentują stały narzut obliczeniowy związany z ciągłą modyfikacją liczników referencji. Oczywiście istnieją dobrze poznane metody optymalizacji obu typu algorytmów [12, 13], które zacierają wszelkie różnice w ich charakterystykach wydajnościowych.

W implementacji ThesisVM zdecydowano się wykorzystać mechanizm automatycznej kolekcji "śmieci", oparty o *leniwe zliczanie referencji*, na podstawie wnikliwej analizy zawartej w [13] oraz w związku z wykorzystaniem podobnych algorytmów kolekcji danych binarnych w standardowej implementacji języka `Erlang`. Rozwiązanie to zostało szczegółowo opisane w sekcji 4, a implementacja umożliwia późniejsze jej rozszerzenie o dodatkowe optymalizacje. Do alternatywnych rozwiązań należą te zaprezentowane w [14] oraz [15].

2.5. Przetwarzanie współbieżne

Systemy współbieżne często realizują model symetrycznego multiprocessingu (*SMP*), którego cechą szczególną jest istnienie wielu identycznych jednostek operacyjnych wykonujących jednakowe zadania na różnych zbiorach danych (`SMPn` na rysunku 2.4).



Rysunek 2.4: Porównanie modeli przetwarzania współbieżnego.

Alternatywnym rozwiązaniem jest model asymetrycznego multiprocessingu (*AMP*) (`AMPn` na rysunku 2.4), gdzie dla różnych typów zadań istnieją dedykowane, wyspecjalizowane jednostki operacyjne, takie jak wątki, lub procesy systemu operacyjnego.

Rozwiązania asymetryczne są interesujące ze względu na zupełnie nowe klasy algorytmów, których implementację umożliwiają (na przykład algorytm zarządzania pamięcią

VCGC [15] wykorzystujący trzy asymetryczne wątki), jednak charakteryzują się skomplikowaniem interakcji poszczególnych jednostek operacyjnych a niejednokrotnie także słabą skalowalnością całego rozwiązania.

Model przetwarzania współbieżnego został już przybliżony przy okazji ogólnego opisu architektury ThesisVM na początku rozdziału. Wybrany został model SMP, który w kontekście maszyny wirtualnej polega na zrównolegleniu wielu interpreterów kodu bajtowego operujących na różnych kontekstach procesów (zbiorach rejestrów i danych znajdujących się na ich stosach) w celu osiągnięcia realnej współbieżności interpretowanego kodu.

Dodatkową zaletą modelu SMP jest jego kompatybilność z Modelem Aktorowym [1], którego głównym założeniem jest istnienie autonomicznych aktorów, którzy reagując na zmiany otoczenia dążą do swoich celów porozumiewając się z innymi aktorami za pośrednictwem wysyłania wiadomości [2]. W modelu SMP zastosowanym w maszynie wirtualnej ThesisVM aktorami są poszczególne procesy, które porozumiewają się za pomocą asynchronicznych wiadomości przesyłanych poprzez nieblokujące kolejki FIFO (ang. *First In First Out*).

Szczegółowy opis implementacji symetrycznego multiprocesora i realizacja Modelu Aktorowego za jego pomocą zostały zawarte w sekcji 5.

3. Interpreter kodu bajtowego

Niniejszy rozdział opisuje implementację interpretera kodu bajtowego ThesisVM. Jak już wspomniano w poprzedniej sekcji, praca implementuje model *Three Instruction Machine*, opisany szczegółowo w [16] oraz [6], wprowadzając do niego szereg modyfikacji.

Three Instruction Machine (TIM) jest nieskomplikowanym modelem maszyny wirtualnej opartym o trzy rejestry, służące do manipulacji danych:

- **IP** - wskaźnik *kodu* następnej instrukcji,
- **Stack** - stos *kontynuacji* składających się z wskaźnika do kodu oczekującego na ewaluację oraz kontekstu, w którym należy ów kod ewaluować,
- **Env** - stos będący obecnym *kontekstem* ewaluacji kodu, który jest analogiczny do leksykalnego zasięgu zmiennych w kodzie źródłowym programu;

oraz trzy bazowe instrukcje przyjmujące od zera do jednego argumentu, które w zupełności wystarczą do implementacji leniwych, funkcyjnych języków programowania:

- **PUSH arg** - tworzy kontynuację argumentu, która umożliwia jej późniejszą ewaluację w odpowiednim kontekście, odkładając ją na stos `Stack`,
- **TAKE** - pobiera kontynuację ze stosu Stack i przenosi ją na stos `Env` rozszerzając obecny kontekst ewaluacji kodu i przygotowując środowisko ewaluacji danej funkcji,
- **ENTER arg** - inicjuje ewaluację kontynuacji wskazywanej przez argument instrukcji odpowiednio modyfikując wartość rejestrów `IP` i `Env`.

Dodatkowo, instrukcje TIM posiadają różne typy adresowania argumentów, które wpływają na sposób interpretacji argumentu instrukcji:

- **VAL** - argument jest traktowany jako konkretna wartość,
- **CODE** - argument jest traktowany jako wskaźnik do konkretnej wartości,
- **ARG** - argument jest traktowany jako indeks stosu `Env`,

Ewaluacja kodu bajtowego TIM przebiega w standardowy sposób. Instrukcje pobierane są z adresu wskazywanego przez wskaźnik następnej instrukcji `IP`, po czym są dekodowane i wykonywane. Dekodowanie instrukcji polega na pobraniu kodu instrukcji oraz sposobu odresowania argumentu. Ostatnią fazą jest ustalenie konkretnej wartości argumentu na podstawie wcześniej ustalonego adresowania.

```
1       PUSH ARG 0  # func
2       ENTER ARG 0 # func
3 func: TAKE
4       PUSH ARG 0  # arg
5       ENTER ARG 1 # func
```

Listing 6: Przykład kodu bajtowego Three Instruction Machine.

Na listingu 6 zawarto przykład kodu bajtowego definicji funkcji `func`, która przyjmuje jeden argument `arg` oraz wywołuje samą siebie z tym argumentem. Przed definicją funkcji (dwie instrukcji przed etykietą `func:`) zawarto także przykładowe wywołanie tej funkcji.

Warto zauważyć, że argumenty przekazywane do funkcji w modelu TIM są ewaluowane *leniwie* - w przykładzie widniejącym na listingu 6 widać, że argument `arg` nigdy nie jest ewaluowany, nawet pomimo faktu, że funkcja `func` przekazuje go do następnego wywołania. Argumenty są ewaluowane dopiero w momencie, gdy maszyna potrzebuje ich konkretnej wartości.

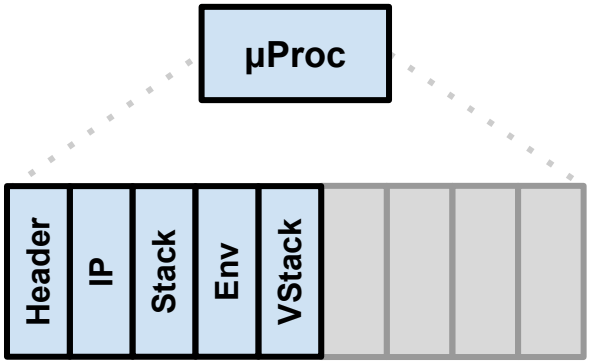
Drugim ważnym spostrzeżeniem jest wsparcie *optymalizacji rekursji ogonowej* modelu TIM - jeśli ostatnią instrukcją kodu ciała funkcji jest wywołanie innej funkcji, to wynikowy kod bajtowy zakończony będzie instrukcją ENTER, która nie wymaga zapisywania adresu powrotnego i tym samym gwarantuje stałą wielkość stosu programu.

Model Three Instruction Machine został wybrany jako podstawa implementacji ThesisVM ze względu na swoją prostotę i niewątpliwe zalety jakie posiada w kontekście implementacji funkcyjnych języków programowania. Istnieje wiele alternatywnych modeli działania maszyn wirtualnych, jak na przykład model *SECD* [17] oraz jego rekursywny ogonowo wariant *TR-SECD* [18], czy bardziej adekwatne dla języków z rodziny Lisp modele opisane w [5] oraz [9].

3.1. Modyfikacje i implementacja modelu TIM

Zaprezentowany powyżej model jest bardzo prosty i pomimo swojej niewątpliwej ekspresywności, maszyna wirtualna go implementująca nie byłaby w stanie uruchamiać programów o praktycznym zastosowaniu. W związku z tym, model został rozszerzony o dodatkowy rejestr wskazujący na stos danych "prostych", nie będących kontynuacjami, a

także szereg instrukcji implementujących podstawowe instrukcje arytmetyczne, logiczne i związane z implementacją Modelu Aktorowego.



Rysunek 3.1: Schemat stanu maszyny wirtualnej.

Na rysunku 3.1 widnieje schemat rejestrów wykorzystywanych przez interpreter kodu bajtowego. Wymienione rejestry wraz z pozostałymi, opisanymi w następnych sekcjach pracy, składają się na kontekst mikroprocesów ThesisVM.

Rejestr **Header** zawiera informacje o typie procesu oraz metadane kolektora obiektów nieosiągalnych. Konteksty mikroprocesów maszyny ThesisVM są dostępne z poziomu kodu źródłowego, ponieważ są obiektami pierwszej klasy (ang. *first-class object*). Więcej informacji na temat zastosowania tego rejestru zostało zawarte w sekcji 3.3 opisującej implementację obiektów złożonych ThesisVM.

Rejestr **IP** służy do przechowywania wskaźnika następnej instrukcji kodu bajtowego. Jest wykorzystywany w dokładnie taki sam sposób, jak analogiczny rejestr modelu TIM. Rejestry **Env** oraz **Stack** podobnie jak rejestr `IP` również wykorzystywane są zgodnie z opisem modelu TIM.

Ostatni rejestr, **VStack** wskazuje na stos przechowujący dane "proste", czyli obiekty, które nie wymagają ewaluacji przez interpreter i mogą być wykorzystywane przez operacje prymitywne. Funkcjonalność tego stosu nie mogła zostać połączona z funkcjonalnością stosu `Stack`, ponieważ część instrukcji polega na homogeniczności danych znajdujących się na stosie `Stack` - jeśli istnienie na tym stosie danych innych niż kontynuacje zostałoby dozwolone, to część instrukcji wymagałaby kosztownego przeszukiwania i modyfikacji stosu.

Implementacja ThesisVM modyfikuje semantykę trzech bazowych instrukcji TIM:

– **NEXT addr arg** - jest to bardziej adekwatnie nazwany analog instrukcji `PUSH` podstawowego modelu TIM, w zależności od typu adresowania argument instrukcja ta tworzy i umieszcza na stosie `Stack` samo-ewaluującą do wartości argumentu

kontynuację (`addr` równe `VAL`), kontynuację składającą się z obecnego kontekstu i wartości wskazywanej przez argument (`addr` równe `CODE`) lub wartść kontynuacji znajdującej się na stosie `Env` (wartość `addr` równa `ARG`);

– **TAKE** - podobnie jak w przypadku modelu bazowego, pobiera jedną kontynuację ze stosu kontynuacji `Stack` i umieszcza ją w obecnym kontekście ewaluacji `Env`;

– **ENTER addr arg** - w zależności od typu adresacji argumentu odpowiednio modyfikuje wartości rejestrów `Env` oraz `IP` podstawiając wartość `IP` na wartość argumentu (`addr` równe `CODE`), lub ewaluując kontynuację znajdującą się na stosie `Env` (wartość `addr` równa `ARG`);

oraz wprowadza kilka nowych instrukcji służących do obsługi dodatkowego rejestru i operacji prymitywnych z nim związanych:

– **PUSH arg** - jest to prosta instrukcja, której jedynym zadaniem jest umieszczenie argumentu na stosie `VStack`;

– **PRIMOP arg** - wykonuje operację prymitywną o identyfikatorze równym wartości argumentu. Więcej informacji o implementacji operacji prymitywnych zawarto w sekcji 3.5;

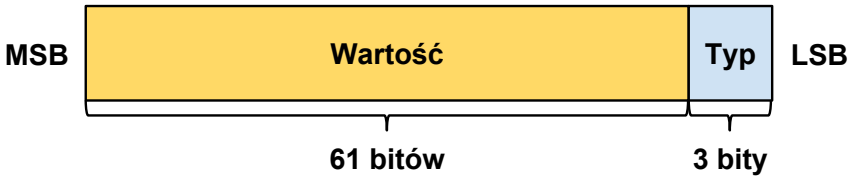
– **COND arg** - jest to instrukcja warunkowa, która sprawdza wartość znajdującą się na wierzchu stosu `VStack` i w zależności od jej wartości wybiera jedną z dwóch gałęzi kodu wskazywanych przez argument i ustawia jej wartość jako nową wartość rejestru `IP`;

– **SPAWN arg** - jest to instrukcja związana z implementacją Modelu Aktorowego, tworzy ona nowy kontekst mikroprocesu ThesisVM i aranżuje ewaluację kontynuacji znajdującej się na stosie `Env` pod indeksem równym wartości argumentu instrukcji (`arg`) przekazując jej jako parametr wartość znajdującą się na szczycie stosu `VStack`. Tak zaaranżowany kontekst mikroprocesu jest następnie dodawany do kolejki uruchomieniowej jednego z symetrycznych multiprocesorów wybranego zgodnie z zasadami równoważenia obciążenia (opisanymi w sekcji 5.2);

– **HALT** - instrukcja ta usypia proces na czas nieokreślony efektywnie kończąc jego działanie. Tak zatrzymany proces następnie podlega kolekcji przez kolektor obiektów nieosiągalnych, ponieważ mogą istnieć referencje nań wskazujące, które są wykorzystywane przez inne procesy.

3.2. Implementacja obiektów prostych

Dane programów w maszynie ThesisVM reprezentowane są za pomocą dwóch rodzajów obiektów - obiektów "prostych" oraz obiektów złożonych. Rysunek 3.2 zawiera schemat reprezentacji obiektów prostych, które należą do jednego z trzech wspieranych typów podstawowych: `POINTER`, `FLOATING` lub `INTEGER`.



Rysunek 3.2: Schemat reprezentacji obiektów prostych ThesisVM.

Implementacja przechowuje dane obiektów prostych w strukturze o wielkości jednego słowa procesora (**64** bity w obecnej implementacji maszyny wirtualej przystosowanej do architektury $\mathbf{x86}_{64}$), która zapewnia dostęp do dwóch pól identyfikujących odpowiednio 61-bitową wartość przechowywaną w strukturze oraz 3-bitowy typ, do którego owa wartość należy.

Wartość przechowywane są wraz z informacją o ich typie, w celu umożliwienia implementacji języków dynamicznie typowanych oraz ułatwienia pracy kolektora obiektów nieosiągalnych - dzięki informacji o typie może on precyzyjnie określić, czy dany obiekt jest referencją, czy też nie.

Typy obiektow prostych przechowywane są w trzech najmniej znaczących bitach (ang. *least significant bits*, *LSB*) reprezentacji, umożliwiając implementację ośmiu różnych typów podstawowych, zgodnie ze szczegółowym opisem zawartym w [19]. Reprezentacja taka posiada szereg zalet począwszy od kompaktowości, przez brak konieczności alokacji pamięci dla typów podstawowych, kończąc na wielu ciekawych optymalizacjach, które umożliwia.

Na przykład, jeśli alokator maszyny wirtualnej wymusi *wyrównywanie pamięci* (ang. *alignment*) do wielkości słowa procesora, to trzy najmniej znaczące bity (na architekturze 64-bitowej) reprezentacji wskaźników zawsze będą zerowe. W związku z tym, zerem można reprezentować typ wskaźnikowy obiektow prostych ThesisVM (typ `POINTER`), co umożliwia wykorzystywanie ich reprezentacji bezpośrednio, bez konieczności przeprowadzenia operacji bitowego maskowania.

Podobne optymalizacje mogą zostać zastosowane w przypadku reprezentacji obiektów numerycznych. Na przykład, w celu dodania dwóch liczb całkowitych (o typie `INTEGER`)

można posłużyć się ostatnią zależnością zaprezentowaną na listingu 7 zamiast wielokrotnie wykorzystywać kosztowne operacje `tag` i `untag`, które realizują przejścia pomiędzy reprezentacją wewnętrzną obiektów maszyny wirtualnej a reprezentacją języka jej implementacji.

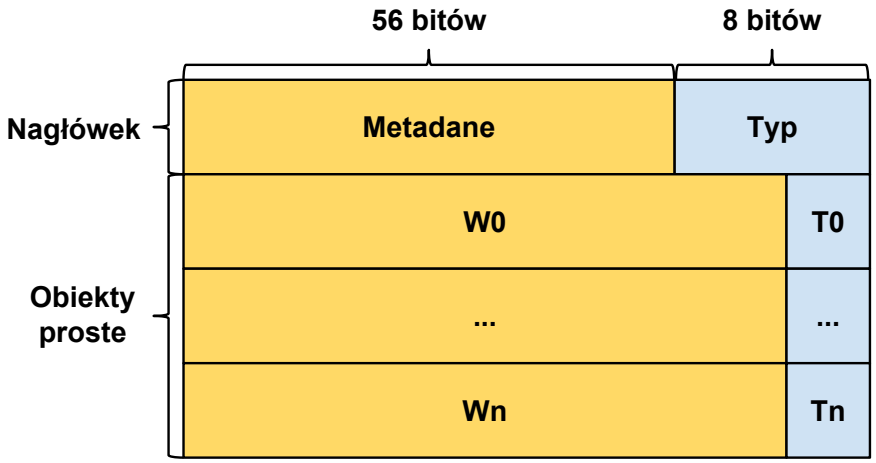
```
1 result = tag(TVMValue.INTEGER, untag(a) + untag(b));
2 result = a.rawValue + b.rawValue - TVMValue.INTEGER;
```

Listing 7: Optymalizacja dodawania liczb całkowitych.

Wiele ciekawych optymalizacji związanych ze sposobem reprezentacji typów obiektów zostało zawarte w [19].

3.3. Implementacja obiektów złożonych

Ważnym elementem każdego języka programowania są złożone struktury danych takie jak listy lub drzewa. Rysunek 3.3 prezentuje schemat reprezentacji obiektów złożonych ThesisVM, które służą do budowy takich struktur danych.



Rysunek 3.3: Schemat reprezentacji obiektów złożonych ThesisVM.

Obiekty te składają się z wielu słów procesora ułożonych kolejno w pamięci. Pierwszym słowem składającym się na obiekt złożony jest jego **nagłówek**, który zawiera między innymi ośmio-bitowy identyfikator typu obiektu oraz metadane kolektora obiektów nieosiągalnych.

Podobnie jak w przypadku obiektów prostych, informacja o typie jest wykorzystywana do implementacji języków dynamicznie typowanych oraz w celu ułatwienia pracy kolektora "śmieci". Identyfikator typu obiektu jest jednak znacznie większy pozwalając na reprezentację 256 różnych wartości, a co za tym idzie 256 różnych typów. Obecna implementacja nie wykorzystuje potencjału dłuższego identyfikatora typu w pełni, ale w przyszłości może zostać rozwinięta, na przykład w celu umożliwienia definiowania nowych typów danych.

Dodatkowym atutem stosowania nagłówka jest fakt, że wszystkie obiekty złożone ThesisVM mogą być traktowane w jednolity sposób za pośrednictwem wskaźników do owego nagłówka. Informacja o typie obiektu w nim zawarta może zostać wykorzystana do łatwego określenia faktycznej struktury obiektu znajdującego się w pamięci. Metoda ta została szeroko opisana w [19] i jest standardowym rozwiązaniem wielu maszyn wirtualnych.

Po nagłówku występują właściwe dane w postaci `n` obiektów prostych, gdzie `n` jest dowolną liczbą naturalną. Ich zawartość `Wn` oraz typy `Tn` zależą w dużej mierze od typu całego obiektu złożonego, ale w ogólności podlegają wszystkim zasadom, którym podlegają obiekty proste.

Obecna implementacja definiuje 4 typy obiektów złożonych: `PAIR`, `CLOSURE`, `SYMBOL` oraz `UPROC`.

Pary składają się z nagłówka oraz dwóch obiektów prostych odpowiadających odpowiednio pierwszemu i drugiemu elementowi pary. Pary są wykorzysytwane do implementacji list, które z kolei są podstawowymi strukturami danych języka TVMIR, podobnie jak w innych językach z rodziny Lisp.

Obiekty funkcyjne, zwane czasami domknięciami leksykalnymi (ang. *closures*), służą do reprezentowania skompilowanych funkcji TVMIR oraz **kontynuacji** będących podstawą działania modelu TIM. W obu przypadkach obiekty funkcyjne składają się z dwóch obiektów prostych odpowiadających kolejno rejestrowi `IP` oraz stosowi `Env`.

Różnica pomiędzy skompilowanymi funkcjami oraz kontynuacjami sprowadza się do zestawu instrukcji zawartego w komponencie `IP` obiektu funkcyjnego - funkcje przyjmujące parametry wymagają pobrania ich wartości za pomocą instrukcji `TAKE`.

Symbole również składają się z nagłówka oraz dwóch obiektów prostych, które oznaczają odpowiednio wskaźnik na zewnętrzny fragment pamięci zawierający tekstową reprezentację symbolu i długość owej reprezentacji. Reprezentacja symboli została pomyślana w taki sposób, by umożliwiała bezpośrednie mapowania reprezentacji tekstowej na dostępny w języku D typ danych `string`, co znacząco ułatwia obsługę symboli w implementacji maszyny wirtualnej.

Ostatnim dostępnym typem danych ThesisVM jest **deskryptor mikroprocesu**. Deskryptory te są obiektami pierwszej klasy, co oznacza, że są w pełni dostępne dla użytkownika ThesisVM. Konteksty mikroprocesów składają się z nagłówka oraz ośmiu obiektów

prostych, z których cztery pierwsze odpowiadają opisanym w poprzedniej sekcji rejestrom maszyny wirtualnej, a cztery następne zawierają dane wykorzystywane przez pozostałe moduły maszyny wirtualnej. Obiekty te zostały opisane w sekcjach 4 oraz 5.

3.4. Implementacja i obsługa kodu bajtowego

Instrukcje kodu bajtowego ThesisVM dzielą reprezentację z obiektami złożonymi. Podobnie jak pary składają się z nagłówka obiektu oraz dwóch obiektów prostych, z których pierwszy określa identyfikator instrukcji oraz sposób adresowania argumentu, a drugi przechowuje wartość argumentu instrukcji. Dostępne instrukcje kodu bajtowego zostały już opisane w sekcji 3.1.

Reprezentacja kodu bajtowego wykorzystywana obecnie w maszynie wirtualnej niestety jest sub-optymalna. Alternatywnym rozwiązaniem mogło by być zastosowanie reprezentacji opartej o obiekty proste polegającej na przechowywaniu identyfikatorów instrukcji w najbardziej znaczącym bajcie obiektu.

Dekodowanie argumentu instrukcji wymagałoby wówczas jedynie przeskalowania pozostałych bajtów obiektu prostego celem odtworzenia rzeczywistej jego wartości, lub w przypadku niewielkich liczb całkowitych i typu wskaźnikowego, jedynie zastosowania masek bitowych.

Instrukcje pobierane są ze *strumienia kodu bajtowego* wskazywanego przez rejestr `IP`. W obecnej implementacji strumień kodu bajtowego zrealizowany jest jako lista pojedynczo wiązana zbudowana z obiektów złożonych ThesisVM - par.

Reprezentacja ta została wybrana ze względu na charakter języka TVMIR (jest to język z rodziny Lisp) oraz przez wzgląd na podobieństwo to obsługi stosów `Stack`, `Env` oraz `VStack`, które również zostały zrealizowane w oparciu o listy pojedynczo wiązane. W przyszłości implementacja ta może zostać zastąpiona rozwiązaniem szybszym, niekoniecznie opartym o listy (więcej informacji na ten temat zawarto w sekcji 6.4).

Po przechwyceniu pierwszego elementu listy wskazywanej przez rejestr `IP` następuje ustalenie identyfikatora instrukcji, sposobu adresowania argumentu oraz samej wartości argumentu instrukcji. Następnie interpreter ewaluuje instrukcję zgodnie z regułami opisanymi w sekcji 3.1.

3.5. Implementacja operacji prymitywnych

Operacje wbudowane, takie jak arytmetyka, czy funkcje związane z implementacją Modelu Aktorowego zostały zrealizowane w oparciu o rejestr `VStack` - argumenty operacji prymitywnych są pobierane ze stosu a wartości przez nie zwracane są nań odkładane. Listing 8 zawiera ogólny algorytm implementacji operacji prymitywnych ThesisVM.

```
arg0 = typecheck(t0, pop(uProc.vstack));
// ...
argN = typecheck(tN, pop(uProc.vstack));

// Obliczenia charakterystyczne dla danej operacji.
result = compute(arg0, ..., argN);

push(uProc.vstack, result);
```

Listing 8: Ogólny algorytm implementacji operacji prymitywnych ThesisVM.

Instrukcja wykonujące operacje primitywne, `PRIMOP id`, wykorzystuje metodę LUT (ang. *look-up table*) w celu skorelowania identyfikatora operacji prymitywnej i fragmentu kodu odpowiedzialnego za jej wykonanie - wykonanie tej instrukcji polega na przekazaniu przepływu sterowania do odpowiedniej funkcji znajdującej się w *tablicy operacji prymitywnych*.

Dostępne operacje prymitywne to w dużej mierze podstawowe operacje arytmetyczno-logiczne oraz funkcje typowe dla języków z rodziny Lisp, takie jak `cons`, `car`, czy `cdr` (więcej informacji można znaleźć w [5]). Lista wszystkich dostępnych operacji prymitywnych została zawarta w dodatku B.

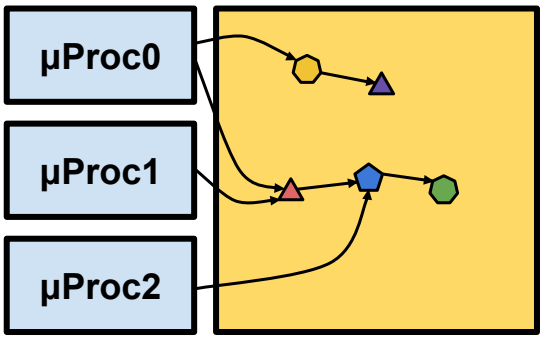
Ponownie uwagę można zwrócić na podobieństwo traktowania operacji prymitywnych do języka `Core Erlang` [8]. Podobieństwo to nie jest przypadkowe, a wybrany sposób reprezentacji i działania operacji prymitywnych został zaimplementowany w taki sposób, by w przyszłości umożliwić łatwe rozszerzenie implementacji maszyny wirtualnej ThesisVM oraz wsparcie pełni języka `Core Erlang`.

4. Model zarządzania pamięcią

Niniejsza sekcja opisuje implementację modelu zarządzania pamięcią zastosowanego w maszynie wirtualnej ThesisVM, na który składa się architektura wykorzystania pamięci, algorytm alokacji obiektów oraz algorytm kolekcji obiektów nieosiągalnych. Wstępny opis wybranego modelu pamięci oraz motywacja tego wyboru zostały zawarte w sekcji 2.4.

4.1. Architektura współdzielonej sterty

Na rysunku 4.1 zawarto schemat architektury sterty wykorzystanej w ThesisVM. Jest to architektura współdzielonej sterty, w której każdy z mikroprocesów alokuje obiekty na własny użytek. Obiekty te lub ich części mogą być następnie współdzielone pomiędzy mikroprocesami w wyniku przesyłania wiadomości.



Rysunek 4.1: Model współdzielonej pamięci ThesisVM.

Kluczową zaletą wybranej architektury wykorzystania pamięci jest brak konieczności kopiowania danych przesyłanych pomiędzy mikroprocesami. Ponieważ wszystkie dane są zaalokowane w jednej puli pamięci, przesyłanie wiadomości sprowadza się jedynie do przekazania wkaźników do owych wiadomości pomiędzy mikroprocesami, co jest operacją o złożoności czasowej i pamięciowej rzędu *O(1)*.

Alternatywne rozwiązania polegające na separacji danych mikroprocesów poprzez wykorzystanie osobnych puli pamięci dla każdego mikroprocesu nie posiadają tej zalety i wymagają kosztownego kopiowania wszystkich przesyłanych wiadomości, co w przypadku wzmożonej komunikacji pomiędzy mikroprocesami powoduje znaczną degradację wydajności.

Kolejną ważną zaletą wybranej architektury jest łatwość jej implementacji z wykorzystaniem sterty procesu maszyny wirtualej, zarządzanej przez system operacyjny, na którym jest ona uruchomiona. Pozwala to na wykorzystanie gotowego, standardowego interfejsu alokacji udostępnianego przez system operacyjny.

Do niewątpliwych wad zastosowanej architektury należą wyzwania, jakie stawia ona algorytmom kolekcji obiektów nieosiągalnych. Zadaniem tych algorytmów jest automatyczne zwolnienie nieużywanej pamięci mikroprocesów, co w wyniku współdzielenia danych jest znacznie utrudnione i może prowadzić do długich przerw w działaniu maszyny wirtualnej przeznaczonych na cykle kolekcji "śmieci".

Kolejną wadą architektury współdzielonej sterty jest fakt, że pamięć mikroprocesu nie może zostać od razu i w całość zwolniona po zakończeniu jego działania. Ponieważ dane mogą być wciąż wykorzystywane przez inne mikroprocesy, po zakończeniu działania jednego z nich musi zostać wykonany pełen cykl kolekcji obiektów nieosiągalnych.

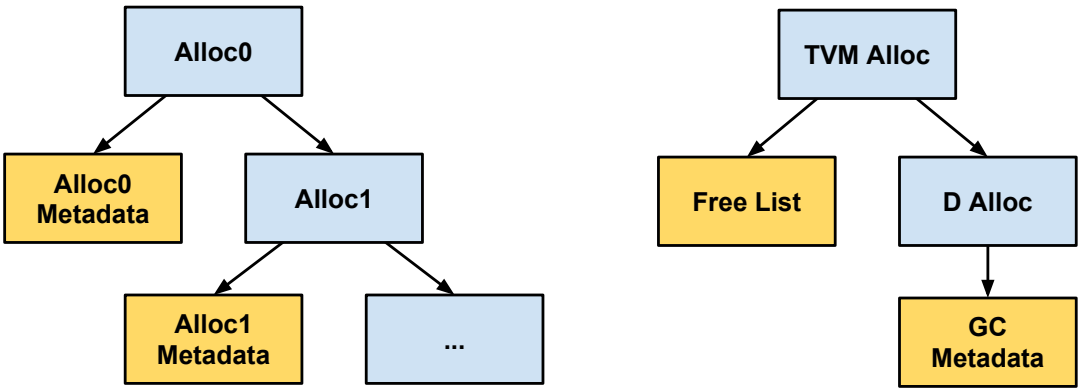
Szczegółowa analiza zalet i wad architektury współdzielonej sterty w kontekście implementacji języka `Erlang` została zawarta w [10].

4.2. Implementacja alokatora obiektów

Maszyna wirtualna ThesisVM wykorzystuje algorytm kaskadowych alokatorów polegający na kompozycji wielu algorytmów alokacji obiektów wraz z wykorzystywanymi przez nie metadanymi w taki sposób, by umożliwić algorytmowi na danym poziomie odwoływanie się do algorytmu na niższym poziomie. Schematyczna reprezentacja takiego rozwiązania została zawarta na rysunku 4.2.

W momencie, gdy algorytm na danym poziomie ustali, że nie jest w stanie obsłużyć żądania użytkownika przepływ sterowania zostanie przekazany do algorytmu leżącego poziom niżej, gdzie obsługa żądania będzie kontynuowana. Dzięki takiemu rozwiązaniu możliwe jest zaimplementowanie szeregu ciekawych algorytmów alokacji i dowolne ich komponowanie.

Obecnie, implementacja ThesisVM wykorzystuje dwu-poziomowy alokator składający się z algorytmów **TVM Alloc** oraz **D Alloc**, który został zaprezentowany ra rysunku 4.2.

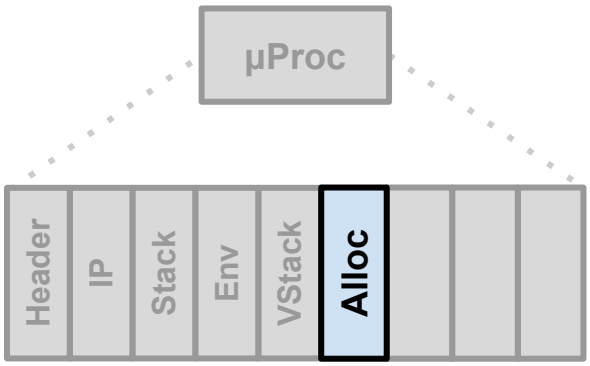


Rysunek 4.2: Schemat kaskadowych alokatorów wykorzystanych w ThesisVM.

D Alloc jest standardowym interfejsem alokatora języka `D`, który wykorzystuje metadane kolektora obiektów nieosiągalnych języka `D` i udostępnia wyrównaną do słowa procesora (8 bajtów na architekturze x86_64) pamięć przezeń zarządzaną. Wybór tego algorytmu zostanie umotywowany w sekcji 4.4.

Alternatywnym rozwiązaniem dla `D Alloc` byłoby wykorzystanie interfejsu manualnego zarządzania pamięcią poprzez wykorzystanie funkcji `malloc` i `free` ze standardowej biblioteki języka `C`, która wchodzi w skład standardowej biblioteki języka `D`.

TVM Alloc jest dodatkowym algorytmem alokacji zbudowanym w oparciu o interfejs `D Alloc`, który dodatkowo zapewnia buforowanie (ang. *caching*) pamięci za pomocą **listy niedawno zwolnionych obiektów**, która jest przeszukiwana w pierwszej kolejności podczas żądania alokacji.



Rysunek 4.3: Schemat rejestrów wymaganych przez implementację alokatora obiektów.

Każdy mikroproces posiada własną listę niedawno zwolnionych obiektów (rysunek 4.3), co zapewnia lepsze wykorzystanie pamięci przez *zwiększenie lokalności referencji* - obiekty

zwalniane podczas działania mikroprocesu trafiają na listę niedawno zwolnionych obiektów i bardzo szybko są wykorzystywane powtórnie bez konieczności odwoływania się do alokatorów z niższych poziomów.

Wykorzystanie listy niedawno zwolnionych obiektów do buforowania alokacji umożliwia także zaimplementowanie zupełnie nowej klasy algorytmów kolekcji obiektów nieosiągalnych w oparciu o *leniwe cykle kolekcji*. Algorytm taki został opisany w następnej sekcji.

Zagadnienie alokacji pamięci jest bardzo rozległe i w kontekście języków programowania zależy od wielu różnych czynników, takich jak charakterystyki zużycia pamięci konkretnych programów, wielkości alokowanych obiektów, czy czasy ich życia. Przegląd [11] zawiera szczegółową analizę wydajności wielu różnych algorytmów alokacji pamięci w warunkach symulowanych oraz dla rzeczywistych programów, co było niezastąpionym źródłem wiedzy pomocnym przy wyborze i implementacji algorytmu alokacji obiektów maszyny wirtualnej ThesisVM.

4.3. Kolekcja obiektów nieosiągalnych

Różne podejścia do problemu automatycznego zwalniania nieużywanej pamięci zostały już opisane w sekcji 2.4, której konkluzją był wybór algorytmu **zliczania referencji** jako głównego algorytmu kolekcji obiektów nieosiągalnych.

Algorytm zliczania referencji polega na przechowywaniu i modyfikacji liczników aktywnych referencji wskazujących na dany obiekt w pamięci. Liczniki te przechowywane są w nagłówkach obiektów, dzięki czemu algorytm jest w stanie zdecydować, czy konkretny obiekt jest w dalszym ciągu w użyciu jedynie na podstawie wkaźnika na jego nagłówek.

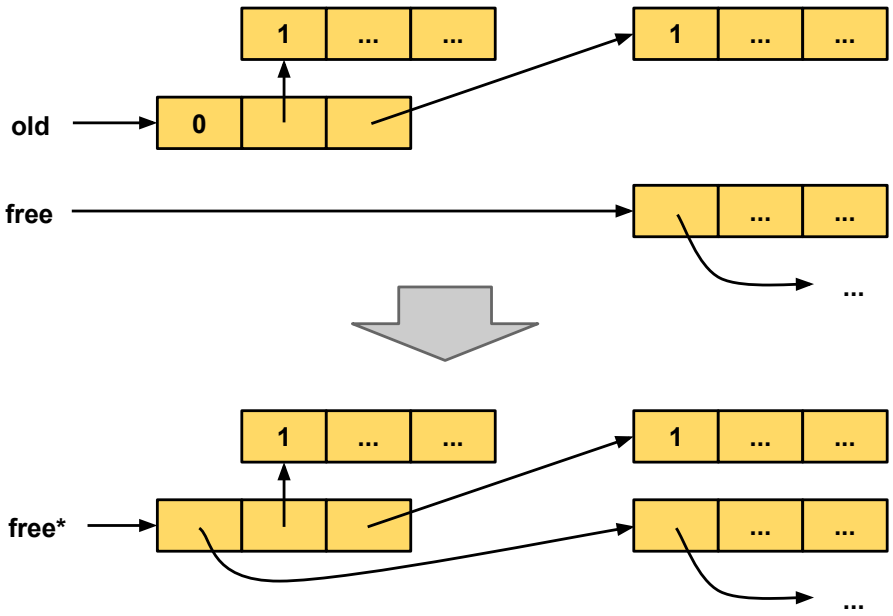
Liczniki modyfikowane są podczas tworzenia nowych i usuwania istniejących referencji - stworzenie nowej referencji do konkretnego obiektu powoduje inkrementację jego licznika referencji, natomiast usunięcie istniejącej referencji powoduje jego dekrementację. W momencie, gdy wartość licznika osiągnie zero obiekt jest uznawany za *nieosiągalny* i następuje zwolnienie jego pamięci ora usunięcie wszystkich referencji wchodzących w jego skład.

Wariant algorytmu zaimplementowany w maszynie wirtualnej ThesisVM to tak zwane **leniwe zliczanie referencji**, którego implementacja jest możliwa dzięki zastosowaniu alokatora buforującego zwalniane obiekty. Algorytm ten polega na opóźnieniu usuwania referencji wchodzących w skład usuwanego obiektu do czasu aż jego pamięć zostanie powtórnie wykorzystana.

Modyfikacja ta jest bardzo prosta i pozwala osiągnąć dużo lepsze charakterystyki czasowe kolekcji obiektów nieosiągalnych kosztem zwiększenia ogólnego zużycia pamięci - obiekty nie są zwalniane natychmiastowo, a dopiero przy następnej alokacji. Dokładne

badanie wpływu opisanej modyfikacji algorytmu zliczania referencji na zużycie pamięci zostało przedstawione w [20].

Rysunek 4.4 zawiera schemat dealokacji obiektu z wykorzystaniem opisanego powyżej algorytmu.

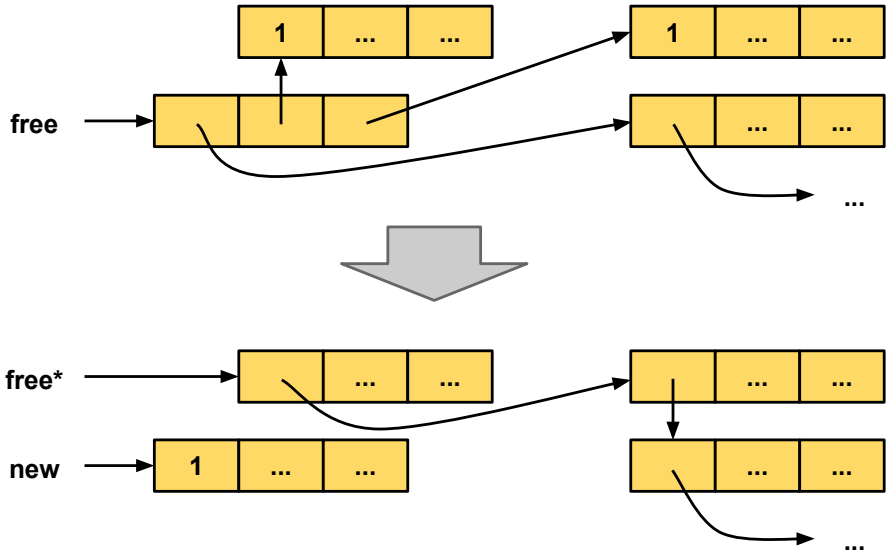


Rysunek 4.4: Schemat działania zwalniania pamięci obiektów.

Obiekt `old`, którego licznik referencji osiągnął wartość 0 w wyniku usunięcia ostatniej aktywnej referencji zostaje przeniesiony do listy niedawno zwolnionych obiektów `free`. Lista ta wykorzystuje ów licznik do przechowywania wskaźnika na następny element listy, dzięki czemu możliwe jest zachowanie danych obiektu bez zmian w celu późniejszego, leniwego ich usunięcia podczas następnej alokacji (rysunek 4.5).

Alokacja nowego obiektu `new` polega na pobraniu pierwszego elementu listy niedawno zwolnionych obiektów `free` oraz usunięciu wszystkich referencji wchodzących w jego skład. Pociąga to za sobą dekrementację liczników referencji obiektów, na które owe referencje wskazują i ewentualną dealokację tych obiektów, jeśli ich liczniki osiągnęły wartość 0. W przypadku, gdy lista `free` jest pusta tworzony jest zupełnie nowy obiekt, który jest dodawany do wspólnej puli pamięci.

W przypadku języków programowania wspierających przetwarzanie współbieżne algorytm dodatkowo komplikuje konieczność wykorzystywania *operacji atomowych* na licznikach referencji obiektów, które mogą być modyfikowane jednocześnie przez wiele wątków sprzętowego procesora. Dodatkowo ważne jest wykorzystanie *barier pamięci*, które uniemożliwiają zmiany kolejności wykonywania operacji na pamięci, co jest częstym zabiegiem optymalizacyjnym w nowoczesnych procesorach.



Rysunek 4.5: Schemat działania alokacji pamięci nowych obiektów.

Konieczność stosowania operacji atomowych i barier pamięci powoduje nieznaczny spadek wydajności maszyny wirtualnej, który jednak jest wart odnotowania. Implementacja kolektora "śmieci" ThesisVM wykorzystuje wbudowane w język `D` kwalifikatory typów `shared`, które gwarantują stosowanie operacji atomowych i barier pamięci w stategicznych miejscach.

Implementacja optymalizuje także modyfikacje liczników referencji przez ich opóźnienie lub całkowite wyeliminowanie (arg. *deferred reference counting*), jeśli nie są konieczne - na przykład w przypadku transferu referencji pomiędzy dwoma obiektami.

Jest to potencjalnie niebezpieczna technika wymagająca manualnego dekrementowania i inkrementowania liczników referencji za pomocą funkcji `use` oraz `free`. Alternatywnym rozwiązaniem jest wykorzystanie inteligentnych wskaźników (ang *smart pointers*), które gwarantują deterministyczną inkrementację i dekrementację liczników.

Więcej możliwych usprawnień algorytmu kolekcji obiektow nieosiągalnych za pomocą zliczania referencji zostało przedstawionych w [13] oraz [12].

4.4. Kolekcja obiektów cyklicznych

Dużą wadą kolektorów zliczających referencje jest ich słabe wsparcie dla zwalniania pamięci struktur cyklicznych, które nie są dłużej użytkowane w programie. Sytuacja ta ma miejsce, gdy pewna struktura danych zawiera referencje do siebie samej, co w efekcie uniemożliwia jej dealokację, ponieważ jej licznik referencji nigdy nie osiąga wartości zerowej.

W maszynie wirtualnej ThesisVM problem ten objawia się przy wykorzystywaniu wbudowanego operatora `self`, który zwraca referencję na obecnie działający mikroproces. Referencja ta może zostać zapisana w stanie procesu efektywnie tworząc cykl i uniemożliwiając kolekcję danych procesu po zakończeniu jego działania.

Aby temu zaradzić implementacja alokatora wykorzystuje wbudowany w język `D` kolektor śledzący, który jest uruchamiany co pewien interwał w celu dealokacji struktur cyklicznych, takich jak mikroprocesy referujące same siebie.

Implementacja alokatora jest jednak na tyle generyczna, by umożliwić w przyszłości zaimplementowanie alternatywnego, zapasowego kolektora śledzącego, który w przeciwieństwie do kolektora języka `D` mógłby wykorzystywać dane o typach obiektów ThesisVM w celu prowadzenia precyzyjniejszych i szybszych kolekcji.

5. Model przetwarzania współbieżnego

Niniejsza sekcja opisuje implementację modelu przetwarzania współbieżnego zastosowanego w maszynie wirtualej ThesisVM. Model ten przewiduje wykorzystanie symetrycznego multiprocessingu oraz implementację Modelu Aktorowego interakcji mikroprocesów. Wstępny opis wybranego modelu zawarto w sekcji **??**.

Model Aktorowy [1] został wybrany przez wzgląd na jego relatywne nieskomplikowanie i wielką ekspresywność, zwłaszcza w kontekście funkcyjnych języków programowania. Model ten zakłada istnienie autonomicznych *aktorów*, którzy porozumiewają się za pomocą przekazywanych asynchronicznie *wiadomości*, co bardzo łatwo można przetłumaczyć na istnienie wielu działających konkurencyjnie mikroprocesów.

Każdy mikroproces po otrzymaniu wiadomości może na nie reagować poprzez zmianę swojego wewnętrznego *zachowania*, wygenerowanie skończonej liczby *nowych wiadomości*, wysyłanych konkurencyjnie do innych mikroprocesów, lub poprzez stworzenie skończonej liczby *nowych aktorów* - uruchomienie dodatkowych mikroprocesów ThesisVM. Dokładny opis założeń Modelu Aktorowego został zawarty w [2].

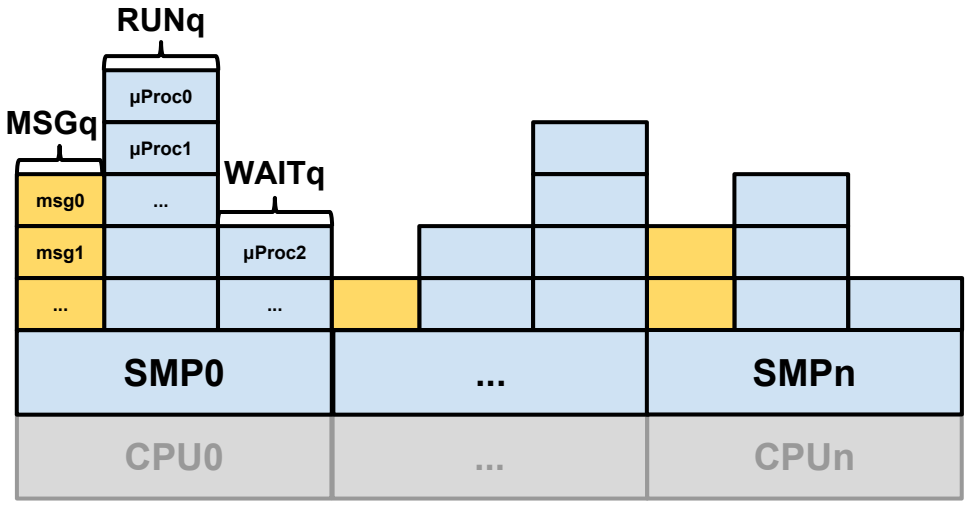
5.1. Implementacja symetrycznego multiprocessingu

Implementacja wykorzystuje wątki systemu operacyjnego, na którym uruchomiona jest maszyna wirtualna ThesisVM w celu zrównoleglenia działania wielu interpreterów kodu bajtowego. Rysunek 5.1 zawiera schematyczną reprezentację struktury symetrycznych multiprocesorów (SMP).

SMP komunikują się ze sobą poprzez wiadomości kontrolne przekazywane za pośrednictwem kolejki wiadomości `MSGq`, efektywnie wykorzystując Model Aktorowy. Implementacja taka jest więc bardzo skalowalna i umożliwia dowolną zmianę ilość uruchomionych jednostek także podczas działania maszyny wirtualnej.

Obecnie wiadomości kontrolne wykorzystywane są przy tworzeniu nowych mikroprocesów, ale implementacja może zostać w przyszłości rozszerzona w celu umożliwienia stosowania zaawansowanych algorytmów równoważenia obciążenia, strategii uruchomie-

niowych mikroprocesów a także propagacji i kolekcji danych diagnostycznych działania symetrycznych multiprocesorów. Więcej informacji o przyszłych kierunkach rozwoju projektu zostało zawarte w sekcji 6.4.



Rysunek 5.1: Schemat symetrycznego multiprocesingu ThesisVM.

Każdy z symetrycznych multiprocesorów (SMP) zarządza szeregiem struktur danych wykorzystywanych do przechowywania kontekstów mikroprocesów oraz harmonogramowania (ang. *scheduling*) ich interpretacji (rysunek ref:fig:tvm-smp).

Struktury te to kolejki priorytetowe zaimplementowane w oparciu o, gwarantujące logarytmiczną złożoność wszystkich operacji, **drzewa czerwono-czarne**:

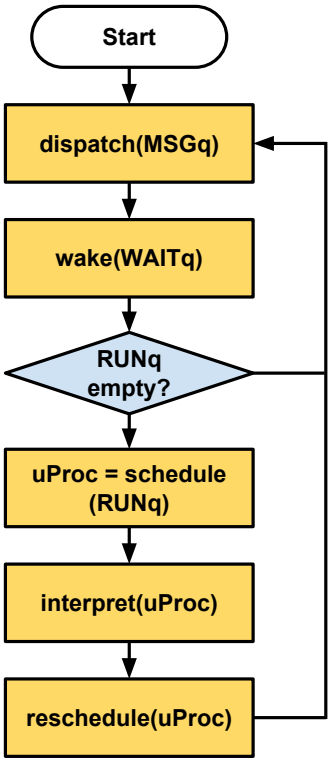
– `RUNq` - kolejka ustalającą kolejność uruchamiania aktywnych mikroprocesów,

– `WAITq` - kolejka przechowująca mikroprocesy będące w uśpieniu, ułatwiająca ustalenie kolejności ich powrotu do działania.

SMP działają według uproszczonego algorytmu zaprezentowanego na diagramie 5.2. Po starcie maszyny wirtualnej wszystkie SMP przechodzą do fazy obsługiwania wiadomości kontrolnych, `dispatch`. Podczas tej fazy wiadomości otrzymane asynchronicznie przez SMP są analizowane i obsługiwane tak szybko, jak to tylko możliwe.

Następną fazą jest faza `wake`, której zadaniem jest przywrócenie uśpionych mikroprocesów do ponownego działania. Wykorzystuje ona kolejkę `WAITq`, dzięki czemu ustalenie, czy istnieją mikroprocesy gotowe do przebudzenia mogło zostać zrealizowane w czasie logarytmicznym poprzez analizę wartości skrajnie lewego poddrzewa reprezentacji kolejki, gdzie znajduje się element najmniejszy.

Jeśli jakiekolwiek mikroprocesy zostały przebudzone i tym samym przeniesione do kolejki `RUNq`, następuje faza ich harmonogramowania - `schedule`. Faza ta ustala kolejność uruchamiania i interpretacji kodu poszczególnych mikroprocesów i została szczegółowo opisana w następnej sekcji. Jeśli żaden mikroproces nie oczekuje na uruchomienie SMP wraca do pierwszej fazy oczekując na nowe wiadomości kontrolne.



Rysunek 5.2: Algorytm postępowania symetrycznych multiprocesorów ThesisVM.

Po ustaleniu mikroprocesu gotowego do uruchomienia następuje faza interpretacji jego kodu - `interpret`. Faza ta jest ograniczona czasowo (ang. *time-based scheduling*) a czas jej trwania zależy od obecnego obciążenia SMP. Alternatywnym rozwiązaniem jest ograniczenie maksymalnej ilości kroków interpretera kodu bajtowego (ang. *work-based scheduling*).

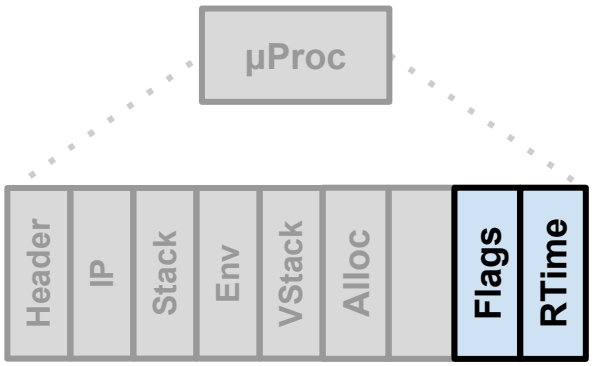
Ostatnia faza, `reschedule` polega na wywłaszczeniu mikroprocesu i przeniesieniu go do jednej z kolejek `RUNq` lub `WAITq` w zależności od efektów jego działania. Po tej fazie następuje koniec cyklu i symetryczny multiprocesor ponownie przechodzi do fazy obsługiwania wiadomości kontrolnych.

5.2. Harmonogramowanie procesów

Najważniejszą fazą opisanego w poprzedniej sekcji algorytmu jest faza harmonogramowania procesów. W celu ustalenia kolejności mikroprocesów wykorzystanie został algorytm **Completely Fair Scheduling** (CFS), który jest stosowany między innymi w jądrze systemu Linux od wersji 2.6.23.

Kluczową cechą algorytmu CFS jest wykorzystanie **wirtualnych czasów** działania zadań, które obliczane są w różny sposób w zależności od priorytetu zadania, co pozwala na wykorzystanie jednej kolejki do harmonogramowania procesów o różnych priorytetach, zamiast wielu osobnych kolejek, dla różnych priorytetów.

Wirtualne czasy działania zadań przechowywane są osobno dla każdego zadania i są modyfikowane po każdym cyklu ich uruchomienia. Rysunek 5.3 zawiera schemat rozmieszczenia dodatkowych rejestrów mikroprocesów ThesisVM koniecznych do zaimplementowania algorytmu CFS.



Rysunek 5.3: Schemat rejestrów wymaganych przez usprawnienia hanmonogramowania SMP.

Rejestr `Flags` przechowuje informację o priorytecie mikroprocesu, a rejestr `RTime` o dotychczasowym, *rzeczywistym* czasie jego działania. Wartość czasu *wirtualnego* wyznaczana jest jako iloczyn priorytetu i rzeczywistego czasu działania.

W każdym cyklu działania SMP mikroproces o najniższej wartości wirtualnego czasu działania pobierany jest z kolejki `RUNq`. Operacja ta, podobnie jak analogiczna operacja dotycząca uśpionych mikroprocesów wykonywana jest w czasie logarytmicznym dzięki wykorzystaniu drzew czerwono czarnych w implementacji kolejki `RUNq`.

Dla tak desygnowanego procesu obliczany jest czas dostępu do procesora (ang. *fair share*), który zależy od konfigurowalnej wartości maksymalnej oraz ilości mikroprocesów aktualnie oczekujących na uruchomienie. Zapewnia to zwiększenie *interaktywności* mikroprocesów kosztem zwiększenia liczby zmian ich kontekstów.

Nowo utworzone mikroprocesy, a także te reaktywowane po czasie uśpienia dodawane są do kolejki `RUNq` z aktualnie minimalną wartością wirtualnego czasu działania. Technika ta nosi miano **sleeper fairness** i gwarantuje, że mikroprocesy, które przez dłuższy czas były w stanie uśpienia otrzymają porównywalny udział czasu procesora sprzętowego.

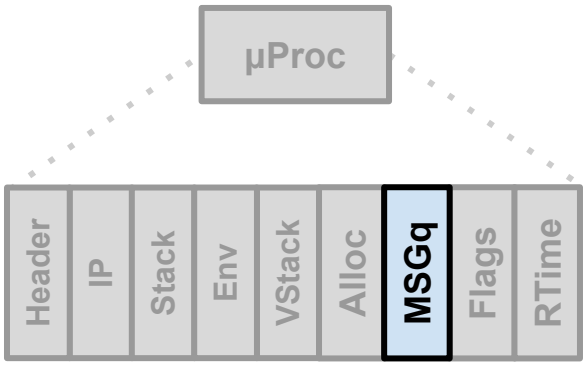
Obecna implementacja harmonogramowania nie wykorzystuje niestety algorytmów równoważenia obciążenia. W przyszłości może zostać jednak rozwinięta umożliwiając podział kolejki `RUNq` i przekazanie części przynależących do niej mikroprocesów do pozostałych symetrycznych multiprocesorów.

5.3. Implementacja Modelu Aktorowego

Implementacja Modelu Aktorowego w maszynie wirtualnej ThesisVM objawia się wykorzystaniem autonomicznych mikroprocesów, które porozumiewają się za pomocą asynchronicznego przekazywania wiadomości.

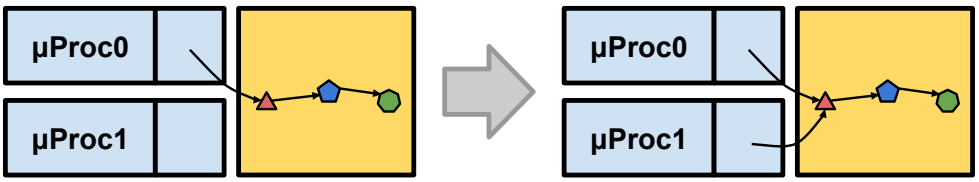
Mikroprocesy tworzone są za pomocą prymitywnej operacji `spawn`, która jako argumenty przyjmuje nazwę funkcji startowej nowego mikroprocesu oraz argument, który zostanie jej przekazany. Tworzony jest wtedy nowy kontekst mikroprocesu, którego rejestry zostają odpowiednio spreparowane by umożliwić natychmiastowe jego wykonanie. Kontekst jest następnie przekazywany do jednego z symetrycznych multiprocesorów, gdzie rozpoczyna swoje działanie.

Działające współbieżnie mikroprocesy mogą komunikować się za pośrednictwem prymitywnych operacji `send` oraz `recv`, które odpowiednio inicjalizują wysyłanie wiadomości i przechwytują następną wiadomość z kolejki wiadomości `MSGq` mikroprocesu. Kolejka ta przechowywana jest w osobnym rejestrze kontekstu mikroprocesu; rysunek 5.4 zawiera schemat położenia rejestru `MSGq` w obiekcie reprezentującym mikroproces.



Rysunek 5.4: Schemat rejestrów wymaganych przez implementację Modelu Aktorowego.

Implementacja zapewnia *logiczną separację* pamięci procesów, ponieważ *nie wspiera mutacji* danych, co gwarantuje, iż jedynym sposobem ich współdzielenia między aktorami jest przekazywanie wiadomości. Należy zwrócić uwagę na fakt, że nie jest to *separacja fizyczna*. Dane przesyłane pomiędzy mikroprocesami egzystują w jednej puli pamięci i nie są w żaden sposób kopiowane (rysunek 5.5).



Rysunek 5.5: Schemat działania przesyłania wiadomości.

Komunikujące się mikroprocesy zwyczajnie przesyłają referencje na interesujące je fragmenty pamięci, co jest operacją o złożoności czasowej *O(1)*.

5.4. Implementacja przesyłania wiadomości

Asynchroniczne przesyłanie wiadomości zostało zrealizowane z wykorzystaniem kolejek nieblokujących (ang. *non-blocking queue*) zaimplementowanych zgodnie z algorytmem **Michael'a i Scott'a** [21]. Algorytm ten wykorzystuje podstawową atomową operację **Compare And Swap** (CAS), której pseudo-implementacja w języku `D` została zaprezentowana na listingu 9.

```
1 bool CAS(T)(shared(T)* address, shared T oldValue, shared T newValue) {
2     if(*address == oldValue) {
3         *address = newValue;
4         return true;
5     }
6     return false;
7 }
```

Listing 9: Fragment kodu obrazujący operację Compare And Swap.

Operacja ta polega na atomowym sprawdzeniu czy pod adresem pamięci `address` w dalszym ciągu znajduje się jej stara wartość `oldValue`, załadowana wcześniej przez

program. Jeśli taka sytuacja ma miejsce pod adresem `address` zostaje zapisana nowa wartość `newValue` i zwrócona zostaje wartość logicznej prawdy pozwalająca określić, czy przypisanie miało miejsce. W przeciwnym wypadku zwrócona zostaje wartość logicznego fałszu.

Z operacją CAS wiąże się zjawisko **ABA**, które polega na podwójnej zmianie wartości pod danym adresem pamięci (z `A` na `B` i ponownie na `A`), co może spowodować, że operacja CAS się powiedzie, mimo że nie powinna. Jest to sytuacja niepożądana i znane są sposoby jej przeciwdziałania, na przykład przez wykorzystanie liczników modyfikacji.

Algorytm Michael'a i Scott'a daje wiele gwarancji, z których najważniejszą jest skończony czas wykonywaniaoperacji dodawania i pobierania wartości z kolejki [21, 22]. Oznacza to, że algorytm ten jest nieblokujący i może być wykorzystany w systemach wymagających gwarancji **soft real-time**.

Istnieje wiele usprawnień algorytmu Michael'a i Scott'a zapewniających dodatkowe gwarancje, takie jak brak oczekiwania (ang. *wait-freedom*) i cechy, jak dynamiczne dostosowywanie rozmiaru. Optymalizacje te zostały opisane między innymi w [23, 24, 25].

Asynchroniczne przekazywanie wiadomości nie jest jedyną metodą komunikacji procesów znaną w literaturze. Alternatywne rozwiązania obejmują między innymi wykorzystanie synchronicznego przesyłania wiadomości poprzez kanały wiadomości (ang. *message channels*) oraz wykorzystanie pamięci współdzielonej i niskopoziomowych prymitywów synchronizacji, takich jak mutex'y, czy semafory.

6. Podsumowanie

– Opisać co udało się zrobić.

– Opisać czego nie udało się zrobić (+ możliwe usprawnienia).

6.1. Interpreter kodu bajtowego

– Opisać brak apdejtowania już obliczonych wartości i dać link do sec:future-development.

– Opisać możliwość zastosowania lepszej reprezentacji bytecodu i bytecode threading.

6.2. Kolektor obiektów nieosiągalnych

– Przeanalizować szybkość, pauzy, zużycie pamięci.

6.3. Przetwarzanie współbieżne

– Przeanalizować szybkość przesyłania wiadomości/konieczność czekania procesów, wielkość kolejek wiadomości.

6.4. Kierunki przyszłego rozwoju

– Opisać plany na przyszły rozwój projektu (priorytet procesów, load balancing SMP, wsparcie dla `Core Erlang`, bytecode threading, przebiegi optymalizacyjne podczas kompilacji, umożliwienie dystrybucji na wiele maszyn, zapasowy kolektor śmieci cyklicznych, opcja wykorzystania sterty prywatnej i autonomicznego alokatora, natywna kompilacja JIT, wektory, data-level parallelism, optymalizacja wykorzystania stosu, hardłerowa implementacja interpretera kodu bajtowego).

Bibliografia

[1] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, (San Francisco, CA, USA), pp. 235–245, Morgan Kaufmann Publishers Inc., 1973.

[2] W. D. Clinger, "Foundations of actor semantics," tech. rep., Cambridge, MA, USA, 1981.

[3] A. Alexandrescu, *The D Programming Language*. Pearson Education, 2010.

[4] J. Armstrong, R. Virding, C. Wikström, and M. Williams, *Concurrent Programming in ERLANG (2Nd Ed.)*. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1996.

[5] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*. Cambridge, MA, USA: MIT Press, 2nd ed., 1996.

[6] S. P. Jones and D. Lester, *Implementing functional languages: a tutorial*. Prentice Hall, 1992. Free online version.

[7] R. Carlsson, "An introduction to Core Erlang," in *In Proceedings of the PLI'01 Erlang Workshop*, 2001.

[8] R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S.-O. Nyström, M. Pettersson, and R. Virding, "Core Erlang 1.0.3 language specification," tech. rep., Department of Information Technology, Uppsala University, Nov. 2004.

[9] G. L. Steele Jr and G. J. Sussman, "The art of the interpreter of the modularity complex (parts zero, one, and two)," 1978.

[10] J. Wilhelmsson, *Efficient Memory Management for Message-Passing Concurrency — part I: Single-threaded execution*. Licentiate thesis, Department of Information Technology, Uppsala University, May 2005.

[11] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," 1995.

[12] R. Shahriyar, S. M. Blackburn, and D. Frampton, "Down for the count? getting reference counting back in the ring," in *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12, (New York, NY, USA), pp. 73–84, ACM, 2012.

[13] D. F. Bacon, P. Cheng, and V. T. Rajan, "A unified theory of garbage collection," in *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, (New York, NY, USA), pp. 50–68, ACM, 2004.

[14] J. Armstrong and R. Virding, "One pass real-time generational mark-sweep garbage collection," in *IN INTERNATIONAL WORKSHOP ON MEMORY MANAGEMENT*, pp. 313–322, Springer-Verlag, 1995.

[15] L. Huelsbergen and P. Winterbottom, "Very concurrent mark-&-sweep garbage collection without fine-grain synchronization," in *Proceedings of the 1st International Symposium on Memory Management*, ISMM '98, (New York, NY, USA), pp. 166–175, ACM, 1998.

[16] J. Fairbairn and S. Wray, "TIM: A simple, lazy abstract machine to execute supercombinators," in *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, (London, UK, UK), pp. 34–45, Springer-Verlag, 1987.

[17] D. Van Horn and M. Might, "Abstracting abstract machines," in *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, (New York, NY, USA), pp. 51–62, ACM, 2010.

[18] J. D. Ramsdell, "The Tail-Recursive SECD Machine," *Journal of Automated Reasoning*, vol. 23, no. 1, pp. 43–62, 1999.

[19] D. Gudeman, "Representing type information in dynamically typed languages," 1993.

[20] H.-J. Boehm, "The space cost of lazy reference counting," in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, (New York, NY, USA), pp. 210–219, ACM, 2004.

[21] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, (New York, NY, USA), pp. 267–275, ACM, 1996.

[22] L. Groves, "Verifying Michael and Scott's lock-free queue algorithm using trace reduction," in *Proceedings of the Fourteenth Symposium on Computing: The Australasian Theory - Volume 77*, CATS '08, (Darlinghurst, Australia, Australia), pp. 133–142, Australian Computer Society, Inc., 2008.

[23] M. Herlihy, V. Luchangco, P. Martin, and M. Moir, "Dynamic-sized lock-free data structures," tech. rep., 2002.

[24] A. Kogan and E. Petrank, "Wait-free queues with multiple enqueuers and dequeuers," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, (New York, NY, USA), pp. 223–234, ACM, 2011.

[25] E. Ladan-Mozes and N. Shavit, "An optimistic approach to lock-free FIFO queues," 2004.

A. Przykładowe programy

Niniejszy dodatek zawiera instrukcje użytkowania maszyny wirtualnej ThesisVM oraz kilka przykładowych programów w języku TVMIR, które można na niej uruchomić.

A.1. Kompilacja maszyny wirtualnej ThesisVM

Do skompilowania projektu wymagana jest jedna z dwóch wspieranych implementacji języka `D`:

– `DMD 2.063.2` - referencyjna implementacja języka.

– `GDC 4.8.2` - implementacja oparta o GNU Compiler Collection.

W celu skompilowania projektu wystarczy uruchomić polecenie `make` w głównym jego katalogu, co zakończy się zbudowaniem wykonywalnego programu `tvm` z wykorzystaniem kompilatora `DMD`. W celu zbudowania projektu za pomocą kompilatora `GDC` należy wykonać polecenie `make -f Makefile.gdc`.

A.2. Interfejs i użytkowanie ThesisVM

Do projektu dołączony został wygodny w użyciu program `tvm`, którego zadaniem jest uruchomienie maszyny wirtualnej i interakcja z użytkownikiem. Uruchamianie programu odbywa się zgodnie z poniższą instrukcją:

```
./tvm [OPCJE] program.tvmir
```

Dodatkowo, możliwe jest wykorzystanie wielu opcji odpowiedzialnych za parametry maszyny wirtualnej i sposób traktowania danych wejściowych:

– `-n --smp-num=NUM` - określa ilość równocześnie działających symetrycznych multiprocesorów (SMP); akceptowane wartości: [1, 256], domyślna wartość: 4.

– `-m --smp-msgq-size=SIZE` - określa rozmiar kolejki wiadomości SMP; akceptowane wartości: [1, 65535], domyślna wartość: 16.

- `--smp-max-preeption-time=TIME` - określa maksymalny czas uruchamiania pojedynczego mikroprocesu w mikrosekundach; akceptowane wartości: [10000, 1000000], domyślna wartość: 100000.
- `--smp-spin-time=TIME` - określa czas aktywnego oczekiwania SMP w mikrosekundach; akceptowane wartości: [1000, 1000000], domyślna wartość: 10000.
- `--smp-sleep-time=TIME` - określa czas "spoczynku" SMP w mikrosekundach, który nie ma aktywnych mikroprocesów; akceptowane wartości: [1000, 1000000000], domyślna wartość: 1000000000.
- `-M --uproc-msgq-size=SIZE` - określa domyślny rozmiar kolejki wiadomości mikroprocesów; akceptowane wartości: [1, 65535], domyślna wartość: 8.
- `-H --uproc-heap-chunk=SIZE` - określa domyślny rozmiar pre-alokowanej pamięci procesu; akceptowane wartości: [0, 262144], domyślna wartość: 0.
- `-P --uproc-default-priority=LEVEL` - określa domyślny priorytet mikroprocesów; akceptowane wartości: [0, 63], domyślna wartość: 32.
- `--debug=OPTION` - przełącza program `tvm` w jeden z poniższych trybów działania, domyślna wartość to `run`:
  - `scan` - przerywa przetwarzanie kodu źródłowego po fazie analizy leksykalnej,
  - `filter` - przerywa przetwarzanie kodu źródłowego po fazie filtracji (ostatnia faza analizy leksykalnej),
  - `parse` - przerywa przetwarzanie kodu źródłowego po fazie analizy syntaktycznej,
  - `transform` - przerywa przetwarzanie kodu źródłowego po fazie analizy semantycznej,
  - `optimize` - przerywa przetwarzanie kodu źródłowego po fazie optymalizacji,
  - `compile` - przerywa przetwarzanie kodu źródłowego po fazie generacji kodu,
  - `interpret` - interpretuje kod źródłowy krok po kroku wykorzystując jeden SMP,
  - `run` - uruchamia kod źródłowy na pełnej maszynie ThesisVM.
- `-v --version` - wypisuje wersję programu `tvm`.
- `-h --help` - wypisuje informacje o użytkowaniu maszyny wirtualnej ThesisVM.

A.3. “Hello world!”

Program wypisuje wiadomość “Hello world!” na ekranie.

```
(define (hello-world str)
  (print str))

(define (main args)
  (hello-world "Hello world!"))
```

A.4. Funkcje i operacje prymitywne

Program prezentuje wykorzystanie kilku funkcji wbudowanych, takich jak `cons`, `+` i `print`.

```
(define (main args)
  (print (cons (+ (- (* 42 23)
                     (/ 13 7))
                  5)
               args)))
```

A.5. Silnia

Program implementuje funkcję obliczającą wartości silni.

```
(define (fact n)
  (if (< n 2)
      n
      (* n (fact (- n 1)))))

(define (main args)
  (print (fact 100)))
```

A.6. Funkcja Fibonacciego

Program implementuje funkcję obliczającą wartości funkcji Fibonacciego.

```
(define (fib n)
  (if (< n 2)
```

```
      1
      (+ (fib (- n 2)) (fib (- n 1)))))

(define (main args)
  (print (fib 20)))
```

A.7. Współbieżne “Hello world!”

Program tworzy nowy proces i wysyła do niego wiadomość “Hello world!”, która następnie jest wypisywana na ekranie.

```
(define (loop timeout)
  (if (print (recv timeout))
      'done
      (loop timeout)))

(define (main args)
  (send (spawn loop 1000) "Hello world!"))
```

A.8. Współbieżne obliczanie funkcji Fibonacciego

Program tworzy trzy nowe procesy, które współbieżnie obliczają wartości funkcji Fibonacciego.

```
(define (fib n)
  (if (< n 2)
      1
      (+ (fib (- n 2)) (fib (- n 1)))))

(define (proc n)
  (print (fib n)))

(define (main args)
  (if (spawn proc 23)
      (if (spawn proc 23)
          (if (spawn proc 23)
              (proc 23)
              'failed)
          'failed)
      'failed))
```

B. Spisy wbudowanych operatorów i funkcji

Spis operatorów wbudowanych

- `+` - pobiera dwa parametry typu numerycznego i zwraca wynik ich dodawania.
- `-` - pobiera dwa parametry typu numerycznego i zwraca wynik ich odejmowania.
- `*` - pobiera dwa parametry typu numerycznego i zwraca wynik ich mnożenia.
- `/` - pobiera dwa parametry typu numerycznego i zwraca wynik ich dzielenia.
- `mod` - pobiera dwa parametry typu numerycznego i zwraca wynik przeprowadzenia na nich operacji modulo.
- `pow` - pobiera dwa parametry typu numerycznego i zwraca wynik przeprowadzenia na nich operacji potęgowania.
- `inc` - pobiera jeden parametr typu numerycznego i zwraca jego wartość inkrementowaną o 1.
- `dec` - pobiera jeden parametr typu numerycznego i zwraca jego wartość dekrementowaną o 1.
- = - pobiera dwa parametry typu numerycznego i zwraca 1 w przypadku, gdy są równe, lub pustą listę w przeciwnym przypadku.
- `<` - pobiera dwa parametry typu numerycznego i zwraca 1 w przypadku, gdy pierwszy z nich ma mniejszą wartość od drugiego, lub pustą listę w przeciwnym przypadku.
- `>` - pobiera dwa parametry typu numerycznego i zwraca 1 w przypadku, gdy pierwszy z nich ma większą wartość od drugiego, lub pustą listę w przeciwnym przypadku.
- `<=` - pobiera dwa parametry typu numerycznego i zwraca 1 w przypadku, gdy pierwszy z nich ma mniejszą bądź równą wartość od drugiego, lub pustą listę w przeciwnym przypadku.

- `>=` - pobiera dwa parametry typu numerycznego i zwraca 1 w przypadku, gdy pierwszy z nich ma większą bądź równą wartość od drugiego, lub pustą listę w przeciwnym przypadku.

- `null?` - pobiera jeden parametr dowolnego typu i zwraca 1 w przypadku, gdy jest on pustą listą, lub pustą listę w przeciwnym przypadku.

- `null` - nie pobiera żadnych parametrów, zwraca pustą listę.

- `cons` - pobiera dwa parametry dowolnych typów i zwraca parę składającą się z obu pobranych wartości.

- `car` - pobiera jeden parametr, który musi być parą i zwraca pierwszy jej element.

- `cdr` - pobiera jeden parametr, który musi być parą i zwraca drugi jej element.

- `typeof` - pobiera jeden parametr dowolnego typu i zwraca wartość liczbową identyfikującą jego typ.

- `sleep` - pobiera jeden parametr typu numerycznego i usypia mikroproces na taką ilość milisekund, zwraca ilość milisekund, na które mikroproces został uśpiony.

- `print` - pobiera jeden parametr dowolnego typu i wyświetla jego tekstową reprezentację, zwraca parametr.

- `self` - nie pobiera parametrów, zwraca deskryptor obecnie działającego mikroprocesu.

- `send` - pobiera dwa parametry, z których pierwszy musi być deskryptorem mikroprocesu, wysyła drugi parametr jako wiadomość do mikroprocesu z pierwszego parametru.

- `recv` - pobiera jeden parametr typu numerycznego, sprawdza, czy w kolejce wiadomości mikroprocesu znajduje się wiadomość i ją zwraca; jeśli w kolejce wiadomości mikroprocesu nie znajdują się żadne wiadomości zwraca pustą listę i usypia mikroproces na czas przekazany w parametrze.

Spis funkcji wbudowanych

– `not` - przyjmuje jeden parametr i zwraca jego logiczną negację (1 w przypadku logicznej prawdy i pustą listę w przypadku logicznego fałszu).

– `and` - przyjmuje dwa parametry i zwraca ich logiczną koniunkcję; funkcja ta mogła zostać zaimplementowana dzięki leniwej naturze maszyny wirtualnej ThesisVM.

– `or` - przyjmuje dwa parametry i zwraca ich logiczną alternatywę; funkcja ta mogła zostać zaimplementowana dzięki leniwej naturze maszyny wirtualnej ThesisVM.

Dodatkowo wszystkie prymitywne operatory wymienione w powyższej liście mają swoje odpowiedniki funkcyjne o takiej samej nazwie i semantyce.

C. Spisy rysunków i fragmentów kodu

Spis rysunków

Spis listingów